# CONTENTS

1 About PsychoPy .......... 1
2 General issues .......... 3
3 Installation .......... 29
4 Getting Started .......... 33
5 Builder .......... 39
6 Coder .......... 77
7 Running and sharing studies online .......... 97
8 Reference Manual (API) .......... 115
9 Timing information for PsychoPy .......... 523
10 Troubleshooting .......... 525
11 Alerts .......... 529
12 Frequently Asked Questions (FAQs) .......... 537
13 Resources (e.g. for teaching) .......... 541
14 For Developers .......... 543
15 PsychoPy Experiment file format (.psyexp) .......... 565
Python Module Index .......... 569
Index .......... 571
CHAPTER
ONE

ABOUT PSYCHOPY

1.1 Citing PsychoPy

If you use this software, please cite one of the publications that describe it. For most people the 2019 paper is probably the most relevant (the papers from 2009, 2007 did not mention Builder at all, for instance).


Citing these papers gives the reviewer/reader of your study information about how the system works and it attributes some credit for its original creation. Academic assessment (whether for promotion or even getting appointed to a job in the first place) prioritises publications over making useful tools for others. Citations provide a way for the developers to justify their continued involvement in the development of the package.

1.2 License for use

PsychoPy is licensed under a GPL3 license which means, essentially, that:

- you can use it (and adapt it) for free in your work, and you can even release those versions
- but you must include the original PsychoPy license
- AND you must also make your release open source using the same license

What that means is you’re free to use PsychoPy’s goodwill in being open source, you are required to pass on that goodwill!
GENERAL ISSUES

These are issues that users should be aware of, whether they are using Builder or Coder views.

2.1 Monitor Center

PsychoPy provides a simple and intuitive way for you to calibrate your monitor and provide other information about it and then import that information into your experiment.

Information is inserted in the Monitor Center (Tools menu), which allows you to store information about multiple monitors and keep track of multiple calibrations for the same monitor.

For experiments written in the Builder view, you can then import this information by simply specifying the name of the monitor that you wish to use in the Experiment settings dialog. For experiments created as scripts you can retrieve the information when creating the Window by simply naming the monitor that you created in Monitor Center. e.g.:

```python
from psychopy import visual
win = visual.Window([1024,768], mon='SonyG500')
```

Of course, the name of the monitor in the script needs to match perfectly the name given in the Monitor Center.

2.1.1 Real world units

One of the particular features of PsychoPy is that you can specify the size and location of stimuli in units that are independent of your particular setup, such as degrees of visual angle (see Units for the window and stimuli). In order for this to be possible you need to inform PsychoPy of some characteristics of your monitor. Your choice of units determines the information you need to provide:

<table>
<thead>
<tr>
<th>Units</th>
<th>Requires</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘norm’ (normalised to width/height)</td>
<td>n/a</td>
</tr>
<tr>
<td>‘pix’ (pixels)</td>
<td>Screen width in pixels</td>
</tr>
<tr>
<td>‘cm’ (centimeters on the screen)</td>
<td>Screen width in pixels and screen width in cm</td>
</tr>
<tr>
<td>‘deg’ (degrees of visual angle)</td>
<td>Screen width (pixels), screen width (cm) and distance (cm)</td>
</tr>
</tbody>
</table>
2.1.2 Calibrating your monitor

PsychoPy can also store and use information about the gamma correction required for your monitor. If you have a Spectrascan PR650 (other devices will hopefully be added) you can perform an automated calibration in which PsychoPy will measure the necessary gamma value to be applied to your monitor. Alternatively this can be added manually into the grid to the right of the Monitor Center. To run a calibration, connect the PR650 via the serial port and, immediately after turning it on press the Find PR650 button in the Monitor Center.

Note that, if you don't have a photometer to hand then there is a method for determining the necessary gamma value psychophysically included in PsychoPy (see gammaMotionNull and gammaMotionAnalysis in the demos menu).

The two additional tables in the Calibration box of the Monitor Center provide conversion from DKL and LMS colour spaces to RGB.

2.2 Units for the window and stimuli

One of the key advantages of PsychoPy over many other experiment-building software packages is that stimuli can be described in a wide variety of real-world, device-independent units. In most other systems you provide the stimuli at a fixed size and location in pixels, or percentage of the screen, and then have to calculate how many cm or degrees of visual angle that was.

In PsychoPy, after providing information about your monitor, via the Monitor Center, you can simply specify your stimulus in the unit of your choice and allow PsychoPy to calculate the appropriate pixel size for you.

Your choice of unit depends on the circumstances. For conducting demos, the two normalised units ('norm' and 'height') are often handy because the stimulus scales naturally with the window size. For running an experiment it's usually best to use something like ‘cm’ or ‘deg’ so that the stimulus is a fixed size irrespective of the monitor/window.

For all units, the centre of the screen is represented by coordinates (0,0), negative values mean down/left, positive values mean up/right.

2.2.1 Height units

With ‘height’ units everything is specified relative to the height of the window (note the window, not the screen). As a result, the dimensions of a screen with standard 4:3 aspect ratio will range (-0.6667,-0.5) in the bottom left to (+0.6667,+0.5) in the top right. For a standard widescreen (16:10 aspect ratio) the bottom left of the screen is (-0.8,-0.5) and top-right is (+0.8,+0.5). This type of unit can be useful in that it scales with window size, unlike Degrees of visual angle or Centimeters on screen, but stimuli remain square, unlike Normalised units units. Obviously it has the disadvantage that the location of the right and left edges of the screen have to be determined from a knowledge of the screen dimensions. (These can be determined at any point by the Window.size attribute.)

Spatial frequency: cycles per stimulus (so will scale with the size of the stimulus).

Requires : No monitor information
2.2.2 Normalised units

In normalised ('norm') units the window ranges in both x and y from -1 to +1. That is, the top right of the window has coordinates (1,1), the bottom left is (-1,-1). Note that, in this scheme, setting the height of the stimulus to be 1.0, will make it half the height of the window, not the full height (because the window has a total height of 1:-1 = 2!). Also note that specifying the width and height to be equal will not result in a square stimulus if your window is not square - the image will have the same aspect ratio as your window. e.g. on a 1024x768 window the size=(0.75,1) will be square.

Spatial frequency: cycles per stimulus (so will scale with the size of the stimulus).

Requires : No monitor information

2.2.3 Centimeters on screen

Set the size and location of the stimulus in centimeters on the screen.

Spatial frequency: cycles per cm

Requires : information about the screen width in cm and size in pixels

Assumes : pixels are square. Can be verified by drawing a stimulus with matching width and height and verifying that it is in fact square. For a CRT this can be controlled by setting the size of the viewable screen (settings on the monitor itself).

2.2.4 Degrees of visual angle

Use degrees of visual angle to set the size and location of the stimulus. This is, of course, dependent on the distance that the participant sits from the screen as well as the screen itself, so make sure that this is controlled, and remember to change the setting in Monitor Center if the viewing distance changes.

Spatial frequency: cycles per degree

Requires : information about the screen width in cm and pixels and the viewing distance in cm

There are actually three variants: 'deg', 'degFlat', and 'degFlatPos'

'deg' : Most people using degrees of visual angle choose to make the assumption that a degree of visual angle spans the same number of pixels at all parts of the screen. This isn't actually true for standard flat screens - a degree of visual angle at the edge of the screen spans more pixels because it is further from the eye. For moderate eccentricities the error is small (a 0.2% error in size calculation at 3 deg eccentricity) but grows as stimuli are placed further from the centre of the screen (a 2% error at 10 deg). For most studies this form of calculation is preferred, as it does not result in a warped appearance of visual stimuli, but if you need greater precision at far eccentricities then choose one of the alternatives below.

'degFlatPos' : This accounts for flat screens in calculating position coordinates of visual stimuli but leaves size and spatial frequency uncorrected. This means that an evenly spaced grid of visual stimuli will appear warped in position but will

'degFlat' : This corrects the calculations of degrees for flatness of the screen for each vertex of your stimuli. Square stimuli in the periphery will, therefore, become more spaced apart but they will also get larger and rhomboid in the pixels that they occupy.
2.2.5 Pixels on screen

You can also specify the size and location of your stimulus in pixels. Obviously this has the disadvantage that sizes are
specific to your monitor (because all monitors differ in pixel size).

Spatial frequency: `cycles per pixel` (this catches people out but is used to be in keeping with the other units.
If using pixels as your units you probably want a spatial frequency in the range 0.2-0.001 (i.e. from 1 cycle every 5 pixels
to one every 100 pixels).

Requires: information about the size of the screen (not window) in pixels, although this can often be deduce from the
operating system if it has been set correctly there.

Assumes: nothing

2.3 Color spaces

The color of stimuli can be specified when creating a stimulus and when using setColor() in a variety of ways. There are
three basic color spaces that PsychoPy can use, RGB, DKL and LMS but colors can also be specified by a name (e.g.
‘DarkSalmon’) or by a hexadecimal string (e.g. ‘#00FF00’).

examples:

```
stim = visual.GratingStim(win, color=[1,-1,-1], colorSpace='rgb')  #will be red
stim.setColor('Firebrick')#one of the web/X11 color names
stim.setColor('#FFFAF0')#an off-white
stim.setColor([0,90,1], colorSpace='dkl')#modulate along S-cone axis in isoluminant plane
stim.setColor([1,0,0], colorSpace='lms')#modulate only on the L cone
stim.setColor([1,1,1], colorSpace='rgb')#all guns to max
stim.setColor([1,0,0])#this is ambiguous - you need to specify a color space
```

2.3.1 Colors by name

Any of the web/X11 color names can be used to specify a color. These are then converted into RGB space by PsychoPy.
These are not case sensitive, but should not include any spaces.

2.3.2 Colors by hex value

This is really just another way of specifying the r,g,b values of a color, where each gun’s value is given by two hexadecimal
characters. For some examples see this chart. To use these in PsychoPy they should be formatted as a string, beginning
with # and with no spaces. (NB on a British Mac keyboard the # key is hidden - you need to press Alt-3)
2.3.3 RGB color space

This is the simplest color space, in which colors are represented by a triplet of values that specify the red green and blue intensities. These three values each range between -1 and 1.

Examples:
- \([1,1,1]\) is white
- \([0,0,0]\) is grey
- \([-1,-1,-1]\) is black
- \([1.0,-1,-1]\) is red
- \([1.0,0.6,0.6]\) is pink

The reason that these colors are expressed ranging between 1 and -1 (rather than 0:1 or 0:255) is that many experiments, particularly in visual science where PsychoPy has its roots, express colors as deviations from a grey screen. Under that scheme a value of -1 is the maximum decrement from grey and +1 is the maximum increment above grey.

Note that PsychoPy will use your monitor calibration to linearize this for each gun. E.g., 0 will be halfway between the minimum luminance and maximum luminance for each gun, if your monitor gammaGrid is set correctly.

2.3.4 HSV color space

Another way to specify colors is in terms of their Hue, Saturation and ‘Value’ (HSV). For a description of the color space see the Wikipedia HSV entry. The Hue in this case is specified in degrees, the saturation ranging 0:1 and the ‘value’ also ranging 0:1.

Examples:
- \([0,1,1]\) is red
- \([0,0.5,1]\) is pink
- \([90,1,1]\) is cyan
- \([anything, 0, 1]\) is white
- \([anything, 0, 0.5]\) is grey
- \([anything, anything,0]\) is black

Note that colors specified in this space (like in RGB space) are not going to be the same another monitor; they are devicespecific. They simply specify the intensity of the 3 primaries of your monitor, but these differ between monitors. As with the RGB space gamma correction is automatically applied if available.

2.3.5 DKL color space

To use DKL color space the monitor should be calibrated with an appropriate spectrophotometer, such as a PR650.

In the Derrington, Krauskopf and Lennie color space (based on the Macleod and Boynton chromaticity diagram) colors are represented in a 3-dimensional space using spherical coordinates that specify the elevation from the isoluminant plane, the azimuth (the hue) and the contrast (as a fraction of the maximal modulations along the cardinal axes of the space).


In PsychoPy these values are specified in units of degrees for elevation and azimuth and as a float (ranging -1:1) for the contrast.

Note that not all colors that can be specified in DKL color space can be reproduced on a monitor. Here is a movie plotting in DKL space (showing cartesian coordinates, not spherical coordinates) the gamut of colors available on an example CRT monitor.

Examples:
- \([90,0,1]\) is white (maximum elevation aligns the color with the luminance axis)
- \([0,0,1]\) is an isoluminant stimulus, with azimuth 0 (S-axis)
- \([0,45,1]\) is an isoluminant stimulus, with an oblique azimuth

### 2.3.6 LMS color space

To use LMS color space the monitor should be calibrated with an appropriate spectrophotometer, such as a PR650.

In this color space you can specify the relative strength of stimulation desired for each cone independently, each with a value from -1:1. This is particularly useful for experiments that need to generate cone isolating stimuli (for which modulation is only affecting a single cone type).
2.4 Preferences

The Preferences dialog allows to adjust general settings for different parts of PsychoPy. The preferences settings are saved in the configuration file `userPrefs.cfg`. The labels in brackets for the different options below represent the abbreviations used in the `userPrefs.cfg` file.

In rare cases, you might want to adjust the preferences on a per-experiment basis. See the API reference for the Preferences class here.

2.4.1 General settings (General)

- **window type (winType):** PsychoPy can use one of two ‘backends’ for creating windows and drawing: pygame, pyglet and glfw. Here you can set the default backend to be used.
- **units (units):** Default units for windows and visual stimuli (‘deg’, ‘norm’, ‘cm’, ‘pix’). See Units for the window and stimuli. Can be overridden by individual experiments.
- **full-screen (fullscr):** Should windows be created full screen by default? Can be overridden by individual experiments.
- **allow GUI (allowGUI):** When the window is created, should the frame of the window and the mouse pointer be visible. If set to False then both will be hidden.
- **paths (paths):** Paths for additional Python packages can be specified. See more information here.
- **flac audio compression (flac):** Set flac audio compression.
- **parallel ports (parallelPorts):** This list determines the addresses available in the drop-down menu for the Parallel Port Out Component.

2.4.2 Application settings (App)

These settings are common to all components of the application (Coder and Builder etc)

- **show start-up tips (showStartupTips):** Display tips when starting PsychoPy.
- **large icons (largeIcons):** Do you want large icons (on some versions of wx on macOS this has no effect)?
- **default view (defaultView):** Determines which view(s) open when the PsychoPy app starts up. Default is ‘last’, which fetches the same views as were open when PsychoPy last closed.
- **reset preferences (resetPrefs):** Reset preferences to defaults on next restart of PsychoPy.
- **auto-save prefs (autoSavePrefs):** Save any unsaved preferences before closing the window.
- **debug mode (debugMode):** Enable features for debugging PsychoPy itself, including unit-tests.
- **locale (locale):** Language to use in menus etc.; not all translations are available. Select a value, then restart the app.
  Think about adding translations for your language.
2.4.3 Builder settings (Builder)

reload previous exp (reloadPrevExp): Select whether to automatically reload a previously opened experiment at start-up.

uncluttered namespace (unclutteredNamespace): If this option is selected, the scripts will use more complex code, but the advantage is that there is less of a chance that name conflicts will arise.

components folders (componentsFolders): A list of folder path names that can hold additional custom components for the Builder view; expects a comma-separated list.

hidden components (hiddenComponents): A list of components to hide (e.g., because you never use them)

unpacked demos dir (unpackedDemosDir): Location of Builder demos on this computer (after unpacking).

saved data folder (savedDataFolder): Name of the folder where subject data should be saved (relative to the script location).

Flow at top (topFlow): If selected, the “Flow” section will be shown topmost and the “Components” section will be on the left. Restart PsychoPy to activate this option.

always show readme (alwaysShowReadme): If selected, PsychoPy always shows the Readme file if you open an experiment. The Readme file needs to be located in the same folder as the experiment file.

max favorites (maxFavorites): Upper limit on how many components can be in the Favorites menu of the Components panel.

2.4.4 Coder settings (Coder)

code font (codeFont): A list of font names to be used for code display. The first found on the system will be used.

comment font (commentFont): A list of font names to be used for comments sections. The first found on the system will be used.

output font (outputFont): A list of font names to be used in the output panel. The first found on the system will be used.

code font size (codeFontSize): An integer between 6 and 24 that specifies the font size for code display in points.

output font size (outputFontSize): An integer between 6 and 24 that specifies the font size for output display in points.

show source asst (showSourceAsst): Do you want to show the source assistant panel (to the right of the Coder view)? On Windows this provides help about the current function if it can be found. On macOS the source assistant is of limited use and is disabled by default.

show output (showOutput): Show the output panel in the Coder view. If shown all python output from the session will be output to this panel. Otherwise it will be directed to the original location (typically the terminal window that called PsychoPy application to open).

reload previous files (reloadPrevFiles): Should PsychoPy fetch the files that you previously had open when it launches?

preferred shell (preferredShell): Specify which shell should be used for the coder shell window.

newline convention (newlineConvention): Specify which character sequence should be used to encode newlines in code files: unix = n (line feed only), dos = nn (carriage return plus line feed).
2.4.5 Connection settings (Connections)

proxy (proxy): The proxy server used to connect to the internet if needed. Must be of the form http://111.222.333.444:5555

auto-proxy (autoProxy): PsychoPy should try to deduce the proxy automatically. If this is True and autoProxy is successful, then the above field should contain a valid proxy address.

allow usage stats (allowUsageStats): Allow PsychoPy to ping a website at when the application starts up. Please leave this set to True. The info sent is simply a string that gives the date, PsychoPy version and platform info. There is no cost to you: no data is sent that could identify you and PsychoPy will not be delayed in starting as a result. The aim is simple: if we can show that lots of people are using PsychoPy there is a greater chance of it being improved faster in the future.

check for updates (checkForUpdates): PsychoPy can (hopefully) automatically fetch and install updates. This will only work for minor updates and is still in a very experimental state (as of v1.51.00).

timeout (timeout): Maximum time in seconds to wait for a connection response.

2.4.6 Hardware settings

audioLib : Select your choice of audio library with a list of names specifying the order they should be tried. We recommend ['PTB', 'sounddevice', 'pyo', 'pygame'] for lowest latency.

audioLatencyMode [0, 1, 2, 3, 4] Latency mode for PsychToolbox audio (3 is good for most applications. See PTB_latency_mode

audioDriver: 'portaudio' Some of PsychoPy's audio engines provide the option not to sue portaudio but go directly to another lib (e.g. to coreaudio) but some don’t allow that

    # audio driver to use audioDriver = list(default=list('portaudio')) # audio device to use (if audioLib allows control) audioDevice = list(default=list('default')) # a list of parallel ports parallelPorts = list(default=list('0x0378', '0x03BC')) # The name of the Qmix pump configuration to use qmixConfiguration = string(default='qmix_config')

2.4.7 Key bindings

There are many shortcut keys that you can use in PsychoPy. For instance did you realise that you can indent or outdent a block of code with Ctrl-[ and Ctrl-]?

2.5 Data outputs

There are a number of different forms of output that PsychoPy can generate, depending on the study and your preferred analysis software. Multiple file types can be output from a single experiment (e.g. Excel data file for a quick browse, Log file to check for error messages and PsychoPy data file (.psydat) for detailed analysis)
### 2.5.1 Log file

Log files are actually rather difficult to use for data analysis but provide a chronological record of everything that happened during your study. The level of content in them depends on you. See *Logging data* for further information.

### 2.5.2 PsychoPy data file (.psydat)

This is actually a *TrialHandler* or *StairHandler* object that has been saved to disk with the python cPickle module.

.psydat files can be useful for retrieving data that you forgot to explicitly tell PsychoPy to save. They can also be more directly used by experienced users with previous experience of python and, probably, matplotlib. The contents of the file can be explored with dir(), as any other python object.

.psydat files are ideal for batch analysis with a python script and plotting via *matplotlib*. They contain more information than the Excel or csv data files, and can even be used to (re)create those files.

**Of particular interest might be the following attributes and methods of the Handler:**

- **entries** a list of dictionaries. Each entry/dictionary in the list represents a single trial’s (a single routine ‘run/iteration) data.
- **saveAsPickle()** a method for saving all of the entries’ data in a Python pickle file
- **saveAsWideText()** a method for saving all of the entrée’s data in a .csv file.

If you just want to recover data or first wish to try things out in a familiar format you can put all of the data in a .csv file, very similar to the .csv files that are produced by default when running PsychoPy experiments. The following script assumes you’re using a command-line interface (e. g. Terminal on Mac, or the Command Prompt on Windows) where you’ve opened up a Python shell, and that you have installed PsychoPy as a Python package.

```python
# import PsychoPy function for loading Pickle/JSON data from psychopy.misc
import fromFile  # (replace with the file path to your .psydat file)
filepath = '/Users/my_user/myexperiments/myexperiment/participant_expname_date.psydat'  # load in the data
psydata = fromFile(filepath)  # (replace with the file path to where you want the resulting .csv # to be saved)
save_path = '/Users/my_user/Desktop/test_out.csv'  # save the data as a .csv file, ie separating the values with a # comma. ‘CSV’ simply means ‘comma-separated values’
psydata.saveAsWideText(save_path, delim=',')
```

To get started using the data directly in Python, you can try opening a psydat file and printing all its entries:

```python
from psychopy.misc import fromFile  # (replace with the file path to your .psydat file)
filepath = 'path/to/file.psydat'
psydata = fromFile(filepath)
for entry in psydata.entries:
    print(entry)
```

Ideally, we should provide a demo script here for fetching and plotting some data (feel free to *contribute*).
2.5.3 Long-wide data file

This form of data file is the default data output from Builder experiments as of v1.74.00. Rather than summarising data in a spreadsheet where one row represents all the data from a single condition (as in the summarised data format), in long-wide data files the data is not collapsed by condition, but written chronologically with one row representing one trial (hence it is typically longer than summarised data files). One column in this format is used for every single piece of information available in the experiment, even where that information might be considered redundant (hence the format is also 'wide').

Although these data files might not be quite as easy to read quickly by the experimenter, they are ideal for import and analysis under packages such as R, SPSS or Matlab.

2.5.4 Excel data file

Excel 2007 files (.xlsx) are a useful and flexible way to output data as a spreadsheet. The file format is open and supported by nearly all spreadsheet applications (including older versions of Excel and also OpenOffice). N.B. because .xlsx files are widely supported, the older Excel file format (.xls) is not likely to be supported by PsychoPy unless a user contributes the code to the project.

Data from PsychoPy are output as a table, with a header row. Each row represents one condition (trial type) as given to the TrialHandler. Each column represents a different type of data as given in the header. For some data, where there are multiple columns for a single entry in the header. This indicates multiple trials. For example, with a standard data file in which response time has been collected as 'rt' there will be a heading rt_raw with several columns, one for each trial that occurred for the various trial types, and also an rt_mean heading with just a single column giving the mean reaction time for each condition.

If you're creating experiments by writing scripts then you can specify the sheet name as well as file name for Excel file outputs. This way you can store multiple sessions for a single subject (use the subject as the filename and a date-stamp as the sheet name) or a single file for multiple subjects (give the experiment name as the filename and the participant as the sheet name).

Builder experiments use the participant name as the file name and then create a sheet in the Excel file for each loop of the experiment. e.g. you could have a set of practice trials in a loop, followed by a set of main trials, and these would each receive their own sheet in the data file.

2.5.5 Delimited text files (.csv, .tsv, .txt)

For maximum compatibility, especially for legacy analysis software, you can choose to output your data as a delimited text file. Typically this would be comma-separated values (.csv file) or tab-delimited (.tsv file). The format of those files is exactly the same as the Excel file, but is limited by the file format to a single sheet.

2.6 Gamma correcting a monitor

Monitors typically don't have linear outputs; when you request luminance level of 127, it is not exactly half the luminance of value 254. For experiments that require the luminance values to be linear, a correction needs to be put in place for this nonlinearity which typically involves fitting a power law or gamma ($\gamma$) function to the monitor output values. This process is often referred to as gamma correction.

PsychoPy can help you perform gamma correction on your monitor, especially if you have one of the supported photometers/spectroradiometers.

There are various different equations with which to perform gamma correction. The simple equation (2.1) is assumed by most hardware manufacturers and gives a reasonable first approximation to a linear correction. The full gamma correction
equation (2.3) is more general, and likely more accurate especially where the lowest luminance value of the monitor is bright, but also requires more information. It can only be used in labs that do have access to a photometer or similar device.

2.6.1 Simple gamma correction

The simple form of correction (as used by most hardware and software) is this:

$$L(V) = a + kV^\gamma$$

(2.1)

where $L$ is the final luminance value, $V$ is the requested intensity (ranging 0 to 1), $a$, $k$ and $\gamma$ are constants for the monitor. This equation assumes that the luminance where the monitor is set to ‘black’ ($V=0$) comes entirely from the surround and is therefore not subject to the same nonlinearity as the monitor. If the monitor itself contributes significantly to $a$ then the function may not fit very well and the correction will be poor.

The advantage of this function is that the calibrating system (PsychoPy in this case) does not need to know anything more about the monitor than the gamma value itself (for each gun). For the full gamma equation (2.3), the system needs to know about several additional variables. The look-up table (LUT) values required to give a (roughly) linear luminance output can be generated by:

$$LUT(V) = V^{1/\gamma}$$

(2.2)

where $V$ is the entry in the LUT, between 0 (black) and 1 (white).

2.6.2 Full gamma correction

For very accurate gamma correction PsychoPy uses a more general form of the equation above, which can separate the contribution of the monitor and the background to the lowest luminance level:

$$L(V) = a + (b + kV)^\gamma$$

(2.3)

This equation makes no assumption about the origin of the base luminance value, but requires that the system knows the values of $b$ and $k$ as well as $\gamma$.

The inverse values, required to build the LUT are found by:

$$LUT(V) = \frac{((1 - V)b^\gamma + V(b + k)^\gamma)^{1/\gamma} - b}{k}$$

(2.4)

This is derived below, for the interested reader. ;-)

And the associated luminance values for each point in the LUT are given by:

$$L(V) = a + (1 - V)b^\gamma + V(b + k)^\gamma$$

2.6.3 Deriving the inverse full equation

The difficulty with the full gamma equation (2.3) is that the presence of the $b$ value complicates the issue of calculating the inverse values for the LUT. The simple inverse of (2.3) as a function of output luminance values is:

$$LUT(L) = \frac{(L - a)^{1/\gamma} - b}{k}$$

(2.5)

To use this equation we need to first calculate the linear set of luminance values, $L$, that we are able to produce the current monitor and lighting conditions and then deduce the LUT value needed to generate that luminance value.
We need to insert into the LUT the values between 0 and 1 (to use the maximum range) that map onto the linear range from the minimum, \( m \), to the maximum \( M \) possible luminance. From the parameters in (2.3) it is clear that:

\[
\begin{align*}
m &= a + b \gamma \\
M &= a + (b + k) \gamma
\end{align*}
\]  
(2.6)

Thus, the luminance value, \( L \) at any given point in the LUT, \( V \), is given by

\[
L(V) = m + (M - m)V \\
= a + b \gamma + (a + (b + k) \gamma - a - b \gamma)V \\
= a + b \gamma + ((b + k) \gamma - b \gamma)V \\
= a + (1 - V)b \gamma + V(b + k) \gamma
\]  
(2.7)

where \( V \) is the position in the LUT as a fraction.

Now, to generate the LUT as needed we simply take the inverse of (2.3):

\[
LUT(L) = \frac{(L - a)\gamma - b}{k}
\]  
(2.8)

and substitute our \( L(V) \) values from (2.7):

\[
LUT(V) = \frac{(a + (1 - V)b \gamma + V(b + k) \gamma - a)\gamma - b}{k} \\
= \frac{((1 - V)b \gamma + V(b + k) \gamma)\gamma - b}{k}
\]  
(2.9)

### 2.6.4 References

### 2.7 OpenGL and Rendering

All rendering performed by PsychoPy uses hardware-accelerated OpenGL rendering where possible. This means that, as much as possible, the necessary processing to calculate pixel values is performed by the graphics card GPU rather than by the CPU. For example, when an image is rotated the calculations to determine what pixel values should result, and any interpolation that is needed, are determined by the graphics card automatically.

In the double-buffered system, stimuli are initially drawn into a piece of memory on the graphics card called the ‘back buffer’, while the screen presents the ‘front buffer’. The back buffer initially starts blank (all pixels are set to the window’s defined color) and as stimuli are ‘rendered’ they are gradually added to this back buffer. The way in which stimuli are combined according to transparency rules is determined by the blend mode of the window. At some point in time, when we have rendered to this buffer all the objects that we wish to be presented, the buffers are ‘flipped’ such that the stimuli we have been drawing are presented simultaneously. The monitor updates at a very precise fixed rate and the flipping of the window will be synchronised to this monitor update if possible (see Sync to VBL and wait for VBL).

Each update of the window is referred to as a ‘frame’ and this ultimately determines the temporal resolution with which stimuli can be presented (you cannot present your stimulus for any duration other than a multiple of the frame duration). In addition to synchronising flips to the frame refresh rate, PsychoPy can optionally go a further step of not allowing the code to continue until a screen flip has occurred on the screen, which is useful in ascertaining exactly when the frame refresh occurred (and, thus, when your stimulus actually appeared to the subject). These timestamps are very precise on most computers. For further information about synchronising and waiting for the refresh see Sync to VBL and wait for VBL.

If the code/processing required to render all you stimuli to the screen takes longer to complete than one screen refresh then you will ‘drop/skip a frame’. In this case the previous frame will be left on screen for a further frame period and the flip will only take effect on the following screen update. As a result, time-consuming operations such as disk accesses
or execution of many lines of code, should be avoided while stimuli are being dynamically updated (if you care about the precise timing of your stimuli). For further information see the sections on Detecting dropped frames and Reducing dropped frames.

### 2.7.1 Fast and slow functions

The fact that modern graphics processors are extremely powerful; they can carry out a great deal of processing from a very small number of commands. Consider, for instance, the PsychoPy Coder demo `elementArrayStim` in which several hundred Gabor patches are updated frame by frame. The graphics card has to blend a sinusoidal grating with a grey background, using a Gaussian profile, several hundred times each at a different orientation and location and it does this in less than one screen refresh on a good graphics card.

There are three things that are relatively slow and should be avoided at critical points in time (e.g. when rendering a dynamic or brief stimulus). These are:

1. disk accesses
2. passing large amounts of data to the graphics card
3. making large numbers of python calls.

Functions that are very fast:

1. Calls that move, resize, rotate your stimuli are likely to carry almost no overhead
2. Calls that alter the color, contrast or opacity of your stimulus will also have no overhead IF your graphics card supports OpenGL Shaders
3. Updating of stimulus parameters for psychopy.visual.ElementArrayStim is also surprisingly fast BUT you should try to update your stimuli using `numpy` arrays for the maths rather than `for`... loops

Notable slow functions in PsychoPy calls:

1. Calls to set the image or set the mask of a stimulus. This involves having to transfer large amounts of data between the computer’s main processor and the graphics card, which is a relatively time-consuming process.
2. Any of your own code that uses a Python `for`... loop is likely to be slow if you have a large number of cycles through the loop. Try to ‘vectorise’ your code using a numpy array instead.

### 2.7.2 Tips to render stimuli faster

1. Keep images as small as possible. This is meant in terms of number of pixels, not in terms of Mb on your disk. Reducing the size of the image on your disk might have been achieved by image compression such as using jpeg images but these introduce artefacts and do nothing to reduce the problem of send large amounts of data from the CPU to the graphics card. Keep in mind the size that the image will appear on your monitor and how many pixels it will occupy there. If you took your photo using a 10 megapixel camera that means the image is represented by 30 million numbers (a red, green and blue) but your computer monitor will have, at most, around 2 megapixels (1960x1080).
2. Try to use square powers of two for your image sizes. This is efficient because computer memory is organised according to powers of two (did you notice how often numbers like 128, 512, 1024 seem to come up when you buy your computer?). Also several mathematical routines (anything involving Fourier maths, which is used a lot in graphics processing) are faster with power-of-two sequences. For the `psychopy.visual.GratingStim` a texture/mask of this size is required and if you don’t provide one then your texture will be ‘upsampled’ to the next larger square-power-of-2, so you can save this interpolation step by providing it in the right shape initially.
3. Get a faster graphics card. Upgrading to a more recent card will cost around £30. If you're currently using an integrated Intel graphics chip then almost any graphics card will be an advantage. Try to get an nVidia or an ATI Radeon card.
2.7.3 OpenGL Shaders

You may have heard mention of ‘shaders’ on the users mailing list and wondered what that meant (or maybe you didn’t wonder at all and just went for a donut!). OpenGL shader programs allow modern graphics cards to make changes to things during the rendering process (i.e. while the image is being drawn). To use this you need a graphics card that supports OpenGL 2.1 and PsychoPy will only make use of shaders if a specific OpenGL extension that allows floating point textures is also supported. Nowadays nearly all graphics cards support these features - even Intel chips from Intel!

One example of how such shaders are used is the way that PsychoPy colors greyscale images. If you provide a greyscale image as a 128x128 pixel texture and set its color to be red then, without shaders, PsychoPy needs to create a texture that contains the 3x128x128 values where each of the 3 planes is scaled according to the RGB values you require. If you change the color of the stimulus a new texture has to be generated with the new weightings for the 3 planes. However, with a shader program, that final step of scaling the texture value according to the appropriate RGB value can be done by the graphics card. That means we can upload just the 128x128 texture (taking 1/3 as much time to upload to the graphics card) and then each time we change the color of the stimulus we just need to upload a new RGB triplet (only 3 numbers) without having to recalculate the texture. As a result, on graphics cards that support shaders, changing colors, contrasts and opacities etc. has almost zero overhead.

2.7.4 Blend Mode

A ‘blend function’ determines how the values of new pixels being drawn should be combined with existing pixels in the ‘frame buffer’.

\[ \text{blendMode} = \text{‘avg’} \]

This mode is exactly akin to the real-world scenario of objects with varying degrees of transparency being placed in front of each other; increasingly transparent objects allow increasing amounts of the underlying stimuli to show through. Opaque stimuli will simply occlude previously drawn objects. With each increasing semi-transparent object to be added, the visibility of the first object becomes increasingly weak. The order in which stimuli are rendered is very important since it determines the ordering of the layers. Mathematically, each pixel colour is constructed from \( \text{opacity} \times \text{stimRGB} + (1-\text{opacity}) \times \text{backgroundRGB} \). This was the only mode available before PsychoPy version 1.80 and remains the default for the sake of backwards compatibility.

\[ \text{blendMode} = \text{‘add’} \]

If the window \text{blendMode} is set to ‘add’ then the value of the new stimulus does not in any way replace that of the existing stimuli that have been drawn; it is added to it. In this case the value of \text{opacity} still affects the weighting of the new stimulus being drawn but the first stimulus to be drawn is never ‘occluded’ as such. The sum is performed using the signed values of the color representation in PsychoPy, with the mean grey being represented by zero. So a dark patch added to a dark background will get even darker. For grating stimuli this means that contrast is summed correctly.

This blend mode is ideal if you want to test, for example, the way that subjects perceive the sum of two potentially overlapping stimuli. It is also needed for rendering stereo/dichoptic stimuli to be viewed through colored anaglyph glasses.

If stimuli are combined in such a way that an impossible luminance value is requested of any of the monitor guns then that pixel will be out of bounds. In this case the pixel can either be clipped to provide the nearest possible colour, or can be artificially colored with noise, highlighting the problem if the user would prefer to know that this has happened.
2.7.5 Sync to VBL and wait for VBL

PsychoPy will always, if the graphics card allows it, synchronise the flipping of the window with the vertical blank interval (VBL aka VBI) of the screen. This prevents visual artefacts such as ‘tearing’ of moving stimuli. This does not, itself, indicate that the script also waits for the physical frame flip to occur before continuing. If the waitBlanking window argument is set to False then, although the window refreshes themselves will only occur in sync with the screen VBL, the win.flip() call will not actually wait for this to occur, such that preparations can continue immediately for the next frame. For rendering purposes this is actually optimal and will reduce the likelihood of frames being dropped during rendering.

By default the PsychoPy Window will also wait for the VBL (waitBlanking=True). Although this is slightly less efficient for rendering purposes it is necessary if we need to know exactly when a frame flip occurred (e.g. to timestamp when the stimulus was physically presented). On most systems this will provide a very accurate measure of when the stimulus was presented (with a variance typically well below 1ms but this should be tested on your system).

2.8 Timing Issues and synchronisation

One of the key requirements of experimental control software is that it has good temporal precision. PsychoPy aims to be as precise as possible in this domain and can achieve excellent results depending on your experiment and hardware. It also provides you with a precise log file of your experiment to allow you to check the precision with which things occurred. Some general considerations are discussed here and there are links with specific Timing.

Many scientists have asked “Can PsychoPy provide sub-millisecond timing precision?”. The short answer is yes it can - PsychoPy’s timing is as good as any software package we’ve tested (we’ve tested quite a lot).

BUT there are many components to getting good timing, and many ways that your timing could be less-than-perfect. So if timing is important to you then you should really read this entire section of the PsychoPy manual and you should test your timing using dedicated hardware (photodiodes, microphones or, ideally the Black Box Toolkit). We can’t emphasise enough how many ways there are for your hardware and/or operating system to break the good timing that PsychoPy is providing.

2.8.1 Can PsychoPy deliver millisecond precision?

The simple answer is ‘yes’. PsychoPy’s timing is as good as (or in most cases better than) any package out there.

The longer answer is that you should test the timing of your own experimental stimuli on your own hardware. Very often a computer is not optimally configured to produce good timing, and a poorly code experiment could also destroy your timing. Many software and hardware manufacturers will suggest that the key to this is using computer clocks with high precision (lots of decimal places) but that is not the answer at all. The sources of error in stimulus/response timing are almost never to do with the poor precision of the clock. The following issues are extremely common and until you actually test your experiment you don’t realise that your timing is being affected by them:

- Additional delays caused by monitors: e.g. the monitor taking additional time to process the image before presentation
- Delays caused by drivers and OS: Windows, Linux and Mac all perform further processing on the images, depending on settings and this can delay your visual stimulus delivery by a further frame interval or more
- Delays caused by coding errors
- Delays caused by keyboards
- Audio delays

The clocks that PsychoPy uses do have sub-millisecond precision but your keyboard has a latency of 4-25ms depending on your platform and keyboard. You could buy a response pad (e.g. a Labhackers Millikey) for response timing with a
sub-ms precision, but note that there will still be an apparent lag that is dependent on the monitor’s absolute lag and the position of the stimulus on it.

Also note that the variance in terms of response times between your participants, and from trial to trial within a participant, probably dwarfs that of your keyboard and monitor issues! That said, PsychoPy does aim to give you as high a temporal precision as possible and, in a well-configured system achieves this very well.

### 2.8.2 Computer monitors

There are several issues with monitors that you should be aware of.

1. **Monitors have fixed refresh rates**
2. **The top of the screen appears 5-15 ms before the bottom**
3. **Additional delays caused by monitors**

#### Monitors have fixed refresh rates

Most monitors have fixed refresh rates, typically 60 Hz for a flat-panel display. You probably knew that but it’s very easy to forget that this means certain stimulus durations won’t be possible. If your screen is a standard 60 Hz monitor then your frame period is 1/60 s, roughly 16.7 ms. That means you can generate stimuli that last for 16.7 ms, or 33.3 ms or 50 ms, but you **cannot** present a stimulus for 20, 40, or 60 ms.

The caveat to this is that you can now buy specialist monitors that support variable refresh rates (although not below at least 5 ms between refreshes). These are using a technology called G-Sync (nVidia) or FreeSync (everyone else) and PsychoPy can make use of those technologies but support isn’t built in to the library. See the publication by Poth et al (2018) for example code.

#### The top of the screen appears 5-15 ms before the bottom

For most monitor technologies, the lines of pixels are drawn sequentially from the top to the bottom and once the bottom line has been drawn the screen is finished and the line returns to the top (the Vertical Blank Interval, VBI). Most of your frame interval is spent drawing the lines, with 1-2ms being left for the VBI. This means that the pixels at the bottom are drawn “up to 10 ms later” than the pixels at the top of the screen. At what point are you going to say your stimulus ‘appeared’ to the participant?

#### Additional delays caused by monitors

Monitors themselves often cause delays on top of the unavoidable issue of having a refresh rate. Modern displays often have features to optimize the image, which will be often labelled as modes like “Movie Mode”, “Game Mode” etc. If your display has any such settings then you want to turn them off so as not to change your image. Not only do these settings entail altering the color of the pixels that your experiment generator is send to the screen (if you’ve spent time carefully calibrating your colors and then the monitor changes them it would be annoying) but these forms of “post-processing” take time and often a **variable** time.

If your monitor has any such “post-processing” enabled then you might well be seeing an additional 20-30 ms of (variable) lag added to the stimulus onset as a result. This **will not** be detected by psychoPy (or any other system) and will not show up in your log files.
Fig. 1: Figure 1: photodiode trace at top of screen. The image above shows the luminance trace of a CRT recorded by a fast photo-sensitive diode at the top of the screen when a stimulus is requested (shown by the square wave). The square wave at the bottom is from a parallel port that indicates when the stimulus was flipped to the screen. Note that on a CRT the screen at any point is actually black for the majority of the time and just briefly bright. The visual system integrates over a large enough time window not to notice this. On the next frame after the stimulus ‘presentation time’ the luminance of the screen flash increased.
Fig. 2: Figure 2: photodiode trace of the same large stimulus at bottom of screen. The image above shows comes from exactly the same script as the above but the photodiode is positioned at the bottom of the screen. In this case, after the stimulus is 'requested' the current frame (which is dark) finishes drawing and then, 10ms later than the above image, the screen goes bright at the bottom.
2.8.3 Delays caused by drivers and OS

All three major operating systems are capable of introducing timing errors into your visual presentations, although these are usually observed as (relatively) constant lags. The particularly annoying factor here is that your experiment might work with very good timing for a while and then the operating system performs and automatic update and the timing gets worse! Again, the only way you would typically know about these sorts of changes is by testing with hardware.

**Triple buffering:** In general PsychoPy, and similar graphics systems, are expecting a double-buffered rendering pipeline, whereby we are drawing to one copy of the screen (the “back buffer”) and when we have finished drawing our stimuli we “flip” the screen, at which point it will wait for the next screen refresh period and become visible as the “front buffer”. Triple-buffering is a system whereby the images being rendered to the screen are put in a 3rd buffer, and the operating system can do further processing as the rendered image moves from this 3rd buffer to the back buffer. Such a system means that your images all appear exactly one frame later than expected.

Errors caused by triple buffering, either by the operating system or by the monitor, cannot be detected by PsychoPy and will not show up in your log files.

**MacOS**

The stimulus presentation on MacOS used to be very good, up until version 10.12. In MacOS 10.13 something changed and it appears that a form of triple buffering has been added and, to date, none of the major experiment generators have managed to turn this off. As a result, since MacOS 10.13 stimuli appear always to be presented a screen refresh period later than expected, resulting in a delay of 16.66 ms in the apparent response times to visual stimuli.

**Windows 10**

In Windows, triple buffering is something that might be turned on by default in your graphics card settings (look for 3D, or OpenGL, settings in the driver control panel to turn this off). The reason it gets used is that it often results in a more consistent frame rate for games, but having the frame appear later then expected is typically bad for experiments!

As well as the graphics card performing triple buffering, the operating system itself (via the Desktop Window Manager) does so under certain conditions: - Anytime a window is used (instead of full-screen mode) Windows 10 now uses triple buffering - having Scaling set to anything other than 100% also results in triple-buffering (presumably Microsoft renders the screen once and then scales it during the next refresh).

There are surely other settings in Windows and the graphics card that will alter the timing performance and, again, until you test these you aren’t likely to know.

**Linux**

In Linux, again, timing performance of the visual stimuli depends on the graphics card driver but we have also seen timing issues arising from the Window Compositor and with interactions between compositor and driver.

The real complication here is that in Linux there are many different window compositors (Compiz, XFwm, Enlightenment,…), as well as different options for drivers to install (e.g. for nVidia cards there is a proprietary nVidia driver as well as an open-source “Nouveau” driver which is often the default but has worse performance).

Ultimately, you need to **test the timing with hardware** and work through the driver/compositor settings to optimise the timing performance.
2.8.4 Delays caused by coding errors

It can be really easy, as a user, to introduce timing errors into your experiment with incorrect coding. Even if you really know what you’re doing, it’s easy to make a silly mistake, and if you don’t really know what you’re doing then all bets are definitely off!!

Common ways for this to happen are to forget the operations that are potentially time-consuming. The biggest of these is the loading of images from disk.

For image stimuli where the image is constant the image should be loaded from disk at the beginning of the script (Builder-generate experiments will do so automatically for you). When an image stimulus has to change on each trial, it must be loaded from disk at some point. That typically takes several milliseconds (possibly hundreds of milliseconds for a large image) and while that is happening the screen will not be refreshing. You need to take your image-loading time into account and allow it to occur during a static period of the screen.

In Builder experiments if you set something to update “On every repeat” then it will update as that Routine begins so, if your trial Routine simply begins with 0.5s fixation period, all your stimuli can be loaded/updated in that period and you will have no further problems. Sometimes you want to load/update your stimulus explicitly at a different point in time and then you can insert a “Static Component” into your Builder experiment (a “Static Period” in the Python API) and then set your stimulus to update during that period (it will show up as an update option after you insert the Static Component).

The good news is that a lot of the visual timing issues caused by coding problems are visible in the log files, unlike the problems with hardware and operating systems introducing lags.

2.8.5 Delays caused by keyboards

Keyboards are hopeless for timing. We should expand on that. But for now, it’s all you need to know! Get yourself a button box, like the LabHackers Millikey.

2.8.6 Audio delays

PsychoPy has a number of settings for audio and the main issue here is that the user needs to know to turn on the optimal settings.

For years we were looking for a library that provided fast reliable audio and we went through an number of libraries to optimize that (pygame was the first, with 100ms latencies, then pyo and sounddevice which were faster).

Most recently we added support for the Psychophysics Toolbox audio library (PsychPortAudio), which Mario Kleiner has ported Python in 2018. With that library we can achieve really remarkable audio timing (thanks to Mario for his fantastic work). But still there are several things you need to check to make use of this library and use it to its full potential:

- Make sure you’re running with a 64bit installation of Python3. The PsychPortAudio code has not, and almost certainly will not, be built to support legacy Python installations
- Set the PsychoPy preferences to use it! As of PsychoPy version 3.2.x the PTB backend was not the default. In future versions this will probably be the default, but as of version 3.2.x you need to set PsychoPy to use it (we didn’t want to make it the default until it had been used without issue in a number of labs in “the wild”).
- Make sure that the library settings are using a high

For further information please see the documentation about the Sound library
Fig. 3: With the new PTB library you can achieve not only sub-millisecond precision, but roughly sub-millisecond lags!! You do need to know how to configure this though and testing it can only be done with hardware.

### 2.8.7 Non-slip timing for imaging

For most behavioural/psychophysics studies timing is most simply controlled by setting some timer (e.g. a `Clock()`) to zero and waiting until it has reached a certain value before ending the trial. We might call this a 'relative' timing method, because everything is timed from the start of the trial/epoch. In reality this will cause an overshoot of some fraction of one screen refresh period (10ms, say). For imaging (EEG/MEG/fMRI) studies adding 10ms to each trial repeatedly for 10 minutes will become a problem, however. After 100 stimulus presentations your stimulus and scanner will be de-synchronised by 1 second.

There are two ways to get around this:

1. **Time by frames** If you are confident that you *aren’t dropping frames* then you could base your timing on frames instead to avoid the problem.

2. **Non-slip (global) clock timing** The other way, which for imaging is probably the most sensible, is to arrange timing based on a global clock rather than on a relative timing method. At the start of each trial you add the (known) duration that the trial will last to a *global* timer and then wait until that timer reaches the necessary value. To facilitate this, the PsychoPy `Clock()` was given a new `add()` method as of version 1.74.00 and a `CountdownTimer()` was also added.

The non-slip method can only be used in cases where the trial is of a known duration at its start. It cannot, for example, be used if the trial ends when the subject makes a response, as would occur in most behavioural studies.
Non-slip timing from the Builder

(new feature as of version 1.74.00)

When creating experiments in the Builder, PsychoPy will attempt to identify whether a particular Routine has a known endpoint in seconds. If so then it will use non-slip timing for this Routine based on a global countdown timer called routineTimer. Routines that are able to use this non-slip method are shown in green in the Flow, whereas Routines using relative timing are shown in red. So, if you are using PsychoPy for imaging studies then make sure that all the Routines within your loop of epochs are showing as green. (Typically your study will also have a Routine at the start waiting for the first scanner pulse and this will use relative timing, which is appropriate).

2.8.8 Detecting dropped frames

Occasionally you will drop frames if you:

- try to do too much drawing
- do it in an inefficient manner (write poor code)
- have a poor computer/graphics card

Things to avoid:

- recreating textures for stimuli
- building new stimuli from scratch (create them once at the top of your script and then change them using stim.setOri(ori)(), stim.setPos([x,y],...)

Turn on frame time recording

The key sometimes is knowing if you are dropping frames. PsychoPy can help with that by keeping track of frame durations. By default, frame time tracking is turned off because many people don’t need it, but it can be turned on any time after Window creation:

```python
from psychopy import visual
win = visual.Window([800,600])
win.recordFrameIntervals = True
```

Since there are often dropped frames just after the system is initialised, it makes sense to start off with a fixation period, or a ready message and don’t start recording frame times until that has ended. Obviously if you aren’t refreshing the window at some point (e.g. waiting for a key press with an unchanging screen) then you should turn off the recording of frame times or it will give spurious results.

Warn me if I drop a frame

The simplest way to check if a frame has been dropped is to get PsychoPy to report a warning if it thinks a frame was dropped:

```python
from __future__ import division, print_function
from psychopy import visual, logging
win = visual.Window([800,600])
win.recordFrameIntervals = True
```

(continues on next page)
# By default, the threshold is set to 120% of the estimated refresh
duration, but arbitrary values can be set.

I've got 85Hz monitor and want to allow 4 ms tolerance; any refresh that
takes longer than the specified period will be considered a "dropped"
frame and increase the count of win.nDroppedFrames.

```python
win.refreshThreshold = 1 / 85 + 0.004
```

# Set the log module to report warnings to the standard output window
# (default is errors only).

```python
logging.console.setLevel(logging.WARNING)
```

```python
print('Overall, %i frames were dropped.' % win.nDroppedFrames)
```

**Show me all the frame times that I recorded**

While recording frame times, these are simply appended, every frame to win.frameIntervals (a list). You can simply plot these at the end of your script using matplotlib:

```python
import matplotlib.pyplot as plt
plt.plot(win.frameIntervals)
plt.show()
```

Or you could save them to disk. A convenience function is provided for this:

```python
win.saveFrameIntervals(fileName=None, clear=True)
```

The above will save the currently stored frame intervals (using the default filename, ‘lastFrameIntervals.log’) and then clears the data. The saved file is

- a simple text file.

At any time you can also retrieve the time of the /last/ frame flip using win.lastFrameT (the time is synchronised with logging.defaultClock so it will match any logging commands that your script uses).

**'Blocking' on the VBI**

As of version 1.62 PsychoPy ‘blocks’ on the vertical blank interval meaning that, once Window.flip() has been called, no code will be executed until that flip actually takes place. The timestamp for the above frame interval measurements is taken immediately after the flip occurs. Run the timeByFrames demo in Coder to see the precision of these measurements on your system. They should be within 1ms of your mean frame interval.

Note that Intel integrated graphics chips (e.g. GMA 945) under win32 do not sync to the screen at all and so blocking on those machines is not possible.
2.8.9 Reducing dropped frames

There are many things that can affect the speed at which drawing is achieved on your computer. These include, but are probably not limited to; your graphics card, CPU, operating system, running programs, stimuli, and your code itself. Of these, the CPU and the OS appear to make rather little difference. To determine whether you are actually dropping frames see Detecting dropped frames.

Things to change on your system:

1. make sure you have a good graphics card. Avoid integrated graphics chips, especially Intel integrated chips and especially on laptops (because on these you don’t get to change your mind so easily later). In particular, try to make sure that your card supports OpenGL 2.0

2. shut down as many programs, including background processes. Although modern processors are fast and often have multiple cores, substantial disk/memory accessing can cause frame drops
   - anti-virus auto-updating (if you're allowed)
   - email checking software
   - file indexing software
   - backup solutions (e.g. TimeMachine)
   - Dropbox
   - Synchronisation software

Writing optimal scripts

1. run in full-screen mode (rather than simply filling the screen with your window). This way the OS doesn't have to spend time working out what application is currently getting keyboard/mouse events.

2. don't generate your stimuli when you need them. Generate them in advance and then just modify them later with the methods like setContrast(), setOrientation() etc…

3. calls to the following functions are comparatively slow; they require more CPU time than most other functions and then have a call to setTexture(). This is especially true when you need to load an image from disk too as the texture.
   1. GratingStim.setTexture()
   2. RadialStim.setTexture()
   3. TextStim.setText()

4. if you don't have OpenGL 2.0 then calls to setContrast, setRGB and setOpacity will also be slow, because they also make a call to setTexture(). If you have shader support then this call is not necessary and a large speed increase will result.

5. avoid loops in your python code (use numpy arrays to do maths with lots of elements)

6. if you need to create a large number (e.g. greater than 10) similar stimuli, then try the ElementArrayStim
Possible good ideas

It isn’t clear that these actually make a difference, but they might.

1. disconnect the internet cable (to prevent programs performing auto-updates?)
2. on Macs you can actually shut down the Finder. It might help. See Alex Holcombe’s page [here](#).
3. use a single screen rather than two (probably there is some graphics card overhead in managing double the number of pixels?)

### 2.9 Glossary

**Adaptive staircase** An experimental method whereby the choice of stimulus parameters is not pre-determined but based on previous responses. For example, the difficulty of a task might be varied trial-to-trial based on the participant’s responses. These are often used to find psychophysical thresholds. Contrast this with the **method of constants**.

**CPU** **Central Processing Unit** is the main processor of your computer. This has a lot to do, so we try to minimise the amount of processing that is needed, especially during a trial, when time is tight to get the stimulus presented on every screen refresh.

**CRT** **Cathode Ray Tube** ‘Traditional’ computer monitor (rather than an LCD or plasma flat screen).

**csv** **Comma-Separated Value files** Type of basic text file with ‘comma-separated values’. This type of file can be opened with most spreadsheet packages (e.g. MS Excel) for easy reading and manipulation.

**GPU** **Graphics Processing Unit** is the processor on your graphics card. The GPUs of modern computers are incredibly powerful and it is by allowing the GPU to do a lot of the work of rendering that PsychoPy is able to achieve good timing precision despite being written in an interpreted language.

**Method of constants** An experimental method whereby the parameters controlling trials are predetermined at the beginning of the experiment, rather than determined on each trial. For example, a stimulus may be presented for 3 pre-determined time periods (100, 200, 300ms) on different trials, and then repeated a number of times. The order of presentation of the different conditions can be randomised or sequential (in a fixed order). Contrast this method with the **adaptive staircase**.

**VBI** **(Vertical Blank Interval, aka the Vertical Retrace, or Vertical Blank, VBL)**. The period in-between video frames and can be used for synchronising purposes. On a CRT display the screen is black during the VBI and the display beam is returned to the top of the display.

**VBI blocking** The setting whereby all functions are synced to the VBI. After a call to `psychopy.visual.Window.flip()` nothing else occurs until the VBI has occurred. This is optimal and allows very precise timing, because as soon as the flip has occurred a very precise time interval is known to have occurred.

**VBI syncing** (aka vsync) The setting whereby the video drawing commands are synced to the VBI. When psychopy.visual.Window.flip() is called, the current back buffer (where drawing commands are being executed) will be held and drawn on the next VBI. This does not necessarily entail **VBI blocking** (because the system may return and continue executing commands) but does guarantee a fixed interval between frames being drawn.

**xlsx** **Excel OpenXML file format**. A spreadsheet data format developed by Microsoft but with an open (published) format. This is the native file format for Excel (2007 or later) and can be opened by most modern spreadsheet applications including OpenOffice (3.0+), google docs, Apple iWork 08.
3.1 Download

For the easiest installation download and install the Standalone package.

For all versions see the PsychoPy releases on github

PsychoPy is distributed under the GPL3 license

3.2 Manual installations

See below for options if you don’t want to use the Standalone releases:

- pip install
- brew install
- Linux
- Anaconda and Miniconda
- Developers install

3.2.1 pip install

Now that most python libraries can be installed using pip it’s relatively easy to manually install PsychoPy and all it’s dependencies to your own installation of Python.

The steps are to fetch Python. This method should work on any version of Python but we recommend Python 3.6 for now.

You can install PsychoPy and its dependencies (more than you’ll strictly need) by:

```
pip install psychopy
```

If you prefer not to install all the dependencies then you could do:

```
pip install psychopy --no-deps
```

and then install them manually.
3.2.2 brew install

On a MacOS machine, `brew` can be used to install PsychoPy:

```
brew cask install psychopy
```

3.2.3 Linux

There used to be neurodebian and Gentoo packages for PsychoPy but these are both badly outdated. We’d recommend you do:

```
# with --no-deps flag if you want to install dependencies manually
pip install psychopy
```

Then fetch a **wxPython wheel** for your platform from:


and having downloaded the right wheel you can then install it with something like:

```
pip install path/to/your/wxpython.whl
```

wxPython>4.0 and doesn’t have universal wheels yet which is why you have to find and install the correct wheel for your particular flavor of Linux.

**Building Python PsychToolbox bindings:**

The PsychToolbox bindings for Python provide superior timing for sounds and keyboard responses. Unfortunately we haven’t been able to build universal wheels for these yet so you may have to build the pkg yourself. That should not be hard. You need the necessary dev libraries installed first:

```
sudo apt-get install libusb-1.0-0-dev portaudio19-dev libasound2-dev
```

and then you should be able to install using pip and it will build the extensions as needed:

```
pip install psychtoolbox
```

3.2.4 Anaconda and Miniconda

We provide an **environment file** that can be used to install PsychoPy and its dependencies. Download the file, open your terminal, navigate to the directory you saved the file to, and run:

```
conda env create -n psychopy -f psychopy-env.yml
```

This will create an environment named `psychopy`. On Linux, the **wxPython dependency of PsychoPy is linked against webkitgtk**, which needs to be installed manually, e.g. via `sudo apt install libwebkitgtk-1.0` on Debian-based systems like Ubuntu.

To activate the newly-created environment and run PsychoPy, execute:

```
conda activate psychopy
psychopy
```
3.2.5 Developers install

Ensure you have Python 3.6 and the latest version of pip installed:

```
python --version
pip --version
```

Next, follow instructions here to fork and fetch the latest version of the PsychoPy repository.

From the directory where you cloned the latest PsychoPy repository (i.e., where setup.py resides), run:

```
pip install -e .
```

This will install all PsychoPy dependencies to your default Python distribution (which should be Python 3.6). Next, you should create a new PsychoPy shortcut linking your newly installed dependencies to your current version of PsychoPy in the cloned repository. To do this, simply create a new .BAT file containing:

```
"C:\PATH_TO_PYTHON3.6\python.exe C:\PATH_TO_CLONED_PSYCHOPY_REPO\psychopy\app\psychopyApp.py"
```

Alternatively, you can run the psychopyApp.py from the command line:

```
python C:\PATH_TO_CLONED_PSYCHOPY_REPO\psychopy\app\psychopyApp
```

3.3 Recommended hardware

The minimum requirement for PsychoPy is a computer with a graphics card that supports OpenGL. Many newer graphics cards will work well. Ideally the graphics card should support OpenGL version 2.0 or higher. Certain visual functions run much faster if OpenGL 2.0 is available, and some require it (e.g. ElementArrayStim).

If you already have a computer, you can install PsychoPy and the Configuration Wizard will auto-detect the card and drivers, and provide more information. It is inexpensive to upgrade most desktop computers to an adequate graphics card. High-end graphics cards can be very expensive but are only needed for very intensive use.

Generally NVIDIA and ATI (AMD) graphics chips have higher performance than Intel graphics chips so try and get one of those instead.

3.3.1 Notes on OpenGL drivers

On Windows, if you get an error saying “pyglet.gl.ContextException: Unable to share contexts” then the most likely cause is that you need OpenGL drivers and your built-in Windows only has limited support for OpenGL (or possibly you have an Intel graphics card that isn’t very good). Try installing new drivers for your graphics card from its manufacturer’s web page, not from Microsoft. For example, NVIDIA provides drivers for its cards here: https://www.nvidia.com/Download/index.aspx
As an application, PsychoPy has two main views: the **Builder** view, and the **Coder** view. It also has a underlying **Reference Manual (API)** that you can call directly.

1. **Builder.** You can generate a wide range of experiments easily from the Builder using its intuitive, graphical user interface (GUI). This might be all you ever need to do. But you can always compile your experiment into a python script for fine-tuning, and this is a quick way for experienced programmers to explore some of PsychoPy’s libraries and conventions.

1. **Coder.** For those comfortable with programming, the Coder view provides a basic code editor with syntax highlighting, code folding, and so on. Importantly, it has its own output window and Demo menu. The demos illustrate how to do specific tasks or use specific features; they are not whole experiments. The **Coder tutorials** should help get you going, and the **Reference Manual (API)** will give you the details.

The Builder and Coder views are the two main aspects of the PsychoPy application. If you’ve installed the StandAlone version of PsychoPy on **MS Windows** then there should be an obvious link to PsychoPy in your > Start > Programs. If you installed the StandAlone version on **macOS** then the application is where you put it (!). On these two platforms you can open the Builder and Coder views from the View menu and the default view can be set from the preferences. **On Linux,** you can start PsychoPy from a command line, or make a launch icon (which can depend on the desktop and distro). If the PsychoPy app is started with flags —-coder (or -c), or —-builder (or -b), then the preferences will be overridden and that view will be created as the app opens.

For experienced python programmers, it’s possible to use PsychoPy without ever opening the Builder or Coder. Install the PsychoPy libraries and dependencies, and use your favorite IDE instead of the Coder.
Welcome to PsychoPy2!
v1.63.00
4.1 Builder

When learning a new computer language, the classic first program is simply to print or display “Hello world!”. Let’s do it.

4.1.1 A first program

Start PsychoPy, and be sure to be in the Builder view.

- If you have poked around a bit in the Builder already, be sure to start with a clean slate. To get a new Builder view, type Ctrl-N on Windows or Linux, or Cmd-N on Mac.
- Click on a Text component

![Text Properties dialog](image)

and a Text Properties dialog will pop up.

- In the Text field, replace the default text with your message. When you run the program, the text you type here will be shown on the screen.
• Click OK (near the bottom of the dialog box). (Properties dialogs have a link to online help—an icon at the bottom, near the OK button.)

• Your text component now resides in a routine called trial. You can click on it to view or edit it. (Components, Routines, and other Builder concepts are explained in the Builder documentation.)

• Back in the main Builder, type Ctrl-R (Windows, Linux) or Cmd-R (Mac), or use the mouse to click the Run icon.

Assuming you typed in “Hello world!”, your screen should have looked like this (briefly):

![Hello world!](image)

If nothing happens or it looks wrong, recheck all the steps above; be sure to start from a new Builder view.

What if you wanted to display your cheerful greeting for longer than the default time?

• Click on your Text component (the existing one, not a new one).

• Edit the Stop duration (s) to be 3.2; times are in seconds.

• Click OK.

• And finally Run.

When running an experiment, you can quit by pressing the escape key (this can be configured or disabled). You can quit PsychoPy from the File menu, or typing Ctrl-Q / Cmd-Q.

4.1.2 Getting beyond Hello

To do more, you can try things out and see what happens. You may want to consult the Builder documentation. Many people find it helpful to explore the Builder demos, in part to see what is possible, and especially to see how different things are done.

A good way to develop your own first PsychoPy experiment is to base it on the Builder demo that seems closest. Copy it, and then adapt it step by step to become more and more like the program you have in mind. Being familiar with the Builder demos can only help this process.
You could stop here, and just use the Builder for creating your experiments. It provides a lot of the key features that
people need to run a wide variety of studies. But it does have its limitations. When you want to have more complex
designs or features, you’ll want to investigate the Coder. As a segue to the Coder, lets start from the Builder, and see how
Builder programs work.

4.2 Builder-to-coder

Whenever you run a Builder experiment, PsychoPy will first translate it into python code, and then execute that code.
To get a better feel for what was happening “behind the scenes” in the Builder program above:

- In the Builder, load or recreate your “hello world” program.
- Instead of running the program, explicitly convert it into python: Type F5, or click the Compile icon:

The view will automatically switch to the Coder, and display the python code. If you then save and run this code, it would
look the same as running it directly from the Builder.

It is always possible to go from the Builder to python code in this way. You can then edit that code and run it as a python
program. However, you cannot go from code back to a Builder representation.

To switch quickly between Builder and Coder views, you can type Ctrl-L / Cmd-L.

4.3 Coder

Being able to inspect Builder-generated code is nice, but it’s possible to write code yourself, directly. With the Coder
and various libraries, you can do virtually anything that your computer is capable of doing, using a full-featured modern
programming language (python).

For variety, lets say hello to the Spanish-speaking world. PsychoPy knows Unicde (UTF-8).

If you are not in the Coder, switch to it now.

- Start a new code document: Ctrl-N / Cmd-N.
- Type (or copy & paste) the following:

```python
from psychopy import visual, core

win = visual.Window()
msg = visual.TextStim(win, text=u"\u00A1Hola mundo!")

msg.draw()
win.flip()
core.wait(1)
win.close()
```

- Save the file (the same way as in Builder).
- Run the script.
Note that the same events happen on-screen with this code version, despite the code being much simpler than the code generated by the Builder. (The Builder actually does more, such as prompt for a subject number.)

**Coder Shell**

The shell provides an interactive python interpreter, which means you can enter commands here to try them out. This provides yet another way to send your salutations to the world. By default, the Coder’s output window is shown at the bottom of the Coder window. Click on the Shell tab, and you should see python’s interactive prompt, `>>>`:

```
PyShell in PsychoPy - type some commands!
Type "help", "copyright", "credits" or "license" for more information.
```  
At the prompt, type:

```
>>> print u"\u00A1Hola mundo!"
```

You can do more complex things, such as type in each line from the Coder example directly into the Shell window, doing so line by line:

```
>>> from psychopy import visual, core
```
and then:

```
>>> win = visual.Window()
```
and so on—watch what happens each line:

```
>>> msg = visual.TextStim(win, text=u"\u00A1Hola mundo!")
>>> msg.draw()
>>> win.flip()
```

and so on. This lets you try things out and see what happens line-by-line (which is how python goes through your program).
Building experiments in a GUI

You can now see a YouTube PsychoPy tutorial showing you how to build a simple experiment in the Builder interface.

Note: The Builder view is now (at version 1.75) fairly well-developed and should be able to construct a wide variety of studies. But you should still check carefully that the stimuli and response collection are as expected.

Contents:
5.1 Builder concepts

5.1.1 Routines and Flow

The Builder view of the PsychoPy application is designed to allow the rapid development of a wide range of experiments for experimental psychology and cognitive neuroscience experiments.

The Builder view comprises two main panels for viewing the experiment’s Routines (upper left) and another for viewing the Flow (lower part of the window).

An experiment can have any number of Routines, describing the timing of stimuli, instructions and responses. These are portrayed in a simple track-based view, similar to that of video-editing software, which allows stimuli to come on and go off repeatedly and to overlap with each other.

The way in which these Routines are combined and/or repeated is controlled by the Flow panel. All experiments have exactly one Flow. This takes the form of a standard flowchart allowing a sequence of routines to occur one after another, and for loops to be inserted around one or more of the Routines. The loop also controls variables that change between repetitions, such as stimulus attributes.

5.1.2 Example 1 - a reaction time experiment

For a simple reaction time experiment there might be 3 Routines, one that presents instructions and waits for a keypress, one that controls the trial timing, and one that thanks the participant at the end. These could then be combined in the Flow so that the instructions come first, followed by trial, followed by the thanks Routine, and a loop could be inserted so that the Routine repeated 4 times for each of 6 stimulus intensities.

5.1.3 Example 2 - an fMRI block design

Many fMRI experiments present a sequence of stimuli in a block. For this there are multiple ways to create the experiment:

- We could create a single Routine that contained a number of stimuli and presented them sequentially, followed by a long blank period to give the inter-epoch interval, and surround this single Routine by a loop to control the blocks.
- Alternatively we could create a pair of Routines to allow presentation of a) a single stimulus (for 1 sec) and b) a blank screen, for the prolonged period. With these Routines we could insert pair of loops, one to repeat the stimulus Routine with different images, followed by the blank Routine, and another to surround this whole set and control the blocks.

5.1.4 Demos

There are a couple of demos included with the package, that you can find in their own special menu. When you load these the first thing to do is make sure the experiment settings specify the same resolution as your monitor, otherwise the screen can appear off-centred and strangely scaled.
Stroop demo

This runs a digital demonstration of the Stroop effect\(^1\). The experiment presents a series of coloured words written in coloured ‘inks’. Subjects have to report the colour of the letters for each word, but find it harder to do so when the letters are spelling out a different (incongruous) colour. Reaction times for the congruent trials (where letter colour matches the written word) are faster than for the incongruent trials.

From this demo you should note:

- How to setup a trial list in a .csv or .xlsx file
- How to record key presses and reaction times (using the \texttt{resp Component} in \texttt{trial Routine})
- How to change a stimulus parameter on each repetition of the loop. The text and rgb values of the \texttt{word Component} are based on \texttt{thisTrial}, which represents a single iteration of the \texttt{trials loop}. They have been set to change every repeat (don't forget that step!)
- How to present instructions: just have a long-lasting \texttt{TextStim} and then force end of the \texttt{Routine} when a key is pressed (but don't bother storing the key press).

Psychophysics Staircase demo

This is a mini psychophysics experiment, designed to find the contrast detection threshold of a gabor i.e. find the contrast where the observer can just see the stimulus.

From this demo you should note:

- The opening dialog box requires the participant to enter the orientation of the stimulus, the required fields here are determined by ‘Experiment Info’ in ‘Preferences’ which is a python dictionary. This information is then entered into the stimulus parameters using '$expInfo['ori']'
- The phase of the stimulus is set to change every frame and its value is determined by the value of \texttt{trial-Clock.getTime()*2}. Every \texttt{Routine} has a clock associated with it that gets reset at the beginning of the iteration through the \texttt{Routine}. There is also a \texttt{globalClock} that can be used in the same way. The phase of a \texttt{Patch Component} ranges 0-1 (and wraps to that range if beyond it). The result in this case is that the grating drifts at a rate of 2Hz.
- The contrast of the stimulus is determined using an adaptive staircase. The \texttt{Staircase methods} are different to those used for a loop which uses predetermined values. An important thing to note is that you must define the correct answer.

5.2 Routines

An experiment consists of one or more Routines. A Routine might specify the timing of events within a trial or the presentation of instructions or feedback. Multiple Routines can then be combined in the \texttt{Flow}, which controls the order in which these occur and the way in which they repeat.

To create a new Routine, use the Experiment menu. The display size of items within a routine can be adjusted (see the View menu).

Within a Routine there are a number of components. These components determine the occurrence of a stimulus, or the recording of a response. Any number of components can be added to a Routine. Each has its own line in the Routine view that shows when the component starts and finishes in time, and these can overlap.

For now the time axis of the Routines panel is fixed, representing seconds (one line is one second). This will hopefully change in the future so that units can also be number of frames (more precise) and can be scaled up or down to allow very long or very short Routines to be viewed easily. That’s on the wishlist…

5.3 Flow

In the Flow panel a number of Routines can be combined to form an experiment. For instance, your study may have a Routine that presented initial instructions and waited for a key to be pressed, followed by a Routine that presented one trial which should be repeated 5 times with various different parameters set. All of this is achieved in the Flow panel. You can adjust the display size of the Flow panel (see View menu).

5.3.1 Adding Routines

The Routines that the Flow will use should be generated first (although their contents can be added or altered at any time). To insert a Routine into the Flow click the appropriate button in the left of the Flow panel or use the Experiment menu. A dialog box will appear asking which of your Routines you wish to add. To select the location move the mouse to the section of the flow where you wish to add it and click on the black disk.

5.3.2 Loops

Loops control the repetition of Routines and the choice of stimulus parameters for each. PsychoPy can generate the next trial based on the method of constants or using an adaptive staircase. To insert a loop use the button on the left of the Flow panel, or the item in the Experiment menu of the Builder. The start and end of a loop is set in the same way as the location of a Routine (see above). Loops can encompass one or more Routines and other loops (i.e. they can be nested).

As with components in Routines, the loop must be given a name, which must be unique and made up of only alpha-numeric characters (underscores are allowed). I would normally use a plural name, since the loop represents multiple repeats of something. For example, trials, blocks or epochs would be good names for your loops.

It is usually best to use trial information that is contained in an external file (.xlsx or .csv). When inserting a loop into the flow you can browse to find the file you wish to use for this. An example of this kind of file can be found in the Stroop demo (trialTypes.xlsx). The column names are turned into variables (in this case text, letterColor, corrAns and congruent), these can be used to define parameters in the loop by putting a $ sign before them e.g. $text. As the column names from the input file are used in this way they must have legal variable names i.e. they must be unique, have no punctuation or spaces (underscores are ok) and must not start with a digit.

The parameter Is trials exists because some loops are not there to indicate trials per se but a set of stimuli within a trial, or a set of blocks. In these cases we don’t want the data file to add an extra line with each pass around the loop. This parameter can be unchecked to improve (hopefully) your data file outputs. [Added in v1.81.00]

Method of Constants

Selecting a loop type of random, sequential, or fullRandom will result in a method of constants experiment, whereby the types of trials that can occur are predetermined. That is, the trials cannot vary depending on how the subject has responded on a previous trial. In this case, a file must be provided that describes the parameters for the repeats. This should be an Excel 2007 (.xlsx) file or a comma-separated-value (.csv) file in which columns refer to parameters that are needed to describe stimuli etc. and rows one for each type of trial. These can easily be generated from a spreadsheet package like Excel. (Note that csv files can also be generated using most text editors, as long as they allow you to save the file as “plain text”; other output formats will not work, including “rich text”). The top row should be a row of headers: text labels describing the contents of the respective columns. (Headers must also not include spaces or other characters other
than letters, numbers or underscores and must not be the same as any variable names used elsewhere in your experiment.)
For example, a file containing the following table:

<table>
<thead>
<tr>
<th>ori</th>
<th>text</th>
<th>corrAns</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>aaa</td>
<td>left</td>
</tr>
<tr>
<td>90</td>
<td>aaa</td>
<td>left</td>
</tr>
<tr>
<td>0</td>
<td>bbb</td>
<td>right</td>
</tr>
<tr>
<td>90</td>
<td>bbb</td>
<td>right</td>
</tr>
</tbody>
</table>

would represent 4 different conditions (or trial types, one per line). The header line describes the parameters in the 3 columns: ori, text and corrAns. It's really useful to include a column called corrAns that shows what the correct key press is going to be for this trial (if there is one).

If the loop type is sequential then, on each iteration through the Routines, the next row will be selected in the order listed in the file. Under a random order, the next row will be selected at random (without replacement); it can only be selected again after all the other rows have also been selected. nReps determines how many repeats will be performed (for all conditions). The total number of trials will be the number of conditions (= number of rows in the file, not counting the header row) times the number of repetitions, nReps. With the fullRandom option, the entire list of trials including repetitions is used in random order, allowing the same item to appear potentially many times in a row, and to repeat without necessarily having done all of the other trials. For example, with 3 repetitions, a file of trial types like this:

<table>
<thead>
<tr>
<th>letter</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
</tr>
<tr>
<td>b</td>
</tr>
<tr>
<td>c</td>
</tr>
</tbody>
</table>

could result in the following possible sequences. sequential could only ever give one sequence with this order: [a b c a b c a b c]. random will give one of 216 different orders (= 3! * 3! * 3! = nReps * (nTrials!) ), for example: [b a c a b c a b c]. Here the letters are effectively in sets of (abc) (abc) (abc), and randomization is only done within each set, ensuring (for example) that there are at least two a’s before the subject sees a 3rd b. Finally, fullRandom will return one of 362,880 different orders (= 9! = (nReps * nTrials)! ) , such as [b b c a a c a a b], which random never would. There are no longer mini-blocks or “sets of trials” within the longer run. This means that, by chance, it would also be possible to get a very un-random-looking sequence like [a a a b b b c c c].

It is possible to achieve any sequence you like, subject to any constraints that are logically possible. To do so, in the file you specify every trial in the desired order, and the for the loop select sequential order and nReps=1.

**Selecting a subset of conditions**

In the standard Method of Constants you would use all the rows/conditions within your conditions file. However there are often times when you want to select a subset of your trials before randomising and repeating.

The parameter Select rows allows this. You can specify which rows you want to use by inserting values here:

- 0,2,5 gives the 1st, 3rd and 6th entry of a list - Python starts with index zero)
- $random(4)*10$ gives 4 indices from 0 to 9 (so selects 4 out of 10 conditions)
- 5:10 selects the 6th to 10th rows
- $mysIndices$ uses a variable that you’ve already created

Note in the last case that 5:8 isn’t valid syntax for a variable so you cannot do:

```
myIndices = 5:8
```

but you can do:
myIndices = slice(5,8)  #python object to represent a slice
myIndices = "5:8"  #a string that PsychoPy can then parse as a slice later
myIndices = "5:8:2"  #as above but

Note that PsychoPy uses Python's built-in slicing syntax (where the first index is zero and the last entry of a slice doesn’t get included). You might want to check the outputs of your selection in the Python shell (bottom of the Coder view) like this:

```python
>>> range(100)[5:8]  #slice 5:8 of a standard set of indices
[5, 6, 7]
>>> range(100)[5:10:2]  #slice 5:8 of a standard set of indices
[5, 7, 9, 11, 13, 15, 17, 19]
```

Check that the conditions you wanted to select are the ones you intended!

### Staircase methods

The loop type *staircase* allows the implementation of adaptive methods. That is, aspects of a trial can depend on (or “adapt to”) how a subject has responded earlier in the study. This could be, for example, simple up-down staircases where an intensity value is varied trial-by-trial according to certain parameters, or a stop-signal paradigm to assess impulsivity. For this type of loop a ‘correct answer’ must be provided from something like a *Keyboard Component*. Various parameters for the staircase can be set to govern how many trials will be conducted and how many correct or incorrect answers make the staircase go up or down.

### Accessing loop parameters from components

The parameters from your loops are accessible to any component enclosed within that loop. The simplest (and default) way to address these variables is simply to call them by the name of the parameter, prepended with $ to indicate that this is the name of a variable. For example, if your Flow contains a loop with the above table as its input trial types file then you could give one of your stimuli an orientation $ori which would depend on the current trial type being presented.

Example scenarios:

1. You want to loop randomly over some conditions in a loop called *trials*. Your conditions are stored in a csv file with headings ‘ori’, ‘text’, ‘corrAns’ which you provide to this loop. You can then access these values from any component using $ori, $text, and $corrAns

2. You create a random loop called *blocks* and give it an Excel file with a single column called *movieName* listing filenames to be played. On each repeat you can access this with $movieName

3. You create a staircase loop called *stairs*. On each trial you can access the current value in the staircase with $thisStair

**Note:** When you set a component to use a parameter that will change (e.g on each repeat through the loop) you should remember to change the component parameter from *`constant`* to *`set every repeat`* or *`set every frame`* or it won’t have any effect!
Reducing namespace clutter (advanced)

The downside of the above approach is that the names of trial parameters must be different between every loop, as well as not matching any of the predefined names in python, numpy and PsychoPy. For example, the stimulus called movie cannot use a parameter also called movie (so you need to call it movieName). An alternative method can be used without these restrictions. If you set the Builder preference unclutteredNamespace to True you can then access the variables by referring to parameter as an attribute of the singular name of the loop prepended with this. For example, if you have a loop called trials which has the above file attached to it, then you can access the stimulus ori with $thisTrial.ori. If you have a loop called blocks you could use $thisBlock.corrAns.

Now, although the name of the loop must still be valid and unique, the names of the parameters of the file do not have the same requirements (they must still not contain spaces or punctuation characters).

5.4 Blocks of trials and counterbalancing

Many people ask how to create blocks of trials, how to randomise them, and how to counterbalance their order. This isn’t all that hard, although it does require a bit of thinking!

5.4.1 Blocking similar conditions

The key thing to understand is that you should not create different Routines for different trials in your blocks (if at all possible). Try to define your trials with a single Routine. For instance, let’s imagine you’re trying to create an experiment that presents a block of pictures of houses or a block of faces. It would be tempting to create a Routine called presentFace and another called presentHouse but you actually want just one called presentStim (or just trial) and then set that to differ as needed across different stimuli.

This example is included in the Builder demos, as of PsychoPy 1.85, as “images_blocks”.

You can add a loop around your trials, as normal, to control the trials within a block (e.g. randomly selecting a number of images) but then you will have a second loop around this to define how the blocks change. You can also have additional Routines like something to inform participants that the next block is about to start.

So, how do you get the block to change from one set of images to another? To do this create three spreadsheets, one for each block, determining the filenames within that block, and then another to control which block is being used:
**Setting up the basic conditions.** The facesBlock, and housesBlock, files look more like your usual conditions files. In this example we can just use a variable `stimFile` with values like `stims/face01.jpg` and `stims/face02.jpg` while the housesBlock file has `stims/house01.jpg` and `stims/house02.jpg`. In a real experiment you’d probably also have response keys and suchlike as well.

**So, how to switch between these files?** That’s the trick and that’s what the other file is used for. In the `chooseBlocks.xlsx` file you set up a variable called something like `condsFile` and that has values of `facesBlock.xlsx` and `housesBlock.xlsx`. In the outer (blocks) loop you set up the conditions file to be `chooseBlocks.xlsx` which creates a variable `condsFile`. Then, in the inner (trials) loop you set the conditions file not to be any file directly but simply `$condsFile`. Now, when PsychoPy starts this loop it will find the current value of `condsFile` and insert the appropriate thing, which will be the name of an conditions file and we’re away!

Your `chooseBlocks.xlsx` can contain other values as well, such as useful identifiers. In this demo you could add a value `readyText` that says “Ready for some houses”, and “Ready for some faces” and use this in your get ready Routine.

Variables that are defined in the loops are available anywhere within those. In this case, of course, the values in the outer loop are changing less often than the values in the inner loop.

### 5.4.2 Counterbalancing similar conditions

Counterbalancing is simply an extension of blocking. Until now, we have a *randomised block design*, where the order of blocks is set to random. At the moment we also only have one repeat of each block, but we could also present more than one repeat of each block by controlling the number of rows assigned to each block in our ‘chooseBlocks’ file.

In a counterbalanced design you want to control the order explicitly and you want to provide a different order for different groups of participants. Maybe group A always gets faces first, then houses, and group B always gets houses first, then faces.

Now we need to create further conditions files, to specify the exact orders we want, so we’d have something like `chooseBlockA.xlsx` and `chooseBlockB.xlsx`:

![chooseBlocksA.xlsx and chooseBlocksB.xlsx](image)

The last part of the puzzle is how to assign participants to groups. For this you *could* write a Code Component that would generate a variable for you (`if……: groupFile = "groupB.xlsx"`) but the easiest thing is probably that you, the experimenter, chooses this using the GUI we present at the start of the experiment. So, we add a field to our GUI using experiment settings:
Note that entering a list as the default input will present us with a dropdown in our GUI.

Finally, we set parameters of our blocks loop to use the method ‘sequential’ (because we are using a predefined order) and we enter the following into the conditions field: 
```
"$chooseBlocks"+expinfo['group']+.xlsx"
```
This will concatenate the string “chooseBlocks” with our selected group (“A” or “B”) and the required file extension (in this case “xlsx”) in order to select the correct order.

Even though our outer loop is now sequential, your inner loop still probably wants to be random (to shuffle the image order within a block).

### 5.4.3 Counterbalancing different subtasks

The above example is useful when we have multiple blocks where the routines we present would be largely similar (i.e. both blocks present an image component), but what about situations where we have totally different tasks we need to counterbalance (e.g. an auditory stroop and an n-back task). The following method is an extension of the logic used in the ‘branchedExp’ demo available in PsychoPy builderview. A working version of the example we will work through here is available [here](#).

So, imagine we have 4 very different tasks. Our flow might look something like this:

Here we have 4 totally different tasks, each with its own loop. Now imagine one participant is presented with these tasks using the order Task1 -> Task2 -> Task3 -> Task 4 (for ease let’s call this group, ‘ABCD’) whilst another is presented with Task2 -> Task3 -> Task4 -> Task 1 (let’s call them group ‘BCDA’).

The loop surrounding each task will look something like this (although here I have stripped the parameters to the bare minimum, you will likely have a conditions file):
Where the number of times that block is repeated (or occurs at all!) is determined by the outer loop (e.g. Task1 nReps = ‘nRepsTask1’, Task2 nReps = ‘nRepsTask2’ and so on).

For our outer loop we will use conditions files that look something like this:

```
<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>nRepsTask1</td>
<td>nRepsTask2</td>
<td>nRepsTask3</td>
<td>nRepsTask4</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
```

Each row corresponds to how many times a subtask routine (or set of routines) will be repeated per iteration of the outer loop. The example conditions file above would be used for a participant in group ‘ABCD’ (on the first iteration Task 1 will repeat once, on the second iteration Task 2 will repeat once and so on).

Just like before we create a field in our experiment settings called group (but let’s say that the group names this time are ‘ABCD’, ‘BCDA’ and so on where the content of the conditions file differs). Finally, we use the following parameters for our outermost loop to select which, preordered, conditions file we are using.

Using this method, we could present several subtasks in a counterbalanced order (without having to create new experiment files for each order!).

What about going online? Well, things are more difficult there, but not impossible let’s talk about Counterbalancing online.
5.5 Components

Routines in the Builder contain any number of components, which typically define the parameters of a stimulus or an input/output device.

The following components are available, as at version 1.65, but further components will be added in the future including Parallel/Serial ports and other visual stimuli (e.g. GeometricStim).

5.5.1 Aperture Component

This component can be used to filter the visual display, as if the subject is looking at it through an opening. Currently only circular apertures are supported. Moreover, only one aperture is enabled at a time. You can’t “double up”: a second aperture takes precedence.

name [string] Everything in a PsychoPy experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

start [float or integer] The time that the aperture should start having its effect. See Defining the onset/duration of components for details.

stop : When the aperture stops having its effect. See Defining the onset/duration of components for details.

Layout

How should the stimulus be laid out? Padding, margins, size, position, etc.

pos [[X,Y]] The position of the centre of the aperture, in the units specified by the stimulus or window.

size [integer] The size controls how big the aperture will be, in pixels, default = 120

spatial units [pix] What units to use (currently only pix).

See also:

API reference for Aperture

5.5.2 Brush Component

The Brush component is a freehand drawing tool.

Properties

Name [string] Everything in a PsychoPy experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

Start [int, float] The time that the stimulus should first appear.

Stop [int, float] Governs the duration for which the stimulus is presented.

Press Button [bool] Should the participant have to press a button to paint, or should it be always on?
Appearance

How should the stimulus look? Colour, borders, etc.

**Brush Size** [int, float] Width of the line drawn by the brush, in pixels

**Opacity** : Vary the transparency, from 0.0 = invisible to 1.0 = opaque

**Brush Color** [color] Colour of the brush

**Brush Color Space** [rgb, dkl, lms, hsv] See Color spaces

See also:
API reference for Brush

5.5.3 Cedrus Button Box Component

This component allows you to connect to a Cedrus Button Box to collect key presses.

*Note that there is a limitation currently that a button box can only be used in a single Routine. Otherwise PsychoPy tries to initialise it twice which raises an error.* As a workaround, you need to insert the start-routine and each-frame code from the button box into a code component for a second routine.

Properties

**Name** [string] Everything in a PsychoPy experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

**Start** : The time that the button box is first read. See Defining the onset/duration of components for details.

**Stop** : Governs the duration for which the button box is first read. See Defining the onset/duration of components for details.

**Force end of Routine** [true/false] If this is checked, the first response will end the routine.

Data

What information to save, how to lay it out and when to save it.

**Allowed keys** [None, or an integer, list, or tuple of integers 0-7] This field lets you specify which buttons (None, or some or all of 0 through 7) to listen to.

**Store** [(choice of: first, last, all, nothing)] Which button events to save in the data file. Events and the response times are saved, with RT being recorded by the button box (not by PsychoPy).

**Store correct** [true/false] If selected, a correctness value will be saved in the data file, based on a match with the given correct answer.

**Discard previous** [true/false] If selected, any previous responses will be ignored (typically this is what you want).
Hardware

Parameters for controlling hardware.

**Device number**: integer  This is only needed if you have multiple Cedrus devices connected and you need to specify which to use.

**Use box timer** [true/false]  Set this to True to use the button box timer for timing information (may give better time resolution)

See also:

API reference for *iolab*

5.5.4 Code Component

The *Code Component* can be used to insert short pieces of python code into your experiments. This might be create a variable that you want for another *Component*, to manipulate images before displaying them, to interact with hardware for which there isn’t yet a pre-packaged component in *PsychoPy* (e.g. writing code to interact with the serial/parallel ports). See *code uses* below.

Be aware that the code for each of the components in your *Routine* are executed in the order they appear on the *Routine* display (from top to bottom). If you want your *Code Component* to alter a variable to be used by another component immediately, then it needs to be above that component in the view. You may want the code not to take effect until next frame however, in which case put it at the bottom of the *Routine*. You can move *Components* up and down the *Routine* by right-clicking on their icons.

Within your code you can use other variables and modules from the script. For example, all routines have a stopwatch-style *Clock*:

```
currentT = trialClock.getTime()
```

To see what other variables you might want to use, and also what terms you need to avoid in your chunks of code, *compile your script* before inserting the code object and take a look at the contents of that script.

Note that this page is concerned with *Code Components* specifically, and not all cases in which you might use python syntax within the Builder. It is also possible to put code into a non-code input field (such as the duration or text of a *Text Component*). The syntax there is slightly different (requiring a `\$` to trigger the special handling, or `\$` to avoid triggering special handling). The syntax to use within a Code Component is always regular python syntax.

Parameters

The parameters of the *Code Component* simply specify the code that will get executed at 5 different points within the experiment. You can use as many or as few of these as you need for any *Code Component*:

  - **Before Experiment**: Things that need to be done just once, like importing a supporting module, which do not need the experiment window to exist yet.
  - **Begin Experiment**: Things that need to be done just once, like initialising a variable for later use, which may need to refer to the experiment window.
  - **Begin Routine**: Certain things might need to be done just once at the start of a *Routine* e.g. at the beginning of each trial you might decide which side a stimulus will appear
  - **Each Frame**: Things that need to updated constantly, throughout the experiment. Note that these will be executed exactly once per video frame (on the order of every 10ms), to give dynamic displays. Static displays do not need to be updated every frame.
  - **End Routine**: At the end of the *Routine* (e.g. the trial) you may need to do additional things, like checking if the participant got the right answer.
End Experiment: Use this for things like saving data to disk, presenting a graph(?), or resetting hardware to its original state.

Example code uses

1. Set a random location for your target stimulus

There are many ways to do this, but you could add the following to the Begin Routine section of a Code Component at the top of your Routine. Then set your stimulus position to be $targetPos and set the correct answer field of a Keyboard Component to be $corrAns (set both of these to update on every repeat of the Routine):

```python
if random() > 0.5:
    targetPos = [-2.0, 0.0] # on the left
    corrAns = 'left'
else:
    targetPos = [+2.0, 0.0] # on the right
    corrAns = 'right'
```

2. Create a patch of noise

As with the above there are many different ways to create noise, but a simple method would be to add the following to the Begin Routine section of a Code Component at the top of your Routine. Then set the image as $noiseTexture:

```python
noiseTexture = random.rand((128, 128)) * 2.0 - 1
```

3. Send a feedback message at the end of the experiment

Make a new routine, and place it at the end of the flow (i.e., the end of the experiment). Create a Code Component with this in the Begin Experiment field:

```python
expClock = core.Clock()
```

and put this in the Begin routine field:

```python
msg = "Thanks for participating - that took %.2f minutes in total" % (expClock.getTime() / 60.0)
```

Next, add a Text Component to the routine, and set the text to $msg. Be sure that the text field's updating is set to “Set every repeat” (and not “Constant”).

4. End a loop early.

Code components can also be used to control the end of a loop. See examples in Recipes:builderTerminateLoops.
What variables are available to use?

The most complete way to find this out for your particular script is to compile it and take a look at what's in there. Below are some options that appear in nearly all scripts. Remember that those variables are Python objects and can have attributes of their own. You can find out about those attributes using:

```
dir(myObject)
```

Common PsychoPy variables:

- **expInfo**: This is a Python Dictionary containing the information from the starting dialog box. e.g. That generally includes the ‘participant’ identifier. You can access that in your experiment using `exp['participant']`
- **t**: the current time (in seconds) measured from the start of this Routine
- **frameN**: the number of /completed/ frames since the start of the Routine (=0 in the first frame)
- **win**: the *Window* that the experiment is using

Your own variables:

- anything you’ve created in a Code Component is available for the rest of the script. (Sometimes you might need to define it at the beginning of the experiment, so that it will be available throughout.)
- the name of any other stimulus or the parameters from your file also exist as variables.
- most Components have a `status` attribute, which is useful to determine whether a stimulus has NOT_STARTED, STARTED or FINISHED. For example, to play a tone at the end of a Movie Component (of unknown duration) you could set start of your tone to have the ‘condition’

```
myMovieName.status==FINISHED
```

Selected contents of the *numpy library* and *numpy.random* are imported by default. The entire numpy library is imported as *np*, so you can use a several hundred maths functions by prepending things with 'np.':

- **random**, **randint**, **normal**, **shuffle** options for creating arrays of random numbers.
- **sin**, **cos**, **tan**, and **pi**: For geometry and trig. By default angles are in radians, if you want the cosine of an angle specified in degrees use `cos(angle*180/pi)`, or use numpy’s conversion functions, `rad2deg(angle)` and `deg2rad(angle)`.
- **linspace()**: Create an array of linearly spaced values.
- **log**, **log10**: The natural and base-10 log functions, respectively. (It is a lowercase-L in log).
- **sum**, **len**: For the sum and length of a list or array. To find an average, it is better to use `average()` (due to the potential for integer division issues with `sum()/len()`).
- **average**, **sqrt**, **std**: For average (mean), square root, and standard deviation, respectively. **Note**: Be sure that the numpy standard deviation formula is the one you want!
- **np.****: Many math-related features are available through the complete numpy libraries, which are available within psychopy builder scripts as ‘np.’. For example, you could use `np.hanning(3)` or `np.random.poisson(10, 10)` in a code component.
5.5.5 Dots (RDK) Component

The Dots Component allows you to present a Random Dot Kinematogram (RDK) to the participant of your study. These are fields of dots that drift in different directions and subjects are typically required to identify the ‘global motion’ of the field.

There are many ways to define the motion of the signal and noise dots. In PsychoPy the way the dots are configured follows Scase, Braddick & Raymond (1996). Although Scase et al (1996) show that the choice of algorithm for your dots actually makes relatively little difference there are some potential gotchas. Think carefully about whether each of these will affect your particular case:

- **limited dot lifetimes**: as your dots drift in one direction they go off the edge of the stimulus and are replaced randomly in the stimulus field. This could lead to a higher density of dots in the direction of motion providing subjects with an alternative cue to direction. Keeping dot lives relatively short prevents this.

- **noiseDots='direction'**: some groups have used noise dots that appear in a random location on each frame (noiseDots='location'). This has the disadvantage that the noise dots not only have a random direction but also a random speed (whereas signal dots have a constant speed and constant direction)

- **signalDots='same'**: on each frame the dots constituting the signal could be the same as on the previous frame or different. If ‘different’, participants could follow a single dot for a long time and calculate its average direction of motion to get the ‘global’ direction, because the dots would sometimes take a random direction and sometimes take the signal direction.

As a result of these, the defaults for PsychoPy are to have signalDots that are from a ‘different’ population, noise dots that have random ‘direction’ and a dot life of 3 frames.

**Parameters**

- **name**: Everything in a PsychoPy experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

- **start**: The time that the stimulus should first appear. See Defining the onset/duration of components for details.

- **stop**: Governs the duration for which the stimulus is presented. See Defining the onset/duration of components for details.

**Appearance**

How should the stimulus look? Colour, borders, etc.

- **dot color**: See Color spaces

- **dot color space**: [rgb, dkl or lms] See Color spaces

- **opacity**: Vary the transparency, from 0.0 = invisible to 1.0 = opaque

**Layout**

How should the stimulus be laid out? Padding, margins, size, position, etc.

- **fieldPos** [(x,y) or [x,y]] specifying the location of the centre of the stimulus.

- **fieldSize** [a single value, specifying the diameter of the field] Sizes can be negative and can extend beyond the window.

- **fieldShape**: Defines the shape of the field in which the dots appear. For a circular field the nDots represents the average number of dots per frame, but on each frame this may vary a little.
**spatial units** [None, ‘norm’, ‘cm’, ‘deg’ or ‘pix’] If None then the current units of the Window will be used. See Units for the window and stimuli for explanation of other options.

**Dots**

Parameters unique to the Dots component

- **coherence** [float] fraction moving in the signal direction on any one frame
- **nDots** [int] number of dots to be generated
- **dotSize** Always specified in pixels
- **dotLife** [int] Number of frames each dot lives for (-1=infinite)
- **dir** [float (degrees)] Direction of the signal dots
- **speed** [float] Speed of the dots (in units per frame)

**signalDots** : If ‘same’ then the signal and noise dots are constant. If different then the choice of which is signal and which is noise gets randomised on each frame. This corresponds to Scase et al's (1996) categories of RDK.

**noiseDots** [‘direction’, ‘position’ or ‘walk’] Determines the behaviour of the noise dots, taken directly from Scase et al's (1996) categories. For ‘position’, noise dots take a random position every frame. For ‘direction’ noise dots follow a random, but constant direction. For ‘walk’ noise dots vary their direction every frame, but keep a constant speed.

**dot refresh rule** [repeat, none] When should the sample of dots be refreshed?

See also:

API reference for DotStim

---

### 5.5.6 Emotiv Marking Component

The Emotiv Marking component causes Psychopy to send a marker to the EEG datastream at the time that the stimuli are presented.

For the Emotiv Marking component to work an emotiv_recording component should have already been added to the experiment.

By default markers with labels and values can be added. A time interval can be specified by sending a stop marker. If the Marker intervals overlap it is important that the labels are unique. Additionally the length of the interval must be greater than 0.2 seconds. If you need higher speeds than this, it is best to record the times of your markers manually and compare them to the times in the raw EEG data.

**Parameters**

- **Name** [string] Everything in a PsychoPy experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).
- **Start** : The time to send the marker to the EEG datastream
- **Stop Marker**: If selected the stop marker will be sent as specified by the Stop parameter. If no stop marker is sent then the marker will be an “instance” marker and will indicate a point in time. If a stop marker is sent the marker will be an “interval” marker and have a startDateTime and endDateTime associated with it.
- **Stop** : Governs the duration for which the stimulus is presented. See Defining the onset/duration of components for details.
- **marker label** [string] The label assigned to this marker
5.5.7 Emotiv Record Component

The emotiv_record component causes Psychopy to connect to the headset so that markers can be sent to the datastream. The emotiv_record component should be added ONCE before any stimuli have been presented at the top of first trial of the experiment.

We recommend that you use the EmotivApp and or EmotivPro software to establish that the headset is connected and the quality of the signals are good before running the experiment with Psychopy.

If you do want PschoPy to record the data into a gzipped csv file you need to set an environment variable CORTEX_DATA=1. Otherwise we recommend viewing the eeg data in EmotivPro from which it can be exported as a csv or edf file.

Getting Started

Before you can connect Psychopy to Emotiv hardware, you need to register your AppId on the Emotiv website (https://emotiv.com) Goto My Account > Cortex Apps. There you will get a client_id and a client_secret that you need to copy into a file called .emotivcreds in your home directory. One line should have “client_id” (without the quotes) then a space and then the client_id, another line should have “client_secret” (without the quotes and then as space and then the client secret. A line beginning with a hash will be ignored. eg

—begin file—
# client_id and client_secret for Emotiv application
client_id abcd1234…
client_secret wxyz78910…
—end file—

Parameters

**Name**  [string] Everything in a PsychoPy experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

**Start** : Set this to 0

**Stop** : Set this to 1 seconds

Setting these values just allows the routine to finish
5.5.8 Form Component

The Form component enables Psychopy to be used as a questionnaire tool, where participants can be presented with a series of questions requiring responses. Form items, defined as questions and response pairs, are presented simultaneously onscreen with a scrollable viewing window.

Properties

Name [string] Everything in a PsychoPy experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

Start [int, float] The time that the stimulus should first appear.

Stop [int, float] Governs the duration for which the stimulus is presented.

Items [List of dicts or csv / xlsx file]

A list of dicts or csv file should have the following key, value pairs / column headers:

- **index** The item index as a number
- **itemText** The item question string
- **itemWidth** The question width between 0 : 1
- **type** The type of rating e.g., ‘radio’, ‘rating’, ‘slider’
- **responseWidth** The question width between 0 : 1
- **options** A sequence of tick labels for options e.g., yes, no
- **layout** Response object layout e.g., ‘horiz’ or ‘vert’
- **itemColor** The question text font color
- **responseColor** The response object color

Missing column headers will be replaced by default entries, with the exception of **itemText** and **type**, which are required. The default entries are:

- **index** 0 (increments for each item)
- **itemWidth** 0.7
- **responseWidth** 0.3
- **options** Yes, No
- **layout** horiz
- **itemColor** from style
- **responseColor** from style

Data format [menu] Choose whether to store items data by column or row in your datafile.

randomize [bool] Randomize order of Form elements
Appearance

How should the stimulus look? Color, borders, etc.

**style**  [light, dark] The overall colors to be used for items in your forms - this sets foreground, background and emphasis colors to pre-defined values.

**border color**  [color] See Color spaces

**border color space**  [rgb, dkl, lms, hsv] See Color spaces

**fill color**  [color] See Color spaces

**fill color space**  [rgb, dkl, lms, hsv] See Color spaces

**opacity** : Vary the transparency, from 0.0 = invisible to 1.0 = opaque

Layout

How should the stimulus be laid out? Padding, margins, size, position, etc.

**Size**  [[X,Y]] Size of the stimulus, to be specified in ‘height’ units.

**Pos**  [[X,Y]] The position of the centre of the stimulus, to be specified in ‘height’ units.

**Item padding**  [float] Space or padding between Form elements (i.e., question and response text), to be specified in ‘height’ units.

Formatting

Formatting text

**Text height**  [float] Text height of the Form elements (i.e., question and response text).

See also:

API reference for Form

5.5.9 Grating Component

The Grating stimulus allows a texture to be wrapped/cycled in 2 dimensions, optionally in conjunction with a mask (e.g. Gaussian window). The texture can be a bitmap image from a variety of standard file formats, or a synthetic texture such as a sinusoidal grating. The mask can also be derived from either an image, or mathematical form such as a Gaussian.

When using gratings, if you want to use the spatial frequency setting then create just a single cycle of your texture and allow PsychoPy to handle the repetition of that texture (do not create the cycles you’re expecting within the texture).

Gratings can have their position, orientation, size and other settings manipulated on a frame-by-frame basis. There is a performance advantage (in terms of milliseconds) to using images which are square and powers of two (32, 64, 128, etc.), however this is slight and would not be noticed in the majority of experiments.
Parameters

Name  [string] Everything in a PsychoPy experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

Start : The time that the stimulus should first appear. See Defining the onset/duration of components for details.

Stop : Governs the duration for which the stimulus is presented. See Defining the onset/duration of components for details.

Appearance

How should the stimulus look? Colour, borders, etc.

blend mode  [average, add] How should colours blend when overlayed onto something? Should colours be averaged, or added?

foreground color : See Color spaces

foreground color space  [rgb, dkl or lms] See Color spaces

Opacity  [0-1] Can be used to create semi-transparent gratings

Layout

How should the stimulus be laid out? Padding, margins, size, position, etc.

Orientation  [degrees] The orientation of the entire patch (texture and mask) in degrees.

Position  [[X,Y]] The position of the centre of the stimulus, in the units specified by the stimulus or window

Size  [[sizex, sizey] or a single value (applied to x and y)] The size of the stimulus in the given units of the stimulus/window. If the mask is a Gaussian then the size refers to width at 3 standard deviations on either side of the mean (i.e. sd=size/6)

Units  [deg, cm, pix, norm, or inherit from window] See Units for the window and stimuli

Texture

Control how the stimulus handles textures.

Texture: a filename, a standard name (sin, sqr) or a variable giving a numpy array  This specifies the image that will be used as the texture for the visual patch. The image can be repeated on the patch (in either x or y or both) by setting the spatial frequency to be high (or can be stretched so that only a subset of the image appears by setting the spatial frequency to be low). Filenames can be relative or absolute paths and can refer to most image formats (e.g. tif, jpg, bmp, png, etc.). If this is set to none, the patch will be a flat colour.

Mask  [a filename, a standard name (gauss, circle, raisedCos) or a numpy array of dimensions NxNx1] The mask can define the shape (e.g. circle will make the patch circular) or something which overlays the patch e.g. noise.

Interpolate : If linear is selected then linear interpolation will be applied when the image is rescaled to the appropriate size for the screen. Nearest will use a nearest-neighbour rule.

Phase  [single float or pair of values [X,Y]] The position of the texture within the mask, in both X and Y. If a single value is given it will be applied to both dimensions. The phase has units of cycles (rather than degrees or radians), wrapping at 1. As a result, setting the phase to 0,1,2… is equivalent, causing the texture to be centered on the mask. A phase of 0.25 will cause the image to shift by half a cycle (equivalent to pi radians). The advantage of this is that if you set the phase according to time it is automatically in Hz.
Spatial Frequency  
[[SFx, SFy] or a single value (applied to x and y)] The spatial frequency of the texture on the patch. The units are dependent on the specified units for the stimulus/window; if the units are deg then the SF units will be cycles/deg, if units are norm then the SF units will be cycles per stimulus. If this is set to none then only one cycle will be displayed.

Texture Resolution  
[an integer (power of two)] Defines the size of the resolution of the texture for standard textures such as sin, sqr etc. For most cases a value of 256 pixels will suffice, but if stimuli are going to be very small then a lower resolution will use less memory.

See also:
API reference for GratingStim

5.5.10 Image Component

The Image stimulus allows an image to be presented, which can be a bitmap image from a variety of standard file formats, with an optional transparency mask that can effectively control the shape of the image. The mask can also be derived from an image file, or mathematical form such as a Gaussian.

It is a really good idea to get your image in roughly the size (in pixels) that it will appear on screen to save memory. If you leave the resolution at 12 megapixel camera, as taken from your camera, but then present it on a standard screen at 1680x1050 (=1.6 megapixels) then PsychoPy and your graphics card have to do an awful lot of unnecessary work. There is a performance advantage (in terms of milliseconds) to using images which are square and powers of two (32, 64, 128, etc.), but this is slight and would not be noticed in the majority of experiments.

Images can have their position, orientation, size and other settings manipulated on a frame-by-frame basis.

Parameters

Name  [string] Everything in a PsychoPy experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

Start  :  The time that the stimulus should first appear. See Defining the onset/duration of components for details.

Stop  :  Governs the duration for which the stimulus is presented. See Defining the onset/duration of components for details.

Image  [a filename or a standard name (sin, sqr)] Filenames can be relative or absolute paths and can refer to most image formats (e.g. tif, jpg, bmp, png, etc.). If this is set to none, the patch will be a flat colour.

Appearance

How should the stimulus look? Colour, borders, etc.

opacity  [value from 0 to 1] If opacity is reduced then the underlying images/stimuli will show through

foreground color  [Colors can be applied to luminance-only images (not to rgb images)] See Color spaces

foreground color space  [to be used if a color is supplied] See Color spaces
**Layout**

How should the stimulus be laid out? Padding, margins, size, position, etc.

**Position**  [[X,Y]] The position of the centre of the stimulus, in the units specified by the stimulus or window

**Size**  [[sizex, sizey] or a single value (applied to x and y)] The size of the stimulus in the given units of the stimulus/window. If the mask is a Gaussian then the size refers to width at 3 standard deviations on either side of the mean (i.e. sd=size/6) Set this to be blank to get the image in its native size.

**Orientation**  [degrees] The orientation of the entire patch (texture and mask) in degrees.

**Units**  [deg, cm, pix, norm, or inherit from window] See *Units for the window and stimuli*

**flip horizontally**  [bool] Flip the image along the horizontal axis

**flip vertically**  [bool] Flip the image along the vertical axis

**spatial units**  [deg, cm, pix, norm, or inherit from window] See *Units for the window and stimuli*

**Texture**

Control how the stimulus handles textures.

**Mask**  [a filename, a standard name (gauss, circle, raisedCos) or a numpy array of dimensions NxNx1] The mask can define the shape (e.g. circle will make the patch circular) or something which overlays the patch e.g. noise.

**Interpolate** : If *linear* is selected then linear interpolation will be applied when the image is rescaled to the appropriate size for the screen. *Nearest* will use a nearest-neighbour rule.

**Texture Resolution:** This is only needed if you use a synthetic texture (e.g. sinusoidal grating) as the image.

See also:

API reference for *ImageStim*

### 5.5.11 ioLab Systems buttonbox Component

A button box is a hardware device that is used to collect participant responses with high temporal precision, ideally with true ms accuracy.

Both the response (which button was pressed) and time taken to make it are returned. The time taken is determined by a clock on the device itself. This is what makes it capable (in theory) of high precision timing.

Check the log file to see how long it takes for PsychoPy to reset the button box’s internal clock. If this takes a while, then the RT timing values are not likely to be high precision. It might be possible for you to obtain a correction factor for your computer + button box set up, if the timing delay is highly reliable.

The ioLabs button box also has a built-in voice-key, but PsychoPy does not have an interface for it. Use a microphone component instead.
Properties

name  [string] Everything in a PsychoPy experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

start : The time that the stimulus should first appear. See Defining the onset/duration of components for details.

stop : The duration for which the stimulus is presented. See Defining the onset/duration of components for details.

Force end of Routine  [checkbox] If this is checked, the first response will end the routine.

Data

What information to save, how to lay it out and when to save it.

Active buttons  [None, or an integer, list, or tuple of integers 0-7] The ioLabs box lets you specify a set of active buttons. Responses on non-active buttons are ignored by the box, and never sent to PsychoPy. This field lets you specify which buttons (None, or some or all of 0 through 7).

Store  [(choice of: first, last, all, nothing)] Which button events to save in the data file. Events and the response times are saved, with RT being recorded by the button box (not by PsychoPy).

Store correct  [checkbox] If selected, a correctness value will be saved in the data file, based on a match with the given correct answer.

Discard previous  [checkbox] If selected, any previous responses will be ignored (typically this is what you want).

Hardware

Parameters for controlling hardware.

Lights off  [checkbox] If selected, all lights will be turned off at the end of each routine.

Lights : If selected, the lights above the active buttons will be turned on.

Using code components, it is possible to turn on and off specific lights within a trial. See the API for iolab.

See also:
API reference for iolab

5.5.12 JoyButtons Component

The JoyButtons component can be used to collect gamepad/joystick button responses from a participant.

By not storing the button number pressed and checking the forceEndTrial box it can be used simply to end a Routine If no gamepad/joystick is installed the keyboard can be used to simulate button presses by pressing ‘ctrl’ + ‘alt’ + digit(0-9).
Parameters

Name  [string] Everything in a PsychoPy experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

Start  [float or integer] The time that joyButtons should first get checked. See Defining the onset/duration of components for details.

Stop  [float or integer] When joyButtons should no longer get checked. See Defining the onset/duration of components for details.

Force end routine : If this box is checked then the Routine will end as soon as one of the allowed buttons is pressed.

Data

What information to save, how to lay it out and when to save it.

Allowed buttons : A list of allowed buttons can be specified here, e.g. [0,1,2,3], or the name of a variable holding such a list. If this box is left blank then any button that is pressed will be read. Only allowed buttons count as having been pressed; any other button will not be stored and will not force the end of the Routine. Note that button numbers (0, 1, 2, 3, …), should be separated by commas.

Store : Which button press, if any, should be stored; the first to be pressed, the last to be pressed or all that have been pressed. If the button press is to force the end of the trial then this setting is unlikely to be necessary, unless two buttons happen to be pressed in the same video frame. The response time will also be stored if a button press is recorded. This time will be taken from the start of joyButtons checking (e.g. if the joyButtons was initiated 2 seconds into the trial and a button was pressed 3.2s into the trials the response time will be recorded as 1.2s).

Store correct : Check this box if you wish to store whether or not this button press was correct. If so then fill in the next box that defines what would constitute a correct answer e.g. 1 or $corrAns$ (note this should not be in inverted commas). This is given as Python code that should return True (1) or False (0). Often this correct answer will be defined in the settings of the Loops.

Hardware

Parameters for controlling hardware.

Device number  [integer] Which gamepad/joystick device number to use. The first device found is numbered 0.

5.5.13 Joystick Component

The Joystick component can be used to collect responses from a participant. The coordinates of the joystick location are given in the same coordinates as the Window, with (0,0) in the centre. Coordinates are correctly scaled for 'norm' and 'height' units. User defined scaling can be set by updating joystick.xFactor and joystick.yFactor to the desired values. Joystick.device.getX() and joystick.device.getY() always return 'norm' units. Joystick.getX() and joystick.getY() are scaled by xFactor or yFactor.

No cursor is drawn to represent the joystick current position, but this is easily provided by updating the position of a partially transparent '.png' image on each screen frame using the joystick coordinates: joystick.getX() and joystick.getY(). To ensure that the cursor image is drawn on top of other images it should be the last image in the trial.

Joystick Emulation If no joystick device is found, the mouse and keyboard are used to emulate a joystick device. Joystick position corresponds to mouse position and mouse buttons correspond to joystick buttons (0,1,2). Other buttons can be simulated with key chords: ‘ctrl’ + ‘alt’ + digit(0..9).
Scenarios

This can be used in various ways. Here are some scenarios (email the list if you have other uses for your joystick):

Use the joystick to record the location of a button press

**Use the joystick to control stimulus parameters** Imagine you want to use your joystick to make your ‘patch’ bigger or smaller and save the final size. Call your `joystickComponent` ‘joystick’, set it to save its state at the end of the trial and set the button press to end the Routine. Then for the size setting of your Patch stimulus insert `$joystick.getX()` to use the x position of the joystick to control the size or `$joystick.getY()` to use the y position.

Tracking the entire path of the joystick during a period

### Parameters Basic

**Name** [string] Everything in a PsychoPy experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

**start** : The time that the joystick should first be checked. See *Defining the onset/duration of components* for details.

**stop** : When the joystick is no longer checked. See *Defining the onset/duration of components* for details.

**Force End Routine on Press** If this box is checked then the Routine will end as soon as one of the joystick buttons is pressed.

### Data

What information to save, how to lay it out and when to save it.

**Save Joystick State** How often do you need to save the state of the joystick? Every time the subject presses a joystick button, at the end of the trial, or every single frame? Note that the text output for cases where you store the joystick data repeatedly per trial (e.g. every press or every frame) is likely to be very hard to interpret, so you may then need to analyse your data using the psydat file (with python code) instead. Hopefully in future releases the output of the text file will be improved.

**Time Relative To** Whenever the joystick state is saved (e.g. on button press or at end of trial) a time is saved too. Do you want this time to be relative to start of the Routine, or the start of the whole experiment?

**Clickable Stimulus** A comma-separated list of your stimulus names that can be “clicked” by the participant. e.g. target, foil.

**Store params for clicked** The params (e.g. name, text), for which you want to store the current value, for the stimulus that was “clicked” by the joystick. Make sure that all the clickable objects have all these params.

**Allowed Buttons** Joystick buttons accepted for input (blank for any) numbers separated by ‘commas’.

### Hardware

Parameters for controlling hardware.

**Device Number** If you have multiple joystick/gamepad devices which one do you want (0, 1, 2, …).
5.5.14 Keyboard Component

The Keyboard component can be used to collect responses from a participant.

By not storing the key press and checking the forceEndTrial box it can be used simply to end a Routine

Parameters

name [string] Everything in a PsychoPy experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

start [float or integer] The time that the keyboard should first get checked. See Defining the onset/duration of components for details.

stop : When the keyboard is no longer checked. See Defining the onset/duration of components for details.

force end routine If this box is checked then the Routine will end as soon as one of the allowed keys is pressed.

Data

What information to save, how to lay it out and when to save it.

allowed keys [list] A list of allowed keys can be specified here, e.g. [‘m’, ‘z’, ‘1’, ‘2’], or the name of a variable holding such a list. If this box is left blank then any key that is pressed will be read. Only allowed keys count as having been pressed; any other key will not be stored and will not force the end of the Routine. Note that key names (even for number keys) should be given in single quotes, separated by commas. Cursor control keys can be accessed with ‘up’, ‘down’, and so on; the space bar is ‘space’. To find other special keys, run the Coder Input demo, “what_key.py”, press the key, and check the Coder output window.

store [last key, first key, all keys, nothing] Which key press, if any, should be stored; the first to be pressed, the last to be pressed or all that have been pressed. If the key press is to force the end of the trial then this setting is unlikely to be necessary, unless two keys happen to be pressed in the same video frame. The response time will also be stored if a keypress is recorded. This time will be taken from the start of keyboard checking (e.g. if the keyboard was initiated 2 seconds into the trial and a key was pressed 3.2s into the trials the response time will be recorded as 1.2s).

store correct [bool] Check this box if you wish to store whether or not this key press was correct. If so then fill in the next box that defines what would constitute a correct answer e.g. left, 1 or $corrAns$ (note this should not be in inverted commas). This is given as Python code that should return True (1) or False (0). Often this correct answer will be defined in the settings of the Loops.

discard previous [bool] Check this box to ensure that only key presses that occur during this keyboard checking period are used. If this box is not checked a keyboard press that has occurred before the start of the checking period will be interpreted as the first keyboard press. For most experiments this box should be checked.

See also:

API reference for psychopy.event
5.5.15 Microphone Component

Please note: This is a new component, and is subject to change.

The microphone component provides a way to record sound during an experiment. To do so, specify the starting time relative to the start of the routine (see `start` below) and a stop time (= duration in seconds). A blank duration evaluates to recording for 0.000s.

The resulting sound files are saved in .wav format (at 48000 Hz, 16 bit), one file per recording. The files appear in a new folder within the data directory (the subdirectory name ends in `_wav`). The file names include the unix (epoch) time of the onset of the recording with milliseconds, e.g., `mic-1346437545.759.wav`.

It is possible to stop a recording that is in progress by using a code component. Every frame, check for a condition (such as key ‘q’, or a mouse click), and call the `.stop()` method of the microphone component. The recording will end at that point and be saved. For example, if `mic` is the name of your microphone component, then in the code component, do this on Each frame:

```python
if event.getKeys(['q']):
    mic.stop()
```

**Parameters**

- **name**  [string] Everything in a PsychoPy experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).
- **start**  [float or integer] The time that the stimulus should first play. See *Defining the onset/duration of components* for details.
- **stop (duration):** The length of time (sec) to record for. An expected duration can be given for visualisation purposes. See *Defining the onset/duration of components* for details; note that only seconds are allowed.
- **channel**  [int] What audio channel is mic connected through?
- **stereo**  [bool] Whether to use mono or stereo audio

See also:

API reference for *AdvAudioCapture*

5.5.16 Mouse Component

The Mouse component can be used to collect responses from a participant. The coordinates of the mouse location are given in the same coordinates as the Window, with (0,0) in the centre.

**Scenarios**

This can be used in various ways. Here are some scenarios (email the list if you have other uses for your mouse):

- Use the mouse to record the location of a button press

Use the mouse to control stimulus parameters

Imagine you want to use your mouse to make your ‘patch’ bigger or smaller and save the final size. Call your `mouse` ‘mouse’, set it to save its state at the end of the trial and set the button press to end the Routine. Then for the size setting of your Patch stimulus insert `$mouse.getPos()[0]` to use the x position of the mouse to control the size or `$mouse.getPos()[1]` to use the y position.

Tracking the entire path of the mouse during a period
Parameters

name [string] Everything in a PsychoPy experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

start : The time that the mouse should first be checked. See Defining the onset/duration of components for details.

stop : When the mouse is no longer checked. See Defining the onset/duration of components for details.

Force End Routine on Press If this box is checked then the Routine will end as soon as one of the mouse buttons is pressed.

Data

What information to save, how to lay it out and when to save it.

save mouse state How often do you need to save the state of the mouse? Every time the subject presses a mouse button, at the end of the trial, or every single frame? Note that the text output for cases where you store the mouse data repeatedly per trial (e.g. every press or every frame) is likely to be very hard to interpret, so you may then need to analyse your data using the psydat file (with python code) instead. Hopefully in future releases the output of the text file will be improved.

time relative to Whenever the mouse state is saved (e.g. on button press or at end of trial) a time is saved too. Do you want this time to be relative to start of the Routine, or the start of the whole experiment?

new clicks only [bool] Store only new clicks

clickable stimuli [list] List of stimulus names within the same routine which can be clicked on

store params for clicked [list] List of parameter names to store from stimuli which are clicked on

See also:

API reference for Mouse

5.5.17 Movie Component

The Movie component allows movie files to be played from a variety of formats (e.g. mpeg, avi, mov).

The movie can be positioned, rotated, flipped and stretched to any size on the screen (using the Units for the window and stimuli given).

Parameters

name [string] Everything in a PsychoPy experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

start : The time that the stimulus should first appear. See Defining the onset/duration of components for details.

stop : Governs the duration for which the stimulus is presented (if you want to cut a movie short). Usually you can leave this blank and insert the Expected duration just for visualisation purposes. See Defining the onset/duration of components for details.

movie [string] The filename of the movie, including the path. The path can be absolute or relative to the location of the experiment (.psyexp) file.

forceEndRoutine : If checked, when the movie finishes, the routine will be ended.
Appearance

How should the stimulus look? Colour, borders, etc.

**opacity** [float] Vary the transparency, from 0.0 = invisible to 1.0 = opaque

Layout

How should the stimulus be laid out? Padding, margins, size, position, etc.

**ori** [degrees] Movies can be rotated in real-time too! This specifies the orientation of the movie in degrees.

**pos** [[X,Y]] The position of the centre of the stimulus, in the units specified by the stimulus or window

**size** [[sizex, sizey] or a single value (applied to both x and y)] The size of the stimulus in the given units of the stimulus/window.

**spatial units** [deg, cm, pix, norm, or inherit from window] See Units for the window and stimuli

Playback

How should stimulus play? Speed, volume, etc.

**backend** [moviepy, avbin, opencv] What Python packages should be used to play the movie?

**no audio** [bool] Tick to mute audio

**loop playback** [bool] Should video loop on completion?

See also:

API reference for MovieStim

5.5.18 Parallel Port Out Component

This component allows you to send triggers to a parallel port or to a LabJack device.

An example usage would be in EEG experiments to set the port to 0 when no stimuli are present and then set it to an identifier value for each stimulus synchronised to the start/stop of that stimulus. In that case you might set the Start data to be $ID$ (with ID being a column in your conditions file) and set the Stop Data to be 0.

Properties

**Name** [string] Everything in a PsychoPy experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

**Start** : The time that the stimulus should first appear. See Defining the onset/duration of components for details.

**Stop** : Governs the duration for which the stimulus is presented. See Defining the onset/duration of components for details.
Data

What information to save, how to lay it out and when to save it.

**Start data** [0-255] When the start time/condition occurs this value will be sent to the parallel port. The value is given as a byte (a value from 0-255) controlling the 8 data pins of the parallel port.

**Stop data** [0-255] As with start data but sent at the end of the period.

**Sync to screen** [boolean] If true then the parallel port will be sent synchronised to the next screen refresh, which is ideal if it should indicate the onset of a visual stimulus. If set to False then the data will be set on the parallel port immediately.

Hardware

Parameters for controlling hardware.

**Port address** [select the appropriate option] You need to know the address of the parallel port you wish to write to. The options that appear in this drop-down list are determined by the application preferences. You can add your particular port there if you prefer.

See also:

API reference for [iolab](https://psychopy.org/api/iolab.html)

### 5.5.19 Patch (image) Component

The Patch stimulus allows images to be presented in a variety of forms on the screen. It allows the combination of an image, which can be a bitmap image from a variety of standard file formats, or a synthetic repeating texture such as a sinusoidal grating. A transparency mask can also be control the shape of the image, and this can also be derived from either a second image, or mathematical form such as a Gaussian.

Patches can have their position, orientation, size and other settings manipulated on a frame-by-frame basis. There is a performance advantage (in terms of milliseconds) to using images which are square and powers of two (32, 64, 128, etc.), however this is slight and would not be noticed in the majority of experiments.

Parameters

**name** [string] Everything in a PsychoPy experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

**start** : The time that the stimulus should first appear. See *Defining the onset/duration of components* for details.

**stop** : Governs the duration for which the stimulus is presented. See *Defining the onset/duration of components* for details.

**image** [a filename, a standard name (‘sin’, ‘sqr’) or a numpy array of dimensions NxNx1 or NxNx3] This specifies the image that will be used as the *texture* for the visual patch. The image can be repeated on the patch (in either x or y or both) by setting the spatial frequency to be high (or can be stretched so that only a subset of the image appears by setting the spatial frequency to be low). Filenames can be relative or absolute paths and can refer to most image formats (e.g. tif, jpg, bmp, png, etc.). If this is set to none, the patch will be a flat colour.

**mask** [a filename, a standard name (‘gauss’, ‘circle’) or a numpy array of dimensions NxNx1] The mask can define the shape (e.g. circle will make the patch circular) or something which overlays the patch e.g. noise.

**ori** [degrees] The orientation of the entire patch (texture and mask) in degrees.

**pos** [[X,Y]] The position of the centre of the stimulus, in the units specified by the stimulus or window.
size  [[sizex, sizey] or a single value (applied to x and y)] The size of the stimulus in the given units of the stimulus/window. 
If the mask is a Gaussian then the size refers to width at 3 standard deviations on either side of the mean (i.e. 
sd=size/6)  

units  [deg, cm, pix, norm, or inherit from window] See *Units for the window and stimuli*  

**Advanced Settings**  

**colour** : See *Color spaces*  

**colour space**  [rgb, dkl or lms] See *Color spaces*  

**SF**  [[SFx, SFy] or a single value (applied to x and y)] The spatial frequency of the texture on the patch. The units are 
dependent on the specified units for the stimulus/window; if the units are *deg* then the SF units will be *cycles/deg*, 
if units are *norm* then the SF units will be cycles per stimulus. If this is set to none then only one cycle will be 
displayed.  

**phase**  [single float or pair of values [X,Y]] The position of the texture within the mask, in both X and Y. If a single 
value is given it will be applied to both dimensions. The phase has units of cycles (rather than degrees or radians), 
wrapping at 1. As a result, setting the phase to 0,1,2… is equivalent, causing the texture to be centered on the 
mask. A phase of 0.25 will cause the image to shift by half a cycle (equivalent to pi radians). The advantage of this 
is that if you set the phase according to time it is automatically in Hz.  

**Texture Resolution**  [an integer (power of two)] Defines the size of the resolution of the texture for standard textures 
such as sin, sqr etc. For most cases a value of 256 pixels will suffice, but if stimuli are going to be very small then 
a lower resolution will use less memory.  

**interpolate**  : If *linear* is selected then linear interpolation will be applied when the image is rescaled to the appropriate 
size for the screen. *Nearest* will use a nearest-neighbour rule.  

See also:  

API reference for *PatchStim*  

**5.5.20 Polygon (shape) Component**  

(added in version 1.78.00)  

The Polygon stimulus allows you to present a wide range of regular geometric shapes. The basic control comes from setting the number of vertices:  

- 2 vertices give a line  
- 3 give a triangle  
- 4 give a rectangle etc.  
- a large number will approximate a circle/ellipse  

The size parameter takes two values. For a line only the first is used (then use ori to specify the orientation). For triangles 
and rectangles the size specifies the height and width as expected. Note that for pentagons upwards, however, the size 
determines the width/height of the ellipse on which the vertices will fall, rather than the width/height of the vertices 
themselves (slightly smaller typically).
**Parameters**

**name** [string] Everything in a PsychoPy experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

**start** : The time that the stimulus should first appear. See *Defining the onset/duration of components* for details.

**stop** : Governs the duration for which the stimulus is presented. See *Defining the onset/duration of components* for details.

**shape** [line, triangle, rectangle, cross, star, regular polygon] What shape the stimulus is

**num vertices** [int] The number of vertices for your shape (2 gives a line, 3 gives a triangle,… a large number results in a circle/ellipse). It is not (currently) possible to vary the number of vertices dynamically.

**Appearance**

How should the stimulus look? Colour, borders, etc.

**fill color** [color] See *Color spaces*

**fill color space** [rgb, dkl, lms, hsv] See *Color spaces*

**border color** [color] See *Color spaces*

**border color space** [rgb, dkl, lms, hsv] See *Color spaces*

**line width** [int | float] How wide should the line be? Width is specified in chosen spatial units, see *Units for the window and stimuli*

**opacity** : Vary the transparency, from 0.0 = invisible to 1.0 = opaque

**Layout**

How should the stimulus be laid out? Padding, margins, size, position, etc.

**ori** [degrees] The orientation of the entire patch (texture and mask) in degrees.

**pos** [[X,Y]] The position of the centre of the stimulus, in the units specified by the stimulus or window

**size** [(width, height)] Size of the stimulus on screen

**spatial units** [deg, cm, pix, norm, or inherit from window] See *Units for the window and stimuli*

**Texture**

Control how the stimulus handles textures.

**interpolate** [linear, nearest] Should textures be interpolated?

**units** [deg, cm, pix, norm, or inherit from window] See *Units for the window and stimuli*

See also:

API reference for *Polygon* API reference for *Rect* API reference for *ShapeStim* #for arbitrary vertices

---

5.5. Components
5.5.21 Pump Component

This component allows you to deliver liquid stimuli using a Cetoni neMESYS syringe pump.

Please specify the name of the pump configuration to use in the PsychoPy preferences under Hardware / Qmix pump configuration. See the readme file of the pyqmix project for details on how to set up your computer and create the configuration file.

Properties

Name [string] Everything in a PsychoPy experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

Start : The time that the stimulus should first appear.

Stop : Governs the duration for which the stimulus is presented.

Sync to screen [bool] Whether to synchronize the pump operations (starting, stopping) to the screen refresh. This ensures better synchronization with visual stimuli.

Hardware

Parameters for controlling hardware.

Pump index [int] The index of the pump: The first pump's index is 0, the second pump's index is 1, etc. You may insert the name of a variable here to adjust this value dynamically.

Syringe type [select the appropriate option] Currently, 25 mL and 50 mL glass syringes are supported. This setting ensures that the pump will operate at the correct flow rate.

Pump action [aspirate or dispense] Whether to fill (aspirate) or to empty (dispense) the syringe.

Flow rate [float] The flow rate in the selected flow rate units.

Flow rate unit [mL/s or mL/min] The unit in which the flow rate values are supplied.

Switch valve after dosing [bool] Whether to switch the valve position after the pump operation has finished. This can be used to ensure a sharp(er) stimulus offset.

5.5.22 RatingScale Component

The Rating Scale Component is in the process of deprecation, if possible we recommend using the newer Slider component instead.

A rating scale is used to collect a numeric rating or a choice from a few alternatives, via the mouse, the keyboard, or both. Both the response and time taken to make it are returned.

A given routine might involve an image (patch component), along with a rating scale to collect the response. A routine from a personality questionnaire could have text plus a rating scale.

Three common usage styles are enabled on the first settings page: ‘visual analog scale’: the subject uses the mouse to position a marker on an unmarked line

‘category choices’: choose among verbal labels (categories, e.g., “True, False” or “Yes, No, Not sure”)

‘scale description’: used for numeric choices, e.g., 1 to 7 rating

Complete control over the display options is available as an advanced setting, ‘customize_everything’.
Properties

name [string] Everything in a PsychoPy experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

start : The time that the stimulus should first appear. See Defining the onset/duration of components for details.

stop : The duration for which the stimulus is presented. See Defining the onset/duration of components for details.

category choices [string] Instead of a numeric scale, you can present the subject with words or phrases to choose from. Enter all the words as a string. (Probably more than 6 or so will not look so great on the screen.) Spaces are assumed to separate the words. If there are any commas, the string will be interpreted as a list of words or phrases (possibly including spaces) that are separated by commas.

scaleDescription [string] Brief instructions, reminding the subject how to interpret the numerical scale, default = “1 = not at all … extremely = 7”

forceEndRoutine [bool] If checked, when the subject makes a rating the routine will be ended.

Layout

How should the stimulus be laid out? Padding, margins, size, position, etc.

size [float] The size controls how big the scale will appear on the screen. (Same as “displaySizeFactor”.) Larger than 1 will be larger than the default, smaller than 1 will be smaller than the default.

pos [[X,Y]] The position of the centre of the stimulus, in the units specified by the stimulus or window. Default is centered left-right, and somewhat lower than the vertical center (0, -0.4).

Data

What information to save, how to lay it out and when to save it.

visualAnalogScale [checkbox] If this is checked, a line with no tick marks will be presented using the ‘glow’ marker, and will return a rating from 0.00 to 1.00 (quasi-continuous). This is intended to bias people away from thinking in terms of numbers, and focus more on the visual bar when making their rating. This supersedes either choices or scaleDescription.

low [str] The lowest number (bottom end of the scale), default = 1. If it’s not an integer, it will be converted to lowAnchorText (see Advanced).

high [str] The highest number (top end of the scale), default = 7. If it’s not an integer, it will be converted to highAnchorText (see Advanced).

labels [str] What labels should be applied

data marker start : Where should the marker start at

data store history [bool] Store full record of how participant moved on the slider

data store rating [bool] Save the rating that was selected

data store rating time [bool] Save the time from the beginning of the trial until the participant responds.

See also:

API reference for RatingScale
5.5.23 Slider Component

A slider uses mouse input to collect ratings, either on a continuous scale or Likert scale (1-to-7).

**Properties**

- **Name** [string] Everything in a PsychoPy experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).
- **Start** : The time that the stimulus should first appear. See *Defining the onset/duration of components* for details.
- **Stop** : The duration for which the stimulus is presented. See *Defining the onset/duration of components* for details.
- **Size** [(width, height)] The size controls the width and height of the slider. The slider is oriented horizontally when the width is greater than the height, and oriented vertically otherwise. Default is (1.0, 0.1)
- **Position** [(X,Y)] The position of the centre of the stimulus, in the units specified by the stimulus or window. Default is centered left-right, and somewhat lower than the vertical center (0, -0.4).
- **Ticks** [(list or tuple of integers)] The ticks that will be place on the slider scale. The first and last ticks will be placed on the ends of the slider, and the remaining are spaced between the endpoints corresponding to their values. For example, (1, 2, 3, 4, 5) will create 5 evenly spaced ticks. (1, 3, 5) will create three ticks, one at each end and one in the middle.
- **Labels** [(list or tuple of strings)] The text to go with each tick (or spaced evenly across the ticks). If you give 3 labels but 5 tick locations then the end and middle ticks will be given labels. If the labels can’t be distributed across the ticks then an error will be raised. If you want an uneven distribution you should include a list matching the length of ticks but with some values set to None.
- **Granularity** : Specifies step size for rating. 0 corresponds to a continuous scale, 1 corresponds to an integer or discrete scale.
- **Force end of Routine** : If checked, when the subject makes a rating the routine will be ended.
- **Opacity** [value from 0 to 1] If opacity is reduced then the underlying images/stimuli will show through.
- **Units** : See *Units for the window and stimuli*.

**Appearance**

- **Font** : Font for labels.
- **Flip** : By default labels are below the scale or left of the scale. By checking this checkbox, the labels are placed above the scale or to the right of the scale.
- **Color** : Color of the line, ticks, and labels. See *Color spaces*.
- **Styles** : A selection of pre-defined styles. Multiple styles can be selected.

See also:

API reference for *Slider*
5.5.24 Sound Component

Parameters

name [string] Everything in a PsychoPy experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

start [float or integer] The time that the stimulus should first play. See Defining the onset/duration of components for details.

stop : For sounds loaded from a file leave this blank and then give the Expected duration below for visualisation purposes. See Defining the onset/duration of components for details.

sound : This sound can be described in a variety of ways:
  • a number can specify the frequency in Hz (e.g. 440)
  • a letter gives a note name (e.g. “C”) and sharp or flat can also be added (e.g. “C#” “Bb”)
  • a filename, which can be a relative or absolute path (mid, wav, and ogg are supported).

Playback

How should stimulus play? Speed, volume, etc.

volume [float or integer] The volume with which the sound should be played. It's a normalized value between 0 (minimum) and 1 (maximum).

hamming window [bool] Should there be a hamming window between stimuli?

See also:
API reference for SoundPyo

5.5.25 Static Component

(Added in Version 1.78.00)

The Static Component allows you to have a period where you can preload images or perform other time-consuming operations that not be possible while the screen is being updated.

Typically a static period would be something like an inter-trial or inter-stimulus interval (ITI/ISI). During this period you should not have any other objects being presented that are being updated (this isn’t checked for you - you have to make that check yourself), but you can have components being presented that are themselves static. For instance a fixation point never changes and so it can be presented during the static period (it will be presented and left on-screen while the other updates are being made).

Any stimulus updates can be made to occur during any static period defined in the experiment (it does not have to be in the same Routine). This is done in the updates selection box- once a static period exists it will show up here as well as the standard options of constant and every repeat etc. Many parameter updates (e.g. orientation are made so quickly that using the static period is of no benefit but others, most notably the loading of images from disk, can take substantial periods of time and these should always be performed during a static period to ensure good timing.

If the updates that have been requested were not completed by the end of the static period (i.e. there was a timing overshoot) then you will receive a warning to that effect. In this case you either need a longer static period to perform the actions or you need to reduce the time required for the action (e.g. use an image with fewer pixels).
Parameters

name : Everything in a PsychoPy experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

start : The time that the static period begins. See Defining the onset/duration of components for details.

stop : The time that the static period ends. See Defining the onset/duration of components for details.

Custom

Parameters for injecting custom code

custom code : After running the component updates (which are defined in each component, not here) any code inserted here will also be run

See also:
API reference for StaticPeriod

5.5.26 Text Component

This component can be used to present text to the participant, either instructions or stimuli.

name [string] Everything in a PsychoPy experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

start : The time that the stimulus should first appear. See Defining the onset/duration of components for details.

stop : The duration for which the stimulus is presented. See Defining the onset/duration of components for details.

text [string] Text to be shown

Appearance

How should the stimulus look? Colour, borders, etc.

foreground color : See Color spaces

foreground color space [rgb, dkl or lms] See Color spaces

opacity : Vary the transparency, from 0.0 = invisible to 1.0 = opaque

Layout

How should the stimulus be laid out? Padding, margins, size, position, etc.

flip : Whether to mirror-reverse the text: ‘horiz’ for left-right mirroring, ‘vert’ for up-down mirroring. The flip can be set dynamically on a per-frame basis by using a variable, e.g., $mirror, as defined in a code component or conditions file and set to either ‘horiz’ or ‘vert’.

ori [degrees] The orientation of the stimulus in degrees.

pos [[X,Y]] The position of the centre of the stimulus, in the units specified by the stimulus or window

spatial units [deg, cm, pix, norm, or inherit from window] See Units for the window and stimuli

wrap width [code] How many characters in should text be wrapped at?
**Formatting**

Formatting text

**font** [string] What font should the text be set in? Must be the name of a font installed on your computer

**language style** [LTR, RTL, Arabic] Should text be laid out from left to right (LTR), from right to left (RTL), or laid out like Arabic script?

**letter height** [integer or float] The height of the characters in the given units of the stimulus/window. Note that nearly all actual letters will occupy a smaller space than this, depending on font, character, presence of accents etc. The width of the letters is determined by the aspect ratio of the font.

See also:

API reference for `TextStim`

### 5.5.27 Textbox Component

This component can be used either to present text to the participant, or to allow free-text answers via the keyboard.

**name** [string] Everything in a PsychoPy experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

**start:** The time that the stimulus should first appear. See *Defining the onset/duration of components* for details.

**stop:** The duration for which the stimulus is presented. See *Defining the onset/duration of components* for details.

**text** [string] Text to be shown

**Appearance**

How should the stimulus look? Colour, borders, etc.

**foreground color:** See *Color spaces*

**foreground color space** [rgb, dkl or lms] See *Color spaces*

**opacity:** Vary the transparency, from 0.0 = invisible to 1.0 = opaque

**Layout**

How should the stimulus be laid out? Padding, margins, size, position, etc.

**flip:** Whether to mirror-reverse the text: ‘horiz’ for left-right mirroring, ‘vert’ for up-down mirroring. The flip can be set dynamically on a per-frame basis by using a variable, e.g., $mirror, as defined in a code component or conditions file and set to either ‘horiz’ or ‘vert’.

**ori** [degrees] The orientation of the stimulus in degrees.

**pos** [[X,Y]] The position of the centre of the stimulus, in the units specified by the stimulus or window

**spatial units** [deg, cm, pix, norm, or inherit from window] See *Units for the window and stimuli*

**wrap width** [code] How many characters in should text be wrapped at?
Formatting

Formatting text

**font** [string] What font should the text be set in? Must be the name of a font installed on your computer

**language style** [LTR, RTL, Arabic] Should text be laid out from left to right (LTR), from right to left (RTL), or laid out like Arabic script?

**letter height** [integer or float] The height of the characters in the given units of the stimulus/window. Note that nearly all actual letters will occupy a smaller space than this, depending on font, character, presence of accents etc. The width of the letters is determined by the aspect ratio of the font.

See also:

API reference for TextStim

5.5.28 Variable Component

A variable can hold quantities or values in memory that can be referenced using a variable name. You can store values in a variable to use in your experiments.

Parameters

**Name** [string] Everything in a PsychoPy experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces). The variable name references the value stored in memory, so that your stored values can be used in your experiments.

**Start** [int, float or bool] The time or condition from when you want your variable to be defined. The default value is None, and so will be defined at the beginning of the experiment, trial or frame. See Defining the onset/duration of components for details.

**Stop** [int, float or bool] The duration for which the variable is defined/updated. See Defining the onset/duration of components for details.

**Experiment start value:** any The variable can take any value at the beginning of the experiment, so long as you define your variables using literals or existing variables.

**Routine start value** [any] The variable can take any value at the beginning of a routine/trial, and can remain a constant, or be defined/updated on every routine.

**Frame start value** [any] The variable can take any value at the beginning of a frame, or during a condition bases on Start and/or Stop.

Data

What information to save, how to lay it out and when to save it.

**Save exp start value** [bool] Choose whether or not to save the experiment start value to your data file.

**Save routine start value** [bool] Choose whether or not to save the routine start value to your data file.

**Save frame value** [bool and drop=down menu] Frame values are contained within a list for each trial, and discarded at the end of each trial. Choose whether or not to take the first, last or average variable values from the frame container, and save to your data file.

**Save routine end value** [bool] Choose whether or not to save the routine end value to your data file.

**Save exp end value** [bool] Choose whether or not to save the experiment end value to your data file.
5.5.29 Entering parameters

Most of the entry boxes for Component parameters simply receive text or numeric values or lists (sequences of values surrounded by square brackets) as input. In addition, the user can insert variables and code into most of these, which will be interpreted either at the beginning of the experiment or at regular intervals within it.

To indicate to PsychoPy that the value represents a variable or python code, rather than literal text, it should be preceded by a $. For example, inserting intensity into the text field of the Text Component will cause that word literally to be presented, whereas $intensity will cause python to search for the variable called intensity in the script.

Variables associated with Loops can also be entered in this way (see Accessing loop parameters from components for further details). But it can also be used to evaluate arbitrary python code.

For example:

- $random(2) will generate a pair of random numbers
- $"yn"[randint(2)] will randomly choose the first or second character (y or n)
- $globalClock.getTime() will insert the current time in secs of the globalClock object
- $[sin(angle), cos(angle)] will insert the sin and cos of an angle (e.g. into the x,y coords of a stimulus)

5.5.30 How often to evaluate the variable/code

If you do want the parameters of a stimulus to be evaluated by code in this way you need also to decide how often it should be updated. By default, the parameters of Components are set to be constant; the parameter will be set at the beginning of the experiment and will remain that way for the duration. Alternatively, they can be set to change either on every repeat in which case the parameter will be set at the beginning of the Routine on each repeat of it. Lastly many parameters can even be set on every frame, allowing them to change constantly on every refresh of the screen.

5.6 Experiment settings

The settings menu can be accessed by clicking the icon at the top of the window. It allows the user to set various aspects of the experiment, such as the size of the window to be used or what information is gathered about the subject and determine what outputs (data files) will be generated.

5.6.1 Settings

Basic settings

**Experiment name:** A name that will be stored in the metadata of the data file.

**Show info dlg:** If this box is checked then a dialog will appear at the beginning of the experiment allowing the Experiment Info to be changed.

**Experiment Info:** This information will be presented in a dialog box at the start and will be saved with any data files and so can be used for storing information about the current run of the study. The information stored here can also be used within the experiment. For example, if the Experiment Info included a field called ori then Builder Components could access expInfo[‘ori’] to retrieve the orientation set here. Obviously this is a useful way to run essentially the same experiment, but with different conditions set at run-time.

**Enable escape:** If ticked then the Esc key can be used to exit the experiment at any time (even without a keyboard component)
Data settings

Data filename: A formatted string to control the base filename and path, often based on variables such as the date and/or the participant. This base filename will be given the various extensions for the different file types as needed. Examples:

```python
# all in data folder relative to experiment file: data/JWP_memoryTask_2014_Feb_15_1648
'data/%s_%s_%s' %(expInfo['participant'], expName, expInfo['date'])

# group by participant folder: data/JWP/memoryTask-2014_Feb_15_1648
'data/%s/%s-%s' %(expInfo['participant'], expName, expInfo['date'])

# put into dropbox: ~/dropbox/data/memoryTask/JWP-2014_Feb_15_1648
# os.path.expanduser replaces '~' with the path to your home directory,
# os.path.join joins the path components together correctly, regardless of OS
# os.path.relpath creates a relative path between the specified path and the...
'~os.path.relpath(os.path.join(os.path.expanduser('~'), 'dropbox', 'data',...
  +expName, expInfo['participant'] + '-' + expInfo['date']))
```

Save Excel file: If this box is checked an Excel data file (.xlsx) will be stored.

Save csv file: If this box is checked a comma separated variable (.csv) will be stored.

Save psydat file: If this box is checked a PsychoPy data file (.psydat) will be stored. This is a Python specific format (.pickle files) which contains more information than .xlsx or .csv files that can be used with data analysis and plotting scripts written in Python. Whilst you may not wish to use this format it is recommended that you always save a copy as it contains a complete record of the experiment at the time of data collection.

Save log file: A log file provides a record of what occurred during the experiment in chronological order, including information about any errors or warnings that may have occurred.

Logging level: How much detail do you want to be output to the log file, if it is being saved. The lowest level is error, which only outputs error messages; warning outputs warnings and errors; info outputs all info, warnings and errors; debug outputs all info that can be logged. This system enables the user to get a great deal of information while generating their experiments, but then reducing this easily to just the critical information needed when actually running the study. If your experiment is not behaving as you expect it to, this is an excellent place to begin to work out what the problem is.

Screen settings

Monitor: The name of the monitor calibration. Must match one of the monitor names from Monitor Center.

Screen: If multiple screens are available (and if the graphics card is not an intel integrated graphics chip) then the user can choose which screen they use (e.g. 1 or 2).

Full-screen window: If this box is checked then the experiment window will fill the screen (overriding the window size setting and using the size that the screen is currently set to in the operating system settings).

Window size: The size of the window in pixels, if this is not to be a full-screen window.

Units: The default units of the window (see Units for the window and stimuli). These can be overridden by individual Components.
5.7 Defining the onset/duration of components

As of version 1.70.00, the onset and offset times of stimuli can be defined in several ways.

Start and stop times can be entered in terms of seconds (time (s)), by frame number (frameN) or in relation to another stimulus (condition). Condition would be used to make Components start or stop depending on the status of something else, for example when a sound has finished. Duration can also be varied using a Code Component.

If you need very precise timing (particularly for very brief stimuli for instance) then it is best to control your onset/duration by specifying the number of frames the stimulus will be presented for.

Measuring duration in seconds (or milliseconds) is not very precise because it doesn’t take into account the fact that your monitor has a fixed frame rate. For example if the screen has a refresh rate of 60Hz you cannot present your stimulus for 120ms; the frame rate would limit you to 116.7ms (7 frames) or 133.3ms (8 frames). The duration of a frame (in seconds) is simply 1/refresh rate in Hz.

Condition would be used to make Components start or stop depending on the status of something else, for example when a movie has finished. Duration can also be varied using a code component.

In cases where PsychoPy cannot determine the start/endpoint of your Component (e.g. because it is a variable) you can enter an ‘Expected’ start/duration. This simply allows components with variable durations to be drawn in the Routine window. If you do not enter the approximate duration it will not be drawn, but this will not affect experimental performance.

For more details of how to achieve good temporal precision see Timing Issues and synchronisation

5.7.1 Examples

- Use time(s) or frameN and simply enter numeric values into the start and duration boxes.
- Use time(s) or frameN and enter a numeric value into the start time and set the duration to a variable name by preceding it with a $ as described here. Then set expected time to see an approximation in your routine
- Use condition to cause the stimulus to start immediately after a movie component called myMovie, by entering $myMovie.status==FINISHED into the start time.

5.8 Generating outputs (datafiles)

There are 4 main forms of output file from PsychoPy:

- Excel 2007 files (.xlsx) see Excel Data Files for more details
- text data files (.csv, .tsv, or .txt) see Delimited Text Files for more details
- binary data files (.psydat) see PsychoPy Data Files for more details
- log files (.log) see Log Files for more details
5.9 Common Mistakes (aka Gotcha’s)

5.9.1 General Advice

• Python and therefore PsychoPy is CASE SENSITIVE
• To use a dollar sign ($) for anything other than to indicate a code snippet for example in a text, precede it with a backslash \$ (the backslash won’t be printed)
• Have you entered your the settings for your monitor? If you are using degrees as a unit of measurement and haven’t entered your monitor settings, the size of stimuli will not be accurate.
• If your experiment is not behaving in the way that you expect. Have you looked at the log file? This can point you in the right direction. Did you know you can change the type of information that is stored in the log file in preferences by changing the logging level.
• Have you tried compiling the script and running it. Does this produce a particular error message that points you at a particular problem area? You can also change things in a more detailed way in the coder view and if you are having problems, reading through the script can highlight problems. Reading a compiled script can also help with the creation of a Code Component

5.9.2 My stimulus isn’t appearing, there’s only the grey background

• Have you checked the size of your stimulus? If it is 0.5x0.5 pixels you won’t be able to see it!
• Have you checked the position of your stimulus? Is it positioned off the screen?

5.9.3 The loop isn’t using my Excel spreadsheet

• Have you remembered to specify the file you want to use when setting up the loop?
• Have you remembered to add the variables proceeded by the $ symbol to your stimuli?

5.9.4 I just want a plain square, but it’s turning into a grating

• If you don’t want your stimulus to have a texture, you need Image to be None

5.9.5 The code snippet I’ve entered doesn’t do anything

• Have you remembered to put a $ symbol at the beginning (this isn’t necessary, and should be avoided in a Code Component)?
• A dollar sign as the first character of a line indicates to PsychoPy that the rest of the line is code. It does not indicate a variable name (unlike in perl or php). This means that if you are, for example, using variables to determine position, enter $[x,y]. The temptation is to use [x,y], which will not work.
5.9.6 My stimulus isn’t changing as I progress through the loop

- Have you changed the setting for the variable that you want to change to ‘change every repeat’ (or ‘change every frame’)?

5.9.7 I’m getting the error message AttributeError: ‘unicode object has no attribute ‘XXXX’

- This type of error is usually caused by a naming conflict. Whilst we have made every attempt to make sure that these conflicts produce a warning message it is possible that they may still occur.
- The most common source of naming conflicts in an external file which has been imported to be used in a loop i.e. .xlsx, .csv.
- Check to make sure that all of the variable names are unique. There can be no repeated variable names anywhere in your experiment.

5.9.8 The window opens and immediately closes

- Have you checked all of your variable entries are accepted commands e.g. gauss but not Gaussian
- If you compile your experiment and run it from the coder window what does the error message say? Does it point you towards a particular variable which may be incorrectly formatted?

If you are having problems getting the application to run please see Troubleshooting

5.10 Compiling a Script

If you click the compile script icon this will display the script for your experiment in the Coder window.

This can be used for debugging experiments, entering small amounts of code and learning a bit about writing scripts amongst other things.

The code is fully commented and so this can be an excellent introduction to writing your own code.

5.11 Set up your monitor properly

It’s a really good idea to tell PsychoPy about the set up of your monitor, especially the size in cm and pixels and its distance, so that PsychoPy can present your stimuli in units that will be consistent in another lab with a different set up (e.g. cm or degrees of visual angle).

You should do this in Monitor Center which can be opened from Builder by clicking on the icon that shows two monitors. In Monitor Center you can create settings for multiple configurations, e.g. different viewing distances or different physical devices and then select the appropriate one by name in your experiments or scripts.

Having set up your monitor settings you should then tell PsychoPy which of your monitor setups to use for this experiment by going to the Experiment settings dialog.
5.12 Future developments

The new big feature, which we're really excited about is that Builder experiments are going to web-enabled very soon! Make sure you watch for new posts in the PsychoPy forum Announcements category so you get updates of when this is available.
Note: These do not teach you about Python per se, and you are recommended also to learn about that (Python has many excellent tutorials for programmers and non-programmers alike). In particular, dictionaries, lists and numpy arrays are used a great deal in most PsychoPy experiments.

You can learn to use the scripting interface to PsychoPy in several ways, and you should probably follow a combination of them:

- **Basic Concepts**: some of the logic of PsychoPy scripting
- **PsychoPy Tutorials**: walk you through the development of some semi-complete experiments
- **demos**: in the demos menu of Coder view. Many and varied
- **use the Builder to compile a script** and see how it works
- **check the Reference Manual (API) for further details**
- **ultimately go into PsychoPy and start examining the source code. It’s just regular python!**

6.1 Basic Concepts

6.1.1 Presenting Stimuli

Note: Before you start, tell PsychoPy about your monitor(s) using the **Monitor Center**. That way you get to use units (like degrees of visual angle) that will transfer easily to other computers.

**Stimulus objects**

Python is an ‘object-oriented’ programming language, meaning that most stimuli in PsychoPy are represented by python objects, with various associated methods and information.

Typically you should create your stimulus with the initial desired attributes once, at the beginning of the script, and then change select attributes later (see section below on setting stimulus attributes). For instance, create your text and then change its color any time you like:
from psychopy import visual, core
win = visual.Window([400,400])
message = visual.TextStim(win, text='hello')
message.autoDraw = True  # Automatically draw every frame
win.flip()
core.wait(2.0)
message.text = 'world'  # Change properties of existing stim
win.flip()
core.wait(2.0)

Setting stimulus attributes

Stimulus attributes are typically set using either

• a string, which is just some characters (as message.text = 'world' above)
• a scalar (a number; see below)
• an x,y-pair (two numbers; see below)

x,y-pair: PsychoPy is very flexible in terms of input. You can specify the widely used x,y-pairs using these types:

• A Tuple (x, y) with two elements
• A List [x, y] with two elements
• A numpy array([x, y]) with two elements

However, PsychoPy always converts the x,y-pairs to numpy arrays internally. For example, all three assignments of pos are equivalent here:

stim.pos = (0.5, -0.2)  # Right and a bit up from the center
print stim.pos  # array([0.5, -0.2])
stim.pos = [0.5, -0.2]
print stim.pos  # array([0.5, -0.2])
stim.pos = numpy.array([0.5, -0.2])
print stim.pos  # array([0.5, -0.2])

Choose your favorite :-) However, you can’t assign elementwise:

stim.pos[1] = 4  # has no effect

Scalar: Int or Float.

Mostly, scalars are no-brainers to understand. E.g.:

stim.ori = 90  # Rotate stimulus 90 degrees
stim.opacity = 0.8  # Make the stimulus slightly transparent.

However, scalars can also be used to assign x,y-pairs. In that case, both x and y get the value of the scalar. E.g.:

stim.size = 0.5
print stim.size  # array([0.5, 0.5])

Operations on attributes: Operations during assignment of attributes are a handy way to smoothly alter the appearance of your stimuli in loops.

Most scalars and x,y-pairs support the basic operations:
They are easy to use and understand on scalars:

```python
stim.ori = 5      # 5.0, set rotation
stim.ori += 3.8   # 8.8, rotate clockwise
stim.ori -= 0.8   # 8.0, rotate counterclockwise
stim.ori /= 2     # 4.0, home in on zero
stim.ori **= 3    # 64.0, exponential increase in rotation
stim.ori %= 10    # 4.0, modulus 10
```

However, they can also be used on x,y-pairs in very flexible ways. Here you can use both scalars and x,y-pairs as operators. In the latter case, the operations are element-wise:

```python
stim.size = 5      # array([5.0, 5.0]), set quadratic size
stim.size += 2     # array([7.0, 7.0]), increase size
stim.size /= 2     # array([3.5, 3.5]), downscale size
stim.size += (0.5, 2.5) # array([4.0, 6.0]), a little wider and much taller
stim.size *= (2, 0.25)  # array([8.0, 1.5]), upscale horizontal and downscale vertical
```

Operations are not meaningful for strings.

### Timing

There are various ways to measure and control timing in PsychoPy:

- using frame refresh periods (most accurate, least obvious)
- checking the time on Clock objects
- using `core.wait()` commands (most obvious, least flexible/accurate)

Using `core.wait()`, as in the above example, is clear and intuitive in your script. But it can’t be used while something is changing. For more flexible timing, you could use a `Clock()` object from the `core` module:

```python
from psychopy import visual, core

# Setup stimulus
win = visual.Window([400, 400])
gabor = visual.GratingStim(win, tex='sin', mask='gauss', sf=5, name='gabor')
gabor.autoDraw = True    # Automatically draw every frame
gabor.autoLog = False    # Or we’ll get many messages about phase change

# Let's draw a stimulus for 2s, drifting for middle 0.5s
 clock = core.Clock()
while clock.getTime() < 2.0:  # Clock times are in seconds
    if 0.5 <= clock.getTime() < 1.0:
        gabor.phase += 0.1  # Increment by 10th of cycle
    win.flip()
```

Clocks are accurate to around 1ms (better on some platforms), but using them to time stimuli is not very accurate because it fails to account for the fact that one frame on your monitor has a fixed frame rate. In the above, the stimulus does not
actually get drawn for exactly 0.5s (500ms). If the screen is refreshing at 60Hz (16.7ms per frame) and the getTime() call reports that the time has reached 1.999s, then the stimulus will draw again for a frame, in accordance with the while loop statement and will ultimately be displayed for 2.0167s. Alternatively, if the time has reached 2.001s, there will not be an extra frame drawn. So using this method you get timing accurate to the nearest frame period but with little consistent precision. An error of 16.7ms might be acceptable to long-duration stimuli, but not to a brief presentation. It also might also give the false impression that a stimulus can be presented for any given period. At 60Hz refresh you can not present your stimulus for, say, 120ms; the frame period would limit you to a period of 116.7ms (7 frames) or 133.3ms (8 frames).

As a result, the most precise way to control stimulus timing is to present them for a specified number of frames. The framerate is extremely precise, much better than ms-precision. Calls to Window.flip() will be synchronised to the frame refresh; the script will not continue until the flip has occurred. As a result, on most cards, as long as frames are not being ‘dropped’ (see Detecting dropped frames) you can present stimuli for a fixed, reproducible period.

Note: Some graphics cards, such as Intel GMA graphics chips under win32, don’t support frame sync. Avoid integrated graphics for experiment computers wherever possible.

Using the concept of fixed frame periods and flip() calls that sync to those periods we can time stimulus presentation extremely precisely with the following:

```python
from psychopy import visual, core

# Setup stimulus
win = visual.Window([400, 400])
gabor = visual.GratingStim(win, tex='sin', mask='gauss', sf=5,
name='gabor', autoLog=False)
fixation = visual.GratingStim(win, tex=None, mask='gauss', sf=0, size=0.02,
name='fixation', autoLog=False)

# Let's draw a stimulus for 200 frames, drifting for frames 50:100
for frameN in range(200):
    # For exactly 200 frames
    if 10 <= frameN < 150:
        fixation.draw()
    # Present stim for a different subset
    if 50 <= frameN < 100:
        gabor.phase += 0.1  # Increment by 10th of cycle
        gabor.draw()
    win.flip()
```

Using autoDraw

Stimuli are typically drawn manually on every frame in which they are needed, using the draw() function. You can also set any stimulus to start drawing every frame using stim.autoDraw = True or stim.autoDraw = False. If you use these commands on stimuli that also have autoLog=True, then these functions will also generate a log message on the frame when the first drawing occurs and on the first frame when it is confirmed to have ended.
6.1.2 Logging data

TrialHandler and StairHandler can both generate data outputs in which responses are stored, in relation to the stimulus conditions. In addition to those data outputs, PsychoPy can create detailed chronological log files of events during the experiment.

Log levels and targets

Log messages have various levels of severity: ERROR, WARNING, DATA, EXP, INFO and DEBUG

Multiple targets can also be created to receive log messages. Each target has a particular critical level and receives all logged messages greater than that. For example, you could set the console (visual output) to receive only warnings and errors, have a central log file that you use to store warning messages across studies (with file mode append), and another to create a detailed log of data and events within a single study with level=INFO:

```python
from psychopy import logging
logging.console.setLevel(logging.WARNING)
# overwrite (filemode='w') a detailed log of the last run in this dir
lastLog = logging.LogFile("lastRun.log", level=logging.INFO, filemode='w')
# also append warnings to a central log file
centralLog = logging.LogFile("C:\psychopyExps.log", level=logging.WARNING, filemode='a')
```

Updating the logs

For performance purposes log files are not actually written when the log commands are ‘sent’. They are stored in a list and processed automatically when the script ends. You might also choose to force a flush of the logged messages manually during the experiment (e.g. during an inter-trial interval):

```python
from psychopy import logging
...
logging.flush()  # write messages out to all targets
```

This should only be necessary if you want to see the logged information as the experiment progresses.

AutoLogging

New in version 1.63.00

Certain events will log themselves automatically by default. For instance, visual stimuli send log messages every time one of their parameters is changed, and when autoDraw is toggled they send a message that the stimulus has started/stopped. All such log messages are timestamped with the frame flip on which they take effect. To avoid this logging, for stimuli such as fixation points that might not be critical to your analyses, or for stimuli that change constantly and will flood the logging system with messages, the autoLogging can be turned on/off at initialisation of the stimulus and can be altered afterwards with .setAutoLog(True/False)
Manual methods

In addition to a variety of automatic logging messages, you can create your own, of various levels. These can be timestamped immediately:

```python
from psychopy import logging
logging.log(level=logging.WARN, msg='something important')
logging.log(level=logging.EXP, msg='something about the conditions')
logging.log(level=logging.DATA, msg='something about a response')
logging.log(level=logging.INFO, msg='something less important')
```

There are additional convenience functions for the above: logging.warn('a warning') etc.

For stimulus changes you probably want the log message to be timestamped based on the frame flip (when the stimulus is next presented) rather than the time that the log message is sent:

```python
from psychopy import logging, visual
win = visual.Window([400, 400])
win.flip()
logging.log(level=logging.EXP, msg='sent immediately')
win.logOnFlip(level=logging.EXP, msg='sent on actual flip')
win.flip()
```

Using a custom clock for logs

New in version 1.63.00

By default times for log files are reported as seconds after the very beginning of the script (often it takes a few seconds to initialise and import all modules too). You can set the logging system to use any given `core.Clock` object (actually, anything with a `getTime()` method):

```python
from psychopy import core, logging
globalClock = core.Clock()
logging.setDefaultClock(globalClock)
```

6.1.3 Handling Trials and Conditions

TrialHandler

This is what underlies the random and sequential loop types in Builder, they work using the method of constants. The trialHandler presents a predetermined list of conditions in either a sequential or random (without replacement) order. see TrialHandler for more details.

TrialHandlerExt (For oddball paradigms)

For handling trial sequences in a non-counterbalanced design (i.e. oddball paradigms, https://en.wikipedia.org/wiki/Oddball_paradigm). The oddball paradigm is very popular in EEG research.

Its functions are a superset of the class TrialHandler, and as such, can also be used for normal trial handling. see TrialHandlerExt for more details.
### StairHandler

This generates the next trial using an *adaptive staircase*. The conditions are not predetermined and are generated based on the participant’s responses.

Staircases are predominately used in psychophysics to measure the discrimination and detection thresholds. However they can be used in any experiment which varies a numeric value as a result of a 2 alternative forced choice (2AFC) response.

The StairHandler systematically generates numbers based on staircase parameters. These can then be used to define a stimulus parameter e.g. spatial frequency, stimulus presentation duration. If the participant gives the incorrect response the number generated will get larger and if the participant gives the correct response the number will get smaller.

see *StairHandler* for more details

#### 6.1.4 Global Event Keys

Global event keys are single keys (or combinations of a single key and one or more “modifier” keys such as Ctrl, Alt, etc.) with an associated Python callback function. This function will be executed if the key (or key/modifiers combination) was pressed.

**Note:** Global event keys only work with the *pyglet* backend, which is the default.

PsychoPy fully automatically monitors and processes key presses during most portions of the experimental run, for example during `core.wait()` periods, or when calling `win.flip()`. If a global event key press is detected, the specified function will be run immediately. You are not required to manually poll and check for key presses. This can be particularly useful to implement a global “shutdown” key, or to trigger laboratory equipment on a key press when testing your experimental script – without cluttering the code. But of course the application is not limited to these two scenarios. In fact, you can associate any Python function with a global event key.

All active global event keys are stored in `event.globalKeys`.

**Adding a global event key (simple)**

First, let’s ensure no global event keys are currently set by calling `func:event.globalKeys.clear`.

```python
>>> from psychopy import event
>>> event.globalKeys.clear()
```

To add a new global event key, you need to invoke `func:event.globalKeys.add`. This function has two required arguments: the key name, and the function to associate with that key.

```python
>>> key = 'a'
>>> def myfunc():
...     pass
... >>> event.globalKeys.add(key=key, func=myfunc)
```

Look at `event.globalKeys`, we can see that the global event key has indeed been created.

```python
>>> event.globalKeys
<GlobalEventKeys :
    [A] -> 'myfunc' <function myfunc at 0x10669ba28>
>
```

6.1. Basic Concepts 91
Your output should look similar. You may happen to spot We can take a closer look at the specific global key event we added.

```python
>>> event.globalKeys['a']
_GlobalEvent(func=<function myfunc at 0x10669ba28>, func_args=(), func_kwargs={},...
→name='myfunc')
```

This output tells us that

- our key a is associated with our function `myfunc`
- `myfunc` will be called without passing any positional or keyword arguments (`func_args` and `func_kwargs`, respectively)
- the event name was automatically set to the name of the function.

**Note:** Pressing the key won’t do anything unless a `psychopy.visual.Window` is created and and its `:func:~`psychopy.visual.Window.flip` method or `psychopy.core.wait()` are called.

### Adding a global event key (advanced)

We are going to associate a function with a more complex calling signature (with positional and keyword arguments) with a global event key. First, let’s create the dummy function:

```python
>>> def myfunc2(*args, **kwargs):
...    pass
...
```

Next, compile some positional and keyword arguments and a custom name for this event. Positional arguments must be passed as tists or tuples, and keyword arguments as dictionaries.

```python
>>> args = (1, 2)
>>> kwargs = dict(foo=3, bar=4)
>>> name = 'my name'
```

**Note:** Even when intending to pass only a single positional argument, `args` must be a list or tuple, e.g., `args = [1]` or `args = (1,)`.

Finally, specify the key and a combination of modifiers. While key names are just strings, modifiers are lists or tuples of modifier names.

```python
>>> key = 'b'
>>> modifiers = ['ctrl', 'alt']
```

**Note:** Even when specifying only a single modifier key, `modifiers` must be a list or tuple, e.g., `modifiers = ['ctrl']` or `modifiers = ('ctrl',)`.

We are now ready to create the global event key.

```python
>>> event.globalKeys.add(key=key, modifiers=modifiers,
...                       func=myfunc2, func_args=args, func_kwargs=kwargs,
...                       name=name)
```
Check that the global event key was successfully added.

```python
>>> event.globalKeys
<_GlobalEventKeys :
    [A] -> 'myfunc' <function myfunc at 0x10669ba28>
    [CTRL] + [ALT] + [B] -> 'my name' <function myfunc2 at 0x112eeeb90>
>
```

The key combination `[CTRL] + [ALT] + [B]` is now associated with the function `myfunc2`, which will be called in the following way:

```python
globalKeys.add()
```

### Indexing

`event.globalKeys` can be accessed like an ordinary dictionary. The index keys are `(key, modifiers)` namedtuples.

```python
>>> event.globalKeys.keys()
[<_IndexKey(key='a', modifiers=()), _IndexKey(key='b', modifiers=('ctrl', 'alt'))]
```

To access the global event associated with the key combination `[CTRL] + [ALT] + [B]`, we can do

```python
>>> event.globalKeys['b', ['ctrl', 'alt']]
_GlobalEvent(func=<function myfunc2 at 0x112eeeb90>, func_args=(1, 2), func_kwargs={
    →'foo': 3, 'bar': 4}, name='my name')
```

To make access more convenient, specifying the modifiers is optional in case none were passed to `psychopy.event.globalKeys.add()` when the global event key was added, meaning the following commands are identical.

```python
>>> event.globalKeys['a', ()]
_GlobalEvent(func=<function myfunc at 0x10669ba28>, func_args=(), func_kwargs={},
    →name='myfunc')
```

All elements of a global event can be accessed directly.

```python
>>> index = ('b', ['ctrl', 'alt'])
>>> event.globalKeys[index].func
<function myfunc2 at 0x112eeeb90>
>>> event.globalKeys[index].func_args
(1, 2)
>>> event.globalKeys[index].func_kwargs
{'foo': 3, 'bar': 4}
>>> event.globalKeys[index].name
'my name'
```
Number of active event keys

The number of currently active event keys can be retrieved by passing `event.globalKeys` to the `len()` function.

```python
>>> len(event.globalKeys)
2
```

Removing global event keys

There are three ways to remove global event keys:

- using `psychopy.event.globalKeys.remove()`
- using `del` and
- using `psychopy.event.globalKeys.pop()`.

`psychopy.event.globalKeys.remove()`

To remove a single key, pass the key name and modifiers (if any) to `psychopy.event.globalKeys.remove()`.

```python
>>> event.globalKeys.remove(key='a')
```

A convenience method to quickly delete all global event keys is to pass `key='all'

```python
>>> event.globalKeys.remove(key='all')
```

`del`

Like with other dictionaries, items can be removed from `event.globalKeys` by using the `del` statement. The provided index key must be specified as described in `Indexing`.

```python
>>> index = ('b', ['ctrl', 'alt'])
>>> del event.globalKeys[index]
```

`psychopy.event.globalKeys.pop()`

Again, as other dictionaries, `event.globalKeys` provides a `pop` method to retrieve an item and remove it from the dict. The first argument to `pop` is the index key, specified as described in `Indexing`. The second argument is optional. Its value will be returned in case no item with the matching indexing key could be found, for example if the item had already been removed previously.

```python
>>> r = event.globalKeys.pop('a', None)
>>> print(r)
_GlobalEvent(func=<function myfunc at 0x10669ba28>, func_args=(), func_kwargs={}, ...
,name='myfunc')
>>> r = event.globalKeys.pop('a', None)
>>> print(r)
None
```
Global shutdown key

The PsychoPy preferences for `shutdownKey` and `shutdownKeyModifiers` (both unset by default) will be used to automatically create a global shutdown key. To demonstrate this automated behavior, let us first change the preferences programmatically (these changes will be lost when quitting the current Python session).

```python
>>> from psychopy.preferences import prefs
>>> prefs.general['shutdownKey'] = 'q'
```

We can now check if a global shutdown key has been automatically created.

```python
>>> from psychopy import event
>>> event.globalKeys
<_GlobalEventKeys : [Q] -> 'shutdown (auto-created from prefs)' <function quit at 0x10c171938> >
```

And indeed, it worked!

What happened behind the scenes? When importing the `psychopy.event` module, the initialization of `event.globalKeys` checked for valid shutdown key preferences and automatically initialized a shutdown key accordingly. This key is associated with the :func:`psychopy.core.quit` function, which will shut down PsychoPy.

```python
>>> from psychopy.core import quit
>>> event.globalKeys['q'].func == quit
True
```

Of course you can very easily add a global shutdown key manually, too. You simply have to associate a key with :func:`psychopy.core.quit`.

```python
>>> from psychopy import core, event
>>> event.globalKeys.add(key='q', func=core.quit, name='shutdown')
```

That's it!

A working example

In the above code snippets, our global event keys were not actually functional, as we didn't create a window, which is required to actually collect the key presses. Our working example will thus first create a window and then add global event keys to change the window color and quit the experiment, respectively.

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from __future__ import print_function
from psychopy import core, event, visual

def change_color(win, log=False):
    win.color = 'blue' if win.color == 'gray' else 'gray'
    if log:
        print('Changed color to %s' % win.color)

win = visual.Window(color='gray')
text = visual.TextStim(win,
```

(continues on next page)
text='Press C to change color,\nCTRL + Q to quit.\n'

# Global event key to change window background color.
event.globalKeys.add(key='c',
    func=change_color,
    func_args=[win],
    func_kwargs=dict(log=True),
    name='change window color')

# Global event key (with modifier) to quit the experiment ("shutdown key").
event.globalKeys.add(key='q', modifiers=['ctrl'], func=core.quit)

while True:
    text.draw()
    win.flip()

6.2 PsychoPy Tutorials

6.2.1 Tutorial 1: Generating your first stimulus

A tutorial to get you going with your first stimulus display.

Know your monitor

PsychoPy has been designed to handle your screen calibrations for you. It is also designed to operate (if possible) in the final experimental units that you like to use e.g. degrees of visual angle.

In order to do this PsychoPy needs to know a little about your monitor. There is a GUI to help with this (select MonitorCenter from the tools menu of PsychoPyIDE or run ...site-packages/monitors/MonitorCenter.py).

In the MonitorCenter window you can create a new monitor name, insert values that describe your monitor and run calibrations like gamma corrections. For now you can just stick to the [testMonitor] but give it correct values for your screensize in number of pixels and width in cm.

Now, when you create a window on your monitor you can give it the name ‘testMonitor’ and stimuli will know how they should be scaled appropriately.

Your first stimulus

Building stimuli is extremely easy. All you need to do is create a Window, then some stimuli. Draw those stimuli, then update the window. PsychoPy has various other useful commands to help with timing too. Here’s an example. Type it into a coder window, save it somewhere and press run.

```
from psychopy import visual, core  # import some libraries from PsychoPy

# create a window
mywin = visual.Window([800,600], monitor="testMonitor", units="deg")

# create some stimuli
grating = visual.GratingStim(win=mywin, mask="circle", size=3, pos=[-4,0], sf=3)
fixation = visual.GratingStim(win=mywin, size=0.5, pos=[0,0], sf=0, rgb=-1)
```
# draw the stimuli and update the window
grating.draw()
fixation.draw()
mywin.update()

# pause, so you get a chance to see it!
core.wait(5.0)

Note: For those new to Python. Did you notice that the grating and the fixation stimuli both call `GratingStim` but have different arguments? One of the nice features about python is that you can select which arguments to set. GratingStim has over 15 arguments that can be set, but the others just take on default values if they aren't needed.

That's a bit easy though. Let's make the stimulus move, at least! To do that we need to create a loop where we change the phase (or orientation, or position…) of the stimulus and then redraw. Add this code in place of the drawing code above:

```python
for frameN in range(200):
    grating.setPhase(0.05, '+')  # advance phase by 0.05 of a cycle
    grating.draw()
    fixation.draw()
    mywin.update()
```

That ran for 200 frames (and then waited 5 seconds as well). Maybe it would be nicer to keep updating until the user hits a key instead. That's easy to add too. In the first line add `event` to the list of modules you'll import. Then replace the line:

```python
for frameN in range(200):
```

with the line:

```python
while True:  # this creates a never-ending loop
```

Then, within the loop (make sure it has the same indentation as the other lines) add the lines:

```python
if len(event.getKeys())>0:
    break
    event.clearEvents()
```

the first line counts how many keys have been pressed since the last frame. If more than zero are found then we break out of the never-ending loop. The second line clears the event buffer and should always be called after you’ve collected the events you want (otherwise it gets full of events that we don’t care about like the mouse moving around etc…).

Your finished script should look something like this:

```python
from psychopy import visual, core, event  # import some libraries from PsychoPy
# create a window
mywin = visual.Window([800, 600], monitor="testMonitor", units="deg")
# create some stimuli
grating = visual.GratingStim(win=mywin, mask='circle', size=3, pos=[-4, 0], sf=3)
fixation = visual.GratingStim(win=mywin, size=0.2, pos=[0, 0], sf=0, rgb=-1)
# draw the stimuli and update the window
```
while True:  # this creates a never-ending loop
    grating.setPhase(0.05, '+')  # advance phase by 0.05 of a cycle
    grating.draw()
    fixation.draw()
    mywin.flip()

    if len(event.getKeys()) > 0:
        break
    event.clearEvents()

# cleanup
mywin.close()
core.quit()

There are several more simple scripts like this in the demos menu of the Coder and Builder views and many more to download. If you're feeling like something bigger then go to Tutorial 2: Measuring a JND using a staircase procedure which will show you how to build an actual experiment.

6.2.2 Tutorial 2: Measuring a JND using a staircase procedure

This tutorial builds an experiment to test your just-noticeable-difference (JND) to orientation, that is it determines the smallest angular deviation that is needed for you to detect that a gabor stimulus isn't vertical (or at some other reference orientation). The method presents a pair of stimuli at once with the observer having to report with a key press whether the left or the right stimulus was at the reference orientation (e.g. vertical).

You can download the full code here. Note that the entire experiment is constructed of less than 100 lines of code, including the initial presentation of a dialogue for parameters, generation and presentation of stimuli, running the trials, saving data and outputting a simple summary analysis for feedback. Not bad, eh?

There are a great many modifications that can be made to this code, however this example is designed to demonstrate how much can be achieved with very simple code. Modifying existing is an excellent way to begin writing your own scripts, for example you may want to try changing the appearance of the text or the stimuli.

Get info from the user

The first lines of code import the necessary libraries. We need lots of the PsychoPy modules for a full experiment, as well as numpy (which handles various numerical/mathematical functions):

```python
from psychopy import core, visual, gui, data, event
from psychopy.tools.filetools import fromFile, toFile
import numpy, random
```

Also note that there are two ways to insert comments in Python (and you should do this often!). Using triple quotes, as in """Here's my comment""", allows you to write a comment that can span several lines. Often you need that at the start of your script to leave yourself a note about the implementation and history of what you've written. For single-line comments, as you'll see below, you can use a simple # to indicate that the rest of the line is a comment.

The try:...except:... lines allow us to try and load a parameter file from a previous run of the experiment. If that fails (e.g. because the experiment has never been run) then create a default set of parameters. These are easy to store in a python dictionary that we'll call expInfo:
try:  # try to get a previous parameters file
    expInfo = fromFile('lastParams.pickle')
except:  # if not there then use a default set
    expInfo = {'observer': 'jwp', 'refOrientation': 0}
expInfo['dateStr'] = data.getDateStr()  # add the current time

The last line adds the current date to to the information, whether we loaded from a previous run or created default values.

So having loaded those parameters, let’s allow the user to change them in a dialogue box (which we’ll call dlg). This is the simplest form of dialogue, created directly from the dictionary above. The dialogue will be presented immediately to the user and the script will wait until they hit OK or Cancel.

If they hit OK then dlg.OK=True, in which case we’ll use the updated values and save them straight to a parameters file (the one we try to load above).

If they hit Cancel then we’ll simply quit the script and not save the values.

# present a dialogue to change params
dlg = gui.DlgFromDict(expInfo, title='simple JND Exp', fixed=['dateStr'])
if dlg.OK:
    toFile('lastParams.pickle', expInfo)  # save params to file for next time
else:
    core.quit()  # the user hit cancel so exit

 Setup the information for trials

We’ll create a file to which we can output some data as text during each trial (as well as outputting a binary file at the end of the experiment). PsychoPy actually has supporting functions to do this automatically, but here we’re showing you the manual way to do it.

We’ll create a filename from the subject+date+.csv (note how easy it is to concatenate strings in python just by ‘adding’ them). csv files can be opened in most spreadsheet packages. Having opened a text file for writing, the last line shows how easy it is to send text to this target document.

# make a text file to save data
fileName = expInfo['observer'] + expInfo['dateStr']
dataFile = open(fileName + '.csv', 'w')  # a simple text file with 'comma-separated-values'
dataFile.write('targetSide,oriIncrement,correct\n')

PsychoPy allows us to set up an object to handle the presentation of stimuli in a staircase procedure, the StairHandler. This will define the increment of the orientation (i.e. how far it is from the reference orientation). The staircase can be configured in many ways, but we’ll set it up to begin with an increment of 20deg (very detectable) and home in on the 80% threshold value. We’ll step up our increment every time the subject gets a wrong answer and step down if they get three right answers in a row. The step size will also decrease after every 2 reversals, starting with an 8dB step (large) and going down to 1dB steps (smallish). We’ll finish after 50 trials.

# create the staircase handler
staircase = data.StairHandler(startVal = 20.0,
    stepType = 'db', stepSizes=[8, 4, 4, 2],
    nUp=1, nDown=3,  # will home in on the 80% threshold
    nTrials=1)
Build your stimuli

Now we need to create a window, some stimuli and timers. We need a `~psychopy.visual.Window` in which to draw our stimuli, a fixation point and two `~psychopy.visual.GratingStim` stimuli (one for the target probe and one as the foil). We can have as many timers as we like and reset them at any time during the experiment, but I generally use one to measure the time since the experiment started and another that I reset at the beginning of each trial.

```python
# create window and stimuli
win = visual.Window([800,600],allowGUI=True,
    monitor='testMonitor', units='deg')
foil = visual.GratingStim(win, sf=1, size=4, mask='gauss',
    ori=expInfo['refOrientation'])
target = visual.GratingStim(win, sf=1, size=4, mask='gauss',
    ori=expInfo['refOrientation'])
fixation = visual.GratingStim(win, color=-1, colorSpace='rgb',
    tex=None, mask='circle', size=0.2)
# and some handy clocks to keep track of time
globalClock = core.Clock()
trialClock = core.Clock()
```

Once the stimuli are created we should give the subject a message asking if they’re ready. The next two lines create a pair of messages, then draw them into the screen and then update the screen to show what we’ve drawn. Finally we issue the command `event.waitKeys()` which will wait for a keypress before continuing.

```python
# display instructions and wait
message1 = visual.TextStim(win, pos=[0, 3], text='Hit a key when ready.
message2 = visual.TextStim(win, pos=[0, -3],
    text="Then press left or right to identify the %.1f deg probe." %expInfo['refOrientation'])
message1.draw()
message2.draw()
fixation.draw()
win.flip() # to show our newly drawn 'stimuli'
# pause until there's a keypress
event.waitKeys()
```

Control the presentation of the stimuli

OK, so we have everything that we need to run the experiment. The following uses a for-loop that will iterate over trials in the experiment. With each pass through the loop the `staircase` object will provide the new value for the intensity (which we will call `thisIncrement`). We will randomly choose a side to present the target stimulus using `numpy.random.random()`, setting the position of the target to be there and the foil to be on the other side of the fixation point.

```python
for thisIncrement in staircase:  # will continue the staircase until it terminates!
    # set location of stimuli
    targetSide = random.choice([-1, 1])  # will be either +1(right) or -1(left)
    foil.setPos([-5*targetSide, 0])
    target.setPos([5*targetSide, 0])  # in other location
    # set orientation of probe
    foil.setOri(expInfo['refOrientation'] + thisIncrement)
```

(continues on next page)
# draw all stimuli
foil.draw()
target.draw()
fixation.draw()
win.flip()

Wait for presentation time of 500ms and then blank the screen (by updating the screen after drawing just the fixation point).

# wait 500ms; but use a loop of x frames for more accurate timing
core.wait(0.5)

(This is not the most precise way to time your stimuli - you’ll probably overshoot by one frame - but its easy to understand. PsychoPy allows you to present a stimulus for acertian number of screen refreshes instead which is better for short stimuli.)

## Get input from the subject

Still within the for-loop (note the level of indentation is the same) we need to get the response from the subject. The method works by starting off assuming that there hasn’t yet been a response and then waiting for a key press. For each key pressed we check if the answer was correct or incorrect and assign the response appropriately, which ends the trial. We always have to clear the event buffer if we’re checking for key presses like this

```python
# get response
thisResp=None
while thisResp==None:
    allKeys=event.waitKeys()
    for thisKey in allKeys:
        if thisKey=='left':
            if targetSide==-1: thisResp = 1 # correct
            else: thisResp = -1 # incorrect
        elif thisKey=='right':
            if targetSide== 1: thisResp = 1 # correct
            else: thisResp = -1 # incorrect
        elif thisKey in ['q', 'escape']:
            core.quit() # abort experiment
            event.clearEvents() # clear other (eg mouse) events - they clog the buffer

Now we must tell the staircase the result of this trial with its addData() method. Then it can work out whether the next trial is an increment or decrement. Also, on each trial (so still within the for-loop) we may as well save the data as a line of text in that .csv file we created earlier.

```
Output your data and clean up

OK! We’re basically done! We’ve reached the end of the for-loop (which occurred because the staircase terminated) which means the trials are over. The next step is to close the text data file and also save the staircase as a binary file (by ‘pickling’ the file in Python speak) which maintains a lot more info than we were saving in the text file.

```python
# staircase has ended
dataFile.close()
staircase.saveAsPickle(fileName)  # special python binary file to save all the info
```

While we’re here, it’s quite nice to give some immediate feedback to the user. Let’s tell them the intensity values at the all the reversals and give them the mean of the last 6. This is an easy way to get an estimate of the threshold, but we might be able to do a better job by trying to reconstruct the psychometric function. To give that a try see the staircase analysis script of Tutorial 3.

Having saved the data you can give your participant some feedback and quit!

```python
# give some output to user in the command line in the output window
print('reversals: ')
print(staircase.reversalIntensities)
approxThreshold = numpy.average(staircase.reversalIntensities[-6:])
print('mean of final 6 reversals = %.3f' % (approxThreshold))

# give some on-screen feedback
feedback1 = visual.TextStim(
    win, pos=[0,+3],
    text='mean of final 6 reversals = %.3f' % (approxThreshold))
feedback1.draw()
fixation.draw()
win.flip()
event.waitKeys()  # wait for participant to respond
win.close()
core.quit()
```

6.2.3 Tutorial 3: Analysing data in Python

You could simply output your data as tab- or comma-separated text files and analyse the data in some spreadsheet package. But the matplotlib library in Python also allows for very neat and simple creation of publication-quality plots.

This script shows you how to use a couple of functions from PsychoPy to open some data files (psychopy.gui.fileOpenDlg()) and create a psychometric function out of some staircase data (psychopy.data.functionFromStaircase()).

Matplotlib is then used to plot the data.

Note: Matplotlib and pylab. Matplotlib is a python library that has similar command syntax to most of the plotting functions in Matlab(tm). It can be imported in different ways; the import pylab line at the beginning of the script is the way to import matplotlib as well as a variety of other scientific tools (that aren’t strictly to do with plotting per se).

```python
from __future__ import print_function
#This analysis script takes one or more staircase datafiles as input
```
#from a GUI. It then plots the staircases on top of each other on
#the left and a combined psychometric function from the same data
#on the right

from psychopy import data, gui, core
from psychopy.tools.filetools import fromFile
import pylab

#Open a dialog box to select files from
files = gui.fileOpenDlg('.
if not files:
    core.quit()

#get the data from all the files
allInten, allResp = [],[]
for thisFileName in files:
    thisDat = fromFile(thisFileName)
    allInten.append( thisDat.intensities )
    allResp.append( thisDat.data )

#plot each staircase
pylab.subplot(121)
colors = 'brgkcmbrgkcm'
lines, names = [],[]
for fileN, thisStair in enumerate(allInten):
    #lines.extend(pylab.plot(thisStair))
    #names = files[fileN]
    pylab.plot(thisStair, label = files[fileN])
    #pylab.legend()

#get combined data
combinedInten, combinedResp, combinedN = \
    data.functionFromStaircase(allInten, allResp, 5)
#fit curve - in this case using a Weibull function
fit = data.FitWeibull(combinedInten, combinedResp, guess=[0.2, 0.5])
smoothInt = pylab.arange(min(combinedInten), max(combinedInten), 0.001)
smoothResp = fit.eval(smoothInt)
thresh = fit.inverse(0.8)
print(thresh)

#plot curve
pylab.subplot(122)
pylab.plot(smoothInt, smoothResp, '-')
pylab.plot([thresh, thresh],[0,0.8], '--'); pylab.plot([0, thresh],
[0.8,0.8], '--')
pylab.title('threshold = %0.3f' %thresh)
#plot points
pylab.plot(combinedInten, combinedResp, 'o')
pylab.ylim([0,1])
pylab.show()
In January 2018 we began a Wellcome Trust grant to make online studies possible from PsychoPy. This is what we call PsychoPy3 - the 3rd major phase of PsychoPy’s development.

The key steps to this are basically to:

• generate a JavaScript experiment ready to run online
• upload it to Pavlovia.org to be launched
• set up your recruitment procedure

Those steps are covered in detail here:

### 7.1 Creating online studies from Builder

PsychoPy can’t export all possible experiments to PsychoJS scripts yet. “Standard” studies using images, text and keyboards will work. Studies with other stimuli, or that use code components, probably won’t.

These are the steps you need to follow to get up and running online:

• Check if your study is fully supported
• Check your experiment settings
• Export the HTML files
• Uploading files to your own server
• Debug your online experiments
• Recruiting participants
• Fetching your data

#### 7.1.1 Check if your study is fully supported

Keep checking the Status of online options to see what is supported. You might want to sign up to the PsychoPy forum and turn on “watching” for the Online Studies category to get updates. That way you’ll know if we update/improve something and you’ll see if other people are having issues.
7.1.2 Check your experiment settings

In your Experiment Settings there is an “Online” tab to control the settings.

Path: When you upload your study to Pavlovia it will expect to find an ‘html’ folder in the root of the repository, so you want to set this up with that in mind. By default the output path will be for a folder called html next to the experiment file. So if that is in the root of the folder you sync online then you’ll be good to go! Usually you would have a folder structure something like this and sync that entire folder with pavlovia.org:

![Folder Structure Example](image)

7.1.3 Export the HTML files

Once you’ve checked your settings you can either go to >File>Export HTML from the Builder view with your experiment open OR press ‘sync’ from the globe icons.

Both of these will generate all the necessary files (HTML and JS) that you need for your study, however sync will also create a project in pavlovia.org
7.1.4 Uploading files to your own server

We really don't recommend this and can only provide limited help if you go this route. If you do want to use your own server:

You will need some way to save the data. PsychoJS can output to either:

- csv files in ../data (i.e. a folder called data next to the html folder). You'll need this to have permissions so that the web server can write to it
- a relational database
- You should make sure your server is using https to encrypt the data you collect from your participants, in keeping with GDPR legislation
- You will need to install the server-side script
- You will need to adapt PsychoPy Builder's output scripts (index.html and the <experimentName>.js) so that the references to lib/ and lib/vendors are pointing to valid library locations (which you will either need to create, or point to original online sources)

7.1.5 Debug your online experiments

This is going to be trickier (for now) than the PsychoPy/Python scripts. The main resources we recommend are:

- The PsychoPy to JS crib sheet a fantastic resource developed by Wakefield Morys-Carter
- The PsychoJS API
- The forum

7.1.6 Common problems

The majority of problems and solutions are outlined at The PsychoPy to JS crib sheet but some common problems include.

“My experiment works locally, but doesn’t run online”*

This means there is a JS error. Likely because of one of the following reasons:

- **You have a code component where you have used something that can’t be translated to JS (e.g. python libraries like ‘numpy’)**
  - *Solution* check if the function you are trying to use has been added to the crib sheet. If it has, implement it as below. If it has not, try searching ‘JS equivalent of X function’ in stack overflow https://stackoverflow.com/, try to implement it as below, if it works share your solution on discourse so that we can add it to the resources, if it does not, post your issue on discourse and tell us what you tried.

- **You have a non-code component, but one of the parameters has been modified to a value that ‘doesn’t make sense’ in JS.**
  - *Solution* look at the PsychoJS API to see what types of variable each parameter of each component takes.

“I have an ‘Unknown Resources’ Error”*

You can control what resources are loaded using the ‘online’ tab in your experiment settings.
7.1.7 Finding the problem

If you encounter an error when getting online and see a red error message box or you are stuck on initializing follow these steps:

1. open the developer tools (outlined on crib sheet).
2. look for the red line that corresponds to your task. In the example below, my task was called ‘selectShapes’ and we can see that there is an error on line 256 of the compiled JS code.
3. Examine what is happening on the line causing the error, either by clicking on the message in developer tools or by opening your.js file (usually automatically created on your local desktop when you synced) in a text editor with line numbering. In this example, I was trying to use a python method dir which cannot be use in JS.

7.1.8 Fixing the problem

We do not recommend making edits to your.js script. Make corrections to your builder .psyexp file

In this example, I would look for the code component where I used the dir() method and remove it I could a) keep my code type as Auto->JS, remove it from the left hand (python side) and it will automatically be removed from the JS side 
b) change my code type to ‘both’ and remove it only from the right hand (JS) side (once this is done, code changes made in python will not be auto translated to JS).

Why do we not recommend making changes to your.js file?

- changes you make in your.js file will not be reflected back in your builder file, it is a one way street.
- it becomes more difficult to sync your experiment with pavlovia from psychopy (since the creation of experiment pages is mainly done through the builder interface)
- You might be comfortable using JS, but will future users? If you want future researchers to use your experiment having a visualisation of the experiment in GUI format may be more accessible to researchers less familiar with JS.
Adding JS functions

If you have a function you want to use, and you find the equivalent on the crib sheet or stack overflow, add an 'initialization' code component to the start of your experiment. Set code type to be 'JS' and copy and paste the function(s) you want there in the 'Begin experiment' tab. These functions will then be available to be called throughout the rest of the task.

7.1.9 Posting the issue on the forum

If you are struggling to debug your error you can always ask for help on the forum. So that the team and community can help, it is really helpful if we can see your files either by making your project public or by adding a member of the PsychoPy team as a member to your project.

Making your task public

To make your task public navigate to your experiment page then select > View code <> > settings > permissions (set to public)
If you cannot make your task public share your problem and what you tried on the discourse and add a member of the team to your project (settings> members>add member - use the user name given by whoever is supporting you)

7.1.10 Activate on Pavlovia

Once your experiment is online you will see your experiment in dashboard> experiment, you can read more about the Experiment page here.

7.1.11 Recruiting participants

Once you’ve uploaded your folder with the correct permissions you can simply provide that as a URL/link to your prospective participants. When they go to this link they’ll see the info dialog box (with the same settings as the one you use in your standard PsychoPy study locally, but a little prettier). That dialog box may show a progress bar while the resources (e.g. image files) are downloading to the local computer. When they’ve finished downloading the ‘OK’ button will be available and the participant can carry on to your study.

Alternatively you may well want to recruit participants using an online service such as prolific you can read how to connect your experiment with prolific here.
7.1.12 Fetching your data

The data are saved in a data folder next to the html file. You should see csv files there that are similar to your PsychoPy standard output files. (There won’t be any psydat files though - that isn’t possible from JavaScript).

You could just download the data folder or, if you’ve set it up to sync with an OSF project then you could simply sync your PsychoPy project with OSF (from the projects menu) and your data will be fetched to your local computer! :-)

7.2 Using Pavlovia.org

Pavlovia n. where behavior is studied

Pavlovia.org is a site created by the PsychoPy team to make it easy to:

• run studies online
• host your experiments securely and easily without knowing about server technologies
• share studies with other scientists with collaborators of publicly (and find public studies shared by others)
• raise awareness of your study
• version control your work (using Git)

Most of the main tasks you will perform with Pavlovia can be carried out either in the PsychoPy application or on the Pavlovia website. Synchronizing your files can also be done with any Git client if you prefer.

7.2.1 Creating an account on Pavlovia

To create and log in to your account on Pavlovia, you will need an active internet connection. If you have not created your account, you can either 1) go to Pavlovia and create your account, or 2) click the login button highlighted in Figure 1, and create an account through the dialog box. Once you have an account on Pavlovia, check to see that you are logged in via Builder by clicking button (4) highlighted below, in Figure 1.

![Figure 1. PsychoPy 3 Builder icons for building and running online studies](image)

111 7.2. Using Pavlovia.org
7.2.2 Creating projects and uploading files

Creating your project repository is your first step to running your experiment from Pavlovia. To create your project, first make sure that you have an internet connection and are logged in to Pavlovia. Once you are logged in create your project repository by syncing your project with the server using button (1) in Figure 1.

A dialog box will appear, informing you that your .psyexp file does not belong to an existing project. Click “Create a project” if you wish to create a project, or click “Cancel” if you wish to return to your experiment in Builder. See Figure 2.

![Figure 2. The dialog that appears when an online project does not exist.](image)

If you clicked the “Create a project” button, another window will appear. This window is designed to collect important metadata about your project, see Figure 3 below.

![Figure 3. Dialog for creating your project on Pavlovia.org](image)
Use this window to add information to store your project on Pavlovia:

Name: This is the name of your project on Pavlovia
Group/Owner: The user or group to upload the project
Local folder: The (local) project path on your computer. Use the Browse button to find your local directory, if required.
Description: Describe your experiment – similar to the readme files used for describing PsychoPy experiments.
Tags (comma separated): The tag will be used to filter and search for experiments by key words.
Public: Tick this box if you would like to make your repository public, for anyone to see.

When you have completed all fields in the Project window, click “Create project on Pavlovia” button to push your experiment up to the online repository. Click “Cancel” if you wish to return to your experiment in Builder.

7.2.3 Viewing your experiment files

After you have uploaded your project to Pavlovia via Builder, you can go and have a look at your project online. To view your project, go to Pavlovia. From the Pavlovia home page, you can explore your own existing projects, or other users public projects that have been made available to all users. To find your study, click the Explore tab on the home page (see Figure 4)
When exploring studies online, you are presented with a series of thumbnail images for all of the projects on Pavlovia. See Figure 5.

From the “Explore” page, you can filter projects by setting the filter buttons to a) Public or Private, B) Active or Inactive, and C) sort by number of forks, name, date and number of stars. The default sorting method is Stars. You can also search for projects using the search tool using key words describing your area of interest, e.g., Stroop, or attention.

When you have found your project, you have several options (see Figure 6).

1) Run your task from the Pavlovia server
2) Activate or deactivate your experiment
3) view your project code and resources on the Pavlovia repository via Gitlab repository.

Figure 6. Projects on Pavlovia
7.2.4 Running your experiment on Pavlovia.org from Builder

If you wish to run your experiment online, in a web-browser, you have two options. You can run your experiment directly from pavlovia.org, as described above, or you can run your experiment directly from Builder. (There is also the option to send your experiment URL – more on that later in Recruitment Pools).

To run your experiment on Pavlovia via Builder, you must first ensure you have a valid internet connection, are logged in, and have created a repository for your project on Pavlovia. Once you have completed these steps, simply click button (2) in the Builder frame, as shown in Figure 1 above.

7.2.5 Searching for experiments from Builder

If you wish to search for your own existing projects on Pavlovia, or other users public projects, you can do this via the Builder interface. To search for a project, click button (3) on the Builder Frame in Figure 1. Following this, a search dialog will appear, see Figure 7. The search dialog presents several options that allow users to search, fork and synchronize projects.

Figure 7. The search dialog in Builder

To search for a project (see Fig 7, Box A), type in search terms in the text box and click the “Search” button to find related projects on Pavlovia. Use the search filters (e.g., “My group”, “Public” etc) above the text box to filter the search output. The output of your search will be listed in the search panel below the search button, where you can select your project of interest.

To fork and sync a project is to take your own copy of a project from Pavlovia (fork) and copy a version to your local desktop or laptop computer (sync). To fork a project, select the local folder to download the project using the “Browse” button, and then click “Sync” when you are ready - (see Fig 7, Box B). You should now have a local copy of the project from Pavlovia ready to run in PsychoPy!

Now you can run your synced project online from Pavlovia!
Chapter 7. Running and sharing studies online
7.3 Manual coding of PsychoJS studies

Note that PsychoJS is very much under development and all parts of the API are subject to change

Some people may want to write a JS script from scratch or convert their PsychoPy Python script into PsychoJS. The PsychoJS library looks much like its PsychoPy (Python) equivalent; it has classes like Window and ImageStim and these have the same attributes. So, from that aspect, things are relatively similar and if you already know your way around a PsychoPy script then reading and tweaking the PsychoJS script should be fairly intuitive.

Obviously there are some syntax changes that you’d need to understand and convert (e.g. JavaScript requires semi-colons between lines and uses {} to indicate code blocks). There are some tools like Jiphy that can help with this. The problem is that the conversion is not as simple as a line-by-line conversion.

There are a few key differences that you need to understand moving from Python code to the equivalent PsychoJS script.

7.3.1 Schedulers

A Python script runs essentially in sequence; when one line of code is called the script waits for that line to finish and then the next lines begins. JavaScript is designed to be asynchronous; all parts of your web page should load at once.

As a result, PsychoJS needed something to control the running order of the different parts of the script (e.g. the trials need to occur one after the other, waiting for the previous one to finish). To do this PsychoJS adds the concept of the Scheduler. For instance, you could think of the Flow in PsychoPy as being a Schedule with various items being added to it. Some of those items, such as trial loops also schedule further events (the individual trials to be run) and these can even be nested: the Flow could schedule some blocks, which could schedule a trials loop, which would schedule each individual trial.

If you export a script from one of your Builder experiments you can examine this to see how it works.

7.3.2 Functions

Some people will be delighted to see that in PsychoJS scripts output by Builder there are functions specifying what should happen at different parts of the experiment (a function to begin the Routine, a function for each frame of the Routine etc.). The essence of the PsychoJS script is that you have any number of these functions and then add them to your scheduler to control the flow of the experiment.

In fact, many experienced programmers might feel that this is the “right” thing to do and that we should change the structure of the Python scripts to match this. The key difference that makes it easy in the JavaScript, but not in the Python version, is that variables in JS are inherently global. When a stimulus is created during the Routine’s initialization function it will still be visible to the each-frame function. In the PsychoPy Python script we would have to use an awful lot of global statements and users would probably have a lot of confusing problems. So, no, we aren’t about to change it unless you have a good solution to that issue.

7.4 Recruiting participants and connecting with online services

Having created your study in Builder, uploaded it to Pavlovia, and activated it to run, you now need to recruit your participants to run the study.

At the simplest level you can get the URL for the study and distribute it to participants manually (e.g. by email or social media). To get the URL to run you can either press the Builder button to “Run online” and then you can select the URL in the resulting browser window that should appear.
PsychoPy can also connect to a range of other online systems as well, however, some of which are helpful in recruiting participants. Below we describe the general approach before describing the specifics for some common systems:

### 7.4.1 Recruiting with Sona Systems

Sorry, the documentation for this hasn’t yet been written.

The features are in place to do this, and the general description of how to make it work are on the page *Recruiting participants and connecting with online services*.

### 7.4.2 Recruiting with Prolific

**Prolific** is a dedicated service designed specifically for behavioural scientists. It aims to provide improved data quality over the likes of Mechanical Turk, with better participant selection and screening, and to provide more ethical pay levels to participants in your study.

As described in the page *Recruiting participants and connecting with online services*, connecting Prolific to PsychoPy is simply a matter of telling Prolific the URL for your study (including parameters to receive the Study ID etc) and then telling PsychoPy the URL to use when the participant completes the study.

Example link to provide to **Prolific** as your study URL (you will need to replace *myUserName* and *myStudyName*):

```
http://run.pavlovia.org/myUserName/myStudyName/index.html?participant={{%PROLIFIC_PID →%}}&study_id={{%STUDY_ID%}}&session={{%SESSION_ID%}}
```

Example link to provide to **PsychoPy** as your completion URL (you will need to change your study ID number):

```
https://app.prolific.co/submissions/complete?cc=T8Z142EG
```

Further details on how to find and set these links and parameters are as follows. See also [Integrating Prolific with your study](#)
Setting the study URL in Prolific

To recruit participants to your PsychoPy study you should see this screen while creating/modifying your study at https://prolific.ac:

![Screenshot of Prolific study creation](image)

Note in the above that I have set the participant, session and study_id for our study using a URL query string. These values will be populated by `Prolific Academic` when participants are sent to the study URL. Prolific will help you to format these correctly if you tick the Include URL Parameters? box which will bring up the following dialog. I've changed the default values that PsychoPy will use to store the variables (e.g. to be participant and session which are the default names for these in PsychoPy):
In each of the boxes in the figure above, you can see the name that Prolific gives to this value (e.g. \textit{PROLIFIC\_PID}) and the name that we want PsychoPy to use to store it (e.g. \textit{participant}).

**Setting the completion URL in PsychoPy**

The last thing you need to do is copy the \textit{Completion URL} from the main control panel above and paste that into the online tab for your PsychoPy \textit{Experiment Settings} as below:
7.4.3 Integrating with Amazon's Mechanical Turk (MTurk)

Sorry, the documentation for this hasn’t yet been written.

The features are in place to do this, and the general description of how to make it work are on the page Recruiting participants and connecting with online services.

**Warning:** Using Mechanical Turk (MTurk)

Note that Mechanical Turk was not designed with behavioural science in mind, but as a way for Amazon to test computing technologies in cases where a human was needed to push the buttons. They don’t particularly care about the quality of this behavioral data as a result, nor about the ethics of the participants involved.

To get better quality data, and to run studies that your local ethical review board is less likely to be concerned about, then we would suggest you use a dedicated service like Prolific Academic instead.

7.4.4 Daisy-chaining with Qualtrics

Sorry, the documentation for this hasn’t yet been written.

The features are in place to do this, and the general description of how to make it work are on the page Recruiting participants and connecting with online services.

7.4.5 The general principle

All the systems below use the same general principle to connect the different services:

1. the recruiting system needs the URL of your study to send participants there. It probably needs to add the participant ID to that URL so that your study can store that information and (potentially) send it back to the recruiting system
2. at the end of the study the participant should be redirected back to the recruiting system so they can be credited with completing your task

Step 1. is obviously fairly easy if you know how to use the recruiting system. There will be a place somewhere in that system for you to enter the URL of the study being run. The key part is how to provide and store the participant/session ID, as described below.

7.4.6 Passing in a participant/session ID

PsychoPy experiments bring up a dialog box at the start of the study to collect information about a run, which usually includes information about the participant. You can adjust the fields of that box in the Experiment Settings dialog box but usually there is a **participant** field (and we recommend you keep that!)

However, any variables can be passed to the experiment using the URL instead of the dialog box, and this is how you would typically pass the participant ID to your study. This is done by using “Query strings” which are a common part of online web addresses.

If your experiment has the address:

https://run.pavlovia.org/yourUsername/yourStudyName/index.html

then this URL will run the same study but with the **participant** variable set to be 10101010:
and if you want two variables to be set then you can use ? for the first and & for each subsequent. For instance this would set a participant as well as a group variable:

```
https://run.pavlovia.org/yourUsername/yourStudyName/index.html?participant=10101010&group=A
```

If you want to use that variable within your study you can do so using `expInfo`. For instance you could set a thank you message with this JavaScript code in your study:

```
msg = "Thanks, you're done. Your ID is " + expInfo['participant'];
```

## 7.4.7 Redirecting at the end of the study

This is really simple. In the Experiment Settings dialog again you can select the **Online** tab and that has a setting to provide a link for completed and failed-to-complete participants:

![Experiment Settings dialog](image)

## 7.5 Counterbalancing online

If you are manually recruiting your participants (i.e. sending our your experiment URL to a unique population or group) the methods described in *Blocks of trials and counterbalancing* will also work online, and can be used in the same way. However, if you have your experiment URL advertised on a recruitment website, it could be that 10s or hundreds of participants click your link. Manually assigning participants and keeping track of participant groupings in these situations is going to be difficult. So, what do we do?

At the moment PsychoPy and Pavlovia doesn't have an internal method for keeping track of how many participants have already completed your task (and this is needed for counterbalancing). However, some core contributers have developed
some excellent tools to help out, in particular this tool developed by Wakefield Morys Carter that generates sequential participant IDs for your task. Although not a counterbalance tool per se, we can use this to assign out participants to specific groups.

Add a code component to the beginning of your task that looks something like this:

Here we are checking if your participant ID is divisible by 2 (i.e. odd of even) and creating the variable ‘group’ using that. We would then use the same methods outlined previously in *Blocks of trials and counterbalancing* except this time we replace any instance of `expInfo['group']` with `group`

So the conditions files used is selected based on the participant ID!

but you should also be aware of the following:

#### 7.6 Caveats and cautions

The first caution to be aware of here is that PsychoJS was only written in 2016. It hasn’t been widely battle-tested and almost certainly has some rough edges. Use it carefully and check your study does what you expect.

For an in-depth examination of the pros and cons of running studies online (including a consideration of timing issues) see [Woods et al (2015)](https://peerj.com/articles/1058/)

#### 7.6.1 Differences between PsychoPy and PsychoJS studies

Most of the issues below affect your study if you have additional code components inserted into the study and do not affect pure Builder-based designs.

- Builder does not convert your Python code into JavaScript. It writes the JavaScript from scratch from the experiment logic. If you use code in a Builder Component then that code will need to be valid JavaScript code. We hope, in the future, to perform rudimentary automated conversion. Even then, that will only convert the syntax but will not be able to find equivalent function names
- Resources for your study (images, conditions files etc.) must be in the *resources* folder next to the html file that PsychoPy outputs. In most cases PsychoPy can find these and include them automatically but if your study
7.6.2 Timing expectations

In general timing of web-based experiments will be poorer than locally-run studies. That said, we think PsychoJS will have timing about as good as it can be! What are the specific considerations?

**Variable internet connection:** Surprisingly no, this isn’t one to worry about! PsychoJS will make sure that the script and all the resources for your study (image files etc) are downloaded to the computer beforehand. On a slow internet connection it may take longer for your study to start but the performance won’t be limited by that while it runs. Happy days!

**Visual stimuli:** PsychoJS is using WebGL (high performance web rendering using advanced graphics card features) and we have confirmed that PsychoJS is able to run with frame-precise timing. That means, if you ask for a stimulus to last for, say, 6 frames then you will get exactly 100 ms of stimulus presentation.

**Response timing:** Again this won’t be affected by your internet connection (because the keypresses are being time-stamped locally, at your computer, not by the web server). The major problem, as with any software, is that keyboards have long and variable latencies (10-30 ms typically). On a local lab-based setup you can get around this by using custom hardware (a button box) but this obviously isn’t possible when your user is anywhere in the world!

7.6.3 Supported browsers

We’d recommend running on an updated browser but pretty much any modern browser (released since roughly 2011) should be able to run these studies. It needs to support HTML5/canvas and ideally it would support WebGL (if not Canvas then will used but that is less efficient).

Specifically, these support Canvas (minimum requirement):

- Firefox 10+ (released 2012)
- Chrome 11+ (2011)
- Safari 2.0+ (2005)
- Opera 12+ (2011)
- Internet Explorer 9+ (released in 2011) but we really recommend you avoid it!

Browsers supporting WebGL (hardware-accelerated graphics in the browser):

- Firefox 15+ (2012)
- Chrome 11+ (2011)
- Safari 5.1+ (2011/12?)
- Opera 19+
- Microsoft Edge
- IE 11+ (2013)
## 7.7 Status of online options

The table below shows you the current state of play with capabilities of online studies.

<table>
<thead>
<tr>
<th>Inputs</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Keyboard</td>
<td>Done</td>
</tr>
<tr>
<td>Mic</td>
<td>Not done (but could be)</td>
</tr>
<tr>
<td>Mouse</td>
<td>Done</td>
</tr>
<tr>
<td>Webcam</td>
<td>Not done (but could be)</td>
</tr>
<tr>
<td>Rating Scales (Slider)</td>
<td>Done</td>
</tr>
<tr>
<td>Free text (and similar)</td>
<td>Not done (but could be)</td>
</tr>
<tr>
<td>Multi-touch devices</td>
<td>Not done (but could be)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stimuli</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Image</td>
<td>Done</td>
</tr>
<tr>
<td>Noise</td>
<td>Not done (but could be)</td>
</tr>
<tr>
<td>Text</td>
<td>Done</td>
</tr>
<tr>
<td>Gratings</td>
<td>Not done (but could be)</td>
</tr>
<tr>
<td>Sounds</td>
<td>Done</td>
</tr>
<tr>
<td>Apertures</td>
<td>Not done (but could be)</td>
</tr>
<tr>
<td>Movies</td>
<td>Done</td>
</tr>
<tr>
<td>Dots (RDKs)</td>
<td>Not done (but could be)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Data</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CSV files</td>
<td>Done</td>
</tr>
<tr>
<td>XLSX files</td>
<td>Not done (but could be)</td>
</tr>
<tr>
<td>MongoDB database</td>
<td>Done</td>
</tr>
<tr>
<td>Log files</td>
<td>Not done (but could be)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Logic</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Loops (including nesting)</td>
<td>Done</td>
</tr>
<tr>
<td>Staircases</td>
<td>Not done (but could be)</td>
</tr>
<tr>
<td>Randomization</td>
<td>Done</td>
</tr>
<tr>
<td>Code Components</td>
<td>Not done (but could be)</td>
</tr>
<tr>
<td>Conditions files</td>
<td>Not done (but could be)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Precision</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Frame-by-frame timing</td>
<td>Done</td>
</tr>
<tr>
<td>callOnFlip</td>
<td>Not done (but could be)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>External tools</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Synchronize with Pavlovia.org</td>
<td>Done</td>
</tr>
<tr>
<td>Sona Systems</td>
<td>Not done (but could be)</td>
</tr>
<tr>
<td>Mechanical Turk</td>
<td>Done</td>
</tr>
<tr>
<td>Prolific</td>
<td>Not done (but could be)</td>
</tr>
</tbody>
</table>

Anything else we should add to the list above?
7.8 How does it work?

The first stage of this is that there is now a JavaScript library, PsychoJS, that mirrors the PsychoPy Python library classes and functions.

PsychoPy Builder is effectively just writing a script for you based on the visual representation of your study so the new feature is for it simply to write a html/JavaScript/PsychoJS page instead.

Modern browsers are remarkably powerful. Most browsers released since 2011 have allowed HTML5 which supports more flexible rendering of web pages (images and text can be positioned precisely enough to run “proper” behavioural experiments). Since 2013 most have supported WebGL. That allows graphics to be rendered really quickly using “hardware acceleration” on your graphics card. The result is rich pages that can be updated very rapidly and can be forced to sync to screen refresh, which is critical for stimulus timing.

All this means we can do great things with online experiments that actually have good temporal precision!

The way it works is that you have a web page containing JavaScript (generated by PsychoPy Builder). You upload that to a web server. The participant of your study uses their web browser to visit the page you’ve created with a standard URL you send them.

Now, JavaScript executes on their computer (as opposed to scripts like PHP that operate on the server and aren’t directly visible to the viewer/browser). In this case the PsychoJS script will present a dialog box at the start of the study to get the participant ID and any other basic information you need. While that dialog box is presented the script will be downloading all your stimuli and files to the local computer and storing them in memory. When all the necessary files are downloaded the participant can press “OK” and the experiment will start.

The experiment supports all the standard timing aspects of any PsychoPy Builder experiment; you can specify your stimuli in terms of time presented or number of screen refreshes etc (and the actual refresh rate of your participant’s computer will be stored in your data file). When it’s finished it saves the data into a comma-separated-value (CSV) file in the “data” folder on the web server. This looks very much like the standard CSV outputs of your same PsychoPy experiment run locally.

Not all components are currently supported. Keep an eye on the Status of online options page to see what objects you can use already.

7.8.1 How does this compare with jsPsych?

In jsPsych you use one of the pre-programmed “types” of trial (like single stimulus or 2-alternative-forced-choice) and you have rather little flexibility over how that gets conducted. If you wanted to alter the positioning of the stimuli, for instance, in a 2-alternative-force-choice task or you wanted a stimulus to change in time (appear gradually or move location) then you would need to write a new trial “type” using raw javascript.

PsychoPy, by comparison, is designed to give you total flexibility. You decide what constitutes a “trial” and how things should operate in time. We think that control is very important to creating a wide range of studies.
8.1 psychopy.core - basic functions (clocks etc.)

Basic functions, including timing, rush (imported), quit

```python

class psychopy.core.Clock
    A convenient class to keep track of time in your experiments. You can have as many independent clocks as you like (e.g. one to time responses, one to keep track of stimuli…)

    This clock is identical to the MonotonicClock except that it can also be reset to 0 or another value at any point.

    add(t)
        Add more time to the clock’s ‘start’ time (t0).

        Note that, by adding time to t0, you make the current time appear less. Can have the effect that getTime() returns a negative number that will gradually count back up to zero.

        e.g.:

        ```python
timer = core.Clock()
timer.add(5)
while timer.getTime() < 0:
    # do something
```

    reset(newT=0.0)
        Reset the time on the clock. With no args time will be set to zero. If a float is received this will be the new time on the clock

    class psychopy.core.CountdownTimer(start=0)
        Similar to a Clock except that time counts down from the time of last reset

        Typical usage:

        ```python
timer = core.CountdownTimer(5)
while timer.getTime() > 0:
    # after 5s will become negative
    # do stuff
```

    getTime()
        Returns the current time left on this timer in secs (sub-ms precision)
```
```
reset(t=None)
Reset the time on the clock. With no args, time will be set to the time used for last reset (or start time if no previous resets). If a float is received, this will be the new time on the clock.

class psychopy.core.MonotonicClock(start_time=None)
A convenient class to keep track of time in your experiments using a sub-millisecond timer.

Unlike the Clock this cannot be reset to arbitrary times. For this clock t=0 always represents the time that the clock was created.

Don’t confuse this class with core.monotonicClock which is an instance of it that got created when PsychoPy.core was imported. That clock instance is deliberately designed always to return the time since the start of the study.

Version Notes: This class was added in PsychoPy 1.77.00

g getLastResetTime()
Returns the current offset being applied to the high resolution timebase used by Clock.

getTime(applyZero=True)
Returns the current time on this clock in secs (sub-ms precision).

If applying zero then this will be the time since the clock was created (typically the beginning of the script).

If not applying zero then it is whatever the underlying clock uses as its base time but that is system dependent.

e.g. can be time since reboot, time since Unix Epoch etc

class psychopy.core.StaticPeriod(screenHz=None, win=None, name='StaticPeriod')
A class to help insert a timing period that includes code to be run.

Typical usage:
```
fixation.draw()
win.flip()
ISI = StaticPeriod(screenHz=60)
ISI.start(0.5)  # start a period of 0.5s
stim.image = 'largeFile.bmp'  # could take some time
ISI.complete()  # finish the 0.5s, taking into account one 60Hz frame

stim.draw()
win.flip()  # the period takes into account the next frame flip
# time should now be at exactly 0.5s later than when ISI.start()
# was called
```

Parameters

- **screenHz** – the frame rate of the monitor (leave as None if you don’t want this accounted for)
- **win** – if a visual.Window is given then StaticPeriod will also pause/restart frame interval recording
- **name** – give this StaticPeriod a name for more informative logging messages

```
complete()
Completes the period, using up whatever time is remaining with a call to wait()

    Returns  1 for success, 0 for fail (the period overran)
```

```
start(duration)
Start the period. If this is called a second time, the timer will be reset and starts again

    Parameters duration – The duration of the period, in seconds.
```
psychopy.core.getAbsTime()
Return unix time (i.e., whole seconds elapsed since Jan 1, 1970).

This uses the same clock-base as the other timing features, like getTime(). The time (in seconds) ignores the
time-zone (like time.time() on linux). To take the timezone into account, use int(time.mktime(time.gmtime())).

Absolute times in seconds are especially useful to add to generated file names for being unique, informative (= a
meaningful time stamp), and because the resulting files will always sort as expected when sorted in chronological,
alphabetical, or numerical order, regardless of locale and so on.

Version Notes: This method was added in PsychoPy 1.77.00

psychopy.core.getTime(applyZero=True)
Get the current time since psychopy.core was loaded.

Version Notes: Note that prior to PsychoPy 1.77.00 the behaviour of getTime() was platform dependent (on OSX
and linux it was equivalent to psychopy.core.getAbsTime() whereas on windows it returned time since
loading of the module, as now)

psychopy.core.wait(secs, hogCPUperiod=0.2)
Wait for a given time period.

If secs=10 and hogCPU=0.2 then for 9.8s python's time.sleep function will be used, which is not especially precise,
but allows the cpu to perform housekeeping. In the final hogCPUperiod the more precise method of constantly
polling the clock is used for greater precision.

If you want to obtain key-presses during the wait, be sure to use pyglet and to hogCPU for the entire time, and then
call psychopy.event.getKeys() after calling wait()

If you want to suppress checking for pyglet events during the wait, do this once:

    core.checkPygletDuringWait = False

and from then on you can do:

    core.wait(sec)

This will preserve terminal-window focus during command line usage.

8.2 psychopy.visual - many visual stimuli

8.2.1 Aperture

class psychopy.visual.Aperture(*args, **kwargs)
Restrict a stimulus visibility area to a basic shape or list of vertices.

When enabled, any drawing commands will only operate on pixels within the Aperture. Once disabled, subsequent
draw operations affect the whole screen as usual.

If shape is ‘square’ or ‘triangle’ then that is what will be used If shape is ‘circle’ or None then a polygon with nVerts
will be used

    (120 for a rough circle)

If shape is a list or numpy array (Nx2) then it will be used directly as the vertices to a ShapeStim

If shape is a filename then it will be used to load and image as a ImageStim. Note that transparent parts
in the image (e.g. in a PNG file) will not be included in the mask shape. The color of the image will be
ignored.
See demos/stimuli/aperture.py for example usage

Author 2011, Yuri Spitsyn 2011, Jon Peirce added units options,
Jeremy Gray added shape & orientation
2014, Jeremy Gray added .contains() option 2015, Thomas Emmerling added ImageStim option

8.2.2 BoundingBox

Attributes

BoundingBox([extents])  Class for representing object bounding boxes.

Details

class psychopy.visual.BoundingBox (extents=None)
Class for representing object bounding boxes.

A bounding box is a construct which represents a 3D rectangular volume about some pose, defined by its minimum and maximum extents in the reference frame of the pose. The axes of the bounding box are aligned to the axes of the world or the associated pose.

Bounding boxes are primarily used for visibility testing; to determine if the extents of an object associated with a pose (eg. the vertices of a model) falls completely outside of the viewing frustum. If so, the model can be culled during rendering to avoid wasting CPU/GPU resources on objects not visible to the viewer.

  _computeCorners ()
  Compute the corners of the bounding box.
  These values are cached to speed up computations if extents hasn’t been updated.

  clear ()
  Clear a bounding box, invalidating it.

  property extents

  fit (verts)
  Fit the bounding box to vertices.

  property isValid
  True if the bounding box is valid.

8.2.3 BoxStim

Attributes

BoxStim(win[, size, flipFaces, pos, ori, …])  Class for drawing 3D boxes.
class psychopy.visual.BoxStim(win, size=(0.5, 0.5, 0.5), flipFaces=False, pos=(0.0, 0.0, 0.0), ori=(0.0, 0.0, 0.0, 1.0), color=(0.0, 0.0, 0.0), colorSpace='rgb', contrast=1.0, opacity=1.0, useMaterial=None, useShaders=False, textureScale=None, name='', autoLog=True)

Class for drawing 3D boxes.

Draws a rectangular box with dimensions specified by size (length, width, height) in scene units.

Calling the draw method will render the box to the current buffer. The render target (FBO or back buffer) must have a depth buffer attached to it for the object to be rendered correctly. Shading is used if the current window has light sources defined and lighting is enabled (by setting useLights=True before drawing the stimulus).

**Warning:** This class is experimental and may result in undefined behavior.

**Parameters**

- **win** (*psychopy.visual.Window*) – Window this stimulus is associated with. Stimuli cannot be shared across windows unless they share the same context.
- **size** (*tuple or float*) – Dimensions of the mesh. If a single value is specified, the box will be a cube. Provide a tuple of floats to specify the width, length, and height of the box (eg. size=(0.2, 1.3, 2.1)) in scene units.
- **flipFaces** (*bool*, optional) – If True, normals and face windings will be set to point inward towards the center of the box. Texture coordinates will remain the same. Default is False.
- **pos** (*array_like*) – Position vector [x, y, z] for the origin of the rigid body.
- **ori** (*array_like*) – Orientation quaternion [x, y, z, w] where x, y, z are imaginary and w is real. If you prefer specifying rotations in axis-angle format, call setOriAxisAngle after initialization.
- **useMaterial** (*PhongMaterial*, optional) – Material to use. The material can be configured by accessing the material attribute after initialization. If not material is specified, the diffuse and ambient color of the shape will track the current color specified by glColor. color : array_like Diffuse and ambient color of the stimulus if useMaterial is not specified. Values are with respect to colorSpace.
- **colorSpace** (*str*) – Colorspace of color to use.
- **contrast** (*float*) – Contrast of the stimulus, value modulates the color.
- **opacity** (*float*) – Opacity of the stimulus ranging from 0.0 to 1.0. Note that transparent objects look best when rendered from farthest to nearest.
- **textureScale** (*array_like or float*, optional) – Scaling factors for texture coordinates (sx, sy). By default, a factor of 1 will have the entire texture cover the surface of the mesh. If a single number is provided, the texture will be scaled uniformly.
- **name** (*str*) – Name of this object for logging purposes.
- **autoLog** (*bool*) – Enable automatic logging on attribute changes.

_createVAO (vertices, textureCoords, normals, faces)

Create a vertex array object for handling vertex attribute data.
PsychoPy - Psychology software for Python, Release 2020.2.10

_getDesiredRGB (rgb, colorSpace, contrast)
Convert color to RGB while adding contrast. Requires self.rgb, self.colorSpace and self.contrast

-selectWindow (win)
Switch drawing to the specified window. Calls the window’s _setCurrent() method which handles the switch.

-updateList ()
The user shouldn’t need this method since it gets called after every call to .set() Chooses between using and not using shaders each call.

color
Color of the stimulus
Value should be one of:

- string: to specify a Colors by name. Any of the standard html/X11 color names <http://www.w3schools.com/html/html_colornames.asp> can be used.
- Colors by hex value
- numerically: (scalar or triplet) for DKL, RGB or other Color spaces. For these, operations are supported.

When color is specified using numbers, it is interpreted with respect to the stimulus’ current colorSpace. If color is given as a single value (scalar) then this will be applied to all 3 channels.

Examples
For whatever stim you have:

```
stim.color = 'white'
stim.color = 'RoyalBlue'  # (the case is actually ignored)
stim.color = '#DDA0DD'  # DDA0DD is hexadecimal for plum
stim.color = [1.0, -1.0, -1.0]  # if stim.colorSpace='rgb':
    # a red color in rgb space
stim.color = [0.0, 45.0, 1.0]  # if stim.colorSpace='dkl':
    # DKL space with elev=0, azimuth=45
stim.color = [0, 0, 255]  # if stim.colorSpace='rgb255':
    # a blue stimulus using rgb255 space
stim.color = 255  # interpreted as (255, 255, 255)
    # which is white in rgb255.
```

Operations work as normal for all numeric colorSpaces (e.g. ‘rgb’, ‘hsv’ and ‘rgb255’) but not for strings, like named and hex. For example, assuming that colorSpace='rgb':

```
stim.color += [1, 1, 1]  # increment all guns by 1 value
stim.color *= -1  # multiply the color by -1 (which in this
    # space inverts the contrast)
stim.color *= [0.5, 0, 1]  # decrease red, remove green, keep blue
```

You can use setColor if you want to set color and colorSpace in one line. These two are equivalent:

```
stim.setColor((0, 128, 255), 'rgb255')
# ... is equivalent to
stim.colorSpace = 'rgb255'
stim.color = (0, 128, 255)
```

colorSpace
The name of the color space currently being used
Valueshouldbe: astringsorNone

Forstringsandhexvaluesthisisnotneeded. If None the default colorSpace for the stimulus is used (defined
during initialisation).

Please notethatchanging colorSpace does not change stimulus parameters. Thus you usually want to specify
colorSpacebefore setting the color. Example:

```python
# A light green text
stim = visual.TextStim(win, 'Color me!',
                      color=(0, 1, 0), colorSpace='rgb')

# An almost-black text
stim.colorSpace = 'rgb255'

# Make it light green again
stim.color = (128, 255, 128)
```

**contrast**

A value that is simply multiplied by the color

**Value should be:** a float between -1 (negative) and 1 (unchanged). Operations supported.

Set the contrast of the stimulus, i.e. scales how far the stimulus deviates from the middle grey. You can also
use the stimulus opacity to control contrast, but that cannot be negative.

Examples:

```python
stim.contrast = 1.0   # unchanged contrast
stim.contrast = 0.5   # decrease contrast
stim.contrast = 0.0   # uniform, no contrast
stim.contrast = -0.5  # slightly inverted
stim.contrast = -1.0  # totally inverted
```

Setting contrast outside range -1 to 1 is permitted, but may produce strange results if color values exceeds the
monitor limits.:.

```python
stim.contrast = 1.2   # increases contrast
stim.contrast = -1.2  # inverts with increased contrast
```

**draw (win=None)**

Draw the stimulus.

This should work for stimuli using a single VAO and material. More complex stimuli with multiple materials
should override this method to correctly handle that case.

**Parameters win (~psychopy.visual.Window) — Window this stimulus is associated with. Stimuli**
**cannot be shared across windows unless they share the same context.**

**getOri()**

**getOriAxisAngle (degrees=True)**

Get the axis and angle of rotation for the 3D stimulus. Converts the orientation defined by the ori quaternion
to and axis-angle representation.

**Parameters degrees (bool, optional) — Specify True if angle is in degrees, or else it**
**will be treated as radians. Default is True.**

**Returns** Axis [rx, ry, rz] and angle.

**Return type** tuple
getPos()

getAddressIntersectBounds(rayOrig, rayDir)
    Get the point which a ray intersects the bounding box of this mesh.

    Parameters
    • rayOrig (array_like) – Origin of the ray in space [x, y, z].
    • rayDir (array_like) – Direction vector of the ray [x, y, z], should be normalized.

    Returns Coordinate in world space of the intersection and distance in scene units from rayOrig.
    Returns None if there is no intersection.

    Return type tuple

isVisible()
    Check if the object is visible to the observer.
    Test if a pose’s bounding box or position falls outside of an eye’s view frustum.
    Poses can be assigned bounding boxes which enclose any 3D models associated with them. A model is not
    visible if all the corners of the bounding box fall outside the viewing frustum. Therefore any primitives (i.e.
    triangles) associated with the pose can be culled during rendering to reduce CPU/GPU workload.

    Returns True if the object’s bounding box is visible.

    Return type bool

Examples

You can avoid running draw commands if the object is not visible by doing a visibility test first:

```python
if myStim.isVisible():
    myStim.draw()
```

**property ori**
    Orientation quaternion (X, Y, Z, W).

**property pos**
    Position vector (X, Y, Z).

**setColor** (color, colorSpace=None, operation='', log=None)
    Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the
    log message and/or set colorSpace simultaneously.

**setContrast** (newContrast, operation='', log=None)
    Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the
    log message

**setFillColor** (color, colorSpace=None, operation='', log=None)

**setOri** (ori)

**setOriAxisAngle** (axis, angle, degrees=True)
    Set the orientation of the 3D stimulus using an axis and angle. This sets the quaternion at ori.

    Parameters
    • axis (array_like) – Axis of rotation [rx, ry, rz].
    • angle (float) – Angle of rotation.
• **degrees** *(bool, optional)* – Specify True if *angle* is in degrees, or else it will be treated as radians. Default is True.

**setPos** *(pos)*

**setUseShaders** *(value=True, log=None)*

Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message.

**property thePose**

The pose of the rigid body. This is a class which has *pos* and *ori* attributes.

**units**

None, ‘norm’, ‘cm’, ‘deg’, ‘degFlat’, ‘degFlatPos’, or ‘pix’

If None then the current units of the Window will be used. See *Units for the window and stimuli* for explanation of other options.

Note that when you change units, you don’t change the stimulus parameters and it is likely to change appearance. Example:

```python
# This stimulus is 20% wide and 50% tall with respect to window
stim = visual.PatchStim(win, units='norm', size=(0.2, 0.5))

# This stimulus is 0.2 degrees wide and 0.5 degrees tall.
stim.units = 'deg'
```

**useShaders**

Should shaders be used to render the stimulus (typically leave as True)

If the system support the use of OpenGL shader language then leaving this set to True is highly recommended. If shaders cannot be used then various operations will be slower (notably, changes to stimulus color or contrast)

**win**

*The Window object in which the* stimulus will be rendered by default. (required)

Example, drawing same stimulus in two different windows and display simultaneously. Assuming that you have two windows and a stimulus (win1, win2 and stim):

```python
stim.win = win1  # stimulus will be drawn in win1
stim.draw()     # stimulus is now drawn to win1
stim.win = win2  # stimulus will be drawn in win2
stim.draw()     # it is now drawn in win2
win1.flip(waitBlanking=False)  # do not wait for next # monitor update
win2.flip()     # wait for vertical blanking.
```

Note that this just changes **default** window for stimulus. You could also specify window-to-draw-to when drawing::

```python
stim.draw(win1)
stim.draw(win2)
```
8.2.4 **BufferImageStim**

**Attributes**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>BufferImageStim(win[, buffer, rect, ...])</code></td>
<td>Take a “screen-shot”, save as an ImageStim (RBGA object).</td>
</tr>
<tr>
<td><code>BufferImageStim.win</code></td>
<td>The Window object in which the</td>
</tr>
<tr>
<td><code>BufferImageStim.mask</code></td>
<td>The alpha mask that can be used to control the outer shape of the stimulus</td>
</tr>
<tr>
<td><code>BufferImageStim.units</code></td>
<td>None, 'norm', 'cm', 'deg', 'degFlat', 'degFlatPos', or 'pix'</td>
</tr>
<tr>
<td><code>BufferImageStim.pos</code></td>
<td>The position of the center of the stimulus in the stimulus units</td>
</tr>
<tr>
<td><code>BufferImageStim.ori</code></td>
<td>The orientation of the stimulus (in degrees).</td>
</tr>
<tr>
<td><code>BufferImageStim.size</code></td>
<td>The size (width, height) of the stimulus in the stimulus units</td>
</tr>
<tr>
<td><code>BufferImageStim.contrast</code></td>
<td>A value that is simply multiplied by the color</td>
</tr>
<tr>
<td><code>BufferImageStim.color</code></td>
<td>Color of the stimulus</td>
</tr>
<tr>
<td><code>BufferImageStim.colorSpace</code></td>
<td>The name of the color space currently being used</td>
</tr>
<tr>
<td><code>BufferImageStim.opacity</code></td>
<td>Determines how visible the stimulus is relative to background</td>
</tr>
<tr>
<td><code>BufferImageStim.interpolate</code></td>
<td>Whether to interpolate (linearly) the texture in the stimulus</td>
</tr>
<tr>
<td><code>BufferImageStim.name</code></td>
<td>String or None.</td>
</tr>
<tr>
<td><code>BufferImageStim.autoLog</code></td>
<td>Whether every change in this stimulus should be auto logged.</td>
</tr>
<tr>
<td><code>BufferImageStim.draw([win])</code></td>
<td>Draws the BufferImage on the screen, similar to <code>ImageStim.draw()</code>.</td>
</tr>
<tr>
<td><code>BufferImageStim.autoDraw</code></td>
<td>Determines whether the stimulus should be automatically drawn on every frame flip.</td>
</tr>
</tbody>
</table>

**Details**

```python
class psychopy.visual.BufferImageStim(win, buffer='back', rect=(-1, 1, 1, -1), sqPower2=False, stim=(), interpolate=True, flipHoriz=False, flipVert=False, mask='None', pos=(0, 0), name=None, autoLog=None)
```

Take a “screen-shot”, save as an ImageStim (RBGA object).

The screen-shot is a single collage image composed of static elements that you can treat as being a single stimulus. The screen-shot can be of the visible screen (front buffer) or hidden (back buffer).

`BufferImageStim` aims to provide fast rendering, while still allowing dynamic orientation, position, and opacity. It’s fast to draw but slower to init (same as an ImageStim).

You specify the part of the screen to capture (in norm units), and optionally the stimuli themselves (as a list of items to be drawn). You get a screenshot of those pixels. If your OpenGL does not support arbitrary sizes, the image will be larger, using square powers of two if needed, with the excess image being invisible (using alpha). The aim is to preserve the buffer contents as rendered.

Checks for OpenGL 2.1+, or uses square-power-of-2 images.

**Example:**
# define lots of stimuli, make a list:
mySimpleImageStim = ...  
myTextStim = ...  
stimList = [mySimpleImageStim, myTextStim]

# draw stim list items & capture (slow; see EXP log for times):
screenshot = visual.BufferImageStim(myWin, stim=stimList)

# render to screen (very fast, except for the first draw):
while <conditions>:
    screenshot.draw()  # fast; can vary .ori, .pos, .opacity  
    other_stuff.draw()  # dynamic  
    myWin.flip()

See coder Demos > stimuli > bufferImageStim.py for a demo, with timing stats.

Author

- 2010 Jeremy Gray, with on-going fixes

Parameters

- **buffer**: the screen buffer to capture from, default is ‘back’ (hidden). ‘front’ is the buffer in view after win.flip()
- **rect**: a list of edges [left, top, right, bottom] defining a screen rectangle which is the area to capture from the screen, given in norm units. default is fullscreen: [-1, 1, 1, -1]
- **stim**: a list of item(s) to be drawn to the back buffer (in order). The back buffer is first cleared (without the win being flip()ed), then stim items are drawn, and finally the buffer (or part of it) is captured. Each item needs to have its own .draw() method, and have the same window as win.
- **interpolate**: whether to use interpolation (default = True, generally good, especially if you change the orientation)
- **sqPower2**:  
  - False (default) = use rect for size if OpenGL = 2.1+  
  - True = use square, power-of-two image sizes
- **flipHoriz**: horizontally flip (mirror) the captured image, default = False
- **flipVert**: vertically flip (mirror) the captured image; default = False

_calcPosRendered()  
DEPRECATED in 1.80.00. This functionality is now handled by _updateVertices() and verticesPix.

_calSizeRendered()  
DEPRECATED in 1.80.00. This functionality is now handled by _updateVertices() and verticesPix

_createTexture(tex, id, pixFormat, stim, res=128, maskParams=None, forcePOW2=True, dataType=None, wrapping=True)

Params

- **id**: is the texture ID
- **pixFormat**: GL.GL_ALPHA, GL.GL_RGB
- **useShaders**: bool
- **interpolate**: bool (determines whether texture will use GL_LINEAR or GL_NEAREST
res: the resolution of the texture (unless a bitmap image is used)

dataType: None, GL.GL_UNSIGNED_BYTE, GL.FLOAT. Only affects image files
(numpy arrays will be float)

For grating stimuli (anything that needs multiple cycles) forcePOW2 should be set to be True. Otherwise the
wrapping of the texture will not work.

_getDesiredRGB (rgb, colorSpace, contrast)
Convert color to RGB while adding contrast. Requires self.rgb, self.colorSpace and self.contrast

_getPolyAsRendered ()
DEPRECATED. Return a list of vertices as rendered.

_selectWindow (win)
Switch drawing to the specified window. Calls the window’s _setCurrent() method which handles the switch.

_set (attrib, val=",", log=None)
DEPRECATED since 1.80.04 + 1. Use setAttribute() and val2array() instead.

_updateList ()
The user shouldn’t need this method since it gets called after every call to .set() Chooses between using and
not using shaders each call.

_updateListNoShaders ()
The user shouldn’t need this method since it gets called after every call to .set() Basically it updates the
OpenGL representation of your stimulus if some parameter of the stimulus changes. Call it if you change a
property manually rather than using the .set() command.

_updateListShaders ()
The user shouldn’t need this method since it gets called after every call to .set() Basically it updates the
OpenGL representation of your stimulus if some parameter of the stimulus changes. Call it if you change a
property manually rather than using the .set() command

_updateVertices ()
Sets Stim.verticesPix and _borderPix from pos, size, ori, flipVert, flipHoriz

autoDraw
Determines whether the stimulus should be automatically drawn on every frame flip.

Value should be: True or False. You do NOT need to set this on every frame flip!

autoLog
Whether every change in this stimulus should be auto logged.

Value should be: True or False. Set to False if your stimulus is updating frequently (e.g. updating its position
every

frame) and you want to avoid swamping the log file with

messages that aren’t likely to be useful.

clearTextures ()
Clear all textures associated with the stimulus.

As of v1.61.00 this is called automatically during garbage collection of your stimulus, so doesn’t need calling
explicitly by the user.

color
Color of the stimulus

Value should be one of:
• string: to specify a *Colors by name.* Any of the standard html/X11 *color names*<http://www.w3schools.com/html/html_colornames.asp> can be used.

• *Colors by hex value*

• **numerically:** (scalar or triplet) for DKL, RGB or other *Color spaces.* For these, *operations* are supported.

When color is specified using numbers, it is interpreted with respect to the stimulus’ current *colorSpace.* If color is given as a single value (scalar) then this will be applied to all 3 channels.

### Examples

For whatever stim you have:

```python
stim.color = 'white'
stim.color = 'RoyalBlue'  # (the case is actually ignored)
stim.color = '#DDA0DD'  # DDA0DD is hexadecimal for plum
stim.color = [1.0, -1.0, -1.0]  # if stim.colorSpace='rgb':
    # a red color in rgb space
stim.color = [0.0, 45.0, 1.0]  # if stim.colorSpace='dkl':
    # DKL space with elev=0, azimuth=45
stim.color = [0, 0, 255]  # if stim.colorSpace='rgb255':
    # a blue stimulus using rgb255 space
stim.color = 255  # interpreted as (255, 255, 255)
    # which is white in rgb255.
```

*Operations* work as normal for all numeric *colorSpaces* (e.g. ‘rgb’, ‘hsv’ and ‘rgb255’) but not for strings, like named and hex. For example, assuming that colorSpace='rgb':

```python
stim.color += [1, 1, 1]  # increment all guns by 1 value
stim.color *= -1  # multiply the color by -1 (which in this
    # space inverts the contrast)
stim.color *= [0.5, 0, 1]  # decrease red, remove green, keep blue
```

You can use *setColor* if you want to set color and colorSpace in one line. These two are equivalent:

```python
stim.setColor((0, 128, 255), 'rgb255')
# ... is equivalent to
stim.colorSpace = 'rgb255'
stim.color = (0, 128, 255)
```

**colorSpace**

The name of the color space currently being used

Value should be: a string or None

For strings and hex values this is not needed. If None the default *colorSpace* for the stimulus is used (defined during initialisation).

Please note that changing *colorSpace* does not change stimulus parameters. Thus you usually want to specify *colorSpace* before setting the color. Example:

```python
# A light green text
stim = visual.TextStim(win, 'Color me!',
    color=(0, 1, 0), colorSpace='rgb')

# An almost-black text
```

(continues on next page)
contains \((x, y=None, units=None)\)

Returns True if a point \(x, y\) is inside the stimulus’ border.

Can accept variety of input options:

- two separate args, \(x\) and \(y\)
- one arg (list, tuple or array) containing two vals \((x, y)\)
- an object with a \(\text{getPos}()\) method that returns \(x, y\), such as a \text{Mouse}.

Returns True if the point is within the area defined either by its border attribute (if one defined), or its vertices attribute if there is no border. This method handles complex shapes, including concavities and self-crossings.

Note that, if your stimulus uses a mask (such as a Gaussian) then this is not accounted for by the contains method; the extent of the stimulus is determined purely by the size, position (pos), and orientation (ori) settings (and by the vertices for shape stimuli).

See Coder demos: shapeContains.py

contrast

A value that is simply multiplied by the color

Value should be: a float between -1 (negative) and 1 (unchanged). Operations supported.

Set the contrast of the stimulus, i.e. scales how far the stimulus deviates from the middle grey. You can also use the stimulus opacity to control contrast, but that cannot be negative.

Examples:

```python
stim.contrast = 1.0  # unchanged contrast
stim.contrast = 0.5  # decrease contrast
stim.contrast = 0.0  # uniform, no contrast
stim.contrast = -0.5 # slightly inverted
stim.contrast = -1.0 # totally inverted
```

Setting contrast outside range -1 to 1 is permitted, but may produce strange results if color values exceeds the monitor limits.: 

```python
stim.contrast = 1.2  # increases contrast
stim.contrast = -1.2 # inverts with increased contrast
```

depth

DEPRECATED. Depth is now controlled simply by drawing order.

draw \(\text{(win=None)}\)

Draws the BufferImage on the screen, similar to ImageStim.draw(). Allows dynamic position, size, rotation, mirroring, and opacity. Limitations / bugs: not sure what happens with shaders and self._updateList()

flipHoriz

If set to True then the image will be flipped horizontally (left-to-right). Note that this is relative to the original image, not relative to the current state.

flipVert

If set to True then the image will be flipped vertically (left-to-right). Note that this is relative to the original image, not relative to the current state.
image
The image file to be presented (most formats supported).

interpolate
Whether to interpolate (linearly) the texture in the stimulus
If set to False then nearest neighbour will be used when needed, otherwise some form of interpolation will be used.

mask
The alpha mask that can be used to control the outer shape of the stimulus

• None, ‘circle’, ‘gauss’, ‘raisedCos’
• or the name of an image file (most formats supported)
• or a numpy array (1xN or NxN) ranging -1:1

maskParams
Various types of input. Default to None.
This is used to pass additional parameters to the mask if those are needed.

• For ‘gauss’ mask, pass dict {‘sd’: 5} to control standard deviation.
• For the ‘raisedCos’ mask, pass a dict: {‘fringeWidth’:0.2}, where ‘fringeWidth’ is a parameter (float, 0-1), determining the proportion of the patch that will be blurred by the raised cosine edge.

name
String or None. The name of the object to be using during logged messages about this stim. If you have multiple stimuli in your experiment this really helps to make sense of log files!
If name = None your stimulus will be called “unnamed <type>”, e.g. visual.TextStim(win) will be called “unnamed TextStim” in the logs.

opacity
Determines how visible the stimulus is relative to background
The value should be a single float ranging 1.0 (opaque) to 0.0 (transparent). Operations are supported. Precisely how this is used depends on the Blend Mode.

ori
The orientation of the stimulus (in degrees).
Should be a single value (scalar). Operations are supported.
Orientation convention is like a clock: 0 is vertical, and positive values rotate clockwise. Beyond 360 and below zero values wrap appropriately.

overlaps (polygon)
Returns True if this stimulus intersects another one.
If polygon is another stimulus instance, then the vertices and location of that stimulus will be used as the polygon. Overlap detection is typically very good, but it can fail with very pointy shapes in a crossed-swords configuration.
Note that, if your stimulus uses a mask (such as a Gaussian blob) then this is not accounted for by the overlaps method; the extent of the stimulus is determined purely by the size, pos, and orientation settings (and by the vertices for shape stimuli).
See coder demo, shapeContains.py

pos
The position of the center of the stimulus in the stimulus units
value should be an \textit{x,y-pair}. \textit{Operations} are also supported.

Example:

\begin{verbatim}
stim.pos = (0.5, 0)  # Set slightly to the right of center
stim.pos += (0.5, -1)  # Increment pos rightwards and upwards.
    is now (1.0, -1.0)
stim.pos *= 0.2  # Move stim towards the center.
    is now (0.2, -0.2)
\end{verbatim}

Tip: If you need the position of stim in pixels, you can obtain it like this:

\begin{verbatim}
from psychopy.tools.monitorunittools import posToPix
posPix = posToPix(stim)
\end{verbatim}

\texttt{setAutoDraw} (\texttt{value, log=None})

Sets autoDraw. Usually you can use \texttt{stim.attribute = value} syntax instead, but use this method to suppress the log message.

\texttt{setAutoLog} (\texttt{value=True, log=None})

Usually you can use \texttt{stim.attribute = value} syntax instead, but use this method if you need to suppress the log message.

\texttt{setBorderColor} (\texttt{color, colorSpace=None, operation='', log=None})

\texttt{setColor} (\texttt{color, colorSpace=None, operation='', log=None})

Usually you can use \texttt{stim.attribute = value} syntax instead, but use this method if you need to suppress the log message and/or set colorSpace simultaneously.

\texttt{setContrast} (\texttt{newContrast, operation='', log=None})

Usually you can use \texttt{stim.attribute = value} syntax instead, but use this method if you need to suppress the log message

\texttt{setDKL} (\texttt{newDKL, operation='})

DEPRECATED since v1.60.05: Please use the \texttt{color} attribute

\texttt{setDepth} (\texttt{newDepth, operation='}), \texttt{log=None})

Usually you can use \texttt{stim.attribute = value} syntax instead, but use this method if you need to suppress the log message

\texttt{setFillColor} (\texttt{color, colorSpace=None, operation='', log=None})

\texttt{setFlipHoriz} (\texttt{newVal=True, log=None})

Usually you can use \texttt{stim.attribute = value} syntax instead, but use this method if you need to suppress the log message.

\texttt{setFlipVert} (\texttt{newVal=True, log=None})

Usually you can use \texttt{stim.attribute = value} syntax instead, but use this method if you need to suppress the log message.

\texttt{setImage} (\texttt{value, log=None})

Usually you can use \texttt{stim.attribute = value} syntax instead, but use this method if you need to suppress the log message.

\texttt{setLMS} (\texttt{newLMS, operation='})

DEPRECATED since v1.60.05: Please use the \texttt{color} attribute

\texttt{setMask} (\texttt{value, log=None})

Usually you can use \texttt{stim.attribute = value} syntax instead, but use this method if you need to suppress the log message

\texttt{setOpacity} (\texttt{newOpacity, operation='}), \texttt{log=None})

Usually you can use \texttt{stim.attribute = value} syntax instead, but use this method if you need to suppress the log message.
setOri (newOri, operation='', log=None)
    Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setPos (newPos, operation='', log=None)
    Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

setRGB (newRGB, operation='', log=None)
    DEPRECATED since v1.60.05: Please use the color attribute

setSize (newSize, operation='', units=None, log=None)
    Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setUseShaders (value=True, log=None)
    Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

size
    The size (width, height) of the stimulus in the stimulus units
    Value should be x,y-pair, scalar (applies to both dimensions) or None (resets to default). Operations are supported.
    Sizes can be negative (causing a mirror-image reversal) and can extend beyond the window.
    Example:

    ```python
    stim.size = 0.8  # Set size to (xsize, ysize) = (0.8, 0.8)
    print(stim.size)  # Outputs array([0.8, 0.8])
    stim.size += (0.5, -0.5)  # make wider and flatter: (1.3, 0.3)
    ```

    Tip: if you can see the actual pixel range this corresponds to by looking at stim._sizeRendered

texRes
    Power-of-two int. Sets the resolution of the mask and texture. texRes is overridden if an array or image is provided as mask.
    Operations supported.

units
    None, 'norm', 'cm', 'deg', 'degFlat', 'degFlatPos', or 'pix'
    If None then the current units of the Window will be used. See Units for the window and stimuli for explanation of other options.
    Note that when you change units, you don't change the stimulus parameters and it is likely to change appearance.
    Example:

    ```python
    # This stimulus is 20% wide and 50% tall with respect to window
    stim = visual.PatchStim(win, units='norm', size=(0.2, 0.5))
    # This stimulus is 0.2 degrees wide and 0.5 degrees tall.
    stim.units = 'deg'
    ```

useShaders
    Should shaders be used to render the stimulus (typically leave as True)
    If the system support the use of OpenGL shader language then leaving this set to True is highly recommended.
    If shaders cannot be used then various operations will be slower (notably, changes to stimulus color or contrast)
property verticesPix
This determines the coordinates of the vertices for the current stimulus in pixels, accounting for size, ori, pos and units

win

The Window object in which the stimulus will be rendered by default. (required)

Example, drawing same stimulus in two different windows and display simultaneously. Assuming that you have two windows and a stimulus (win1, win2 and stim):

```python
stim.win = win1  # stimulus will be drawn in win1
stim.draw()  # stimulus is now drawn to win1
stim.win = win2  # stimulus will be drawn in win2
stim.draw()  # it is now drawn in win2
win1.flip(waitBlanking=False)  # do not wait for next
    # monitor update
win2.flip()  # wait for vertical blanking.
```

Note that this just changes *default* window for stimulus. You could also specify window-to-draw-to when drawing::

```python
stim.draw(win1)
stim.draw(win2)
```

### 8.2.5 Circle

```python
class psychopy.visual.Circle(*args, **kwargs)
```

Creates a Circle with a given radius as a special case of a ShapeStim

(New in version 1.72.00)

Circle accepts all input parameters that `psychopy.visual.ShapeStim` accept, except for vertices and closeShape.

### 8.2.6 CustomMouse

```python
class psychopy.visual.CustomMouse(*args, **kwargs)
```

Class for more control over the mouse, including the pointer graphic and bounding box.

Seems to work with pyglet or pygame. Not completely tested.

Known limitations: - only norm units are working - getRel() always returns [0,0] - mouseMoved() is always False; maybe due to

```python
self.mouse.visible == False -> held at [0,0]
```

- no idea if clickReset() works

Author: Jeremy Gray, 2011

Class for customizing the appearance and behavior of the mouse.

Use a custom mouse for extra control over the pointer appearance and function. It’s probably slower to render than the regular system mouse. Create your `visual.Window` before creating a CustomMouse.

**Parameters**

- **win** [required, `visual.Window`] the window to which this mouse is attached
visible  [True or False] makes the mouse invisible if necessary

newPos  [None or [x,y]] gives the mouse a particular starting position

leftLimit  :  left edge of a virtual box within which the mouse can move

topLimit  :  top edge of virtual box

rightLimit  :  right edge of virtual box

bottomLimit  :  lower edge of virtual box

showLimitBox  [default is False] display the boundary within which the mouse can move.

pointer  :  The visual display item to use as the pointer; must have .draw() and setPos() methods.

If your item has .setOpacity(), you can alter the mouse’s opacity.

clickOnUp  [when to count a mouse click as having occurred] default is False, record a click when the mouse is first pressed down. True means record a click when the mouse button is released.

Note  CustomMouse is a new feature, and subject to change. setPos() does not work yet. getRel() returns [0,0] and mouseMoved() always returns False. clickReset() may not be working.

---

8.2.7 DotStim

**class** psychopy.visual.DotStim(*args, **kwargs)**

This stimulus class defines a field of dots with an update rule that determines how they change on every call to the .draw() method.

This single class can be used to generate a wide variety of dot motion types. For a review of possible types and their pros and cons see Scase, Braddick & Raymond (1996). All six possible motions they describe can be generated with appropriate choices of the signalDots (which determines whether signal dots are the ‘same’ or ‘different’ on each frame), noiseDots (which determines the locations of the noise dots on each frame) and the dotLife (which determines for how many frames the dot will continue before being regenerated).

The default settings (as of v1.70.00) is for the noise dots to have identical velocity but random direction and signal dots remain the ‘same’ (once a signal dot, always a signal dot).

For further detail about the different configurations see Dots (RDK) Component in the Builder Components section of the documentation.

If further customisation is required, then the DotStim should be subclassed and its _update_dotsXY and _newDotsXY methods overridden.

The maximum number of dots that can be drawn is limited by system performance.

- **fieldShape**
  - ‘sqr’ or ‘circle’. Defines the envelope used to present the dots. If changed while drawing, dots outside new envelope will be respawned.

  Type  str

- **dotSize**
  - Dot size specified in pixels (overridden if element is specified). operations are supported.

  Type  float

- **dotLife**
  - Number of frames each dot lives for (-1=infinite). Dot lives are initiated randomly from a uniform distribution from 0 to dotLife. If changed while drawing, the lives of all dots will be randomly initiated again.

  Type  int
signalDots
If ‘same’ then the signal and noise dots are constant. If ‘different’ then the choice of which is signal and which
is noise gets randomised on each frame. This corresponds to Scase et al’s (1996) categories of RDK.

Type str

noiseDots
Determines the behaviour of the noise dots, taken directly from Scase et al’s (1996) categories. For ‘position’,
noise dots take a random position every frame. For ‘direction’ noise dots follow a random, but constant
direction. For ‘walk’ noise dots vary their direction every frame, but keep a constant speed.

Type str

element
This can be any object that has a .draw() method and a .setPos([x,y]) method (e.g. a GratingStim, TextStim…)! DotStim assumes that the element uses pixels as units. None defaults to dots.

Type object

fieldPos
Specifying the location of the centre of the stimulus using a x,y-pair. See e.g. ShapeStim for more docu-
mentation/examples on how to set position. operations are supported.

Type array_like

fieldSize
Specifying the size of the field of dots using a x,y-pair. See e.g. ShapeStim for more docu-
mentation/examples on how to set position. operations are supported.

Type array_like

coherence
Change the coherence (%) of the DotStim. This will be rounded according to the number of dots in the
stimulus.

Type float

dir
Direction of the coherent dots in degrees. operations are supported.

Type float

speed
Speed of the dots (in units/frame). operations are supported.

Type float

Parameters

• win (window.Window) – Window this stimulus is associated with.
• units (str) – Units to use.
• nDots (int) – Number of dots to present in the field.
• coherence (float) – Proportion of dots which are coherent. This value can be set using the coherence property after initialization.
• fieldPos (array_like) – (x,y) or [x,y] position of the field. This value can be set using the fieldPos property after initialization.
• fieldSize (array_like, int or float) – (x,y) or [x,y] or single value (applied to both dimensions). Sizes can be negative and can extend beyond the window. This value can be set using the fieldSize property after initialization.
• **fieldShape** (str) – Defines the envelope used to present the dots. If changed while drawing by setting the `fieldShape` property, dots outside new envelope will be respawned. Valid values are ‘square’, ‘sqr’ or ‘circle’.

• **dotSize** (array_like or float) – Size of the dots. If given an array, the sizes of individual dots will be set. The array must have length `nDots`. If a single value is given, all dots will be set to the same size.

• **dotLife** (int) – Lifetime of a dot in frames. Dot lives are initiated randomly from a uniform distribution from 0 to dotLife. If changed while drawing, the lives of all dots will be randomly initiated again. A value of -1 results in dots having an infinite lifetime. This value can be set using the `dotLife` property after initialization.

• **dir** (float) – Direction of the coherent dots in degrees. At 0 degrees, coherent dots will move from left to right. Increasing the angle will rotate the direction counter-clockwise. This value can be set using the `dir` property after initialization.

• **speed** (float) – Speed of the dots (in units per frame). This value can be set using the `speed` property after initialization.

• **rgb** (array_like, optional) – Color of the dots in form (r, g, b) or [r, g, b]. Deprecated, use `color` instead.

• **color** (array_like or str) – Color of the dots in form (r, g, b) or [r, g, b].

• **colorSpace** (str) – Colorspace to use.

• **opacity** (float) – Opacity of the dots from 0.0 to 1.0.

• **contrast** (float) – Contrast of the dots 0.0 to 1.0. This value is simply multiplied by the `color` value.

• **depth** (float) – Deprecated. depth is now controlled simply by drawing order.

• **element** (object) – This can be any object that has a `.draw()` method and a `.setPos([x, y])` method (e.g. a GratingStim, TextStim…)! DotStim assumes that the element uses pixels as units. None defaults to dots.

• **signalDots** (str) – If ‘same’ then the signal and noise dots are constant. If different then the choice of which is signal and which is noise gets randomised on each frame. This corresponds to Scase et al.’s (1996) categories of RDK. This value can be set using the `signalDots` property after initialization.

• **noiseDots** (str) – Determines the behaviour of the noise dots, taken directly from Scase et al.’s (1996) categories. For ‘position’, noise dots take a random position every frame. For ‘direction’ noise dots follow a random, but constant direction. For ‘walk’ noise dots vary their direction every frame, but keep a constant speed. This value can be set using the `noiseDots` property after initialization.

• **name** (str, optional) – Optional name to use for logging.

• **autoLog** (bool) – Enable automatic logging.
8.2.8 ElementArrayStim

class psychopy.visual.ElementArrayStim(*args, **kwargs)

This stimulus class defines a field of elements whose behaviour can be independently controlled. Suitable for creating ‘global form’ stimuli or more detailed random dot stimuli.

This stimulus can draw thousands of elements without dropping a frame, but in order to achieve this performance, uses several OpenGL extensions only available on modern graphics cards (supporting OpenGL2.0). See the ElementArray demo.

Parameters

win : a Window object (required)

units : [None, ‘height’, ‘norm’, ‘cm’, ‘deg’ or ‘pix’] If None then the current units of the Window will be used. See Units for the window and stimuli for explanation of other options.

nElements : number of elements in the array.

8.2.9 GratingStim

Attributes

<table>
<thead>
<tr>
<th>GratingStim(win[, tex, mask, units, pos, …])</th>
<th>Stimulus object for drawing arbitrary bitmaps that can repeat (cycle) in either dimension.</th>
</tr>
</thead>
<tbody>
<tr>
<td>GratingStim.win</td>
<td>The Window object in which the stimulus is drawn.</td>
</tr>
<tr>
<td>GratingStim.tex</td>
<td>Texture to used on the stimulus as a grating (aka carrier)</td>
</tr>
<tr>
<td>GratingStim.mask</td>
<td>The alpha mask (forming the shape of the image)</td>
</tr>
<tr>
<td>GratingStim.units</td>
<td>None, ‘norm’, ‘cm’, ‘deg’, ‘degFlat’, ‘degFlatPos’, or ‘pix’</td>
</tr>
<tr>
<td>GratingStim.sf</td>
<td>Spatial frequency of the grating texture</td>
</tr>
<tr>
<td>GratingStim.pos</td>
<td>The position of the center of the stimulus in the stimulus units</td>
</tr>
<tr>
<td>GratingStim.ori</td>
<td>The orientation of the stimulus (in degrees).</td>
</tr>
<tr>
<td>GratingStim.size</td>
<td>The size (width, height) of the stimulus in the stimulus units</td>
</tr>
<tr>
<td>GratingStim.contrast</td>
<td>A value that is simply multiplied by the color</td>
</tr>
<tr>
<td>GratingStim.color</td>
<td>Color of the stimulus</td>
</tr>
<tr>
<td>GratingStim.colorSpace</td>
<td>The name of the color space currently being used</td>
</tr>
<tr>
<td>GratingStim.opacity</td>
<td>Determines how visible the stimulus is relative to background</td>
</tr>
<tr>
<td>GratingStim.interpolate</td>
<td>Whether to interpolate (linearly) the texture in the stimulus</td>
</tr>
<tr>
<td>GratingStim.name</td>
<td>String or None.</td>
</tr>
<tr>
<td>GratingStim.autoLog</td>
<td>Whether every change in this stimulus should be automatically logged.</td>
</tr>
<tr>
<td>GratingStim.draw([win])</td>
<td>Draw the stimulus in its relevant window.</td>
</tr>
<tr>
<td>GratingStim.autoDraw</td>
<td>Determines whether the stimulus should be automatically drawn on every frame flip.</td>
</tr>
</tbody>
</table>
Details

class psychopy.visual.GratingStim

Stimulus object for drawing arbitrary bitmaps that can repeat (cycle) in either dimension.

One of the main stimuli for PsychoPy.

Formally GratingStim is just a texture behind an optional transparency mask (an 'alpha mask'). Both the texture and mask can be arbitrary bitmaps and their combination allows an enormous variety of stimuli to be drawn in realtime.

Examples:

```python
myGrat = GratingStim(tex='sin', mask='circle')  # circular grating
myGabor = GratingStim(tex='sin', mask='gauss')  # gives a 'Gabor'
```

A GratingStim can be rotated scaled and shifted in position, its texture can be drifted in X and/or Y and it can have a spatial frequency in X and/or Y (for an image file that simply draws multiple copies in the patch).

Also since transparency can be controlled two GratingStims can combine e.g. to form a plaid.

Using GratingStim with images from disk (jpg, tif, png, …)

Ideally texture images to be rendered should be square with 'power-of-2' dimensions e.g. 16 x 16, 128 x 128. Any image that is not will be upscaled (with linear interpolation) to the nearest such texture by PsychoPy. The size of the stimulus should be specified in the normal way using the appropriate units (deg, pix, cm, …). Be sure to get the aspect ratio the same as the image (if you don’t want it stretched!).

_params

__calcPosRendered__()  
DEPRECATED in 1.80.00. This functionality is now handled by _updateVertices() and verticesPix.

__calcSizeRendered__()  
DEPRECATED in 1.80.00. This functionality is now handled by _updateVertices() and verticesPix.

__createTexture__(tex, id, pixFormat, stim, res=128, maskParams=None, forcePOW2=True, dataType=None, wrapping=True)

Params

id: is the texture ID

pixFormat: GL.GL_ALPHA, GL.GL_RGB

useShaders: bool

interpolate: bool (determines whether texture will use GL_LINEAR or GL_NEAREST

res: the resolution of the texture (unless a bitmap image is used)

dataType: None, GL.GL_UNSIGNED_BYTE, GL.FLOAT. Only affects image files (numpy arrays will be float)

For grating stimuli (anything that needs multiple cycles) forcePOW2 should be set to be True. Otherwise the wrapping of the texture will not work.

_getDesiredRGB__(rgb, colorSpace, contrast)

Convert color to RGB while adding contrast. Requires self.rgb, self.colorSpace and self.contrast
_getPolyAsRendered()
DEPRECATED. Return a list of vertices as rendered.

_selectWindow(win)
Switch drawing to the specified window. Calls the window’s _setCurrent() method which handles the switch.

_set (attrib, val, op=',', log=None)
DEPRECATED since 1.80.04 + 1. Use setAttribute() and val2array() instead.

_updateList()
The user shouldn’t need this method since it gets called after every call to .set() Chooses between using and not using shaders each call.

_updateListNoShaders()
The user shouldn’t need this method since it gets called after every call to .set() Basically it updates the OpenGL representation of your stimulus if some parameter of the stimulus changes. Call it if you change a property manually rather than using the .set() command

_updateListShaders()
The user shouldn’t need this method since it gets called after every call to .set() Basically it updates the OpenGL representation of your stimulus if some parameter of the stimulus changes. Call it if you change a property manually rather than using the .set() command

_updateVertices()
Sets Stim.verticesPix and _borderPix from pos, size, ori, flipVert, flipHoriz

autoDraw
Determines whether the stimulus should be automatically drawn on every frame flip.
Value should be: True or False. You do NOT need to set this on every frame flip!

autoLog
Whether every change in this stimulus should be auto logged.
Value should be: True or False. Set to False if your stimulus is updating frequently (e.g. updating its position every frame) and you want to avoid swamping the log file with messages that aren’t likely to be useful.

blendmode
The OpenGL mode in which the stimulus is draw
Can the ‘avg’ or ‘add’. Average (avg) places the new stimulus over the old one with a transparency given by its opacity. Opaque stimuli will hide other stimuli transparent stimuli won’t. Add performs the arithmetic sum of the new stimulus and the ones already present.

clearTextures()
Clear all textures associated with the stimulus.
As of v1.61.00 this is called automatically during garbage collection of your stimulus, so doesn’t need calling explicitly by the user.

color
Color of the stimulus
Value should be one of:
  • string: to specify a Colors by name. Any of the standard html/X11 color names <http://www.w3schools.com/html/html_colornames.asp> can be used.
    • Colors by hex value
• numerically: (scalar or triplet) for DKL, RGB or other Color spaces. For these, operations are supported.

When color is specified using numbers, it is interpreted with respect to the stimulus’ current colorSpace. If color is given as a single value (scalar) then this will be applied to all 3 channels.

Examples

For whatever stim you have:

```python
stim.color = 'white'
stim.color = 'RoyalBlue'  # (the case is actually ignored)
stim.color = '#DDA0DD'   # DDA0DD is hexadecimal for plum
stim.color = [1.0, -1.0, -1.0]  # if stim.colorSpace='rgb':
    # a red color in rgb space
stim.color = [0.0, 45.0, 1.0]  # if stim.colorSpace='dkl':
    # DKL space with elev=0, azimuth=45
stim.color = [0, 0, 255]  # if stim.colorSpace='rgb255':
    # a blue stimulus using rgb255 space
stim.color = 255  # interpreted as (255, 255, 255)
    # which is white in rgb255.
```

Operations work as normal for all numeric colorSpaces (e.g. ‘rgb’, ‘hsv’ and ‘rgb255’) but not for strings, like named and hex. For example, assuming that colorSpace='rgb':

```python
stim.color += [1, 1, 1]  # increment all guns by 1 value
stim.color *= -1  # multiply the color by -1 (which in this
    # space inverts the contrast)
stim.color *= [0.5, 0, 1]  # decrease red, remove green, keep blue
```

You can use setColor if you want to set color and colorSpace in one line. These two are equivalent:

```python
stim.setColor((0, 128, 255), 'rgb255')
# ... is equivalent to
stim.colorSpace = 'rgb255'
stim.color = (0, 128, 255)
```

colorSpace

The name of the color space currently being used

Value should be: a string or None

For strings and hex values this is not needed. If None the default colorSpace for the stimulus is used (defined during initialisation).

Please note that changing colorSpace does not change stimulus parameters. Thus you usually want to specify colorSpace before setting the color. Example:

```python
# A light green text
stim = visual.TextStim(win, 'Color me!',
    color=(0, 1, 0), colorSpace='rgb')

# An almost-black text
stim.colorSpace = 'rgb255'

# Make it light green again
stim.color = (128, 255, 128)
```
contains \((x, y=None, units=None)\)

Returns True if a point \(x, y\) is inside the stimulus’ border.

**Can accept variety of input options:**

- two separate args, \(x\) and \(y\)
- one arg (list, tuple or array) containing two vals \((x, y)\)
- an object with a `getPos()` method that returns \(x, y\), such as a `Mouse`.

Returns `True` if the point is within the area defined either by its `border` attribute (if one defined), or its `vertices` attribute if there is no `.border`. This method handles complex shapes, including concavities and self-crossings.

Note that, if your stimulus uses a mask (such as a Gaussian) then this is not accounted for by the `contains` method; the extent of the stimulus is determined purely by the size, position (pos), and orientation (ori) settings (and by the vertices for shape stimuli).

See Coder demos: `shapeContains.py` See Coder demos: `shapeContains.py`

**contrast**

A value that is simply multiplied by the color

**Value should be: a float between -1 (negative) and 1 (unchanged). Operations supported.**

Set the contrast of the stimulus, i.e. scales how far the stimulus deviates from the middle grey. You can also use the stimulus `opacity` to control contrast, but that cannot be negative.

Examples:

```python
stim.contrast = 1.0  # unchanged contrast
stim.contrast = 0.5  # decrease contrast
stim.contrast = 0.0  # uniform, no contrast
stim.contrast = -0.5 # slightly inverted
stim.contrast = -1.0 # totally inverted
```

Setting contrast outside range -1 to 1 is permitted, but may produce strange results if color values exceed the monitor limits.:  

```python
stim.contrast = 1.2  # increases contrast
stim.contrast = -1.2 # inverts with increased contrast
```

depth

DEPRECATED. Depth is now controlled simply by drawing order.

draw \((\text{win}=\text{None})\)

Draw the stimulus in its relevant window. You must call this method after every `MyWin.flip()` if you want the stimulus to appear on that frame and then update the screen again.

interpolate

Whether to interpolate (linearly) the texture in the stimulus

If set to False then nearest neighbour will be used when needed, otherwise some form of interpolation will be used.

mask

The alpha mask (forming the shape of the image)

**This can be one of various options:**

- `circle`, `gauss`, `raisedCos`, `cross`
- `None` (resets to default)
• the name of an image file (most formats supported)
• a numpy array (1xN or NxN) ranging -1:1

**maskParams**
Various types of input. Default to None.

This is used to pass additional parameters to the mask if those are needed.

• For ‘gauss’ mask, **pass dict** `{‘sd’: 5} to control** standard deviation.

• For the ‘raisedCos’ mask, **pass a dict**: `{‘fringeWidth’:0.2}, where ‘fringeWidth’ is a parameter (float, 0-1), determining the proportion of the patch that will be blurred by the raised cosine edge.

**name**
String or None. The name of the object to be using during logged messages about this stim. If you have multiple stimuli in your experiment this really helps to make sense of log files!

If name = None your stimulus will be called “unnamed <type>”, e.g. visual.TextStim(win) will be called “unnamed TextStim” in the logs.

**opacity**
Determines how visible the stimulus is relative to background

The value should be a single float ranging 1.0 (opaque) to 0.0 (transparent). **Operations** are supported. Precisely how this is used depends on the **Blend Mode**.

**ori**
The orientation of the stimulus (in degrees).

Should be a single value (**scalar**). **Operations** are supported.

Orientation convention is like a clock: 0 is vertical, and positive values rotate clockwise. Beyond 360 and below zero values wrap appropriately.

**overlaps** (**polygon**)
Returns **True** if this stimulus intersects another one.

If **polygon** is another stimulus instance, then the vertices and location of that stimulus will be used as the polygon. Overlap detection is typically very good, but it can fail with very pointy shapes in a crossed-swords configuration.

Note that, if your stimulus uses a mask (such as a Gaussian blob) then this is not accounted for by the **overlaps** method; the extent of the stimulus is determined purely by the size, pos, and orientation settings (and by the vertices for shape stimuli).

See coder demo, shapeContains.py

**phase**
The phase of the stimulus in each dimension of the texture.

Should be an **x,y-pair** or **scalar**

**NB** phase has modulus 1 (rather than 360 or 2*pi) This is a little unconventional but has the nice effect that setting phase=t*n drifts a stimulus at n Hz

**pos**
The position of the center of the stimulus in the stimulus **units**

**value** should be an **x,y-pair**. **Operations** are also supported.

Example:
```python
stim.pos = (0.5, 0)  # Set slightly to the right of center
stim.pos += (0.5, -1)  # Increment pos rightwards and upwards.
    Is now (1.0, -1.0)
stim.pos *= 0.2  # Move stim towards the center.
    Is now (0.2, -0.2)
```

Tip: If you need the position of stim in pixels, you can obtain it like this:

```python
from psychopy.tools.monitorunittools import posToPix
posPix = posToPix(stim)
```

**setAutoDraw** *(value, log=None)*
Sets autoDraw. Usually you can use ‘stim.attribute = value’ syntax instead, but use this method to suppress the log message.

**setAutoLog** *(value=True, log=None)*
Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message.

**setBlendmode** *(value, log=None)*
DEPRECATED. Use ‘stim.parameter = value’ syntax instead

**setBorderColor** *(color, colorSpace=None, operation='', log=None)*

**setColor** *(color, colorSpace=None, operation='', log=None)*
Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message and/or set colorSpace simultaneously.

**setContrast** *(newContrast, operation='', log=None)*
Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message

**setDKL** *(newDKL, operation='')*
DEPRECATED since v1.60.05: Please use the `color` attribute

**setDepth** *(newDepth, operation='', log=None)*
Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message

**setFillColor** *(color, colorSpace=None, operation='', log=None)*

**setLMS** *(newLMS, operation='')*
DEPRECATED since v1.60.05: Please use the `color` attribute

**setMask** *(value, log=None)*
Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message.

**setOpacity** *(newOpacity, operation='', log=None)*
Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message

**setOri** *(newOri, operation='', log=None)*
Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message

**setPhase** *(value, operation='', log=None)*
DEPRECATED. Use ‘stim.parameter = value’ syntax instead

**setPos** *(newPos, operation='', log=None)*
Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message.
PsychoPy - Psychology software for Python, Release 2020.2.10

**setRGB** *(newRGB, operation='', log=None)*  
DEPRECATED since v1.60.05: Please use the *color* attribute

**setSF** *(value, operation='', log=None)*  
DEPRECATED. Use ‘stim.parameter = value’ syntax instead

**setSize** *(newSize, operation='', units=None, log=None)*  
Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message

**setTex** *(value, log=None)*  
DEPRECATED. Use ‘stim.parameter = value’ syntax instead

**setUseShaders** *(value=True, log=None)*  
Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message

**sf**  
Spatial frequency of the grating texture  
Should be a *x,y-pair* or *scalar* or None. If *units* == ‘deg’ or ‘cm’ units are in cycles per deg or cm as appropriate.

If *units* == ‘norm’ then sf units are in cycles per stimulus  (and so SF scales with stimulus size).

If texture is an image loaded from a file then *sf=None* defaults to 1/stimSize to give one cycle of the image.

**size**  
The size (width, height) of the stimulus in the stimulus *units*

Value should be *x,y-pair*, *scalar* (applies to both dimensions) or None (resets to default). *Operations* are supported.

Sizes can be negative (causing a mirror-image reversal) and can extend beyond the window.

Example:

```
stim.size = 0.8  # Set size to (xsize, ysize) = (0.8, 0.8)
print(stim.size)  # Outputs array([0.8, 0.8])
stim.size += (0.5, -0.5)  # make wider and flatter: (1.3, 0.3)
```

Tip: if you can see the actual pixel range this corresponds to by looking at *stim._sizeRendered*

**tex**  
Texture to used on the stimulus as a grating (aka carrier)

This can be one of various options:

- ‘sin’, ‘sqr’, ‘saw’, ‘tri’, None (resets to default)
- the name of an image file (most formats supported)
- a numpy array (1xN or NxN) ranging -1:1

If specifying your own texture using an image or numpy array you should ensure that the image has square power-of-two dimensions (e.g. 256 x 256). If not then PsychoPy will upsample your stimulus to the next larger power of two.

**texRes**  
Power-of-two int. Sets the resolution of the mask and texture. *texRes* is overridden if an array or image is provided as mask.
Operations supported.

units
None, ‘norm’, ‘cm’, ‘deg’, ‘degFlat’, ‘degFlatPos’, or ‘pix’
If None then the current units of the Window will be used. See Units for the window and stimuli for explanation of other options.

Note that when you change units, you don’t change the stimulus parameters and it is likely to change appearance. Example:

```python
# This stimulus is 20% wide and 50% tall with respect to window
stim = visual.PatchStim(win, units='norm', size=(0.2, 0.5))

# This stimulus is 0.2 degrees wide and 0.5 degrees tall.
stim.units = 'deg'
```

useShaders
Should shaders be used to render the stimulus (typically leave as True)
If the system support the use of OpenGL shader language then leaving this set to True is highly recommended. If shaders cannot be used then various operations will be slower (notably, changes to stimulus color or contrast)

property verticesPix
This determines the coordinates of the vertices for the current stimulus in pixels, accounting for size, ori, pos and units

win
The Window object in which the stimulus will be rendered by default. (required)

Example, drawing same stimulus in two different windows and display simultaneously. Assuming that you have two windows and a stimulus (win1, win2 and stim):

```python
stim.win = win1  # stimulus will be drawn in win1
stim.draw()     # stimulus is now drawn to win1
stim.win = win2  # stimulus will be drawn in win2
stim.draw()     # it is now drawn in win2
win1.flip(waitBlanking=False)    # do not wait for next
                                 # monitor update
win2.flip()      # wait for vertical blanking.

Note that this just changes *default* window for stimulus.
You could also specify window-to-draw-to when drawing::

stim.draw(win1)
stim.draw(win2)
```

8.2.10 Helper functions

psychopy.visual.helpers.pointInPolygon(x, y, poly)
Determine if a point is inside a polygon; returns True if inside.

(x, y) is the point to test. poly is a list of 3 or more vertices as (x,y) pairs. If given an object, such as a ShapeStim, will try to use its vertices and position as the polygon.

Same as the .contains() method elsewhere.

psychopy.visual.helpers.polygonsOverlap(poly1, poly2)
Determine if two polygons intersect; can fail for very pointy polygons.
Accepts two polygons, as lists of vertices (x,y) pairs. If given an object with with (vertices + pos), will try to use that as the polygon.

Checks if any vertex of one polygon is inside the other polygon. Same as the .overlaps() method elsewhere.

**Notes**

We implement special handling for the Line stimulus as it is not a proper polygon. We do not check for class instances because this would require importing of visual.Line, creating a circular import. Instead, we assume that a “polygon” with only two vertices is meant to specify a line. Pixels between the endpoints get interpolated before testing for overlap.

psychopy.visual.helpers.groupFlipVert (flipList, yReflect=0)

Reverses the vertical mirroring of all items in list flipList.

Reverses the .flipVert status, vertical (y) positions, and angular rotation (.ori). Flipping preserves the relations among the group's visual elements. The parameter yReflect is the y-value of an imaginary horizontal line around which to reflect the items; default = 0 (screen center).

Typical usage is to call once prior to any display; call again to un-flip. Can be called with a list of all stim to be presented in a given routine.

Will flip a) all psychopy.visual.xyzStim that have a setFlipVert method, b) the y values of .vertices, and c) items in n x 2 lists that are mutable (i.e., list, np.array, no tuples): [[x1, y1], [x2, y2], …]

### 8.2.11 ImageStim

As of PsychoPy version 1.79.00 some of the properties for this stimulus can be set using the syntax:

```python
stim.pos = newPos
```

others need to be set with the older syntax:

```python
stim.setImage(newImage)
```

**Attributes**

<table>
<thead>
<tr>
<th>ImageStim(win[, image, mask, units, pos, …])</th>
<th>Display an image on a psychopy.visual.Window object</th>
</tr>
</thead>
<tbody>
<tr>
<td>ImageStim.win</td>
<td>The Window object in which the stimulus is displayed</td>
</tr>
<tr>
<td>ImageStim.setImage(value[, log])</td>
<td>Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.</td>
</tr>
<tr>
<td>ImageStim.setMask(value[, log])</td>
<td>Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.</td>
</tr>
<tr>
<td>ImageStim.units</td>
<td>None, ‘norm’, ‘cm’, ‘deg’, ‘degFlat’, ‘degFlatPos’, or ‘pix’</td>
</tr>
<tr>
<td>ImageStim.pos</td>
<td>The position of the center of the stimulus in the stimulus units</td>
</tr>
<tr>
<td>ImageStim_ori</td>
<td>The orientation of the stimulus (in degrees).</td>
</tr>
<tr>
<td>ImageStim.size</td>
<td>The size (width, height) of the stimulus in the stimulus units</td>
</tr>
<tr>
<td>ImageStim.contrast</td>
<td>A value that is simply multiplied by the color</td>
</tr>
<tr>
<td>ImageStim.color</td>
<td>Color of the stimulus</td>
</tr>
<tr>
<td>ImageStim.colorSpace</td>
<td>The name of the color space currently being used</td>
</tr>
</tbody>
</table>

continues on next page
### Table 5 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ImageStim.opacity</code></td>
<td>Determines how visible the stimulus is relative to background</td>
</tr>
<tr>
<td><code>ImageStim.interpolate</code></td>
<td>Whether to interpolate (linearly) the texture in the stimulus</td>
</tr>
<tr>
<td><code>ImageStim.contains(x, y, units)</code></td>
<td>Returns True if a point x,y is inside the stimulus’ border.</td>
</tr>
<tr>
<td><code>ImageStim.overlaps(polygon)</code></td>
<td>Returns True if this stimulus intersects another one.</td>
</tr>
<tr>
<td><code>ImageStim.name</code></td>
<td>String or None.</td>
</tr>
<tr>
<td><code>ImageStim.autoLog</code></td>
<td>Whether every change in this stimulus should be auto logged.</td>
</tr>
<tr>
<td><code>ImageStim.draw([win])</code></td>
<td>Draw.</td>
</tr>
<tr>
<td><code>ImageStim.autoDraw</code></td>
<td>Determines whether the stimulus should be automatically drawn on every frame flip.</td>
</tr>
<tr>
<td><code>ImageStim.clearTextures()</code></td>
<td>Clear all textures associated with the stimulus.</td>
</tr>
</tbody>
</table>

### Details

```python
class psychopy.visual.ImageStim(win, image=None, mask=None, units='', pos=(0.0, 0.0),
size=None, ori=0.0, color=(1.0, 1.0, 1.0), colorSpace='rgb',
contrast=1.0, opacity=1.0, depth=0, interpolate=False,
flipHoriz=False, flipVert=False, texRes=128, name=None,
autoLog=None, maskParams=None)
```

Display an image on a `psychopy.visual.Window`

**_calcPosRendered_**

DEPRECATED in 1.80.00. This functionality is now handled by `_updateVertices()` and verticesPix.

**_calcSizeRendered_**

DEPRECATED in 1.80.00. This functionality is now handled by `_updateVertices()` and verticesPix.

**_createTexture_**

For grating stimuli (anything that needs multiple cycles) forcePOW2 should be set to be True. Otherwise the wrapping of the texture will not work.

**_getDesiredRGB_**

Convert color to RGB while adding contrast. Requires self.rgb, self.colorSpace and self.contrast

**_getPolyAsRendered_**

DEPRECATED. Return a list of vertices as rendered.

**_selectWindow_**

Switch drawing to the specified window. Calls the window’s _setCurrent() method which handles the switch.
DEPRECATED since 1.80.04 + 1. Use setAttribute() and val2array() instead.

The user shouldn’t need this method since it gets called after every call to .set() Chooses between using and not using shaders each call.

The user shouldn’t need this method since it gets called after every call to .set() Basically it updates the OpenGL representation of your stimulus if some parameter of the stimulus changes. Call it if you change a property manually rather than using the .set() command.

The user shouldn’t need this method since it gets called after every call to .set() Basically it updates the OpenGL representation of your stimulus if some parameter of the stimulus changes. Call it if you change a property manually rather than using the .set() command

Sets Stim.verticesPix and ._borderPix from pos, size, ori, flipVert, flipHoriz

determines whether the stimulus should be automatically drawn on every frame flip.

Values should be: True or False. You do NOT need to set this on every frame flip!

Values should be: True or False. Set to False if your stimulus is updating frequently (e.g. updating its position every frame) and you want to avoid swamping the log file with messages that aren’t likely to be useful.

Clear all textures associated with the stimulus.

As of v1.61.00 this is called automatically during garbage collection of your stimulus, so doesn’t need calling explicitly by the user.

Color of the stimulus

Values should be one of:

- string: to specify a Colors by name. Any of the standard html/X11 color names <http://www.w3schools.com/html/html_colrnames.asp> can be used.

- Colors by hex value

- numerically: (scalar or triplet) for DKL, RGB or other Color spaces. For these, operations are supported.

When color is specified using numbers, it is interpreted with respect to the stimulus’ current colorSpace. If color is given as a single value (scalar) then this will be applied to all 3 channels.
Examples

For whatever stim you have:

```python
stim.color = 'white'
stim.color = 'RoyalBlue'  # (the case is actually ignored)
stim.color = '#DA0DD'    # DDA0DD is hexadecimal for plum
stim.color = [1.0, -1.0, -1.0]  # if stim.colorSpace='rgb':
    # a red color in rgb space
stim.color = [0.0, 45.0, 1.0]  # if stim.colorSpace='dkl':
    # DKL space with elev=0, azimuth=45
stim.color = [0, 0, 255]   # if stim.colorSpace='rgb255':
    # a blue stimulus using rgb255 space
stim.color = 255           # interpreted as (255, 255, 255)
    # which is white in rgb255.
```

Operations work as normal for all numeric colorSpaces (e.g. 'rgb', 'hsv' and 'rgb255') but not for strings, like named and hex. For example, assuming that colorSpace='rgb':

```python
stim.color += [1, 1, 1]  # increment all guns by 1 value
stim.color *= -1        # multiply the color by -1 (which in this
    # space inverts the contrast)
stim.color *= [0.5, 0, 1]  # decrease red, remove green, keep blue
```

You can use setColor if you want to set color and colorSpace in one line. These two are equivalent:

```python
stim.setColor((0, 128, 255), 'rgb255')
# ... is equivalent to
stim.colorSpace = 'rgb255'
stim.color = (0, 128, 255)
```

colorSpace

The name of the color space currently being used

Value should be: a string or None

For strings and hex values this is not needed. If None the default colorSpace for the stimulus is used (defined during initialisation).

Please note that changing colorSpace does not change stimulus parameters. Thus you usually want to specify colorSpace before setting the color. Example:

```python
# A light green text
stim = visual.TextStim(win, 'Color me!;',
    color=(0, 1, 0), colorSpace='rgb')

# An almost-black text
stim.colorSpace = 'rgb255'

# Make it light green again
stim.color = (128, 255, 128)
```

contains (x, y=None, units=None)

Returns True if a point x,y is inside the stimulus’ border.

Can accept variety of input options:

- two separate args, x and y
- one arg (list, tuple or array) containing two vals (x,y)
• an object with a getPos() method that returns x,y, such as a Mouse.

Returns True if the point is within the area defined either by its border attribute (if one defined), or its vertices attribute if there is no border. This method handles complex shapes, including concavities and self-crossings.

Note that, if your stimulus uses a mask (such as a Gaussian) then this is not accounted for by the contains method; the extent of the stimulus is determined purely by the size, position (pos), and orientation (ori) settings (and by the vertices for shape stimuli).

See Coder demos: shapeContains.py

contrast
A value that is simply multiplied by the color

Value should be: a float between -1 (negative) and 1 (unchanged). Operations supported.

Set the contrast of the stimulus, i.e. scales how far the stimulus deviates from the middle grey. You can also use the stimulus opacity to control contrast, but that cannot be negative.

Examples:

```python
stim.contrast = 1.0  # unchanged contrast
stim.contrast = 0.5  # decrease contrast
stim.contrast = 0.0  # uniform, no contrast
stim.contrast = -0.5 # slightly inverted
stim.contrast = -1.0 # totally inverted
```

Setting contrast outside range -1 to 1 is permitted, but may produce strange results if color values exceeds the monitor limits.:  

```python
stim.contrast = 1.2  # increases contrast
stim.contrast = -1.2 # inverts with increased contrast
```

depth
DEPRECATED. Depth is now controlled simply by drawing order.

draw (win=None)
Draw.

image
The image file to be presented (most formats supported).

interpolate
Whether to interpolate (linearly) the texture in the stimulus

If set to False then nearest neighbour will be used when needed, otherwise some form of interpolation will be used.

mask
The alpha mask that can be used to control the outer shape of the stimulus

• None, ‘circle’, ‘gauss’, ‘raisedCos’
• or the name of an image file (most formats supported)
• or a numpy array (1xN or NxN) ranging -1:1

maskParams
Various types of input. Default to None.

This is used to pass additional parameters to the mask if those are needed.

• For ‘gauss’ mask, pass dict {'sd': 5} to control standard deviation.
• For the ‘raisedCos’ mask, pass a dict: {'fringeWidth':0.2}, where ‘fringeWidth’ is a parameter (float, 0-1), determining the proportion of the patch that will be blurred by the raised cosine edge.

name
String or None. The name of the object to be using during logged messages about this stim. If you have multiple stimuli in your experiment this really helps to make sense of log files!

If name = None your stimulus will be called “unnamed <type>”, e.g. visual.TextStim(win) will be called “unnamed TextStim” in the logs.

opacity
Determines how visible the stimulus is relative to background

The value should be a single float ranging 1.0 (opaque) to 0.0 (transparent). Operations are supported. Precisely how this is used depends on the Blend Mode.

ori
The orientation of the stimulus (in degrees).

Should be a single value (scalar). Operations are supported.

Orientation convention is like a clock: 0 is vertical, and positive values rotate clockwise. Beyond 360 and below zero values wrap appropriately.

overlaps (polygon)
Returns True if this stimulus intersects another one.

If polygon is another stimulus instance, then the vertices and location of that stimulus will be used as the polygon. Overlap detection is typically very good, but it can fail with very pointy shapes in a crossed-swords configuration.

Note that, if your stimulus uses a mask (such as a Gaussian blob) then this is not accounted for by the overlaps method; the extent of the stimulus is determined purely by the size, pos, and orientation settings (and by the vertices for shape stimuli).

See coder demo, shapeContains.py

pos
The position of the center of the stimulus in the stimulus units

value should be an x,y-pair. Operations are also supported.

Example:

```
stim.pos = (0.5, 0)  # Set slightly to the right of center
stim.pos += (0.5, -1)  # Increment pos rightwards and upwards.
   Is now (1.0, -1.0)
stim.pos *= 0.2  # Move stim towards the center.
   Is now (0.2, -0.2)
```

Tip: If you need the position of stim in pixels, you can obtain it like this:

```
from psychopy.tools.monitorunittools import posToPix
posPix = posToPix(stim)
```

setAutoDraw (value, log=None)
Sets autoDraw. Usually you can use ‘stim.attribute = value’ syntax instead, but use this method to suppress the log message.

setAutoLog (value=True, log=None)
Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message.

setBorderColor (color, colorSpace=None, operation='', log=None)
**setColor** *(color, colorSpace=None, operation='', log=None)*
Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message and/or set colorSpace simultaneously.

**setContrast** *(newContrast, operation='', log=None)*
Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

**setDKL** *(newDKL, operation='')*
DEPRECATED since v1.60.05: Please use the color attribute

**setDepth** *(newDepth, operation='', log=None)*
Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

**setFillColor** *(color, colorSpace=None, operation='', log=None)*

**setImage** *(value, log=None)*
Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

**setLMS** *(newLMS, operation='')*
DEPRECATED since v1.60.05: Please use the color attribute

**setMask** *(value, log=None)*
Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

**setOpacity** *(newOpacity, operation='', log=None)*
Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

**setOri** *(newOri, operation='', log=None)*
Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

**setPos** *(newPos, operation='', log=None)*
Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

**setRGB** *(newRGB, operation='', log=None)*
DEPRECATED since v1.60.05: Please use the color attribute

**setSize** *(newSize, operation='', units=None, log=None)*
Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

**setUseShaders** *(value=True, log=None)*
Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

**size**
The size (width, height) of the stimulus in the stimulus units

Value should be x,y-pair, scalar (applies to both dimensions) or None (resets to default). Operations are supported.

Sizes can be negative (causing a mirror-image reversal) and can extend beyond the window.

Example:
stim.size = 0.8  # Set size to (xsize, ysize) = (0.8, 0.8)
print(stim.size)  # Outputs array([0.8, 0.8])
stim.size += (0.5, -0.5)  # make wider and flatter: (1.3, 0.3)

Tip: if you can see the actual pixel range this corresponds to by looking at stim._sizeRendered

texRes
Power-of-two int. Sets the resolution of the mask and texture. texRes is overridden if an array or image is provided as mask.

Operations supported.

units
None, 'norm', 'cm', 'deg', 'degFlat', 'degFlatPos', or 'pix'
If None then the current units of the Window will be used. See Units for the window and stimuli for explanation of other options.

Note that when you change units, you don’t change the stimulus parameters and it is likely to change appearance. Example:

```python
# This stimulus is 20% wide and 50% tall with respect to window
stim = visual.PatchStim(win, units='norm', size=(0.2, 0.5))

# This stimulus is 0.2 degrees wide and 0.5 degrees tall.
stim.units = 'deg'
```

useShaders
Should shaders be used to render the stimulus (typically leave as True)

If the system support the use of OpenGL shader language then leaving this set to True is highly recommended. If shaders cannot be used then various operations will be slower (notably, changes to stimulus color or contrast)

property verticesPix
This determines the coordinates of the vertices for the current stimulus in pixels, accounting for size, ori, pos and units

win
The Window object in which the stimulus will be rendered by default. (required)

Example, drawing same stimulus in two different windows and display simultaneously. Assuming that you have two windows and a stimulus (win1, win2 and stim):

```python
stim.win = win1  # stimulus will be drawn in win1
stim.draw()  # stimulus is now drawn to win1
stim.win = win2  # stimulus will be drawn in win2
stim.draw()  # it is now drawn in win2
win1.flip(waitBlanking=False)  # do not wait for next monitor update
win2.flip()  # wait for vertical blanking.
```

Note that this just changes **default** window for stimulus. You could also specify window-to-draw-to when drawing:

```python
stim.draw(win1)
stim.draw(win2)
```
8.2.12 LightSource

Attributes

LightSource(win[, pos, diffuseColor, ...]) Class for representing a light source in a scene.

Details

class psychopy.visual.LightSource (win, pos=(0.0, 0.0, 0.0), diffuseColor=(1.0, 1.0, 1.0), specularColor=(1.0, 1.0, 1.0), ambientColor=(0.0, 0.0, 0.0), colorSpace='rgb', lightType='point', attenuation=(1, 0, 0))

Class for representing a light source in a scene.

Only point and directional lighting is supported by this object for now. The ambient color of the light source contributes to the scene ambient color defined by ambientLight.

Warning: This class is experimental and may result in undefined behavior.

Parameters

- win (~psychopy.visual.Window) – Window associated with this light source.
- pos (array_like) – Position of the light source (x, y, z, w). If w=1.0 the light will be a point source and x, y, and z is the position in the scene. If w=0.0, the light source will be directional and x, y, and z will define the vector pointing to the direction the light source is coming from. For instance, a vector of (0, 1, 0, 0) will indicate that a light source is coming from above.
- diffuseColor (array_like) – Diffuse light color.
- specularColor (array_like) – Specular light color.
- ambientColor (array_like) – Ambient light color.
- colorSpace (str) – Color space for diffuse, specular, and ambient colors.
- attenuation (array_like) – Values for the constant, linear, and quadratic terms of the lighting attenuation formula. Default is (1, 0, 0) which results in no attenuation.

property ambientColor
Ambient color of the material.

property ambientRGB
Diffuse color of the material.

property attenuation
Values for the constant, linear, and quadratic terms of the lighting attenuation formula.

property diffuseColor
Diffuse color of the material.

property diffuseRGB
Diffuse color of the material.

property lightType
Type of light source, can be ‘point’ or ‘directional’.
**property pos**  
Position of the light source in the scene in scene units.

**property specularColor**  
Specular color of the material.

**property specularRGB**  
Diffuse color of the material.

### 8.2.13 Line

```python
class psychopy.visual.Line(*args, **kwargs)
```

Creates a Line between two points.

(New in version 1.72.00)

Line accepts all input parameters, that ShapeStim accepts, except for vertices, closeShape and fillColor.

**Notes**

The `contains` method always return False because a line is not a proper (2D) polygon.

### 8.2.14 MovieStim

**Attributes**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>MovieStim(win[, filename, units, size, pos, ...])</code></td>
<td>A stimulus class for playing movies (mpeg, avi, etc…) in PsychoPy.</td>
</tr>
<tr>
<td><code>MovieStim.win</code></td>
<td>The <code>Window</code> object in which the</td>
</tr>
<tr>
<td><code>MovieStim.units</code></td>
<td>None, ‘norm’, ‘cm’, ‘deg’, ‘degFlat’, ‘degFlatPos’, or ‘pix’</td>
</tr>
<tr>
<td><code>MovieStim.pos</code></td>
<td>The position of the center of the stimulus in the stimulus units</td>
</tr>
<tr>
<td><code>MovieStim.ori</code></td>
<td>The orientation of the stimulus (in degrees).</td>
</tr>
<tr>
<td><code>MovieStim.size</code></td>
<td>The size (width, height) of the stimulus in the stimulus units</td>
</tr>
<tr>
<td><code>MovieStim.opacity</code></td>
<td>Determines how visible the stimulus is relative to background</td>
</tr>
<tr>
<td><code>MovieStim.name</code></td>
<td>String or None.</td>
</tr>
<tr>
<td><code>MovieStim.autoLog</code></td>
<td>Whether every change in this stimulus should be auto logged.</td>
</tr>
<tr>
<td><code>MovieStim.draw([win])</code></td>
<td>Draw the current frame to a particular visual.Window.</td>
</tr>
<tr>
<td><code>MovieStim.autoDraw</code></td>
<td>Determines whether the stimulus should be automatically drawn on every frame flip.</td>
</tr>
<tr>
<td><code>MovieStim.loadMovie(filename[, log])</code></td>
<td>Load a movie from file</td>
</tr>
<tr>
<td><code>MovieStim.play([log])</code></td>
<td>Continue a paused movie from current position.</td>
</tr>
<tr>
<td><code>MovieStim.seek(timestamp[, log])</code></td>
<td>Seek to a particular timestamp in the movie.</td>
</tr>
<tr>
<td><code>MovieStim.pause([log])</code></td>
<td>Pause the current point in the movie (sound will stop, current frame will not advance).</td>
</tr>
<tr>
<td><code>MovieStim.stop([log])</code></td>
<td>Stop the current point in the movie.</td>
</tr>
<tr>
<td><code>MovieStim.setFlipHoriz([newVal, log])</code></td>
<td>If set to True then the movie will be flipped horizontally (left-to-right).</td>
</tr>
<tr>
<td><code>MovieStim.setFlipVert([newVal, log])</code></td>
<td>If set to True then the movie will be flipped vertically (top-to-bottom).</td>
</tr>
</tbody>
</table>
class psychopy.visual.MovieStim(win, filename='', size=None, pos=(0.0, 0.0), ori=0.0, flipVert=False, flipHoriz=False, color=(1.0, 1.0, 1.0), colorSpace='rgb', opacity=1.0, volume=1.0, name=None, loop=False, autoLog=None, depth=0.0)

A stimulus class for playing movies (mpeg, avi, etc...) in PsychoPy.

Example:

```python
mov = visual.MovieStim(myWin, 'testMovie.mp4', flipVert=False)
print(mov.duration)
# give the original size of the movie in pixels:
print(mov.format.width, mov.format.height)
mov.draw()  # draw the current frame (automagically determined)
```

See MovieStim.py for demo.

Parameters

- **filename**: a string giving the relative or absolute path to the movie. Can be any movie that AVbin can read (e.g. mpeg, DivX)
- **flipVert** [True or False] If True then the movie will be top-bottom flipped
- **flipHoriz** [True or False] If True then the movie will be right-left flipped
- **volume**: The nominal level is 1.0, and 0.0 is silence, see pyglet.media.Player
- **loop** [bool, optional] Whether to start the movie over from the beginning if draw is called and the movie is done.

ifecycleRendered()

DEPRECATED in 1.80.00. This functionality is now handled by _updateVertices() and verticesPix.

ifecycleSizeRendered()

DEPRECATED in 1.80.00. This functionality is now handled by _updateVertices() and verticesPix

LifecyclePolyAsRendered()

DEPRECATED. Return a list of vertices as rendered.

_selectWindow(win)

Switch drawing to the specified window. Calls the window’s _setCurrent() method which handles the switch.

_set (attrib, val, op='', log=None)

DEPRECATED since 1.80.04 + 1. Use setAttribute() and val2array() instead.

_updateList()

The user shouldn’t need this method since it gets called after every call to .set() Chooses between using and not using shaders each call.

_updateVertices()

Sets Stim.verticesPix and _borderPix from pos, size, ori, flipVert, flipHoriz

autoDraw

Determines whether the stimulus should be automatically drawn on every frame flip.

Value should be: True or False. You do NOT need to set this on every frame flip!

autoLog

Whether every change in this stimulus should be auto logged.
Value should be: *True* or *False*. Set to *False* if your stimulus is updating frequently (e.g. updating its position every frame) and you want to avoid swamping the log file with messages that aren’t likely to be useful.

**contains** *(x, y=None, units=None)*

Returns True if a point x,y is inside the stimulus’ border.

**Can accept variety of input options:**

- two separate args, x and y
- one arg (list, tuple or array) containing two vals (x,y)
- an object with a `getPos()` method that returns x,y, such as a *Mouse*.

Returns *True* if the point is within the area defined either by its *border* attribute (if one defined), or its *vertices* attribute if there is no .border. This method handles complex shapes, including concavities and self-crossings.

Note that, if your stimulus uses a mask (such as a Gaussian) then this is not accounted for by the contains method; the extent of the stimulus is determined purely by the size, position (pos), and orientation (ori) settings (and by the vertices for shape stimuli).

See Coder demos: shapeContains.py See Coder demos: shapeContains.py

**depth**

DEPRECATED. Depth is now controlled simply by drawing order.

**draw** *(win=None)*

Draw the current frame to a particular visual.Window.

Draw to the default win for this object if not specified. The current position in the movie will be determined automatically.

This method should be called on every frame that the movie is meant to appear.

**loadMovie** *(filename, log=None)*

Load a movie from file

**Parameters**

- **filename**: string The name of the file, including path if necessary

  Brings up a warning if avbin is not found on the computer. After the file is loaded MovieStim.duration is updated with the movie duration (in seconds).

- **name**: String or None. The name of the object to be using during logged messages about this stim. If you have multiple stimuli in your experiment this really helps to make sense of log files!

  If name = None your stimulus will be called “unnamed <type>”, e.g. visual.TextStim(win) will be called “unnamed TextStim” in the logs.

- **opacity**: Determines how visible the stimulus is relative to background

  The value should be a single float ranging 1.0 (opaque) to 0.0 (transparent). *Operations* are supported. Precisely how this is used depends on the *Blend Mode*.

- **ori**: The orientation of the stimulus (in degrees).

  Should be a single value *(scalar)*. *Operations* are supported.
Orientation convention is like a clock: 0 is vertical, and positive values rotate clockwise. Beyond 360 and below zero values wrap appropriately.

overlaps (polygon)
Returns True if this stimulus intersects another one.

If polygon is another stimulus instance, then the vertices and location of that stimulus will be used as the polygon. Overlap detection is typically very good, but it can fail with very pointy shapes in a crossed-swords configuration.

Note that, if your stimulus uses a mask (such as a Gaussian blob) then this is not accounted for by the overlaps method; the extent of the stimulus is determined purely by the size, pos, and orientation settings (and by the vertices for shape stimuli).

See coder demo, shapeContains.py

pause (log=None)
Pause the current point in the movie (sound will stop, current frame will not advance). If play() is called again both will restart.

play (log=None)
Continue a paused movie from current position.

pos
The position of the center of the stimulus in the stimulus units
value should be an x,y-pair. Operations are also supported.

Example:

```python
stim.pos = (0.5, 0)  # Set slightly to the right of center
stim.pos += (0.5, -1)  # Increment pos rightwards and upwards.
    Is now (1.0, -1.0)
stim.pos *= 0.2  # Move stim towards the center.
    Is now (0.2, -0.2)
```

Tip: If you need the position of stim in pixels, you can obtain it like this:

```python
from psychopy.tools.monitorunitools import posToPix
posPix = posToPix(stim)
```

seek (timestamp, log=None)
Seek to a particular timestamp in the movie.

NB this does not seem very robust as at version 1.62, may crash!

setAutoDraw (val, log=None)
Add or remove a stimulus from the list of stimuli that will be automatically drawn on each flip

Parameters

- val: True/False  True to add the stimulus to the draw list, False to remove it

setAutoLog (value=True, log=None)
Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message.

setContrast ()
Not yet implemented for MovieStim.

setDKL (newDKL, operation="")
DEPRECATED since v1.60.05: Please use the color attribute
**PsychoPy - Psychology software for Python, Release 2020.2.10**

`setDepth` *(newDepth, operation='', log=None)*
Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message.

`setFlipHoriz` *(newVal=True, log=None)*
If set to True then the movie will be flipped horizontally (left-to-right). Note that this is relative to the original, not relative to the current state.

`setFlipVert` *(newVal=True, log=None)*
If set to True then the movie will be flipped vertically (top-to-bottom). Note that this is relative to the original, not relative to the current state.

`setLMS` *(newLMS, operation='')*
DEPRECATED since v1.60.05: Please use the *color* attribute.

`setMovie` *(filename, log=None)*
See ~MovieStim.loadMovie (the functions are identical). This form is provided for syntactic consistency with other visual stimuli.

`setOpacity` *(newOpacity, operation='', log=None)*
Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message.

`setOri` *(newOri, operation='', log=None)*
Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message.

`setPos` *(newPos, operation='', log=None)*
Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message.

`setRGB` *(newRGB, operation='', log=None)*
DEPRECATED since v1.60.05: Please use the *color* attribute.

`setSize` *(newSize, operation='', units=None, log=None)*
Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message.

`setUseShaders` *(value=True, log=None)*
Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message.

`size`
The size (width, height) of the stimulus in the stimulus *units*.
Value should be *x,y-pair, scalar* (applies to both dimensions) or *None* (resets to default). *Operations* are supported.
Sizes can be negative (causing a mirror-image reversal) and can extend beyond the window.
Example:

```
stim.size = 0.8  # Set size to (xsize, ysize) = (0.8, 0.8)
print(stim.size)  # Outputs array([0.8, 0.8])
stim.size += (0.5, -0.5)  # make wider and flatter: (1.3, 0.3)
```

Tip: if you can see the actual pixel range this corresponds to by looking at *stim._sizeRendered*.

`stop` *(log=None)*
Stop the current point in the movie.
The sound will stop, current frame will not advance. Once stopped the movie cannot be restarted - it must be loaded again. Use pause() if you may need to restart the movie.

**units**

None, ‘norm’, ‘cm’, ‘deg’, ‘degFlat’, ‘degFlatPos’, or ‘pix’

If None then the current units of the Window will be used. See *Units for the window and stimuli* for explanation of other options.

Note that when you change units, you don’t change the stimulus parameters and it is likely to change appearance. Example:

```python
# This stimulus is 20% wide and 50% tall with respect to window
stim = visual.PatchStim(win, units='norm', size=(0.2, 0.5))

# This stimulus is 0.2 degrees wide and 0.5 degrees tall.
stim.units = 'deg'
```

**useShaders**

Should shaders be used to render the stimulus (typically leave as *True*)

If the system support the use of OpenGL shader language then leaving this set to True is highly recommended.

If shaders cannot be used then various operations will be slower (notably, changes to stimulus color or contrast)

**property verticesPix**

This determines the coordinates of the vertices for the current stimulus in pixels, accounting for size, ori, pos and units

**win**

*The Window object in which the* stimulus will be rendered by default. (required)

Example, drawing same stimulus in two different windows and display simultaneously. Assuming that you have two windows and a stimulus (win1, win2 and stim):

```python
stim.win = win1  # stimulus will be drawn in win1
stim.draw()     # stimulus is now drawn to win1
stim.win = win2  # stimulus will be drawn in win2
stim.draw()     # it is now drawn in win2

win1.flip(waitBlanking=False)   # do not wait for next
                               # monitor update
win2.flip()   # wait for vertical blanking.
```

Note that this just changes **default** window for stimulus.

You could also specify window-to-draw-to when drawing::

```python
stim.draw(win1)
stim.draw(win2)
```
8.2.15 NoiseStim

Attributes

Details

class psychopy.visual.NoiseStim(*args, **kwargs)
A stimulus with 2 textures: a random noise sample and a mask

Example:

```python
noise1 = noise = visual.NoiseStim(
    win=win, name='noise', units='pix', noiseImage='testImg.jpg', mask='circle', ori=1.0, pos=(0, 0),
    size=(512, 512), sf=None, phase=0, color=[1,1,1], colorSpace='rgb', opacity=1, blendmode='add',
    contrast=1.0, texRes=512, filter='None', imageComponent='Phase' noiseType='Gabor', noiseElementSize=4, 
    noiseBaseSf=32.0/512, noiseBW=1.0, noiseBWO=30, noiseFractalPower=-1, 
    noiseFilterLower=3/512, noiseFilterUpper=8.0/512.0, noiseFilterOrder=3.0, noiseClip=3.0,
    filter=False, interpolate=False, depth=-1.0)
```

# gives a circular patch of noise made up of scattered Gabor elements with peak frequency = 32.0/512 cycles
# per pixel, # orientation = 0 , frequency bandwidth = 1 octave and orientation bandwidth 30 degrees

Types of noise available

- Binary, Normal, Uniform - pixel based noise samples drawn from a binary (blank and white), normal or uniform distribution respectively. Binary noise is always exactly zero mean, Normal and Uniform are approximately so.

Parameters - noiseElementSize - (can be a tuple) defines the size of the noise elements in the components units.

noiseClip the values in normally distributed noise are divided by noiseClip to limit excessively high or low values.

However, values can still go out of range -1 to 1 which will throw a soft error message high values of noiseClip are recommended if using 'Normal'

Gabor, Isotropic - Effectively a dense scattering of Gabor elements with random amplitude and fixed orientation for Gabor

Parameters - noiseBaseSf - centre spatial frequency in the component units. noiseBW - spatial frequency bandwidth full width half height in octaves. ori - center orientation for Gabor noise (works as for gratingStim so twists the final image at render time). noiseBWO - orientation bandwidth for Gabor noise full width half height in degrees. noiseOri - alternative center orientation for Gabor which sets the orientation during the image build rather than at render time.

useful for setting the orientation of a filter to be applied to some other noise type with a different base orientation.

In practice the desired amplitude spectrum for the noise is built in Fourier space with a random phase spectrum. DC term is set to zero - ie zero mean.

Filtered - A white noise sample that has been filtered with a low, high or bandpass Butterworth filter. The initial sample can have its spectrum skewed towards low or high frequencies. The contrast of the noise falls by half its maximum (3dB) at the cutoff frequencies. Parameters - noiseFilterUpper - upper cutoff frequency - if greater than texRes/2 cycles per image low pass filter used.

noiseFilterLower - Lower cutoff frequency - if zero low pass filter used. noiseFilterOrder - The order of the filter controls the steepness of the falloff outside the passband is zero no filter is applied.

noiseFractalPower - spectrum = f^n noiseFractalPower - determines the spatial frequency bias of the initial noise sample. 0 = flat spectrum, negative = low frequency bias, positive = high frequency bias, -1 = fractal or brownian noise. noiseClip - determines clipping values and rescaling factor such that final rms contrast is close to that requested by contrast parameter while keeping pixel values in range -1, 1.
PsychoPy - Psychology software for Python, Release 2020.2.10

White - A short cut to obtain noise with a flat, unfiltered spectrum

noiseClip - determines clipping values and rescaling factor such that final rms contrast is close to that requested by contrast parameter while keeping pixel values in range -1, 1.

In practice the desired amplitude spectrum is built in the Fourier Domain with a random phase spectrum. DC term is set to zero - ie zero mean. Note despite name the noise contains all grey levels.

Image - A noise sample whose spatial frequency spectrum is taken from the supplied image.

Parameters - noiseImage name of nparray or image file from which to take spectrum - should be same size as largest side requested for component if units is pix or tex Res x texRes otherwise

imageComponent: ‘Phase’ randomizes the phase spectrum leaving the amplitude spectrum untouched.
‘Amplitude’ randomizes the amplitude spectrum leaving the phase spectrum untouched - retains spatial structure of image. ‘Neither’ keeps the image as is - but you can now apply a spatial filter to the image.

noiseClip - determines clipping values and rescaling factor such that final rms contrast is close to that requested by contrast parameter while keeping pixel values in range -1, 1.

In practice the desired amplitude spectrum is taken from the image and paired with a random phase spectrum. DC term is set to zero - ie zero mean.

**Filter parameter** If the filter parameter = Butterworth then the a spectral filter defined by the filtered noise parameters will be applied to the other noise types. If the filter parameter = Gabor then the a spectral filter defined by the Gabor noise parameters will be applied to the other noise types. If the filter parameter = Isotropic then the a spectral filter defined by the Isotropic noise parameters will be applied to the other noise types.

Updating noise samples and timing The noise is rebuilt at next call of the draw function whenever a parameter starting 'noise' is notionally changed even if the value does not actually change every time. eg. setting a parameter to update every frame will cause a new noise sample on every frame but see below. A rebuild can also be forced at any time using the buildNoise() function. The updateNoise() function can be used at any time to produce a new random saple of noise without doing a full build. ie it is quicker than a full build. Both buildNoise and updateNoise can be slow for large samples. Samples of Binary, Normal or Uniform noise can usually be made at frame rate using noiseUpdate. Updating or building other noise types at frame rate may result in dropped frames. An alternative is to build a large sample of noise at the start of the routine and place it off the screen then cut a samples out of this at random locations and feed that as a numpmy array into the texture of a visible gratingStim.

Notes on size If units = pix and noiseType = Binary, Normal or Uniform will make noise sample of requested size. If units = pix and noiseType is Gabor, Isotropic, Filtered, White, Coloured or Image will make square noise sample with side length equal that of the largest dimetions requested. if units is not pix will make square noise sample with side length equal to texRes then rescale to present.

Notes on interpolation For pixel based noise interpolation = nearest is usually best. For other noise types linear is better if the size of the noise sample does not match the final size of the image well.

Notes on frequency Frequencies for cutoffs etc are converted between units for you but can be counter intuitive. 1/size is always 1 cycle per image. For the sf (final spatial frequency) parameter itself 1/size (or None for units pix) will faithfully represent the image without further scaling.

Filter cutoff and Gabor/Isotropic base frequencies should not be too high you should aim to keep them below 0.5 c/pixel on the screen. The function will produce an error when it can’t draw the stimulus in the buffer but it may still be wrong when displayed.

Notes on orientation and phase The ori parameter twists the final image so the samples in noiseType Binary, Normal or Uniform will no longer be aligned to the sides of the monitor if ori is not a multiple of 90. Most other noise types look broadly the same for all values of ori but the specific sample shown can be made to rotate by changing ori. The dominant orientation for Gabor noise is determined by ori at render time, not before.
The phase parameter similarly shifts the sample around within the display window at render time and will not choose new random phases for the noise sample.

## 8.2.16 ObjMeshStim

### Attributes

`ObjMeshStim` *(win, objFile[, pos, ori, ...])* Class for loading and presenting 3D stimuli in the Wavefront OBJ format.

### Details

```python
class psychopy.visual.ObjMeshStim(win, objFile, pos=(0, 0, 0), ori=(0, 0, 0, 1), useMaterial=None, loadMtlLib=True, color=(0.0, 0.0, 0.0), colorSpace='rgb', contrast=1.0, opacity=1.0, useShaders=False, name='', autoLog=True)
```

Class for loading and presenting 3D stimuli in the Wavefront OBJ format.

Calling the `draw` method will render the mesh to the current buffer. The render target (FBO or back buffer) must have a depth buffer attached to it for the object to be rendered correctly. Shading is used if the current window has light sources defined and lighting is enabled (by setting `useLights=True` before drawing the stimulus).

Vertex positions, texture coordinates, and normals are loaded and packed into a single vertex buffer object (VBO). Vertex array objects (VAO) are created for each material with an index buffer referencing vertices assigned that material in the VBO. For maximum performance, keep the number of materials per object as low as possible, as switching between VAOs has some overhead.

Material attributes are read from the material library file (*.MTL) associated with the *.OBJ file. This file will be automatically searched for and read during loading. Afterwards you can edit material properties by accessing the data structure of the `materials` attribute.

Keep in mind that OBJ shapes are rigid bodies, the mesh itself cannot be deformed during runtime. However, meshes can be positioned and rotated as desired by manipulating the `RigidBodyPose` instance accessed through the `thePose` attribute.

**Warning:** Loading an *.OBJ file is a slow process, be sure to do this outside of any time-critical routines! This class is experimental and may result in undefined behavior.

### Examples

Loading an *.OBJ file from a disk location:

```python
myObjStim = ObjMeshStim(win, '/path/to/file/model.obj')
```

### Parameters

- **win** (*psychopy.visual.Window*) – Window this stimulus is associated with. Stimuli cannot be shared across windows unless they share the same context.
- **size** (*tuple or float*) – Dimensions of the mesh. If a single value is specified, the plane will be a square. Provide a tuple of floats to specify the width and length of the box (e.g. `size=(0.2, 1.3)`).
• **pos** (*array_like*) – Position vector \([x, y, z]\) for the origin of the rigid body.

• **ori** (*array_like*) – Orientation quaternion \([x, y, z, w]\) where \(x, y, z\) are imaginary and \(w\) is real. If you prefer specifying rotations in axis-angle format, call `setOriAxisAngle` after initialization. By default, the plane is oriented with normal facing the +Z axis of the scene.

• **useMaterial** (*PhongMaterial, optional*) – Material to use for all sub-meshes. The material can be configured by accessing the `material` attribute after initialization. If no material is specified, `color` will modulate the diffuse and ambient colors for all meshes in the model. If `loadMtllib` is `True`, this value should be `None`.

• **loadMtllib** (*bool*) – Load materials from the MTL file associated with the mesh. This will override `useMaterial` if it is `None`. The value of `materials` after initialization will be a dictionary where keys are material names and values are materials. Any textures associated with the model will be loaded as per the material requirements.

• **useShaders** (*bool*) – Use shaders when rendering.

```python
_def_createVAO_(vertices, textureCoords, normals, faces)
Create a vertex array object for handling vertex attribute data.

_getDesiredRGB_(rgb, colorSpace, contrast)
Convert color to RGB while adding contrast. Requires `self.rgb`, `self.colorSpace` and `self.contrast`

_loadMtllib_(mtlFile)
Load a material library associated with the OBJ file. This is usually called by the constructor for this class.

Parameters

- **mtlFile** (*str*) – Path to MTL file.

_selectWindow_(win)
Switch drawing to the specified window. Calls the window’s `setCurrent()` method which handles the switch.

_updateList_()
The user shouldn’t need this method since it gets called after every call to `set()` Chooses between using and not using shaders each call.

**color**
Color of the stimulus

Value should be one of:

- **string**: to specify a *Colors by name*. Any of the standard html/X11 color names
  `<http://www.w3schools.com/html/html_colrnames.asp>` can be used.

- **Colors by hex value**

- **numerically**: (scalar or triplet) for DKL, RGB or other *Color spaces*. For these, *operations* are supported.

When color is specified using numbers, it is interpreted with respect to the stimulus’ current colorSpace. If color is given as a single value (scalar) then this will be applied to all 3 channels.
Examples

For whatever stim you have:

```python
stim.color = 'white'
stim.color = 'RoyalBlue'  # (the case is actually ignored)
stim.color = '#DDA0DD'  # DDA0DD is hexadecimal for plum
stim.color = [1.0, -1.0, -1.0]  # if stim.colorSpace='rgb':
    # a red color in rgb space
stim.color = [0.0, 45.0, 1.0]  # if stim.colorSpace='dkl':
    # DKL space with elev=0, azimuth=45
stim.color = [0, 0, 255]  # if stim.colorSpace='rgb255':
    # a blue stimulus using rgb255 space
stim.color = 255  # interpreted as (255, 255, 255)
    # which is white in rgb255.
```

**Operations** work as normal for all numeric colorSpaces (e.g. 'rgb', 'hsv' and 'rgb255') but not for strings, like named and hex. For example, assuming that colorSpace='rgb':

```python
stim.color += [1, 1, 1]  # increment all guns by 1 value
stim.color *= -1  # multiply the color by -1 (which in this
    # space inverts the contrast)
stim.color *= [0.5, 0, 1]  # decrease red, remove green, keep blue
```

You can use `setColor` if you want to set color and colorSpace in one line. These two are equivalent:

```python
stim.setColor((0, 128, 255), 'rgb255')
    # ... is equivalent to
stim.colorSpace = 'rgb255'
stim.color = (0, 128, 255)
```

colorSpace

The name of the color space currently being used

Value should be: a string or None

For strings and hex values this is not needed. If None the default colorSpace for the stimulus is used (defined during initialisation).

Please note that changing colorSpace does not change stimulus parameters. Thus you usually want to specify colorSpace before setting the color. Example:

```python
# A light green text
stim = visual.TextStim(win, 'Color me!',
    color=(0, 1, 0), colorSpace='rgb')

# An almost-black text
stim.colorSpace = 'rgb255'

# Make it light green again
stim.color = (128, 255, 128)
```

contrast

A value that is simply multiplied by the color

**Value should be:** a float between -1 (negative) and 1 (unchanged). **Operations** supported.

Set the contrast of the stimulus, i.e. scales how far the stimulus deviates from the middle grey. You can also use the stimulus `opacity` to control contrast, but that cannot be negative.
Examples:

<table>
<thead>
<tr>
<th>stim.contrast</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td># unchanged contrast</td>
</tr>
<tr>
<td>0.5</td>
<td># decrease contrast</td>
</tr>
<tr>
<td>0.0</td>
<td># uniform, no contrast</td>
</tr>
<tr>
<td>-0.5</td>
<td># slightly inverted</td>
</tr>
<tr>
<td>-1.0</td>
<td># totally inverted</td>
</tr>
</tbody>
</table>

Setting contrast outside range -1 to 1 is permitted, but may produce strange results if color values exceed the monitor limits:

<table>
<thead>
<tr>
<th>stim.contrast</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.2</td>
<td># increases contrast</td>
</tr>
<tr>
<td>-1.2</td>
<td># inverts with increased contrast</td>
</tr>
</tbody>
</table>

draw (win=None)

Draw the mesh.

Parameters

- **win** (~psychopy.visual.Window) – Window this stimulus is associated with. Stimuli cannot be shared across windows unless they share the same context.

getOri ()

getOriAxisAngle (degrees=True)

Get the axis and angle of rotation for the 3D stimulus. Converts the orientation defined by the ori quaternion to an axis-angle representation.

Parameters

- **degrees** (bool, optional) – Specify True if angle is in degrees, or else it will be treated as radians. Default is True.

Returns

- Axis [rx, ry, rz] and angle.

Return type tuple

getPos ()

getRayIntersectBounds (rayOrig, rayDir)

Get the point which a ray intersects the bounding box of this mesh.

Parameters

- **rayOrig** (array_like) – Origin of the ray in space [x, y, z].
- **rayDir** (array_like) – Direction vector of the ray [x, y, z], should be normalized.

Returns

- Coordinate in world space of the intersection and distance in scene units from rayOrig. Returns None if there is no intersection.

Return type tuple

isVisible ()

Check if the object is visible to the observer.

Test if a pose’s bounding box or position falls outside of an eye’s view frustum.

Poses can be assigned bounding boxes which enclose any 3D models associated with them. A model is not visible if all the corners of the bounding box fall outside the viewing frustum. Therefore any primitives (i.e. triangles) associated with the pose can be culled during rendering to reduce CPU/GPU workload.

Returns

- True if the object’s bounding box is visible.

Return type bool
Examples

You can avoid running draw commands if the object is not visible by doing a visibility test first:

```python
if myStim isVisible():
    myStim.draw()
```

**property ori**
Orientation quaternion (X, Y, Z, W).

**property pos**
Position vector (X, Y, Z).

**setBorderColor** (color, colorSpace=None, operation='', log=None)

**setColor** (color, colorSpace=None, operation='', log=None)

Usually you can use `stim.attribute = value` syntax instead, but use this method if you need to suppress the log message and/or set colorSpace simultaneously.

**setContrast** (newContrast, operation='', log=None)

Usually you can use `stim.attribute = value` syntax instead, but use this method if you need to suppress the log message.

**setFillColor** (color, colorSpace=None, operation='', log=None)

**setOri** (ori)

**setOriAxisAngle** (axis, angle, degrees=True)

Set the orientation of the 3D stimulus using an `axis` and `angle`. This sets the quaternion at `ori`.

Parameters:

- **axis** (array_like) – Axis of rotation [rx, ry, rz].
- **angle** (float) – Angle of rotation.
- **degrees** (bool, optional) – Specify True if `angle` is in degrees, or else it will be treated as radians. Default is True.

**setPos** (pos)

**setUseShaders** (value=True, log=None)

Usually you can use `stim.attribute = value` syntax instead, but use this method if you need to suppress the log message.

**property thePose**

The pose of the rigid body. This is a class which has `pos` and `ori` attributes.

**units**

None, ‘norm’, ‘cm’, ‘deg’, ‘degFlat’, ‘degFlatPos’, or ‘pix’

If None then the current units of the Window will be used. See Units for the window and stimuli for explanation of other options.

Note that when you change units, you don’t change the stimulus parameters and it is likely to change appearance. Example:

```python
# This stimulus is 20% wide and 50% tall with respect to window
stim = visual.PatchStim(win, units='norm', size=(0.2, 0.5))

# This stimulus is 0.2 degrees wide and 0.5 degrees tall.
stim.units = 'deg'
```
**useShaders**

Should shaders be used to render the stimulus (typically leave as True)

If the system support the use of OpenGL shader language then leaving this set to True is highly recommended. If shaders cannot be used then various operations will be slower (notably, changes to stimulus color or contrast)

**win**

The **Window object in which the** stimulus will be rendered by default. (required)

Example, drawing same stimulus in two different windows and display simultaneously. Assuming that you have two windows and a stimulus (win1, win2 and stim):

```python
stim.win = win1  # stimulus will be drawn in win1
stim.draw()     # stimulus is now drawn to win1
stim.win = win2  # stimulus will be drawn in win2
stim.draw()     # it is now drawn in win2
win1.flip(waitBlanking=False)  # do not wait for next monitor update
win2.flip()     # wait for vertical blanking.
```

Note that this just changes *default* window for stimulus. You could also specify window-to-draw-to when drawing:

```python
stim.draw(win1)
stim.draw(win2)
```

### 8.2.17 PatchStim (deprecated)

class `psychopy.visual.PatchStim(*args,**kwargs)`

Deprecated (as of version 1.74.00): please use the GratingStim or the ImageStim classes.

The GratingStim has identical abilities to the PatchStim (but possibly different initial values) whereas the ImageStim is designed to be use for non-cyclic images (photographs, not gratings).

### 8.2.18 BlinnPhongMaterial

**Attributes**

`BlinnPhongMaterial([win, diffuseColor, ...])`  
Class representing a material using the Blinn-Phong lighting model.

**Details**

class `psychopy.visual.BlinnPhongMaterial`(win=None, diffuseColor=(0.5, 0.5, 0.5), specularColor=(-1.0, -1.0, -1.0), ambientColor=(-1.0, -1.0, -1.0), emissionColor=(-1.0, -1.0, -1.0), shininess=10.0, colorSpace='rgb', diffuseTexture=None, opacity=1.0, contrast=1.0, face='front')

Class representing a material using the Blinn-Phong lighting model.

This class stores material information to modify the appearance of drawn primitives with respect to lighting, such as color (diffuse, specular, ambient, and emission), shininess, and textures. Simple materials are intended to work
with features supported by the fixed-function OpenGL pipeline.

If shaders are enabled, the colors of objects will appear different than without. This is due to the lighting/material colors being computed on a per-pixel basis, and the formulation of the lighting model. The Phong shader determines the ambient color/intensity by adding up both the scene and light ambient colors, then multiplies them by the diffuse color of the material, as the ambient light’s color should be a product of the surface reflectance (albedo) and the light color (the ambient light needs to reflect off something to be visible). Diffuse reflectance is Lambertian, where the cosine angle between the incident light ray and surface normal determines color. The size of specular highlights are related to the shininess factor which ranges from 1.0 to 128.0. The greater this number, the tighter the specular highlight making the surface appear smoother. If shaders are not being used, specular highlights will be computed using the Phong lighting model. The emission color is optional, it simply adds to the color of every pixel much like ambient lighting does. Usually, you would not really want this, but it can be used to add bias to the overall color of the shape.

If there are no lights in the scene, the diffuse color is simply multiplied by the scene and material ambient color to give the final color.

Lights are attenuated (fall-off with distance) using the formula:

\[
\text{attenuationFactor} = \frac{1.0}{k0 + k1 \times \text{distance} + k2 \times \text{pow}(\text{distance}, 2)}
\]

The coefficients for attenuation can be specified by setting `attenuation` in the lighting object. Values \(k0=1.0, k1=0.0,\) and \(k2=0.0\) results in a light that does not fall-off with distance.

**Warning:** This class is experimental and may result in undefined behavior.

**Parameters**

- `win (~psychopy.visual.Window or None)` – Window this material is associated with, required for shaders and some color space conversions.
- `diffuseColor (array_like)` – Diffuse material color (r, g, b, a) with values between 0.0 and 1.0.
- `specularColor (array_like)` – Specular material color (r, g, b, a) with values between 0.0 and 1.0.
- `ambientColor (array_like)` – Ambient material color (r, g, b, a) with values between 0.0 and 1.0.
- `emissionColor (array_like)` – Emission material color (r, g, b, a) with values between 0.0 and 1.0.
- `shininess (float)` – Material shininess, usually ranges from 0.0 to 128.0.
- `colorSpace (float)` – Color space for `diffuseColor`, `specularColor`, `ambientColor`, and `emissionColor`.
- `diffuseTexture (TexImage2D)` –
- `opacity (float)` – Opacity of the material. Ranges from 0.0 to 1.0 where 1.0 is fully opaque.
- `contrast (float)` – Contrast of the material colors.
- `face (str)` – Face to apply material to. Values are `front`, `back` or `both`.
- `textures (dict, optional)` – Texture maps associated with this material. Textures are specified as a list. The index of textures in the list will be used to set the corresponding texture unit they are bound to.
property ambientColor
    Ambient color of the material.

property ambientRGB
    Diffuse color of the material.

begin (useTextures=True, useShaders=False)
    Use this material for successive rendering calls.

    Parameters useTextures (bool) – Enable textures.

property diffuseColor
    Diffuse color of the material.

property diffuseRGB
    Diffuse color of the material.

property diffuseTexture
    Diffuse color of the material.

property emissionColor
    Emission color of the material.

property emissionRGB
    Diffuse color of the material.

end (clear=True)
    Stop using this material.

    Must be called after begin before using another material or else later drawing operations may have undefined behavior.

    Upon returning, GL_COLOR_MATERIAL is enabled so material colors will track the current glColor.

    Parameters clear (bool) – Overwrite material state settings with default values. This ensures material colors are set to OpenGL defaults. You can forgo clearing if successive materials are used which overwrite glMaterialfv values for GL_DIFFUSE, GL_SPECULAR, GL_AMBIENT, GL_EMISSION, and GL_SHININESS. This reduces a bit of overhead if there is no need to return to default values intermittently between successive material begin and end calls. Textures and shaders previously enabled will still be disabled.

property shininess

property specularColor
    Specular color of the material.

property specularRGB
    Diffuse color of the material.

8.2.19 Pie

class psychopy.visual.Pie(*args, **kwargs)
    Creates a pie shape which is a circle with a wedge cut-out.

    This shape is sometimes referred to as a Pac-Man shape which is often used for creating Kanizsa figures. However, the shape can be adapted for other uses.

    start, end
    Start and end angles of the filled region of the shape in degrees. Shapes are filled counter clockwise between the specified angles.

    Type float or int
**radius**

Radius of the shape. Avoid using `size` for adjusting figure dimensions if radius \(!= 0.5\) which will result in undefined behavior.

**Type** float or int

**Parameters**

- `win (~psychopy.visual.Window)` – Window this shape is associated with.
- `radius (float or int)` – Radius of the shape. Avoid using `size` for adjusting figure dimensions if radius \(!= 0.5\) which will result in undefined behavior.
- `start (float or int)` – Start and end angles of the filled region of the shape in degrees. Shapes are filled counter clockwise between the specified angles.
- `end (float or int)` – Start and end angles of the filled region of the shape in degrees. Shapes are filled counter clockwise between the specified angles.
- `edges (int)` – Number of edges to use when drawing the figure. A greater number of edges will result in smoother curves, but will require more time to compute.

## 8.2.20 PlaneStim

**Attributes**

`PlaneStim(win[, size, pos, ori, color, ...])` Class for drawing planes.

**Details**

```python
class psychopy.visual.PlaneStim(win, size=(0.5, 0.5), pos=(0.0, 0.0, 0.0), ori=(0.0, 0.0, 0.0, 1.0), color=(0.0, 0.0, 0.0), colorSpace='rgb', contrast=1.0, opacity=1.0, useMaterial=None, useShaders=False, textureScale=None, name='', autoLog=True)
```

Class for drawing planes.

Draws a plane with dimensions specified by `size` (length, width) in scene units.

Calling the `draw` method will render the plane to the current buffer. The render target (FBO or back buffer) must have a depth buffer attached to it for the object to be rendered correctly. Shading is used if the current window has light sources defined and lighting is enabled (by setting `useLights=True` before drawing the stimulus).

**Warning:** This class is experimental and may result in undefined behavior.

**Parameters**

- `win (~psychopy.visual.Window)` – Window this stimulus is associated with. Stimuli cannot be shared across windows unless they share the same context.
- `size (tuple or float)` – Dimensions of the mesh. If a single value is specified, the plane will be a square. Provide a tuple of floats to specify the width and length of the plane (eg. `size=(0.2, 1.3)`).
- `pos (array_like)` – Position vector \([x, y, z]\) for the origin of the rigid body.
• ori (array_like) – Orientation quaternion [x, y, z, w] where x, y, z are imaginary and w is real. If you prefer specifying rotations in axis-angle format, call setOriAxisAngle after initialization. By default, the plane is oriented with normal facing the +Z axis of the scene.

• useMaterial (PhongMaterial, optional) – Material to use. The material can be configured by accessing the material attribute after initialization. If not material is specified, the diffuse and ambient color of the shape will track the current color specified by glColor.

• colorSpace (str) – Color space of color to use.

• contrast (float) – Contrast of the stimulus, value modulates the color.

• opacity (float) – Opacity of the stimulus ranging from 0.0 to 1.0. Note that transparent objects look best when rendered from farthest to nearest.

• textureScale (array_like or float, optional) – Scaling factors for texture coordinates (sx, sy). By default, a factor of 1 will have the entire texture cover the surface of the mesh. If a single number is provided, the texture will be scaled uniformly.

• name (str) – Name of this object for logging purposes.

• autoLog (bool) – Enable automatic logging on attribute changes.

_createVAO (vertices, textureCoords, normals, faces)
Create a vertex array object for handling vertex attribute data.

_getDesiredRGB (rgb, colorSpace, contrast)
Convert color to RGB while adding contrast. Requires self.rgb, self.colorSpace and self.contrast

_selectWindow (win)
Switch drawing to the specified window. Calls the window’s _setCurrent() method which handles the switch.

_updateList ()
The user shouldn’t need this method since it gets called after every call to .set() Chooses between using and not using shaders each call.

color
Color of the stimulus

Value should be one of:

• string: to specify a Colors by name. Any of the standard html/X11 color names <http://www.w3schools.com/html/html_colornames.asp> can be used.

• Colors by hex value

• numerically: (scalar or triplet) for DKL, RGB or other Color spaces. For these, operations are supported.

When color is specified using numbers, it is interpreted with respect to the stimulus’ current colorSpace. If color is given as a single value (scalar) then this will be applied to all 3 channels.
Examples

For whatever stim you have:

```python
stim.color = 'white'
stim.color = 'RoyalBlue'  # (the case is actually ignored)
stim.color = '#DDA0DD'  # DDA0DD is hexadecimal for plum
stim.color = [1.0, -1.0, -1.0]  # if stim.colorSpace='rgb':
    # a red color in rgb space
stim.color = [0.0, 45.0, 1.0]  # if stim.colorSpace='dkl':
    # DKL space with elev=0, azimuth=45
stim.color = [0, 0, 255]  # if stim.colorSpace='rgb255':
    # a blue stimulus using rgb255 space
stim.color = 255  # interpreted as (255, 255, 255)
    # which is white in rgb255.
```

*Operations* work as normal for all numeric colorSpaces (e.g. ‘rgb’, ‘hsv’ and ‘rgb255’) but not for strings, like named and hex. For example, assuming that colorSpace='rgb':

```python
stim.color += [1, 1, 1]  # increment all guns by 1 value
stim.color *= -1  # multiply the color by -1 (which in this
    # space inverts the contrast)
stim.color *= [0.5, 0, 1]  # decrease red, remove green, keep blue
```

You can use `setColor` if you want to set color and colorSpace in one line. These two are equivalent:

```python
stim.setColor((0, 128, 255), 'rgb255')
# ... is equivalent to
stim.colorSpace = 'rgb255'
stim.color = (0, 128, 255)
```

colorSpace

The name of the color space currently being used

Value should be: a string or None

For strings and hex values this is not needed. If None the default colorSpace for the stimulus is used (defined during initialisation).

Please note that changing colorSpace does not change stimulus parameters. Thus you usually want to specify colorSpace before setting the color. Example:

```python
# A light green text
stim = visual.TextStim(win, 'Color me!',
    color=(0, 1, 0), colorSpace='rgb')

# An almost-black text
stim.colorSpace = 'rgb255'

# Make it light green again
stim.color = (128, 255, 128)
```

contrast

A value that is simply multiplied by the color

Value should be: a float between -1 (negative) and 1 (unchanged). *Operations* supported.

Set the contrast of the stimulus, i.e. scales how far the stimulus deviates from the middle grey. You can also use the stimulus *opacity* to control contrast, but that cannot be negative.
Examples:

```
stim.contrast = 1.0  # unchanged contrast
stim.contrast = 0.5  # decrease contrast
stim.contrast = 0.0  # uniform, no contrast
stim.contrast = -0.5  # slightly inverted
stim.contrast = -1.0  # totally inverted
```

Setting contrast outside range -1 to 1 is permitted, but may produce strange results if color values exceeds the
monitor limits.:  

```
stim.contrast = 1.2  # increases contrast
stim.contrast = -1.2  # inverts with increased contrast
```

draw(\texttt{win=None})

Draw the stimulus.

This should work for stimuli using a single VAO and material. More complex stimuli with multiple materials
should override this method to correctly handle that case.

---

**Parameters** \texttt{win (~psychopy.visual.Window)} – Window this stimulus is associated with. Stimuli
cannot be shared across windows unless they share the same context.

---

\textbf{getOri()}

\textbf{getOriAxisAngle (\texttt{degrees=True})}

Get the axis and angle of rotation for the 3D stimulus. Converts the orientation defined by the \texttt{ori}
quaternion to and axis-angle representation.

---

**Parameters** \texttt{degrees (bool, optional)} – Specify \texttt{True} if \texttt{angle} is in degrees, or else it
will be treated as radians. Default is \texttt{True}.

---

**Returns** Axis \texttt{[rx, ry, rz]} and angle.

---

**Return type** tuple

\textbf{getPos()}

\textbf{getRayIntersectBounds (\texttt{rayOrig}, \texttt{rayDir})}

Get the point which a ray intersects the bounding box of this mesh.

---

**Parameters**

- \texttt{rayOrig (array_like)} – Origin of the ray in space \texttt{[x, y, z]}.  
- \texttt{rayDir (array_like)} – Direction vector of the ray \texttt{[x, y, z]}, should be normalized.

---

**Returns** Coordinate in world space of the intersection and distance in scene units from \texttt{rayOrig}.

Returns \texttt{None} if there is no intersection.

---

**Return type** tuple

\textbf{isVisible()}

Check if the object is visible to the observer.

Test if a pose’s bounding box or position falls outside of an eye’s view frustum.

Poses can be assigned bounding boxes which enclose any 3D models associated with them. A model is not
visible if all the corners of the bounding box fall outside the viewing frustum. Therefore any primitives (i.e.
triangles) associated with the pose can be culled during rendering to reduce CPU/GPU workload.

---

**Returns** \texttt{True} if the object’s bounding box is visible.

---

**Return type** bool
Examples

You can avoid running draw commands if the object is not visible by doing a visibility test first:

```python
if myStim.isVisible():
    myStim.draw()
```

**property ori**

Orientation quaternion (X, Y, Z, W).

**property pos**

Position vector (X, Y, Z).

**setBorderColor** `(color, colorSpace=None, operation='', log=None)`

Usually you can use `stim.attribute = value` syntax instead, but use this method if you need to suppress the log message and/or set colorSpace simultaneously.

**setContrast** `(newContrast, operation='', log=None)`

Usually you can use `stim.attribute = value` syntax instead, but use this method if you need to suppress the log message.

**setFillColor** `(color, colorSpace=None, operation='', log=None)`

**setOri** `(ori)`

**setOriAxisAngle** `(axis, angle, degrees=True)`

Set the orientation of the 3D stimulus using an axis and angle. This sets the quaternion at ori.

Parameters

- **axis** *(array_like)* – Axis of rotation [rx, ry, rz].
- **angle** *(float)* – Angle of rotation.
- **degrees** *(bool, optional)* – Specify True if angle is in degrees, or else it will be treated as radians. Default is True.

**setPos** `(pos)`

**setUseShaders** *(value=True, log=None)*

Usually you can use `stim.attribute = value` syntax instead, but use this method if you need to suppress the log message.

**property thePose**

The pose of the rigid body. This is a class which has pos and ori attributes.

**units**

None, ‘norm’, ‘cm’, ‘deg’, ‘degFlat’, ‘degFlatPos’, or ‘pix’

If None then the current units of the Window will be used. See Units for the window and stimuli for explanation of other options.

Note that when you change units, you don’t change the stimulus parameters and it is likely to change appearance. Example:

```python
# This stimulus is 20% wide and 50% tall with respect to window
stim = visual.PatchStim(win, units='norm', size=(0.2, 0.5))

# This stimulus is 0.2 degrees wide and 0.5 degrees tall.
stim.units = 'deg'
```
PsychoPy - Psychology software for Python, Release 2020.2.10

useShaders
Should shaders be used to render the stimulus (typically leave as True)
If the system support the use of OpenGL shader language then leaving this set to True is highly recommended. If shaders cannot be used then various operations will be slower (notably, changes to stimulus color or contrast)

win
The Window object in which the stimulus will be rendered by default. (required)
Example, drawing same stimulus in two different windows and display simultaneously. Assuming that you have two windows and a stimulus (win1, win2 and stim):

```python
stim.win = win1  # stimulus will be drawn in win1
stim.draw()      # stimulus is now drawn to win1
stim.win = win2  # stimulus will be drawn in win2
stim.draw()      # it is now drawn in win2
win1.flip(waitBlanking=False)  # do not wait for next monitor update
win2.flip()      # wait for vertical blanking.
```

Note that this just changes *default* window for stimulus. You could also specify window-to-draw-to when drawing:

```python
stim.draw(win1)
stim.draw(win2)
```

8.2.21 Polygon

class psychopy.visual.Polygon(*args, **kwargs)

Creates a regular polygon (triangles, pentagrams, …). A special case of a ShapeStim.

(New in version 1.72.00)

Polygon accepts all input parameters that ShapeStim accepts, except for vertices and closeShape.

8.2.22 RadialStim

Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RadialStim.win</td>
<td>The Window object in which the stimulus will be rendered.</td>
</tr>
<tr>
<td>RadialStim.tex</td>
<td>Texture to used on the stimulus as a grating (aka carrier)</td>
</tr>
<tr>
<td>RadialStim.mask</td>
<td>The alpha mask that forms the shape of the resulting image.</td>
</tr>
<tr>
<td>RadialStim.units</td>
<td>None, 'norm', 'cm', 'deg', 'degFlat', 'degFlatPos', or 'pix'</td>
</tr>
<tr>
<td>RadialStim.pos</td>
<td>The position of the center of the stimulus in the stimulus units</td>
</tr>
<tr>
<td>RadialStim_ori</td>
<td>The orientation of the stimulus (in degrees).</td>
</tr>
<tr>
<td>RadialStim.size</td>
<td>The size (width, height) of the stimulus in the stimulus units</td>
</tr>
<tr>
<td>RadialStim_contrast</td>
<td>A value that is simply multiplied by the color</td>
</tr>
<tr>
<td>RadialStim_color</td>
<td>Color of the stimulus</td>
</tr>
</tbody>
</table>

continues on next page
Table 11 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RadialStim.colorSpace</td>
<td>The name of the color space currently being used</td>
</tr>
<tr>
<td>RadialStim.opacity</td>
<td>Determines how visible the stimulus is relative to background</td>
</tr>
<tr>
<td>RadialStim.interpolate</td>
<td>Whether to interpolate (linearly) the texture in the stimulus</td>
</tr>
<tr>
<td>RadialStim.setAngularCycles(value[,...])</td>
<td>Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message</td>
</tr>
<tr>
<td>RadialStim.setAngularPhase(value[,...])</td>
<td>Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message</td>
</tr>
<tr>
<td>RadialStim.setRadialCycles(value[,...])</td>
<td>Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message</td>
</tr>
<tr>
<td>RadialStim.setRadialPhase(value[,...])</td>
<td>Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message</td>
</tr>
<tr>
<td>RadialStim.name</td>
<td>String or None.</td>
</tr>
<tr>
<td>RadialStim.autoLog</td>
<td>Whether every change in this stimulus should be auto logged.</td>
</tr>
<tr>
<td>RadialStim.draw([win])</td>
<td>Draw the stimulus in its relevant window.</td>
</tr>
<tr>
<td>RadialStim.autoDraw</td>
<td>Determines whether the stimulus should be automatically drawn on every frame flip.</td>
</tr>
<tr>
<td>RadialStim.clearTextures()</td>
<td>Clear all textures associated with the stimulus.</td>
</tr>
</tbody>
</table>

Details

class psychopy.visual.RadialStim(win, tex='sqrXsqr', mask='none', units='', pos=(0.0, 0.0), size=(1.0, 1.0), radialCycles=3, angularCycles=4, radialPhase=0, angularPhase=0, ori=0.0, texRes=64, angularRes=100, visibleWedge=(0, 360), rgb=None, color=(1.0, 1.0, 1.0), colorSpace='rgb', dkl=None, lms=None, contrast=1.0, opacity=1.0, depth=0, rgbPedestal=(0.0, 0.0, 0.0), interpolate=False, name=None, autoLog=None, maskParams=None)

Stimulus object for drawing radial stimuli.
Examples: annulus, rotating wedge, checkerboard.
Ideal for fMRI retinotopy stimuli!
Many of the capabilities are built on top of the GratingStim.
This stimulus is still relatively new and I’m finding occasional glitches. It also takes longer to draw than a typical GratingStim, so not recommended for tasks where high frame rates are needed.

_classPosRendered()
DEPRECATED in 1.80.00. This functionality is now handled by _updateVertices() and verticesPix.

_classSizeRendered()
DEPRECATED in 1.80.00. This functionality is now handled by _updateVertices() and verticesPix

_createTexture(tex, id, pixFormat, stim, res=128, maskParams=None, forcePOW2=True, dataType=None, wrapping=True)

Params
id: is the texture ID

**pixFormat:** GL.GL_ALPHA, GL.GL_RGB

**useShaders:** bool

**interpolate:** bool (determines whether texture will use GL_LINEAR or GL_NEAREST

**res:** the resolution of the texture (unless a bitmap image is used)

**dataType:** None, GL.GL_UNSIGNED_BYTE, GL_FLOAT. Only affects image files (numpy arrays will be float)

For grating stimuli (anything that needs multiple cycles) forcePOW2 should be set to be True. Otherwise the wrapping of the texture will not work.

```python
_getDesiredRGB(rgb, colorSpace, contrast)
```

Convert color to RGB while adding contrast. Requires self.rgb, self.colorSpace and self.contrast

```python
_getPolyAsRendered()
```

DEPRECATED. Return a list of vertices as rendered.

```python
_selectWindow(win)
```

Switch drawing to the specified window. Calls the window’s _setCurrent() method which handles the switch.

```python
_set(attrib, val, op='', log=None)
```

DEPRECATED since 1.80.04 + 1. Use setAttribute() and val2array() instead.

```python
_setRadialAttribute(attr, value)
```

Internal helper function to reduce redundancy

```python
_updateEverything()
```

Internal helper function for angularRes and visibleWedge (and init)

```python
_updateList()
```

The user shouldn’t need this method since it gets called after every call to .set() Chooses between using and not using shaders each call.

```python
_updateListNoShaders()
```

The user shouldn’t need this method since it gets called after every call to .set() Basically it updates the OpenGL representation of your stimulus if some parameter of the stimulus changes. Call it if you change a property manually rather than using the .set() command

```python
_updateListShaders()
```

The user shouldn’t need this method since it gets called after every call to .set() Basically it updates the OpenGL representation of your stimulus if some parameter of the stimulus changes. Call it if you change a property manually rather than using the .set() command

```python
_updateMaskCoords()
```

_calculate mask coords

```python
_updateTextureCoords()
```

_calculate texture coordinates if angularCycles or Phase change

```python
_updateVertices()
```

Sets Stim.verticesPix and ._borderPix from pos, size, ori, flipVert, flipHoriz

```python
_updateVerticesBase()
```

Update the base vertices if angular resolution changes.

These will be multiplied by the size and rotation matrix before rendering.
angularCycles
Float (but Int is prettiest). Set the number of cycles going around the stimulus. i.e. it controls the number of ‘spokes’.

*Operations* supported.

angularPhase
Float. Set the angular phase (like orientation) of the texture (wraps 0-1).
This is akin to setting the orientation of the texture around the stimulus in radians. If possible, it is more efficient to rotate the stimulus using its ori setting instead.

*Operations* supported.

angularRes
The number of triangles used to make the stimulus.

*Operations* supported.

autoDraw
Determines whether the stimulus should be automatically drawn on every frame flip.
Value should be: *True* or *False*. You do NOT need to set this on every frame flip!

autoLog
Whether every change in this stimulus should be auto logged.
Value should be: *True* or *False*. Set to *False* if your stimulus is updating frequently (e.g. updating its position every frame) and you want to avoid swamping the log file with messages that aren’t likely to be useful.

blendmode
The OpenGL mode in which the stimulus is drawn
Can the ‘avg’ or ‘add’. Average (avg) places the new stimulus over the old one with a transparency given by its opacity. Opaque stimuli will hide other stimuli transparent stimuli won’t. Add performs the arithmetic sum of the new stimulus and the ones already present.

clearTextures()
Clear all textures associated with the stimulus.
As of v1.61.00 this is called automatically during garbage collection of your stimulus, so doesn’t need calling explicitly by the user.

color
Color of the stimulus

*Value should be one of:*

- string: to specify a [*Colors by name*]. Any of the standard html/X11 color names (<http://www.w3schools.com/html/html_colornames.asp>) can be used.

- *Colors by hex value*

- *numerically*: (scalar or triplet) for DKL, RGB or other *Color spaces*. For these, *operations* are supported.

When color is specified using numbers, it is interpreted with respect to the stimulus’ current colorSpace. If color is given as a single value (scalar) then this will be applied to all 3 channels.
Examples

For whatever stim you have:

```
stim.color = 'white'
stim.color = 'RoyalBlue'  # (the case is actually ignored)
stim.color = '#DDA0DD'   # DDA0DD is hexadecimal for plum
stim.color = [1.0, -1.0, -1.0]  # if stim.colorSpace='rgb':
    # a red color in rgb space
stim.color = [0.0, 45.0, 1.0]  # if stim.colorSpace='dkl':
    # DKL space with elev=0, azimuth=45
stim.color = [0, 0, 255]    # if stim.colorSpace='rgb255':
    # a blue stimulus using rgb255 space
stim.color = 255   # interpreted as (255, 255, 255)
    # which is white in rgb255.
```

*Operations* work as normal for all numeric colorSpaces (e.g. ‘rgb’, ‘hsv’ and ‘rgb255’) but not for strings, like named and hex. For example, assuming that colorSpace='rgb':

```
stim.color += [1, 1, 1]   # increment all guns by 1 value
stim.color *= -1         # multiply the color by -1 (which in this
    # space inverts the contrast)
stim.color *= [0.5, 0, 1] # decrease red, remove green, keep blue
```

You can use *setColor* if you want to set color and colorSpace in one line. These two are equivalent:

```
stim.setColor((0, 128, 255), 'rgb255')
  # ... is equivalent to
stim.colorSpace = 'rgb255'
stim.color = (0, 128, 255)
```

**colorSpace**

The name of the color space currently being used

Value should be: a string or None

For strings and hex values this is not needed. If None the default colorSpace for the stimulus is used (defined during initialisation).

Please note that changing colorSpace does not change stimulus parameters. Thus you usually want to specify colorSpace before setting the color. Example:

```
# A light green text
stim = visual.TextStim(win, 'Color me!
    color=(0, 1, 0), colorSpace='rgb')

# An almost-black text
stim.colorSpace = 'rgb255'

# Make it light green again
stim.color = (128, 255, 128)
```

**contains** *(x, y=None, units=None)*

Returns True if a point x,y is inside the stimulus’ border.

Can accept variety of input options:

- two separate args, x and y
- one arg (list, tuple or array) containing two vals (x,y)
• an object with a getPos() method that returns x,y, such as a Mouse.

Returns True if the point is within the area defined either by its border attribute (if one defined), or its vertices attribute if there is no border. This method handles complex shapes, including concavities and self-crossings.

Note that, if your stimulus uses a mask (such as a Gaussian) then this is not accounted for by the contains method; the extent of the stimulus is determined purely by the size, position (pos), and orientation (ori) settings (and by the vertices for shape stimuli).

See Coder demos: shapeContains.py See Coder demos: shapeContains.py

**contrast**

A value that is simply multiplied by the color

Value should be: a float between -1 (negative) and 1 (unchanged). Operations supported.

Set the contrast of the stimulus, i.e. scales how far the stimulus deviates from the middle grey. You can also use the stimulus opacity to control contrast, but that cannot be negative.

Examples:

```python
stim.contrast = 1.0  # unchanged contrast
stim.contrast = 0.5  # decrease contrast
stim.contrast = 0.0  # uniform, no contrast
stim.contrast = -0.5 # slightly inverted
stim.contrast = -1.0 # totally inverted
```

Setting contrast outside range -1 to 1 is permitted, but may produce strange results if color values exceeds the monitor limits.: 

```python
stim.contrast = 1.2  # increases contrast
stim.contrast = -1.2 # inverts with increased contrast
```

**depth**

DEPRECATED. Depth is now controlled simply by drawing order.

**draw** *(win=None)*

Draw the stimulus in its relevant window. You must call this method after every win.flip() if you want the stimulus to appear on that frame and then update the screen again.

If win is specified then override the normal window of this stimulus.

**interpolate**

Whether to interpolate (linearly) the texture in the stimulus

If set to False then nearest neighbour will be used when needed, otherwise some form of interpolation will be used.

**mask**

The alpha mask that forms the shape of the resulting image.

Value should be one of:

- ‘circle’, ‘gauss’, ‘raisedCos’, None (resets to default)
- or the name of an image file (most formats supported)
- or a numpy array (1xN) ranging -1:1

Note that the mask for RadialStim is somewhat different to the mask for ImageStim. For RadialStim it is a 1D array specifying the luminance profile extending outwards from the center of the stimulus, rather than a 2D array
**maskParams**

Various types of input. Default to None.

This is used to pass additional parameters to the mask if those are needed.

- For ‘gauss’ mask, pass dict {'sd': 5} to control standard deviation.
- For the ‘raisedCos’ mask, pass a dict: {'fringeWidth':0.2}, where ‘fringeWidth’ is a parameter (float, 0-1), determining the proportion of the patch that will be blurred by the raised cosine edge.

**name**

String or None. The name of the object to be using during logged messages about this stim. If you have multiple stimuli in your experiment this really helps to make sense of log files!

If name = None your stimulus will be called “unnamed <type>”, e.g. visual.TextStim(win) will be called “unnamedTextStim” in the logs.

**opacity**

Determines how visible the stimulus is relative to background.

The value should be a single float ranging 1.0 (opaque) to 0.0 (transparent). Operations are supported. Precisely how this is used depends on the Blend Mode.

**ori**

The orientation of the stimulus (in degrees).

Should be a single value (scalar). Operations are supported.

Orientation convention is like a clock: 0 is vertical, and positive values rotate clockwise. Beyond 360 and below zero values wrap appropriately.

**overlaps (polygon)**

Returns True if this stimulus intersects another one.

If polygon is another stimulus instance, then the vertices and location of that stimulus will be used as the polygon. Overlap detection is typically very good, but it can fail with very pointy shapes in a crossed-swords configuration.

Note that, if your stimulus uses a mask (such as a Gaussian blob) then this is not accounted for by the overlaps method; the extent of the stimulus is determined purely by the size, pos, and orientation settings (and by the vertices for shape stimuli).

See coder demo, shapeContains.py

**phase**

Phase of the stimulus in each dimension of the texture.

Should be an x,y-pair or scalar

NB phase has modulus 1 (rather than 360 or 2*pi) This is a little unconventional but has the nice effect that setting phase=t*n drifts a stimulus at n Hz

**pos**

The position of the center of the stimulus in the stimulus units

value should be an x,y-pair. Operations are also supported.

Example:

```python
stim.pos = (0.5, 0)  # Set slightly to the right of center
stim.pos += (0.5, -1)  # Increment pos rightwards and upwards.
    Is now (1.0, -1.0)
stim.pos *= 0.2  # Move stim towards the center.
    Is now (0.2, -0.2)
```
Tip: If you need the position of stim in pixels, you can obtain it like this:

```python
from psychopy.tools.monitorunittools import posToPix
posPix = posToPix(stim)
```

**radialCycles**
Float (but Int is prettiest). Set the number of texture cycles from centre to periphery, i.e. it controls the number of 'rings'.

*Operations supported.*

**radialPhase**
Float. Set the radial phase of the texture (wraps 0-1). This is the phase of the texture from the centre to the perimeter of the stimulus (in radians). Can be used to drift concentric rings out/inwards.

*Operations supported.*

**setAngularCycles** *(value, operation='', log=None)*
Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

**setAngularPhase** *(value, operation='', log=None)*
Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

**setAutoDraw** *(value, log=None)*
Sets autoDraw. Usually you can use 'stim.attribute = value' syntax instead, but use this method to suppress the log message.

**setAutoLog** *(value=True, log=None)*
Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

**setBlendmode** *(value, log=None)*
DEPRECATED. Use 'stim.parameter=value' syntax instead

**setBorderColor** *(color, colorSpace=None, operation='', log=None)*

**setColor** *(color, colorSpace=None, operation='', log=None)*
Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message and/or set colorSpace simultaneously.

**setContrast** *(newContrast, operation='', log=None)*
Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

**setDKL** *(newDKL, operation='')*
DEPRECATED since v1.60.05: Please use the *color* attribute

**setDepth** *(newDepth, operation='', log=None)*
Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

**setFillColor** *(color, colorSpace=None, operation='', log=None)*

**setLMS** *(newLMS, operation='')*
DEPRECATED since v1.60.05: Please use the *color* attribute

**setMask** *(value, log=None)*
Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message
setOpacity (newOpacity, operation='', log=None)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setOri (newOri, operation='', log=None)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setPhase (value, operation='', log=None)

DEPRECATED. Use 'stim.parameter = value' syntax instead

setPos (newPos, operation='', log=None)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

setRGB (newRGB, operation='', log=None)

DEPRECATED since v1.60.05: Please use the color attribute

setRadialCycles (value, operation='', log=None)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setRadialPhase (value, operation='', log=None)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setSF (value, operation='', log=None)

DEPRECATED. Use 'stim.parameter = value' syntax instead

setSize (newSize, operation='', units=None, log=None)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setTex (value, log=None)

DEPRECATED. Use 'stim.parameter = value' syntax instead

setUseShaders (value=True, log=None)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

sf

Spatial frequency of the grating texture

Should be a float or scalar or None. If units == 'deg' or 'cm' units are in cycles per deg or cm as appropriate.

If units == 'norm' then sf units are in cycles per stimulus (and so SF scales with stimulus size).

If texture is an image loaded from a file then sf=None defaults to 1/stimSize to give one cycle of the image.

size

The size (width, height) of the stimulus in the stimulus units

Value should be float or scalar (applies to both dimensions) or None (resets to default). Operations are supported.

Sizes can be negative (causing a mirror-image reversal) and can extend beyond the window.

Example:
PsychoPy - Psychology software for Python, Release 2020.2.10

```
stim.size = 0.8  # Set size to (xsize, ysize) = (0.8, 0.8)
print(stim.size)  # Outputs array([0.8, 0.8])
stim.size += (0.5, -0.5)  # make wider and flatter: (1.3, 0.3)
```

Tip: if you can see the actual pixel range this corresponds to by looking at `stim._sizeRendered`

tex
Texture to used on the stimulus as a grating (aka carrier)

**This can be one of various options:**
- ‘sin’, ‘sqr’, ‘saw’, ‘tri’, None (resets to default)
- the name of an image file (most formats supported)
- a numpy array (1xN or NxN) ranging -1:1

If specifying your own texture using an image or numpy array you should ensure that the image has square power-of-two dimensions (e.g. 256 x 256). If not then PsychoPy will upsample your stimulus to the next larger power of two.

texRes
Power-of-two int. Sets the resolution of the mask and texture. texRes is overridden if an array or image is provided as mask.

**Operations** supported.

units
None, ‘norm’, ‘cm’, ‘deg’, ‘degFlat’, ‘degFlatPos’, or ‘pix’

If None then the current units of the Window will be used. See *Units for the window and stimuli* for explanation of other options.

Note that when you change units, you don’t change the stimulus parameters and it is likely to change appearance.

```
# This stimulus is 20% wide and 50% tall with respect to window
stim = visual.PatchStim(win, units='norm', size=(0.2, 0.5))

# This stimulus is 0.2 degrees wide and 0.5 degrees tall.
stim.units = 'deg'
```

useShaders
Should shaders be used to render the stimulus (typically leave as *True*)

If the system support the use of OpenGL shader language then leaving this set to True is highly recommended. If shaders cannot be used then various operations will be slower (notably, changes to stimulus color or contrast)

**property verticesPix**
This determines the coordinates of the vertices for the current stimulus in pixels, accounting for size, ori, pos and units

**visibleWedge**
tuple (start, end) in degrees. Determines visible range.

(0, 360) is full visibility.

**Operations** supported.

**win**

*The Window object in which the* stimulus will be rendered by default. (required)
Examples, drawing same stimulus in two different windows and display simultaneously. Assuming that you have two windows and a stimulus (win1, win2 and stim):

```python
stim.win = win1  # stimulus will be drawn in win1
stim.draw()     # stimulus is now drawn to win1
stim.win = win2  # stimulus will be drawn in win2
stim.draw()     # it is now drawn in win2
win1.flip(waitBlanking=False)  # do not wait for next
                           # monitor update
win2.flip()     # wait for vertical blanking.
```

Note that this just changes *default* window for stimulus. You could also specify window-to-draw-to when drawing::

```python
stim.draw(win1)
stim.draw(win2)
```

### 8.2.23 RatingScale

class `psychopy.visual.RatingScale(*args,**kwargs)`

A class for obtaining ratings, e.g., on a 1-to-7 or categorical scale.

A RatingScale instance is a re-usable visual object having a `draw()` method, with customizable appearance and response options. `draw()` displays the rating scale, handles the subject’s mouse or key responses, and updates the display. When the subject accepts a selection, `.noResponse` goes `False` (i.e., there is a response).

You can call the `getRating()` method anytime to get a rating, `getRT()` to get the decision time, or `getHistory()` to obtain the entire set of (rating, RT) pairs.

There are five main elements of a rating scale: the `scale` (text above the line intended to be a reminder of how to use the scale), the `line` (with tick marks), the `marker` (a moveable visual indicator on the line), the `labels` (text below the line that label specific points), and the `accept` button. The appearance and function of elements can be customized by the experimenter; it is not possible to orient a rating scale to be vertical. Multiple scales can be displayed at the same time, and continuous real-time ratings can be obtained from the history.

The Builder RatingScale component gives a restricted set of options, but also allows full control over a RatingScale via the ‘customize_everything’ field.

A RatingScale instance has no idea what else is on the screen. The experimenter has to draw the item to be rated, and handle `escape` to break or quit, if desired. The subject can use the mouse or keys to respond. Direction keys (left, right) will move the marker in the smallest available increment (e.g., 1/10th of a tick-mark if precision = 10).

**Example 1:**

A basic 7-point scale:

```python
ratingScale = visual.RatingScale(win)
item = <statement, question, image, movie, ..>
while ratingScale.noResponse:
    item.draw()
    ratingScale.draw()
    win.flip()
rating = ratingScale.getRating()
decisionTime = ratingScale.getRT()
choiceHistory = ratingScale.getHistory()
```

**Example 2:**
For fMRI, sometimes only a keyboard can be used. If your response box sends keys 1-4, you could specify left, right, and accept keys, and not need a mouse:

```python
ratingScale = visual.RatingScale(
    win, low=1, high=5, markerStart=4,
    leftKeys='1', rightKeys = '2', acceptKeys='4')
```

**Example 3:**

Categorical ratings can be obtained using choices:

```python
ratingScale = visual.RatingScale(
    win, choices=['agree', 'disagree'],
    markerStart=0.5, singleClick=True)
```

For other examples see Coder Demos -> stimuli -> ratingScale.py.

**Authors**

- 2010 Jeremy Gray: original code and on-going updates
- 2012 Henrik Singmann: tickMarks, labels, ticksAboveLine
- 2014 Jeremy Gray: multiple API changes (v1.80.00)

**Parameters**

- `win`: A `Window` object (required).
- `choices`: A list of items which the subject can choose among. `choices` takes precedence over `low`, `high`, `precision`, `scale`, `labels`, and `tickMarks`.
- `low`: Lowest numeric rating (integer), default = 1.
- `high`: Highest numeric rating (integer), default = 7.
- `precision`: Portions of a tick to accept as input [1, 10, 60, 100]; default = 1 (a whole tick). Pressing a key in `leftKeys` or `rightKeys` will move the marker by one portion of a tick. `precision=60` is intended to support ratings of time-based quantities, with seconds being fractional minutes (or minutes being fractional hours). The display uses a colon (min:sec, or hours:min) to signal this to participants. The value returned by `getRating()` will be a proportion of a minute (e.g., 1:30 -> 1.5, or 59 seconds -> 59/60 = 0.98333). Hours:minutes:seconds is not supported.
- `scale`: Optional reminder message about how to respond or rate an item, displayed above the line; default = '<low>=not at all, <high>=extremely'. To suppress the scale, set `scale=None`.
- `labels`: Text to be placed at specific tick marks to indicate their value. Can be just the ends (if given 2 labels), ends + middle (if given 3 labels), or all points (if given the same number of labels as points).
- `tickMarks`: List of positions at which tick marks should be placed from low to high. The default is to space tick marks equally, one per integer value.
- `tickHeight`: The vertical height of tick marks: 1.0 is the default height (above line), -1.0 is below the line, and 0.0 suppresses the display of tickmarks. `tickHeight` is purely cosmetic, and can be fractional, e.g., 1.2.
- `marker`: The moveable visual indicator of the current selection. The predefined styles are 'triangle', 'circle', 'glow', 'slider', and 'hover'. A slider moves smoothly when there are enough screen positions to move through, e.g., low=0, high=100. Hovering requires a set of choices, and allows clicking directly on individual choices; dwell-time is not recorded. Can also be set to a custom marker stimulus: any object with a .draw() method and .pos will work, e.g., `visual.TextStim(win, text='[]', units='norm').`
markerStart : The location or value to be pre-selected upon initial display, either numeric or one of the choices. Can be fractional, e.g., midway between two options.

markerColor : Color to use for a predefined marker style, e.g., ‘DarkRed’.

markerExpansion : Only affects the glow marker: How much to expand or contract when moving rightward; 0 = none, negative shrinks.

singleClick : Enable a mouse click to both select and accept the rating, default = False. A legal key press will also count as a singleClick. The ‘accept’ box is visible, but clicking it has no effect.

pos [tuple (x, y)] Position of the rating scale on the screen. The midpoint of the line will be positioned at (x, y); default = (0.0, -0.4) in norm units.

size : How much to expand or contract the overall rating scale display. Default size = 1.0. For larger than the default, set size > 1; for smaller, set < 1.

stretch: Like size, but only affects the horizontal direction.

textSize : The size of text elements, relative to the default size (i.e., a scaling factor, not points).

textColor : Color to use for labels and scale text; default = ‘LightGray’.

textFont : Name of the font to use; default = ‘Helvetica Bold’.

showValue : Show the subject their current selection default = True. Ignored if singleClick is True.

showAccept : Show the button to click to accept the current value by using the mouse; default = True.

acceptPreText : The text to display before any value has been selected.

acceptText : The text to display in the ‘accept’ button after a value has been selected.

acceptSize : The width of the accept box relative to the default (e.g., 2 is twice as wide).

acceptKeys : A list of keys that are used to accept the current response; default = ‘return’.

leftKeys : A list of keys that each mean “move leftwards”; default = ‘left’.

rightKeys : A list of keys that each mean “move rightwards”; default = ‘right’.

respKeys : A list of keys to use for selecting choices, in the desired order. The first item will be the left-most choice, the second item will be the next choice, and so on.

skipKeys : List of keys the subject can use to skip a response, default = ‘tab’. To require a response to every item, set skipKeys=None.

lineColor : The RGB color to use for the scale line, default = ‘White’.

mouseOnly : Require the subject to use the mouse (any keyboard input is ignored), default = False. Can be used to avoid competing with other objects for keyboard input.

noMouse: Require the subject to use keys to respond; disable and hide the mouse. markerStart will default to the left end.

minTime : Seconds that must elapse before a response can be accepted, default = 0.4.

maxTime : Seconds after which a response cannot be accepted. If maxTime <= minTime, there’s no time limit. Default = 0.0 (no time limit).

disappear : Whether the rating scale should vanish after a value is accepted. Can be useful when showing multiple scales.

flipVert : Whether to mirror-reverse the rating scale in the vertical direction.
8.2.24 Rect

```python
class psychopy.visual.Rect(*args, **kwargs)
```

Creates a rectangle of given width and height as a special case of a `ShapeStim`

(New in version 1.72.00)

- **width**, **height**
  - The width and height of the rectangle. Values are aliased with fields in the `size` attribute. Use these values to adjust the size of the rectangle in a single dimension after initialization.
  - **Type**: `float` or `int`

**Parameters**

- **win** (~`psychopy.visual.Window`) – Window object to be associated with this stimuli.
- **width** (`float` or `int`) – The width and height of the rectangle. **DEPRECATED** use `size` to define the dimensions of the rectangle on initialization. If `size` is specified the values of `width` and `height` are ignored. This is to provide legacy compatibility for existing applications.
- **height** (`float` or `int`) – The width and height of the rectangle. **DEPRECATED** use `size` to define the dimensions of the rectangle on initialization. If `size` is specified the values of `width` and `height` are ignored. This is to provide legacy compatibility for existing applications.
- **size** (`array_like, float` or `int`) – Width and height of the rectangle as (w, h) or [w, h]. If a single value is provided, the width and height will be set to the same specified value. If `None` is specified, the `size` will be set with values passed to `width` and `height`.

8.2.25 `psychopy.visual.Rift`

**Overview**

- **Rift**([`fovType`, `trackingOriginType`, …])
  - Class provides a display and peripheral interface for the Oculus Rift (see: [https://www.oculus.com/](https://www.oculus.com/)) head-mounted display.
- **Rift.close()**
  - Close the window and cleanly shutdown the LibOVR session.
- **Rift.size**
  - Size property to get the dimensions of the view buffer instead of the window.
- **Rift.setSize(value[, log])**
- **Rift.perfHudMode([mode])**
  - Set the performance HUD mode.
- **Rift.hidePerfHud()**
  - Hide the performance HUD.
- **Rift.stereoDebugHudMode(mode)**
  - Set the debug stereo HUD mode.
- **Rift.setStereoDebugHudOption(option, value)**
  - Configure stereo debug HUD guides.
- **Rift.userHeight**
  - Get user height in meters (`float`).
- **Rift.eyeHeight**
  - Eye height in meters (`float`).
- **Rift.eyeToNoseDistance**
  - Eye to nose distance in meters (`float`).
- **Rift.eyeOffset**
  - Eye separation in centimeters (`float`).
- **Rift.hasPositionTracking**
  - True if the HMD is capable of tracking position.
- **Rift.hasOrientationTracking**
  - True if the HMD is capable of tracking orientation.
- **Rift.hasMagYawCorrection**
  - True if this HMD supports yaw drift correction.
- **Rift.manufacturer**
  - Get the connected HMD’s manufacturer (`str`).

continues on next page
Table 12 – continued from previous page

Rift.serialNumber
Get the connected HMD’s unique serial number (str).

Rift.hid
USB human interface device (HID) identifiers (int, int).

Rift.displayResolution
Get the HMD’s raster display size (int, int).

Rift.displayRefreshRate
Get the HMD’s display refresh rate in Hz (float).

Rift.pixelsPerTanAngleAtCenter
Horizontal and vertical pixels per tangent angle (=1) at the center of the display.

Rift.tanAngleToNDC(horzTan, vertTan)
Convert tan angles to the normalized device coordinates for the current buffer.

Rift.trackerCount
Number of attached trackers.

Rift.getTrackerInfo(trackerIdx)
Get tracker information.

Rift.headLocked
True if head locking is enabled.

Rift.trackingOriginType
Current tracking origin type (str).

Rift.recenterTrackingOrigin()
Recenter the tracking origin using the current head position.

Rift.specifyTrackingOrigin(pose)
Specify a tracking origin.

Rift.specifyTrackingOriginPosOri([pos, ori])
Specify a tracking origin using a pose and orientation.

Rift.clearShouldRecenterFlag()
Clear the ‘shouldRecenter’ status flag at the API level.

Rift.testBoundary(deviceType[, boundary-Type])
Test if tracked devices are colliding with the play area boundary.

Rift.sensorSampleTime
Sensor sample time (float).

Rift.getDevicePose(deviceName[, absTime, …])
Get the pose of a tracked device. For head (HMD) and hand poses (Touch controllers) it is better to use :py:method:`getTrackingState` instead.

Rift.getTrackingState([absTime, latency-Marker])
Get the tracking state of the head and hands.

Rift.calcEyePoses(headPose[, originPose])
Calculate eye poses for rendering.

Rift.eyeRenderPose
Computed eye pose for the current buffer.

Rift.shouldQuit
True if the user requested the application should quit through the headset’s interface.

Rift.isVisible
True if the app has focus in the HMD and is visible to the viewer.

Rift.hmdMounted
True if the HMD is mounted on the user’s head.

Rift.hmdPresent
True if the HMD is present.

Rift.shouldRecenter
True if the user requested the origin be re-centered through the headset’s interface.

Rift.hasInputFocus
True if the application currently has input focus.

Rift.overlayPresent
Set the active draw buffer.

Rift.getPredictedDisplayTime()
Get the predicted time the next frame will be displayed on the HMD.

Rift.getTimeInSeconds()
Absolute time in seconds.

Rift.viewMatrix
The view matrix for the current eye buffer. Only valid after a :py:method:`calcEyePoses` call. Note that setting viewMatrix manually will break visibility culling.

Rift.nearClip
Distance to the near clipping plane in meters.

Rift.farClip
Distance to the far clipping plane in meters.

Rift.projectionMatrix
Get the projection matrix for the current eye buffer.

Rift.isBoundaryVisible
True if the VR boundary is visible.

Rift.getBoundaryDimensions([boundaryType])
Get boundary dimensions.

Rift.connectedControllers
Connected controller types (list of str)

continues on next page
Table 12 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Rift.updateInputState([controllers])</code></td>
<td>Update all connected controller states.</td>
</tr>
<tr>
<td><code>Rift.flip([clearBuffer, drawMirror])</code></td>
<td>Submit view buffer images to the HMD’s compositor for display at next V-SYNC and draw the mirror texture to the on-screen window.</td>
</tr>
<tr>
<td><code>Rift.multiplyViewMatrixGL()</code></td>
<td>Multiply the local eye pose transformation modelMatrix obtained from the SDK using <code>glMultMatrixf</code>.</td>
</tr>
<tr>
<td><code>Rift.multiplyProjectionMatrixGL()</code></td>
<td>Multiply the current projection modelMatrix obtained from the SDK using <code>glMultMatrixf</code>.</td>
</tr>
<tr>
<td><code>Rift.setRiftView([clearDepth])</code></td>
<td>Set head-mounted display view.</td>
</tr>
<tr>
<td><code>Rift.setDefaultView([clearDepth])</code></td>
<td>Return to default projection.</td>
</tr>
<tr>
<td><code>Rift.getThumbstickValues([controller, dead-zone])</code></td>
<td>Get controller thumbstick values.</td>
</tr>
<tr>
<td><code>Rift.getIndexTriggerValues([controller,...])</code></td>
<td>Get the values of the index triggers.</td>
</tr>
<tr>
<td><code>Rift.getHandTriggerValues([controller, dead-zone])</code></td>
<td>Get the values of the hand triggers.</td>
</tr>
<tr>
<td><code>Rift.getButtons(buttons[, controller, testState])</code></td>
<td>Get button states from a controller.</td>
</tr>
<tr>
<td><code>Rift.getTouches(touches[, controller, testState])</code></td>
<td>Get touch states from a controller.</td>
</tr>
<tr>
<td><code>Rift.startHaptics(controller[, frequency,...])</code></td>
<td>Start haptic feedback (vibration).</td>
</tr>
<tr>
<td><code>Rift.stopHaptics(controller)</code></td>
<td>Stop haptic feedback.</td>
</tr>
<tr>
<td><code>Rift.createHapticsBuffer(samples)</code></td>
<td>Create a new haptics buffer.</td>
</tr>
<tr>
<td><code>Rift.submitControllerVibration(controller, ...)</code></td>
<td>Submit a haptics buffer to begin controller vibration.</td>
</tr>
<tr>
<td><code>Rift.createPose([pos, ori])</code></td>
<td>Create a new Rift pose object (LibOVRPose).</td>
</tr>
<tr>
<td><code>Rift.createBoundingBox([extents])</code></td>
<td>Create a new bounding box object (LibOVRBounds).</td>
</tr>
<tr>
<td><code>Rift.isPoseVisible(pose)</code></td>
<td>Check if a pose object is visible to the present eye.</td>
</tr>
</tbody>
</table>

Details

class psychopy.visual.rift.Rift (fovType='recommended', trackingOriginType='floor', texelsPerPixel=1.0, headLocked=False, highQuality=True, monoscopic=False, samples=1, mirrorMode='default', mirrorRes=None, warnAppFrameDropped=True, autoUpdateInput=True, legacyOpenGL=True, *args, **kwargs)

Class provides a display and peripheral interface for the Oculus Rift (see: https://www.oculus.com/) head-mounted display.

Parameters

- `fovType (str)` – Field-of-view (FOV) configuration type. Using 'recommended' auto-configures the FOV using the recommended parameters computed by the runtime. Using 'symmetric' forces a symmetric FOV using optimal parameters from the SDK, this mode is required for displaying 2D stimuli. Specifying 'max' will use the maximum FOVs supported by the HMD.

- `trackingOriginType (str)` – Specify the HMD origin type. If 'floor', the height of the user is added to the head tracker by LibOVR.

- `texelsPerPixel (float)` – Texture pixels per display pixel at FOV center. A value of 1.0 results in 1:1 mapping. A fractional value results in a lower resolution draw buffer which may increase performance.

- `headLocked (bool)` – Lock the compositor render layer in-place, disabling Asynchronous Space Warp (ASW). Enable this if you plan on computing eye poses using custom or modified head poses.
• **highQuality** *(bool)* – Configure the compositor to use anisotropic texture sampling (4x). This reduces aliasing artifacts resulting from high frequency details particularly in the periphery.

• **nearClip** *(float)* – Location of the near and far clipping plane in GL units (meters by default) from the viewer. These values can be updated after initialization.

• **farClip** *(float)* – Location of the near and far clipping plane in GL units (meters by default) from the viewer. These values can be updated after initialization.

• **monoscopic** *(bool)* – Enable monoscopic rendering mode which presents the same image to both eyes. Eye poses used will be both centered at the HMD origin. Monoscopic mode uses a separate rendering pipeline which reduces VRAM usage. When in monoscopic mode, you do not need to call ‘setBuffer’ prior to rendering (doing so will do have no effect).

• **samples** *(int or str)* – Specify the number of samples for multi-sample anti-aliasing (MSAA). When >1, multi-sampling logic is enabled in the rendering pipeline. If ‘max’ is specified, the largest number of samples supported by the platform is used. If floating point textures are used, MSAA sampling is disabled. Must be power of two value.

• **mirrorMode** *(str)* – On-screen mirror mode. Values ‘left’ and ‘right’ show rectilinear images of a single eye. Value ‘distortion’ shows the post-distortion image after being processed by the compositor. Value ‘default’ displays rectilinear images of both eyes side-by-side.

• **mirrorRes** *(list of int)* – Resolution of the mirror texture. If None, the resolution will match the window size. The value of mirrorRes is used for to define the resolution of movie frames.

• **warnAppFrameDropped** *(bool)* – Log a warning if the application drops a frame. This occurs when the application fails to submit a frame to the compositor on-time. Application frame drops can have many causes, such as running routines in your application loop that take too long to complete. However, frame drops can happen sporadically due to driver bugs and running background processes (such as Windows Update). Use the performance HUD to help diagnose the causes of frame drops.

• **autoUpdateInput** *(bool)* – Automatically update controller input states at the start of each frame. If False, you must manually call updateInputState before getting input values from LibOVR managed input devices.

• **legacyOpenGL** *(bool)* – Disable ‘immediate mode’ OpenGL calls in the rendering pipeline. Specifying False maintains compatibility with existing PsychoPy stimuli drawing routines. Use True when computing transformations using some other method and supplying shaders matrices directly.

```python
_assignFlipTime(obj, attrib)

Helper function to assign the time of last flip to the obj.attrib

Parameters

• obj *(dict or object)* – A mutable object (usually a dict of class instance).

• attrib *(str)* – Key or attribute of obj to assign the flip time to.

_checkMatchingSizes(requested, actual)

Checks whether the requested and actual screen sizes differ. If not then a warning is output and the window size is set to actual

_endOfFlip(clearBuffer)

Override end of flip with custom color channel masking if required.

_getFrame(rect=None, buffer='mirror')

Return the current HMD view as an image.
```
Parameters

- **rect** (*array_like*) – Rectangle [x, y, w, h] defining a sub-region of the frame to capture. This should remain `None` for HMD frames.

- **buffer** (*str, optional*) – Buffer to capture. For the HMD, only ‘mirror’ is available at this time.

Returns Buffer pixel contents as a PIL/Pillow image object.

Return type Image

__getRegionOfFrame__ *(rect=-1, 1, 1, -1), buffer='front', power2=False, squarePower2=False)*

Deprecated function, here for historical reasons. You may now use `py:attr:`~Window._getFrame()` and specify a rect to get a sub-region, just as used here.

power2 can be useful with older OpenGL versions to avoid interpolation in PatchStim. If power2 or squarePower2, it will expand rect dimensions up to next power of two. squarePower2 uses the max dimensions. You need to check what your hardware & OpenGL supports, and call __getRegionOfFrame__ as appropriate.

__prepareMonoFrame__ *(clear=True)*

Prepare a frame for monoscopic rendering. This is called automatically after __startHmdFrame__() if monoscopic rendering is enabled.

__renderFBO__ *

Perform a warp operation.

(in this case a copy operation without any warping)

__resolveMSAA__ *

Resolve multisample anti-aliasing (MSAA). If MSAA is enabled, drawing operations are diverted to a special multisample render buffer. Pixel data must be ‘resolved’ by blitting it to the swap chain texture. If not, the texture will be blank.

Notes

You cannot perform operations on the default FBO (at framebuffer) when MSAA is enabled. Any changes will be over-written when ‘flip’ is called.

__setCurrent__ *

Make this window’s OpenGL context current.

If called on a window whose context is current, the function will return immediately. This reduces the number of redundant calls if no context switch is required. If useFBO=True, the framebuffer is bound after the context switch.

__setupFrameBuffer__ *

Override the default framebuffer init code in window.Window to use the HMD swap chain. The HMD’s swap texture and render buffer are configured here.

If multisample anti-aliasing (MSAA) is enabled, a secondary render buffer is created. Rendering is diverted to the multi-sample buffer when drawing, which is then resolved into the HMD’s swap chain texture prior to committing it to the chain. Consequently, you cannot pass the texture attached to the FBO specified by framebuffer until the MSAA buffer is resolved. Doing so will result in a blank texture.

__setupGL__ *

Setup OpenGL state for this window.
_setupGamma (gammaVal)
A private method to work out how to handle gamma for this Window given that the user might have specified an explicit value, or maybe gave a Monitor.

_startHmdFrame ()
Prepare to render an HMD frame. This must be called every frame before flipping or setting the view buffer.
This function will wait until the HMD is ready to begin rendering before continuing. The current frame texture from the swap chain are pulled from the SDK and made available for binding.

_startOfFlip ()
Custom _startOfFlip for HMD rendering. This finalizes the HMD texture before diverting drawing operations back to the on-screen window. This allows flip to swap the on-screen and HMD buffers when called. This function always returns True.

Returns
Return type True

_updatePerfStats ()
Update and process performance statistics obtained from LibOVR. This should be called at the beginning of each frame to get the stats of the last frame.
This is called automatically when _waitToBeginHmdFrame () is called at the beginning of each frame.

_updateProjectionMatrix ()
Update or re-calculate projection matrices based on the current render descriptor configuration.

_waitToBeginHmdFrame ()
Wait until the HMD surfaces are available for rendering.

addEditable (editable)
Adds an editable element to the screen (something to which characters can be sent with meaning from the keyboard).
The current editable object receiving chars is Window.currentEditable

Parameters editable –

Returns

property ambientLight
Ambient light color for the scene [r, g, b, a]. Values range from 0.0 to 1.0. Only applicable if useLights is True.

Examples
Setting the ambient light color:

```python
win.ambientLight = [0.5, 0.5, 0.5]
# don't do this!!!
win.ambientLight[0] = 0.5
win.ambientLight[1] = 0.5
win.ambientLight[2] = 0.5
```

applyEyeTransform (clearDepth=True)
Apply the current view and projection matrices.
Matrices specified by attributes viewMatrix and projectionMatrix are applied using ‘immediate mode’ OpenGL functions. Subsequent drawing operations will be affected until flip() is called.
All transformations in GL_PROJECTION and GL_MODELVIEW matrix stacks will be cleared (set to identity) prior to applying.

**Parameters**

- **clearDepth (bool)** – Clear the depth buffer. This may be required prior to rendering 3D objects.

**Examples**

Using a custom view and projection matrix:

```python
# Must be called every frame since these values are reset after
# `flip()` is called!
win.viewMatrix = viewtools.lookAt( ... )
win.projectionMatrix = viewtools.perspectiveProjectionMatrix( ... )
win.applyEyeTransform()
# draw 3D objects here ...
```

The source data specified to `headPose` can originate from the tracking state retrieved by calling `getTrackingState`, or from other sources. If a custom head pose is specified (for instance, from a motion tracker), you must ensure `head-locking` is enabled to prevent the ASW feature of the compositor from engaging. Furthermore, you must specify sensor sample time for motion-to-photon calculation derived from the sample time of the custom tracking source.

**Parameters**

- **headPose (LibOVRPose)** – Head pose to use.
- **originPose (LibOVRPose, optional)** – Origin of tracking in the VR scene.

**Examples**

Get the tracking state and calculate the eye poses:

```python
# get tracking state at predicted mid-frame time
trackingState = hmd.getTrackingState()

# get the head pose from the tracking state
headPose = trackingState.headPose.thePose
hmd.calcEyePoses(headPose)  # compute eye poses

# begin rendering to each eye
for eye in ('left', 'right'):
```

(continues on next page)
Using a custom head pose (make sure headLocked=True before doing this):

```python
headPose = createPose((0., 1.75, 0.))
hmd.calcEyePoses(headPose)  # compute eye poses
```

**callOnFlip** *(function, *args, **kwargs)*

Call a function immediately after the next `flip()` command.

The first argument should be the function to call, the following args should be used exactly as you would for your normal call to the function (can use ordered arguments or keyword arguments as normal).

e.g. If you have a function that you would normally call like this:

```python
pingMyDevice(portToPing, channel=2, level=0)
```

then you could call `callOnFlip()` to have the function call synchronized with the frame flip like this:

```python
win.callOnFlip(pingMyDevice, portToPing, channel=2, level=0)
```

**clearBuffer** *(color=True, depth=False, stencil=False)*

Clear the present buffer (to which you are currently drawing) without flipping the window.

Useful if you want to generate movie sequences from the back buffer without actually taking the time to flip the window.

Set color prior to clearing to set the color to clear the color buffer to. By default, the depth buffer is cleared to a value of 1.0.

**Parameters**

- **color** *(bool)* – Buffers to clear.
- **depth** *(bool)* – Buffers to clear.
- **stencil** *(bool)* – Buffers to clear.

**Examples**

Clear the color buffer to a specified color:

```python
win.color = (1, 0, 0)
win.clearBuffer(color=True)
```

Clear only the depth buffer, `depthMask` must be `True` or else this will have no effect. Depth mask is usually `True` by default, but may change:

```python
win.depthMask = True
win.clearBuffer(color=False, depth=True, stencil=False)
```

**clearShouldRecenterFlag** *

Clear the ‘shouldRecenter’ status flag at the API level.

**close** *

Close the window and cleanly shutdown the LibOVR session.
**color**

Set the color of the window.

This command sets the color that the blank screen will have on the next clear operation. As a result, it effectively takes TWO flip() operations to become visible (the first uses the color to create the new screen, the second presents that screen to the viewer). For this reason, if you want to change the background color of the window “on the fly”, it might be a better idea to draw a Rect that fills the whole window with the desired Rect.fillColor attribute. That’ll show up on first flip.

See other stimuli (e.g. GratingStim.color) for more info on the color attribute which essentially works the same on all PsychoPy stimuli.

See **Color spaces** for further information about the ways to specify colors and their various implications.

**colorSpace**

Documentation for colorSpace is in the stimuli.

e.g. GratingStim.colorSpace

Usually used in conjunction with color like this:

```python
win.colorSpace = 'rgb255'  # changes colorSpace but not the value of win.color
win.color = [0, 0, 255]    # clear blue in rgb255
```

See **Color spaces** for further information about the ways to specify colors and their various implications.

**property connectedControllers**

Connected controller types (list of str)

**property convergeOffset**

Convergence offset from monitor in centimeters.

This is value corresponds to the offset from screen plane to set the convergence plane (or point for toe-in projections). Positive offsets move the plane farther away from the viewer, while negative offsets nearer. This value is used by setPerspectiveView and should be set before calling it to take effect.

**Notes**

- This value is only applicable for setToeIn and setOffAxisView.

**coordToRay** *(screenXY)*

Convert a screen coordinate to a direction vector.

Takes a screen/window coordinate and computes a vector which projects a ray from the viewpoint through it (line-of-sight). Any 3D point touching the ray will appear at the screen coordinate.

Uses the current viewport and projectionMatrix to calculate the vector. The vector is in eye-space, where the origin of the scene is centered at the viewpoint and the forward direction aligned with the -Z axis. A ray of (0, 0, -1) results from a point at the very center of the screen assuming symmetric frustums.

Note that if you are using a flipped/mirrored view, you must invert your supplied screen coordinates (screenXY) prior to passing them to this function.

**Parameters**

- **screenXY** *(array_like)* – X, Y screen coordinate. Must be in units of the window.

**Returns**

- Normalized direction vector [x, y, z].

**Return type**

ndarray
Examples

Getting the direction vector between the mouse cursor and the eye:

```python
mx, my = mouse.getPos()
dir = win.coordToRay((mx, my))
```

Set the position of a 3D stimulus object using the mouse, constrained to a plane. The object origin will always be at the screen coordinate of the mouse cursor:

```python
# the eye position in the scene is defined by a rigid body pose
win.viewMatrix = camera.getViewMatrix()
win.applyEyeTransform()

# get the mouse location and calculate the intercept
mx, my = mouse.getPos()
ray = win.coordToRay([mx, my])
result = intersectRayPlane(  # from mathtools
    orig= camera.pos,
    dir= camera.transformNormal(ray),
    planeOrig=(0, 0, -10),
    planeNormal=(0, 1, 0))

# if result is 'None', there is no intercept
if result is not None:
    pos, dist = result
    objModel.thePose.pos = pos
else:
    objModel.thePose.pos = (0, 0, -10)  # plane origin
```

If you don’t define the position of the viewer with a `RigidBodyPose`, you can obtain the appropriate eye position and rotate the ray by doing the following:

```python
pos = numpy.linalg.inv(win.viewMatrix)[:3, 3]
ray = win.coordToRay([mx, my]).dot(win.viewMatrix[:3, :3])

# then ...
result = intersectRayPlane(
    orig=pos,
    dir=ray,
    planeOrig=(0, 0, -10),
    planeNormal=(0, 1, 0))
```

```python
static createBoundingBox (extents=None)
Create a new bounding box object (LibOVRBounds).
LibOVRBounds represents an axis-aligned bounding box with dimensions defined by extents. Bounding boxes are primarily used for visibility testing and culling by PsychXR. The dimensions of the bounding box can be specified explicitly, or fitted to meshes by passing vertices to the fit() method after initialization.

This function exposes the LibOVRBounds class so you don’t need to access it by importing psychxr.

Parameters extents (array_like or None) – Extents of the bounding box as (mins, maxs). Where mins (x, y, z) is the minimum and maxs (x, y, z) is the maximum extents of the bounding box in world units. If None is specified, the returned bounding box will be invalid.

The bounding box can be later constructed using the fit() method or the extents attribute.

Returns Object representing a bounding box.

Return type ~psychxr.libovr.LibOVRBounds
```

8.2. psychopy.visual - many visual stimuli
Examples

Add a bounding box to a pose:

```py
# create a 1 meter cube bounding box centered with the pose
bbox = Rift.createBoundingBox(((-.5, -.5, -.5), (.5, .5, .5)))

# create a pose and attach the bounding box
modelPose = Rift.createPose()
modelPose.boundingBox = bbox
```

Perform visibility culling on the pose using the bounding box by using the isVisible() method:

```py
if hmd.isPoseVisible(modelPose):
    modelPose.draw()
```

**static createHapticsBuffer** *(samples)*

Create a new haptics buffer.

A haptics buffer is object which stores vibration amplitude samples for playback through the Touch controllers. To play a haptics buffer, pass it to `py:method:`submitHapticsBuffer`.

**Parameters**

samples *(array_like)* – 1-D array of amplitude samples, ranging from 0 to 1. Values outside of this range will be clipped. The buffer must not exceed HAPTICS_BUFFER_SAMPLES_MAX samples, any additional samples will be dropped.

**Returns**

Haptics buffer object.

**Return type**

LibOVRHapticsBuffer

**Notes**

Methods `startHaptics` and `stopHaptics` cannot be used interchangeably with this function.

**Examples**

Create a haptics buffer where vibration amplitude ramps down over the course of playback:

```py
samples = np.linspace(1.0, 0.0, num=HAPTICS_BUFFER_SAMPLES_MAX-1, dtype=np.float32)
hbuff = Rift.createHapticsBuffer(samples)

# vibrate right Touch controller
hmd.submitControllerVibration(CONTROLLER_TYPE_RTOUCH, hbuff)
```

**static createPose** *(pos=(0.0, 0.0, 0.0), ori=(0.0, 0.0, 0.0, 1.0))*

Create a new Rift pose object *(LibOVRPose)*.

**LibOVRPose** is used to represent a rigid body pose mainly for use with the PsychXR’s LibOVR module. There are several methods associated with the object to manipulate the pose.

This function exposes the **LibOVRPose** class so you don’t need to access it by importing *psychxr*.

**Parameters**

- pos *(tuple, list, or ndarray of float)* – Position vector/coordinate (x, y, z).
- ori (tuple, list, or ndarray of float) – Orientation quaternion (x, y, z, w).

Returns Object representing a rigid body pose for use with LibOVR.

Return type LibOVRPose

property cullFace
  True if face culling is enabled.

property cullFaceMode
  Face culling mode, either back, front or both.

property currentEditable
  The editable (Text?) object that currently has key focus

property depthFunc
  Depth test comparison function for rendering.

property depthMask
  True if depth masking is enabled. Writing to the depth buffer will be disabled.

property depthTest
  True if depth testing is enabled.

classmethod dispatchAllWindowEvents()
  Dispatches events for all pyglet windows. Used by iohub 2.0 psychopy kb event integration.

property displayRefreshRate
  Get the HMD’s display refresh rate in Hz (float).

property displayResolution
  Get the HMD’s raster display size (int, int).

property draw3d
  True if 3D drawing is enabled on this window.

property eyeHeight
  Eye height in meters (float).

property eyeOffset
  Eye separation in centimeters (float).

property eyeRenderPose
  Computed eye pose for the current buffer. Only valid after calling calcEyePoses().

property eyeToNoseDistance
  Eye to nose distance in meters (float).

Examples

Generate your own eye poses. These are used when :py:method:`calcEyePoses` is called:

```python
leftEyePose = Rift.createPose((-self.eyeToNoseDistance, 0., 0.))
rightEyePose = Rift.createPose((self.eyeToNoseDistance, 0., 0.))
```

Get the inter-axial separation (IAS) reported by LibOVR:

```python
iad = self.eyeToNoseDistance * 2.0
```

property farClip
  Distance to the far clipping plane in meters.
property firmwareVersion
    Get the firmware version of the active HMD (int, int).

flip (clearBuffer=True, drawMirror=True)
    Submit view buffer images to the HMD's compositor for display at next V-SYNC and draw the mirror texture
to the on-screen window. This must be called every frame.

    Parameters
    • clearBuffer (bool) – Clear the frame after flipping.
    • drawMirror (bool) – Draw the HMD mirror texture from the compositor to the window.

    Returns Absolute time in seconds when control was given back to the application. The difference
    between the current and previous values should be very close to 1 / refreshRate of the HMD.

    Return type float

Notes
    • The HMD compositor and application are asynchronous, therefore there is no guarantee that the times-
tamp returned by 'flip' corresponds to the exact vertical retrace time of the HMD.

fps ()
    Report the frames per second since the last call to this function (or since the window was created if this is
    first call)

property frameBufferSize
    Size of the framebuffer in pixels (w, h).

property frontFace
    Face winding order to define front, either ccw or cw.

fullscr
    Set whether fullscreen mode is True or False (not all backends can toggle an open window).

gamma
    Set the monitor gamma for linearization.

    Warning: Don’t use this if using a Bits++ or Bits#, as it overrides monitor settings.

gammaRamp
    Sets the hardware CLUT using a specified 3xN array of floats ranging between 0.0 and 1.0.
    Array must have a number of rows equal to 2^ max(bpc).

getActualFrameRate (nIdentical=10, nMaxFrames=100, nWarmUpFrames=10, threshold=1)
    Measures the actual frames-per-second (FPS) for the screen.
    This is done by waiting (for a max of nMaxFrames) until nIdentical frames in a row have identical frame times
    (std dev below threshold ms).

    Parameters
    • nIdentical (int, optional) – The number of consecutive frames that will be evaluated. Higher -> greater precision. Lower -> faster.
    • nMaxFrames (int, optional) – The maximum number of frames to wait for a matching set of nIdentical.
- **nWarmUpFrames** (*int, optional*) – The number of frames to display before starting the test (this is in place to allow the system to settle after opening the Window for the first time.

- **threshold** (*int, optional*) – The threshold for the std deviation (in ms) before the set are considered a match.

Returns Frame rate (FPS) in seconds. If there is no such sequence of identical frames a warning is logged and *None* will be returned.

Return type  *float* or *None*

**getBoundaryDimensions** (*boundaryType='PlayArea'*)

Get boundary dimensions.

Parameters  **boundaryType** (*str*) – Boundary type, can be 'PlayArea' or 'Outer'.

Returns  Dimensions of the boundary meters [x, y, z].

Return type  *ndarray*

**getButtons** (*buttons, controller='Xbox', testState='continuous'*)

Get button states from a controller.

Returns  *True* if any names specified to *buttons* reflect *testState* since the last *updateInputState* or *flip* call. If multiple button names are specified as a *list* or *tuple* to *buttons*, multiple button states are tested, returning *True* if all the buttons presently satisfy the *testState*. Note that not all controllers available share the same buttons. If a button is not available, this function will always return *False*.

Parameters

- **buttons** (*list of str or str*) – Buttons to test. Valid *buttons* names are 'A', 'B', 'RThumb', 'RShoulder' 'X', 'Y', 'LThumb', 'LShoulder', 'Up', 'Down', 'Left', 'Right', 'Enter', 'Back', 'VolUp', 'VolDown', and 'Home'. Names can be passed as a *list* to test multiple button states.

- **controller** (*str*) – Controller name.

- **testState** (*str*) – State to test. Valid values are:

  - **continuous** - Button is presently being held down.

  - **rising** or **pressed** - Button has been *pressed* since the last update.

  - **falling** or **released** - Button has been *released* since the last update.

Returns  Button state and timestamp in seconds the controller was polled.

Return type  *tuple of bool, float*

**Examples**

Check if the ‘Enter’ button on the Oculus remote was released:

```python
isPressed, tsec = hmd.getButtons(['Enter', 'Remote', 'falling'])
```

Check if the ‘A’ button was pressed on the touch controller:

```python
isPressed, tsec = hmd.getButtons(['A', 'Touch', 'pressed'])
```

**getDevicePose** (*deviceName*, *absTime=None*, *latencyMarker=False*)

Get the pose of a tracked device. For head (HMD) and hand poses (Touch controllers) it is better to use :py:method:`getTrackingState` instead.
Parameters


- **absTime** *(float, optional)* – Absolute time in seconds the device pose refers to. If not specified, the predicted time is used.

- **latencyMarker** *(bool)* – Insert a marker for motion-to-photon latency calculation. Should only be True if the HMD pose is being used to compute eye poses.

Returns Pose state object. None if device tracking was lost.

Return type *LibOVRPoseState or None*

g**etFutureFlipTime**(targetTime=0, clock=None)

The expected time of the next screen refresh. This is currently calculated as *win._lastFrameTime + refresh-Interval*

Parameters

- **targetTime** *(float)* – The delay from now for which you want the flip time. 0 will give the because that the earliest we can achieve. 0.15 will give the schedule flip time that gets as close to 150 ms as possible

- **clock** *(None, 'ptb', 'now' or any Clock object)* – If True then the time returned is compatible with ptb.GetSecs()

- **verbose** *(bool)* – Set to True to view the calculations along the way

g**etHandTriggerValues**(controller='Touch', deadzone=False)

Get the values of the hand triggers.

Parameters

- **controller** *(str)* – Name of the controller to get hand trigger values. Possible values for *controller* are ‘Touch’, ‘RTouch’, ‘LTouch’, ‘Object0’, ‘Object1’, ‘Object2’, and ‘Object3’; the only devices with hand triggers the SDK manages. For additional controllers, use PsychPy’s built-in event or hardware support.

- **deadzone** *(bool)* – Apply the deadzone to hand trigger values. This pre-filters stick input to apply a dead-zone where 0.0 will be returned if the trigger returns a displacement within 0.2746.

Returns Left and right hand trigger values. Displacements are represented as *tuple* of two float representing the left and right displacement values, which range from 0.0 to 1.0. The returned values reflect the controller state since the last *updateInputState* or *flip* call.

Return type *tuple*

g**getIndexTriggerValues**(controller='Xbox', deadzone=False)

Get the values of the index triggers.

Parameters

- **controller** *(str)* – Name of the controller to get index trigger values. Possible values for *controller* are ‘Xbox’, ‘Touch’, ‘RTouch’, ‘LTouch’, ‘Object0’, ‘Object1’, ‘Object2’, and ‘Object3’; the only devices with index triggers the SDK manages. For additional controllers, use PsychPy’s built-in event or hardware support.

- **deadzone** *(bool)* – Apply the deadzone to index trigger values. This pre-filters stick input to apply a dead-zone where 0.0 will be returned if the trigger returns a displacement within 0.2746.
Returns Left and right index trigger values. Displacements are represented as tuple of two float representing the left and right displacement values, which range from 0.0 to 1.0. The returned values reflect the controller state since the last `updateInputState` or `flip` call.

Return type tuple of float

getMovieFrame `buffer='mirror'`  
Capture the current HMD frame as an image.

Saves to stack for `saveMovieFrames()` As of v1.81.00 this also returns the frame as a PIL image.

This can be done at any time (usually after a `flip()` command).

Frames are stored in memory until a `saveMovieFrames()` command is issued. You can issue `getMovieFrame()` as often as you like and then save them all in one go when finished.

For HMD frames, you should call `getMovieFrame` after calling `flip()` to ensure that the mirror texture saved reflects what is presently being shown on the HMD. Note, that this function is somewhat slow and may impact performance. Only call this function when you’re not collecting experimental data.

Parameters `buffer (str, optional)` – Buffer to capture. For the HMD, only ‘mirror’ is available at this time.

Returns Buffer pixel contents as a PIL/Pillow image object.

Return type Image

getMsPerFrame `nFrames=60, showVisual=False, msg='', msDelay=0.0`  
Assesses the monitor refresh rate (average, median, SD) under current conditions, over at least 60 frames.

Records time for each refresh (frame) for n frames (at least 60), while displaying an optional visual. The visual is just eye-candy to show that something is happening when assessing many frames. You can also give it text to display instead of a visual, e.g., `msg='(testing refresh rate...) ' `; setting `msg` implies `showVisual = False`.

To simulate refresh rate under cpu load, you can specify a time to wait within the loop prior to doing the `flip()`. If `0 < msDelay < 100`, wait for that long in ms.

Returns timing stats (in ms) of:

- average time per frame, for all frames
- standard deviation of all frames
- median, as the average of 12 frame times around the median (~monitor refresh rate)

Author

- 2010 written by Jeremy Gray

gPredictedDisplayTime `()`  
Get the predicted time the next frame will be displayed on the HMD. The returned time is referenced to the clock `LibOVR` is using.

Returns Absolute frame mid-point time for the given frame index in seconds.

Return type float

gThumblestickValues `controller='Xbox', deadzone=False`  
Get controller thumbstick values.

Parameters
• **controller** *(str)* – Name of the controller to get thumbstick values. Possible values for `controller` are ‘Xbox’, ‘Touch’, ‘RTouch’, ‘LTouch’, ‘Object0’, ‘Object1’, ‘Object2’, and ‘Object3’; the only devices with thumbsticks the SDK manages. For additional controllers, use PsychPy’s built-in event or hardware support.

• **deadzone** *(bool)* – Apply the deadzone to thumbstick values. This pre-filters stick input to apply a dead-zone where 0.0 will be returned if the sticks return a displacement within -0.2746 to 0.2746.

**Returns** Left and right, X and Y thumbstick values. Axis displacements are represented in each tuple by floats ranging from -1.0 (full left/down) to 1.0 (full right/up). The returned values reflect the controller state since the last `updateInputState` or `flip` call.

**Return type** tuple

**getTimeInSeconds** ()

Absolute time in seconds. The returned time is referenced to the clock _LibOVR_ is using.

**Returns** Time in seconds.

**Return type** float

**getTouches** *(touches, controller='Touch', testState='continuous')*

Get touch states from a controller.

Returns `True` if any names specified to `touches` reflect `testState` since the last `updateInputState` or `flip` call. If multiple button names are specified as a *list* or *tuple* to `touches`, multiple button states are tested, returning `True` if all the touches presently satisfy the `testState`. Note that not all controllers available support touches. If a touch is not supported or available, this function will always return `False`.

Special states can be used for basic gesture recognition, such as ‘LThumbUp’, ‘RThumbUp’, ‘LIndexPointing’, and ‘RIndexPointing’.

**Parameters**


• **controller** *(str)* – Controller name.

• **testState** *(str)* – State to test. Valid values are:
  - **continuous** - User is touching something on the controller.
  - **rising** or **pressed** - User began touching something since the last call to `updateInputState`.
  - **falling** or **released** - User stopped touching something since the last call to `updateInputState`.

**Returns** Touch state and timestamp in seconds the controller was polled.

**Return type** tuple of bool, float
Examples

Check if the ‘Enter’ button on the Oculus remote was released:

```python
isPressed, tsec = hmd.getButtons(['Enter'], 'Remote', 'falling')
```

Check if the ‘A’ button was pressed on the touch controller:

```python
isPressed, tsec = hmd.getButtons(['A'], 'Touch', 'pressed')
```

getTrackerInfo(trackerIdx)
Get tracker information.

- **Parameters**
  - `trackerIdx` (*int*) – Tracker index, ranging from 0 to `trackerCount`.

- **Returns**
  - Object containing tracker information.

- **Return type**
  - `LibOVRTrackerInfo`

- **Raises**
  - `IndexError` – Raised when `trackerIdx` out of range.

getTrackingState(absTime=None, latencyMarker=True)
Get the tracking state of the head and hands.

Calling this function retrieves the tracking state of the head (HMD) and hands at `absTime` from the `LibOVR` runtime. The returned object is a `LibOVRTrackingState` instance with poses, motion derivatives (i.e. linear and angular velocity/acceleration), and tracking status flags accessible through its attributes.

The pose states of the head and hands are available by accessing the `headPose` and `handPoses` attributes, respectively.

- **Parameters**
  - `absTime` (*float*, optional) – Absolute time the the tracking state refers to. If not specified, the predicted display time is used.
  - `latencyMarker` (*bool*, optional) – Set a latency marker upon getting the tracking state. This is used for motion-to-photon calculations.

- **Returns**
  - Tracking state object. For more information about this type see:

- **Return type**
  - `LibOVRTrackingState`

See also:

- **getPredictedDisplayTime** Time at mid-frame for the current frame index.

Examples

Get the tracked head pose and use it to calculate render eye poses:

```python
# get tracking state at predicted mid-frame time
absTime = getPredictedDisplayTime()
trackingState = hmd.getTrackingState(absTime)

# get the head pose from the tracking state
headPose = trackingState.headPose.thePose
hmd.calcEyePoses(headPose)  # compute eye poses
```

Get linear/angular velocity and acceleration vectors of the right touch controller:
# right hand is the second value (index 1) at `handPoses`
rightHandState = trackingState.handPoses[1]  # is `LibOVRPoseState`

# access `LibOVRPoseState` fields to get the data
linearVel = rightHandState.linearVelocity # m/s
angularVel = rightHandState.angularVelocity # rad/s
linearAcc = rightHandState.linearAcceleration # m/s^2
angularAcc = rightHandState.angularAcceleration # rad/s^2

# extract components like this if desired
vx, vy, vz = linearVel
ax, ay, az = angularVel

Above is useful for physics simulations, where one can compute the magnitude and direction of a force applied to a virtual object.

It's often the case that object tracking becomes unreliable for some reason, for instance, if it becomes occluded and is no longer visible to the sensors. In such cases, the reported pose state is invalid and may not be useful. You can check if the position and orientation of a tracked object is invalid using flags associated with the tracking state. This shows how to check if head position and orientation tracking was valid when sampled:

```python
if trackingState.positionValid and trackingState.orientationValid:
    print('Tracking valid.')
```

It's up to the programmer to determine what to do in such cases. Note that tracking may still be valid even if

Get the calibrated origin used for tracking during the sample period of the tracking state:

```python
calibratedOrigin = trackingState.calibratedOrigin
calibPos, calibOri = calibratedOrigin.posOri
```

Time integrate a tracking state. This extrapolates the pose over time given the present computed motion derivatives. The contrived example below shows how to implement head pose forward prediction:

```python
# get current system time
absTime = getTimeInSeconds()

# get the elapsed time from `absTime` to predicted v-sync time,
# again this is an example, you would usually pass predicted time to
# `getTrackingState` directly.
dt = getPredictedDisplayTime() - absTime

# get the tracking state for the current time, poses will lag where
# they are expected at predicted time by `dt` seconds
trackingState = hmd.getTrackingState(absTime)

# time integrate a pose by `dt`
headPoseState = trackingState.headPose
headPosePredicted = headPoseState.timeIntegrate(dt)

# calc eye poses with predicted head pose, this is a custom pose to
# head-locking should be enabled!
hmd.calcEyePoses(headPosePredicted)
```

The resulting head pose is usually very close to what `getTrackingState` would return if the predicted time was used. Simple forward prediction with time integration becomes increasingly unstable as the prediction interval increases. Under normal circumstances, let the runtime handle forward prediction by using the pose states returned at the predicted display time. If you plan on doing your own forward prediction, you need enable
head-locking, clamp the prediction interval, and apply some sort of smoothing to keep the image as stable as possible.

**property hasInputFocus**

*True* if the application currently has input focus.

**property hasMagYawCorrection**

*True* if this HMD supports yaw correction.

**property hasOrientationTracking**

*True* if the HMD is capable of tracking orientation.

**property hasPositionTracking**

*True* if the HMD is capable of tracking position.

**property headLocked**

*True* if head locking is enabled.

**property hid**

USB human interface device (HID) identifiers *(int, int)*.

**hidePerfHud()**

Hide the performance HUD.

**property hmdMounted**

*True* if the HMD is mounted on the user’s head.

**property hmdPresent**

*True* if the HMD is present.

**property isBoundaryVisible**

*True* if the VR boundary is visible.

**isPoseVisible (pose)**

Check if a pose object if visible to the present eye. This method can be used to perform visibility culling to avoid executing draw commands for objects that fall outside the FOV for the current eye buffer.

If `boundingBox` has a valid bounding box object, this function will return *False* if all the box points fall completely to one side of the view frustum. If `boundingBox` is *None*, the point at `pos` is checked, returning *False* if it falls outside of the frustum. If the present buffer is not 'left' or 'right', this function will always return *False*.

**Parameters**

- **pose** *(LibOVRPose)* – Pose to test for visibility.

**Returns**

- **True** if pose’s bounding box or origin is outside of the view frustum.

**Return type**

- **bool**

**property isVisible**

*True* if the app has focus in the HMD and is visible to the viewer.

**property lights**

Scene lights.

This is specified as an array of `~psychopy.visual.LightSource` objects. If a single value is given, it will be converted to a list before setting. Set `useLights` to *True* before rendering to enable lighting/shading on subsequent objects. If `lights` is *None* or an empty list, no lights will be enabled if `useLights=True`, however, the scene ambient light set with `ambientLight` will be still be used.
Examples

Create a directional light source and add it to scene lights:

```python
dirLight = gltools.LightSource((0., 1., 0.), lightType='directional')
win.lights = dirLight  # `win.lights` will be a list when accessed!
```

Multiple lights can be specified by passing values as a list:

```python
myLights = [gltools.LightSource((0., 5., 0.)),
            gltools.LightSource((-2., -2., 0.))]
win.lights = myLights
```

`logOnFlip(msg, level, obj=None)`
Send a log message that should be time-stamped at the next `flip()` command.

Parameters

- `msg (str)` – The message to be logged.
- `level (int)` – The level of importance for the message.
- `obj (object, optional)` – The python object that might be associated with this message if desired.

`property manufacturer`  
Get the connected HMD’s manufacturer (str).

`mouseVisible`  
Sets the visibility of the mouse cursor.

If `Window` was initialized with `allowGUI=False` then the mouse is initially set to invisible, otherwise it will initially be visible.

Usage:

```python
win.mouseVisible = False
win.mouseVisible = True
```

`multiFlip(flips=1, clearBuffer=True)`  
Flip multiple times while maintaining the display constant. Use this method for precise timing.

WARNING: This function should not be used. See the *Notes* section for details.

Parameters

- `flips (int, optional)` – The number of monitor frames to flip. Floats will be rounded to integers, and a warning will be emitted. `Window.multiFlip(flips=1)` is equivalent to `Window.flip()`. Defaults to 1.
- `clearBuffer (bool, optional)` – Whether to clear the screen after the last flip. Defaults to `True`.  

220 Chapter 8. Reference Manual (API)
Notes

- This function can behave unpredictably, and the PsychoPy authors recommend against using it. See https://github.com/psychopy/psychopy/issues/867 for more information.

Examples

Example of using multiFlip:

```python
# Draws myStim1 to buffer
myStim1.draw()
# Show stimulus for 4 frames (90 ms at 60Hz)
myWin.multiFlip(clearBuffer=False, flips=6)
# Draw myStim2 "on top of" myStim1
# (because buffer was not cleared above)
myStim2.draw()
# Show this for 2 frames (30 ms at 60Hz)
myWin.multiFlip(flips=2)
# Show blank screen for 3 frames (buffer was cleared above)
myWin.multiFlip(flips=3)
```

multiplyProjectionMatrixGL()  
Multiply the current projection modelMatrix obtained from the SDK using glMultMatrixf. The projection matrix used depends on the current eye buffer set by setBuffer().

multiplyViewMatrixGL()  
Multiply the local eye pose transformation modelMatrix obtained from the SDK using glMultMatrixf. The modelMatrix used depends on the current eye buffer set by setBuffer().

Returns

Return type None

property nearClip  
Distance to the near clipping plane in meters.

nextEditable(chars='')  
Moves focus of the cursor to the next editable window

onResize(width, height)  
A default resize event handler.

This default handler updates the GL viewport to cover the entire window and sets the GL_PROJECTION matrix to be orthogonal in window space. The bottom-left corner is (0, 0) and the top-right corner is the width and height of the Window in pixels.

Override this event handler with your own to create another projection, for example in perspective.

property overlayPresent  

perfHudMode(mode='Off')  
Set the performance HUD mode.

Parameters mode (str) – HUD mode to use.

property pixelsPerTanAngleAtCenter  
Horizontal and vertical pixels per tangent angle (=1) at the center of the display.

This can be used to compute pixels-per-degree for the display.
property productName
Get the HMD's product name (str).

property projectionMatrix
Get the projection matrix for the current eye buffer. Note that setting projectionMatrix manually will break visibility culling.

recenterTrackingOrigin()
Recenter the tracking origin using the current head position.

recordFrameIntervals
Record time elapsed per frame.

Provides accurate measures of frame intervals to determine whether frames are being dropped. The intervals are the times between calls to flip(). Set to True only during the time-critical parts of the script. Set this to False while the screen is not being updated, i.e., during any slow, non-frame-time-critical sections of your code, including inter-trial-intervals, event.waitkeys(), core.wait(), or image.setImage().

Examples
Enable frame interval recording, successive frame intervals will be stored:

```python
win.recordFrameIntervals = True
```

Frame intervals can be saved by calling the saveFrameIntervals method:

```python
win.saveFrameIntervals()
```

removeEditable(editable)

resetEyeTransform(clearDepth=True)
Restore the default projection and view settings to PsychoPy defaults. Call this prior to drawing 2D stimuli objects (i.e. GratingStim, ImageStim, Rect, etc.) if any eye transformations were applied for the stimuli to be drawn correctly.

Parameters clearDepth (bool) – Clear the depth buffer upon reset. This ensures successive draw commands are not affected by previous data written to the depth buffer. Default is True.

Notes

- Calling flip() automatically resets the view and projection to defaults. So you don’t need to call this unless you are mixing 3D and 2D stimuli.

Examples
Going between 3D and 2D stimuli:

```python
# 2D stimuli can be drawn before setting a perspective projection
win.setPerspectiveView()
# draw 3D stimuli here ...
win.resetEyeTransform()
# 2D stimuli can be drawn here again ...
win.flip()
```
**resetViewport()**

Reset the viewport to cover the whole framebuffer.

Set the viewport to match the dimensions of the back buffer or framebuffer (if `useFBO=True`). The scissor rectangle is also set to match the dimensions of the viewport.

**saveFrameIntervals(fileName=None, clear=True)**

Save recorded screen frame intervals to disk, as comma-separated values.

**Parameters**

- `fileName` (None or str) – None or the filename (including path if necessary) in which to store the data. If None then ‘lastFrameIntervals.log’ will be used.
- `clear` (bool) – Clear buffer frames intervals were stored after saving. Default is `True`.

**saveMovieFrames(fileName, codec='libx264', fps=30, clearFrames=True)**

Writes any captured frames to disk.

Will write any format that is understood by PIL (tif, jpg, png, …)

**Parameters**

- `filename` (str) – Name of file, including path. The extension at the end of the file determines the type of file(s) created. If an image type (e.g. .png) is given, then multiple static frames are created. If it is .gif then an animated GIF image is created (although you will get higher quality GIF by saving PNG files and then combining them in dedicated image manipulation software, such as GIMP). On Windows and Linux .mpeg files can be created if `pymedia` is installed. On macOS .mov files can be created if the pyobjc-frameworks-QTKit is installed. Unfortunately the libs used for movie generation can be flaky and poor quality. As for animated GIFs, better results can be achieved by saving as individual .png frames and then combining them into a movie using software like ffmpeg.
- `codec` (str, optional) – The codec to be used by moviepy for mp4/mpeg/mov files. If `None` then the default will depend on file extension. Can be one of `libx264`, `mpeg4` for mp4/mov files. Can be `rawvideo`, `png` for avi files (not recommended). Can be `libvorbis` for ogv files. Default is `libx264`.
- `fps` (int, optional) – The frame rate to be used throughout the movie. Only for quicktime (.mov) movies. Default is 30.
- `clearFrames` (bool, optional) – Set this to `False` if you want the frames to be kept for additional calls to `saveMovieFrames`. Default is `True`.

**Examples**

Writes a series of static frames as frame001.tif, frame002.tif etc.:

```python
myWin.saveMovieFrames('frame.tif')
```

As of PsychoPy 1.84.1 the following are written with moviepy:

```python
myWin.saveMovieFrames('stimuli.mp4')  # codec = 'libx264' or 'mpeg4'
myWin.saveMovieFrames('stimuli.mov')
myWin.saveMovieFrames('stimuli.gif')
```

**property scissor**

Scissor rectangle (x, y, w, h) for the current draw buffer.
Values $x$ and $y$ define the origin, and $w$ and $h$ the size of the rectangle in pixels. The scissor operation is only active if `scissorTest=True`.

Usually, the scissor and viewport are set to the same rectangle to prevent drawing operations from *spilling* into other regions of the screen. For instance, calling `clearBuffer` will only clear within the scissor rectangle.

Setting the scissor rectangle but not the viewport will restrict drawing within the defined region (like a rectangular aperture), not changing the positions of stimuli.

**property scissorTest**

*True* if scissor testing is enabled.

**property sensorSampleTime**

Sensor sample time (*float*). This value corresponds to the time the head (HMD) position was sampled, which is required for computing motion-to-photon latency. This does not need to be specified if `getTrackingState` was called with `latencyMarker=True`.

**property serialNumber**

Get the connected HMD’s unique serial number (*str*).

Use this to identify a particular unit if you own many.

**setBlendMode** (*blendMode*, *log=None*)

Usually you can use `stim.attribute = value` syntax instead, but use this method if you need to suppress the log message.

**setBuffer** (*buffer*, *clear=True*)

Set the active draw buffer.

> Warning: The window.Window.size property will return the buffer’s dimensions in pixels instead of the window’s when `setBuffer` is set to ‘left’ or ‘right’.

**Parameters**

- **buffer** (*str*) – View buffer to divert successive drawing operations to, can be either ‘left’ or ‘right’.
- **clear** (*boolean*) – Clear the color, stencil and depth buffer.

**setColor** (*color*, *colorSpace=None*, *operation='', log=None*)

Usually you can use `stim.attribute = value` syntax instead, but use this method if you want to set color and colorSpace simultaneously.

See `color` for documentation on colors.

**setDefaultView** (*clearDepth=True*)

Return to default projection. Call this before drawing PsychoPy’s 2D stimuli after a stereo projection change.

Note: This only has an effect if using Rift in legacy immediate mode OpenGL.

**Parameters** **clearDepth** (*bool*) – Clear the depth buffer prior after configuring the view parameters.

**setGamma** (*gamma*, *log=None*)

Usually you can use `stim.attribute = value` syntax instead, but use this method if you need to suppress the log message.

**setMouseType** (*name='arrow'*)

Change the appearance of the cursor for this window. Cursor types provide contextual hints about how to interact with on-screen objects.
The graphics used ‘standard cursors’ provided by the operating system. They may vary in appearance and hot spot location across platforms. The following names are valid on most platforms:

- **arrow**: Default pointer.
- **ibeam**: Indicates text can be edited.
- **crosshair**: Crosshair with hot-spot at center.
- **hand**: A pointing hand.
- **hresize**: Double arrows pointing horizontally.
- **vresize**: Double arrows pointing vertically.

**Parameters**

- **name** (str) – Type of standard cursor to use (see above). Default is arrow.

**Notes**

- On Windows the **crosshair** option is negated with the background color. It will not be visible when placed over 50% grey fields.

**setMouseVisible** (visibility, log=None)

Usually you can use ’stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message.

**setOffAxisView** (applyTransform=True, clearDepth=True)

Set an off-axis projection.

Create an off-axis projection for subsequent rendering calls. Sets the viewMatrix and projectionMatrix accordingly so the scene origin is on the screen plane. If eyeOffset is correct and the view distance and screen size is defined in the monitor configuration, the resulting view will approximate ortho-stereo viewing.

The convergence plane can be adjusted by setting convergeOffset. By default, the convergence plane is set to the screen plane. Any points on the screen plane will have zero disparity.

**Parameters**

- **applyTransform** (bool) – Apply transformations after computing them in immediate mode. Same as calling applyEyeTransform() afterwards.
- **clearDepth** (bool, optional) – Clear the depth buffer.

**setPerspectiveView** (applyTransform=True, clearDepth=True)

Set the projection and view matrix to render with perspective.

Matrices are computed using values specified in the monitor configuration with the scene origin on the screen plane. Calculations assume units are in meters. If eyeOffset != 0, the view will be transformed laterally, however the frustum shape will remain the same.

Note that the values of projectionMatrix and viewMatrix will be replaced when calling this function.

**Parameters**

- **applyTransform** (bool) – Apply transformations after computing them in immediate mode. Same as calling applyEyeTransform() afterwards if False.
- **clearDepth** (bool, optional) – Clear the depth buffer.

**setRGB** (newRGB)

Deprecated: As of v1.61.00 please use setColor() instead
**setRecordFrameIntervals** *(value=True, log=None)*

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

**setRiftView** *(clearDepth=True)*

Set head-mounted display view. Gets the projection and view matrices from the HMD and applies them.

Note: This only has an effect if using Rift in legacy immediate mode OpenGL.

**Parameters**

- **clearDepth** *(bool)* – Clear the depth buffer prior after configuring the view parameters.

**setScale** *(units, font='dummyFont', prevScale=(1.0, 1.0))*

DEPRECATED: this method used to be used to switch between units for stimulus drawing but this is now handled by the stimuli themselves and the window should always be left in units of ‘pix’

**setSize** *(value, log=True)*

**setStereoDebugHudOption** *(option, value)*

Configure stereo debug HUD guides.

**Parameters**

- **option** *(str)* – Option to set. Valid options are *InfoEnable, Size, Position, YawPitchRoll,* and *Color.*

- **value** *(array_like or bool)* – Value to set for a given option. Appropriate types for each option are:
  - *Size* - array_like, [w, h] in meters.
  - *Position* - array_like, [x, y, z] in meters.
  - *Color* - array_like, [r, g, b] as floats ranging 0.0 to 1.0.

**Returns** *True* if the option was successfully set.

**Return type** *bool*

**Examples**

Configuring a stereo debug HUD guide:

```python
# show a quad with a crosshair
hmd.stereoDebugHudMode('QuadWithCrosshair')
# enable displaying guide information
hmd.setStereoDebugHudOption('InfoEnable', True)
# set the position of the guide quad in the scene
hmd.setStereoDebugHudOption('Position', [0.0, 1.7, -2.0])
```

**setToeInView** *(applyTransform=True, clearDepth=True)*

Set toe-in projection.

Create a toe-in projection for subsequent rendering calls. Sets the *viewMatrix* and *projectionMatrix* accordingly so the scene origin is on the screen plane. The value of *convergeOffset* will define the convergence point of the view, which is offset perpendicular to the center of the screen plane. Points falling on a vertical line at the convergence point will have zero disparity.

**Parameters**
• **applyTransform** (*bool*) – Apply transformations after computing them in immediate mode. Same as calling `applyEyeTransform()` afterwards.

• **clearDepth** (*bool, optional*) – Clear the depth buffer.

**Notes**

• This projection mode is only ‘correct’ if the viewer’s eyes are converged at the convergence point. Due to perspective, this projection introduces vertical disparities which increase in magnitude with eccentricity. Use `setOffAxisView` if you want to display something the viewer can look around the screen comfortably.

**setUnits** (*value, log=True*)

**setViewPos** (*value, log=True*)

**property shouldQuit**

*True* if the user requested the application should quit through the headset’s interface.

**property shouldRecenter**

*True* if the user requested the origin be re-centered through the headset’s interface.

**property size**

Size property to get the dimensions of the view buffer instead of the window. If there are no view buffers, always return the dims of the window.

**specifyTrackingOrigin** (*pose*)

Specify a tracking origin. If `trackingOriginType='floor'`, this function sets the origin of the scene in the ground plane. If `trackingOriginType='eye'`, the scene origin is set to the known eye height.

**Parameters**

- **pose** (*LibOVRPose*) – Tracking origin pose.

**specifyTrackingOriginPosOri** (*pos=(0.0, 0.0, 0.0), ori=(0.0, 0.0, 0.0, 1.0)*)

Specify a tracking origin using a pose and orientation. This is the same as `specifyTrackingOrigin`, but accepts a position vector [x, y, z] and orientation quaternion [x, y, z, w].

**Parameters**

- **pos** (*tuple or list of float, or ndarray*) – Position coordinate of origin (x, y, z).
- **ori** (*tuple or list of float, or ndarray*) – Quaternion specifying orientation (x, y, z, w).

**startHaptics** (*controller, frequency='low', amplitude=1.0*)

Start haptic feedback (vibration).

Vibration is constant at fixed frequency and amplitude. Vibration lasts 2.5 seconds, so this function needs to be called more often than that for sustained vibration. Only controllers which support vibration can be used here.

There are only two frequencies permitted ‘high’ and ‘low’, however, amplitude can vary from 0.0 to 1.0. Specifying ‘frequency’=’off’ stops vibration if in progress.

**Parameters**

- **controller** (*str*) – Name of the controller to vibrate.
- **frequency** (*str*) – Vibration frequency. Valid values are: ‘off’, ‘low’, or ‘high’.
- **amplitude** (*float*) – Vibration amplitude in the range of [0.0 and 1.0]. Values outside this range are clamped.
property stencilTest
    True if stencil testing is enabled.

stereoDebugHudMode (mode)
    Set the debug stereo HUD mode.
    This makes the compositor add stereoscopic reference guides to the scene. You can configure the HUD can be configured using other methods.

    Parameters mode (str) – Stereo debug mode to use. Valid options are Off, Quad, QuadWithCrosshair, and CrosshairAtInfinity.

Examples

Enable a stereo debugging guide:

```
hmd.stereoDebugHudMode('CrosshairAtInfinity')
```

Hide the debugging guide. Should be called before exiting the application since it’s persistent until the Oculus service is restarted:

```
hmd.stereoDebugHudMode('Off')
```

stopHaptics (controller)
    Stop haptic feedback.
    Convenience function to stop controller vibration initiated by the last vibrateController call. This is the same as calling vibrateController(controller, frequency='off').

    Parameters controller (str) – Name of the controller to stop vibrating.

submitControllerVibration (controller, hapticsBuffer)
    Submit a haptics buffer to begin controller vibration.

    Parameters

    • controller (str) – Name of controller to vibrate.
    • hapticsBuffer (LibOVRHapticsBuffer) – Haptics buffer to playback.

Notes

Methods startHaptics and stopHaptics cannot be used interchangeably with this function.

tanAngleToNDC (horzTan, vertTan)
    Convert tan angles to the normalized device coordinates for the current buffer.

    Parameters

    • horzTan (float) – Horizontal tan angle.
    • vertTan (float) – Vertical tan angle.

    Returns Normalized device coordinates X, Y. Coordinates range between -1.0 and 1.0. Returns None if an invalid buffer is selected.

    Return type tuple of float

testBoundary (deviceType, boundaryType='PlayArea')
    Test if tracked devices are colliding with the play area boundary.
    This returns an object containing test result data.
Parameters

- `deviceType (str, list or tuple)` – The device to check for boundary collision. If a list of names is provided, they will be combined and all tested.

- `boundaryType (str)` – Boundary type to test.

`timeOnFlip (obj, attrib)`
Retrieves the time on the next flip and assigns it to the `attrib` for this `obj`.

Parameters

- `obj (dict or object)` – A mutable object (usually a dict of class instance).

- `attrib (str)` – Key or attribute of `obj` to assign the flip time to.

Examples

Assign time on flip to the `tStartRefresh` key of `myTimingDict`:

```python
win.getTimeOnFlip(myTimingDict, 'tStartRefresh')
```

property `trackerCount`
Number of attached trackers.

property `trackingOriginType`
Current tracking origin type (`str`).

Valid tracking origin types are ‘floor’ and ‘eye’.

`units`
`None`, ‘height’ (of the window), ‘norm’, ‘deg’, ‘cm’, ‘pix’ Defines the default units of stimuli initialized in the window. I.e. if you change units, already initialized stimuli won’t change their units.

Can be overridden by each stimulus, if units is specified on initialization.

See `Units for the window and stimuli` for explanation of options.

`update ()`
Deprecated: use `Window.flip()` instead

`updateInputState (controllers=None)`
Update all connected controller states. This updates controller input states for an input device managed by `LibOVR`.

The polling time for each device is accessible through the `controllerPollTimes` attribute. This attribute returns a dictionary where the polling time from the last `updateInputState` call for a given controller can be retrieved by using the name as a key.

Parameters `controllers (tuple or list, optional)` – List of controllers to poll. If `None`, all available controllers will be polled.
Examples

Poll the state of specific controllers by name:

```python
controllers = ['XBox', 'Touch']
updateInputState(controllers)
```

`updateLights (index=None)`

Explicitly update scene lights if they were modified.

This is required if modifications to objects referenced in `lights` have been changed since assignment. If you removed or added items of `lights` you must refresh all of them.

Parameters `index` *(int, optional)* – Index of light source in `lights` to update. If `None`, all lights will be refreshed.

Examples

Call `updateLights` if you modified lights directly like this:

```python
win.lights[1].diffuseColor = [1., 0., 0.]
win.updateLights(1)
```

**property useLights**

Enable scene lighting.

Lights will be enabled if using legacy OpenGL lighting. Stimuli using shaders for lighting should check if `useLights` is `True` since this will have no effect on them, and disable or use a no lighting shader instead. Lights will be transformed to the current view matrix upon setting to `True`.

Lights are transformed by the present `GL_MODELVIEW` matrix. Setting `useLights` will result in their positions being transformed by it. If you want lights to appear at the specified positions in world space, make sure the current matrix defines the view/eye transformation when setting `useLights=True`.

This flag is reset to `False` at the beginning of each frame. Should be `False` if rendering 2D stimuli or else the colors will be incorrect.

**property userHeight**

Get user height in meters *(float)*.

**property viewMatrix**

The view matrix for the current eye buffer. Only valid after a `:py:method:`calcEyePoses` call. Note that setting `viewMatrix` manually will break visibility culling.

**viewPos**

The origin of the window onto which stimulus-objects are drawn.

The value should be given in the units defined for the window. NB: Never change a single component (x or y) of the origin, instead replace the `viewPos`-attribute in one shot, e.g.:

```python
win.viewPos = [new_xval, new_yval]  # This is the way to do it
win.viewPos[0] = new_xval  # DO NOT DO THIS! Errors will result.
```

**property viewport**

Viewport rectangle *(x, y, w, h)* for the current draw buffer.

Values x and y define the origin, and w and h the size of the rectangle in pixels.
This is typically set to cover the whole buffer, however it can be changed for applications like multi-view rendering. Stimuli will draw according to the new shape of the viewport, for instance and stimulus with position (0, 0) will be drawn at the center of the viewport, not the window.

Examples

Constrain drawing to the left and right halves of the screen, where stimuli will be drawn centered on the new rectangle. Note that you need to set both the viewport and the scissor rectangle:

```
x, y, w, h = win.frameBufferSize  # size of the framebuffer
win.viewport = win.scissor = [x, y, w / 2.0, h]
# draw left stimuli ...
win.viewport = win.scissor = [x + (w / 2.0), y, w / 2.0, h]
# draw right stimuli ...
# restore drawing to the whole screen
win.viewport = win.scissor = [x, y, w, h]
```

waitBlanking

After a call to flip() should we wait for the blank before the script continues.

8.2.26 RigidBodyPose

Attributes

```
RigidBodyPose([pos, ori]) Class for representing rigid body poses.
```

Details

```
class psychopy.visual.RigidBodyPose(pos=(0.0, 0.0, 0.0), ori=(0.0, 0.0, 0.0, 1.0))
Class for representing rigid body poses.

This class is an abstract representation of a rigid body pose, where the position of the body in a scene is represented by a vector/coordinate and the orientation with a quaternion. Pose can be manipulated and interacted with using class methods and attributes. Rigid body poses assume a right-handed coordinate system (-Z is forward and +Y is up).

Poses can be converted to 4x4 transformation matrices with getModelMatrix. One can use these matrices when rendering to transform the vertices of a model associated with the pose by passing them to OpenGL. Matrices are cached internally to avoid recomputing them if pos and ori attributes have not been updated.

Operators * and ~ can be used on RigidBodyPose objects to combine and invert poses. For instance, you can multiply (*) poses to get a new pose which is the combination of both orientations and translations by:

```
newPose = rb1 * rb2
```

Likewise, a pose can be inverted by using the ~ operator:

```
invPose = ~rb
```

Multiplying a pose by its inverse will result in an identity pose with no translation and default orientation where pos=[0, 0, 0] and ori=[0, 0, 0, 1]:

8.2. psychopy.visual - many visual stimuli
identityPose = -rb * rb

**Warning:** This class is experimental and may result in undefined behavior.

**Parameters**

- **pos** (`array_like`) – Position vector \([x, y, z]\) for the origin of the rigid body.
- **ori** (`array_like`) – Orientation quaternion \([x, y, z, w]\) where \(x, y, z\) are imaginary and \(w\) is real.

**alignTo** (`alignTo`)

Align this pose to another point or pose.

This sets the orientation of this pose to one which orients the forward axis towards `alignTo`.

**Parameters**

- **alignTo** (`array_like` or `LibOVRPose`) – Position vector \([x, y, z]\) or pose to align to.

**property at**

Vector defining the forward direction (-Z) of this pose.

**property bounds**

Bounding box associated with this pose.

**copy()**

Get a new `RigidBodyPose` object which copies the position and orientation of this one. Copies are independent and do not reference each others data.

**Returns**

Copy of this pose.

**Return type** `RigidBodyPose`

**distanceTo** (`v`)

Get the distance to a pose or point in scene units.

**Parameters**

- **v** (`RigidBodyPose` or `array_like`) – Pose or point \([x, y, z]\) to compute distance to.

**Returns**

Distance to \(v\) from this pose’s origin.

**Return type** `float`

**getModelMatrix** (`inverse=False, out=None`)

Get the present rigid body transformation as a 4x4 matrix.

Matrices are computed only if the `pos` and `ori` attributes have been updated since the last call to `getModelMatrix`. The returned matrix is an `ndarray` and row-major.

**Parameters**

- **inverse** (`bool`, `optional`) – Return the inverse of the model matrix.
- **out** (`ndarray` or `None`) – Optional 4x4 array to write values to. Values written are computed using 32-bit float precision regardless of the data type of `out`.

**Returns**

4x4 transformation matrix.

**Return type** `ndarray`
Examples

Using a rigid body pose to transform something in OpenGL:

```python
rb = RigidBodyPose((0, 0, -2))  # 2 meters away from origin

# Use `array2pointer` from `psychopy.tools.arraytools` to convert
# array to something OpenGL accepts.
mv = array2pointer(rb.modelMatrix)

# use the matrix to transform the scene
glMatrixMode(GL_MODELVIEW)
glPushMatrix()
glLoadIdentity()
glMultTransposeMatrixf(mv)

# draw the thing here ...

glPopMatrix()
```

**getNormalMatrix** (*out=None*)

Get the present normal matrix.

- **Parameters**
  - `out` *(ndarray or None)*: Optional 4x4 array to write values to. Values written are computed using 32-bit float precision regardless of the data type of `out`.

- **Returns**
  - 4x4 normal transformation matrix.

- **Return type**
  - ndarray

**getOriAxisAngle** (*degrees=True*)

Get the axis and angle of rotation for the rigid body. Converts the orientation defined by the ori quaternion to and axis-angle representation.

- **Parameters**
  - `degrees` *(bool, optional)*: Specify True if `angle` is in degrees, or else it will be treated as radians. Default is True.

- **Returns**
  - Axis `[rx, ry, rz]` and angle.

- **Return type**
  - tuple

**getViewMatrix** (*inverse=False*)

Convert this pose into a view matrix.

Creates a view matrix which transforms points into eye space using the current pose as the eye position in the scene. Furthermore, you can use view matrices for rendering shadows if light positions are defined as RigidBodyPose objects.

- **Parameters**
  - `inverse` *(bool)*: Return the inverse of the view matrix. Default is False.

- **Returns**
  - 4x4 transformation matrix.

- **Return type**
  - ndarray

**getYawPitchRoll** (*degrees=True*)

Get the yaw, pitch and roll angles for this pose relative to the -Z world axis.

- **Parameters**
  - `degrees` *(bool, optional)*: Specify True if `angle` is in degrees, or else it will be treated as radians. Default is True.

**interp** *(end, s)*

Interpolate between poses.
Linear interpolation is used on position (Lerp) while the orientation has spherical linear interpolation (Slerp) applied taking the shortest arc on the hypersphere.

Parameters

- **end** (*LibOVRPose*) – End pose.
- **s** (*float*) – Interpolation factor between interval 0.0 and 1.0.

Returns Rigid body pose whose position and orientation is at $s$ between this pose and $end$.

Return type *RigidBodyPose*

**property inverseModelMatrix**

Inverse of the pose as a 4x4 model matrix (read-only).

**invert**()

Invert this pose.

**inverted**()

Get a pose which is the inverse of this one.

Returns This pose inverted.

Return type *RigidBodyPose*

**isEqual**(other)

Check if poses have similar orientation and position.

Parameters other (*RigidBodyPose*) – Other pose to compare.

Returns Returns *True* if poses are effectively equal.

Return type *bool*

**property modelMatrix**

Pose as a 4x4 model matrix (read-only).

**property normalMatrix**

The normal transformation matrix.

**property ori**

Orientation quaternion (X, Y, Z, W).

**property pos**

Position vector (X, Y, Z).

**property posOri**

The position (x, y, z) and orientation (x, y, z, w).

**setIdentity**()

Clear rigid body transformations.

**setOriAxisAngle**(axis, angle, degrees=True)

Set the orientation of the rigid body using an axis and angle. This sets the quaternion at ori.

Parameters

- **axis**(*array_like*) – Axis of rotation [rx, ry, rz].
- **angle**(*float*) – Angle of rotation.
- **degrees**(*bool*, optional) – Specify True if angle is in degrees, or else it will be treated as radians. Default is True.

**transform**(v, out=None)

Transform a vector using this pose.
Parameters

- \(v\) (array_like) – Vector to transform \([x, y, z]\).
- \(out\) (ndarray or None, optional) – Optional array to write values to. Must have the same shape as \(v\).

Returns
Transformed points.

Return type
ndarray

transformNormal \((n)\)
Rotate a normal vector with respect to this pose.

Rotates a normal vector \(n\) using the orientation quaternion at \(ori\).

Parameters
\(n\) (array_like) – Normal to rotate (1-D with length 3).

Returns
Rotated normal \(n\).

Return type
ndarray

property up
Vector defining the up direction (+Y) of this pose.

8.2.27 SceneSkybox

Attributes

8.2.27 SceneSkybox

Attributes

\(\text{SceneSkybox}(\text{win}, \text{tex}, \text{ori}, \text{axis})\)
Class to render scene skyboxes.

Details

\texttt{psychopy.visual.SceneSkybox}(\texttt{win, tex=(), ori=0.0, axis=(0, 1, 0)})

Class to render scene skyboxes.

A skybox provides background imagery to serve as a visual reference for the scene. Background images are projected onto faces of a cube centered about the viewpoint regardless of any viewpoint translations, giving the illusion that the background is very far away. Usually, only one skybox can be rendered per buffer each frame. Render targets must have a depth buffer associated with them.

Background images are specified as a set of image paths passed to faceTextures:

\[
\text{sky} = \text{SceneSkybox(}
\text{win, ('rt.jpg', 'lf.jpg', 'up.jpg', 'dn.jpg', 'bk.jpg', 'ft.jpg'))}
\]

The skybox is rendered by calling \texttt{draw()} after drawing all other 3D stimuli.

Skyboxes are not affected by lighting, however, their colors can be modulated by setting the window’s sceneAmbient value. Skyboxes should be drawn after all other 3D stimuli, but before any successive call that clears the depth buffer (eg. setPerspectiveView, resetEyeTransform, etc.)

Parameters

- \texttt{win} (~psychopy.visual.Window) – Window this skybox is associated with.
- \texttt{tex} (list or tuple or TexCubeMap) – List of files paths to images to use for each face. Images are assigned to faces depending on their index within the list \([+X, -X, +Y, -Y, +Z, -Z]\) or \([\text{right}, \text{left}, \text{top}, \text{bottom}, \text{back}, \text{front}]\). If \texttt{None} is specified, the cube map may be

8.2. \texttt{psychopy.visual} - many visual stimuli 235
specified later by setting the `cubemap` attribute. Alternatively, you can specify a `TexCubeMap` object to set the cube map directly.

- **ori** *(float)* – Rotation of the skybox about `axis` in degrees.
- **axis** *(array_like)* – Axis [ax, ay, az] to rotate about, default is (0, 1, 0).

```python
draw(win=None)
```

Draw the skybox.

This should be called last after drawing other 3D stimuli for performance reasons.

**Parameters**

- **win** *(~psychopy.visual.Window, optional)* – Window to draw the skybox to. If `None`, the window set when initializing this object will be used. The window must share a context with the window which this objects was initialized with.

```python
property skyCubeMap
```

Cubemap for the sky.

### 8.2.28 EnvelopeGrating

**Attributes**

**Details**

```python
class psychopy.visual.EnvelopeGrating(*args, **kwargs)
```

Second-order envelope stimuli with 3 textures; a carrier, an envelope and a mask

**Examples:**

```python
env1 = EnvelopeGrating(win,ori=0, carrier='sin', envelope='sin', mask = 'gauss', sf=24, envsf=4, size=1, contrast=0.5, moddepth=1.0, envori=0, pos=[-,.5,.5],interpolate=0) # gives a circular patch of high frequency carrier with a # low frequency envelope

env2 = EnvelopeGrating(win,ori=0, carrier=noise, envelope='sin', mask = None, sf=24, envsf=4, size=1, contrast=0.5, moddepth=0.8, envori=0, pos=[-,.5,.5],interpolate=0) # If noise is some numpy array containing random values gives a # patch of noise with a low frequency sinusoidal envelope

env4 = EnvelopeGrating(win,ori=90, carrier='sin', envelope='sin', mask = 'gauss', sf=24, envsf=4, size=1, contrast=0.5, moddepth=1.0, envori=0, pos=[.5,.5], beat=True, interpolate=0) # Setting beat will create a second order beat stimulus which # critically contains no net energy at the carrier frequency # even though it appears to be present. In this case carrier # and envelope are at 90 degree to each other
```

With an EnvelopeStim the carrier and envelope can have different spatial frequencies, phases and orientations. Its position can be shifted as a whole.

contrast controls the contrast of the carrier and moddepth the modulation depth of the envelope. contrast and moddepth must work together, for moddepth=1 the max carrier contrast is 0.5 otherwise the displayable range will be exceeded. If moddepth < 1 higher contrasts can be accommodated.

Opacity controls the transparency of the whole stimulus.

Because orientation is implemented very differently for the carrier and envelope using this function without a broadly circular mask may produce unexpected results

**Using EnvelopeStim with images from disk (jpg, tif, png, ...)**

Ideally texture images to be rendered should be square with ‘power-of-2’ dimensions e.g. 16x16, 128x128. Any image that is not will be upsampled (with linear interpolation) to the nearest such texture by PsychoPy. The size of the stimulus should be specified in the normal way using the appropriate units (deg, pix, cm, ...). Be sure to get the aspect ratio the same as the image (if you don’t want it stretched!).

---

**Chapter 8. Reference Manual (API)**
8.2.29 ShapeStim

Attributes

- **ShapeStim**(win[, units, lineWidth, ...])
  - A class for arbitrary shapes defined as lists of vertices (x,y).
- **ShapeStim.win**
  - The Window object in which the
- **ShapeStim.units**
  - None, ‘norm’, ‘cm’, ‘deg’, ‘degFlat’, ‘degFlatPos’, or ‘pix’
- **ShapeStim.vertices**
  - A list of lists or a numpy array (N x 2) specifying xy positions of each vertex, relative to the center of the field.
- **ShapeStim.closeShape**
  - True or False Should the last vertex be automatically connected to the first?
- **ShapeStim.pos**
  - The position of the center of the stimulus in the stimulus units
- **ShapeStim.ori**
  - The orientation of the stimulus (in degrees).
- **ShapeStim.size**
  - Int/Float or x,y-pair.
- **ShapeStim.contrast**
  - A value that is simply multiplied by the color
- **ShapeStim.lineColor**
  - Sets the color of the shape lines.
- **ShapeStimLineColorSpace**
  - Sets color space for line color.
- **ShapeStim.fillColor**
  - Sets the color of the shape fill.
- **ShapeStim.fillColorSpace**
  - Sets color space for fill color.
- **ShapeStim.opacity**
  - Determines how visible the stimulus is relative to backround
- **ShapeStim.interpolate**
  - True or False If True the edge of the line will be antialiased.
- **ShapeStim.name**
  - String or None.
- **ShapeStim.autoLog**
  - Whether every change in this stimulus should be auto logged.
- **ShapeStim.draw**(win, keepMatrix)
  - Draw the stimulus in the relevant window.
- **ShapeStim.autoDraw**
  - Determines whether the stimulus should be automatically drawn on every frame flip.

Details

```python
class psychopy.visual.ShapeStim(win, units='', lineWidth=1.5, lineColor='white', lineColorSpace='rgb', fillColor=None, fillColorSpace='rgb', vertices=((0.0, 0.0), (0.5, 0.0), (0.0, 0.5)), windingRule=None, closeShape=True, pos=(0.0, 0.0), size=1.0, ori=0.0, opacity=1.0, contrast=1.0, depth=0.0, interpolate=True, name=None, autoLog=False, autoDraw=False)
```

A class for arbitrary shapes defined as lists of vertices (x,y).

Shapes can be lines, polygons (concave, convex, self-crossing), or have holes or multiple regions.

`vertices` is typically a list of points (x,y). By default, these are assumed to define a closed figure (polygon); set closeShape=False for a line. closeShape cannot be changed dynamically, but individual vertices can be changed on a frame-by-frame basis. The stimulus as a whole can be rotated, translated, or scaled dynamically (using .ori, .pos, .size).

Vertices can be a string, giving the name of a known set of vertices, although “cross” is the only named shape available at present.

Advanced shapes: `vertices` can also be a list of loops, where each loop is a list of points (x,y), e.g., to define a shape
with a hole. Borders and contains() are not supported for multi-loop stimuli.

windPathRule is an advanced feature to allow control over the GLU tessellator winding rule (default: GLU_TESS_WINDING_ODD). This is relevant only for self-crossing or multi-loop shapes. Cannot be set dynamically.

See Coder demo > stimuli > shapes.py

Changed Nov 2015: v1.84.00. Now allows filling of complex shapes. This is almost completely backwards compatible (see changelog). The old version is accessible as psychopy.visual.BaseShapeStim.

_calcPosRendered()
  DEPRECATED in 1.80.00. This functionality is now handled by _updateVertices() and verticesPix.

_calcSizeRendered()
  DEPRECATED in 1.80.00. This functionality is now handled by _updateVertices() and verticesPix

_getDesiredRGB (rgb, colorSpace, contrast)
  Convert color to RGB while adding contrast. Requires self.rgb, self.colorSpace and self.contrast

_getPolyAsRendered()
  DEPRECATED. Return a list of vertices as rendered.

_selectWindow (win)
  Switch drawing to the specified window. Calls the window’s _setCurrent() method which handles the switch.

_set (attrib, val, op='', log=None)
  DEPRECATED since 1.80.04 + 1. Use setAttribute() and val2array() instead.

_tesselate (newVertices)
  Set the .vertices and .border to new values, invoking tessellation.

_updateList()
  The user shouldn’t need this method since it gets called after every call to .set() Chooses between using and not using shaders each call.

_updateVertices()
  Sets Stim.verticesPix and _borderPix from pos, size, ori, flipVert, flipHoriz

autoDraw
  Determines whether the stimulus should be automatically drawn on every frame flip.
  Value should be: True or False. You do NOT need to set this on every frame flip!

autoLog
  Whether every change in this stimulus should be auto logged.
  Value should be: True or False. Set to False if your stimulus is updating frequently (e.g. updating its position every frame) and you want to avoid swamping the log file with messages that aren’t likely to be useful.

closeShape
  True or False Should the last vertex be automatically connected to the first?
  If you’re using Polygon, Circle or Rect, closeShape= True is assumed and shouldn’t be changed.

color

colorSpace
  The name of the color space currently being used
  Value should be: a string or None
For strings and hex values this is not needed. If None the default colorSpace for the stimulus is used (defined during initialisation).

Please note that changing colorSpace does not change stimulus parameters. Thus you usually want to specify colorSpace before setting the color. Example:

```python
# A light green text
stim = visual.TextStim(win, 'Color me!
    color=(0, 1, 0), colorSpace='rgb')

# An almost-black text
stim.colorSpace = 'rgb255'

# Make it light green again
stim.color = (128, 255, 128)
```

`contains(x, y=None, units=None)`

Returns True if a point x,y is inside the stimulus’ border.

**Can accept variety of input options:**

- two separate args, x and y
- one arg (list, tuple or array) containing two vals (x,y)
- an object with a getPos() method that returns x,y, such as a `Mouse`.

Returns `True` if the point is within the area defined either by its `border` attribute (if one defined), or its `vertices` attribute if there is no `border`. This method handles complex shapes, including concavities and self-crossings.

Note that, if your stimulus uses a mask (such as a Gaussian) then this is not accounted for by the `contains` method; the extent of the stimulus is determined purely by the size, position (pos), and orientation (ori) settings (and by the vertices for shape stimuli).

See Coder demos: shapeContains.py

**contrast**

A value that is simply multiplied by the color

**Value should be: a float between -1 (negative) and 1 (unchanged).** **Operations** supported.

Set the contrast of the stimulus, i.e. scales how far the stimulus deviates from the middle grey. You can also use the stimulus opacity to control contrast, but that cannot be negative.

Examples:

```python
stim.contrast = 1.0  # unchanged contrast
stim.contrast = 0.5  # decrease contrast
stim.contrast = 0.0  # uniform, no contrast
stim.contrast = -0.5 # slightly inverted
stim.contrast = -1.0 # totally inverted
```

Setting contrast outside range -1 to 1 is permitted, but may produce strange results if color values exceed the monitor limits.: 

```python
stim.contrast = 1.2  # increases contrast
stim.contrast = -1.2 # inverts with increased contrast
```

**depth**

DEPRECATED. Depth is now controlled simply by drawing order.
**draw** *(win=None, keepMatrix=False)*

Draw the stimulus in the relevant window. You must call this method after every win.flip() if you want the stimulus to appear on that frame and then update the screen again.

**fillColor**

Sets the color of the shape fill.

See `psychopy.visual.GratingStim.color()` for further details of how to use colors.

Note that shapes where some vertices point inwards will usually not ‘fill’ correctly.

**fillColorSpace**

Sets color space for fill color. See documentation for fillColorSpace

**interpolate**

True or False If True the edge of the line will be antialiased.

**lineColor**

Sets the color of the shape lines.

See `psychopy.visual.GratingStim.color()` for further details of how to use colors.

**lineColorSpace**

Sets color space for line color. See documentation for lineColorSpace

**lineWidth**

int or float specifying the line width in pixels

*Operations* supported.

**name**

String or None. The name of the object to be using during logged messages about this stim. If you have multiple stimuli in your experiment this really helps to make sense of log files!

If name = None your stimulus will be called “unnamed <type>”, e.g. visual.TextStim(win) will be called “unnamed TextStim” in the logs.

**opacity**

Determines how visible the stimulus is relative to background

The value should be a single float ranging 1.0 (opaque) to 0.0 (transparent). *Operations* are supported. Precisely how this is used depends on the *Blend Mode*.

**ori**

The orientation of the stimulus (in degrees).

Should be a single value (scalar). *Operations* are supported.

Orientation convention is like a clock: 0 is vertical, and positive values rotate clockwise. Beyond 360 and below zero values wrap appropriately.

**overlaps**(polygon)

Returns True if this stimulus intersects another one.

If polygon is another stimulus instance, then the vertices and location of that stimulus will be used as the polygon. Overlap detection is typically very good, but it can fail with very pointy shapes in a crossed-swords configuration.

Note that, if your stimulus uses a mask (such as a Gaussian blob) then this is not accounted for by the overlaps method; the extent of the stimulus is determined purely by the size, pos, and orientation settings (and by the vertices for shape stimuli).

See coder demo, shapeContains.py
pos

The position of the center of the stimulus in the stimulus units value should be an x,y-pair. Operations are also supported.

Example:

```
stim.pos = (0.5, 0)  # Set slightly to the right of center
stim.pos += (0.5, -1)  # Increment pos rightwards and upwards.
    Is now (1.0, -1.0)
stim.pos *= 0.2  # Move stim towards the center.
    Is now (0.2, -0.2)
```

Tip: If you need the position of stim in pixels, you can obtain it like this:

```
from psychopy.tools.monitorunittools import posToPix
posPix = posToPix(stim)
```

setAutoDraw (value, log=None)

Sets autoDraw. Usually you can use ‘stim.attribute = value’ syntax instead, but use this method to suppress the log message.

setAutoLog (value=True, log=None)

Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message.

setBorderColor (color, colorSpace=None, operation='', log=None)

Sets both the line and fill to be the same color

setContrast (newContrast, operation='', log=None)

Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message.

setDKL (newDKL, operation='')

DEPRECATED since v1.60.05: Please use the color attribute

setDepth (newDepth, operation='', log=None)

Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message.

setFillColor (color, colorSpace=None, operation='', log=None)

Sets the color of the shape fill.

See `psychopy.visual.GratingStim.color()` for further details.

Note that shapes where some vertices point inwards will usually not ‘fill’ correctly.

setFillRGB (value, operation='')

DEPRECATED since v1.60.05: Please use fillColor()

setLMS (newLMS, operation='')

DEPRECATED since v1.60.05: Please use the color attribute

setLineColor (color, colorSpace=None, operation='', log=None)

Sets the color of the shape edge.

See `psychopy.visual.GratingStim.color()` for further details.

setLineRGB (value, operation='')

DEPRECATED since v1.60.05: Please use lineColor()

setLineWidth (value, operation=' ', log=None)
setOpacity(newOpacity, operation='', log=None)
Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message.

setOri(newOri, operation='', log=None)
Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message.

setPos(newPos, operation='', log=None)
Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message.

setRGB(newRGB, operation='', log=None)
DEPRECATED since v1.60.05: Please use the color attribute

setSize(value, operation='', log=None)
Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message.

setUseShaders(value=True, log=None)
Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message.

setVertices(value=None, operation='', log=None)
Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message.

size
Int/Float or x,y-pair. Sets the size of the shape. Size is independent of the units of shape and will simply scale the shape’s vertices by the factor given. Use a tuple or list of two values to scale asymmetrically.

Operations supported.

units
None, ‘norm’, ‘cm’, ‘deg’, ‘degFlat’, ‘degFlatPos’, or ‘pix’
If None then the current units of the Window will be used. See Units for the window and stimuli for explanation of other options.
Note that when you change units, you don’t change the stimulus parameters and it is likely to change appearance. Example:

```
# This stimulus is 20% wide and 50% tall with respect to window
stim = visual.PatchStim(win, units='norm', size=(0.2, 0.5)

# This stimulus is 0.2 degrees wide and 0.5 degrees tall.
stim.units = 'deg'
```

useShaders
Should shaders be used to render the stimulus (typically leave as True)
If the system support the use of OpenGL shader language then leaving this set to True is highly recommended.
If shaders cannot be used then various operations will be slower (notably, changes to stimulus color or contrast)

vertices
A list of lists or a numpy array (Nx2) specifying xy positions of each vertex, relative to the center of the field.
Assigning to vertices can be slow if there are many vertices.

Operations supported with .setVertices().
property verticesPix
This determines the coordinates of the vertices for the current stimulus in pixels, accounting for size, ori, pos and units

win
The Window object in which the stimulus will be rendered by default. (required)

Example, drawing same stimulus in two different windows and display simultaneously. Assuming that you have two windows and a stimulus (win1, win2 and stim):

```python
stim.win = win1  # stimulus will be drawn in win1
stim.draw()     # stimulus is now drawn to win1
stim.win = win2  # stimulus will be drawn in win2
stim.draw()     # it is now drawn in win2
win1.flip(waitBlanking=False) # do not wait for next monitor update
win2.flip()     # wait for vertical blanking.
```

Note that this just changes *default* window for stimulus.
You could also specify window-to-draw-to when drawing::

```python
stim.draw(win1)
stim.draw(win2)
```

8.2.30 SimpleImageStim
class psychopy.visual.SimpleImageStim(*args, **kwargs)
A simple stimulus for loading images from a file and presenting at exactly the resolution and color in the file (subject to gamma correction if set).

Unlike the ImageStim, this type of stimulus cannot be rescaled, rotated or masked (although flipping horizontally or vertically is possible). Drawing will also tend to be marginally slower, because the image isn’t preloaded to the graphics card. The slight advantage, however is that the stimulus will always be in its original aspect ratio, with no interpolation or other transformation, and it is slightly faster to load into PsychoPy.

8.2.31 Slider
Attributes

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slider(win[, ticks, labels, pos, size, …])</td>
<td>A class for obtaining ratings, e.g., on a 1-to-7 or categorical scale.</td>
</tr>
<tr>
<td>Slider.getRating()</td>
<td>Get the current value of rating (or None if no response yet)</td>
</tr>
<tr>
<td>Slider.getRT()</td>
<td>Get the RT for most recent rating (or None if no response yet)</td>
</tr>
<tr>
<td>Slider.markerPos</td>
<td>The position on the scale where the marker should be.</td>
</tr>
<tr>
<td>Slider.setReadOnly([value, log])</td>
<td>When the rating scale is read only no responses can be made and the scale contrast is reduced</td>
</tr>
<tr>
<td>Slider.contrast</td>
<td>Set all elements of the Slider (labels, ticks, line) to a contrast</td>
</tr>
<tr>
<td>Slider.style</td>
<td>Sets some predefined styles or use these to create your own.</td>
</tr>
</tbody>
</table>

continues on next page
Table 16 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slider.getHistory()</td>
<td>Return a list of the subject’s history as (rating, time) tuples.</td>
</tr>
<tr>
<td>Slider.getMouseResponses()</td>
<td>Instructs the rating scale to check for valid mouse responses.</td>
</tr>
<tr>
<td>Slider.reset()</td>
<td>Resets the slider to its starting state (so that it can be restarted on each trial with a new stimulus)</td>
</tr>
</tbody>
</table>

Details

class psychopy.visual.Slider(win, ticks=(1, 2, 3, 4, 5), labels=None, pos=None, size=None, units=None, flip=False, style='rating', granularity=0, read-Only=False, color='LightGray', font='Helvetica Bold', depth=0, name=None, labelHeight=None, labelWrapWidth=None, auto-Draw=False, autoLog=True)

A class for obtaining ratings, e.g., on a 1-to-7 or categorical scale.

A simpler alternative to RatingScale, to be customised with code rather than with arguments.

A RatingScale instance is a re-usable visual object having a draw() method, with customizable appearance and response options. draw() displays the rating scale, handles the subject's mouse or key responses, and updates the display. When the subject accepts a selection, .noResponse goes False (i.e., there is a response).

You can call the getRating() method anytime to get a rating, getRT() to get the decision time, or getH-istory() to obtain the entire set of (rating, RT) pairs.

For other examples see Coder Demos -> stimuli -> ratingsNew.py.

Authors

- 2018: Jon Peirce

Parameters

- win (psychopy.visual.Window) – Into which the scale will be rendered
- ticks (list or tuple) – A set of values for tick locations. If given a list of numbers then these determine the locations of the ticks (the first and last determine the endpoints and the rest are spaced according to their values between these endpoints.
- labels (a list or tuple) – The text to go with each tick (or spaced evenly across the ticks). If you give 3 labels but 5 tick locations then the end and middle ticks will be given labels. If the labels can't be distributed across the ticks then an error will be raised. If you want an uneven distribution you should include a list matching the length of ticks but with some values set to None
- pos (XY pair (tuple, array or list)) –
- size (w,h pair (tuple, array or list)) – The size for the scale defines the area taken up by the line and the ticks. This also controls whether the scale is horizontal or vertical.
- units (the units to interpret the pos and size) –
- flip (bool) – By default the labels will be below or left of the line. This puts them above (or right)
- granularity (int or float) – The smallest valid increments for the scale. 0 gives a continuous (e.g. “VAS”) scale. 1 gives a traditional likert scale. Something like 0.1 gives a limited fine-grained scale.
• **color** – Color of the line/ticks/labels according to the color space

• **font** *(font name)* –

• **autodraw** –

• **depth** –

• **name** –

• **autoLog** –

  _granularRating*(rating)_
  Handle granularity for the rating

  _setLabelLocs()_
  Calculates the locations of the line, tickLines and labels from the rating info

  _setTickLocs()_
  Calculates the locations of the line, tickLines and labels from the rating info

**Property color**
Color of the line/ticks/labels according to the color space.

**contrast**
Set all elements of the Slider (labels, ticks, line) to a contrast

  **Parameters contrast** –

  **draw()**
  Draw the Slider, with all its constituent elements on this frame

  **getHistory()**
  Return a list of the subject's history as (rating, time) tuples.
  The history can be retrieved at any time, allowing for continuous ratings to be obtained in real-time. Both numerical and categorical choices are stored automatically in the history.

  **getMouseResponses()**
  Instructs the rating scale to check for valid mouse responses.
  This is usually done during the draw() method but can be done by the user as well at any point in time. The rating will be returned but will ALSO automatically be set as the current rating response.
  While the mouse button is down we will alter self.markerPos but don't set a value for self.rating until button comes up
  
  **Returns**
  **Return type** A rating value or **None**

  **getRT()**
  Get the RT for most recent rating (or None if no response yet)

  **getRating()**
  Get the current value of rating (or None if no response yet)

**Property horiz**
(readonly) determines from self.size whether the scale is horizontal

**knownStyles** = ['slider', 'rating', 'radio', 'labels45', 'whiteOnBlack', 'triangleMarker', ...

**markerPos**
The position on the scale where the marker should be. Note that this does not alter the value of the reported rating, only its visible display. Also note that this position is in scale units, not in coordinates
pos
    Set position of slider

    Parameters value (tuple, list) – The new position of slider

rating
    The most recent rating from the participant or None. Note that the position of the marker can be set using
current without looking like a change in the marker position

recordRating (rating, rt=None, log=None)
    Sets the current rating value

reset()
    Resets the slider to its starting state (so that it can be restarted on each trial with a new stimulus)

setReadOnly (value=True, log=None)
    When the rating scale is read only no responses can be made and the scale contrast is reduced

    Parameters
        • value (bool (True)) – The value to which we should set the readOnly flag
        • log (bool or None) – Force the autologging to occur or leave as default

property size
    The size for the scale defines the area taken up by the line and the ticks.

style
    Sets some predefined styles or use these to create your own.

    If you fancy creating and including your own styles that would be great!

    Parameters style (list of strings) – Known styles currently include:

        'rating': the marker is a circle 'triangleMarker': the marker is a triangle 'slider': looks
            more like an application slider control 'whiteOnBlack': a sort of color-inverse rating scale
        'labels45': the text is rotated by 45 degrees 'scrollbar': looks like a scrollbar for a window

    Styles can be combined in a list e.g. ['whiteOnBlack', 'labels45']

8.2.32 SphereStim

Attributes

    SphereStim(win[, radius, subdiv, flipFaces, ...]) Class for drawing a UV sphere.

Details

class psychopy.visual.SphereStim (win, radius=0.5, subdiv=(32, 32), flipFaces=False, pos=(0.0, 0.0, 0.0), ori=(0.0, 0.0, 0.0, 1.0), color=(0.0, 0.0, 0.0), colorSpace='rgb', contrast=1.0, opacity=1.0, useMaterial=None, useShaders=False, name='', autoLog=True)

    Class for drawing a UV sphere.

    The resolution of the sphere mesh can be controlled by setting sectors and stacks which controls the number of
    latitudinal and longitudinal subdivisions, respectively. The radius of the sphere is defined by setting radius expressed
    in scene units (meters if using a perspective projection).

    Calling the draw method will render the sphere to the current buffer. The render target (FBO or back buffer) must
have a depth buffer attached to it for the object to be rendered correctly. Shading is used if the current window has light sources defined and lighting is enabled (by setting useLights=True before drawing the stimulus).

**Warning:** This class is experimental and may result in undefined behavior.

### Examples

Creating a red sphere 1.5 meters away from the viewer with radius 0.25:

```python
redSphere = SphereStim(win,
                      pos=(0., 0., -1.5),
                      radius=0.25,
                      color=(1, 0, 0))
```

### Parameters

- **win** (*psychopy.visual.Window*) – Window this stimulus is associated with. Stimuli cannot be shared across windows unless they share the same context.
- **radius** (*float*) – Radius of the sphere in scene units.
- **subdiv** (*array_like*) – Number of latitudinal and longitudinal subdivisions (lat, long) for the sphere mesh. The greater the number, the smoother the sphere will appear.
- **flipFaces** (*bool, optional*) – If True, normals and face windings will be set to point inward toward the center of the sphere. Texture coordinates will remain the same. Default is False.
- **pos** (*array_like*) – Position vector [x, y, z] for the origin of the rigid body.
- **ori** (*array_like*) – Orientation quaternion [x, y, z, w] where x, y, z are imaginary and w is real. If you prefer specifying rotations in axis-angle format, call setOriAxisAngle after initialization.
- **useMaterial** (*PhongMaterial, optional*) – Material to use. The material can be configured by accessing the material attribute after initialization. If no material is specified, the diffuse and ambient color of the shape will be set by color.
- **color** (*array_like*) – Diffuse and ambient color of the stimulus if useMaterial is not specified. Values are with respect to colorSpace.
- **colorSpace** (*str*) – Colorspace of color to use.
- **contrast** (*float*) – Contrast of the stimulus, value modulates the color.
- **opacity** (*float*) – Opacity of the stimulus ranging from 0.0 to 1.0. Note that transparent objects look best when rendered from farthest to nearest.
- **name** (*str*) – Name of this object for logging purposes.
- **autoLog** (*bool*) – Enable automatic logging on attribute changes.

**_createVAO** (*vertices, textureCoords, normals, faces*)

Create a vertex array object for handling vertex attribute data.

**_getDesiredRGB** (*rgb, colorSpace, contrast*)

Convert color to RGB while adding contrast. Requires self.rgb, self.colorSpace and self.contrast.
_selectWindow(win)
Switch drawing to the specified window. Calls the window’s _setCurrent() method which handles the switch.

_updateList()
The user shouldn’t need this method since it gets called after every call to .set() Chooses between using and not using shaders each call.

color
Color of the stimulus
Value should be one of:

- string: to specify a Colors by name. Any of the standard html/X11 color names <http://www.w3schools.com/html/html_colornames.asp> can be used.
- Colors by hex value
- numerically: (scalar or triplet) for DKL, RGB or other Color spaces. For these, operations are supported.

When color is specified using numbers, it is interpreted with respect to the stimulus’ current colorSpace. If color is given as a single value (scalar) then this will be applied to all 3 channels.

Examples
For whatever stim you have:

```python
stim.color = 'white'
stim.color = 'RoyalBlue'  # (the case is actually ignored)
stim.color = '#DDA0DD'  # DDA0DD is hexadecimal for plum
stim.color = [1.0, -1.0, -1.0]  # if stim.colorSpace='rgb':
    # a red color in rgb space
stim.color = [0.0, 45.0, 1.0]  # if stim.colorSpace='dkl':
    # DKL space with elev=0, azimuth=45
stim.color = [0, 0, 255]  # if stim.colorSpace='rgb255':
    # a blue stimulus using rgb255 space
stim.color = 255  # interpreted as (255, 255, 255)
    # which is white in rgb255.
```

Operations work as normal for all numeric colorSpaces (e.g. ‘rgb’, ‘hsv’ and ‘rgb255’) but not for strings, like named and hex. For example, assuming that colorSpace='rgb':

```python
stim.color += [1, 1, 1]  # increment all guns by 1 value
stim.color *= -1  # multiply the color by -1 (which in this
    # space inverts the contrast)
stim.color *= [0.5, 0, 1]  # decrease red, remove green, keep blue
```

You can use setColor if you want to set color and colorSpace in one line. These two are equivalent:

```python
stim.setColor((0, 128, 255), 'rgb255')
# ... is equivalent to
stim.colorSpace = 'rgb255'
stim.color = (0, 128, 255)
```

colorSpace
The name of the color space currently being used
Value should be: a string or None
For strings and hex values this is not needed. If None the default colorSpace for the stimulus is used (defined during initialisation).

Please note that changing colorSpace does not change stimulus parameters. Thus you usually want to specify colorSpace before setting the color. Example:

```python
# A light green text
stim = visual.TextStim(win, 'Color me!',
    color=(0, 1, 0), colorSpace='rgb')

# An almost-black text
stim.colorSpace = 'rgb255'

# Make it light green again
stim.color = (128, 255, 128)
```

**contrast**

A value that is simply multiplied by the color

**Value should be: a float between -1 (negative) and 1 (unchanged).** Operations supported.

Set the contrast of the stimulus, i.e. scales how far the stimulus deviates from the middle grey. You can also use the stimulus opacity to control contrast, but that cannot be negative.

Examples:

```python
stim.contrast = 1.0  # unchanged contrast
stim.contrast = 0.5  # decrease contrast
stim.contrast = 0.0  # uniform, no contrast
stim.contrast = -0.5 # slightly inverted
stim.contrast = -1.0 # totally inverted
```

Setting contrast outside range -1 to 1 is permitted, but may produce strange results if color values exceeds the monitor limits.:

```python
stim.contrast = 1.2  # increases contrast
stim.contrast = -1.2 # inverts with increased contrast
```

**draw**(win=None)

Draw the stimulus.

This should work for stimuli using a single VAO and material. More complex stimuli with multiple materials should override this method to correctly handle that case.

**Parameters** win (~psychopy.visual.Window) – Window this stimulus is associated with. Stimuli cannot be shared across windows unless they share the same context.

**getOri()**

**getOriAxisAngle**(degrees=True)

Get the axis and angle of rotation for the 3D stimulus. Converts the orientation defined by the ori quaternion to and axis-angle representation.

**Parameters** degrees (bool, optional) – Specify True if angle is in degrees, or else it will be treated as radians. Default is True.

**Returns** Axis [rx, ry, rz] and angle.

**Return type** tuple

**getPos()**
**getRayIntersectBounds** *(rayOrig, rayDir)*

Get the point which a ray intersects the bounding box of this mesh.

**Parameters**

- **rayOrig** (*array_like*) – Origin of the ray in space [x, y, z].
- **rayDir** (*array_like*) – Direction vector of the ray [x, y, z], should be normalized.

**Returns**

Coordinate in world space of the intersection and distance in scene units from *rayOrig*. Returns *None* if there is no intersection.

**Return type** *tuple*

**getRayIntersectSphere** *(rayOrig, rayDir)*

Get the point which a ray intersects the sphere.

**Parameters**

- **rayOrig** (*array_like*) – Origin of the ray in space [x, y, z].
- **rayDir** (*array_like*) – Direction vector of the ray [x, y, z], should be normalized.

**Returns**

Coordinate in world space of the intersection and distance in scene units from *rayOrig*. Returns *None* if there is no intersection.

**Return type** *tuple*

**isVisible** *

Check if the object is visible to the observer.

Test if a pose’s bounding box or position falls outside of an eye’s view frustum.

Poses can be assigned bounding boxes which enclose any 3D models associated with them. A model is not visible if all the corners of the bounding box fall outside the viewing frustum. Therefore any primitives (i.e. triangles) associated with the pose can be culled during rendering to reduce CPU/GPU workload.

**Returns** *True* if the object’s bounding box is visible.

**Return type** *bool*

**Examples**

You can avoid running draw commands if the object is not visible by doing a visibility test first:

```python
if myStim.isVisible():
    myStim.draw()
```

**property ori**

Orientation quaternion (X, Y, Z, W).

**property pos**

Position vector (X, Y, Z).

**setBorderColor** *(color, colorSpace=None, operation='', log=None)*

Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message and/or set colorSpace simultaneously.

**setColor** *(color, colorSpace=None, operation='', log=None)*

Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message.
PsychoPy - Psychology software for Python, Release 2020.2.10

setFillColor (color, colorSpace=None, operation='', log=None)

setOri (ori)

setOriAxisAngle (axis, angle, degrees=True)

Set the orientation of the 3D stimulus using an axis and angle. This sets the quaternion at ori.

Parameters

- **axis** (array_like) – Axis of rotation [rx, ry, rz].
- **angle** (float) – Angle of rotation.
- **degrees** (bool, optional) – Specify True if angle is in degrees, or else it will be treated as radians. Default is True.

setPos (pos)

setUseShaders (value=True, log=None)

Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message

property thePose

The pose of the rigid body. This is a class which has pos and ori attributes.

units

None, ‘norm’, ‘cm’, ‘deg’, ‘degFlat’, ‘degFlatPos’, or ‘pix’

If None then the current units of the Window will be used. See Units for the window and stimuli for explanation of other options.

Note that when you change units, you don’t change the stimulus parameters and it is likely to change appearance. Example:

```python
# This stimulus is 20% wide and 50% tall with respect to window
stim = visual.PatchStim(win, units='norm', size=(0.2, 0.5)

# This stimulus is 0.2 degrees wide and 0.5 degrees tall.
stim.units = 'deg'
```

useShaders

Should shaders be used to render the stimulus (typically leave as True)

If the system support the use of OpenGL shader language then leaving this set to True is highly recommended. If shaders cannot be used then various operations will be slower (notably, changes to stimulus color or contrast)

win

The Window object in which the stimulus will be rendered by default. (required)

Example, drawing same stimulus in two different windows and display simultaneously. Assuming that you have two windows and a stimulus (win1, win2 and stim):

```
stim.win = win1  # stimulus will be drawn in win1
stim.draw()     # stimulus is now drawn to win1
stim.win = win2  # stimulus will be drawn in win2
stim.draw()     # it is now drawn in win2
win1.flip(waitBlanking=False) # do not wait for next # monitor update
    # wait for vertical blanking.
win2.flip() # wait for vertical blanking.
```

Note that this just changes **default** window for stimulus.
You could also specify window-to-draw-to when drawing::

(continues on next page)
8.2.33 TextBox

Attributes

| TextBox[(window, text, font_name, bold, …)] | Similar to the visual.TextStim component, TextBox can be used to display text within a psychopy window. |

Note: The following set______() attributes all have equivalent get______() attributes:

| TextBox.setText(text_source) | Set the text to be displayed within the Textbox. |
| TextBox.setPosition(pos) | Set the (x,y) position of the TextBox on the Monitor. |
| TextBox.setHorzAlign(v) | Specify how the horizontal (x) component of the TextBox position is to be interpreted. |
| TextBox.setVertAlign(v) | Specify how the vertical (y) component of the TextBox position is to be interpreted. |
| TextBox.setHorzJust(v) | Specify how text within the TextBox should be aligned horizontally. |
| TextBox.setVertJust(v) | Specify how text within the TextBox should be aligned vertically. |
| TextBox.setFontColor(c) | Set the color to use when drawing text glyphs within the TextBox. |
| TextBox.setBorderColor(c) | Set the color to use for the border of the TextBox. |
| TextBox.setBackgroundColor(c) | Set the fill color used to fill the rectangular area of the TextBox stim. |
| TextBox.setTextGridLineColor(c) | Set the color used when drawing text grid lines. |
| TextBox.setTextGridLineWidth(c) | Set the stroke width (in pixels) to use for the text grid character bounding boxes. |
| TextBox.setInterpolated(interpolate) | Specify whether interpolation should be enabled for the TextBox when it is drawn. |
| TextBox.setOpacity(o) | Sets the TextBox transparency level to use for color related attributes of the Textbox. |
| TextBox.setAutoLog(v) | Specify if changes to TextBox attribute values should be logged automatically by PsychoPy. True enables auto logging; False disables it. *Currently not supported by TextBox. |
| TextBox.draw() | Draws the TextBox to the back buffer of the graphics card. |

Note: TextBox also provides the following read-only functions:
TextBox getSize() Return the width, height of the TextBox, using the unit type being used by the stimulus.

TextBox getName() Same as the GetLabel method.

TextBox getDisplayedText() Return the text that fits within the TextBox and therefore is actually seen.

TextBox getValidStrokeWidths() Returns the stroke width range supported by the graphics card being used.

TextBox getLineSpacing() Return the additional spacing being applied between rows of text.

TextBox getGlyphPositionForTextIndex(char_index) For the provided char_index, which is the index of one character in

TextBox getTextGridCellPlacement() Returns a 3d numpy array containing position information for each text grid cell in the TextBox.

Helper functions:

getFontManager()

FontManager provides a simple API for finding and loading font files (.ttf) via the FreeType lib

The FontManager finds supported font files on the computer and initially creates a dictionary containing the information about available fonts. This can be used to quickly determine what font family names are available on the computer and what styles (bold, italic) are supported for each family.

This font information can then be used to create the resources necessary to display text using a given font family, style, size, color, and dpi.

The FontManager is currently used by the psychopy.visual.TextBox stim type. A user script can access the FontManager via:

    font_mngr=visual.textbox.getFontManager()

Once a font of a given size and dpi has been created; it is cached by the FontManager and can be used by all TextBox instances created within the experiment.

Details

class psychopy.visual.TextBox (window=None, text='Default Test Text.', font_name=None, bold=False, italic=False, font_size=32, font_color=(0, 0, 0, 1), dpi=72, line_spacing=0, line_spacing_units='pix', background_color=None, border_color=None, border_stroke_width=1, size=None, textgrid_shape=None, pos=(0.0, 0.0), align_horz='center', align_vert='center', units='norm', grid_color=None, grid_stroke_width=1, color_space='rgb', opacity=1.0, grid_horz_justification='left', grid_vert_justification='top', autoLog=True, interpolate=False, name=None)

Similar to the visual.TextStim component, TextBox can be used to display text within a psychopy window. TextBox and TextStim each have different strengths and weaknesses. You should select the most appropriate text component type based on how it will be used within the experiment.

NOTE: As of PsychoPy 1.79, TextBox should be considered experimental. The two TextBox demo scripts provided have been tested on all PsychoPy supported OS's and run without exceptions. However there are very likely bugs in the existing TextBox code and the TextBox API will be further enhanced and improved (i.e. changed) over the next couple months.
• Text character placement is very well defined, useful when the exact positioning of each letter needs to be known.

• The text string that is displayed can be changed ( setText() ) and drawn ( win.draw() ) very quickly. See the TextBox vs. TextStim comparison table for details.

• Built-in font manager; providing easy access to the font family names and styles that are available on the computer being used.

• TextBox is a composite stimulus type, with the following graphical elements:
  – TextBox Border / Outline
  – TextBox Fill Area
  – Text Grid Cell Lines
  – Text Glyphs

Attributes for each of the TextBox graphical elements can be changed to control many aspects of how the TextBox is displayed.

• When using ‘rgb’ or ‘rgb255’ color spaces, colors can be specified as a list/tuple of 3 elements (red, green, blue), or with four elements (red, green, blue, alpha) which allows different elements of the TextBox to use different opacity settings if desired. For colors that include the alpha channel value, it will be applied instead of the opacity setting of the TextBox, effectively overriding the stimulus defined opacity for that part of the textbox graphics. Colors that do not include an alpha channel use the opacity setting as normal.

• Text Line Spacing can be controlled.

• Only Monospace Fonts are supported.

• TextBox component is not a completely standard psychopy visual stim and has the following functional difference:
  – TextBox attributes are never accessed directly; get* and set* methods are always used (this will be changed to use class properties in the future).
  – Setting an attribute of a TextBox only supports value replacement, ( textbox.setFontColor([1.0,1.0,1.0]) ) and does not support specifying operators.

• Some key word arguments supported by other stimulus types in general, or by TextStim itself, are not supported by TextBox. See the TextBox class definition for the arguments that are supported.

• When a new font, style, and size are used it takes about 1 second to load and process the font. This is a one time delay for a given font name, style, and size. After first being loaded, the same font style can be used or re-applied to multiple TextBox components with no significant delay.

• Auto logging or auto drawing is not currently supported.

<table>
<thead>
<tr>
<th>Feature</th>
<th>TextBox</th>
<th>TextStim</th>
</tr>
</thead>
<tbody>
<tr>
<td>Change text + redraw time</td>
<td>1.513 msec</td>
<td>28.537 msec</td>
</tr>
<tr>
<td>No change + redraw time</td>
<td>0.240 msec</td>
<td>0.931 msec</td>
</tr>
<tr>
<td>Initial Creation time</td>
<td>0.927 msec</td>
<td>0.194 msec</td>
</tr>
<tr>
<td>MonoSpace Font Support</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Non MonoSpace Font Support</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Adjustable Line Spacing</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Precise Text Pos. Info</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Auto logging Support</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Rotation Support</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Word Wrapping Support</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
^ Times are in msec.usec format. Tested using the textstim_vsTextbox.py demo script provided with the PsychoPy distribution. Results are dependent on text length, video card, and OS. Displayed results are based on 120 character string with an average of 24 words. Test computer used Windows 7 64 bit, PsychoPy 1.7.9, with a i7 3.4 Ghz CPU, 8 GB RAM, and NVIDIA 480 GTX 2GB graphics card.

from psychopy import visual
win=visual.Window((…))

# A Textbox stim that will look similar to a TextStim component #
textstimlike=visual.TextBox(
    window=win, text="This textbox looks most like a textstim.", font_size=18, font_color=[-1,-1,1], color_space='rgb', size=(1.8,.1), pos=(0.0,.5), units='norm'
)

# A Textbox stim that uses more of the supported graphical features #
textboxloaded=visual.TextBox(
    window=win, text='TextBox showing all supported graphical elements', font_size=32, font_color=[1,1,1], border_color=[-1,-1,1], # draw a blue border around stim border_stroke_width=4, # border width of 4 pix. background_color=[-1,-1,-1], # fill the stim background grid_color=[1,-1,-1,0.5], # draw a red line around each grid_stroke_width=1, # with a width of 1 pix textgrid_shape=[20,2], # specify area of text box # by the number of cols x # number of rows of text to support # instead of by a screen # units width x height. pos=(0.0,-.5), # If the text string length < num rows * num cols in # textgrid_shape, how should text be justified? # grid_horz_justification='center', grid_vert_justification='center', )
textstimlike.draw() textboxloaded.draw() win.flip()

draw()
Draws the TextBox to the back buffer of the graphics card. Then call win.flip() to display the changes drawn. If draw() is not called prior to a call to win.flip(), the TextBox will not be displayed for that retrace.

getAutoLog()
Indicates if changes to TextBox attribute values should be logged automatically by PsychoPy. *Currently not supported by TextBox.

getBackgroundColor()
Get the color used to fill the rectangular area of the TextBox stim. All other graphical elements of the TextBox are drawn on top of the background.

getBorderColor()
A border can be drawn around the perimeter of the TextBox. This method sets the color of that border.

getBorderWidth()
Get the stroke width of the optional TextBox area outline. This is always given in pixel units.

getColorSpace()
Returns the psychopy color space used when specifying colors for the TextBox. Supported values are:

• ‘rgb’
• ‘rgb255’
• ‘norm’
• hex (implicit)
• html name (implicit)

See the Color Space section of the PsychoPy docs for details.

**getDisplayedText()**

Return the text that fits within the TextBox and therefore is actually seen. This is equal to:

```python
text_length = len(self.getText())
cols, rows = self.getTextGridShape()
displayed_text = self.getText()[0:min(text_length, rows*cols)]
```

**getFontColor()**

Return the color used when drawing text glyphs.

**getGlyphPositionForTextIndex(char_index)**

For the provided char_index, which is the index of one character in the current text being displayed by the TextBox (getDisplayedText()), return the bounding box position, width, and height for the associated glyph drawn to the screen. This factors in the glyphs position within the textgrid cell it is being drawn in, so the returned bounding box is for the actual glyph itself, not the textgrid cell.

For textgrid cell placement information, see the getTextGridCellPlacement() method.

The glyph position for the given text index is returned as a tuple (x,y,width,height), where x,y is the top left hand corner of the bounding box.

**Special Cases:**

- If the index provided is out of bounds for the currently displayed text, None is returned.
- For u' ' (space) characters, the full textgrid cell bounding box is returned.
- For u'

  ' ( new line ) characters, the textgrid cell bounding box is returned, but with the box width set to 0.

**getHorzAlign()**

Return what textbox x position should be interpreted as. Valid options are ‘left’, ‘center’, or ‘right’.

**getHorzJust()**

Return how text should laid out horizontally when the number of columns of each text grid row is greater than the number needed to display the text for that text row.

**getInterpolated()**

Returns whether interpolation is enabled for the TextBox when it is drawn. When True, GL_LINE_SMOOTH and GL_POLYGON_SMOOTH are enabled within OpenGL; otherwise they are disabled.

**getLabel()**

Return the label / name assigned to the textbox. This does not impact how the stimulus looks when drawn, and instead is used for internal purposes only.

**getLineSpacing()**

Return the additional spacing being applied between rows of text. The value is in units specified by the textbox getUnits() method.

**getName()**

Same as the GetLabel method.

**getOpacity()**

Get the default TextBox transparency level used for color related attributes. 0.0 equals fully transparent, 1.0 equals fully opaque.

**getPosition()**

Return the x,y position of the textbox, in getUnitType() coord space.
**getSizes()**
Return the width, height of the TextBox, using the unit type being used by the stimulus.

**getText()**
Return the text to display.

**getTextGridCellForCharIndex(char_index)**

**getTextGridCellPlacement()**
Returns a 3d numpy array containing position information for each text grid cell in the TextBox. The array has the shape (num_cols, num_rows, cell_bounds), where num_cols is the number of textgrid columns in the TextBox, num_rows is the number of textgrid rows in the TextBox, cell_bounds is a 4 element array containing the (x pos, y pos, width, height) data for the given cell. Position fields are for the top left hand corner of the cell box. Column and Row indices start at 0.

To get the shape of the textgrid in terms of columns and rows, use:

```python
cell_pos_array = textbox.getTextGridCellPlacement()
col_row_count = cell_pos_array.shape[:2]
```

To access the position, width, and height for textgrid cell at column 0 and row 0 (so the top left cell in the textgrid):

```python
cell00 = cell_pos_array[0,0, :]
```

For the cell at col 3, row 1 (so 4th cell on second row):

```python
cell41 = cell_pos_array[4,1, :]
```

**getTextGridLineColor()**
Return the color used when drawing the outline of the text grid cells. Each letter displayed in a TextBox populates one of the text cells defined by the shape of the TextBox text grid. Color value must be valid for the color space being used by the TextBox.

A value of None indicates drawing of the textgrid lines is disabled.

**getTextGridLineWidth()**
Return the stroke width (in pixels) of the optional lines drawn around the text grid cell areas.

**getUnitType()**
Returns which of the psychopy coordinate systems are used by the TextBox. Position and size related attributes must be specified relative to the unit type being used. Valid options are:

- pix
- norm
- cm

**getValidStrokeWidths()**
Returns the stroke width range supported by the graphics card being used. If the TextBox is Interpolated, a tuple is returns using float values, with the following structure:

```python
((min_line_width, max_line_width), line_width_granularity)
```

If Interpolation is disabled for the TextBox, the returned tuple elements are int values, with the following structure:

```python
(min_line_width, max_line_width)
```

**getVertAlign()**
Return what textbox y position should be interpreted as. Valid options are ‘top’, ‘center’, or ‘bottom’.

**getVertJust()**
Return how text should laid out vertically when the number of text grid rows is greater than the number needed to display the current text
**getWindow()**

Returns the psychopy window that the TextBox is associated with.

**setAutoLog(v)**

Specify if changes to TextBox attribute values should be logged automatically by PsychoPy. True enables auto logging; False disables it. *Currently not supported by TextBox.*

**setBackgroundColor(c)**

Set the fill color used to fill the rectangular area of the TextBox stim. Color value must be valid for the color space being used by the TextBox.

A value of None will disable drawing of the TextBox background.

**setBorderColor(c)**

Set the color to use for the border of the TextBox. The TextBox border is a rectangular outline drawn around the edges of the TextBox stim. Color value must be valid for the color space being used by the TextBox.

A value of None will disable drawing of the border.

**setBorderWidth(c)**

Set the stroke width (in pixels) to use for the border of the TextBox stim. Border values must be within the range of stroke widths supported by the OpenGL driver used by the graphics. Setting the width outside the valid range will result in the stroke width being clamped to the nearest end of the valid range.

Use the TextBox.getValidStrokeWidths() to access the minimum - maximum range of valid line widths.

**setFontColor(c)**

Set the color to use when drawing text glyphs within the TextBox. Color value must be valid for the color space being used by the TextBox. For ‘rgb’, ‘rgb255’, and ‘norm’ based colors, three or four element lists are valid. Three element colors use the TextBox.getOpacity() value to determine the alpha channel for the color. Four element colors use the value of the fourth element to set the alpha value for the color.

**setHorzAlign(v)**

Specify how the horizontal (x) component of the TextBox position is to be interpreted. left = x position is the left edge, right = x position is the right edge x position, and center = the x position is used to center the stim horizontally.

**setHorzJust(v)**

Specify how text within the TextBox should be aligned horizontally. For example, if a text grid has 10 columns, and the text being displayed is 6 characters in length, the horizontal justification determines if the text should be drawn starting at the left of the text columns (left), or should be centered on the columns (‘center’, in this example there would be two empty text cells to the left and right of the text.), or should be drawn such that the last letter of text is drawn in the last column of the text row (‘right’).

**setInterpolated(interpolate)**

Specify whether interpolation should be enabled for the TextBox when it is drawn. When interpolate == True, GL_LINE_SMOOTH and GL_POLYGON_SMOOTH are enabled within OpenGL. When interpolate is set to False, GL_POLYGON_SMOOTH and GL_LINE_SMOOTH are disabled.

**setOpacity(o)**

Sets the TextBox transparency level to use for color related attributes of the Textbox. 0.0 equals fully transparent, 1.0 equals fully opaque.

If opacity is set to None, it is assumed to have a default value of 1.0.

When a color is defined with a 4th element in the colors element list, then this opacity value is ignored and the alpha value provided in the color itself is used for that TextGrid element instead.

**setPosition(pos)**

Set the (x,y) position of the TextBox on the Monitor. The position must be given using the unit coord type used by the stim.
The TextBox position is interpreted differently depending on the Horizontal and Vertical Alignment settings of the stim. See getHorzAlignment() and getVertAlignment() for more information.

For example, if the TextBox alignment is specified as left, top, then the position specifies the top left hand corner of where the stim will be drawn. An alignment of bottom, right indicates that the position value will define where the bottom right corner of the TextBox will be drawn. A horz., vert. alignment of center, center will place the center of the TextBox at pos.

**setText** *(text_source)*

Set the text to be displayed within the TextBox.

Note that once a TextBox has been created, the number of character rows and columns is static. To change the size of a TextBox, a new TextBox stim must be created to replace the current TextBox stim. Therefore ensure that the textbox is large enough to display the largest length string to be presented in the TextBox. Characters that do not fit within the TextBox will not be displayed.

Color value must be valid for the color space being used by the TextBox.

**setTextGridLineColor** *(c)*

Set the color used when drawing text grid lines. These are lines that can be drawn which mark the bounding box for each character within the TextBox text grid. Color value must be valid for the color space being used by the TextBox.

Provide a value of None to disable drawing of text grid lines.

**setTextGridLineWidth** *(c)*

Set the stroke width (in pixels) to use for the text grid character bounding boxes. Border values must be within the range of stroke widths supported by the OpenGL driver used by the computer graphics card. Setting the width outside the valid range will result in the stroke width being clamped to the nearest end of the valid range.

Use the TextBox.getGLLineRanges() to access a dict containing some OpenGL parameters which provide the minimum, maximum, and resolution of valid line widths.

**setVertAlign** *(v)*

Specify how the vertical (y) component of the TextBox position is to be interpreted. top = y position is the top edge, bottom = y position is the bottom edge y position, and center = the y position is used to center the stim vertically.

**setVertJust** *(v)*

Specify how text within the TextBox should be aligned vertically. For example, if a text grid has 3 rows for text, and the text being displayed all fits on one row, the vertical justification determines if the text should be draw on the top row of the text grid (top), or should be centered on the rows (center’, in this example there would be one row above and below the row used to draw the text), or should be drawn on the last row of the text grid, (‘bottom’).

### 8.2.34 TextStim

**class psychopy.visual.TextStim**(win, text='Hello World', font='', pos=(0.0, 0.0), depth=0, rgb=None, color=(1.0, 1.0, 1.0), colorSpace='rgb', opacity=1.0, contrast=1.0, units='', ori=0.0, height=None, antialias=True, bold=False, italic=False, alignHoriz=None, alignVert=None, alignText='center', anchorHoriz='center', anchorVert='center', fontFiles=(), wrapWidth=None, flipHoriz=False, flipVert=False, languageStyle='LTR', name=None, autoLog=None)

Class of text stimuli to be displayed in a Window

Performance OBS: in general, TextStim is slower than many other visual stimuli, i.e. it takes longer to change some attributes. In general, it's the attributes that affect the shapes of the letters: text, height, font, bold
etc. These make the next .draw() slower because that sets the text again. You can make the draw() quick by calling re-setting the text (myTextStim.text = myTextStim.text) when you’ve changed the parameters.

In general, other attributes which merely affect the presentation of unchanged shapes are as fast as usual. This includes pos, opacity etc.

The following attribute can only be set at initialization (see further down for a list of attributes which can be changed after initialization):

**languageStyle**  Apply settings to correctly display content from some languages that are written right-to-left. Currently there are three (case-insensitive) values for this parameter:

- **'LTR'** is the default, for typical left-to-right, Latin-style languages.
- **'RTL'** will correctly display text in right-to-left languages such as Hebrew. By applying the bidirectional algorithm, it allows mixing portions of left-to-right content (such as numbers or Latin script) within the string.
- **'Arabic'** applies the bidirectional algorithm but additionally will _reshape_ Arabic characters so they appear in the cursive, linked form that depends on neighbouring characters, rather than in their isolated form. May also be applied in other scripts, such as Farsi or Urdu, that use Arabic-style alphabets.

**Parameters**

- **_calcPosRendered()**
  DEPRECATED in 1.80.00. This functionality is now handled by _updateVertices() and verticesPix.

- **_calcSizeRendered()**
  DEPRECATED in 1.80.00. This functionality is now handled by _updateVertices() and verticesPix.

- **_getDesiredRGB(rgb, colorSpace, contrast)**
  Convert color to RGB while adding contrast. Requires self.rgb, self.colorSpace and self.contrast

- **_getPolyAsRendered()**
  DEPRECATED. Return a list of vertices as rendered.

- **_selectWindow(win)**
  Switch drawing to the specified window. Calls the window’s _setCurrent() method which handles the switch.

- **_set(attrib, val='", log=None)**
  DEPRECATED since 1.80.04 + 1. Use setAttribute() and val2array() instead.

- **_setTextNoShaders(value=None)**
  Set the text to be rendered using the current font

- **_setTextShaders(value=None)**
  Set the text to be rendered using the current font

- **_updateList()**
  The user shouldn’t need this method since it gets called after every call to .set() Chooses between using and not using shaders each call.

- **_updateListNoShaders()**
  The user shouldn’t need this method since it gets called after every call to .set() Basically it updates the OpenGL representation of your stimulus if some parameter of the stimulus changes. Call it if you change a property manually rather than using the .set() command

- **_updateListShaders()**
  Only used with pygame text - pyglet handles all from the draw()
_updateVertices()
Sets Stim.verticesPix and _borderPix from pos, size, ori, flipVert, flipHoriz

alignHoriz
Deprecated in PsychoPy 3.3. Use alignText and anchorHoriz instead

alignText
Aligns the text content within the bounding box (‘left’, ‘right’ or ‘center’) See also anchorX to set alignment of the box itself relative to pos

alignVert
Deprecated in PsychoPy 3.3. Use anchorVert

anchorHoriz
The horizontal alignment (‘left’, ‘right’ or ‘center’)

anchorVert
The vertical alignment (‘top’, ‘bottom’ or ‘center’) of the box relative to the text pos.

antialias
Allow antialiasing the text (True or False). Sets text, slow.

autoDraw
Determines whether the stimulus should be automatically drawn on every frame flip.

Value should be: True or False. You do NOT need to set this on every frame flip!

autoLog
Whether every change in this stimulus should be auto logged.

Value should be: True or False. Set to False if your stimulus is updating frequently (e.g. updating its position every frame) and you want to avoid swamping the log file with messages that aren’t likely to be useful.

bold
Make the text bold (True, False) (better to use a bold font name).

property boundingBox
(read only) attribute representing the bounding box of the text (w,h). This differs from width in that the width represents the width of the margins, which might differ from the width of the text within them.

NOTE: currently always returns the size in pixels (this will change to return in stimulus units)

color
Color of the stimulus

Value should be one of:

- string: to specify a Colors by name. Any of the standard html/X11 color names <http://www.w3schools.com/html/html_colornames.asp> can be used.

- Colors by hex value

- numerically: (scalar or triplet) for DKL, RGB or other Color spaces. For these, operations are supported.

When color is specified using numbers, it is interpreted with respect to the stimulus’ current colorSpace. If color is given as a single value (scalar) then this will be applied to all 3 channels.
Examples

For whatever stim you have:

```python
stim.color = 'white'
stim.color = 'RoyalBlue'  # (the case is actually ignored)
stim.color = '#DDA0DD'  # DDA0DD is hexadecimal for plum
stim.color = [1.0, -1.0, -1.0]  # if stim.colorSpace='rgb':
    # a red color in rgb space
stim.color = [0.0, 45.0, 1.0]  # if stim.colorSpace='dkl':
    # DKL space with elev=0, azimuth=45
stim.color = [0, 0, 255]  # if stim.colorSpace='rgb255':
    # a blue stimulus using rgb255 space
stim.color = 255  # interpreted as (255, 255, 255)
    # which is white in rgb255.
```

Operations work as normal for all numeric colorSpaces (e.g. 'rgb', 'hsv' and 'rgb255') but not for strings, like named and hex. For example, assuming that colorSpace='rgb':

```python
stim.color += [1, 1, 1]  # increment all guns by 1 value
stim.color *= -1  # multiply the color by -1 (which in this
    # space inverts the contrast)
stim.color *= [0.5, 0, 1]  # decrease red, remove green, keep blue
```

You can use `setColor` if you want to set color and colorSpace in one line. These two are equivalent:

```python
stim.setColor((0, 128, 255), 'rgb255')
    # ... is equivalent to
stim.colorSpace = 'rgb255'
stim.color = (0, 128, 255)
```

**colorSpace**

The name of the color space currently being used

Value should be: a string or None

For strings and hex values this is not needed. If None the default colorSpace for the stimulus is used (defined during initialisation).

Please note that changing colorSpace does not change stimulus parameters. Thus you usually want to specify colorSpace before setting the color. Example:

```python
# A light green text
stim = visual.TextStim(win, 'Color me!',
    color=(0, 1, 0), colorSpace='rgb')

# An almost-black text
stim.colorSpace = 'rgb255'

# Make it light green again
stim.color = (128, 255, 128)
```

**contains** *(x, y=None, units=None)*

Returns True if a point x,y is inside the stimulus’ border.

**Can accept variety of input options:**

- two separate args, x and y
- one arg (list, tuple or array) containing two vals (x,y)
• an object with a getPos() method that returns x,y, such as a Mouse.

Returns True if the point is within the area defined either by its border attribute (if one defined), or its vertices attribute if there is no border. This method handles complex shapes, including concavities and self-crossings. Note that, if your stimulus uses a mask (such as a Gaussian) then this is not accounted for by the contains method; the extent of the stimulus is determined purely by the size, position (pos), and orientation (ori) settings (and by the vertices for shape stimuli).

See Coder demos: shapeContains.py See Coder demos: shapeContains.py

**contrast**

A value that is simply multiplied by the color

Value should be: a float between -1 (negative) and 1 (unchanged). Operations supported.

Set the contrast of the stimulus, i.e. scales how far the stimulus deviates from the middle grey. You can also use the stimulus opacity to control contrast, but that cannot be negative.

Examples:

```python
stim.contrast = 1.0  # unchanged contrast
stim.contrast = 0.5  # decrease contrast
stim.contrast = 0.0  # uniform, no contrast
stim.contrast = -0.5 # slightly inverted
stim.contrast = -1.0 # totally inverted
```

Setting contrast outside range -1 to 1 is permitted, but may produce strange results if color values exceeds the monitor limits.:

```python
stim.contrast = 1.2  # increases contrast
stim.contrast = -1.2 # inverts with increased contrast
```

**depth**

DEPRECATED. Depth is now controlled simply by drawing order.

**draw** *(win=None)*

Draw the stimulus in its relevant window. You must call this method after every MyWin.flip() if you want the stimulus to appear on that frame and then update the screen again.

If win is specified then override the normal window of this stimulus.

flipHoriz

If set to True then the text will be flipped left-to-right. The flip is relative to the original, not relative to the current state.

flipVert

If set to True then the text will be flipped top-to-bottom. The flip is relative to the original, not relative to the current state.

**font**

String. Set the font to be used for text rendering. font should be a string specifying the name of the font (in system resources).

**fontFiles**

A list of additional files if the font is not in the standard system location (include the full path).

OBS: fonts are added every time this value is set. Previous are not deleted.

E.g.
height
The height of the letters (Float/int or None = set default).
Height includes the entire box that surrounds the letters in the font. The width of the letters is then defined
by the font.
*Operations* supported.

italic
True/False. Make the text italic (better to use a italic font name).

name
String or None. The name of the object to be using during logged messages about this stim. If you have
multiple stimuli in your experiment this really helps to make sense of log files!
If name = None your stimulus will be called “unnamed <type>”, e.g. visual.TextStim(win) will be called
“unnamed TextStim” in the logs.

ori
The orientation of the stimulus (in degrees).
Should be a single value (*scalar*). *Operations* are supported.
Orientation convention is like a clock: 0 is vertical, and positive values rotate clockwise. Beyond 360 and
below zero values wrap appropriately.

overlaps (*polygon*)
Returns *True* if this stimulus intersects another one.
If *polygon* is another stimulus instance, then the vertices and location of that stimulus will be used as the
polygon. Overlap detection is typically very good, but it can fail with very pointy shapes in a crossed-swords
configuration.
Note that, if your stimulus uses a mask (such as a Gaussian blob) then this is not accounted for by the *overlaps*
method; the extent of the stimulus is determined purely by the size, pos, and orientation settings (and by the
vertices for shape stimuli).
See coder demo, shapeContains.py

pos
The position of the center of the stimulus in the stimulus *units*
value should be an *x,y-pair*. *Operations* are also supported.

Example:

```
stim.pos = (0.5, 0)  # Set slightly to the right of center
stim.pos += (0.5, -1)  # Increment pos rightwards and upwards.
    Is now (1.0, -1.0)
stim.pos *= 0.2  # Move stim towards the center.
    Is now (0.2, -0.2)
```

Tip: If you need the position of stim in pixels, you can obtain it like this:
```
from psychopy.tools.monitorunittools import posToPix
posPix = posToPix(stim)
```

**property** posPix
This determines the coordinates in pixels of the position for the current stimulus, accounting for pos and units.
This property should automatically update if *pos* is changed
setAutoDraw(value, log=None)
    Sets autoDraw. Usually you can use ‘stim.attribute = value’ syntax instead, but use this method to suppress the log message.

setAutoLog(value=True, log=None)
    Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message.

setColor(color, colorSpace=None, operation='', log=None)
    Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message and/or set colorSpace simultaneously.

setContrast(newContrast, operation='', log=None)
    Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message

setDKL(newDKL, operation='')
    DEPRECATED since v1.60.05: Please use the color attribute

setDepth(newDepth, operation='', log=None)
    Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message

setFlip(direction, log=None)
    (used by Builder to simplify the dialog)

setFlipHoriz(newVal=True, log=None)
    Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message.

setFlipVert(newVal=True, log=None)
    Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message

setFont(font, log=None)
    Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message.

setHeight(height, log=None)
    Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message.

setLMS(newLMS, operation='')
    DEPRECATED since v1.60.05: Please use the color attribute

setOpacity(newOpacity, operation='', log=None)
    Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message

setOri(newOri, operation='', log=None)
    Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message

setPos(newPos, operation='', log=None)
    Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message

setRGB(newRGB, operation='', log=None)
    DEPRECATED since v1.60.05: Please use the color attribute
**setSize** (*newSize, operation='', units=None, log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

**setText** (*text=None, log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

**setUseShaders** (*value=True, log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

**size**

The size (width, height) of the stimulus in the stimulus **units**

Value should be *x,y-pair, scalar* (applies to both dimensions) or None (resets to default). **Operations** are supported.

Sizes can be negative (causing a mirror-image reversal) and can extend beyond the window.

Example:

```python
stim.size = 0.8  # Set size to (xsize, ysize) = (0.8, 0.8)
print(stim.size)  # Outputs array([0.8, 0.8])
stim.size += (0.5, -0.5)  # make wider and flatter: (1.3, 0.3)
```

Tip: if you can see the actual pixel range this corresponds to by looking at **stim._sizeRendered**

**text**

The text to be rendered. Use `\n` to make new lines.

Issues: May be slow, and pyglet has a memory leak when setting text. For these reasons, this function checks so that it only updates the text if it has changed. So scripts can safely set the text on every frame, with no need to check if it has actually altered.

**units**

None, `norm`, `cm`, `deg`, `degFlat`, `degFlatPos`, or `pix`

If None then the current units of the Window will be used. See **Units for the window and stimuli** for explanation of other options.

Note that when you change units, you don’t change the stimulus parameters and it is likely to change appearance.

Example:

```python
# This stimulus is 20% wide and 50% tall with respect to window
stim = visual.PatchStim(win, units='norm', size=(0.2, 0.5))

# This stimulus is 0.2 degrees wide and 0.5 degrees tall.
stim.units = 'deg'
```

**useShaders**

Should shaders be used to render the stimulus (typically leave as **True**)

If the system support the use of OpenGL shader language then leaving this set to True is highly recommended. If shaders cannot be used then various operations will be slower (notably, changes to stimulus color or contrast)

**property verticesPix**

This determines the coordinates of the vertices for the current stimulus in pixels, accounting for size, ori, pos and units

**win**
The **Window object in which the** stimulus will be rendered by default. (required)

Example, drawing same stimulus in two different windows and display simultaneously. Assuming that you have two windows and a stimulus (win1, win2 and stim):

```python
stim.win = win1  # stimulus will be drawn in win1
stim.draw()     # stimulus is now drawn to win1
stim.win = win2  # stimulus will be drawn in win2
stim.draw()     # it is now drawn in win2
win1.flip(waitBlanking=False)  # do not wait for next
                               # monitor update
win2.flip()     # wait for vertical blanking.
```

Note that this just changes "default" window for stimulus.
You could also specify window-to-draw-to when drawing::

```python
stim.draw(win1)
stim.draw(win2)
```

**wrapWidth**

Int/float or None (set default). The width the text should run before wrapping.

8.2.35 **Window**

A class representing a window for displaying one or more stimuli.

```python
class psychopy.visual.Window(size=(800, 600), pos=None, color=(0, 0, 0), colorSpace='rgb',
                             rgb=None, dkl=None, lms=None, fullscr=None, allowGui=None,
                             monitor=None, bitsMode=None, winType=None, units=None,
                             gamma=None, blendMode='avg', screen=0, viewScale=None, viewPos=None,
                             viewOri=0.0, waitBlanking=True, allowStencil=False,
                             multiSample=False, numSamples=2, stereo=False, name='window1',
                             checkTiming=True, useFBO=False, useRetina=True, autoLog=True,
                             gammaErrorPolicy='raise', bpc=(8, 8, 8), depthBits=8, stencilBits=8,
                             *args, **kwargs)
```

Used to set up a context in which to draw objects, using either pyglet, pygame, or glfw.

The pyglet backend allows multiple windows to be created, allows the user to specify which screen to use (if more than one is available, duh!) and allows movies to be rendered.

The GLFW backend is a new addition which provides most of the same features as pyglet, but provides greater flexibility for complex display configurations.

Pygame may still work for you but it’s officially deprecated in this project (we won’t be fixing pygame-specific bugs).

These attributes can only be set at initialization. See further down for a list of attributes which can be changed after initialization of the Window, e.g. color, colorSpace, gamma etc.

**Parameters**

- **size** *(array-like of int)* – Size of the window in pixels [x, y].
- **pos** *(array-like of int)* – Location of the top-left corner of the window on the screen [x, y].
- **color** *(array-like of float)* – Color of background as [r, g, b] list or single value. Each gun can take values between -1.0 and 1.0.
• **fullscr** *(bool or None)* – Create a window in ‘full-screen’ mode. Better timing can be achieved in full-screen mode.

• **allowGUI** *(bool or None)* – If set to False, window will be drawn with no frame and no buttons to close etc., use None for value from preferences.

• **winType** *(str or None)* – Set the window type or back-end to use. If None then PsychoPy will revert to user/site preferences.

• **monitor** *(Monitor or None)* – The monitor to be used during the experiment. If None a default monitor profile will be used.

• **units** *(str or None)* – Defines the default units of stimuli drawn in the window (can be overridden by each stimulus). Values can be None, ‘height’ (of the window), ‘norm’ (normalised), ‘deg’, ‘cm’, ‘pix’. See Units for the window and stimuli for explanation of options.

• **screen** *(int)* – Specifies the physical screen that stimuli will appear on (‘pyglet’ and ‘glfw’ winType only). Values can be >0 if more than one screen is present.

• **viewScale** *(array-like of float or None)* – Scaling factors [x, y] to apply custom scaling to the current units of the Window instance.

• **viewPos** *(array-like of float or None)* – If not None, redefines the origin within the window, in the units of the window. Values outside the borders will be clamped to lie on the border.

• **viewOri** *(float)* – A single value determining the orientation of the view in degrees.

• **waitBlanking** *(bool or None)* – After a call to flip() should we wait for the blank before the script continues.

• **bitsMode** – DEPRECATED in 1.80.02. Use BitsSharp class from pycrsld instead.

• **checkTiming** *(bool)* – Whether to calculate frame duration on initialization. Estimated duration is saved in monitorFramePeriod.

• **allowStencil** *(bool)* – When set to True, this allows operations that use the OpenGL stencil buffer (notably, allowing the Aperture to be used).

• **multiSample** *(bool)* – If True and your graphics driver supports multisample buffers, multiple color samples will be taken per-pixel, providing an anti-aliased image through spatial filtering. This setting cannot be changed after opening a window. Only works with ‘pyglet’ and ‘glfw’ winTypes, and useFBO is False.

• **numSamples** *(int)* – A single value specifying the number of samples per pixel if multisample is enabled. The higher the number, the better the image quality, but can delay frame flipping. The largest number of samples is determined by GL_MAX_SAMPLES, usually 16 or 32 on newer hardware, will crash if number is invalid.

• **stereo** *(bool)* – If True and your graphics card supports quad buffers then this will be enabled. You can switch between left and right-eye scenes for drawing operations using setBuffer().

• **useRetina** *(bool)* – In PsychoPy >1.85.3 this should always be True as pyglet (or Apple) no longer allows us to create a non-retina display. NB when you use Retina display the initial win size request will be in the larger pixels but subsequent use of units='pix' should refer to the tiny Retina pixels. Window.size will give the actual size of the screen in Retina pixels.

• **gammaErrorPolicy** *(str)* – If raise, an error is raised if the gamma table is unable to be retrieved or set. If warn, a warning is raised instead. If ignore, neither an error nor a warning are raised.

• **bpc** *(array_like or int)* – Bits per color (BPC) for the back buffer as a tuple to specify bit depths for each color channel separately (red, green, blue), or a single value to set all of
them to the same value. Valid values depend on the output color depth of the display (screen) the window is set to use and the system graphics configuration. By default, it is assumed the display has 8-bits per color (8, 8, 8). Behaviour may be undefined for non-fullscreen windows, or if multiple screens are attached with varying color output depths.

- **depthBits** *(int)* – Back buffer depth bits. Default is 8, but can be set higher (eg. 24) if drawing 3D stimuli to minimize artifacts such a ‘Z-fighting’.
- **stencilBits** *(int)* – Back buffer stencil bits. Default is 8.

### Notes

- Some parameters (e.g. units) can now be given default values in the user/site preferences and these will be used if *None* is given here. If you do specify a value here it will take precedence over preferences.

**size**

Dimensions of the window’s drawing area/buffer in pixels \([w, h]\).

  - **Type** array-like *(float)*

**monitorFramePeriod**

Refresh rate of the display if checkTiming=True on window instantiation.

  - **Type** float

**_assignFlipTime**(obj, attrib)

Helper function to assign the time of last flip to the obj.attrib

  - **Parameters**
    - **obj** *(dict or object)* – A mutable object (usually a dict of class instance).
    - **attrib** *(str)* – Key or attribute of obj to assign the flip time to.

**_checkMatchingSizes**(requested, actual)

Checks whether the requested and actual screen sizes differ. If not then a warning is output and the window size is set to actual

**_endOfFlip**(clearBuffer)

Override end of flip with custom color channel masking if required.

**_getFrame**(rect=None, buffer='front')

Return the current Window as an image.

**_getRegionOfFrame**(rect=(-1, 1, 1, -1), buffer='front', power2=False, squarePower2=False)

Deprecated function, here for historical reasons. You may now use :py:attr:`~Window._getFrame()` and specify a rect to get a sub-region, just as used here.

  - power2 can be useful with older OpenGL versions to avoid interpolation in PatchStim. If power2 or squarePower2, it will expand rect dimensions up to next power of two. squarePower2 uses the max dimensions. You need to check what your hardware & OpenGL supports, and call _getRegionOfFrame() as appropriate.

**_renderFBO**()

Perform a warp operation.

(in this case a copy operation without any warping)

**_setCurrent**()

Make this window’s OpenGL context current.
If called on a window whose context is current, the function will return immediately. This reduces the number
of redundant calls if no context switch is required. If useFBO=True, the framebuffer is bound after the
context switch.

_setupGL()
Setup OpenGL state for this window.

_setupGamma(gammaVal)
A private method to work out how to handle gamma for this Window given that the user might have specified
an explicit value, or maybe gave a Monitor.

_startOfFlip()
Custom hardware classes may want to prevent flipping from occurring and can override this method as needed.
Return True to indicate hardware flip.

addEditable(editable)
Adds an editable element to the screen (something to which characters can be sent with meaning from the
keyboard).

The current editable object receiving chars is Window.currentEditable

Parameters editable –

Returns

property ambientLight
Ambient light color for the scene [r, g, b, a]. Values range from 0.0 to 1.0. Only applicable if useLights is True.

Examples
Setting the ambient light color:

```python
win.ambientLight = [0.5, 0.5, 0.5]
# don't do this!!!
win.ambientLight[0] = 0.5
win.ambientLight[1] = 0.5
win.ambientLight[2] = 0.5
```

applyEyeTransform(clearDepth=True)
Apply the current view and projection matrices.

Matrices specified by attributes viewMatrix and projectionMatrix are applied using ‘immediate
mode’ OpenGL functions. Subsequent drawing operations will be affected until flip() is called.

All transformations in GL_PROJECTION and GL_MODELVIEW matrix stacks will be cleared (set to identity)
prior to applying.

Parameters clearDepth(bool) – Clear the depth buffer. This may be required prior to ren-
dering 3D objects.
Examples

Using a custom view and projection matrix:

```python
# Must be called every frame since these values are reset after
# 'flip()' is called!
win.viewMatrix = viewtools.lookAt( ... )
win.projectionMatrix = viewtools.perspectiveProjectionMatrix( ... )
win.applyEyeTransform()
# draw 3D objects here ...
```

**property aspect**
Aspect ratio of the current viewport (width / height).

**blendMode**
Blend mode to use.

**callOnFlip(function, *args, **kwargs)**
Call a function immediately after the next `flip()` command.

The first argument should be the function to call, the following args should be used exactly as you would for your normal call to the function (can use ordered arguments or keyword arguments as normal).

E.g. If you have a function that you would normally call like this:

```python
pingMyDevice(portToPing, channel=2, level=0)
```

then you could call `callOnFlip()` to have the function call synchronized with the frame flip like this:

```python
win.callOnFlip(pingMyDevice, portToPing, channel=2, level=0)
```

**clearBuffer(color=True, depth=False, stencil=False)**
Clear the present buffer (to which you are currently drawing) without flipping the window.

Useful if you want to generate movie sequences from the back buffer without actually taking the time to flip the window.

Set `color` prior to clearing to set the color to clear the color buffer to. By default, the depth buffer is cleared to a value of 1.0.

**Parameters**

- **color**(bool) – Buffers to clear.
- **depth**(bool) – Buffers to clear.
- **stencil**(bool) – Buffers to clear.

**Examples**

Clear the color buffer to a specified color:

```python
win.color = (1, 0, 0)
win.clearBuffer(color=True)
```

Clear only the depth buffer, `depthMask` must be `True` or else this will have no effect. Depth mask is usually `True` by default, but may change:

```python
win.depthMask = True
win.clearBuffer(color=False, depth=True, stencil=False)
```
close()
Close the window (and reset the Bits++ if necessary).

color
Set the color of the window.
This command sets the color that the blank screen will have on the next clear operation. As a result it effectively takes TWO flip() operations to become visible (the first uses the color to create the new screen, the second presents that screen to the viewer). For this reason, if you want to change the background color of the window “on the fly”, it might be a better idea to draw a Rect that fills the whole window with the desired Rect.fillColor attribute. That’ll show up on first flip.
See other stimuli (e.g., GratingStim.color) for more info on the color attribute which essentially works the same on all PsychoPy stimuli.
See Color spaces for further information about the ways to specify colors and their various implications.

colorSpace
Documentation for colorSpace is in the stimuli.

For example, GratingStim.colorSpace
Usually used in conjunction with color like this:

```
win.colorSpace = 'rgb255'  # changes colorSpace but not
                          # the value of win.color
win.color = [0, 0, 255]   # clear blue in rgb255
```

See Color spaces for further information about the ways to specify colors and their various implications.

property convergeOffset
Convergence offset from monitor in centimeters.
This is value corresponds to the offset from screen plane to set the convergence plane (or point for toe-in projections). Positive offsets move the plane farther away from the viewer, while negative offsets nearer. This value is used by setPerspectiveView and should be set before calling it to take effect.

Notes
• This value is only applicable for setToeIn and setOffAxisView.

coordToRay(screenXY)
Convert a screen coordinate to a direction vector.
Takes a screen/window coordinate and computes a vector which projects a ray from the viewpoint through it (line-of-sight). Any 3D point touching the ray will appear at the screen coordinate.
Uses the current viewport and projectionMatrix to calculate the vector. The vector is in eye-space, where the origin of the scene is centered at the viewpoint and the forward direction aligned with the -Z axis. A ray of (0, 0, -1) results from a point at the very center of the screen assuming symmetric frustums.
Note that if you are using a flipped/mirrored view, you must invert your supplied screen coordinates (screenXY) prior to passing them to this function.

Parameters screenXY (array_like) – X, Y screen coordinate. Must be in units of the window.

Returns Normalized direction vector [x, y, z].

Return type ndarray
Examples

Getting the direction vector between the mouse cursor and the eye:

```python
mx, my = mouse.getPos()
dir = win coordToRay((mx, my))
```

Set the position of a 3D stimulus object using the mouse, constrained to a plane. The object origin will always be at the screen coordinate of the mouse cursor:

```python
# the eye position in the scene is defined by a rigid body pose
win.viewMatrix = camera.getViewMatrix()
win.applyEyeTransform()

# get the mouse location and calculate the intercept
mx, my = mouse.getPos()
ray = win.coordToRay([mx, my])
result = intersectRayPlane(  # from mathtools
    orig=camera.pos,
    dir=normal.transformNormal(ray),
    planeOrig=(0, 0, -10),
    planeNormal=(0, 1, 0))

# if result is 'None', there is no intercept
if result is not None:
pos, dist = result
    objModel.thePose.pos = pos
else:
    objModel.thePose.pos = (0, 0, -10)  # plane origin
```

If you don’t define the position of the viewer with a `RigidBodyPose`, you can obtain the appropriate eye position and rotate the ray by doing the following:

```python
pos = numpy.linalg.inv(win.viewMatrix)[3, 3]
ray = win.coordToRay([mx, my]).dot(win.viewMatrix[3, :3])
# then ...
result = intersectRayPlane(  # from mathtools
    orig=pos,
    dir=ray,
    planeOrig=(0, 0, -10),
    planeNormal=(0, 1, 0))
```

**property cullFace**

*True* if face culling is enabled.

**property cullFaceMode**

Face culling mode, either *back*, *front* or *both*.

**property currentEditable**

The editable (Text?) object that currently has key focus

**property depthFunc**

Depth test comparison function for rendering.

**property depthMask**

*True* if depth masking is enabled. Writing to the depth buffer will be disabled.

**property depthTest**

*True* if depth testing is enabled.
**property draw3d**

*True* if 3D drawing is enabled on this window.

**property eyeOffset**

Eye offset in centimeters.

This value is used by `setPerspectiveView` to apply a lateral offset to the view, therefore it must be set prior to calling it. Use a positive offset for the right eye, and a negative one for the left. Offsets should be the distance to from the middle of the face to the center of the eye, or half the inter-ocular distance.

**property farClip**

Distance to the far clipping plane in meters.

**flip** *(clearBuffer=True)*

Flip the front and back buffers after drawing everything for your frame. (This replaces the `update()` method, better reflecting what is happening underneath).

- **Parameters**
  - `clearBuffer` *(bool, optional)* – Clear the draw buffer after flipping. Default is `True`.
  - **Returns** Wall-clock time in seconds the flip completed. Returns *None* if `waitBlanking` is `False`.
  - **Return type** *float* or *None*

**Notes**

- The time returned when `waitBlanking` is `True` corresponds to when the graphics driver releases the draw buffer to accept draw commands again. This time is usually close to the vertical sync signal of the display.

**Examples**

Results in a clear screen after flipping:

```python
win.flip(clearBuffer=True)
```

The screen is not cleared (so represent the previous screen):

```python
win.flip(clearBuffer=False)
```

**fps**

Report the frames per second since the last call to this function (or since the window was created if this is first call)

**property framebufferSize**

Size of the framebuffer in pixels *(w, h)*.

**property frontFace**

Face winding order to define front, either *ccw* or *cw*.

**fullscr**

Set whether fullscreen mode is *True* or *False* (not all backends can toggle an open window).

**gamma**

Set the monitor gamma for linearization.
Warning: Don’t use this if using a Bits++ or Bits#, as it overrides monitor settings.

**gammaRamp**
Sets the hardware CLUT using a specified 3xN array of floats ranging between 0.0 and 1.0.
Array must have a number of rows equal to $2^\text{max(bpc)}$.

**getActualFrameRate** *(nIdentical=10, nMaxFrames=100, nWarmUpFrames=10, threshold=1)*
Measures the actual frames-per-second (FPS) for the screen.
This is done by waiting (for a max of nMaxFrames) until nIdentical frames in a row have identical frame times (std dev below threshold ms).

Parameters

- **nIdentical** *(int, optional)* – The number of consecutive frames that will be evaluated. Higher \(\rightarrow\) greater precision. Lower \(\rightarrow\) faster.
- **nMaxFrames** *(int, optional)* – The maximum number of frames to wait for a matching set of nIdentical.
- **nWarmUpFrames** *(int, optional)* – The number of frames to display before starting the test (this is in place to allow the system to settle after opening the Window for the first time.
- **threshold** *(int, optional)* – The threshold for the std deviation (in ms) before the set are considered a match.

Returns Frame rate (FPS) in seconds. If there is no such sequence of identical frames a warning is logged and None will be returned.

Return type float or None

**getFutureFlipTime** *(targetTime=0, clock=None)*
The expected time of the next screen refresh. This is currently calculated as win._lastFrameTime + refresh-Interval

Parameters

- **targetTime** *(float)* – The delay from now for which you want the flip time. 0 will give the because that the earliest we can achieve. 0.15 will give the schedule flip time that gets as close to 150 ms as possible
- **clock** *(None, 'ptb', 'now' or any Clock object)* – If True then the time returned is compatible with ptb.GetSecs()
- **verbose** *(bool)* – Set to True to view the calculations along the way

**getMovieFrame** *(buffer='front')*
Capture the current Window as an image.
Saves to stack for saveMovieFrames(). As of v1.81.00 this also returns the frame as a PIL image
This can be done at any time (usually after a flip() command).
Frames are stored in memory until a saveMovieFrames() command is issued. You can issue get-MovieFrame() as often as you like and then save them all in one go when finished.
The back buffer will return the frame that hasn’t yet been ‘flipped’ to be visible on screen but has the advantage that the mouse and any other overlapping windows won’t get in the way.
The default front buffer is to be called immediately after a flip() and gives a complete copy of the screen at the window’s coordinates.
Parameters **buffer** *(str, optional)* – Buffer to capture.

Returns  Buffer pixel contents as a PIL/Pillow image object.

Return type  Image

**getMsPerFrame** *(nFrames=60, showVisual=False, msg='', msDelay=0.0)*
Assesses the monitor refresh rate (average, median, SD) under current conditions, over at least 60 frames.

Records time for each refresh (frame) for n frames (at least 60), while displaying an optional visual. The visual is just eye-candy to show that something is happening when assessing many frames. You can also give it text to display instead of a visual, e.g., `msg='(testing refresh rate...)';`: setting `msg` implies `showVisual == False`.

To simulate refresh rate under cpu load, you can specify a time to wait within the loop prior to doing the `flip()`. If `0 < msDelay < 100`, wait for that long in ms.

Returns timing stats (in ms) of:

• average time per frame, for all frames
• standard deviation of all frames
• median, as the average of 12 frame times around the median (~monitor refresh rate)

Author

• 2010 written by Jeremy Gray

**property lights**

Scene lights.

This is specified as an array of `~psychopy.visual.LightSource` objects. If a single value is given, it will be converted to a list before setting. Set `useLights` to `True` before rendering to enable lighting/shading on subsequent objects. If `lights` is `None` or an empty list, no lights will be enabled if `useLights=True`, however, the scene ambient light set with `ambientLight` will be still be used.

**Examples**

Create a directional light source and add it to scene lights:

```python
dirLight = gltools.LightSource((0., 1., 0.), lightType='directional')
win.lights = dirLight  # `win.lights` will be a list when accessed!
```

Multiple lights can be specified by passing values as a list:

```python
myLights = [gltools.LightSource((0., 5., 0.)),
            gltools.LightSource((-2., -2., 0.))]
win.lights = myLights
```

**logOnFlip** *(msg, level, obj=None)*

Send a log message that should be time-stamped at the next `flip()` command.

Parameters

• **msg** *(str)* – The message to be logged.
• **level** *(int)* – The level of importance for the message.
• **obj** *(object, optional)* – The python object that might be associated with this message if desired.
mouseVisible
Sets the visibility of the mouse cursor.

If Window was initialized with allowGUI=False then the mouse is initially set to invisible, otherwise it will initially be visible.

Usage:

```
win.mouseVisible = False
win.mouseVisible = True
```

property nearClip
Distance to the near clipping plane in meters.

nextEditable(chars='')
Moves focus of the cursor to the next editable window

property projectionMatrix
Projection matrix defined as a 4x4 numpy array.

recordFrameIntervals
Record time elapsed per frame.

Provides accurate measures of frame intervals to determine whether frames are being dropped. The intervals are the times between calls to flip(). Set to True only during the time-critical parts of the script. Set this to False while the screen is not being updated, i.e., during any slow, non-frame-time-critical sections of your code, including inter-trial-intervals, event.waitkeys(), core.wait(), or image.setImage().

Examples

Enable frame interval recording, successive frame intervals will be stored:

```
win.recordFrameIntervals = True
```

Frame intervals can be saved by calling the saveFrameIntervals method:

```
win.saveFrameIntervals()
```

removeEditable(editable)
resetEyeTransform(clearDepth=True)

Restore the default projection and view settings to PsychoPy defaults. Call this prior to drawing 2D stimuli objects (i.e. GratingStim, ImageStim, Rect, etc.) if any eye transformations were applied for the stimuli to be drawn correctly.

Parameters clearDepth (bool) – Clear the depth buffer upon reset. This ensures successive draw commands are not affected by previous data written to the depth buffer. Default is True.
Notes

- Calling `flip()` automatically resets the view and projection to defaults. So you don’t need to call this unless you are mixing 3D and 2D stimuli.

Examples

Going between 3D and 2D stimuli:

```python
# 2D stimuli can be drawn before setting a perspective projection
win.setPerspectiveView()
# draw 3D stimuli here ...
win.resetEyeTransform()
# 2D stimuli can be drawn here again ...
win.flip()
```

`resetViewport()`

Reset the viewport to cover the whole framebuffer.

Set the viewport to match the dimensions of the back buffer or framebuffer (if `useFBO=True`). The scissor rectangle is also set to match the dimensions of the viewport.

`saveFrameIntervals(fileName=None, clear=True)`

Save recorded screen frame intervals to disk, as comma-separated values.

**Parameters**

- `fileName` (*None* or *str*) — *None* or the filename (including path if necessary) in which to store the data. If None then ‘lastFrameIntervals.log’ will be used.

- `clear` (*bool*) — Clear buffer frames intervals were stored after saving. Default is *True*.

`saveMovieFrames(fileName, codec='libx264', fps=30, clearFrames=True)`

 Writes any captured frames to disk.

Will write any format that is understood by PIL (tif, jpg, png, …)

**Parameters**

- `filename` (*str*) — Name of file, including path. The extension at the end of the file determines the type of file(s) created. If an image type (e.g. .png) is given, then multiple static frames are created. If it is .gif then an animated GIF image is created (although you will get higher quality GIF by saving PNG files and then combining them in dedicated image manipulation software, such as GIMP). On Windows and Linux .mpeg files can be created if `pymedia` is installed. On macOS .mov files can be created if the pyobjc-frameworks-QTKit is installed. Unfortunately the libs used for movie generation can be flaky and poor quality. As for animated GIFs, better results can be achieved by saving as individual .png frames and then combining them into a movie using software like ffmpeg.

- `codec` (*str*, *optional*) — The codec to be used by moviepy for mp4/mpg/mov files. If *None* then the default will depend on file extension. Can be one of `libx264`, `mpeg4` for mp4/mov files. Can be `rawvideo`, `png` for avi files (not recommended). Can be `libvorbis` for ogv files. Default is `libx264`.

- `fps` (*int*, *optional*) — The frame rate to be used throughout the movie. **Only for quicktime (.mov) movies.** Default is 30.

- `clearFrames` (*bool*, *optional*) — Set this to *False* if you want the frames to be kept for additional calls to `saveMovieFrames`. Default is *True*. 


**Examples**

Writes a series of static frames as frame001.tif, frame002.tif etc.:

```python
myWin.saveMovieFrames('frame.tif')
```

As of PsychoPy 1.84.1 the following are written with moviepy:

```python
myWin.saveMovieFrames('stimuli.mp4')  # codec = 'libx264' or 'mpeg4'
myWin.saveMovieFrames('stimuli.mov')
myWin.saveMovieFrames('stimuli.gif')
```

**property scissor**

Scissor rectangle (x, y, w, h) for the current draw buffer.

Values x and y define the origin, and w and h the size of the rectangle in pixels. The scissor operation is only active if `scissorTest=True`.

Usually, the scissor and viewport are set to the same rectangle to prevent drawing operations from spilling into other regions of the screen. For instance, calling `clearBuffer` will only clear within the scissor rectangle.

Setting the scissor rectangle but not the viewport will restrict drawing within the defined region (like a rect-angular aperture), not changing the positions of stimuli.

**property scissorTest**

`True` if scissor testing is enabled.

**setBuffer** *(buffer, clear=True)*

Choose which buffer to draw to (‘left’ or ‘right’).

Requires the Window to be initialised with `stereo=True` and requires a graphics card that supports quad buffering (e.g., NVIDIA Quadro series).

PsychoPy always draws to the back buffers, so ‘left’ will use GL_BACK_LEFT. This then needs to be flipped once both eye’s buffers have been rendered.

**Parameters**

- **buffer** *(str)* – Buffer to draw to. Can either be ‘left’ or ‘right’.
- **clear** *(bool, optional)* – Clear the buffer before drawing. Default is `True`.

**Examples**

Stereoscopic rendering example using quad-buffers:

```python
win = visual.Window(..., stereo=True)
while True:
    # clear may not actually be needed
    win.setBuffer('left', clear=True)
    # do drawing for left eye
    win.setBuffer('right', clear=True)
    # do drawing for right eye
    win.flip()
```

**setMouseType** *(name='arrow')*

Change the appearance of the cursor for this window. Cursor types provide contextual hints about how to interact with on-screen objects.
The graphics used ‘standard cursors’ provided by the operating system. They may vary in appearance and hot spot location across platforms. The following names are valid on most platforms:

- **arrow**: Default pointer.
- **ibeam**: Indicates text can be edited.
- **crosshair**: Crosshair with hot-spot at center.
- **hand**: A pointing hand.
- **hresize**: Double arrows pointing horizontally.
- **vresize**: Double arrows pointing vertically.

**Parameters**  
name (str) – Type of standard cursor to use (see above). Default is arrow.

**Notes**

- On Windows the **crosshair** option is negated with the background color. It will not be visible when placed over 50% grey fields.

```python
setOffAxisView(applyTransform=True, clearDepth=True)
```

Set an off-axis projection.

Create an off-axis projection for subsequent rendering calls. Sets the viewMatrix and projectionMatrix accordingly so the scene origin is on the screen plane. If eyeOffset is correct and the view distance and screen size is defined in the monitor configuration, the resulting view will approximate ortho-stereo viewing.

The convergence plane can be adjusted by setting convergeOffset. By default, the convergence plane is set to the screen plane. Any points on the screen plane will have zero disparity.

**Parameters**

- **applyTransform** (bool) – Apply transformations after computing them in immediate mode. Same as calling applyEyeTransform() afterwards.
- **clearDepth** (bool, optional) – Clear the depth buffer.

```python
setPerspectiveView(applyTransform=True, clearDepth=True)
```

Set the projection and view matrix to render with perspective.

Matrices are computed using values specified in the monitor configuration with the scene origin on the screen plane. Calculations assume units are in meters. If eyeOffset != 0, the view will be transformed laterally, however the frustum shape will remain the same.

Note that the values of projectionMatrix and viewMatrix will be replaced when calling this function.

**Parameters**

- **applyTransform** (bool) – Apply transformations after computing them in immediate mode. Same as calling applyEyeTransform() afterwards if False.
- **clearDepth** (bool, optional) – Clear the depth buffer.

```python
setToeInView(applyTransform=True, clearDepth=True)
```

Set toe-in projection.

Create a toe-in projection for subsequent rendering calls. Sets the viewMatrix and projectionMatrix accordingly so the scene origin is on the screen plane. The value of convergeOffset will define the convergence point of the view, which is offset perpendicular to the center of the screen plane. Points falling on a vertical line at the convergence point will have zero disparity.

---

280 Chapter 8. Reference Manual (API)
Parameters

- **applyTransform** *(bool)* – Apply transformations after computing them in immediate mode. Same as calling `applyEyeTransform()` afterwards.
- **clearDepth** *(bool, optional)* – Clear the depth buffer.

Notes

- This projection mode is only ‘correct’ if the viewer’s eyes are converged at the convergence point. Due to perspective, this projection introduces vertical disparities which increase in magnitude with eccentricity. Use `setOffAxisView` if you want to display something the viewer can look around the screen comfortably.

**property size**
Size of the drawable area in pixels (w, h).

**property stencilTest**
True if stencil testing is enabled.

**timeOnFlip** *(obj, attrib)*
Retrieves the time on the next flip and assigns it to the `attrib` for this `obj`.

Parameters

- **obj** *(dict or object)* – A mutable object (usually a dict of class instance).
- **attrib** *(str)* – Key or attribute of `obj` to assign the flip time to.

Examples

Assign time on flip to the `tStartRefresh` key of `myTimingDict`:

```python
win.getTimeOnFlip(myTimingDict, 'tStartRefresh')
```

**units**

*None*, ‘height’ (of the window), ‘norm’, ‘deg’, ‘cm’, ‘pix’ Defines the default units of stimuli initialized in the window. I.e. if you change units, already initialized stimuli won’t change their units.

Can be overridden by each stimulus, if `units` is specified on initialization.

See *Units for the window and stimuli* for explanation of options.

**updateLights** *(index=None)*
Explicitly update scene lights if they were modified.

This is required if modifications to objects referenced in `lights` have been changed since assignment. If you removed or added items of `lights` you must refresh all of them.

Parameters **index** *(int, optional)* – Index of light source in `lights` to update. If `None`, all lights will be refreshed.
Examples

Call `updateLights` if you modified lights directly like this:

```python
win.lights[1].diffuseColor = [1., 0., 0.]
win.updateLights(1)
```

**property useLights**

Enable scene lighting.

Lights will be enabled if using legacy OpenGL lighting. Stimuli using shaders for lighting should check if `useLights` is `True` since this will have no effect on them, and disable or use a no lighting shader instead. Lights will be transformed to the current view matrix upon setting to `True`.

Lights are transformed by the present `GL_MODELVIEW` matrix. Setting `useLights` will result in their positions being transformed by it. If you want lights to appear at the specified positions in world space, make sure the current matrix defines the view/eye transformation when setting `useLights=True`.

This flag is reset to `False` at the beginning of each frame. Should be `False` if rendering 2D stimuli or else the colors will be incorrect.

**property viewMatrix**

View matrix defined as a 4x4 numpy array.

**viewPos**

The origin of the window onto which stimulus-objects are drawn.

The value should be given in the units defined for the window. NB: Never change a single component (x or y) of the origin, instead replace the `viewPos`-attribute in one shot, e.g.:

```python
win.viewPos = [new_xval, new_yval]  # This is the way to do it
win.viewPos[0] = new_xval  # DO NOT DO THIS! Errors will result.
```

**property viewport**

Viewport rectangle `(x, y, w, h)` for the current draw buffer.

Values `x` and `y` define the origin, and `w` and `h` the size of the rectangle in pixels.

This is typically set to cover the whole buffer, however it can be changed for applications like multi-view rendering. Stimuli will draw according to the new shape of the viewport, for instance and stimulus with position `(0, 0)` will be drawn at the center of the viewport, not the window.

Examples

Constrain drawing to the left and right halves of the screen, where stimuli will be drawn centered on the new rectangle. Note that you need to set both the `viewport` and the `scissor` rectangle:

```python
x, y, w, h = win.frameBufferSize  # size of the framebuffer
win.viewport = win.scissor = [x, y, w / 2.0, h]
# draw left stimuli ...

win.viewport = win.scissor = [x + (w / 2.0), y, w / 2.0, h]
# draw right stimuli ...

# restore drawing to the whole screen
win.viewport = win.scissor = [x, y, w, h]
```

**waitBlanking**

After a call to `flip()` should we wait for the blank before the script continues.
8.2.36 `psychopy.visual.windowframepack` - Pack multiple monochrome images into RGB frame

Copyright (C) 2014 Allen Institute for Brain Science

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License Version 3 as published by the Free Software Foundation on 29 June 2007. This program is distributed WITHOUT WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE OR ANY OTHER WARRANTY, EXPRESSED OR IMPLIED. See the GNU General Public License Version 3 for more details. You should have received a copy of the GNU General Public License along with this program. If not, see http://www.gnu.org/licenses/

**ProjectorFramePacker**

class `psychopy.visual.windowframepack.ProjectorFramePacker` *(win)*

Class which packs 3 monochrome images per RGB frame.

Allowing 180Hz stimuli with DLP projectors such as TI LightCrafter 4500.

The class overrides methods of the visual.Window class to pack a monochrome image into each RGB channel. PsychoPy is running at 180Hz. The display device is running at 60Hz. The output projector is producing images at 180Hz.

Frame packing can work with any projector which can operate in 'structured light mode' where each RGB channel is presented sequentially as a monochrome image. Most home and office projectors cannot operate in this mode, but projectors designed for machine vision applications typically will offer this feature.

Example usage to use ProjectorFramePacker:

```python
from psychopy.visual.windowframepack import ProjectorFramePacker
win = Window(monitor='testMonitor', screen=1, fullscr=True, useFBO = True)
framePacker = ProjectorFramePacker (win)
```

**Parameters**

- `win` : Handle to the window.

**endOfFlip** *(clearBuffer)*

Mask RGB cyclically after each flip. We ignore clearBuffer and just auto-clear after each hardware flip.

**startOfFlip** ()

Return True if all channels of the RGB frame have been filled with monochrome images, and the associated window should perform a hardware flip.

8.2.37 `psychopy.visual.windowwarp` - warping to spherical, cylindrical, or other projections

Copyright (C) 2014 Allen Institute for Brain Science

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License Version 3 as published by the Free Software Foundation on 29 June 2007. This program is distributed WITHOUT WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE OR ANY OTHER WARRANTY, EXPRESSED OR IMPLIED. See the GNU General Public License Version 3 for more details. You should have received a copy of the GNU General Public License along with this program. If not, see http://www.gnu.org/licenses/
Warper

class psychopy.visual.windowwarp.Warper(win, warp=None, warpfile=None, warpGriddsize=300, eyepoint=(0.5, 0.5), flipHorizontal=False, flipVertical=False)

Class to perform warps.

Supports spherical, cylindrical, warpfile, or None (disabled) warps

Warping is a final operation which can be optionally performed on each frame just before transmission to the display. It is useful for perspective correction when the eye to monitor distance is small (say, under 50 cm), or when projecting to domes or other non-planar surfaces.

These attributes define the projection and can be altered dynamically using the changeProjection() method.

Parameters

win : Handle to the window.

warp ['spherical', 'cylindrical, 'warpfile' or None] This table gives the main properties of each projection

<table>
<thead>
<tr>
<th>Warp</th>
<th>eyepoint modifies warp</th>
<th>verticals parallel</th>
<th>horizontals parallel</th>
<th>perspective correct</th>
</tr>
</thead>
<tbody>
<tr>
<td>spherical</td>
<td>y</td>
<td>n</td>
<td>n</td>
<td>y</td>
</tr>
<tr>
<td>cylindrical</td>
<td>y</td>
<td>y</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>warpfile</td>
<td>n</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>None</td>
<td>n</td>
<td>y</td>
<td>y</td>
<td>n</td>
</tr>
</tbody>
</table>

warpfile [None or filename containing Blender and Paul Bourke] compatible warp definition. (see http://paulbourke.net/dome/warpingfisheye/)

warpGriddsize [300] Defines the resolution of the warp in both X and Y when not using a warpfile. Typical values would be 64-300 trading off tolerance for jaggies for speed.

eyepoint [(0.5, 0.5] center of the screen] Position of the eye in X and Y as a fraction of the normalized screen width and height. [0,0] is the bottom left of the screen. [1,1] is the top right of the screen.

flipHorizontal: True or False Flip the entire output horizontally. Useful for back projection scenarios.

flipVertical: True or False Flip the entire output vertically. Useful if projector is flipped upside down.

Notes

1) The eye distance from the screen is initialized from the monitor definition.

2) The eye distance can be altered dynamically by changing ‘warper.dist_cm’ and then calling changeProjection().

Example usage to create a spherical projection:

```python
from psychopy.visual.windowwarp import Warper
win = Window(monitor='testMonitor', screen=1,
              fullscr=True, useFBO = True)
warper = Warper(win,
               warp='spherical',
               warpfile = ''
               )
```

(continues on next page)
warpGridsize = 128,
eyepoint = [0.5, 0.5],
flipHorizontal = False,
flipVertical = False)

changeProjection(warp, warpfile=None, eyepoint=(0.5, 0.5), flipHorizontal=False, flipVertical=False)

Allows changing the warp method on the fly. Uses the same parameter definitions as constructor.

Window to display all stimuli below.

Windows and display devices:
- **Window** is the main class to display objects
- **Warper** for non-flat projection screens
- **ProjectorFramePacker** for handling displays with ‘structured light mode’ to achieve high framerates
- **Rift** for Oculus Rift support (Windows 64bit only)

Commonly used:
- **ImageStim** to show images
- **TextStim** to show text
- **TextBox** rewrite of TextStim (faster/better but only monospace fonts)

Shapes (all special classes of **ShapeStim**):
- **ShapeStim** to draw shapes with arbitrary numbers of vertices
- **Rect** to show rectangles
- **Circle** to show circles
- **Polygon** to show polygons
- **Line** to show a line

Images and patterns:
- **ImageStim** to show images
- **SimpleImageStim** to show images without bells and whistles
- **GratingStim** to show gratings
- **RadialStim** to show annulus, a rotating wedge, a checkerboard etc
- **NoiseStim** to show filtered noise patterns of various forms
- **EnvelopeGrating** to generate second-order stimuli (gratings that can have a carrier and envelope)

Multiple stimuli:
- **ElementArrayStim** to show many stimuli of the same type
- **DotStim** to show and control movement of dots

3D shapes, materials, and lighting:
- **LightSource** to define a light source in a scene
- **SceneSkybox** to render a background skybox for VR and 3D scenes
- **BlinnPhongMaterial** to specify a material using the Blinn-Phong lighting model
PsychoPy - Psychology software for Python, Release 2020.2.10

- **RigidBodyPose** to define poses of objects in 3D space
- **BoundingBox** to define bounding boxes around 3D objects
- **SphereStim** to show a 3D sphere
- **BoxStim** to show 3D boxes and cubes
- **PlaneStim** to show 3D plane
- **ObjMeshStim** to show Wavefront OBJ meshes loaded from files

Other stimuli:
- **MovieStim** to show movies
- **Slider** a new improved class to collect ratings
- **RatingScale** to collect ratings
- **CustomMouse** to change the cursor in windows with GUI. OBS: will be deprecated soon

Meta stimuli (stimuli that operate on other stimuli):
- **BufferImageStim** to make a faster-to-show “screenshot” of other stimuli
- **Aperture** to restrict visibility area of other stimuli

Helper functions:
- **filters** for creating grating textures and Gaussian masks etc.
- **visualhelperfunctions** for tests about whether one stimulus contains another
- **unittools** to convert deg<->radians
- **monitorunittools** to convert cm<->pix<->deg etc.
- **colorspacetools** to convert between supported color spaces
- **viewtools** to work with view projections
- **mathtools** to work with vectors, quaternions, and matrices

### 8.3 psychoPy.clock - Clocks and timers

Created on Tue Apr 23 11:28:32 2013

Provides the high resolution timebase used by psychopy, and defines some time related utility Classes.

Moved functionality from core.py so a common code base could be used in core.py and logging.py; vs. duplicating the getTime and Clock logic.

@author: Sol @author: Jon

**class** psychoPy.clock.Clock

A convenient class to keep track of time in your experiments. You can have as many independent clocks as you like (e.g. one to time responses, one to keep track of stimuli...)

This clock is identical to the **MonotonicClock** except that it can also be reset to 0 or another value at any point.

**add(t)**

Add more time to the clock’s ‘start’ time (t0).
Note that, by adding time to \( t_0 \), you make the current time appear less. Can have the effect that \( \text{getTime()} \) returns a negative number that will gradually count back up to zero.

e.g.:

```
timer = core.Clock()
timer.add(5)
while timer.getTime() < 0:
    # do something
```

**reset (new\( T = 0.0 \))**

Reset the time on the clock. With no args time will be set to zero. If a float is received this will be the new time on the clock.

**class psychoPy.clock.CountdownTimer (start=0)**

Similar to a **Clock** except that time counts down from the time of last reset

Typical usage:

```
timer = core.CountdownTimer(5)
while timer.getTime() > 0:  # after 5s will become negative
    # do stuff
```

**getTime ()**

Returns the current time left on this timer in secs (sub-ms precision)

**reset (t=None)**

Reset the time on the clock. With no args, time will be set to the time used for last reset (or start time if no previous resets). If a float is received, this will be the new time on the clock.

**class psychoPy.clock.MonotonicClock (start_time=None)**

A convenient class to keep track of time in your experiments using a sub-millisecond timer.

Unlike the **Clock** this cannot be reset to arbitrary times. For this clock \( t=0 \) always represents the time that the clock was created.

Don’t confuse this **class** with **core.monotonicClock** which is an **instance** of it that got created when PsychoPy.core was imported. That clock instance is deliberately designed always to return the time since the start of the study.

Version Notes: This class was added in PsychoPy 1.77.00

**getLastResetTime ()**

Returns the current offset being applied to the high resolution timebase used by Clock.

**getTime (applyZero=True)**

Returns the current time on this clock in secs (sub-ms precision).

If applying zero then this will be the time since the clock was created (typically the beginning of the script).

If not applying zero then it is whatever the underlying clock uses as its base time but that is system dependent. e.g. can be time since reboot, time since Unix Epoch etc

**class psychoPy.clock.StaticPeriod (screenHz=None, win=None, name='StaticPeriod')**

A class to help insert a timing period that includes code to be run.

Typical usage:

```
fixation.draw()
win.flip()
ISI = StaticPeriod(screenHz=60)
ISI.start(0.5)  # start a period of 0.5s
stim.image = 'largeFile.bmp'  # could take some time
```

(continues on next page)
ISI.complete()  # finish the 0.5s, taking into account one 60Hz frame
stim.draw()
win.flip()  # the period takes into account the next frame flip
# time should now be at exactly 0.5s later than when ISI.start()
# was called

Parameters

- **screenHz** – the frame rate of the monitor (leave as None if you don’t want this accounted for)
- **win** – if a visual.Window is given then StaticPeriod will also pause/restart frame interval recording
- **name** – give this StaticPeriod a name for more informative logging messages

**complete()**
Completes the period, using up whatever time is remaining with a call to wait()

**Returns** 1 for success, 0 for fail (the period overran)

**start(duration)**
Start the period. If this is called a second time, the timer will be reset and starts again

**Parameters** **duration** – The duration of the period, in seconds.

`psychopy.clock.getAbsTime()`
Return unix time (i.e., whole seconds elapsed since Jan 1, 1970).
This uses the same clock-base as the other timing features, like `getTime()`. The time (in seconds) ignores the time-zone (like `time.time()` on linux). To take the timezone into account, use `int(time.mktime(time.gmtime()))`.

Absolute times in seconds are especially useful to add to generated file names for being unique, informative (= a meaningful time stamp), and because the resulting files will always sort as expected when sorted in chronological, alphabetical, or numerical order, regardless of locale and so on.

**Version Notes**: This method was added in PsychoPy 1.77.00

`psychopy.clock.getTime()`
Copyright (c) 2018 Mario Kleiner. Licensed under MIT license.

For detailed help on a subfunction **SUBFUNCTIONNAME**, type `GetSecs('SUBFUNCTIONNAME?')` i.e. the name with a question mark appended. E.g., for detailed help on the subfunction called Version, type this: `GetSecs('Version?')`

`[GetSecsTime, WallTime, syncErrorSecs] = GetSecs('AllClocks' [, maxError=0.000020]);`

`psychopy.clock.wait(secs, hogCPUperiod=0.2)`
Wait for a given time period.
If secs=10 and hogCPU=0.2 then for 9.8s python’s time.sleep function will be used, which is not especially precise, but allows the cpu to perform housekeeping. In the final hogCPUperiod the more precise method of constantly polling the clock is used for greater precision.
If you want to obtain key-presses during the wait, be sure to use pyglet and to hogCPU for the entire time, and then call `psychopy.event.getKeys()` after calling `wait()`
If you want to suppress checking for pyglet events during the wait, do this once:
and from then on you can do:

```python
core.wait(sec)
```

This will preserve terminal-window focus during command line usage.

## 8.4 psychopy.hardware - hardware interfaces

PsychoPy can access a wide range of external hardware. For some devices the interface has already been created in the following sub-packages of PsychoPy. For others you may need to write the code to access the serial port etc. manually.

Contents:

### 8.4.1 Keyboard

To handle input from keyboard (supercedes event.getKeys)

The Keyboard class was new in PsychoPy 3.1 and replaces the older `event.getKeys()` calls.

**Psychtoolbox versus event.getKeys**

On 64 bits Python3 installations it provides access to the Psychotoolbox `kbQueue` series of functions using the same compiled C code (available in python-psychtoolbox library).

On 32 bit installations and Python2 it reverts to the older `psychopy.event.getKeys()` calls.

The new calls have several advantages:

- the polling is performed and timestamped asynchronously with the main thread so that times relate to when the key was pressed, not when the call was made
- the polling is direct to the USB HID library in C, which is faster than waiting for the operating system to poll and interpret those same packets
- we also detect the KeyUp events and therefore provide the option of returning keypress duration
- on Linux and Mac you can also distinguish between different keyboard devices (see `getKeyboards()`)

This library makes use, where possible of the same low-level asynchronous hardware polling as in PsychoToolbox.

Example usage

```python
from psychopy.hardware import keyboard
from psychopy import core

kb = keyboard.Keyboard()

# during your trial
kb.clock.reset()  # when you want to start the timer from
keys = kb.getKeys(['right', 'left', 'quit'], waitRelease=True)
if 'quit' in keys:
    core.quit()
```

(continues on next page)
for key in keys:
    print(key.name, key.rt, key.duration)

Classes and functions

class psychopy.hardware.keyboard.Keyboard (device=-1, bufferSize=10000, waitForStart=False, clock=None)

The Keyboard class provides access to the Psychtoolbox KbQueue-based calls on Python3 64-bit with fall-back to event.getKeys on legacysystems.

Create the device (default keyboard or select one)

Parameters

• device(int or dict) – On Linux/Mac this can be a device index or a dict containing the device info (as from getKeyboards()) or -1 for all devices acting as a unified Keyboard
• bufferSize(int) – How many keys to store in the buffer (before dropping older ones)
• waitForStart(bool (default False)) – Normally we’ll start polling the Keyboard at all times but you could choose not to do that and start/stop manually instead by setting this to True

getKeys (keyList=None, waitRelease=True, clear=True)

Parameters

• keyList(list (or other iterable)) – The keys that you want to listen out for. e.g. ['left', 'right', 'q']
• waitRelease(bool (default True)) – If True then we won’t report any “incomplete” keypress but all presses will then be given a duration. If False then all keys will be presses will be returned, but only those with a corresponding release will contain a duration value (others will have duration=None)
• clear(bool (default True)) – If False then keep the keypresses for further calls (leave the buffer untouched)

Returns

Return type A list of Keypress objects

start ()
Start recording from this keyboard

stop ()
Start recording from this keyboard

class psychopy.hardware.keyboard.KeyPress (code, tDown, name=None)

Class to store key presses, as returned by Keyboard.getKeys()

Unlike keypresses from the old event.getKeys() which returned a list of strings (the names of the keys) we now return several attributes for each key:

.name: the name as a string (matching the previous pyglet name)
.rt: the reaction time (relative to last clock reset)
t.Down: the time the key went down in absolute time
duration: the duration of the keypress (or None if not released)

Although the keypresses are a class they will test ==, != and in based on their name. So you can still do:
```python
kb = Keyboard()
# wait for keypresses here
keys = kb.getKeys()
for thisKey in keys:
    if thisKey=='q':  # it is equivalent to the string 'q'
        core.quit()
    else:
        print(thisKey.name, thisKey.tDown, thisKey.rt)
```

`psychoPy.hardware.keyboard.getKeyboards()`  
Get info about the available keyboards.  
Only really useful on Mac/Linux because on these the info can be used to select a particular physical device when calling `Keyboard`. On Win this function does return information correctly but the `class:Keyboard` can't make use of it.  

**Returns** USB Info including with name, manufacturer, id, etc for each device  
**Return type** A list of dicts

### 8.4.2 BrainProducts

Python support for BrainProducts hardware.  
Here we have implemented support for the Remote Control Server application, which allows you to control recordings, send annotations etc. all from Python.

```python
class psychopy.hardware.brainproducts.RemoteControlServer(host='127.0.0.1', 
                                                          port=6700, timeout=1.0, 
                                                          testMode=False)
```

Provides a remote-control interface to BrainProducts Recorder.  
Example usage:

```python
import time
from psychopy import logging
from psychopy.hardware import brainproducts

logging.console.setLevel(logging.DEBUG)
rcs = brainproducts.RemoteControlServer()
rcs.open('testExp', 
         workspace='C:/Vision/Workfiles/Standard Workspace.rwksp', 
         participant='S0021')
rcs.openRecorder()
time.sleep(2)
rcs.mode = 'monitor' # or 'impedance', or 'default'
rcs.startRecording()
time.sleep(2)
rcs.sendAnnotation('124', 'STIM')
time.sleep(1)
rcs.pauseRecording()
time.sleep(1)
rcs.resumeRecording()
time.sleep(1)
rcs.stopRecording()
time.sleep(1)
rcs.mode = 'default' # stops monitoring mode
```
To initialize the remote control recorder.

Parameters

- **host** (*string, optional*) – The IP address or hostname of the computer running RCS.
  Defaults to 127.0.0.1.

- **port** (*int, optional*) – The port on which RCS is listening for a connection on the EEG
  computer. This should usually not need to be changed. Defaults to 6700.

- **timeout** (*float, optional*) – The timeout (in seconds) to wait for sending/receiving
  commands

- **testMode** (*bool, optional*) – If True, the network connection to the RCS computer
  will not actually be initialized. Defaults to False.

**property amplifier**

Get/set the amplifier to use

**close()**

Closes the recording and deletes all associated workspace variables (e.g. when a participant has been completed)

**dcReset()**

Use this to reset any DC offset that might have accumulated if you aren't using a high-pass filter

**property expName**

Get/set the name of the experiment or study (*string*)

The name will make up the first part of the EEG filename.

Example Usage:

```python
rcs.expName = 'MyTestStudy'
```

**property mode**

Get/set the current mode.

Mode is a string that can be one of:

- ‘default’ or ‘def’ or None will exit special modes
- ‘impedance’ or ‘imp’ for impedance checking
- ‘monitoring’ or ‘mon’
- ‘test’ or ‘tes’ to go into test view

**open** (*expName, participant, workspace*)

Opens a study/workspace on the RCS server

Parameters

- **expName** (*str*) – Name of the experiment. Will make up the first part of the EEG filename.

- **participant** (*str*) – Participant identifier. Will make up the second part of the EEG filename.

- **workspace** (*str*) – The full path to the workspace file (.rwksp), with forward slashes as path separators. e.g. “c:/myFolder/mySetup.rwksp”

**openRecorder()**

Opens the Recorder application from the Remote Control.

Neat, huh?!
property overwriteProtection
Get/set whether the

property participant
Get/set the participant identifier (a string or numeric).
This identifier will make up the center part of the EEG filename.

pauseRecording()
Pause recording EEG without ending the session.

resumeRecording()
Resume a paused recording

sendAnnotation(announcement, annType)
Sends a message to be logged on the Recorder.
The timing of annotations may be imprecise and this should not be trusted as a method of sending sync
triggers.
Annotations can contain any ASCII characters except for “;”

Parameters
• annotation (string) – The description text to be sent in the annotation.
• annType (string) – The category of the annotation which are user-defined strings (e.g.
  stimulus, response)

sendRaw(message, checkOutput='OK')
A helper function to send raw messages (strings) to the RCS.
This is normally only used for debugging purposes and is not needed by most users.

Parameters
• message (string) – The string that will be sent
• checkOutput (string (default=’OK’)) – If a value is provided then this will be
  checked for by this function. If no check is needed then set checkOutput=None

startRecording()
Start recording EEG.

stopRecording()
Stop recording EEG.

property timeout
What is a reasonable timeout in seconds (initially set to 0.5)
For some systems (e.g. when the RCS is the same machine) you might want to set this to a lower value. For
an unpredictable or slow network connection you might want to set this to a higher value.

property workspace
Get/set the path to the workspace file. An absolute path is required.
Example Usage:

rcs.workspace = 'C:\Vision\Worksfiles\testing.rwksp'
8.4.3 Cedrus (response boxes)

The pyxid package, written by Cedrus, is included in the Standalone PsychoPy distributions. See https://github.com/cedrus-opensource/pyxid for further info.

Example usage:

```python
import pyxid2 as pyxid

# get a list of all attached XID devices
devices = pyxid.get_xid_devices()

dev = devices[0]  # get the first device to use
if dev.is_response_device():
    dev.reset_base_timer()
    dev.reset_rt_timer()

    while True:
        dev.poll_for_response()
        if dev.response_queue_size() > 0:
            response = dev.get_next_response()
            # do something with the response
```

Useful functions

`pyxid2.get_xid_device(device_number)`
returns device at a given index.

Raises ValueError if the device at the passed in index doesn’t exist.

`pyxid2.get_xid_devices()`
Returns a list of all Xid devices connected to your computer.

Device classes

`class pyxid2.XidDevice(xid_connection)`
Class for interfacing with a Cedrus XID device.

At the beginning of an experiment, the developer should call:

```
XidDevice.reset_base_timer()
```

Whenever a stimulus is presented, the developer should call:

```
XidDevice.reset_rt_timer()
```

`_send_command(command, expected_bytes)`
Send an XID command to the device

`activate_line(lines=None, bitmask=None, leave_remaining_lines=False)`
Triggers an output line.

There are up to 16 output lines on XID devices that can be raised in any combination. To raise lines 1 and 7, for example, you pass in the list: activate_line(lines=[1, 7]).

To raise a single line, pass in just an integer, or a list with a single element to the lines keyword argument:

```
activate_line(lines=3)
```
or
activate_line(lines=[3])

The `lines` argument must either be an Integer, list of Integers, or None.

If you'd rather specify a bitmask for setting the lines, you can use the bitmask keyword argument. Bitmask must be a Integer value between 0 and 255 where 0 specifies no lines, and 255 is all lines. For a mapping between lines and their bit values, see the `_lines` class variable.

To use this, call the function as so to activate lines 1 and 6:

activate_line(bitmask=33)

`leave_remaining_lines` tells the function to only operate on the lines specified. For example, if lines 1 and 8 are active, and you make the following function call:

activate_line(lines=4, leave_remaining_lines=True)

This will result in lines 1, 4 and 8 being active.

If you call `activate_line(lines=4)` with `leave_remaining_lines=False` (the default), if lines 1 and 8 were previously active, only line 4 will be active after the call.

clear_line(lines=None, bitmask=None, leave_remaining_lines=False)

The inverse of activate_line. If a line is active, it deactivates it.

This has the same parameters as activate_line()

clear_response_queue()

Clears the response queue

get_next_response()

Pops the response at the beginning of the response queue and returns it.

This function returns a dict object with the following keys:

- `pressed`: A boolean value of whether the event was a keypress or key release.
- `key`: The key on the device that was pressed. This is a 0 based index.
- `port`: Device port the response came from. Typically this is 0 on RB-series devices, and 2 on SV-1 voice key devices.
- `time`: For the time being, this just returns 0. There is currently an issue with clock drift in the Cedrus XID devices. Once we have this issue resolved, time will report the value of the RT timer in miliseconds.

has_response()

Do we have responses in the queue

init_device()

Initializes the device with the proper keymaps and name

poll_for_response()

Polls the device for user input

If there is a keymapping for the device, the key map is applied to the key reported from the device. This only applies to port 0 (typically the physical buttons) responses. Rest are unchanged.

If a response is waiting to be processed, the response is appended to the internal response_queue

query_base_timer()

gets the value from the device’s base timer

reset_base_timer()

Resets the base timer
reset_rt_timer()
    Resets the Reaction Time timer.

response_queue_size()
    Number of responses in the response queue

set_pulse_duration(duration)
    Sets the pulse duration for events in milliseconds when activate_line is called

8.4.4 Cambridge Research Systems Ltd.

Cambridge Research Systems makes devices to support particularly vision research.

For stimulus display

BitsPlusPlus

Control a CRS Bits# device. See typical usage in the class summary (and in the menu demos>hardware>BitsBox of PsychoPy's Coder view).

Important: See note on BitsPlusPlusIdentityLUT

Attributes

BitsPlusPlus(win[, contrast, gamma, …])
    The main class to control a Bits++ box. This is usually a class added within the window object and is typically accessed from there. e.g.: from psychopy import visual from psychopy.hardware import crs win = visual.Window([800,600]) bits = crs.BitsPlusPlus(win, mode='bits++') # use bits++ to reduce the whole screen contrast by 50%: bits.setContrast(0.5).

BitsPlusPlus.setContrast(contrast[, …])
    Set the contrast of the LUT for 'bits++' mode only :Parameters: contrast : float in the range 0:1 The contrast for the range being set LUT range : float or array If a float is given then this is the fraction of the LUT to be used. If an array of floats is given, these will specify the start / stop points as fractions of the LUT. If an array of ints (0-255) is given these determine the start stop indices of the LUT .. rubric:: Examples.

BitsPlusPlus.setGamma(newGamma)
    Set the LUT to have the requested gamma value Currently also resets the LUT to be a linear contrast ramp spanning its full range.

continues on next page
Table 21 – continued from previous page

Sets the LUT to a specific range of values in ‘bits++’ mode only. Note that, if you leave gammaCorrect=True then any LUT values you supply will automatically be gamma corrected. The LUT will take effect on the next Window.flip().

**Examples:**
```
bitsBox.setLUT()
```
builds a LUT using bitsBox.contrast and bitsBox.gamma
```
bitsBox.setLUT(newLUT=some256x1array)
```
(NB array should be float 0.0:1.0) Builds a luminance LUT using newLUT for each gun (actually array can be 256x1 or 1x256)
```
bitsBox.setLUT(newLUT=some256x3array)
```
(NB array should be float 0.0:1.0) Allows you to use a different LUT on each gun (NB by using BitsBox.setContr() and BitsBox.setGamma() users may not need this function).

## Details

**class** `psychopy.hardware.crs.bits.BitsPlusPlus` *(win, contrast=1.0, gamma=None, nEntries=256, mode='bits++', rampType='configFile', frameRate=None)*

The main class to control a Bits++ box. This is usually a class added within the window object and is typically accessed from there. E.g.:

```
from psychopy import visual
from psychopy.hardware import crs
win = visual.Window([800,600])
bis = crs.BitsPlusPlus(win, mode='bits++')
# use bits++ to reduce the whole screen contrast by 50%:
bis.setContrast(0.5)
```

**Parameters**

- **contrast**: The contrast to be applied to the LUT. See `BitsPlusPlus.setLUT()` and `BitsPlusPlus.setContrast()` for flexibility on setting just a section of the LUT to a different value.
- **gamma**: The value used to correct the gamma in the LUT.
- **nEntries**: [256] [DEPRECATED feature]
- **mode**: ['bits++' (or 'mono++' or 'color++')] Note that, unlike the Bits#, this only affects the way the window is rendered, it does not switch the state of the Bits++ device itself (because unlike the Bits# have no way to communicate with it). The mono++ and color++ are only supported in PsychoPy 1.82.00 onwards. Even then they suffer from not having gamma correction applied on Bits++ (unlike Bits# which can apply a gamma table in the device hardware).
- **rampType**: ['configFile', None or an integer] if ‘configFile’ then we’ll look for a valid config in the userPrefs folder if an integer then this will be used during win.setGamma(rampType=rampType):

**frameRate**: an estimate the frameRate of the monitor. If None frame rate will be calculated.

**_Goggles()** *(private)* Used to set control the goggles. Should not be needed by user if attached to a `psychopy.visual.Window`
_ResetClock()
(Private) Used to reset Bits hardware clock. Should not be needed by user if attached to a psychopy.
visual.Window since this will automatically draw the reset code as part of the screen refresh.

_drawLUTtoScreen()
(Private) Used to set the LUT in ‘bits++’ mode. Should not be needed by user if attached to a psychopy.
visual.Window since this will automatically draw the LUT as part of the screen refresh.

_drawTrigtoScreen(sendStr=None)
(Private) Used to send a trigger pulse. Should not be needed by user if attached to a psychopy.
visual.Window since this will automatically draw the trigger code as part of the screen refresh.

_protectTrigger()
If Goggles (or analog) outputs are used when the digital triggers are off we need to make a set of blank triggers
first. But the user might have set up triggers in waiting for a later time. So this will protect them.

_restoreTrigger()
Restores the triggers to previous settings

_setHeaders(frameRate)
Sets up the TLock header codes and some flags that are common to operating all CRS devices

_setupShaders()
creates and stores the shader programs needed for mono++ and color++ modes

getPackets()
Returns the number of packets available for trigger pulses.

primeClock()
Primes the clock to reset at the next screen flip - note only 1 clock reset signal will be issued but if the frame(s)
after the reset frame is dropped the reset will be re-issued thus keeping timing good.

Resets continue to be issued on each video frame until the next win.flip so you need to have regular win.flips
for this function to work properly.

**Example**  bits.primeClock() drawImage while not response

# do some processing bits.win.flip()

Will get a clock reset signal ready but wont issue it until the first win.flip in the loop.

reset()
Deprecated: This was used on the old Bits++ to power-cycle the box. It required the compiled dll, which
only worked on windows and doesn’t work with Bits# or Display++.

resetClock()
Issues a clock reset code using 1 screen flip if the next frame(s) is dropped the reset will be re-issued thus
keeping timing good.

Resets continue to be issued on each video frame until the next win.flip so you need to have regular win.flips
for this function to work properly.
Example

`bits.resetClock() drawImage() bits.win.flip()`

Will issue clock resets while the image is being drawn then display the inmage and allow the clock to continue formt he same frame.

Example

`bits.resetClock() bits.RTBoxWait() bits.win.flip()`

Will issue clock resets until a button is presses.

`sendTrigger` *(triggers=0, onTime=0, duration=0, mask=65535)*

Sends a single trigger using up 1 win.flip. The trigger will be sent on the following frame.

The triggers will continue until after the next win.flip.

Actions are always 1 frame after the request.

May do odd things if Goggles and Analog are also in use.

Example:

```python
bits.sendTrigger(0b0000000010, 2.0, 4.0)
bits.win.flip()
```

Will send a 4ms pulse on DOUT1 2ms after the start of the frame. Due to the following win.flip() the pulse should last for 1 frame only.

Triggers will continue until stopTrigger is called.

`setContrast` *(contrast, LUTrange=1.0, gammaCorrect=None)*

Set the contrast of the LUT for ‘bits++’ mode only .Parameters:

- **contrast** [float in the range 0:1] The contrast for the range being set
- **LUTrange** [float or array] If a float is given then this is the fraction of the LUT to be used. If an array of floats is given, these will specify the start / stop points as fractions of the LUT. If an array of ints (0-255) is given these determine the start stop indices of the LUT

Examples

`setContrast(1.0, 0.5)` will set the central 50% of the LUT so that a stimulus with contr=0.5 will actually be drawn with contrast 1.0

`setContrast(1.0, [0.25, 0.5]) setContrast(1.0, [63,127])`

will set the lower-middle quarter of the LUT (which might be useful in LUT animation paradigms)

`setGamma` *(newGamma)*

Set the LUT to have the requested gamma value Currently also resets the LUT to be a linear contrast ramp spanning its full range. May change this to read the current LUT, undo previous gamma and then apply new one?

`setLUT` *(newLUT=None, gammaCorrect=True, LUTrange=1.0)*

Sets the LUT to a specific range of values in ‘bits++’ mode only Note that, if you leave gammaCorrect=True then any LUT values you supply will automatically be gamma corrected. The LUT will take effect on the next `Window.flip()` Examples:
bitsBox.setLUT() builds a LUT using bitsBox.contrast and bitsBox.gamma

bitsBox.setLUT(newLUT=some256x1array) (NB array should be float 0.0:1.0) Builds a luminance LUT using newLUT for each gun (actually array can be 256x1 or 1x256)

bitsBox.setLUT(newLUT=some256x3array) (NB array should be float 0.0:1.0) Allows you to use a different LUT on each gun

(NB by using BitsBox.setContr() and BitsBox.setGamma() users may not need this function)

setTrigger (triggers=0, onTime=0, duration=0, mask=65535)
Quick way to set up triggers.

Triggers is a binary word that determines which triggers will be turned on.

onTime specifies the start time of the trigger within the frame (in S with 100uS resolution)

Duration specifies how long the trigger will last. (in S with 100uS resolution).

Note that mask only protects the digital output lines set by other activities in the Bits. Not other triggers.

Example::

bits.setTrigger(0b0000000010, 2.0, 4.0, 0b0111111111) bits.startTrigger()

Will issue a 4ms long high-going pulse, 2ms after the start of each frame on DOUT1 while protecting the value of DOUT 9.

setTriggerList (triggerList=None, mask=65535)
Sets up Trigger pulses in Bits++ using the fine grained method that can control every trigger line at 100uS intervals.

TriggerList should contain 1 entry for every 100uS packet (see getPackets) the binary word in each entry specifies which trigger line will be active during that time slot.

Note that mask only protects the digital output lines set by other activities in the Bits. Not other triggers.

Example:

```python
packet = [0]*self._NumberPackets
packet[0] = 0b0000000010
bits.setTriggerList(packet)
```

Will sens a 100us pulse on DOUT1 at the start of the frame.

Example 2:

```python
packet = [0]*self._NumberPackets
packet[10] = 0b0000000010
packet[20] = 0b0000000001
bits.setTriggerList(packet)
bits.startTrigger()
```

Will sens a 100us pulse on DOUT1 1000us after the start of the frame and a second 100us pulse on DOUT0 2000us after the start of the frame.

Triggers will continue until stopTrigger is called.

startGoggles (left=0, right=1)
Starts CRS stereo goggles. Note if you are using FE-1 goggles you should start this before connecting the goggles.

Left is the state of the left shutter on the first frame to be presented 0, False or 'closed'=closed; 1, True or 'open' = open,

right is the state of the right shutter on the first frame to be presented 0, False or 'closed'=closed; 1, True or 'open' = open
Note you can set the goggles to be both open or both closed on the same frame.

The system will always toggle the state of each lens so as to not damage FE-1 goggles.

Example:

```
bits.startGoggles(0, 1)
bits.win.flip()
while not response
    bits.win.flip()
    #do some processing
bits.stopGoggles()
bits.win.flip()
```

Starts toggling the goggles with the right eye open in sync with the first win.flip() within the loop. The open eye will alternate.

Example:

```
bits.startGoggles(1, 1)
bits.win.flip()
while not response:
    bits.win.flip()
    #do some processing
bits.stopGoggles()
bits.win.flip()
```

Starts toggling the gaggle with both eyes open in sync with the first win.flip() within the loop. Eyes will alternate between both open and both closed.

Note it is safe to leave the goggles toggling forever, ie to never call stopGoggles().

**startTrigger()**

Start sending triggers on the next win flip and continue until stopped by stopTrigger Triggers start 1 frame after the frame on which the first trigger is sent.

Example:

```
bits.setTrigger(0b0000000010, 2.0, 4.0, 0b0111111111)
bits.startTrigger()
while imageOn:
    #do some processing
    continue
bits.stopTrigger()
bits.win.flip()
```

**stopGoggles()**

Stop the stereo goggles from toggling

Example:

```
bits.startGoggles(0, 1)
bits.win.flip()
while not response:
    bits.win.flip()
    #do some processing
bits.stopGoggles()
bits.win.flip()
```

Starts toggling the goggles with the right eye open in sync with the first win.flip(0) within the loop. The open eye will alternate.
Note it is safer to leave the goggles toggling forever, i.e. to never call stopGoggles().

**stopTrigger()**

Stop sending triggers at the next win flip

Example:

```python
bits.setTrigger(0b0000000010, 2.0, 4.0, 0b0111111111)
bits.startTrigger()
while imageOn:
    #do some processing
continue
bits.stopTrigger()
bits.win.flip()
```

**syncClocks(t)**

Synchronise the Bits/RTBox Clock with the host clock Given by t.

### Finding the identity LUT

For the Bits++ (and related) devices to work correctly it is essential that the graphics card is not altering in any way the values being passed to the monitor (e.g. by gamma correcting). It turns out that finding the ‘identity’ LUT, where exactly the same values come out as were put in, is not trivial. The obvious LUT would have something like 0/255, 1/255, 2/255... in entry locations 0,1,2... but unfortunately most graphics cards on most operating systems are ‘broken’ in one way or another, with rounding errors and incorrect start points etc.

PsychoPy provides a few of the common variants of LUT and that can be chosen when you initialise the device using the parameter `rampType`. If no `rampType` is specified then PsychoPy will choose one for you:

```python
from psychopy import visual
from psychopy.hardware import crs

win = visual.Window([1024,768], useFBO=True) #we need to be rendering to
    `framebuffer`
bits = crs.BitsPlusPlus(win, mode = 'bits++', rampType = 1)
```

The Bits# is capable of reporting back the pixels in a line and this can be used to test that a particular LUT is indeed providing identity values. If you have previously connected a BitsSharp device and used it with PsychoPy then a file will have been stored with a LUT that has been tested with that device. In this case set `rampType = "configFile"` for PsychoPy to use it if such a file is found.

**BitsSharp**

Control a CRS Bits# device. See typical usage in the class summary (and in the menu demos>hardware>BitsBox of PsychoPy's Coder view).
Attributes

**BitsSharp**([win, portName, mode, ...])  
A class to support functions of the Bits# (and most Display++ functions) This device uses the CDC (serial port) connection to the Bits box. To use it you must have followed the instructions from CRS Ltd. to get your box into the CDC communication mode. Typical usage (also see demo in Coder view demos>hardware>BitsBox)::

**BitsSharp.mode**  
Get/set the mode of the BitsSharp to one of: “bits++” “mono++” “color++” “status” “storage” “auto”

**BitsSharp.isAwake()**  
Test whether we have an active connection on the virtual serial port

**BitsSharp.getInfo()**  
Returns a python dictionary of info about the Bits Sharp box

**BitsSharp.checkConfig([level, demoMode, log-File])**  
Checks whether there is a configuration for this device and whether it’s correct :params: level: integer 0: do nothing 1: check that we have a config file and that the graphics card and operating system match that specified in the file. Then assume identity LUT is correct 2: switch the box to status mode and check that the identity LUT is currently working 3: force a fresh search for the identity LUT.

**BitsSharp.gammaCorrectFile**  
Get / set the gamma correction file to be used (as stored on the device)

**BitsSharp.temporalDithering**  
Temporal dithering can be set to True or False

**BitsSharp.monitorEDID**  
Get / set the EDID file for the monitor.

**BitsSharp.beep([freq, dur])**  
Make a beep of a given frequency and duration

**BitsSharp.getVideoLine(lineN, nPixels[, ...])**  
Return the r,g,b values for a number of pixels on a particular video line :param lineN: the line number you want to read :param nPixels: the number of pixels you want to read :param nAttempts: the first time you call this function it has to get to status mode. In this case it sometimes takes a few attempts to make the call work :return: an N\times3 numpy array of uint8 values.

**BitsSharp.start()**  
[Not currently implemented] Used to begin event collection by the device.

**BitsSharp.stop()**  
[Not currently implemented] Used to stop event collection by the device.

Direct communications with the serial port:

**BitsSharp.sendMessage(message[, autoLog])**  
Send a command to the device (does not wait for a reply or sleep())

**BitsSharp.getResponse([length, timeout])**  
Read the latest response from the serial port

Control the CLUT (Bits++ mode only):
**BitsSharp.setContrast(contrast[, LUTrange, ...])**  
Set the contrast of the LUT for 'bits++' mode only. Parameters:  
contrast : float in the range 0:1 The contrast for the range being set LUTrange : float or array If a float is given then this is the fraction of the LUT to be used. If an array of floats is given, these will specify the start / stop points as fractions of the LUT. If an array of ints (0-255) is given these determine the start stop indices of the LUT.  

.. rubric:: Examples.

**BitsSharp.setGamma(newGamma)**  
Set the LUT to have the requested gamma value. Currently also resets the LUT to be a linear contrast ramp spanning its full range.

**BitsSharp.setLUT([newLUT, gammaCorrect, ...])**  
SetLUT is only really needed for bits++ mode of bits# to set the look-up table (256 values with 14bits each).

### Details

**class** psychopy.hardware.crs.bits.BitsSharp *(win=None, portName=None, mode='', checkConfigLevel=1, gammaCorrect='hardware', gamma=None, noComms=False)*

A class to support functions of the Bits# (and most Display++ functions) This device uses the CDC (serial port) connection to the Bits box. To use it you must have followed the instructions from CRS Ltd. to get your box into the CDC communication mode. Typical usage (also see demo in Coder view demos>hardware>BitsBox):

```python
from psychopy import visual
from psychopy.hardware import crs
# we need to be rendering to framebuffer
win = visual.Window([1024, 768], useFBO=True)
b = crs.BitsSharp(win, mode = 'mono++')
# You can continue using your window as normal and OpenGL shaders
# will convert the output as needed
print(b.info)
if not b.OK:
    print('failed to connect to Bits box')
    core.quit()
core.wait(0.1)
# now, you can change modes using
b.mode = 'mono++'  # 'color++', 'mono++', 'bits++', 'status'
```

Note that the firmware in Bits# boxes varies over time and some features of this class may not work for all firmware versions. Also Bits# boxes can be configured in various ways via their config.xml file so this class makes certain assumptions about the configuration. In particular it is assumed that all digital inputs, triggers and analog inputs are reported as part of status updates. If some of these report are disabled in your config.xml file then ‘status’ and ‘event’ commands in this class may not work.

RTBox commands that reset the key mapping have been found not to work on some firmware.

**Parameters**  
win : a PsychoPy Window object, required  
portName : the (virtual) serial port to which the device is connected. If None then PsychoPy will search available serial ports and test communication (on OSX, the first match of /dev/tty.usbmodemfa* will be used and on linux /dev/ttyS0 will be used)

Allows you to specify how much checking of the device is done to ensure a valid identity look-up table. If you specify one level and it fails then the check will be escalated to the next level (e.g. if we check level 1 and find that it fails we try to find a new LUT):

- 0 don’t check at all
- 1 check that the graphics driver and OS version haven’t changed since last LUT calibration
- 2 check that the current LUT calibration still provides identity (requires switch to status mode)
- 3 search for a new identity look-up table (requires switch to status mode)

**gammaCorrect** [string governing how gamma correction is performed]

- ‘hardware’: use the gamma correction file stored on the hardware
- ‘FBO’: gamma correct using shaders when rendering the FBO to back buffer
- ‘bitsMode’: in bits++ mode there is a user-controlled LUT that we can use for gamma correction

**noComms** [bool] If True then don’t try to communicate with the device at all (passive mode). This can be useful if you want to debug the system without actually having a Bits# connected.

**RTBoxAddKeys** *(map)*

Add key mappings to an existing map. RTBox events can be mapped to a number of physical events on Bits#. They can be mapped to digital input lines, triggers and CB6 IR input channels. The format for map is a list of tuples with each tuple containing the name of the RTBox button to be mapped and its source eg (‘btn1’,‘Din1’) maps physical input Din1 to logical button btn1. RTBox has four logical buttons (btn1-4) and three auxiliary events (light, pulse and trigger) Buttons/events can be mapped to multiple physical inputs and stay mapped until reset.

Example:

```python
bits.RTBoxSetKeys([('btn1','Din0'),('btn2','Din1')])
b ..."IRButtonB")])
```

Will link Din0 to button 1 and Din1 to button 2. Then adds IRButtonA and IRButtonB alongside the original mappings.

Now both hard wired and IR inputs will - emulating the same logical button press.

Note that the firmware in Bits# units varies over time and some features of this class may not work for all firmware versions. Also Bits# units can be configured in various ways via their config.xml file so this class makes certain assumptions about the configuration. Such variations may affect key mappings for RTBox commands.

**RTBoxCalibrate** *(N=1)*

Used to assess error between host clock and Bits# button press time stamps.

Prints each sample provided and returns the mean error.

The clock will never be completely in sync but the aim is that there should be that the difference between them should not grow over a serise of button presses.

Note that the firmware in Bits# units varies over time and some features of this class may not work for all firmware versions. Also Bits# units can be configured in various ways via their config.xml file so this class makes certain assumptions about the configuration. Such variations may affect key mappings for RTBox commands.
**RTBoxClear()**
Flushes the serial input buffer. Its good to do this before and after data collection. This just calls flush() so is a wrapper for RTBox.

**RTBoxDisable()**
Disables the detection of RTBox events. This is useful to stop the Bits# from reporting key presses. When you no longer need them, Nad must be done before using any other data logging methods.

It undoes any button - input mappings.

Note that the firmware in Bits# units varies over time and some features of this class may not work for all firmware versions. Also Bits# units can be configured in various ways via their config.xml file so this class makes certain assumptions about the configuration. Such variations may affect key mappings for RTBox commands.

The ability to reset keys mappings has been found not to work on some Bits# firmware.

**RTBoxEnable(mode=None, map=None)**
Sets up the RTBox with preset or bespoke mappings and enables event detection.

RTBox events can be mapped to a number of physical events on Bits#. They can be mapped to digital input lines, tigers and CB6 IR input channels.

Mode is a list of strings. Preset mappings provided via mode:

- CB6 for the CRS CB6 IR response box.
- IO for a three button box connected to Din0-2
- IO6 for a six button box connected to DinO-5

If mode = None or is not set then the value of self.RTBoxMode is used.

Bespoke Mappings overwrite preset ones.

The format for map is a list of tuples with each tuple containing the name of the RT Box button to be mapped and its source eg (’btn1’, ’Din0’) maps physical input Din0 to logical button btn1.

Note the lowest number button event is Btn1

RTBox has four logical buttons (btn1-4) and three auxiliary events (light, pulse and trigger). Buttons/events can be mapped to multiple physical inputs and stay mapped until reset.

Mode is a list of string or list of strings that contains keywords to determine present mappings and modes for RTBox.

If mode includes ‘Down’ button events will be detected when pressed. If mode includes ‘Up’ button events will be detected when released. You can detect both types of event but note that pulse, light and trigger events don’t have an ‘Up’ mode.

**If Trigger is included in mode the trigger** event will be mapped to the trigIn connector.

**Example**

bits.RTBoxEnable(mode = [‘Down’], map = [‘btn1’, ’Din0’], [‘btn2’, ’Din1’])

enable the RTBox emulation to detect Down events on buttons 1 and 2 where they are mapped to DIN0 and DIN1.
PsychoPy-Psychology software for Python, Release 2020.2.10

Example

bits.RTBoxEnable(mode = ['Down','CB6'])

enable the RTBox emulation to detect Down events on the standard CB6 IR response box keys.

If no key direction has been set (mode does not contain 'Up' or 'Down') the default is 'Down'.

Note that the firmware in Bits# units varies over time and some features of this class may not work for all firmware versions. Also Bits# units can be configured in various ways via their config.xml file so this class makes certain assumptions about the configuration. Such variations may affect key mappings for RTBox commands.

The ability to reset keys mappings has been found not to work on some Bits# firmware.

**RTBoxKeysPressed** \((N=1)\)

Check to see if (at least) the appropriate number of RTBox style key presses have been made.

Example

bits.RTBoxKeysPressed(5)

will return false until 5 button presses have been recorded.

Note that the firmware in Bits# units varies over time and some features of this class may not work for all firmware versions. Also Bits# units can be configured in various ways via their config.xml file so this class makes certain assumptions about the configuration. Such variations may affect key mappings for RTBox commands.

**RTBoxResetKeys** ()

Resets the key mappings to no mapping. Has the effect of disabling RTBox input.

Note that the firmware in Bits# units varies over time and some features of this class may not work for all firmware versions. Also Bits# units can be configured in various ways via their config.xml file so this class makes certain assumptions about the configuration. Such variations may affect key mappings for RTBox commands.

The ability to reset keys mappings has been found not to work on some Bits# firmware.

**RTBoxSetKeys** \((map)\)

Set key mappings: first resets existing then adds new ones. Does not reset any event that is not in the new list. RTBox events can be mapped to a number of physical events on Bits# They can be mapped to digital input lines, triggers and CB6 IR input channels. The format for map is a list of tuples with each tuple containing the name of the RTBox button to be mapped and its source eg ('btn1','Din1') maps physical input Din1 to logical button btn1.

RTBox has four logical buttons (btn1-4) and three auxiliary events (light, pulse and trigger) Buttons/events can be mapped to multiple physical inputs and stay mapped until reset.
Example

```python
bits.RTBoxSetKeys([('btn1', 'Din0'), ('light', 'Din9'))
```

Will link Din0 to button 1 and Din9 to the the light input emulation.

Note that the firmware in Bits# units varies over time and some features of this class may not work for all firmware versions. Also Bits# units can be configured in various ways via their config.xml file so this class makes certain assumptions about the configuration. Such variations may affect key mappings for RTBox commands.

**RTBoxWait()**

Waits until (at least) one of RTBox style key presses have been made Pauses program execution in mean time.

Example

```python
res = bits.RTBoxWait()
```

will suspend all other activity until 1 button press has been recorded and will then return a dict / structure containing results.

Results can be accessed as follows:

```python
structure res.dir, res.button, res.time
or dictionary res['dir'], res['button'], res['time']
```

Note that the firmware in Bits# units varies over time and some features of this class may not work for all firmware versions. Also Bits# units can be configured in various ways via their config.xml file so this class makes certain assumptions about the configuration. Such variations may affect key mappings for RTBox commands.

**RTBoxWaitN(N=1)**

Waits until (at least) the appropriate number of RTBox style key presses have been made Pauses program execution in mean time.

Example

```python
res = bits.RTBoxWaitN(5)
```

will suspend all other activity until 5 button presses have been recorded and will then return a list of Dicts containing the 5 results.

Results can be accessed as follows:

```python
structure res[0].dir, res[0].button, res[0].time
or dictionary res[0]['dir'], res[0]['button'], res[0]['time']
```

Note that the firmware in Bits# units varies over time and some features of this class may not work for all firmware versions. Also Bits# units can be configured in various ways via their config.xml file so this class makes certain assumptions about the configuration. Such variations may affect key mappings for RTBox commands.

**_Goggles()**

(private) Used to set control the goggles. Should not be needed by user if attached to a `psychopy.visual.Window`
_RTBoxDecodeResponse (msg, N=1)
Helper function for decoding key presses in the RT response box format.
Not normally needed by user

_ResetClock ()
(private) Used to reset Bits hardware clock. Should not be needed by user if attached to a psychopy.
visual.Window since this will automatically draw the reset code as part of the screen refresh.

drawLUTtoScreen ()
(private) Used to set the LUT in ‘bits++’ mode. Should not be needed by user if attached to a psychopy.
visual.Window since this will automatically draw the LUT as part of the screen refresh.

drawTrigtoScreen (sendStr=None)
(private) Used to send a trigger pulse. Should not be needed by user if attached to a psychopy.visual.
Window since this will automatically draw the trigger code as part of the screen refresh.

_extractStatusEvents ()
Interprets values from status log to pullout any events.
Should not be needed by user if start/stopStatusLog or pollStatus are used
Fills statusEvents with a list of dictionary like objects with the following entries source, input, direction, time.
source = the general source of the event - e.g. DIN for Digital input, IR for IT response box
input = the individual input in the source. direction = ‘up’ or ‘down’ time = time stamp.

Events are recorded relative to the four event flags statusDINBase, initial values for digital ins. statusIR-
Base, initial values for CB6 IR box. statusTrigInBase, initial values for TrigIn. statusMode, direction(s) of events to be reported.
The data can be accessed as statusEvents[i][‘time’] or statusEvents[i].time
Also set status._nEvents to the number of events recorded

_getStatusLog ()
Read the log Queue
Should not be needed by user if start/stopStatusLog or pollStatus are used.
fills statusValues with a list of dictionary like objects with the following entries: sample, time, trigIn, DIN[10],
DWORD, IR[6], ADC[6]
They can be accessed as statusValues[i][‘sample’] or statusValues[i].sample, statusValues[i].ADC[j]
Also sets status._nValues to the number of values recorded.

_inWaiting ()
Helper function to determine how many bytes are waiting on the serial port.

_protectTrigger ()
If Goggles (or analog) outputs are used when the digital triggers are off we need to make a set of blank triggers
first. But the user might have set up triggers in waiting for a later time. So this will protect them.

_restoreTrigger ()
Restores the triggers to previous settings

_setHeaders (frameRate)
Sets up the TLock header codes and some flags that are common to operating all CRS devices

_setupShaders ()
creates and stores the shader programs needed for mono++ and color++ modes
_statusBox()
Should not normally be called by user. Called in its own thread via self.statusBoxEnable() Reads the status reports from the Bits# for default 60 seconds or until self.statusBoxDisable() is called.

Note any non status reports are found on the buffer will cause an error.

args specifies the time over which to record status events. The minimum time is 10ms, less than this results in recording stopping after about 1 status report has been read.

Puts its results into a Queue.

This function is normally run in its own thread so actions can be asynchronous.

_statusDisable()
Stop Bits# from recording data - and clears the buffer

Not normally needed by user

_statusEnable()
Sets the Bits# to continuously send back its status until stopped. You get a lot a data by leaving this going.

Not normally needed by user

_statusLog(args=60)
Should not normally be called by user. Called in its own thread via self.startStatusLog() Reads the status reports from the Bits# for default 60 seconds or until self.stopStatusLog() is called. Ignores the last line as this is can be bogus. Note any non status reports are found on the buffer will cause an error.

args specifies the time over which to record status events. The minimum time is 10ms, less than this results in recording stopping after about 1 status report has been read.

Puts its results into a Queue.

This function is normally run in its own thread so actions can be asynchronous.

beep(freq=800, dur=1)
Make a beep of a given frequency and duration

ccheckConfig(level=1, demoMode=False, logFile="")
Checks whether there is a configuration for this device and whether it’s correct :params:

level: integer 0: do nothing 1: check that we have a config file and that the graphics card and operating system match that specified in the file. Then assume identity LUT is correct

2: switch the box to status mode and check that the identity LUT is currently working

3: force a fresh search for the identity LUT

clock()
Reads the internal clock of the Bits box via the RTBox format but note there will be a delay in reading the value back. The format for the return values is the same as for button box presses. The return value for button will be 9 and the return value for event will be time. The return value for time will be the time of the clock at the moment of the request.
Example

```
res = bits.clock() print(res.time) print(res['time'])
driverFor = []
flush()
```

Flushes the serial input buffer. It’s good to do this before and after data collection, and generally quite often.

**property gammaCorrectFile**

Get/set the gamma correction file to be used (as stored on the device)

**getAllRTBoxResponses()**

Read all of the RTBox style key presses on the input buffer. Returns a list of dict like objects with three members ‘button’, ‘dir’ and ‘time’

‘button’ is a number from 1 to 9 to indicate the event that was detected. 1-4 are the ‘btn1-btn4’ events, 5 and 6 are the ‘light’ and ‘pulse’ events, 7 is the ‘trigger’ event, 9 is a requested timestamp event (see Clock()).

‘dir’ is the direction of the event eg ‘up’ or ‘down’, trigger is described as ‘on’ when low.

‘dir’ is set to ‘time’ if a requested timestamp event has been detected.

‘time’ is the timestamp associated with the event.

Values can be read as a structure eg:

```
res = getAllRTBoxResponses()
res[0].dir, res[0].button, res[0].time
```

or dictionary:

```
res[0]['dir'], res[0]['button'], res[0]['time']
```

Note even if only 1 key press was found a list of dict/objects is returned

Note that the firmware in Bits# units varies over time and some features of this class may not work for all firmware versions. Also Bits# units can be configured in various ways via their config.xml file so this class makes certain assumptions about the configuration. Such variations may affect key mappings for RTBox commands.

**getAllStatusBoxResponses()**

Read all of the statusBox style key presses on the input buffer. Returns a list of dict like objects with three members ‘button’, ‘dir’ and ‘time’

‘button’ is a number from 1 to 9 to indicate the event that was detected. 1-17 are the ‘btn1-btn17’ events.

‘dir’ is the direction of the event eg ‘up’ or ‘down’, trigger is described as ‘on’ when low.

‘dir’ is set to ‘time’ if a requested timestamp event has been detected.

‘time’ is the timestamp associated with the event.

Values can be read as a structure eg:

```
res = getAllStatusBoxResponses()
res[0].dir, res[0].button, res[0].time
```

or dictionary:

```
res[0]['dir'], res[0]['button'], res[0]['time']
```
Note even if only 1 key press was found a list of dict / objects is returned.

Note that the firmware in Bits# units varies over time and some features of this class may not work for all firmware versions. Also Bits# units can be configured in various ways via their config.xml file so this class makes certain assumptions about the configuration. In particular it is assumed that all digital inputs, triggers and analog inputs are reported as part of status updates. If some of these report are disabled in your config.xml file then ‘status’ and ‘event’ commands in this class may not work.

**getAllStatusEvents()**

Returns the whole status event list

Returns a list of dictionary like objects with the following entries source, input, direction, time.

source = the general source of the event - e.g. DIN for Digital input, IR for CB6 IR response box events
input = the individual input in the source. direction = ‘up’ or ‘down’ time = time stamp.
All sources are numbered from zero. Din 0 … 9 IR 0 … 5 ADC 0 … 5
mode specifies which directions of events are captured. e.g ‘up’ will only report up events.
The data can be accessed as value[i][‘time’] or value[i].time

**Example**

```python
bits.startStatusLog() while not event
    # do some processing continue
bits.stopStatusLog() res=getAllStatusEvents() print(bits.res[0].time)
```

**getAllStatusValues()**

Returns the whole status values list.

Returns a list of dict like objects with the following entries sample, time, trigIn, DIN[10], DWORD, IR[6], ADC[6] sample is the sample ID number. time is the time stamp. trigIn is the value of the trigger input. DIN is a list of 10 digital input values. DWORD represents the digital inputs as a single decimal value. IR is a list of 10 infra-red (IR) input values. ADC is a list of 6 analog input values. These can be accessed as value[i][‘sample’] or value[i].sample, values[i].ADC[j].

All sources are numbered from zero. Din 0 … 9 IR 0 … 5 ADC 0 … 5

**Example**

```python
bits.startStatusLog() while not event
    # do some processing continue
bits.stopStatusLog() res=getAllStatusValues() print(bits.res[0].time)
```
**getAnalog** \((N=0)\)
Pulls out the values of the analog inputs for the Nth status entry.

Returns a dictionary with a list of 6 floats (ADC) and a time stamp (time).

All sources are numbered from zero. ADC 0 … 5

**Example**

```python
bits.pollStatus() res=bits.getAnalog() print(res['ADC'])
```

will poll the status display the values of the ADC inputs in the first status entry returned.

Note that the firmware in Bits# units varies over time and some features of this class may not work for all firmware versions. Also Bits# units can be configured in various ways via their config.xml file so this class makes certain assumptions about the configuration. In particular it is assumed that all digital inputs, triggers and analog inputs are reported as part of status updates. If some of these report are disabled in your config.xml file then ‘status’ and ‘event’ commands in this class may not work.

**getDigital** \((N=0)\)
Pulls out the values of the digital inputs for the Nth status entry.

Returns a dictionary with a list of 10 ints that are 1 or 0 (DIN) and a time stamp (time)

All sources are numbered from zero. Din 0 … 9

**Example**

```python
bits.pollStatus() res=bits.getAnalog() print(res['DIN'])
```

will poll the status display the value of the digital inputs in the first status entry returned.

Note that the firmware in Bits# units varies over time and some features of this class may not work for all firmware versions. Also DBits# units can be configured in various ways via their config.xml file so this class makes certain assumptions about the configuration. In particular it is assumed that all digital inputs, triggers and analog inputs are reported as part of status updates. If some of these report are disabled in your config.xml file then ‘status’ and ‘event’ commands in this class may not work.

**getDigitalWord** \((N=0)\)
Pulls out the values of the digital inputs for the Nth status entry.

Returns a dictionary with a 10 bit word representing the binary values of those inputs (DWORD) and a time stamp (time).

**Example**

```python
bits.pollStatus() res=bits.getAnalog() print(res['DWORD'])
```

will poll the status display the value of the digital inputs as a decimal number.

Note that the firmware in Bits# units varies over time and some features of this class may not work for all firmware versions. Also Bits# units can be configured in various ways via their config.xml file so this class makes certain assumptions about the configuration. In particular it is assumed that all digital inputs, triggers and analog inputs are reported as part of status updates. If some of these report are disabled in your config.xml file then ‘status’ and ‘event’ commands in this class may not work.
**getIRBox** \((N=0)\)
Pulls out the values of the CB6 IR response box inputs for the Nth status entry.

Returns a dictionary with a list of 6 ints that are 1 or 0 (IRBox) and a time stamp (time).
Il sources are numbered from zero. IR 0 … 5

### Example

```python
bits.pollStatus() res=bits.getAnalog() print(res['IRBox'])
```
will poll the status display the values of the IR box buttons in the first status entry returned.

Note that the firmware in Bits# units varies over time and some features of this class may not work for all firmware versions. Also Bits# units can be configured in various ways via their config.xml file so this class makes certain assumptions about the configuration. In particular it is assumed that all digital inputs, triggers and analog inputs are reported as part of status updates. If some of these report are disabled in your config.xml file then ‘status’ and ‘event’ commands in this class may not work.

**getInfo**
Returns a python dictionary of info about the Bits Sharp box

### Example::
```python
info=bits.getInfo print(info['ProductType'])
```

**getPackets**
Returns the number of packets available for trigger pulses.

**getRTBoxResponse**
checks for one RTBox style key presses on the input buffer then reads it. Returns a dict like object with three members ‘button’, ‘dir’ and ‘time’

‘button’ is a number from 1 to 9 to indicate the event that was detected. 1-4 are the ‘btn1-btn4’ events, 5 and 6 are the ‘light’ and ‘pulse’ events, 7 is the ‘trigger’ event, 9 is a requested timestamp event (see Clock()).

‘dir’ is the direction of the event eg ‘up’ or ‘down’, trigger is described as ‘on’ when low.

‘dir’ is set to ‘time’ if a requested timestamp event has been detected.

‘time’ is the timestamp associated with the event.

**Value can be read as a structure, eg:**
```python
res= getRTBoxResponse() res.dir, res.button, res.time
```

**or dictionary**
```python
res['dir'], res['button'], res['time']
```

Note that the firmware in Bits# units varies over time and some features of this class may not work for all firmware versions. Also Bits# units can be configured in various ways via their config.xml file so this class makes certain assumptions about the configuration. Such variations may affect key mappings for RTBox commands.

**getRTBoxResponses** \((N=1)\)
checks for (at least) an appropriate number of RTBox style key presses on the input buffer then reads them. Returns a list of dict like objects with three members ‘button’, ‘dir’ and ‘time’

‘button’ is a number from 1 to 9 to indicate the event that was detected. 1-4 are the ‘btn1-btn4’ events, 5 and 6 are the ‘light’ and ‘pulse’ events, 7 is the ‘trigger’ event, 9 is a requested timestamp event (see Clock()).

‘dir’ is the direction of the event eg ‘up’ or ‘down’, trigger is described as ‘on’ when low.

‘dir’ is set to ‘time’ if a requested timestamp event has been detected.

‘time’ is the timestamp associated with the event.

Values can be read as a list of structures eg:
res = getRTBoxResponses(3)
res[0].dir, res[0].button, res[0].time

or dictionaries:

res[0]['dir'], res[0]['button'], res[0]['time']

Note even if only 1 key press was requested a list of dict / objects is returned.

Note that the firmware in Bits# units varies over time and some features of this class may not work for all firmware versions. Also Bits# units can be configured in various ways via their config.xml file so this class makes certain assumptions about the configuration. Such variations may affect key mappings for RTBox commands.

getResponse (length=1, timeout=0.1)
Read the latest response from the serial port

Params

length determines whether we expect: 1: a single-line reply (use readline()) 2: a multiline reply (use readlines() which requires timeout) -1: may not be any EOL character; just read whatever chars are there

getStatus (N=0)
Pulls out the Nth entry in the statusValues list.

Returns a dict like object with the following entries sample, time, trigIn, DIN[10], DWORD, IR[6], ADC[6]
sample is the sample ID number. time is the time stamp. trigIn is the value of the trigger input. DIN is a list of 10 digital input values. DWORD represents the digital inputs as a single decimal value. IR is a list of 10 infra-red (IR) input values. ADC is a list of 6 analog input values. These can be accessed as value['sample'] or value.sample, values.ADC[j].

All sources are numbered from zero. Din 0 … 9 IR 0 … 5 ADC 0 … 5

Example

bits.startStatusLog() while not event
    #do some processing continue

bits.stopStatusLog() res=getStatus(20) print(bits.res.time)

Note that the firmware in Bits# units varies over time and some features of this class may not work for all firmware versions. Also Bits# units can be configured in various ways via their config.xml file so this class makes certain assumptions about the configuration. In particular it is assumed that all digital inputs, triggers and analog inputs are reported as part of status updates. If some of these report are disabled in your config.xml file then ‘status’ and ‘event’ commands in this class may not work.

getStatusBoxResponse ()
checks for one statusBox style key presses on the input buffer then reads it. Returns a dict like object with three members ‘button’, ‘dir’ and ‘time’

‘button’ is a number from 1 to 9 to indicate the event that was detected. 1-17 are the ‘btn1-btn17’ events.
‘dir’ is the direction of the event eg ‘up’ or ‘down’, trigger is described as ‘on’ when low.
‘dir’ is set to ‘time’ if a requested timestamp event has been detected.
‘time’ is the timestamp associated with the event.
Value can be read as a structure, eg:

```python
res = getRTBoxResponse() res.dir, res.button, res.time
```

or dictionary
```
res['dir'], res['button'], res['time']
```

Note that the firmware in Bits# units varies over time and some features of this class may not work for all firmware versions. Also Bits# units can be configured in various ways via their config.xml file so this class makes certain assumptions about the configuration. In particular it is assumed that all digital inputs, triggers and analog inputs are reported as part of status updates. If some of these report are disabled in your config.xml file then ‘status’ and ‘event’ commands in this class may not work.

### getStatusBoxResponses (N=1)

Checks for (at least) an appropriate number of RTBox style key presses on the input buffer then reads them. Returns a list of dict like objects with three members ‘button’, ‘dir’ and ‘time’.

- ‘button’ is a number from 1 to 9 to indicate the event that was detected. 1-4 are the ‘btn1-btn4’ events, 5 and 6 are the ‘light’ and ‘pulse’ events, 7 is the ‘trigger’ event, 9 is a requested timestamp event (see Clock()).
- ‘dir’ is the direction of the event eg ‘up’ or ‘down’, trigger is described as ‘on’ when low.
- ‘dir’ is set to ‘time’ if a requested timestamp event has been detected.
- ‘time’ is the timestamp associated with the event.

Values can be read as a list of structures eg:

```python
res = getStatusBoxResponses(3)
print(res[0].dir, res[0].button, res[0].time)
```

or dictionaries:

```python
print(res[0]['dir'], res[0]['button'], res[0]['time'])
```

Note even if only 1 key press was requested a list of dict / objects is returned.

Note that the firmware in Bits# units varies over time and some features of this class may not work for all firmware versions. Also Bits# units can be configured in various ways via their config.xml file so this class makes certain assumptions about the configuration. In particular it is assumed that all digital inputs, triggers and analog inputs are reported as part of status updates. If some of these report are disabled in your config.xml file then ‘status’ and ‘event’ commands in this class may not work.

### getStatusEvent (N=0)

Pulls out the Nth event from the status event list.

Returns a dictionary like object with the following entries source, input, direction, time.

- source = the general source of the event - e.g. DIN for Digital input, IR for IT response box.
- input = the individual input in the source. direction = ‘up’ or ‘down’ time = time stamp.
- All sources are numbered from zero. Din 0 … 9 IR 0 … 5 ADC 0 … 5
- mode specifies which directions of events are captured, e.g ‘up’ will only report up events.
- The data can be accessed as value['time'] or value.time
Example

bits.startStatusLog() while not event
    # do some processing continue
bits.stopStatusLog() res=getAllStatusEvents(20) print(bits.res.time)

Note that the firmware in Bits# units varies over time and some features of this class may not work for all firmware versions. Also Bits# units can be configured in various ways via their config.xml file so this class makes certain assumptions about the configuration. In particular it is assumed that all digital inputs, triggers and analog inputs are reported as part of status updates. If some of these report are disabled in your config.xml file then 'status' and 'event' commands in this class may not work.

getTrigIn (N=0)
Pulls out the values of the trigger input for the Nth status entry.
Returns dictionary with a 0 or 1 (trigIn) and a time stamp (time)

Example

bits.pollStatus() res=bits.getAnalog() print(res['trigIn'])

will poll the status display the value of the trigger input.

Note that the firmware in Bits# units varies over time and some features of this class may not work for all firmware versions. Also Bits# units can be configured in various ways via their config.xml file so this class makes certain assumptions about the configuration. In particular it is assumed that all digital inputs, triggers and analog inputs are reported as part of status updates. If some of these report are disabled in your config.xml file then 'status' and 'event' commands in this class may not work.

getAddressLine (lineN, nPixels, timeout=10.0, nAttempts=10)
Return the r,g,b values for a number of pixels on a particular video line :param lineN: the line number you want to read :param nPixels: the number of pixels you want to read :param nAttempts: the first time you call this function it has
to get to status mode. In this case it sometimes takes a few attempts to make the call work

Returns an Nx3 numpy array of uint8 values

isAwake()
Test whether we have an active connection on the virtual serial port

property isOpen
longName = ''

property mode
Get/set the mode of the BitsSharp to one of: “bits++” “mono++” “color++” “status” “storage” “auto”

property monitorEDID
Get / set the EDID file for the monitor. The edid files will be located in the EDID subdirectory of the flash disk. The file automatic.edid will be the file read from the connected monitor.

name = b'CRS Bits#'

pause()
Pause for a default period for this device
**pollStatus** \((t=0.0001)\)

Reads the status reports from the Bits# for the specified usually short time period \(t\). The script will wait for this time to lapse so not ideal for time critical applications.

If \(t\) is less than 0.01 polling will continue until at least 1 data entry has been recorded.

If you don’t want to wait while this does its job use startStatusLog and stopStatusLog instead.

Fills the statusValues list with all the status values read during the time period.

Fills the statusEvents list with just those status values that are likely to be meaningful events.

the members statusValues and statusEvents will end up containing dict like objects of the following style: sample, time, trigIn, DIN[10], DWORD, IR[6], ADC[6]

They can be accessed as statusValues[i]['sample'] or statusValues[i].sample, statusValues[x].ADC[j].

Example:

```python
bits.pollStatus()
print(bits.statusValues[0].IR[0])
```

will display the value of the IR InputA in the first sample recorded.

Note: Starts and stops logging for itself.

Note that the firmware in Bits# units varies over time and some features of this class may not work for all firmware versions. Also Bits# units can be configured in various ways via their config.xml file so this class makes certain assumptions about the configuration. In particular it is assumed that all digital inputs, triggers and analog inputs are reported as part of status updates. If some of these report are disabled in your config.xml file then ‘status’ and ‘event’ commands in this class may not work.

**primeClock ()**

Primes the clock to reset at the next screen flip - note only 1 clock reset signal will be issued but if the frame(s) after the reset frame is dropped the reset will be re-issued thus keeping timing good.

Resets continue to be issued on each video frame until the next win.flip so you need to have regular win.flips for this function to work properly.

**Example**  
bits.primeClock() drawImage while not response

```python
#do some processing bits.win.flip()
```

Will get a clock reset signal ready but wont issue it until the first win.flip in the loop.

**read** \((timeout=0.1)\)

Get the current waiting characters from the serial port if there are any.

Mostly used internally but may be needed by user. Note the return message depends on what state the device is in and will need to be decoded. See the Bits# manual but also the other functions herein that do the decoding for you.
Example

message = bits.read()

reset()
Deprecated: This was used on the old Bits++ to power-cycle the box. It required the compiled dll, which only worked on windows and doesn't work with Bits# or Display++.

resetClock()
Issues a clock reset code using 1 screen flip if the next frame(s) is dropped the reset will be re-issued thus keeping timing good.
Resets continue to be issued on each video frame until the next win.flip so you need to have regular win.flips for this function to work properly.

Example

bits.resetClock() drawImage() bits.win.flip()
Will issue clock resets while the image is being drawn then display the inmage and allow the clock to continue formt the same frame.

Example

bits.resetClock() bits.RTBoxWait() bits.win.flip()
Will issue clock resets until a button is presses.

sendAnalog(AOUT1=0, AOUT2=0)
sends a single analog output pulse uses up 1 win flip. pulse will continue until next win flip called. Actions are always 1 frame behind the request.
May conflict with trigger and goggle settings.

Example

bits.sendAnalog(4.5,-2.0) bits.win.flip()

sendMessage(message, autoLog=True)
Send a command to the device (does not wait for a reply or sleep())

sendTrigger(triggers=0, onTime=0, duration=0, mask=65535)
Sends a single trigger using up 1 win.flip. The trigger will be sent on the following frame.
The triggers will continue until after the next win.flip.
Actions are always 1 frame after the request.
May do odd things if Goggles and Analog are also in use.
Example:

```
bits.sendTrigger(0b0000000010, 2.0, 4.0)
b bits.win.flip()
```

Will send a 4ms pulse on DOUT1 2ms after the start of the frame. Due to the following win.flip() the pulse should last for 1 frame only.
Triggers will continue until stopTrigger is called.
**setAnalog** *(AOUT1=0, AOUT2=0)*

Sets up Analog outputs in Bits#. AOUT1 and AOUT2 are the two analog values required in volts. Analog commands are issued at the next win.flip() and actioned 1 video frame later.

**Example**

```python
bits.setAnalog(4.5,-2.2) bits.startAnalog() bits.win.flip()
```

**setContrast** *(contrast, LUTRange=1.0, gammaCorrect=None)*

Set the contrast of the LUT for ‘bits++’ mode only. Parameters:

- `contrast` [float in the range 0:1] The contrast for the range being set
- `LUTRange` [float or array] If a float is given then this is the fraction of the LUT to be used. If an array of floats is given, these will specify the start / stop points as fractions of the LUT. If an array of ints (0-255) is given these determine the start stop indices of the LUT

**Examples**

```python
setContrast(1.0, 0.5) will set the central 50% of the LUT so that a stimulus with contr=0.5 will actually be drawn with contrast 1.0

setContrast(1.0, [0.25, 0.5]) setContrast(1.0, [63, 127])

will set the lower-middle quarter of the LUT (which might be useful in LUT animation paradigms)
```

**setGamma** *(newGamma)*

Set the LUT to have the requested gamma value. Currently also resets the LUT to be a linear contrast ramp spanning its full range. May change this to read the current LUT, undo previous gamma and then apply new one?

**setLUT** *(newLUT=None, gammaCorrect=False, LUTRange=1.0, contrast=None)*

SetLUT is only really needed for bits++ mode of bits# to set the look-up table (256 values with 14 bits each). For the BitsPlusPlus device the default is to perform gamma correction here but on the BitsSharp it seems better to have the device perform that itself as the last step so gamma correction is off here by default. If no contrast has yet been set (it isn’t needed for other modes) then it will be set to 1 here.

**setRTBoxMode** *(mode=['CB6', 'Down', 'Trigger'])*

Sets the RTBox mode data member - does not actually set the RTBox into this mode.

**Example**

```python
bits.setRTBoxMode(['CB6', 'Down']) # set the mode bits.RTBoxEnable() # Enable RTBox emulation with # the preset mode.

sets the RTBox mode settings for a CRS CB6 button box. and for detection of ‘Down’ events only.
```

**setStatusBoxMode** *(mode=['CB6', 'Down', 'Trigger', 'Analog'])*

Sets the statusBox mode data member - does not actually set the statusBox into this mode.
Example

bits.setStatusBoxMode(['CB6','Down']) # set the mode
bits.statusBoxEnable() # Enable status Box emulation
with # the preset mode.

sets the statusBox mode settings for a CRS CB6 button box. and for detection of ‘Down’ events only.

Note that the firmware in Bits# units varies over time and some features of this class may not work for all firmware versions. Also Bits# units can be configured in various ways via their config.xml file so this class makes certain assumptions about the configuration. In particular it is assumed that all digital inputs, triggers and analog inputs are reported as part of status updates. If some of these report are disabled in your config.xml file then ‘status’ and ‘event’ commands in this class may not work.

**setStatusBoxThreshold** *(threshold=None)*

Sets the threshold by which analog inputs must change to trigger a button press event. If None the threshold will be set very high so that no such events are triggered.

Can be used to change the threshold for analog events without having to re enable the status box system as a whole.

Note that the firmware in Bits# units varies over time and some features of this class may not work for all firmware versions. Also Bits# units can be configured in various ways via their config.xml file so this class makes certain assumptions about the configuration. In particular it is assumed that all digital inputs, triggers and analog inputs are reported as part of status updates. If some of these report are disabled in your config.xml file then ‘status’ and ‘event’ commands in this class may not work.

**setStatusEventParams** *(DINBase=1023, IRBase=63, TrigInBase=0, ADCBase=0, threshold=9999.99, mode=['up', 'down'])*

Sets the parameters used to determine if a status value represents a reportable event.

DIN_base = a 10 bit binary word specifying the expected starting values of the 10 digital input lines

IR_base = a 6 bit binary word specifying the expected starting values of the 6 CB6 IR buttons

Trig_base = the starting value of the Trigger input

mode = a list of event types to monitor can be ‘up’ or ‘down’ typically ‘down’ corresponds to a button press or when the input is being pulled down to zero volts.

Example:

```
bits.setStatusEventParams(DINBase=0b1111111111,
                          IRBase=0b111111,
                          TrigInBase=0,
                          ADCBase=0,
                          threshold = 3.4,
                          mode = ['down'])

bits.startStatusLog()
while not event
    #do some processing
    continue
bits.stopStatusLog()
res=getAllStatusEvents(0)
print(bits.res.time)
```

This ill start the event extraction process as if DINs and IRs are all ‘1’, Trigger is ‘0’ ADCs = 0 with an ADC threshold for change of 3.4 volts, and will only register ‘down’ events. Here we display the time stamp of the first event.

Note that the firmware in Display++ units varies over time and some features of this class may not work for all firmware versions. Also Display++ units can be configured in various ways via their config.xml file so this
class makes certain assumptions about the configuration. In particular it is assumed that all digital inputs, 
triggers and analog inputs are reported as part of status updates. If some of these report are disabled in your 
config.xml file then 'status' and 'event' commands in this class may not work.

```python
setTrigger (triggers=0, onTime=0, duration=0, mask=65535)
```
Quick way to set up triggers.

Triggers is a binary word that determines which triggers will be turned on.

onTime specifies the start time of the trigger within the frame (in S with 100uS resolution)

Duration specifies how long the trigger will last. (in S with 100uS resolution).

Note that mask only protects the digital output lines set by other activities in the Bits. Not other triggers.

Example:

```python
  bits.setTrigger(0b0000000010, 2.0, 4.0, 0b0111111111) bits.
  startTrigger() 
```
Will issue a 4ms long high-going pulse, 2ms after the start of each frame on DOUT1 while protecting the 
value of DOUT 9.

```python
setTriggerList (triggerList=None, mask=65535)
```
Overload of Bits# and Display++ Sets up Trigger pulses via the list method while preserving the analog 
output settings.

Sets up Trigger pulses in Bist++ using the fine grained method that can control every trigger line at 100uS 
intervals.

TriggerList should contain 1 entry for every 100uS packet (see getPackets) the binary word in each entry 
specifies which trigger line will be active during that time slot.

Note that mask only protects the digital output lines set by other activities in the Bits. Not other triggers.

Example

```python
  packet = [0]*self._NumberPackets packet[0] = 0b0000000010 bits.setTriggerList(packet)
```
Will sens a 100us pulse on DOUT1 at the start of the frame.

```
            0b0000000001 bits.setTriggerList(packet) bits.startTrigger()
```
Will sens a 100us pulse on DOUT1 1000us after the start of the frame and a second 100us pulse on DOUT0 
2000us after the start of the frame.

Triggers will continue until stopTrigger is called.

```python
start ()
```
[Not currently implemented] Used to begin event collection by the device.

Not really needed as other members now do this.

```python
startAnalog ()
```
will start sending analog signals on the next win flip and continue until stopped.
Example

```python
t = 60
startStatusLog(t=60)
```

Note that the firmware in Bits# units varies over time and some features of this class may not work for all firmware versions. Also Bits# units can be configured in various ways via their config.xml file so this class makes certain assumptions about the configuration. In particular it is assumed that all digital inputs, triggers and analog inputs are reported as part of status updates. If some of these report are disabled in your config.xml file then ‘status’ and ‘event’ commands in this class may not work.

**startTrigger()**

Start sending triggers on the next win flip and continue until stopped by stopTrigger Triggers start 1 frame after the frame on which the first trigger is sent.

Example:

```python
bits.setTrigger(0b0000000010, 2.0, 4.0, 0b0111111111)
bits.startTrigger()
while imageOn:
    #do some processing
    continue
bits.stopTrigger()
bits.win.flip()
```

**statusBoxAddKeys (map)**

Add key mappings to an existing map. statusBox events can be mapped to a number of physical events on Bits# They can be mapped to digital input lines, triggers and CB6 IR input channels. The format for map is a list of tuples with each tuple containing the name of the RTBox button to be mapped and its source eg (‘btn1’,’Din1’) maps physical input Din1 to logical button btn1. statusBox has 23 logical buttons (btn1-23). Unlike RTBox buttons/events can only be partially mapped to multiple physical inputs. That is a logical button can be mapped to more than 1 physical input but a physical input can onlly be mapped to 1 logical button. So, this function over write any existing mappings if the physical input is the same.

Example:

```python
bits.RTBoxSetKeys([('btn1','Din0'),('btn2','Din1')])
bits.RTBoxAddKeys([('btn1','IRButtonA'),('btn2','IRButtonB')])
```

Will link Din0 to button 1 and Din1 to button 2. Then adds IRButtonA and IRButtonB alongside the original mappings.

Now both hard wired and IR inputs will emulate the same logical button press.

To match with the CRS hardware description inputs are labelled as follows.

TrigIn, Din0 … Din9, IRButtonA … IRButtonF, AnalogIn1 … AnalogIn6

Logical buttons are numbered from 1 to 23.

Note that the firmware in Bits# units varies over time and some features of this class may not work for all firmware versions. Also Bits# units can be configured in various ways via their config.xml file so this class makes certain assumptions about the configuration. In particular it is assumed that all digital inputs, triggers and analog inputs are reported as part of status updates. If some of these report are disabled in your config.xml file then ‘status’ and ‘event’ commands in this class may not work.

**statusBoxDisable()**

Disables the detection of statusBox events. This is useful to stop the Bits# from reporting key presses When you no longer need them. And must be done before using any other data logging methods.

It undoes any button - input mappings

**statusBoxEnable (mode=None, map=None, threshold=None)**

Sets up the statusBox with preset or bespoke mappings and enables event detection.
statusBox events can be mapped to a number of physical events on Bits#. They can be mapped to digital input lines, tigers and CB6 IR input channels.

mode is a list of strings. Preset mappings provided via mode:

- CB6 for the CRS CB6 IR response box connected mapped to btn1-6 IO for a three button box connected to Din0-2 mapped to btn1-3 IO6 for a six button box connected to Din0-5 mapped to btn1-6 IO10 for a ten button box connected to Din0-9 mapped to btn1-10 Trigger maps the trigIn to btn17 Analog maps the 6 analog inputs on a Bits# to btn18-23

if CB6 and IOx are used together the Dins are mapped from btn7 onwards.

If mode = None or is not set then the value of self.statusBoxMode is used.

Bespoke Mappings overwrite preset ones.

The format for map is a list of tuples with each tuple containing the name of the button to be mapped and its source eg ('btn1', 'Din0') maps physical input Din0 to logical button btn1.

Note the lowest number button event is Btn1

statusBox has 23 logical buttons (btn1-123). Buttons/events can be mapped to multiple physical inputs and stay mapped until reset.

mode is a string or list of strings that contains keywords to determine present mappings and modes for statusBox.

If mode includes ‘Down’ button events will be detected when pressed. If mode includes ‘Up’ button events will be detected when released. You can detect both types of event noting that the event detector will look for transitions and ignore what it sees as the starting state.

To match with the CRS hardware description inputs are labelled as follows.
TrigIn, Din0… Din9, IRButtonA … IRButtonF, AnalogIn1 … AnalogIn6
Logical buttons are numbered from 1 to 23.

threshold sets the threshold by which analog inputs must change to trigger a button press event. If None the threshold will be set very high so that no such events are triggered. Analog inputs must cycle up and down by threshold to be detected as separate events. So if only ‘Up’ events are detected the input must go up by threshold, then come down again and then go back up to register 2 up events.

Example

bits.statusBoxEnable(mode = ‘Down’), map = [('btn1', 'Din0'), ('btn2', 'Din1')]
enable the statusBox to detect Down events on buttons 1 and 2 where they are mapped to DIN0 and DIN1.

Example

bits.statusBoxEnable(mode = ['Down', 'CB6'])
enable the status Box emulation to detect Down events on the standard CB6 IR response box keys.

Note that the firmware in Bits# units varies over time and some features of this class may not work for all firmware versions. Also Bits# units can be configured in various ways via their config.xml file so this class makes certain assumptions about the configuration. In particular it is assumed that all digital inputs, triggers and analog inputs are reported as part of status updates. If some of these report are disabled in your config.xml file then ‘status’ and ‘event’ commands in this class may not work.
**statusBoxKeysPressed**

\((N=1)\)

Check to see if (at least) the appropriate number of RTBox style key presses have been made.

**Example**

```python
bits.statusBoxKeysPressed(5)
```

will return false until 5 button presses have been recorded.

Note that the firmware in Bits# units varies over time and some features of this class may not work for all firmware versions. Also Bits# units can be configured in various ways via their config.xml file so this class makes certain assumptions about the configuration. In particular it is assumed that all digital inputs, triggers and analog inputs are reported as part of status updates. If some of these report are disabled in your config.xml file then ‘status’ and ‘event’ commands in this class may not work.

**statusBoxResetKeys**

**statusBoxSetKeys** *(map)*

Set key mappings: first resets existing then adds new ones. Does not reset any event that is not in the new list.

statusBox events can be mapped to a number of physical events on Bits#. They can be mapped to digital input lines, triggers and CB6 IR input channels. The format for map is a list of tuples with each tuple containing the name of the RTBox button to be mapped and its source eg (‘btn1’,’Din1’) maps physical input Din1 to logical button btn1.

statusBox has 17 logical buttons (btn1-17) Buttons/events can be mapped to multiple physical inputs and stay mapped until reset.

**Example**

```python
bits.RTBoxSetKeys([(‘btn1’,‘Din0’), (‘btn2’,‘IRButtonA’)])
```

Will link physical Din0 to logical button 1 and IRButtonA to button 2.

To match with the CRS hardware description inputs are labelled as follows.

TrigIn, Din0 … Din9, IRButtonA … IRButtonF, AnalogIn1 … AnalogIn6

Logical buttons are numbered from 1 to 23.

Note that the firmware in Bits# units varies over time and some features of this class may not work for all firmware versions. Also Bits# units can be configured in various ways via their config.xml file so this class makes certain assumptions about the configuration. In particular it is assumed that all digital inputs, triggers and analog inputs are reported as part of status updates. If some of these report are disabled in your config.xml file then ‘status’ and ‘event’ commands in this class may not work.

**statusBoxWait**

Waits until (at least) one of RTBox style key presses have been made Pauses program execution in mean time.
Example

res = bits.statusBoxWait()

will suspend all other activity until 1 button press has been recorded and will then return a dict / structure containing results.

Results can be accessed as follows:

structure  res.dir, res.button, res.time

or dictionary  res['dir'], res['button'], res['time']

Note that the firmware in Bits# units varies over time and some features of this class may not work for all firmware versions. Also DBits# units can be configured in various ways via their config.xml file so this class makes certain assumptions about the configuration. In particular it is assumed that all digital inputs, triggers and analog inputs are reported as part of status updates. If some of these report are disabled in your config.xml file then ‘status’ and ‘event’ commands in this class may not work.

statusBoxWaitN(N=1)

Waits until (at least) the appropriate number of RTBox style key presses have been made Pauses program execution in mean time.

Example

res = bits.statusBoxWaitN(5)

will suspend all other activity until 5 button presses have been recorded and will then return a list of Dicts containing the 5 results.

Results can be accessed as follows:

structure:

res[0].dir, res[0].button, res[0].time

or dictionary:

res[0]['dir'], res[0]['button'], res[0]['time']

Note that the firmware in Bits# units varies over time and some features of this class may not work for all firmware versions. Also Bits# units can be configured in various ways via their config.xml file so this class makes certain assumptions about the configuration. In particular it is assumed that all digital inputs, triggers and analog inputs are reported as part of status updates. If some of these report are disabled in your config.xml file then ‘status’ and ‘event’ commands in this class may not work.

stop()

[Not currently implemented] Used to stop event collection by the device.

Not really needed as other members now do this.

stopAnalog()

bits.win.flip()

stopGoggles()

Stop the stereo goggles from toggling

Example::  bits.startGoggles(0,1) bits.win.flip() while not response

bits.win.flip() #do some processing

bits.stopGoggles() bits.win.flip()
Starts toggling the goggles with the right eye open in sync with the first win.flip(0) within the loop. The open eye will alternate.

Note it is safer to leave the goggles toggling forever, ie to never call stopGoggles().

**stopStatusLog()**

Stop logging data from the Bits# and extracts the raw status values and significant events and puts them in statusValues and statusEvents.

statusValues will end up containing dict like objects of the following style:

```
sample, time, trigIn, DIN[10], DWORD, IR[6], ADC[6]
```

They can be accessed as statusValues[i]['sample'] or statusValues[i].sample, statusValues[x].ADC[j].

StatusEvents will end up containing dict like objects of the following style:

```
source, input, direction, time.
```

The data can be accessed as statusEvents[i]['time'] or statusEvents[i].time

Waits for _statusLog to finish properly so can introduce a timing delay.

Example:

```python
bits.startStatusLog()
while not event
    #do some processing
    continue
bits.stopStatusLog()
print(bits.statusValues[0].time)
print(bits.statusEvents[0].time)
```

Will display the time stamps of the first status value recorded and the first meaningful event.

Note that the firmware in Bits# units varies over time and some features of this class may not work for all firmware versions. Also Bits# units can be configured in various ways via their config.xml file so this class makes certain assumptions about the configuration. In particular it is assumed that all digital inputs, triggers and analog inputs are reported as part of status updates. If some of these report are disabled in your config.xml file then ‘status’ and ‘event’ commands in this class may not work.

**stopTrigger()**

Stop sending triggers at the next win flip.

Example::

```python
bits.setTrigger(0b0000000010, 2.0, 4.0, 0b0111111111) bits.startTrigger() while imageOn:
    # do some processing continue
bits.stopTrigger() bits.win.flip()
```

**syncClocks(t)**

Synchronise the Bits/RTBox Clock with the host clock Given by t.

**property temporalDithering**

Temporal dithering can be set to True or False

**property win**

The window that this box is attached to
For display calibration

**ColorCAL**

Attributes

```
class ColorCAL([port, maxAttempts])  # A class to handle the CRS Ltd ColorCAL device
```

Details

A class to handle the CRS Ltd ColorCAL device

Open serial port connection with Colorcal II device

**Usage**

```python
cc = ColorCAL(port, maxAttempts)
```

If no port is provided then the following defaults will be tried:

- /dev/cu.usbmodem0001 (OSX)
- /dev/ttyACM0
- COM3 (windows)

**calibrateZero()**

Perform a calibration to zero light.

For early versions of the ColorCAL this had to be called after connecting to the device. For later versions the dark calibration was performed at the factory and stored in non-volatile memory.

You can check if you need to run a calibration with:

```
ColorCAL.getNeedsCalibrateZero()
```

**driverFor = ['colorcal']**

**getCalibMatrix()**

Get the calibration matrix from the device, needed for transforming measurements into real-world values.

This is normally retrieved during `__init__` and stored as `ColorCal.calibMatrix` so most users don’t need to call this function.

**getInfo()**

Queries the device for information

**usage::**

```
(ok, serialNumber, firmwareVersion, firmwareBuild) = colorCal.getInfo()
```

`ok` will be True/False Other values will be a string or None.

**getLum()**

Conducts a measurement and returns the measured luminance

**Note:** The luminance is always also stored as .lastLum
getNeedsCalibrateZero()
Check whether the device needs a dark calibration

In initial versions of CRS ColorCAL mkII the device stored its zero calibration in volatile memory and needed to be calibrated in darkness each time you connected it to the USB

This function will check whether your device requires that (based on firmware build number and whether you've already done it since python connected to the device).

Returns  True or False

longName = 'CRS ColorCAL'

measure()
Conduct a measurement and return the X,Y,Z values

Usage:

ok, X, Y, Z = colorCal.measure()

Where:  ok is True/False X, Y, Z are the CIE coordinates (Y is luminance in cd/m**2)

Following a call to measure, the values ColorCAL.lastLum will also be populated with, for compatibility with other devices used by PsychoPy (notably the PR650/PR655)

readline (size=None, eol='
\r')
This should be used in place of the standard serial.Serial.readline() because that doesn't allow us to set the eol character

sendMessage (message, timeout=0.1)
Send a command to the photometer and wait an allotted timeout for a response.

8.4.5 egi (pynetstation)

Interface to EGI Netstation

This is currently a simple import of pynetstation which is now simply called egi on pypi.

egi is included in Standalone distributions of PsychoPy but you can install it with:

pip install egi

For examples on usage see the example_simple and example_multi files on the egi github repository

For an example see the demos menu of the PsychoPy Coder For further documentation see the pynetstation website

8.4.6 Launch an fMRI experiment: Test or Scan

Idea: Run or debug an experiment script using exactly the same code, i.e., for both testing and online data acquisition. To debug timing, you can emulate sync pulses and user responses. Limitations: pyglet only; keyboard events only.

class psychopy.hardware.emulator.ResponseEmulator (simResponses=None)
Class to allow simulation of a user's keyboard responses during a scan.

Given a list of response tuples (time, key), the thread will simulate a user pressing a key at a specific time (relative to the start of the run).

Author: Jeremy Gray; Idea: Mike MacAskill
_delete()
Remove current thread from the dict of currently running threads.

_set_tstate_lock()
Set a lock object which will be released by the interpreter when the underlying thread state (see pystate.h) gets deleted.

**property daemon**
A boolean value indicating whether this thread is a daemon thread.

This must be set before start() is called, otherwise RuntimeError is raised. Its initial value is inherited from the creating thread; the main thread is not a daemon thread and therefore all threads created in the main thread default to daemon = False.

The entire Python program exits when no alive non-daemon threads are left.

**property ident**
Thread identifier of this thread or None if it has not been started.

This is a nonzero integer. See the get_ident() function. Thread identifiers may be recycled when a thread exits and another thread is created. The identifier is available even after the thread has exited.

**isAlive()**
Return whether the thread is alive.

This method returns True just before the run() method starts until just after the run() method terminates. The module function enumerate() returns a list of all alive threads.

**is_alive()**
Return whether the thread is alive.

This method returns True just before the run() method starts until just after the run() method terminates. The module function enumerate() returns a list of all alive threads.

**join**(timeout=None)
Wait until the thread terminates.

This blocks the calling thread until the thread whose join() method is called terminates – either normally or through an unhandled exception or until the optional timeout occurs.

When the timeout argument is present and not None, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof). As join() always returns None, you must call isAlive() after join() to decide whether a timeout happened – if the thread is still alive, the join() call timed out.

When the timeout argument is not present or None, the operation will block until the thread terminates.

A thread can be join()ed many times.

join() raises a RuntimeError if an attempt is made to join the current thread as that would cause a deadlock. It is also an error to join() a thread before it has been started and attempts to do so raises the same exception.

**property name**
A string used for identification purposes only.

It has no semantics. Multiple threads may be given the same name. The initial name is set by the constructor.

**run()**
Method representing the thread’s activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object’s constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.
**start()**

Start the thread’s activity.

It must be called at most once per thread object. It arranges for the object’s run() method to be invoked in a separate thread of control.

This method will raise a RuntimeError if called more than once on the same thread object.

class psychopy.hardware.emulator.SyncGenerator(TR=1.0, TA=1.0, volumes=10, sync='5',
skip=0, sound=False, **kwargs)

Class for a character-emitting metronome thread (emulate MR sync pulse).

Aim: Allow testing of temporal robustness of fMRI scripts by emulating a hardware sync pulse. Adds an arbitrary ‘sync’ character to the key buffer, with sub-millisecond precision (less precise if CPU is maxed). Recommend: TR=1.000 or higher and less than 100% CPU. Shorter TR -> higher CPU load.

Parameters

- **TR** – seconds between volume acquisitions
- **TA** – seconds to acquire one volume
- **volumes** – number of 3D volumes to obtain in a given scanning run
- **sync** – character used as flag for sync timing, default='5'
- **skip** – how many frames to silently omit initially during T1 stabilization, no sync pulse. Not needed to test script timing, but will give more accurate feel to start of run. aka “discdacqs”.
- **sound** – simulate scanner noise

_delete() Remove current thread from the dict of currently running threads.

_set_tstate_lock() Set a lock object which will be released by the interpreter when the underlying thread state (see pystate.h) gets deleted.

**property daemon**

A boolean value indicating whether this thread is a daemon thread.

This must be set before start() is called, otherwise RuntimeError is raised. Its initial value is inherited from the creating thread; the main thread is not a daemon thread and therefore all threads created in the main thread default to daemon = False.

The entire Python program exits when no alive non-daemon threads are left.

**property ident**

Thread identifier of this thread or None if it has not been started.

This is a nonzero integer. See the get_ident() function. Thread identifiers may be recycled when a thread exits and another thread is created. The identifier is available even after the thread has exited.

isAlive() Return whether the thread is alive.

This method returns True just before the run() method starts until just after the run() method terminates. The module function enumerate() returns a list of all alive threads.

is_alive() Return whether the thread is alive.

This method returns True just before the run() method starts until just after the run() method terminates. The module function enumerate() returns a list of all alive threads.
join(timeout=None)
Wait until the thread terminates.

This blocks the calling thread until the thread whose join() method is called terminates – either normally or through an unhandled exception or until the optional timeout occurs.

When the timeout argument is present and not None, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof). As join() always returns None, you must call isAlive() after join() to decide whether a timeout happened – if the thread is still alive, the join() call timed out.

When the timeout argument is not present or None, the operation will block until the thread terminates.

A thread can be join()ed many times.

join() raises a RuntimeError if an attempt is made to join the current thread as that would cause a deadlock.

It is also an error to join() a thread before it has been started and attempts to do so raises the same exception.

property name
A string used for identification purposes only.

It has no semantics. Multiple threads may be given the same name. The initial name is set by the constructor.

run()
Method representing the thread’s activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object’s constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

start()
Start the thread’s activity.

It must be called at most once per thread object. It arranges for the object’s run() method to be invoked in a separate thread of control.

This method will raise a RuntimeError if called more than once on the same thread object.

psychopy.hardware.emulator.launchScan(win, settings, globalClock=None, simResponses=None, mode=None, esc_key='escape', instr='select Scan or Test, press enter', wait_msg='waiting for scanner...', wait_timeout=300, log=True)

Accepts up to four fMRI scan parameters (TR, volumes, sync-key, skip), and launches an experiment in one of two modes: Scan, or Test.

Usage See Coder Demo -> experiment control -> fMRI_launchScan.py.

In brief: 1) from psychopy.hardware.emulator import launchScan; 2) Define your args; and 3) add 'vol = launchScan(args)' at the top of your experiment script.

launchScan() waits for the first sync pulse and then returns, allowing your experiment script to proceed. The key feature is that, in test mode, it first starts an autonomous thread that emulates sync pulses (i.e., emulated by your CPU rather than generated by an MRI machine). The thread places a character in the key buffer, exactly like a keyboard event does. launchScan will wait for the first such sync pulse (i.e., character in the key buffer). launchScan returns the number of sync pulses detected so far (i.e., 1), so that a script can account for them explicitly.

If a globalClock is given (highly recommended), it is reset to 0.0 when the first sync pulse is detected. If a mode was not specified when calling launchScan, the operator is prompted to select Scan or Test.

If scan mode is selected, the script will wait until the first scan pulse is detected. Typically this would be coming from the scanner, but note that it could also be a person manually pressing that key.

If test mode is selected, launchScan() starts a separate thread to emit sync pulses / key presses. Note that this thread is effectively nothing more than a key-pressing metronome, emitting a key at the start of every TR, doing so with high temporal precision.
If your MR hardware interface does not deliver a key character as a sync flag, you can still use launchScan() to test script timing. You have to code your experiment to trigger on either a sync character (to test timing) or your usual sync flag (for actual scanning).

**Parameters**

- **win**: a `Window` object (required)
- **settings**: [a dict containing up to 5 parameters] (2 required: TR, volumes)
  - **TR**: seconds per whole-brain volume (minimum value = 0.1s)
  - **volumes**: number of whole-brain (3D) volumes to obtain in a given scanning run.
  - **sync**: (optional) key for sync timing, default = ‘5’.
  - **skip**: (optional) how many volumes to silently omit initially (during T1 stabilization, no sync pulse). default = 0.
  - **sound**: (optional) whether to play a sound when simulating scanner sync pulses
  - **globalClock**: optional but highly recommended `Clock` to be used during the scan; if one is given, it is reset to 0.000 when the first sync pulse is received.
  - **simResponses**: optional list of tuples [(time, key), (time, key), ...]. time values are seconds after the first scan pulse is received.
  - **esc_key**: key to be used for user-interrupt during launch. default = ‘escape’
  - **mode**: if mode is ‘Test’ or ‘Scan’, launchScan() will start in that mode.
  - **instr**: instructions to be displayed to the scan operator during mode selection.
  - **wait_msg**: message to be displayed to the subject while waiting for the scan to start (i.e., after operator indicates start but before the first scan pulse is received).
  - **wait_timeout**: time in seconds that launchScan will wait before assuming something went wrong and exiting. Defaults to 300 sec (5 min). Raises a RuntimeError if no sync pulse is received in the allowable time.

```python
class psychopy.hardware.emulator.ResponseEmulator(simResponses=None)
```

Class to allow simulation of a user's keyboard responses during a scan.

Given a list of response tuples (time, key), the thread will simulate a user pressing a key at a specific time (relative to the start of the run).

Author: Jeremy Gray; Idea: Mike MacAskill

```python
class psychopy.hardware.emulator.SyncGenerator
```

Class for a character-emitting metronome thread (emulate MR sync pulse).

Aim: Allow testing of temporal robustness of fMRI scripts by emulating a hardware sync pulse. Adds an arbitrary ‘sync’ character to the key buffer, with sub-millisecond precision (less precise if CPU is maxed). Recommend: TR=1.000 or higher and less than 100% CPU. Shorter TR → higher CPU load.

**Parameters**

- **TR** – seconds between volume acquisitions
- **TA** – seconds to acquire one volume
• **volumes** – number of 3D volumes to obtain in a given scanning run
• **sync** – character used as flag for sync timing, default='5'
• **skip** – how many frames to silently omit initially during T1 stabilization, no sync pulse. Not needed to test script timing, but will give more accurate feel to start of run. aka “discdacqs”.
• **sound** – simulate scanner noise

```python
run()
```

Method representing the thread’s activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object’s constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

### 8.4.7 fORP response box

fORP fibre optic (MR-compatible) response devices by CurrentDesigns: [http://www.curdes.com/](http://www.curdes.com/) This class is only useful when the fORP is connected via the serial port.

If you’re connecting via USB, just treat it like a standard keyboard. E.g., use a Keyboard component, and typically listen for Allowed keys '1’, ‘2’, ‘3’, ‘4’, ‘5’. Or use `event.getKeys()`.

```python
class psychopy.hardware.forp.ButtonBox (serialPort=1, baudrate=19200)
```

Serial line interface to the fORP MRI response box.

To use this object class, select the box use setting `serialPort`, and connect the serial line. To emulate key presses with a serial connection, use `getEvents(asKeys=True)` (e.g., to be able to use a RatingScale object during scanning). Alternatively connect the USB cable and use fORP to emulate a keyboard.

fORP sends characters at 800Hz, so you should check the buffer frequently. Also note that the trigger event numpy the fORP is typically extremely short (occurs for a single 800Hz epoch).

**Parameters**

- `serialPort`: should be a number (where 1=COM1, …)
- `baud`: the communication rate (baud), eg, 57600

```python
classmethod _decodePress (pressCode)
```

Returns a list of buttons and whether they’re pressed, given a character code.

- `pressCode`: A number with a bit set for every button currently pressed. Will be between 0 and 31.

```python
_generateEvents (pressCode)
```

For a given button press, returns a list buttons that went from unpressed to pressed. Also flags any unpressed buttons as unpressed.

- `pressCode`: a number with a bit set for every button currently pressed.

```python
clearBuffer ()
```

Empty the input buffer of all characters

```python
clearStatus ()
```

Resets the pressed statuses, so getEvents will return pressed buttons, even if they were already pressed in the last call.

```python
getEvents () (stored as ForpBox.rawEvts)
```

- `returnRaw`: return (not just store) the full event list
asKeys : If True, will also emulate pyglet keyboard events, so that button 1 will register as a keyboard event with value “1”, and as such will be detectable using event.getKeys()

allowRepeats : If True, this will return pressed buttons even if they were held down between calls to getEvents(). If the fORP is on the “Eprime” setting, you will get a stream of button presses while a button is held down. On the “Bitwise” setting, you will get a set of all currently pressed buttons every time a button is pressed or released. This option might be useful if you think your participant may be holding the button down before you start checking for presses.

getUniqueEvents (fullEvts=False)
Returns a Python set of the unique (unordered) events of either a list given or the current rawEvts buffer

8.4.8 iolab
This provides a basic ButtonBox class, and imports the ioLab python library.
class psychopy.hardware.iolab.ButtonBox
  PsychoPy’s interface to ioLabs.usbbox. Voice key completely untested.
  Original author: Jonathan Roberts PsychoPy rewrite: Jeremy Gray, 2013
  Class to detect and report ioLab button box.
The ioLabs library needs to be installed. It is included in the Standalone distributions of PsychoPy as of version 1.62.01. Otherwise try “pip install ioLabs”
Usage:

```python
from psychopy.hardware import iolab
bbox = iolab.ButtonBox()
```

For examples see the demos menu of the PsychoPy Coder or go to the URL above.
All times are reported in units of seconds.

_openTime (log=False)
Return the time on the bbox internal clock, relative to last reset.
Status: rtcget() not working

log=True will log the bbox time and elapsed CPU (python) time.
clearEvents ()
Discard all button / voice key events.

getBaseTime ()
Return the time since init (using the CPU clock, not ioLab bbox).
Aim is to provide a similar API as for a Cedrus box. Could let both clocks run for a long time to assess relative drift.

getEnabled ()
Return a list of the buttons that are currently enabled.

getEvents (downOnly=True)
Detect and return a list of all events (likely just one); no block.
Use downOnly=False to include button-release events.

resetClock (log=True)
Reset the clock on the bbox internal clock, e.g., at the start of a trial.
~1ms for me; logging is much faster than the reset
**setEnabled** *(buttonList=(0, 1, 2, 3, 4, 5, 6, 7), voice=False)*

Set a filter to suppress events from non-enabled buttons.

The ioLabs bbox filters buttons in hardware; here we just tell it what we want: None - disable all buttons an integer (0..7) - enable a single button a list of integers (0..7) - enable all buttons in the list

Set voice=True to enable the voiceKey - gets reported as button 64

**setLights** *(lightList=(0, 1, 2, 3, 4, 5, 6, 7))*

Turn on the specified LEDs (None, 0..7, list of 0..7)

**standby** ()

Disable all buttons and lights.

**waitEvents** *(downOnly=True, timeout=0, escape='escape', wait=0.002)*

Wait for and return the first button press event.

Always calls `clearEvents()` first (like PsychoPy keyboard `waitKeys`).

Use `downOnly=False` to include button-release events.

`escape` is a list/tuple of keyboard events that, if pressed, will interrupt the bbox wait; `waitKeys` will return None in that case.

`timeout` is the max time to wait in seconds before returning None. `timeout` of 0 means no time-out (= default).

### 8.4.9 joystick (pyglet and pygame)

AT THE MOMENT JOYSTICK DOES NOT APPEAR TO WORK UNDER PYGLET. We need someone motivated and capable to go and get this right (problem is with event polling under pyglet)

Control joysticks and gamepads from within PsychoPy.

You do need a window (and you need to be flipping it) for the joystick to be updated.

**Known issues:**

- currently under pyglet the joystick axes initialise to a value of zero and stay like this until the first time that axis moves
- currently pygame (1.9.1) spits out lots of debug messages about the joystick and these can’t be turned off :-(

Typical usage:

```python
from psychopy.hardware import joystick
from psychopy import visual

joystick.backend='pyglet'  # must match the Window
win = visual.Window((400, 400), winType='pyglet')

nJoys = joystick.getNumJoysticks()  # to check if we have any
id = 0
joy = joystick.Joystick(id)  # id must be <= nJoys - 1

nAxes = joy.getNumAxes()  # for interest
while True:  # while presenting stimuli
    joy.getX()  # ...
    win.flip()  # flipping implicitly updates the joystick info
```

**class** `psychopy.hardware.joystick.XboxController(id, *args, **kwargs)`

Joystick template class for the XBox 360 controller.
Usage:

xbctrl = XboxController(0)  # joystick ID
y_btn_state = xbctrl.y  # get the state of the ‘Y’ button.

An object to control a multi-axis joystick or gamepad.

Known issues  Currently under pyglet backends the axis values initialise to zero rather than reading the current true value. This gets fixed on the first change to each axis.

_clip_range(val)
Clip the range of a value between -1.0 and +1.0. Needed for joystick axes.

Parameters val –

Returns

get_a()
Get the ‘A’ button state.

Returns bool, True if pressed down

get_b()
Get the ‘B’ button state.

Returns bool, True if pressed down

get_back()
Get ‘back’ button state (button to the right of the left joystick).

Returns bool, True if pressed down

get_hat_axis()
Get the states of the hat (sometimes called the ‘directional pad’). The hat can only indicate direction but not displacement.

This function reports hat values in the same way as a joystick so it may be used interchangeably with existing analog joystick code.

Returns a tuple (X, Y) indicating which direction the hat is pressed between -1.0 and +1.0. Positive values indicate presses in the right or up direction.

Returns tuple, zero centered X, Y values.

get_left_shoulder()
Get left ‘shoulder’ trigger state.

Returns bool, True if pressed down

get_left_thumbstick()
Get the state of the left joystick button; activated by pressing down on the stick.

Returns bool, True if pressed down

get_left_thumbstick_axis()
Get the axis displacement values of the left thumbstick.

Returns a tuple (X, Y) indicating thumbstick displacement between -1.0 and +1.0. Positive values indicate the stick is displaced right or up.

Returns tuple, zero centered X, Y values.

get_named_buttons(button_names)
Get the states of multiple buttons using names. A list of button states is returned for each string in list ‘names’.

Parameters button_names – tuple or list of button names

Returns
get_right_shoulder()
Get right 'shoulder' trigger state.

Returns bool, True if pressed down

get_right_thumbstick()
Get the state of the right joystick button; activated by pressing down on the stick.

Returns bool, True if pressed down

get_right_thumbstick_axis()
Get the axis displacement values of the right thumbstick.

Returns a tuple (X,Y) indicating thumbstick displacement between -1.0 and +1.0. Positive values indicate the stick is displaced right or up.

Returns tuple, zero centered X, Y values.

get_start()
Get 'start' button state (button to the left of the 'X' button).

Returns bool, True if pressed down

get_trigger_axis()
Get the axis displacement values of both index triggers.

Returns a tuple (L,R) indicating index trigger displacement between -1.0 and +1.0. Values increase from -1.0 to 1.0 the further a trigger is pushed.

Returns tuple, zero centered L, R values.

get_x()
Get the 'X' button state.

Returns bool, True if pressed down

get_y()
Get the 'Y' button state.

Returns bool, True if pressed down

psychopy.hardware.joystick.getNumJoysticks()
Return a count of the number of joysticks available.

class psychopy.hardware.joystick.Joystick(id)
An object to control a multi-axis joystick or gamepad.

Known issues Currently under pyglet backends the axis values initialise to zero rather than reading the current true value. This gets fixed on the first change to each axis.

getAllAxes()
Get a list of all current axis values.

getAllButtons()
Get the state of all buttons as a list.

getAllHats()
Get the current values of all available hats as a list of tuples.

Each value is a tuple (x, y) where x and y can be -1, 0, +1

getAxis(axisId)
Get the value of an axis by an integer id.

(from 0 to number of axes - 1)
PsychoPy - Psychology software for Python, Release 2020.2.10

**getButton**(buttonId)
Get the state of a given button.
buttonId should be a value from 0 to the number of buttons-1

**getHat**(hatId=0)
Get the position of a particular hat.
The position returned is an (x, y) tuple where x and y can be -1, 0 or +1

**getName**()
Return the manufacturer-defined name describing the device.

**getNumAxes**()
Return the number of joystick axes found.

**getNumButtons**()
Return the number of digital buttons on the device.

**getNumHats**()
Get the number of hats on this joystick.
The GLFW backend makes no distinction between hats and buttons. Calling ‘getNumHats()’ will return 0.

**getX**()
Return the X axis value (equivalent to joystick.getAxis(0)).

**getY**()
Return the Y axis value (equivalent to joystick.getAxis(1)).

**getZ**()
Return the Z axis value (equivalent to joystick.getAxis(2)).

### 8.4.10 labjacks (USB I/O devices)

PsychoPy provides an interface to the labjack U3 class with a couple of minor additions.

This is accessible by:

```python
from psychopy.hardware.labjacks import U3
```

Except for the additional `setdata` function the U3 class operates exactly as that in the U3 library that labjack provides, documented here:

http://labjack.com/support/labjackpython

**Note:** To use labjack devices you do need also to install the driver software described on the page above

```
class psychopy.hardware.labjacks.U3(debug=False, autoOpen=True, **kargs)
    Name: U3.__init__(debug = False, autoOpen = True, **openArgs)
    Args: debug, enables debug output autoOpen, if true, the class will try to open a U3 using openArgs **openArgs, the arguments to pass to the open call. See U3.open()
    Desc: Instantiates a new U3 object. If autoOpen == True, then it will also open a U3.
    Examples: Simplest: >>> import u3 >>> d = u3.U3()
    For debug output: >>> import u3 >>> d = u3.U3(debug = True)
    To open a U3 with Local ID = 2: >>> import u3 >>> d = u3.U3(localId = 2)
```
setData((byte, endian='big', address=6701))
Write 1 byte of data to the U3 port

Parameters

- **byte** (-) – the value to write (must be an integer 0:255)
- **endian** (-) – [‘big’ or ‘small’] ignored from 1.84 onwards; automatic?
- **address** (-) – the memory address to send the byte to - 6700 = FIO - 6701 (default) = EIO (the DB15 connector) - 6702 = CIO

8.4.11 Minolta

Minolta light-measuring devices See http://www.konicaminolta.com/instruments

class psychopy.hardware.minolta.LS100 (port, maxAttempts=1)
A class to define a Minolta LS100 (or LS110?) photometer

You need to connect a LS100 to the serial (RS232) port and **when you turn it on press the F key** on the device. This will put it into the correct mode to communicate with the serial port.

usage:

```python
from psychopy.hardware import minolta
phot = minolta.LS100(port)
if phot.OK:  # then we successfully made a connection
    print(phot.getLum())
```

Parameters

port: string
the serial port that should be checked

maxAttempts: int If the device doesn’t respond first time how many attempts should be made?
If you’re certain that this is the correct port and the device is on and correctly configured then this could be set high. If not then set this low.

Troubleshooting Various messages are printed to the log regarding the function of this device, but to see them you need to set the printing of the log to the correct level:

```python
from psychopy import logging
logging.console.setLevel(logging.ERROR)  # error messages only
logging.console.setLevel(logging.INFO)   # more info
logging.console.setLevel(logging.DEBUG)  # log all communications
```

If you’re using a keyspan adapter (at least on macOS) be aware that it needs a driver installed. Otherwise no ports will be found.

Error messages:

**ERROR: Couldn’t connect to Minolta LS100/110 on ____**: This likely means that the device is not connected to that port (although the port has been found and opened). Check that the device has the / in the bottom right of the display; if not turn off and on again holding the F key.
ERROR: No reply from LS100: The port was found, the connection was made and an initial command worked, but then the device stopped communicating. If the first measurement taken with the device after connecting does not yield a reasonable intensity the device can sulk (not a technical term!). The “I” on the display will disappear and you can no longer communicate with the device. Turn it off and on again (with F depressed) and use a reasonably bright screen for your first measurement. Subsequent measurements can be dark (or we really would be in trouble!).

checkOK(msg)
Check that the message from the photometer is OK. If there’s an error show it (printed).

Then return True (OK) or False.

clearMemory()
Clear the memory of the device from previous measurements

getLum()
Makes a measurement and returns the luminance value

measure()
Measure the current luminance and set .lastLum to this value

sendMessage(message, timeout=5.0)
Send a command to the photometer and wait an allotted timeout for a response.

setMaxAttempts(maxAttempts)
Changes the number of attempts to send a message and read the output. Typically this should be low initially, if you aren’t sure that the device is setup correctly but then, after the first successful reading, set it higher.

setMode(mode='04')
Set the mode for measurements. Returns True (success) or False

‘04’ means absolute measurements. ‘08’ = peak ‘09’ = cont

See user manual for other modes

8.4.12 PhotoResearch

Supported devices:
• PR650
• PR655/PR670

PhotoResearch spectrophotometers See http://www.photoresearch.com/

class psychopy.hardware.pr.PR650 (port, verbose=None)
An interface to the PR650 via the serial port.

(Added in version 1.63.02)

example usage:

```python
from psychopy.hardware.pr import PR650
myPR650 = PR650(port)
myPR650.getLum() # make a measurement
nm, power = myPR650.getLastSpectrum() # get a power spectrum for the last measurement
```
NB `psychopy.hardware.findPhotometer()` will locate and return any supported device for you so you can also do:

```python
from psychopy import hardware
phot = hardware.findPhotometer()
print(phot.getLum())
```

**Troubleshooting** Various messages are printed to the log regarding the function of this device, but to see them you need to set the printing of the log to the correct level:

```python
from psychopy import logging
logging.console.setLevel(logging.ERROR)  # error messages only
logging.console.setLevel(logging.INFO)   # will give more info
logging.console.setLevel(logging.DEBUG)  # log all communications
```

If you're using a keyspan adapter (at least on macOS) be aware that it needs a driver installed. Otherwise no ports will be found.

Also note that the attempt to connect to the PR650 must occur within the first few seconds after turning it on.

**getLastLum()**
This retrieves the luminance (in cd/m**2**) from the last call to `.measure()`

**getLastSpectrum (parse=True)**
This retrieves the spectrum from the last call to `.measure()`

If `parse=True` (default): The format is a num array with 100 rows [nm, power]
o otherwise: The output will be the raw string from the PR650 and should then be passed to `.parseSpectrumOutput()`. It's more efficient to parse R,G,B strings at once than each individually.

**getLum()**
Makes a measurement and returns the luminance value

**getSpectrum (parse=True)**
Makes a measurement and returns the current power spectrum

If `parse=True` (default): The format is a num array with 100 rows [nm, power]

If `parse=False` (default): The output will be the raw string from the PR650 and should then be passed to `.parseSpectrumOutput()`. It's slightly more efficient to parse R,G,B strings at once than each individually.

**measure (timeOut=30.0)**
Make a measurement with the device. For a PR650 the device is instructed to make a measurement and then subsequent commands are issued to retrieve info about that measurement.

**parseSpectrumOutput (rawStr)**
 Parses the strings from the PR650 as received after sending the command ‘d5’. The input argument “rawStr” can be the output from a single phosphor spectrum measurement or a list of 3 such measurements [rawR, rawG, rawB].

**sendMessage (message, timeout=0.5, DEBUG=False)**
Send a command to the photometer and wait an allotted timeout for a response (Timeout should be long for low light measurements)

**class psychopy.hardware.pr.PR655 (port)**
An interface to the PR655/PR670 via the serial port.

example usage:
from psychopy.hardware.pr import PR655
myPR655 = PR655(port)
myPR655.getLum()  # make a measurement
nm, power = myPR655.getLastSpectrum()  # get a power spectrum for the last measurement

NB `psychopy.hardware.findPhotometer()` will locate and return any supported device for you so you can also do:

```python
from psychopy import hardware
phot = hardware.findPhotometer()
print(phot.getLum())
```

**Troubleshooting** If the device isn’t responding try turning it off and turning it on again, and/or disconnecting/reconnecting the USB cable. It may be that the port has become controlled by some other program.

```python
def endRemoteMode():
    Puts the colorimeter back into normal mode

getDeviceSN():
    Return the device serial number

getDeviceType():
    Return the device type (e.g. ‘PR-655’ or ‘PR-670’)

getLastColorTemp():
    Fetches (from the device) the color temperature (K) of the last measurement
    Returns list: status, units, exponent, correlated color temp (Kelvins), CIE 1960 deviation
    See also `measure()` automatically populates pr655.lastColorTemp with the color temp in Kelvins

getLastSpectrum(parse=True)
    This retrieves the spectrum from the last call to `measure()`
    If `parse=True` (default):
    The format is a num array with 100 rows [nm, power]
    otherwise:
    The output will be the raw string from the PR650 and should then be passed to `parseSpectrumOutput()`.
    It’s more efficient to parse R,G,B strings at once than each individually.

getLastTristim():
    Fetches (from the device) the last CIE 1931 Tristimulus values
    Returns list: status, units, Tristimulus Values
    See also `measure()` automatically populates pr655.lastTristim with just the tristimulus coordinates

getLastUV():
    Fetches (from the device) the last CIE 1976 u,v coords
    Returns list: status, units, Photometric brightness, u, v
    See also `measure()` automatically populates pr655.lastUV with [u,v]
**getLastXY()**

Fetches (from the device) the last CIE 1931 x,y coords

**Returns**: list: status, units, Photometric brightness, x,y

**See also**: measure() automatically populates pr655.lastXY with [x,y]

**measure**(timeout=30.0)

Make a measurement with the device.

This automatically populates:

- .lastLum
- .lastSpectrum
- .lastCIExy
- .lastCIEuv

**parseSpectrumOutput**(rawStr)

Parses the strings from the PR650 as received after sending the command ‘D5’. The input argument “rawStr” can be the output from a single phosphor spectrum measurement or a list of 3 such measurements [rawR, rawG, rawB].

**startRemoteMode()**

Sets the Colorimeter into remote mode

### 8.4.13 pylink (SR research)

For now the SR Research pylink module is packaged with the Standalone flavours of PsychoPy and can be imported with:

```python
import pylink
```

You do need to install the Display Software (which they also call Eyelink Developers Kit) for your particular platform. This can be found by following the threads from:


for pylink documentation see:


Performing research with eye-tracking equipment typically requires a long-term investment in software tools to collect, process, and analyze data. Much of this involves real-time data collection, saccadic analysis, calibration routines, and so on. The EyeLink® eye-tracking system is designed to implement most of the required software base for data collection and conversion. It is most powerful when used with the Ethernet link interface, which allows remote control of data collection and real-time data transfer. The PyLink toolkit includes Pylink module, which implements all core EyeLink functions and classes for EyeLink connection and the eyelink graphics, such as the display of camera image, calibration, validation, and drift correct. The EyeLink graphics is currently implemented using Simple Direct Media Layer (SDL: www.libsdl.org).

The Pylink library contains a set of classes and functions, which are used to program experiments on many different platforms, such as MS-DOS, Windows, Linux, and the Macintosh. Some programming standards, such as placement of messages in the EDF file by your experiment, and the use of special data types, have been implemented to allow portability of the development kit across platforms. The standard messages allow general analysis tools such as EDF2ASC converter or EyeLink Data Viewer to process your EDF files.

**pylink.alert**(message)

This method is used to give a notification to the user when an error occurs. Parameters <message>: Text message
to be displayed. Return Value: None Remarks: This function does not allow printf formatting as in c. However you can do a formatted string argument in python. This is equivalent to the C API void alert_printf(char*fmt, …);

`pylink.beginRealTimeMode(delay)`
Sets the application priority and cleans up pending Windows activity to place the application in realtime mode. This could take up to 100 milliseconds, depending on the operation system, to set the application priority. Parameters <delay> an integer, used to set the minimum time this function takes, so that this function can act as a useful delay. Return Value None This function is equivalent to the C API void begin_realtime_mode(UINT32 delay);

`pylink.bitmapSave(iwidth,iheight,pixels, xs, ys, width, height,fname,path, sv_options)`

iwidth - original image width
iheight - original image height
pixels - Pixels of the image in one of two possible formats: pixel=[line1, line2, ... linen] line=[pix1, pix2,..., pixn], pix=(r,g,b). pixel=[line1, line2, ... linen] line=[pix1, pix2,..., pixn], pix=0xAARRGGBB.
xs - crop x position
ys - crop y position
width - crop width
height - crop height
fname - file name to save
path - path to save
svoptions - save options(SV_NOREPLACE,SV_MAKEPATH)

`pylink.closeGraphics()`
Notifies the eyelink_core_graphics to close or release the graphics. Parameters None Return Value None This is equivalent to the C API void close_expt_graphics(void); This function should not be used with custom graphics

`pylink.closeMessageFile()`
DOC UNDONE

`pylink.currentTime()`
Returns the current millisecond time since the initialization of the EyeLink library. Parameters None. Return Value Long integer for the current millisecond time since the initialization of the EyeLink library This function is equivalent to the C API UINT32 current_time(void);

`pylink.currentUsec()`
Returns the current microsecond time since the initialization of the EyeLink library. Parameters None. Return Value Long integer for the current microsecond time since the initialization of the EyeLink library This is equivalent to the C API UINT32 current_usec(void);

`pylink.enableExtendedRealtime()`
DOC UNDONE

`pylink.enablePCRSample()`
If enabled, the raw data can be obtained

`pylink.endRealTimeMode()`
Returns the application to a priority slightly above normal, to end realtime mode. This function should execute rapidly, but there is the possibility that Windows will allow other tasks to run after this call, causing delays of 1-20 milliseconds. Parameters None Return Value None This function is equivalent to the C API void end_realtime_mode(void);

`pylink.flushGetkeyQueue()`
Initializes the key queue used by getkey(). It may be called at any time to get rid any of old keys from the queue. Parameters None Return Value None This is equivalent to the C API void flush_getkey_queue(void);

`pylink.getLastError()`
get error number returned by last call to corresponding C-API function

`pylink.inRealTimeMode()`
DOC UNDONE
pylink.msecDelay(delay)
   Does a unblocked delay using currentTime(). Parameters <delay>: an integer for number of milliseconds to delay.
   Return Value None This is equivalent to the C API void msec_delay(UINT32 n);

pylink.openCustomGraphicsInternal()
   DOC UNDONE

pylink.openGraphics()
   openGraphics(dimension, bits): Opens the graphics if the display mode is not set. If the display mode is already set,
   uses the existing display mode. Parameters <dimension>: two-item tuple of display containing width and height
   information. <bits>: color bits. Return Value None or run-time error. This is equivalent to the SDL version C API
   INT16 init_expt_graphics(SDL_Surface * s, DISPLAYINFO *info).

pylink.openMessageFile()
   DOC UNDONE

pylink.pumpDelay(delay)
   During calls to msecDelay(), Windows is not able to handle messages. One result of this is that windows may not
   appear. This is the preferred delay function when accurate timing is not needed. It calls pumpMessages() until the
   last 20 milliseconds of the delay, allowing Windows to function properly. In rare cases, the delay may be longer than
   expected. It does not process modeless dialog box messages. Parameters <delay>: an integer, which sets number
   of milliseconds to delay. Return Value None Use the This is equivalent to the C API void pump_delay(UINT32
   delay);

pylink.resetBackground()
   DOC UNDONE

pylink.sendMessageToFile()
   DOC UNDONE

pylink.setCalibrationColors(foreground_color, background_color)
   Passes the colors of the display background and fixation target to the eyelink_core_graphics library. During cali-
   bration, camera image display, and drift correction, the display background should match the brightness of the ex-
   perimental stimuli as closely as possible, in order to maximize tracking accuracy. This function passes the colors
   of the display background and fixation target to the eyelink_core_graphics library. This also prevents flickering of
   the display at the beginning and end of drift correction. Parameters <foreground_color>: color for foreground calibration
   target. <background_color>: color for foreground calibration background. Both colors must be a three integer
   (from 0 to 255) tuple encoding the red, blue, and green color component. Return Value None This is equivalent
   to the C API void set_calibration_colors(SDL_Color *fg, SDL_Color *bg); Example: setCalibrationColors((0, 0,
   0),(255, 255, 255)) This sets the calibration target in black and calibration background in white.

pylink.setCalibrationSounds(target, good, error)
   Selects the sounds to be played during do_tracker_setup(), including calibration, validation and drift correction.
   These events are the display or movement of the target, successful conclusion of calibration or good validation,
   and failure or interruption of calibration or validation. Note: If no sound card is installed, the sounds are produced as
   beeps from the PC speaker. Otherwise, sounds can be selected by passing a string. If the string is (empty), the
   default sounds are played. If the string is off, no sound will be played for that event. Otherwise, the string should
   be the name of a .WAV file to play. Parameters <target>: Sets sound to play when target moves; <good>: Sets
   sound to play on successful operation; <error>: Sets sound to play on failure or interruption. Return Value None
   This function is equivalent to the C API void set_cal_sounds(char *target, char *good, char *error);

pylink.setCameraPosition(left, top, right, bottom)
   Sets the camera position on the display computer. Moves the top left hand corner of the camera position to new
   location. Parameters <left>: x-coord of upper-left corner of the camera image window; <top>: y-coord of upper-
   left corner of the camera image window; <right>: x-coord of lower-right corner of the camera image window;
   <bottom>: y-coord of lower-right corner of the camera image window. Return Value None

pylink.setDriftCorrectSounds(target, good, setup)
   Selects the sounds to be played during doDriftCorrect(). These events are the display or movement of the target,
successful conclusion of drift correction, and pressing the ESC key to start the Setup menu. Note: If no sound card is installed, the sounds are produced as beeps from the PC speaker. Otherwise, sounds can be selected by passing a string. If the string is empty, the default sounds are played. If the string is off, no sound will be played for that event. Otherwise, the string should be the name of a .WAV file to play. Parameters <target>: Sets sound to play when target moves; <good>: Sets sound to play on successful operation; <setup>: Sets sound to play on ESC key pressed. Return Value None This function is equivalent to the C API void set_dcorr_sounds(char *target, char *good, char *setup);

pylink.setTargetSize(diameter, holesize)
The standard calibration and drift correction target is a disk (for peripheral detectability) with a central hole target (for accurate fixation). The sizes of these features may be set with this function. Parameters <diameter>: Size of outer disk, in pixels. <holesize>: Size of central feature, in pixels. If holesize is 0, no central feature will be drawn. The disk is drawn in the calibration foreground color, and the hole is drawn in the calibration background color. Return Value None This function is equivalent to the C API void set_target_size(UINT16 diameter, UINT16 holesize);

8.4.14 pump - A simple interface to the Cetoni neMESYS syringe pump system

Please specify the name of the pump configuration to use in the PsychoPy preferences under Hardware / Qmix pump configuration. See the readme file of the pyqmix project for details on how to set up your computer and create the configuration file.

Simple interface to the Cetoni neMESYS syringe pump system, based on the pyqmix library. The syringe pump system is described in the following publication:


Behavior Research Methods. doi: 10.3758/s13428-018-1145-1

class psychopy.hardware.qmix.Pump(index, volumeUnit='mL', flowRateUnit='mL/s', syringeType='50 mL glass')
An interface to Cetoni neMESYS syringe pumps, based on the pyqmix library.

Parameters

- index (int) – The index of the pump. The first pump in the system has index=0, the second index=1, etc.
- volumeUnit ('mL') – The unit in which the volumes are provided. Currently, only 'ml' is supported.
- flowRateUnit ('mL/s' or 'mL/min') – The unit in which flow rates are provided.
- syringeType ('25 mL glass' or '50 mL glass') – Type of the installed syringe, as understood by pyqmix.

aspirate (volume, flowRate, waitUntilDone=False, switchValveWhenDone=False)
Aspirate the specified volume.

Parameters

- volume (float) – The volume to aspirate.
- flowRate (float) – The desired flow rate.
- waitUntilDone (bool) – Whether to block program execution until calibration is completed.
• **switchValveWhenDone** (*bool*) – If *True*, switch the valve to dispense position after the aspiration is finished. Implies *wait_until_done=True*.

**calibrate** (*waitUntilDone=False*)

Calibrate the syringe pump.

You must not use this function if a syringe is installed in the pump as the syringe may be damaged!

**Parameters**

- **waitUntilDone** (*bool*) – Whether to block program execution until calibration is completed.

**clearFaultState** ()

Switch the pump back to an operational state after an error had occurred.

**dispense** (*volume, flowRate, waitUntilDone=False, switchValveWhenDone=False*)

Dispense the specified volume.

**Parameters**

- **volume** (*float*) – The volume to dispense.
- **flowRate** (*float*) – The desired flow rate.
- **waitUntilDone** (*bool*) – Whether to block program execution until calibration is completed.
- **switchValveWhenDone** (*bool*) – If *True*, switch the valve to aspiration position after the dispense is finished. Implies *wait_until_done=True*.

**empty** (*flowRate, waitUntilDone=False, switchValveWhenDone=False*)

Empty the syringe entirely.

**Parameters**

- **flowRate** (*float*) – The desired flow rate.
- **waitUntilDone** (*bool*) – Whether to block program execution until calibration is completed.
- **switchValveWhenDone** (*bool*) – If *True*, switch the valve to aspirate position after the dispensing is finished. Implies *wait_until_done=True*.

**fill** (*flowRate, waitUntilDone=False, switchValveWhenDone=False*)

Fill the syringe entirely.

**Parameters**

- **flowRate** (*float*) – The desired flow rate.
- **waitUntilDone** (*bool*) – Whether to block program execution until calibration is completed.
- **switchValveWhenDone** (*bool*) – If *True*, switch the valve to dispense position after the aspiration is finished. Implies *wait_until_done=True*.

**property fillLevel**

Current fill level of the syringe.

**property flowRateUnit**

The unit in which flow rates are provided.

**property isInFaultState**

Whether the pump is currently in a non-operational “fault state”.

To enable the pump again, call **clearFaultState()**.
**property maxFlowRate**

Maximum flow rate the pump can provide with the installed syringe.

**stop()**

Stop any pump operation immediately.

**switchValvePosition()**

Switch the valve to the opposite position.

**property syringeType**

Type of the installed syringe.

**property volumeUnit**

The unit in which the volumes are provided.

*psychopy.hardware.findPhotometer*(ports=None, device=None)

Try to find a connected photometer/photospectrometer!

PsychoPy will sweep a series of serial ports trying to open them. If a port successfully opens then it will try to issue a command to the device. If it responds with one of the expected values then it is assumed to be the appropriate device.

**Parameters**

- **ports** [a list of ports to search] Each port can be a string (e.g. ‘COM1’, ‘/dev/tty.Keyspan1.1’) or a number (for win32 comports only). If none are provided then PsychoPy will sweep COM0-10 on win32 and search known likely port names on macOS and Linux.

- **device** [string giving expected device (e.g. ‘PR650’, ‘PR655’, ‘LS100’, ‘LS110’). If this is not given then an attempt will be made to find a device of any type, but this often fails

**Returns**

- An object representing the first photometer found
- None if the ports didn’t yield a valid response
- None if there were not even any valid ports (suggesting a driver not being installed)

```python
# sweeps ports 0 to 10 searching for a PR655
photom = findPhotometer(device='PR655')
print(photom.getLum())
if hasattr(photom, 'getSpectrum'):
    # can retrieve spectrum (e.g. a PR650)
    print(photom.getSpectrum())
```

---

**8.5 psychopy.iohub - ioHub event monitoring framework**

ioHub monitors for device events in parallel with the PsychoPy experiment execution by running in a separate process than the main PsychoPy script. This means, for instance, that keyboard and mouse event timing is not quantized by the rate at which the window.flip() method is called.

ioHub reports device events to the PsychoPy experiment runtime as they occur. Optionally, events can be saved to a HDF5 file.

All iohub events are timestamped using the PsychoPy global time base (psychopy.core.getTime()). Events can be accessed as a device independent event stream, or from a specific device of interest.
A comprehensive set of examples that each use at least one of the iohub devices is available in the psychopy/demos/coder/iohub folder.

Note: This documentation is in very early stages of being written. Comments and contributions are welcome.

### 8.5.1 Starting the psychopy.iohub Process

To use ioHub within your PsychoPy Coder experiment script, ioHub needs to be started at the start of the experiment script. The easiest way to do this is by calling the `launchHubServer` function.

**launchHubServer function**

```python
psychopy.iohub.client.launchHubServer(**kwargs)
```

Starts the ioHub Server subprocess, and return a `psychopy.iohub.client.ioHubConnection` object that is used to access enabled iohub device’s events, get events, and control the ioHub process during the experiment.

By default (no kwargs specified), the ioHub server does not create an ioHub HDF5 file, events are available to the experiment program at runtime. The following Devices are enabled by default:

- Keyboard: named ‘keyboard’, with runtime event reporting enabled.
- Mouse: named ‘mouse’, with runtime event reporting enabled.
- Monitor: named ‘monitor’.
- Experiment: named ‘experiment’.

To customize how the ioHub Server is initialized when started, use one or more of the following keyword arguments when calling the function:
<table>
<thead>
<tr>
<th>kwarg Name</th>
<th>Value Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>experiment_code</td>
<td>str, &lt;= 24 char</td>
<td>If experiment_code is provided, an ioHub HDF5 file will be created for the session.</td>
</tr>
<tr>
<td>session_code</td>
<td>str, &lt;= 24 char</td>
<td>When specified, used as the name of the ioHub HDF5 file created for the session.</td>
</tr>
<tr>
<td>experiment_info</td>
<td>dict</td>
<td>Can be used to save the following experiment metadata fields:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• code: str, &lt;= 24 char</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• title: str, &lt;= 48 char</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• description: str, &lt; 256 char</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• version: str, &lt;= 6 char</td>
</tr>
<tr>
<td>session_info</td>
<td>dict</td>
<td>Can be used to save the following session metadata fields:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• code: str, &lt;= 24 char</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• name: str, &lt;= 48 char</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• comments: str, &lt; 256 char</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• user_variables: dict</td>
</tr>
<tr>
<td>datastore_name</td>
<td>str</td>
<td>Used to provide an ioHub HDF5 file name different than the session_code.</td>
</tr>
<tr>
<td>psychopy_monitor_name</td>
<td>str</td>
<td>Provides the path of a PsychoPy Monitor Center config file. Information like display size is read and used to update the ioHub Display Device config.</td>
</tr>
<tr>
<td>iohub_config_name</td>
<td>str</td>
<td>Specifies the name of the iohub_config.yaml file that contains the ioHub Device list to be used by the ioHub Server. i.e. the 'device_list' section of the yaml file.</td>
</tr>
<tr>
<td>iohub.device.path</td>
<td>dict</td>
<td>Add an ioHub Device by using the device class path as the key, and the device's configuration in a dict value.</td>
</tr>
</tbody>
</table>

**Examples**

A. Wait for the ‘q’ key to be pressed:

```python
from psychopy.iohub.client import launchHubServer

# Start the ioHub process. 'io' can now be used during the # experiment to access iohub devices and read iohub device events.
io = launchHubServer()

print("Press any Key to Exit Example.....")

# Wait until a keyboard event occurs
keys = io.devices.keyboard.waitForKeys(keys=['q',])

print("Key press detected: {}".format(keys))
print("Exiting experiment....")

# Stop the ioHub Server
io.quit()
```

Please see the psychopy/demos/coder/iohub/launchHub.py demo for examples of different ways to use the launch-
HubServer function.

**ioHubConnection Class**

The psychopy.iohub.ioHubConnection object returned from the launchHubServer function provides methods for controlling the iohub process and accessing iohub devices and events.

```python
class psychopy.iohub.client.ioHubConnection(object)

ioHubConnection is responsible for creating, sending requests to, and reading replies from the ioHub Process. This class is also used to shut down and disconnect the ioHub Server process.

The ioHubConnection class is also used as the interface to any ioHub Device instances that have been created so that events from the device can be monitored. These device objects can be accessed via the ioHubConnection.devices attribute, providing 'dot name' access to enabled devices. Alternatively, the .getDevice(name) method can be used and will return None if the device name specified does not exist.

Using the .devices attribute is handy if you know the name of the device to be accessed and you are sure it is actually enabled on the ioHub Process.

An example of accessing a device using the .devices attribute:

```python
# get the Mouse device, named mouse
mouse=hub.devices.mouse
mouse_position = mouse.getPosition()

print 'mouse position: ', mouse_position
```

This is the same as using the 'natural naming' approach supported by the .devices attribute, i.e:

```python
keyboard = self.devices.kb
kb_events= keyboard.getEvent()
```

However the advantage of using getDevice(device_name) is that an exception is not created if you provide an invalid device name, or if the device is not enabled on the ioHub server; None is returned instead.

**Parameters**

- **deviceName** *(str)* – Name given to the ioHub Device to be returned

**Returns**

The ioHubDeviceView instance for deviceName.

**getEvents** *(device_label=None, as_type='namedtuple')*

Retrieve any events that have been collected by the ioHub Process from monitored devices since the last call to getEvents() or clearEvents().

By default all events for all monitored devices are returned, with each event being represented as a namedtuple of all event attributes.

When events are retrieved from an event buffer, they are removed from that buffer as well.
If events are only needed from one device instead of all devices, providing a valid device name as the device_label argument will result in only events from that device being returned.

Events can be received in one of several object types by providing the optional as_type property to the method. Valid values for as_type are the following str values:

- ‘list’: Each event is a list of ordered attributes.
- ‘namedtuple’: Each event is converted to a namedtuple object.
- ‘dict’: Each event converted to a dict object.
- ‘object’: Each event is converted to a DeviceEvent subclass based on the event’s type.

Parameters

- **device_label** *(str)* – Name of device to retrieve events for. If None (the default) returns device events from all devices.
- **as_type** *(str)* – Returned event object type. Default: ‘namedtuple’.

Returns List of event objects; object type controlled by ‘as_type’.

Return type tuple

clearEvents *(device_label=’all’)*

Clears unread events from the ioHub Server’s Event Buffer(s) so that unneeded events are not discarded.

If device_label is ‘all’, (the default), then events from both the ioHub Global Event Buffer and all Device Event Buffers are cleared.

If device_label is None then all events in the ioHub Global Event Buffer are cleared, but the Device Event Buffers are unaffected.

If device_label is a str giving a valid device name, then that Device Event Buffer is cleared, but the Global Event Buffer is not affected.

Parameters **device_label** *(str)* – device name, ‘all’, or None

Returns None

sendMessageEvent *(text, category='', offset=0.0, sec_time=None)*

Create and send an Experiment MessageEvent to the ioHub Server for storage in the ioDataStore hdf5 file.

**Note:** MessageEvents can be thought of as DeviceEvents from the virtual PsychoPy Process “Device”.

Parameters

- **text** *(str)* – The text message for the message event. 128 char max.
- **category** *(str)* – A str grouping code for the message. Optional. 32 char max.
- **offset** *(float)* – Optional sec.msec offset applied to the message event time stamp. Default 0.
- **sec_time** *(float)* – Absolute sec.msec time stamp for the message in. If not provided, or None, then the MessageEvent is time stamped when this method is called using the global timer (core.getTime()).

Returns True

Return type bool
createTrialHandlerRecordTable(trials, cv_order=None)
Create a condition variable table in the ioHub data file based on the a psychopy TrialHandler. By doing so, the iohub data file can contain the DV and IV values used for each trial of an experiment session, along with all the iohub device events recorded by iohub during the session.

Example psychopy code usage:

```python
# Load a trial handler and
# create an associated table in the iohub data file
#
from psychopy.data import TrialHandler, importConditions

exp_conditions=importConditions('trial_conditions.xlsx')
trials = TrialHandler(exp_conditions, 1)

# Inform the ioHub server about the TrialHandler
#
io.createTrialHandlerRecordTable(trials)

# Read a row of the trial handler for
# each trial of your experiment
#
for trial in trials:
    # do whatever...

# During the trial, trial variable values can be updated
#
trial['TRIAL_START']=flip_time

# At the end of each trial, before getting
# the next trial handler row, send the trial
# variable states to iohub so they can be stored for future
# reference.
#
io.addTrialHandlerRecord(trial)
```

addTrialHandlerRecord(cv_row)

Adds the values from a TrialHandler row / record to the iohub data file for future data analysis use.

**Parameters** cv_row –

**Returns** None

getime()

**Deprecated Method:** Use Computer.getTime instead. Remains here for testing time bases between processes only.

setPriority(level='normal', disable_gc=False)

See Computer.setPriority documentation, where current process will be the iohub process.

g_priority()

See Computer.getPriority documentation, where current process will be the iohub process.

getProcessAffinity()

Returns the current ioHub Process affinity setting, as a list of 'processor' id's (from 0 to getSystemProcessorCount()-1). A Process’s Affinity determines which CPU’s or CPU cores a process can run on. By default the ioHub Process can run on any CPU or CPU core.

This method is not supported on OS X at this time.
Parameters None –

Returns

A list of integer values between 0 and Computer.getSystemProcessorCount()-1, where values in the list indicate processing unit indexes that the ioHub process is able to run on.

Return type list

**setProcessAffinity**(processor_list)

Sets the ioHub Process Affinity based on the value of processor_list.

A Process’s Affinity determines which CPU’s or CPU cores a process can run on. By default the ioHub Process can run on any CPU or CPU core.

The processor_list argument must be a list of ‘processor’ id’s; integers in the range of 0 to Computer.processing_unit_count-1, representing the processing unit indexes that the ioHub Server should be allowed to run on.

If processor_list is given as an empty list, the ioHub Process will be able to run on any processing unit on the computer.

This method is not supported on OS X at this time.

Parameters processor_list (list) – A list of integer values between 0 and Computer.processing_unit_count-1, where values in the list indicate processing unit indexes that the ioHub process is able to run on.

Returns None

**flushDataStoreFile**()

Manually tell the ioDataStore to flush any events it has buffered in memory to disk.”.

Parameters None –

Returns None

**startCustomTasklet**(task_name, task_class_path, **class_kwargs)

Instruct the ioHub server to start running a custom tasklet given by task_class_path. It is important that the custom task does not block for any significant amount of time, or the processing of events by the ioHub server will be negatively effected.

See the customtask.py demo for an example of how to make a long running task not block the rest of the ioHub server.

**stopCustomTasklet**(task_name)

Instruct the ioHub server to stop the custom task that was previously started by calling self.startCustomTasklet(…). task_name identifies which custom task should be stopped and must match the task_name of a previously started custom task.

**shutdown**()

Tells the ioHub Server to close all ioHub Devices, the ioDataStore, and the connection monitor between the PsychoPy and ioHub Processes. Then end the server process itself.

Parameters None –

Returns None

**quit**()

Same as the shutdown() method, but has same name as PsychoPy core.quit() so maybe easier to remember.

**_startServer**(ioHubConfig=None, ioHubConfigAbsPath=None)

Starts the ioHub Process, storing it's process id, and creating the experiment side device representation for IPC access to public device methods.
PsychoPy - Psychology software for Python, Release 2020.2.10

__createDeviceList__(monitor_devices_config)
Create clientsideiohub device views.

__addDeviceView__(dev_cls_name, dev_config)
Add an iohub device view to self.devices

__sendToHubServer__(tx_data)
General purpose local <-> iohub server process UDP based request - reply code. The method blocks until the request is fulfilled and and a response is received from the ioHub server.

Parameters tx_data (tuple) – data to send to iohub server

Return (object): response from the ioHub Server process.

__sendExperimentInfo__(experimentInfoDict)
Sends the experiment info from the experiment config file to the ioHub Server, which passes it to the ioDataStore, determines if the experiment already exists in the hdf5 file based on ‘experiment_code’, and returns a new or existing experiment ID based on that criteria.

__sendSessionInfo__(sess_info)
Sends the experiment session info from the experiment config file and the values entered into the session dialog to the ioHub Server, which passes it to the ioDataStore.

The dataStore determines if the session already exists in the experiment file based on ‘session_code’, and returns a new session ID if session_code is not in use by the experiment.

static __isErrorReply__(data)
Check if an iohub server reply contains an error that should be raised by the local process.

8.5.2 Supported Devices

psycho.pyohub supports several different types of devices, including Keyboards, Mice, and Eye Trackers.

Details for each device can be found in the following sections.

Keyboard Device

The iohub Keyboard device provides methods to:

- Check for any new keyboard events that have occurred since the last time keyboard events were checked or cleared.
- Wait until a keyboard event occurs.
- Clear the device of any unread events.
- Get a list of all currently pressed keys.

class psyche.pyohub.client.keyboard.Keyboard(ioclient, dev_cls_name, dev_config)
The Keyboard device provides access to KeyboardPress and KeyboardRelease events as well as the current keyboard state.
Examples

A. Print all keyboard events received for 5 seconds:

```python
from psychopy.iohub import launchHubServer
from psychopy.core import getTime

# Start the ioHub process. 'io' can now be used during the
# experiment to access iohub devices and read iohub device events.
io = launchHubServer()

keyboard = io.devices.keyboard

# Check for and print any Keyboard events received for 5 seconds.
stime = getTime()
while getTime() - stime < 5.0:
    for e in keyboard.getEvents():
        print(e)

# Stop the ioHub Server
io.quit()
```

B. Wait for a keyboard press event (max of 5 seconds):

```python
from psychopy.iohub import launchHubServer
from psychopy.core import getTime

# Start the ioHub process. 'io' can now be used during the
# experiment to access iohub devices and read iohub device events.
io = launchHubServer()

keyboard = io.devices.keyboard

# Wait for a key keypress event ( max wait of 5 seconds )
presses = keyboard.waitForPresses(maxWait=5.0)

print(presses)

# Stop the ioHub Server
io.quit()
```

_syncDeviceState()  
An optimized iohub server request that receives all device state and event information in one response.

Returns None

getKeys(keys=None, chars=None, mods=None, duration=None, etype=None, clear=True)  
Return a list of any KeyboardPress or KeyboardRelease events that have occurred since the last time either:

• this method was called with the kwarg clear=True (default)

• the keyboard.clear() method was called.

Other than the 'clear' kwarg, any kwargs that are not None or an empty list are used to filter the possible events that can be returned. If multiple filter criteria are provided, only events that match all specified criteria are returned.

If no KeyboardEvent’s are found that match the filtering criteria, an empty tuple is returned.

Returned events are sorted by time.
Parameters

- **keys** – Include events where .key in keys.
- **chars** – Include events where .char in chars.
- **mods** – Include events where .modifiers include >=1 mods element.
- **duration** – Include KeyboardRelease events where .duration > duration or .duration < -(duration).
- **etype** – Include events that match etype of Keyboard.KEY_PRESS or Keyboard.KEY_RELEASE.
- **clear** – True (default) = clear returned events from event buffer, False = leave the keyboard event buffer unchanged.

Returns tuple of KeyboardEvent instances, or ()

**getPresses** *(keys=None, chars=None, mods=None, clear=True)*

See the getKeys() method documentation.

This method is identical, but only returns KeyboardPress events.

**getReleases** *(keys=None, chars=None, mods=None, duration=None, clear=True)*

See the getKeys() method documentation.

This method is identical, but only returns KeyboardRelease events.

**property reporting**

Specifies if the the keyboard device is reporting / recording events.

- True: keyboard events are being reported.
- False: keyboard events are not being reported.

By default, the Keyboard starts reporting events automatically when the ioHub process is started and continues to do so until the process is stopped.

This property can be used to read or set the device reporting state:

```python
# Read the reporting state of the keyboard.
is_reporting_keyboard_event = keyboard.reporting

# Stop the keyboard from reporting any new events.
kkeyboard.reporting = False
```

**property state**

Returns all currently pressed keys as a dictionary of key

**waitForKeys** *(maxWait=None, keys=None, chars=None, mods=None, duration=None, etype=None, clear=True, checkInterval=0.002)*

Blocks experiment execution until at least one matching KeyboardEvent occurs, or until maxWait seconds has passed since the method was called.

Keyboard events are filtered the same way as in the getKeys() method.

As soon as at least one matching KeyboardEvent occurs prior to maxWait, the matching events are returned as a tuple.
Returned events are sorted by time.

Parameters

- **maxWait** – Maximum seconds method waits for >=1 matching event. If <=0.0, method functions the same as getKeys(). If None, the methods blocks indefinitely.
- **keys** – Include events where .key in keys.
- **chars** – Include events where .char in chars.
- **mods** – Include events where .modifiers include >=1 mods element.
- **duration** – Include KeyboardRelease events where .duration > duration or .duration < -(duration).
- **etype** – Include events that match etype of Keyboard.KEY_PRESS or Keyboard.KEY_RELEASE.
- **clear** – True (default) = clear returned events from event buffer, False = leave the keyboard event buffer unchanged.
- **checkInterval** – The time between geyKeys() calls while waiting. The method sleeps between geyKeys() calls, up until checkInterval*2.0 sec prior to the maxWait. After that time, keyboard events are constantly checked until the method times out.

Returns tuple of KeyboardEvent instances, or ()

**waitForPresses** *(maxWait=None, keys=None, chars=None, mods=None, duration=None, clear=True, checkInterval=0.002)*

See the waitForKeys() method documentation.

This method is identical, but only returns KeyboardPress events.

**waitForReleases** *(maxWait=None, keys=None, chars=None, mods=None, duration=None, clear=True, checkInterval=0.002)*

See the waitForKeys() method documentation.

This method is identical, but only returns KeyboardRelease events.

Keyboard Events

The Keyboard device can return two types of events, which represent key press and key release actions on the keyboard.

KeyboardPress Event

**class** `psychopy.iohub.client.keyboard.KeyboardPress` *(ioe_array)*

An iohub Keyboard device key press event.

**property** `char`

The unicode value of the keyboard event, if available. This field is only populated when the keyboard event results in a character that could be printable.

Returns unicode, “” if no char value is available for the event.

**property** `device`

The ioHubDeviceView that is associated with the event, i.e. the iohub device view for the device that generated the event.

Returns ioHubDeviceView
property modifiers
A list of any modifier keys that were pressed when this keyboard event occurred. Each element of the list contains a keyboard modifier string constant. Possible values are:

- ‘lctrl’, ‘rctrl’
- ‘lshift’, ‘rshift’
- ‘lalt’, ‘ralt’ (labelled as ‘option’ keys on Apple Keyboards)
- ‘lcmd’, ‘rcmd’ (map to the ‘windows’ key(s) on Windows keyboards)
- ‘menu’
- ‘capslock’
- ‘numlock’
- ‘function’ (OS X only)
- ‘modhelp’ (OS X only)

If no modifiers were active when the event occurred, an empty list is returned.

Returns tuple

property time
The time stamp of the event. Uses the same time base that is used by psychopy.core.getTime()

Returns float

property type
The event type string constant.

Returns str

KeyboardRelease Event

class psychopy.iohub.client.keyboard.KeyboardRelease(ioe_array)
An iohub Keyboard device key release event.

property duration
The duration (in seconds) of the key press. This is calculated by subtracting the current event.time from the associated keypress.time.

If no matching keypress event was reported prior to this event, then 0.0 is returned. This can happen, for example, when the key was pressed prior to psychopy starting to monitor the device. This condition can also happen when keyboard.reset() method is called between the press and release event times.

Returns float

property pressEventID
The event.id of the associated press event.

The key press id is 0 if no associated KeyboardPress event was found. See the duration property documentation for details on when this can occur.

Returns unsigned int

property char
The unicode value of the keyboard event, if available. This field is only populated when the keyboard event results in a character that could be printable.

Returns unicode, ‘‘ if no char value is available for the event.
property device

The ioHubDeviceView that is associated with the event, i.e. the iohub device view for the device that generated the event.

Returns ioHubDeviceView

property modifiers

A list of any modifier keys that were pressed when this keyboard event occurred. Each element of the list contains a keyboard modifier string constant. Possible values are:

- ‘lctrl’, ‘rctrl’
- ‘lshift’, ‘rshift’
- ‘lalt’, ‘ralt’ (labelled as ‘option’ keys on Apple Keyboards)
- ‘lcmd’, ‘rcmd’ (map to the ‘windows’ key(s) on Windows keyboards)
- ‘menu’
- ‘capslock’
- ‘numlock’
- ‘function’ (OS X only)
- ‘modhelp’ (OS X only)

If no modifiers were active when the event occurred, an empty list is returned.

Returns tuple

property time

The time stamp of the event. Uses the same time base that is used by psychopy.core.getTime()

Returns float

property type

The event type string constant.

Returns str

The ioHub Mouse Device

Platforms: Windows, macOS, Linux

Mouse Event Types

The Mouse device supports the following event types. Device events returned by getEvents() are automatically converted to either namedtuple or dictionary objects with the same attributes / keys as the associated event class attributes.
ioHub Common Eye Tracker Interface

The iohub common eye tracker interface provides a consistent way to configure and collected data from several different eye tracker manufacturers, including GazePoint, SR Research, and Tobii.

Supported Eye Trackers

The following eye trackers are currently supported by iohub.

Gazepoint

Platforms:
  • Windows 7 / 10 only

Required Python Version:
  • Python 3.6 +

Supported Models:
  • Gazepoint GP3

Additional Software Requirements

To use your Gazepoint GP3 during an experiment you must first start the Gazepoint Control software on the computer running PsychoPy.

EyeTracker Class

class psychopy.iohub.devices.eyetracker.hw.gazepoint gp3.EyeTracker

To start iohub with a Gazepoint GP3 eye tracker device, add a GP3 device to the device dictionary passed to launchHubServer or the experiment's iohub_config.yaml:

```
iohub_config = {'eyetracker.hw.gazepoint gp3.EyeTracker':
                {'name': 'tracker', 'device_timer': {'interval': 0.005}}}
```

Note: The Gazepoint control application must be running while using this interface.

Examples

A. Start ioHub with Gazepoint GP3 device and run tracker calibration:

```
from psychopy.iohub import launchHubServer
from psychopy.core import getTime, wait

iohub_config = {'eyetracker.hw.gazepoint gp3.EyeTracker':
                {'name': 'tracker', 'device_timer': {'interval': 0.005}}}

io = launchHubServer(**iohub_config)
```
# Get the eye tracker device.
tracker = io.devices.tracker

# run eyetracker calibration
r = tracker.runSetupProcedure()

B. Print all eye tracker events received for 2 seconds:

```python
# Check for and print any eye tracker events received...
tracker.setRecordingState(True)
stime = getTime()
while getTime() - stime < 2.0:
    for e in tracker.getEvents():
        print(e)
```

C. Print current eye position for 5 seconds:

```python
# Check for and print current eye position every 100 msec.
stime = getTime()
while getTime() - stime < 5.0:
    print(tracker.getPosition())
    wait(0.1)
tracker.setRecordingState(False)

# Stop the ioHub Server
io.quit()
```

clearEvents (event_type=None, filter_id=None, call_proc_events=True)
Clears any DeviceEvents that have occurred since the last call to the device’s getEvents(), or clearEvents() methods.

Note that calling clearEvents() at the device level only clears the given device’s event buffer. The ioHub Process’s Global Event Buffer is unchanged.

Parameters None –

Returns None

enableEventReporting (enabled=True)
enableEventReporting is functionally identical to the eye tracker device specific setRecordingState method.

getConfiguration ()
Retrieve the configuration settings information used to create the device instance. This will the default settings for the device, found in iohub.devices.<device_name>.default_<device_name>.yaml, updated with any device settings provided via launchHubServer(...).

Changing any values in the returned dictionary has no effect on the device state.

Parameters None –

Returns The dictionary of the device configuration settings used to create the device.

Return type (dict)

getEvents (*args, **kwargs)
Retrieve any DeviceEvents that have occurred since the last call to the device’s getEvents() or clearEvents() methods.
Note that calling getEvents() at a device level does not change the Global Event Buffer’s contents.

**Parameters**

- **event_type_id (int)** – If specified, provides the ioHub DeviceEvent ID for which events should be returned for. Events that have occurred but do not match the event ID specified are ignored. Event type ID’s can be accessed via the EventConstants class; all available event types are class attributes of EventConstants.

- **clearEvents (int)** – Can be used to indicate if the events being returned should also be removed from the device event buffer. True (the default) indicates to remove events being returned. False results in events being left in the device event buffer.

- **asType (str)** – Optional kwarg giving the object type to return events as. Valid values are ‘namedtuple’ (the default), ‘dict’, ‘list’, or ‘object’.

**Returns**

New events that the ioHub has received since the last getEvents() or clearEvents() call to the device. Events are ordered by the ioHub time of each event, older event at index 0. The event object type is determined by the asType parameter passed to the method. By default a namedtuple object is returned for each event.

**Return type** (list)

**getLastGazePosition()**

The getLastGazePosition method returns the most recent eye gaze position received from the Eye Tracker. This is the position on the calibrated 2D surface that the eye tracker is reporting as the current eye position. The units are in the units in use by the ioHub Display device.

If binocular recording is being performed, the average position of both eyes is returned.

If no samples have been received from the eye tracker, or the eye tracker is not currently recording data, None is returned.

**Parameters** None –

**Returns**

If this method is not supported by the eye tracker interface, EyeTrackerConstants.EYETRACKER_INTERFACE_METHOD_NOT_SUPPORTED is returned.

None: If the eye tracker is not currently recording data or no eye samples have been received.

tuple: Latest (gaze_x,gaze_y) position of the eye(s)

**Return type** int

**getLastSample()**

The getLastSample method returns the most recent eye sample received from the Eye Tracker. The Eye Tracker must be in a recording state for a sample event to be returned, otherwise None is returned.

**Parameters** None –

**Returns**

If this method is not supported by the eye tracker interface, EyeTrackerConstants.FUNCTIONALITY_NOT_SUPPORTED is returned.

None: If the eye tracker is not currently recording data.

EyeSample: If the eye tracker is recording in a monocular tracking mode, the latest sample event of this event type is returned.

BinocularEyeSample: If the eye tracker is recording in a binocular tracking mode, the latest sample event of this event type is returned.
Return type: `int`

`getPosition()`

The `getPosition` method is the same as the `getLastGazePosition` method, provided as a consistent cross device method to access the current screen position reported by a device.

See `getLastGazePosition` for further details.

`isRecordingEnabled()`

`isRecordingEnabled` returns the recording state from the eye tracking device.

**Parameters**

None

**Returns**

True == the device is recording data; False == Recording is not occurring

Return type: `bool`

`runSetupProcedure()`

`runSetupProcedure` opens the GP3 Calibration window.

`setRecordingState(recording)`

`setRecordingState` is used to start or stop the recording of data from the eye tracking device.

**Parameters**

`recording` (`bool`) – if True, the eye tracker will start recording available eye data and sending it to the experiment program if data streaming was enabled for the device. If recording == False, then the eye tracker stops recording eye data and streaming it to the experiment.

If the eye tracker is already recording, and `setRecordingState(True)` is called, the eye tracker will simply continue recording and the method call is a no-op. Likewise if the system has already stopped recording and `setRecordingState(False)` is called again.

**Parameters**

`recording` (`bool`) – if True, the eye tracker will start recording data.; false = stop recording data.

Return: `trackerTime` bool: the current recording state of the eye tracking device

`trackerSec()`

Same as the GP3 implementation of `trackerTime()`.

`trackerTime()`

Current eye tracker time in the eye tracker’s native time base. The GP3 system uses a sec.usec timebase based on the Windows QPC.

**Parameters**

None

**Returns**

current native eye tracker time in sec.msec format.

Return type: `float`

**Supported Event Types**

The Gazepoint GP3 provides real-time access to binocular sample data. iohub creates a `BinocularEyeSampleEvent` for each sample received from the GP3.

The following fields of the `BinocularEyeSampleEvent` event are supported:

```python
class psychopy.iohub.devices.eyetracker.BinocularEyeSampleEvent(object)
```

The `BinocularEyeSampleEvent` event represents the eye position and eye attribute data collected from one frame or reading of an eye tracker device that is recording both eyes of a participant.

Event Type ID: `EventConstants.BINOCULAR_EYE_SAMPLE`
Event Type String: ‘BINOCULAR_EYE_SAMPLE’

time

time of event, in sec.msec format, using psychopy timebase.

left_gaze_x

The horizontal position of the left eye on the computer screen, in Display Coordinate Type Units. Calibration must be done prior to reading (meaningful) gaze data. Uses Gazepoint LPOGX field.

left_gaze_y

The vertical position of the left eye on the computer screen, in Display Coordinate Type Units. Calibration must be done prior to reading (meaningful) gaze data. Uses Gazepoint LPOGY field.

left_raw_x

The uncalibrated x position of the left eye in a device specific coordinate space. Uses Gazepoint LPCX field.

left_raw_y

The uncalibrated y position of the left eye in a device specific coordinate space. Uses Gazepoint LPCY field.

left_pupil_measure_1

Left eye pupil diameter. (in camera pixels??). Uses Gazepoint LPD field.

right_gaze_x

The horizontal position of the right eye on the computer screen, in Display Coordinate Type Units. Calibration must be done prior to reading (meaningful) gaze data. Uses Gazepoint RPOGX field.

right_gaze_y

The vertical position of the right eye on the computer screen, in Display Coordinate Type Units. Calibration must be done prior to reading (meaningful) gaze data. Uses Gazepoint RPOGY field.

right_raw_x

The uncalibrated x position of the right eye in a device specific coordinate space. Uses Gazepoint RPCX field.

right_raw_y

The uncalibrated y position of the right eye in a device specific coordinate space. Uses Gazepoint RPCY field.

right_pupil_measure_1

Right eye pupil diameter. (in camera pixels??). Uses Gazepoint RPD field.

status

Indicates if eye sample contains ‘valid’ data for left and right eyes. 0 = Eye sample is OK. 2 = Right eye data is likely invalid. 20 = Left eye data is likely invalid. 22 = Eye sample is likely invalid.

iohub also creates basic start and end fixation events by using Gazepoint FPOG* fields. Identical / duplicate fixation events are created for the left and right eye.

class psychopy.iohub.devices.eyetracker.FixationStartEvent (object)

A FixationStartEvent is generated when the beginning of an eye fixation (in very general terms, a period of relatively stable eye position) is detected by the eye trackers sample parsing algorithms.

Event Type ID: EventConstants.FIXATION_START

Event Type String: ‘FIXATION_START’

time

time of event, in sec.msec format, using psychopy timebase.

eye

Eye that generated the event. Either EyeTrackerConstants.LEFT_EYE or EyeTrackerConstants.RIGHT_EYE.
gaze_x
The calibrated horizontal eye position on the computer screen at the start of the fixation. Units are same as Display. Calibration must be done prior to reading (meaningful) gaze data. Uses Gazepoint FPOGX field.

gaze_y
The calibrated horizontal eye position on the computer screen at the start of the fixation. Units are same as Display. Calibration must be done prior to reading (meaningful) gaze data. Uses Gazepoint FPOGY field.

class psychopy.iohub.devices.eyetracker.FixationEndEvent(object)
A FixationEndEvent is generated when the end of an eye fixation (in very general terms, a period of relatively stable eye position) is detected by the eye trackers sample parsing algorithms.

Event Type ID: EventConstants.FIXATION_END
Event Type String: ‘FIXATION_END’

time
time of event, in sec.msec format, using psychopy timebase.

eye
Eye that generated the event. Either EyeTrackerConstants.LEFT_EYE or EyeTrackerConstants.RIGHT_EYE.

average_gaze_x
Average calibrated horizontal eye position during the fixation, specified in Display Units. Uses Gazepoint FPOGX field.

average_gaze_y
Average calibrated vertical eye position during the fixation, specified in Display Units. Uses Gazepoint FPOGY field.

duration
Duration of the fixation in sec.msec format. Uses Gazepoint FPOGD field.

Default Device Settings

eyetracker.hw.gazepoint.gp3.EyeTracker:

    # Indicates if the device should actually be loaded at experiment runtime.
    enable: True

    # The variable name of the device that will be used to access the ioHub Device_class
    name: tracker

    # Should eye tracker events be saved to the ioHub DataStore file when the device # is recording data ?
    save_events: True

    # Should eye tracker events be sent to the Experiment process when the device # is recording data ?
    stream_events: True

    # How many eye events (including samples) should be saved in the ioHub event_buffer before
    # old eye events start being replaced by new events. When the event buffer reaches
    # the maximum event length of the buffer defined here, older events will start to...
    # be dropped.

(continues on next page)
event_buffer_length: 1024

# The GP3 implementation of the common eye tracker interface supports the
# BinocularEyeSampleEvent event type.
monitor_event_types: [BinocularEyeSampleEvent, FixationStartEvent,...
→FixationEndEvent]

device_timer:
  interval: 0.005

calibration:
  # target_duration is the number of sec.msec that a calibration point should
  # be displayed before moving onto the next point.
  # (Sets the GP3 CALIBRATE_TIMEOUT)
  target_duration: 1.25
  # target_delay specifies the target animation duration in sec.msec.
  # (Sets the GP3 CALIBRATE_DELAY)
  target_delay: 0.5

# The model name of the device.
model_name: GP3

# The serial number of the GP3 device.
serial_number:

# manufacturer_name is used to store the name of the maker of the eye tracking
# device. This is for informational purposes only.
manufacturer_name: GazePoint

Last Updated: April, 2019

SR Research

Platforms:
  • Windows 7 / 10
  • Linux
  • macOS

Required Python Version:
  • Python 3.6 +

Supported Models:
  • EyeLink 1000
  • EyeLink 1000 Plus
Additional Software Requirements

The SR Research EyeLink implementation of the ioHub common eye tracker interface uses the pylink package written by SR Research. If using a PsychoPy3 standalone installation, this package should already be included.

If you are manually installing PsychoPy3, please install the appropriate version of pylink. Downloads are available to SR Research customers from their support website.

EyeTracker Class

class psychopy.iohub.devices.eyetracker.hw.sr_research.eyelink.EyeTracker

The SR Research EyeLink implementation of the Common Eye Tracker Interface can be used by providing the following EyeTracker path as the device class in the iohub_config.yaml device settings file:

   eyetracker.hw.sr_research.eyelink

Examples

A. Start ioHub with SR Research EyeLink 1000 and run tracker calibration:

```python
from psychopy.iohub import launchHubServer
from psychopy.core import getTime, wait

iohub_config = {'eyetracker.hw.sr_research.eyelink.EyeTracker':
    {'name': 'tracker',
     'model_name': 'EYELINK 1000 DESKTOP',
     'runtime_settings': {'sampling_rate': 500,
                          'track_eyes': 'RIGHT'}
    }
}

io = launchHubServer(**iohub_config)

# Get the eye tracker device.
tracker = io.devices.tracker

# run eyetracker calibration
r = tracker.runSetupProcedure()
```

B. Print all eye tracker events received for 2 seconds:

```python
# Check for and print any eye tracker events received...
tracker.setRecordingState(True)

stime = getTime()
while getTime() - stime < 2.0:
    for e in tracker.getEvents():
        print(e)
```

C. Print current eye position for 5 seconds:

```python
# Check for and print current eye position every 100 msec.
stime = getTime()
while getTime() - stime < 5.0:
    print(tracker.getPosition())
```

(continues on next page)
wait(0.1)

tracker.setRecordingState(False)

# Stop the ioHub Server
io.quit()

clearEvents(event_type=None, filter_id=None, call_proc_events=True)
Clears any DeviceEvents that have occurred since the last call to the device's getEvents(), or clearEvents() methods.

Note that calling clearEvents() at the device level only clears the given device's event buffer. The ioHub Process's Global Event Buffer is unchanged.

Parameters
None –

Returns
None

enableEventReporting(enabled=True)
enableEventReporting is the device type independent method that is equivalent to the EyeTracker specific setRecordingState method.

getConfiguration()
Retrieve the configuration settings information used to create the device instance. This will the default settings for the device, found in iohub.devices.<device_name>.default_<device_name>.yaml, updated with any device settings provided via launchHubServer(...).

Changing any values in the returned dictionary has no effect on the device state.

Parameters
None –

Returns
The dictionary of the device configuration settings used to create the device.

Return type
dict

getEvents(*args, **kwargs)
Retrieve any DeviceEvents that have occurred since the last call to the device's getEvents() or clearEvents() methods.

Note that calling getEvents() at a device level does not change the Global Event Buffer's contents.

Parameters

• event_type_id (int) – If specified, provides the ioHub DeviceEvent ID for which events should be returned for. Events that have occurred but do not match the event ID specified are ignored. Event type ID's can be accessed via the EventConstants class; all available event types are class attributes of EventConstants.

• clearEvents (int) – Can be used to indicate if the events being returned should also be removed from the device event buffer. True (the default) indicates to remove events being returned. False results in events being left in the device event buffer.

• asType (str) – Optional kwarg giving the object type to return events as. Valid values are 'namedtuple' (the default), 'dict', 'list', or 'object'.

Returns
New events that the ioHub has received since the last getEvents() or clearEvents() call to the device. Events are ordered by the ioHub time of each event, older event at index 0. The event object type is determined by the asType parameter passed to the method. By default a namedtuple object is returned for each event.

Return type
list
**getLastGazePosition()**

getLastGazePosition returns the most recent x,y eye position, in Display device coordinate space, received by the ioHub server from the EyeLink device. In the case of binocular recording, and if both eyes are successfully being tracked, then the average of the two eye positions is returned. If the eye tracker is not recording or is not connected, then None is returned. The getLastGazePosition method returns the most recent eye gaze position retrieved from the eye tracker device. This is the position on the calibrated 2D surface that the eye tracker is reporting as the current eye position. The units are in the units in use by the Display device.

If binocular recording is being performed, the average position of both eyes is returned. If no samples have been received from the eye tracker, or the eye tracker is not currently recording data, None is returned.

**Parameters** None –

**Returns**

If the eye tracker is not currently recording data or no eye samples have been received.

tuple: Latest (gaze_x, gaze_y) position of the eye(s)

**Return type** None

**getLastSample()**

getLastSample returns the most recent EyeSampleEvent received from the EyeLink system. Any position fields are in Display device coordinate space. If the eye tracker is not recording or is not connected, then None is returned.

**Parameters** None –

**Returns**

If the eye tracker is not currently recording data.

EyeSample: If the eye tracker is recording in a monocular tracking mode, the latest sample event of this event type is returned.

BinocularEyeSample: If the eye tracker is recording in a binocular tracking mode, the latest sample event of this event type is returned.

**Return type** None

**getPosition()**

The getPosition method is the same as the getLastGazePosition method, provided as a consistent cross device method to access the current screen position reported by a device.

See getLastGazePosition for further details.

**isRecordingEnabled()**

isRecordingEnabled returns True if the eye tracking device is currently connected and sending eye event data to the ioHub server. If the eye tracker is not recording, or is not connected to the ioHub server, False will be returned.

**Parameters** None –

**Returns** True == the device is recording data; False == Recording is not occurring

**Return type** bool

**runSetupProcedure()**

Start the EyeLink Camera Setup and Calibration procedure.

During the system setup, the following keys can be used on either the Host PC or Experiment PC to control the state of the setup procedure:
• C = Start Calibration
• V = Start Validation
• ENTER should be pressed at the end of a calibration or validation to accept the calibration, or in the case of validation, use the option drift correction that can be performed as part of the validation process in the EyeLink system.
• ESC can be pressed at any time to exit the current state of the setup procedure and return to the initial blank screen state.
• O = Exit the runSetupProcedure method and continue with the experiment.

sendCommand(key, value=None)
The sendCommand method sends an EyeLink command key and value to the EyeLink device. Any valid EyeLink command can be sent using this method. However, not that doing so is a device dependent operation, and will have no effect on other implementations of the Common EyeTracker Interface, unless the other eye tracking device happens to support the same command, value format.

If both key and value are provided, internally they are combined into a string of the form:

“key = value”

and this is sent to the EyeLink device. If only key is provided, it is assumed to include both the command name and any value or arguements required by the EyeLink all in the one arguement, which is sent to the EyeLink device untouched.

sendMessage(message_contents, time_offset=None)
The sendMessage method sends a string (max length 128 characters) to the EyeLink device.

The message will be time stamped and inserted into the native EDF file, if one is being recorded. If no native EyeLink data file is being recorded, this method is a no-op.

setRecordingState(recording)
setRecordingState enables (recording=True) or disables (recording=False) the recording of eye data by the eye tracker and the sending of any eye data to the ioHub Server. The eye tracker must be connected to the ioHub Server by using the setConnectionState(True) method for recording to be possible.

Parameters recording(bool) – if True, the eye tracker will start recordng data.; false = stop recording data.

Returns the current recording state of the eye tracking device

Return type bool

trackerSec()
trackerSec returns the current EyeLink Host Application time in sec.msec format.

trackerTime()
trackerTime returns the current EyeLink Host Application time in msec format as a long integer.

Supported Event Types

The EyeLink implementation of the ioHub eye tracker interface supports monoculor or binocular eye samples as well as fixation, saccade, and blink events.
Eye Samples

class psychopy.iohub.devices.eyetracker.MonocularEyeSampleEvent(object)

A MonocularEyeSampleEvent represents the eye position and eye attribute data collected from one frame or reading of an eye tracker device that is recoding from only one eye, or is recording from both eyes and averaging the binocular data.

Event Type ID: EventConstants.MONOCULAR_EYE_SAMPLE
Event Type String: 'MONOCULAR_EYE_SAMPLE'

time
time of event, in sec.msec format, using psychopy timebase.

eye
Eye that generated the sample. Either EyeTrackerConstants.LEFT_EYE or EyeTrackerConstants.RIGHT_EYE.

gaze_x
The horizontal position of the eye on the computer screen, in Display Coordinate Type Units. Calibration must be done prior to reading (meaningful) gaze data.

gaze_y
The vertical position of the eye on the computer screen, in Display Coordinate Type Units. Calibration must be done prior to reading (meaningful) gaze data.

angle_x
Horizontal eye angle.

angle_y
Vertical eye angle.

raw_x
The uncalibrated x position of the eye in a device specific coordinate space.

raw_y
The uncalibrated y position of the eye in a device specific coordinate space.

pupil_measure_1
Pupil size. Use pupil_measure1_type to determine what type of pupil size data was being saved by the tracker.

pupil_measure1_type
Coordinate space type being used for left_pupil_measure_1.

ppd_x
Horizontal pixels per visual degree for this eye position as reported by the eye tracker.

ppd_y
Vertical pixels per visual degree for this eye position as reported by the eye tracker.

velocity_x
Horizontal velocity of the eye at the time of the sample; as reported by the eye tracker.

velocity_y
Vertical velocity of the eye at the time of the sample; as reported by the eye tracker.

velocity_xy
2D Velocity of the eye at the time of the sample; as reported by the eye tracker.

status
Indicates if eye sample contains ‘valid’ data. 0 = Eye sample is OK. 2 = Eye sample is invalid.
class psychopy.iohub.devices.eyetracker.BinocularEyeSampleEvent(object)

The BinocularEyeSampleEvent event represents the eye position and eye attribute data collected from one frame or reading of an eye tracker device that is recording both eyes of a participant.

Event Type ID: EventConstants.BINOCULAR_EYE_SAMPLE
Event Type String: ‘BINOCULAR_EYE_SAMPLE’

time
time of event, in sec.msec format, using psychopy timebase.

left_gaze_x
The horizontal position of the left eye on the computer screen, in Display Coordinate Type Units. Calibration must be done prior to reading (meaningful) gaze data.

left_gaze_y
The vertical position of the left eye on the computer screen, in Display Coordinate Type Units. Calibration must be done prior to reading (meaningful) gaze data.

left_angle_x
The horizontal angle of left eye the relative to the head.

left_angle_y
The vertical angle of left eye the relative to the head.

left_raw_x
The uncalibrated x position of the left eye in a device specific coordinate space.

left_raw_y
The uncalibrated y position of the left eye in a device specific coordinate space.

left_pupil_measure_1
Left eye pupil diameter.

left_pupil_measure1_type
Coordinate space type being used for left_pupil_measure_1.

left_ppd_x
Pixels per degree for left eye horizontal position as reported by the eye tracker. Display distance must be correctly set for this to be accurate at all.

left_ppd_y
Pixels per degree for left eye vertical position as reported by the eye tracker. Display distance must be correctly set for this to be accurate at all.

left_velocity_x
Horizontal velocity of the left eye at the time of the sample; as reported by the eye tracker.

left_velocity_y
Vertical velocity of the left eye at the time of the sample; as reported by the eye tracker.

left_velocity_xy
2D Velocity of the left eye at the time of the sample; as reported by the eye tracker.

right_gaze_x
The horizontal position of the right eye on the computer screen, in Display Coordinate Type Units. Calibration must be done prior to reading (meaningful) gaze data.

right_gaze_y
The vertical position of the right eye on the computer screen, in Display Coordinate Type Units. Calibration must be done prior to reading (meaningful) gaze data.
right_angle_x
The horizontal angle of right eye relative to the head.

right_angle_y
The vertical angle of right eye relative to the head.

right_raw_x
The uncalibrated x position of the right eye in a device specific coordinate space.

right_raw_y
The uncalibrated y position of the right eye in a device specific coordinate space.

right_pupil_measure_1
Right eye pupil diameter.

right_pupil_measure1_type
Coordinate space type being used for right_pupil_measure1_type.

right_ppd_x
Pixels per degree for right eye horizontal position as reported by the eye tracker. Display distance must be correctly set for this to be accurate at all.

right_ppd_y
Pixels per degree for right eye vertical position as reported by the eye tracker. Display distance must be correctly set for this to be accurate at all.

right_velocity_x
Horizontal velocity of the right eye at the time of the sample; as reported by the eye tracker.

right_velocity_y
Vertical velocity of the right eye at the time of the sample; as reported by the eye tracker.

right_velocity_xy
2D Velocity of the right eye at the time of the sample; as reported by the eye tracker.

status
Indicates if eye sample contains ‘valid’ data for left and right eyes. 0 = Eye sample is OK. 2 = Right eye data is likely invalid. 20 = Left eye data is likely invalid. 22 = Eye sample is likely invalid.

Fixation Events

Successful eye tracker calibration must be performed prior to reading (meaningful) fixation event data.

class psychopy.iohub.devices.eyetracker.FixationStartEvent (object)
A FixationStartEvent is generated when the beginning of an eye fixation (in very general terms, a period of relatively stable eye position) is detected by the eye trackers sample parsing algorithms.

Event Type ID: EventConstants.FIXATION_START
Event Type String: ‘FIXATION_START’

time
time of event, in sec.msec format, using psychopy timebase.

eye
Eye that generated the event. Either EyeTrackerConstants.LEFT_EYE or EyeTrackerConstants.RIGHT_EYE.

gaze_x
Horizontal gaze position at the start of the event, in Display Coordinate Type Units.
BBox: 144,220 890,246

```python
gaze_y
Vertical gaze position at the start of the event, in Display Coordinate Type Units.

angle_x
Horizontal eye angle at the start of the event.

angle_y
Vertical eye angle at the start of the event.

pupil_measure_1
Pupil size. Use pupil_measure1_type to determine what type of pupil size data was being saved by the tracker.

pupil_measure1_type
EyeTrackerConstants.PUPIL_AREA

ppd_x
Horizontal pixels per degree at start of event.

ppd_y
Vertical pixels per degree at start of event.

velocity_xy
2D eye velocity at the start of the event.

status
Event status as reported by the eye tracker.

class psychopy.iohub.devices.eyetracker.FixationEndEvent(object)
A FixationEndEvent is generated when the end of an eye fixation (in very general terms, a period of relatively
stable eye position) is detected by the eye trackers sample parsing algorithms.

Event Type ID: EventConstants.FIXATION_END
Event Type String: 'FIXATION_END'

time
time of event, in sec.msec format, using psychopy timebase.

eye
Eye that generated the event. Either EyeTrackerConstants.LEFT_EYE or EyeTrackerCon-
stants.RIGHT_EYE.

duration
Duration of the event in sec.msec format.

start_gaze_x
Horizontal gaze position at the start of the event, in Display Coordinate Type Units.

start_gaze_y
Vertical gaze position at the start of the event, in Display Coordinate Type Units.

start_angle_x
Horizontal eye angle at the start of the event.

start_angle_y
Vertical eye angle at the start of the event.

start_pupil_measure_1
Pupil size at the start of the event.

start_pupil_measure1_type
EyeTrackerConstants.PUPIL_AREA
```
start_ppd_x
   Horizontal pixels per degree at start of event.

start_ppd_y
   Vertical pixels per degree at start of event.

start_velocity_xy
   2D eye velocity at the start of the event.

deraadge_x
   Horizontal gaze position at the end of the event, in Display Coordinate Type Units.

deraadge_y
   Vertical gaze position at the end of the event, in Display Coordinate Type Units.

end_angle_x
   Horizontal eye angle at the end of the event.

end_angle_y
   Vertical eye angle at the end of the event.

dern pupil measure_1
   Pupil size at the end of the event.

end_pupil_measure1_type
   EyeTrackerConstants.PUPIL_AREA

deraadge_x
   Horizontal pixels per degree at end of event.

end_ppd_y
   Vertical pixels per degree at end of event.

end_velocity_xy
   2D eye velocity at the end of the event.

average_gaze_x
   Average horizontal gaze position during the event, in Display Coordinate Type Units.

average_gaze_y
   Average vertical gaze position during the event, in Display Coordinate Type Units.

average_angle_x
   Average horizontal eye angle during the event.

average_angle_y
   Average vertical eye angle during the event.

average_pupil_measure1
   Average pupil size during the event.

average_pupil_measure1_type
   EyeTrackerConstants.PUPIL_AREA

average_velocity_xy
   Average 2D velocity of the eye during the event.

peak_velocity_xy
   Peak 2D velocity of the eye during the event.

status
   Event status as reported by the eye tracker.
**Saccade Events**

Successful eye tracker calibration must be performed prior to reading (meaningful) saccade event data.

```python
class psychopy.iohub.devices.eyetracker.SaccadeStartEvent(object)
```

- **time**
  - time of event, in sec.msec format, using psychopy timebase.

- **eye**
  - Eye that generated the event. Either EyeTrackerConstants.LEFT_EYE or EyeTrackerConstants.RIGHT_EYE.

- **gaze_x**
  - Horizontal gaze position at the start of the event, in Display Coordinate Type Units.

- **gaze_y**
  - Vertical gaze position at the start of the event, in Display Coordinate Type Units.

- **angle_x**
  - Horizontal eye angle at the start of the event.

- **angle_y**
  - Vertical eye angle at the start of the event.

- **pupil_measure_1**
  - Pupil size. Use pupil_measure1_type to determine what type of pupil size data was being saved by the tracker.

- **pupil_measure1_type**
  - EyeTrackerConstants.PUPIL_AREA

- **ppd_x**
  - Horizontal pixels per degree at start of event.

- **ppd_y**
  - Vertical pixels per degree at start of event.

- **velocity_xy**
  - 2D eye velocity at the start of the event.

- **status**
  - Event status as reported by the eye tracker.

```python
class psychopy.iohub.devices.eyetracker.SaccadeEndEvent(object)
```

- **time**
  - time of event, in sec.msec format, using psychopy timebase.

- **eye**
  - Eye that generated the event. Either EyeTrackerConstants.LEFT_EYE or EyeTrackerConstants.RIGHT_EYE.

- **duration**
  - Duration of the event in sec.msec format.

- **start_gaze_x**
  - Horizontal gaze position at the start of the event, in Display Coordinate Type Units.

- **start_gaze_y**
  - Vertical gaze position at the start of the event, in Display Coordinate Type Units.
start_angle_x
Horizontal eye angle at the start of the event.

start_angle_y
Vertical eye angle at the start of the event.

start_pupil_measure_1
Pupil size at the start of the event.

start_pupil_measure1_type
EyeTrackerConstants.PUPIL_AREA

start_ppd_x
Horizontal pixels per degree at start of event.

start_ppd_y
Vertical pixels per degree at start of event.

start_velocity_xy
2D eye velocity at the start of the event.

end_gaze_x
Horizontal gaze position at the end of the event, in Display Coordinate Type Units.

end_gaze_y
Vertical gaze position at the end of the event, in Display Coordinate Type Units.

end_angle_x
Horizontal eye angle at the end of the event.

end_angle_y
Vertical eye angle at the end of the event.

end_pupil_measure_1
Pupil size at the end of the event.

end_pupil_measure1_type
EyeTrackerConstants.PUPIL_AREA

end_ppd_x
Horizontal pixels per degree at end of event.

end_ppd_y
Vertical pixels per degree at end of event.

end_velocity_xy
2D eye velocity at the end of the event.

average_gaze_x
Average horizontal gaze position during the event, in Display Coordinate Type Units.

average_gaze_y
Average vertical gaze position during the event, in Display Coordinate Type Units.

average_angle_x
Average horizontal eye angle during the event,

average_angle_y
Average vertical eye angle during the event,

average_pupil_measure_1
Average pupil size during the event.
average_pupil_measure1_type
    EyeTrackerConstants.PUPIL_AREA

average_velocity_xy
    Average 2D velocity of the eye during the event.

peak_velocity_xy
    Peak 2D velocity of the eye during the event.

status
    Event status as reported by the eye tracker.

Blink Events

class psychopy.iohub.devices.eyetracker.BlinkStartEvent (object)

    time
        time of event, in sec.msec format, using psychopy timebase.

    eye
        Eye that generated the event. Either EyeTrackerConstants.LEFT_EYE or EyeTrackerCon-
        stants.RIGHT_EYE.

    status
        Event status as reported by the eye tracker.

class psychopy.iohub.devices.eyetracker.BlinkEndEvent (object)

    time
        time of event, in sec.msec format, using psychopy timebase.

    eye
        Eye that generated the event. Either EyeTrackerConstants.LEFT_EYE or EyeTrackerCon-
        stants.RIGHT_EYE.

    duration
        Blink duration, in sec.msec format.

    status
        Event status as reported by the eye tracker.

Default Device Settings

# This section includes all valid sr_research.eyelink.EyeTracker Device
# settings that can be specified in an iohub_config.yaml
# or in a Python dictionary form and passed to the launchHubServer
# method. Any device parameters not specified when the device class is
# created by the ioHub Process will be assigned the default value
# indicated here.
#
# eyetracker.hw.sr_research.eyelink.EyeTracker:
#    # name: The unique name to assign to the device instance created.
#    # The device is accessed from within the PsychoPy script
#    # using the name's value; therefore it must be a valid Python
#    # variable name as well.

(continues on next page)
# name: tracker

# enable: Specifies if the device should be enabled by ioHub and monitored for events.
# True = Enable the device on the ioHub Server Process
# False = Disable the device on the ioHub Server Process. No events for this device will be reported by the ioHub Server.
# enable: True

# saveEvents: *If* the ioHubDataStore is enabled for the experiment, then indicate if events for this device should be saved to the data_collection/keyboard event group in the hdf5 event file.
# True = Save events for this device to the ioDataStore.
# False = Do not save events for this device in the ioDataStore.
# saveEvents: True

# streamEvents: Indicate if events from this device should be made available during experiment runtime to the PsychoPy Process.
# True = Send events for this device to the PsychoPy Process in real-time.
# False = Do *not* send events for this device to the PsychoPy Process in real-time.
# streamEvents: True

# auto_report_events: Indicate if events from this device should start being processed by the ioHub as soon as the device is loaded at the start of an experiment, or if events should only start to be monitored on the device when a call to the device's enableEventReporting method is made with a parameter value of True.
# True = Automatically start reporting events for this device when the experiment starts.
# False = Do not start reporting events for this device until enableEventReporting(True) is set for the device during experiment runtime.
# auto_report_events: False

# event_buffer_length: Specify the maximum number of events (for each event type the device produces) that can be stored by the ioHub Server before each new event results in the oldest event of the same type being discarded from the ioHub device event buffer.
# event_buffer_length: 1024

# device_timer: The EyeLink EyeTracker class uses the polling method to check for new events received from the EyeTracker device.
# device_timer.interval specifies the sec.msec time between device polls.
# 0.001 = 1 msec, so the device will be polled at a rate of 1000 Hz.
# device_timer:
#   interval: 0.001

# monitor_event_types: The eyelink implementation of the common eye tracker interface supports the following event types. If you would like to
# exclude certain events from being saved or streamed during runtime,
# remove them from the list below.
#
monitor_event_types: [MonocularEyeSampleEvent, BinocularEyeSampleEvent,
-FixationStartEvent, FixationEndEvent, SaccadeStartEvent, SaccadeEndEvent,
-BlinkStartEvent, BlinkEndEvent]

calibration:
# IMPORTANT: Note that while the gaze position data provided by ioHub
# will be in the Display’s coordinate system, the EyeLink internally
# always uses a 0,0 pixel_width, pixel_height coordinate system
# since internally calibration point positions are given as integers,
# so if the actual display coordinate system was passed to EyeLink,
# coordinate types like deg and norm would become very coarse in
# possible target locations during calibration.
#
# type: sr_research.eyelink.EyeTracker supports the following
# calibration types:
# THREE_POINTS, FIVE_POINTS, NINE_POINTS, THIRTEEN_POINTS

type: NINE_POINTS
#
# auto_pace: If True, the eye tracker will automatically progress from
# one calibration point to the next. If False, a manual key or button press
# is needed to progress to the next point.
#
auto_pace: True
#
# pacing_speed: The number of sec.msec that a calibration point should
# be displayed before moving onto the next point when auto_pace is set to
# true.
#
# If auto_pace is False, pacing_speed is ignored.
#
pacing_speed: 1.5
#
# screen_background_color: Specifies the r,g,b,a background color to
# set the calibration, validation, etc. screens to. Each element of the
# color should be a value between 0 and 255. 0 == black, 255 == white. In general
# the last value of the color list (alpha) can be left at 255, indicating
# the color not mixed with the background color at all.
screen_background_color: [128,128,128,255]
#
# target_type: Defines what form of calibration graphic should be used
# during calibration, validation, etc. modes. sr_research.eyelink.EyeTracker
# supports the CIRCLE_TARGET type.
#
target_type: CIRCLE_TARGET
#
# target_attributes: The associated target attributes must be supplied
# for the given target_type. If target type attribute sections are provided
# for target types other than the entry associated with the specified
# target_type value they will simple be ignored.
#
target_attributes:
# outer_diameter and inner_diameter are specified in pixels
  outer_diameter: 33
  inner_diameter: 6

(continues on next page)
outer_color: [255,255,255,255]
inner_color: [0,0,0,255]

# network_settings: Specify the Host computer IP address. Normally
# leaving it set to the default value is fine.
#
# network_settings: 100.1.1.1

# default_native_data_file_name: The sr_research.eyelink.EyeTracker supports
# saving a native eye tracker edf data file, the
# default_native_data_file_name value is used to set the default name for
# the file that will be saved, not including the .edf file type extension.
#
# default_native_data_file_name: et_data

# simulation_mode: Indicate if the eye tracker should provide mouse simulated
# eye data instead of sending eye data based on a participants actual
# eye movements.
#
# simulation_mode: False

# enable_interface_without_connection: Specifying if the ioHub Device
# should be enabled without truly connecting to the underlying eye tracking
# hardware. If True, ioHub EyeTracker methods can be called but will
# provide no-op results and no eye data will be received by the ioHub Server.
# This mode can be useful for working on aspects of an eye tracking experiment
# when the
# actual eye tracking device is not available, for example stimulus presentation
# or other non eye tracker dependent experiment functionality.
#
# enable_interface_without_connection: False

runtime_settings:
# sampling_rate: Specify the desired sampling rate to use. Actual
# sample rates depend on the model being used.
# Overall, possible rates are 250, 500, 1000, and 2000 Hz.
#
# sampling_rate: 250

# track_eyes: Which eye(s) should be tracked?
# Supported Values: LEFT_EYE, RIGHT_EYE, BINOCULAR
#
# track_eyes: RIGHT_EYE

# sample_filtering: Defines the native eye tracker filtering level to be
# applied to the sample event data before it is sent to the specified data
# stream.
# The sample filter section can contain multiple key : value entries if
# the tracker implementation supports it, where each key is a sample stream
# and each value is the associated filter level for that sample data stream.
# sr_research.eyelink.EyeTracker supported stream types are:
# FILTER_ALL, FILTER_FILE, FILTER_ONLINE
# Supported sr_research.eyelink.EyeTracker filter levels are:
# FILTER_LEVEL_OFF, FILTER_LEVEL_1, FILTER_LEVEL_2
# Note that if FILTER_ALL is specified, then other sample data stream
# values are
# ignored. If FILTER_ALL is not provided, ensure to specify the setting for both FILTER_FILE and FILTER_ONLINE as in this case if either is not provided then the missing filter type will have filter level set to FILTER_OFF.

```python
# sample_filtering:
FILTER_ALL: FILTER_LEVEL_OFF
```

```python
vog_settings:
```

```python
# pupil_measure_types: sr_research.eyelink.EyeTracker supports one pupil_measure_type parameter that is used for all eyes being tracked.
# Valid options are:
# PUPIL_AREA, PUPIL_DIAMETER,
# pupil_measure_types: PUPIL_AREA
```

```python
# tracking_mode: Define whether the eye tracker should run in a pupil only mode or run in a pupil-cr mode. Valid options are:
# PUPIL_CR_TRACKING, PUPIL_ONLY_TRACKING
# Depending on other settings on the eyelink Host and the model and mode of the eye tracker being used, this parameter may not be able to set the specified tracking mode. Check the mode listed on the camera setup screen of the Host PC after the experiment has started to confirm if the requested tracking mode was enabled. IMPORTANT: only use PUPIL_ONLY_TRACKING mode if using an EyeLink II system, or using the EyeLink 1000 is a head fixed** setup. Any head movement when using PUPIL_ONLY_TRACKING will result in eye position signal drift.

```python
# tracking_mode: PUPIL_CR_TRACKING
```

```python
# pupil_center_algorithm: The pupil_center_algorithm defines what type of image processing approach should be used to determine the pupil center during image processing.
# Valid possible values are for eyetracker.hw.sr_research.eyelink.EyeTracker are:
# ELLIPSE_FIT, CENTROID_FIT
# pupil_center_algorithm: ELLIPSE_FIT
```

```python
# model_name: The model_name setting allows the definition of the eye tracker being used.
# For the eyelink implementation, valid values are:
# 'EYELINK 1000 DESKTOP', 'EYELINK 1000 TOWER', 'EYELINK 1000 REMOTE',
# 'EYELINK 1000 LONG RANGE', 'EYELINK 2'
# model_name: EYELINK 1000 DESKTOP
```

```python
# manufacturer_name: manufacturer_name is used to store the name of the maker of the eye tracking device. This is for informational purposes only.
# manufacturer_name: SR Research Ltd.
```

```python
# model_name: The below parameters are not used by the EyeGaze eye tracker implementation, so they can be left as is, or filled out for FYI only.
# model_name: N/A
```
# serial_number: The serial number for the specific instance of device used can be specified here. It is not used by the ioHub, so is FYI only.
# serial_number: N/A

# manufacture_date: The date of manufacture of the device can be specified here. It is not used by the ioHub, so is FYI only.
# manufacture_date: DD-MM-YYYY

# hardware_version: The device's hardware version can be specified here. It is not used by the ioHub, so is FYI only.
# hardware_version: N/A

# firmware_version: If the device has firmware, its revision number can be indicated here. It is not used by the ioHub, so is FYI only.
# firmware_version: N/A

# model_number: The device model number can be specified here. It is not used by the ioHub, so is FYI only.
# model_number: N/A

# software_version: The device driver and / or SDK software version number. This field is not used by ioHub, so is FYI only.
# software_version: N/A

# device_number: The device number to assign to the Analog Input device. device_number is not used by this device type.
# device_number: 0

Last Updated: April, 2019

Tobii

Platforms:
- Windows 7 / 10
- Linux (not tested)
- macOS (not tested)

Required Python Version:
- Python 3.6

Supported Models:
Any Tobii model that supports screen based calibration and can used the tobiiresearch API. Tested using a Tobii T120.
Additional Software Requirements

To use the ioHub interface for Tobii, the Tobi Pro SDK must be installed in your Python environment. If a recent standalone installation of PsychoPy3, this package should already be included.

To install tobii-research type:

```
pip install tobii-research
```

EyeTracker Class

class psychopy.iohub.devices.eyetracker.hw.tobii.EyeTracker

To start ioHub with a Tobii eye tracker device, add the Tobii device to the dictionary passed to launchHubServer or the experiment’s iohub_config.yaml:

```
eyetracker.hw.tobii.EyeTracker
```

Examples

A. Start ioHub with a Tobii device and run tracker calibration:

```python
from psychopy.iohub import launchHubServer
from psychopy.core import getTime, wait

iohub_config = {'eyetracker.hw.tobii.EyeTracker':
    {'name': 'tracker', 'runtime_settings': {'sampling_rate': 120}}}

io = launchHubServer(**iohub_config)

# Get the eye tracker device.
tracker = io.devices.tracker

# run eyetracker calibration
r = tracker.runSetupProcedure()
```

B. Print all eye tracker events received for 2 seconds:

```python
# Check for and print any eye tracker events received...
tracker.setRecordingState(True)

stime = getTime()
while getTime() - stime < 2.0:
    for e in tracker.getEvents():
        print(e)
```

C. Print current eye position for 5 seconds:

```python
# Check for and print current eye position every 100 msec.
stime = getTime()
while getTime() - stime < 5.0:
    print(tracker.getPosition())
    wait(0.1)

tracker.setRecordingState(False)
```

(continues on next page)
# Stop the ioHub Server

```python
io.quit()
```

## clearEvents

```python
clearEvents(event_type=None, filter_id=None, call_proc_events=True)
```

Clears any DeviceEvents that have occurred since the last call to the device’s `getEvents()`, or `clearEvents()` methods.

Note that calling `clearEvents()` at the device level only clears the given device's event buffer. The ioHub Process's Global Event Buffer is unchanged.

**Parameters**
None

**Returns**
None

## enableEventReporting

```python
enableEventReporting(enabled=True)
```

`enableEventReporting` is functionally identical to the eye tracker device specific `enableEventReporting` method.

## getConfiguration

```python
getConfiguration()
```

Retrieve the configuration settings information used to create the device instance. This will the default settings for the device, found in `iohub.devices.<device_name>.default_<device_name>.yaml`, updated with any device settings provided via `launchHubServer(...)`. Changing any values in the returned dictionary has no effect on the device state.

**Parameters**
None

**Returns**
The dictionary of the device configuration settings used to create the device.

**Return type**
(`dict`)

## getEvents

```python
getEvents(*args, **kwargs)
```

Retrieve any DeviceEvents that have occurred since the last call to the device's `getEvents()` or `clearEvents()` methods.

Note that calling `getEvents()` at a device level does not change the Global Event Buffer's contents.

**Parameters**

- **event_type_id** (`int`) – If specified, provides the ioHub DeviceEvent ID for which events should be returned for. Events that have occurred but do not match the event ID specified are ignored. Event type ID's can be accessed via the `EventConstants` class; all available event types are class attributes of `EventConstants`.

- **clearEvents** (`int`) – Can be used to indicate if the events being returned should also be removed from the device event buffer. True (the default) indicates to remove events being returned. False results in events being left in the device event buffer.

- **asType** (`str`) – Optional kwarg giving the object type to return events as. Valid values are 'namedtuple' (the default), 'dict', 'list', or 'object'.

**Returns**
New events that the ioHub has received since the last `getEvents()` or `clearEvents()` call to the device. Events are ordered by the ioHub time of each event, older event at index 0. The event object type is determined by the `asType` parameter passed to the method. By default a namedtuple object is returned for each event.

**Return type**
(`list`)
**getLastGazePosition()**

Returns the latest 2D eye gaze position retrieved from the Tobii device. This represents where the eye tracker is reporting each eye gaze vector is intersecting the calibrated surface.

In general, the y or vertical component of each eyes gaze position should be the same value, since in typical user populations the two eyes are yoked vertically when they move. Therefore any difference between the two eyes in the y dimension is likely due to eye tracker error.

Differences between the x, or horizontal component of the gaze position, indicate that the participant is being reported as looking behind or in front of the calibrated plane. When a user is looking at the calibration surface, the x component of the two eyes gaze position should be the same. Differences between the x value for each eye either indicates that the user is not focussing at the calibrated depth, or that there is error in the eye data.

The above remarks are true for any eye tracker in general.

The getLastGazePosition method returns the most recent eye gaze position retrieved from the eye tracker device. This is the position on the calibrated 2D surface that the eye tracker is reporting as the current eye position. The units are in the units in use by the Display device.

If binocular recording is being performed, the average position of both eyes is returned.

If no samples have been received from the eye tracker, or the eye tracker is not currently recording data, None is returned.

**Parameters** None

**Returns**

If the eye tracker is not currently recording data or no eye samples have been received.

tuple: Latest (gaze_x,gaze_y) position of the eye(s)

**Return type** None

**getLastSample()**

Returns the latest sample retrieved from the Tobii device. The Tobii system always using the BinocularSample Event type.

**Parameters** None

**Returns**

If the eye tracker is not currently recording data.

EyeSample: If the eye tracker is recording in a monocular tracking mode, the latest sample event of this event type is returned.

BinocularEyeSample: If the eye tracker is recording in a binocular tracking mode, the latest sample event of this event type is returned.

**Return type** None

**getPosition()**

The getPosition method is the same as the getLastGazePosition method, provided as a consistent cross device method to access the current screen position reported by a device.

See getLastGazePosition for further details.

**isRecordingEnabled()**

isRecordingEnabled returns the recording state from the eye tracking device.

**Parameters** None

**Returns** True == the device is recording data; False == Recording is not occurring

**Return type** bool
runSetupProcedure()

runSetupProcedure performs a calibration routine for the Tobii eye tracking system.

setRecordingState(recording)

setRecordingState is used to start or stop the recording of data from the eye tracking device.

Parameters:
recording (bool) – if True, the eye tracker will start recording available eye data and sending it to the experiment program if data streaming was enabled for the device. If recording == False, then the eye tracker stops recording eye data and streaming it to the experiment.

If the eye tracker is already recording, and setRecordingState(True) is called, the eye tracker will simply continue recording and the method call is a no-op. Likewise if the system has already stopped recording and setRecordingState(False) is called again.

Parameters:
recording (bool) – if True, the eye tracker will start recording data; false = stop recording data.

Returns: the current recording state of the eye tracking device

Return type: bool

Supported Event Types

tobii_research provides real-time access to binocular sample data.

The following fields of the ioHub BinocularEyeSample event are supported:

class psychopy.iohub.devices.eyetracker.BinocularEyeSampleEvent(object)

The BinocularEyeSampleEvent event represents the eye position and eye attribute data collected from one frame or reading of an eye tracker device that is recording both eyes of a participant.

Event Type ID: EventConstants.BINOCULAR_EYE_SAMPLE

Event Type String: 'BINOCULAR_EYE_SAMPLE'

time:

time of event, in sec.msec format, using psychopy timebase.

left_gaze_x:
The horizontal position of the left eye on the computer screen, in Display Coordinate Type Units. Calibration must be done prior to reading (meaningful) gaze data. Uses tobii_research gaze data ‘left_gaze_point_on_display_area’[0] field.

left_gaze_y:
The vertical position of the left eye on the computer screen, in Display Coordinate Type Units. Calibration must be done prior to reading (meaningful) gaze data. Uses tobii_research gaze data ‘left_gaze_point_on_display_area’[1] field.

left_eye_cam_x:
The left x eye position in the eye trackers 3D coordinate space. Uses tobii_research gaze data ‘left_gaze_origin_in_trackbox_coordinate_system’[0] field.

left_eye_cam_y:
The left y eye position in the eye trackers 3D coordinate space. Uses tobii_research gaze data ‘left_gaze_origin_in_trackbox_coordinate_system’[1] field.

left_eye_cam_z:
The left z eye position in the eye trackers 3D coordinate space. Uses tobii_research gaze data ‘left_gaze_origin_in_trackbox_coordinate_system’[2] field.
**left_pupil_measure_1**

Left eye pupil diameter in mm. Uses tobii_research gaze data 'left_pupil_diameter' field.

**right_gaze_x**

The horizontal position of the right eye on the computer screen, in Display Coordinate Type Units. Calibration must be done prior to reading (meaningful) gaze data. Uses tobii_research gaze data 'right_gaze_point_on_display_area'[0] field.

**right_gaze_y**

The vertical position of the right eye on the computer screen, in Display Coordinate Type Units. Calibration must be done prior to reading (meaningful) gaze data. Uses tobii_research gaze data 'right_gaze_point_on_display_area'[1] field.

**right_eye_cam_x**

The right x eye position in the eye trackers 3D coordinate space. Uses tobii_research gaze data 'right_gaze_origin_in_trackbox_coordinate_system'[0] field.

**right_eye_cam_y**

The right y eye position in the eye trackers 3D coordinate space. Uses tobii_research gaze data 'right_gaze_origin_in_trackbox_coordinate_system'[1] field.

**right_eye_cam_z**

The right z eye position in the eye trackers 3D coordinate space. Uses tobii_research gaze data 'right_gaze_origin_in_trackbox_coordinate_system'[2] field.

**right_pupil_measure_1**

Right eye pupil diameter in mm. Uses tobii_research gaze data 'right_pupil_diameter' field.

**status**

Indicates if eye sample contains 'valid' data for left and right eyes. 0 = Eye sample is OK. 2 = Right eye data is likely invalid. 20 = Left eye data is likely invalid. 22 = Eye sample is likely invalid.

**Default Device Settings**

```python
eyetracker.hw.tobii.EyeTracker:

# Indicates if the device should actually be loaded at experiment runtime.
   enable: True

# The variable name of the device that will be used to access the ioHub Device class. 
# during experiment run-time, via the devices.[name] attribute of the ioHub 
# connection or experiment runtime class.
   name: tracker

# Should eye tracker events be saved to the ioHub DataStore file when the device 
# is recording data ?
   save_events: True

# Should eye tracker events be sent to the Experiment process when the device 
# is recording data ?
   stream_events: True

# How many eye events (including samples) should be saved in the ioHub event buffer before 
# old eye events start being replaced by new events. When the event buffer reaches 
# the maximum event length of the buffer defined here, older events will start to 
# be dropped.
```

(continues on next page)
event_buffer_length: 1024

# The Tobii implementation of the common eye tracker interface supports the
# BinocularEyeSampleEvent event type.

monitor_event_types: [ BinocularEyeSampleEvent,]

# The model name of the Tobii device that you wish to connect to can be specified...
→ here,
# and only Tobii systems matching that model name will be considered as possible...
→ candidates for connection.
# If you only have one Tobii system connected to the computer, this field can...
→ just be left empty.

model_name:

# The serial number of the Tobii device that you wish to connect to can be...
→ specified here,
# and only the Tobii system matching that serial number will be connected to, if...
→ found.
# If you only have one Tobii system connected to the computer, this field can...
→ just be left empty,
# in which case the first Tobii device found will be connected to.

serial_number:

calibration:

# Should the PsychoPy Window created by the PsychoPy Process be minimized
# before displaying the Calibration Window created by the ioHub Process.
#

minimize_psychopy_win: False

# The Tobii ioHub Common Eye Tracker Interface currently support
# a 3, 5 and 9 point calibration mode.
# THREE_POINTS,FIVE_POINTS,NINE_POINTS
#

type: NINE_POINTS

# Should the target positions be randomized?
#

randomize: True

# auto_pace can be True or False. If True, the eye tracker will
# automatically progress from one calibration point to the next.
# If False, a manual key or button press is needed to progress to
# the next point.
#

auto_pace: True

# pacing_speed is the number of sec.msec that a calibration point should
# be displayed before moving onto the next point when auto_pace is set to...
→true.

# If auto_pace is False, pacing_speed is ignored.
#

pacing_speed: 1.5

# screen_background_color specifies the r,g,b background color to
# set the calibration, validation, etc, screens to. Each element of the color
# should be a value between 0 and 255. 0 == black, 255 == white.
#
screen_background_color: [128, 128, 128]

# Target type defines what form of calibration graphic should be used
# during calibration, validation, etc. modes.
# Currently the Tobii implementation supports the following
# target type: CIRCLE_TARGET.
# To do: Add support for other types, etc.
#
# target_type: CIRCLE_TARGET

# The associated target attribute properties can be supplied
# for the given target_type.

target_attributes:

# CIRCLE_TARGET is drawn using two PsychoPy
# Circle Stim. The _outer_ circle is drawn first, and should be
# be larger than the _inner_ circle, which is drawn on top of the
# outer circle. The target_attributes starting with 'outer_' define
# how the outer circle of the calibration targets should be drawn.
# The target_attributes starting with 'inner_' define
# how the inner circle of the calibration targets should be drawn.
# # outer_diameter: The size of the outer circle of the calibration target
# #
# outer_diameter: 35
# # outer_stroke_width: The thickness of the outer circle edge.
# #
# outer_stroke_width: 2
# # outer_fill_color: RGB255 color to use to fill the outer circle.
# #
# outer_fill_color: [128, 128, 128]
# # outer_line_color: RGB255 color to used for the outer circle edge.
# #
# outer_line_color: [255, 255, 255]
# # inner_diameter: The size of the inner circle calibration target
# #
# inner_diameter: 7
# # inner_stroke_width: The thickness of the inner circle edge.
# #
# inner_stroke_width: 1
# # inner_fill_color: RGB255 color to use to fill the inner circle.
# #
# inner_fill_color: [0, 0, 0]
# # inner_line_color: RGB255 color to used for the inner circle edge.
# #
# inner_line_color: [0, 0, 0]
# # The Tobii Calibration routine supports using moving target graphics.
# # The following parameters control target movement (if any).
# animate:
# # enable: True if the calibration target should be animated.
# # False specifies that the calibration targets could just jump
# # from one calibration position to another.
# #
# enable: True
# # movement_velocity: The velocity that a calibration target
# # graphic should use when gliding from one calibration
# # point to another. Always in pixels / second.
```
# movement_velocity: 600.0
# expansion_ratio: The outer circle of the calibration target
# can expand (and contract) when displayed at each position.
# expansion_ratio gives the largest size of the outer circle
# as a ratio of the outer_diameter length. For example,
# if outer_diameter = 30, and expansion_ratio = 2.0, then
# the outer circle of each calibration point will expand out
# to 60 pixels. Set expansion_ratio to 1.0 for no expansion.
#
# expansion_ratio: 3.0
# expansion_speed: The rate at which the outer circle
# graphic should expand. Always in pixels / second.
#
# expansion_speed: 30.0
# contract_only: If the calibration target should expand from
# the outer circle initial diameter to the larger diameter
# and then contract back to the original diameter, set
# contract_only to False. To only have the outer circle target
# go from an expanded state to the smaller size, set this to True.
#
# contract_only: True

runtime_settings:
    # The supported sampling rates for Tobii are model dependent.
    # Using a default of 60 Hz, with the assumption it is the most common.
    sampling_rate: 60

    # Tobii implementation supports BINOCULAR tracking mode only.
    track_eyes: BINOCULAR

    # manufacturer_name is used to store the name of the maker of the eye tracking
    # device. This is for informational purposes only.
    manufacturer_name: Tobii Technology
```

Last Updated: June 2019

### 8.5.3 psychopy.iohub Specific Requirements

**Computer Specifications**

The design / requirements of your experiment itself can obviously influence what the minimum computer specification should be to provide good timing / performance.

The dual process design when running using psychopy.iohub also influences the minimum suggested specifications as follows:

- Intel i5 or i7 CPU. A minimum of **two** CPU cores is needed.
- 8 GB of RAM
- Windows 7+, OS X 10.7.5+, or Linux Kernel 2.6+

Please see the *Recommended hardware* section for further information that applies to PsychoPy in general.
Usage Considerations

When using psychopy.iohub, the following constraints should be noted:

1. The pyglet graphics backend must be used; pygame is not supported.

2. ioHub devices that report position data use the unit type defined by the PsychoPy Window. However, position data is reported using the full screen area and size the window was created in. Therefore, for accurate window position reporting, the PsychoPy window must be made full screen.

3. On macOS, Assistive Device support must be enabled when using psychopy.iohub.
   - For OS X 10.7 - 10.8.5, instructions can be found here.
   - For OS X 10.9+, the program being used to start your experiment script must be specifically authorized. Example instructions on authorizing an OS X 10.9+ app can be viewed here.

Software Requirements

When running PsychoPy using the macOS or Windows standalone distribution, all the necessary python package dependencies have already been installed, so the rest of this section can be skipped.

Note: Hardware specific software may need to be installed depending on the device being used. See the documentation page for the specific device hardware in question for further details.

If psychopy.iohub is being manually installed, first ensure the python packages listed in the dependencies section of the manual are installed.

psychopy.iohub requires the following extra dependencies to be installed:

1. psutil (version 1.2+) A cross-platform process and system utilities module for Python.
2. msgpack It’s like JSON, but fast and small.
3. greenlet The greenlet package is a spin-off of Stackless, a version of CPython that supports micro-threads called “tasklets”.
4. gevent (version 1.0 or greater)** A coroutine-based Python networking library.
5. numexpr Fast numerical array expression evaluator for Python and NumPy.
6. pytables PyTables is a package for managing hierarchical datasets.
7. pyYAML PyYAML is a YAML parser and emitter for Python.

Windows installations only

1. pyHook Python wrapper for global input hooks in Windows.

Linux installations only

1. python-xlib The Python X11R6 client-side implementation.

OSX installations only

1. pyobjc: A Python ObjectiveC binding.
8.6 psychopy.tools - miscellaneous tools

Container for all miscellaneous functions and classes

8.6.1 psychopy.tools.colorspacetools

Functions and classes related to color space conversion.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>dkl2rgb</strong>(dkl[, conversionMatrix])</td>
<td>Convert from DKL color space (Derrington, Krauskopf &amp; Lennie) to RGB.</td>
</tr>
<tr>
<td><strong>dklCart2rgb</strong>(LUM, LM, S[, conversionMatrix])</td>
<td>Like dkl2rgb except that it uses cartesian coords (L.M,S,LUM) rather than spherical coords for DKL (elev, azim, contr).</td>
</tr>
<tr>
<td><strong>rgb2dklCart</strong>(picture[, conversionMatrix])</td>
<td>Convert an RGB image into Cartesian DKL space.</td>
</tr>
<tr>
<td><strong>hsv2rgb</strong>(hsv_Nx3)</td>
<td>Convert from HSV color space to RGB gun values.</td>
</tr>
<tr>
<td><strong>lms2rgb</strong>(lms_Nx3[, conversionMatrix])</td>
<td>Convert from cone space (Long, Medium, Short) to RGB.</td>
</tr>
<tr>
<td><strong>rgb2lms</strong>(rgb_Nx3[, conversionMatrix])</td>
<td>Convert from RGB to cone space (LMS).</td>
</tr>
<tr>
<td><strong>dkl2rgb</strong>(dkl[, conversionMatrix])</td>
<td>Convert from DKL color space (Derrington, Krauskopf &amp; Lennie) to RGB.</td>
</tr>
<tr>
<td><strong>cielab2rgb</strong>(lab[, whiteXYZ, …])</td>
<td>Transform CIE L<em>a</em>b* (1976) color space coordinates to RGB tristimulus values.</td>
</tr>
<tr>
<td><strong>cielch2rgb</strong>(lch[, whiteXYZ, …])</td>
<td>Transform CIE L<em>C</em>h* coordinates to RGB tristimulus values.</td>
</tr>
<tr>
<td><strong>srgbTF</strong>(rgb[, reverse])</td>
<td>Apply sRGB transfer function (or gamma) to linear RGB values.</td>
</tr>
<tr>
<td><strong>rec709TF</strong>(rgb[, **kwargs])</td>
<td>Apply the Rec.</td>
</tr>
</tbody>
</table>

**Function details**

**psychopy.tools.colorspacetools**.**dkl2rgb**(dkl, conversionMatrix=None)

Convert from DKL color space (Derrington, Krauskopf & Lennie) to RGB.

Requires a conversion matrix, which will be generated from generic Sony Trinitron phosphors if not supplied (note that this will not be an accurate representation of the color space unless you supply a conversion matrix).

usage:

```python
rgb(Nx3) = dkl2rgb(dkl_Nx3(el, az, radius), conversionMatrix)
```

**psychopy.tools.colorspacetools**.**dklCart2rgb**(LUM, LM, S, conversionMatrix=None)

Like dkl2rgb except that it uses cartesian coords (L.M,S,LUM) rather than spherical coords for DKL (elev, azim, contr).

NB: this may return rgb values >1 or <1

**psychopy.tools.colorspacetools**.**rgb2dklCart**(picture, conversionMatrix=None)

Convert an RGB image into Cartesian DKL space.

**psychopy.tools.colorspacetools**.**hsv2rgb**(hsv_Nx3)

Convert from HSV color space to RGB gun values.

usage:
rgb_Nx3 = hsv2rgb(hsv_Nx3)

Note that in some uses of HSV space the Hue component is given in radians or cycles (range 0:1]). In this version H is given in degrees (0:360).

Also note that the RGB output ranges -1:1, in keeping with other PsychoPy functions.

psychopy.tools.colorspacetools.lms2rgb(lms_Nx3, conversionMatrix=None)
Convert from cone space (Long, Medium, Short) to RGB.

Requires a conversion matrix, which will be generated from generic Sony Trinitron phosphors if not supplied (note that you will not get an accurate representation of the color space unless you supply a conversion matrix)

usage:

rgb_Nx3 = lms2rgb(dkl_Nx3(el,az,radius), conversionMatrix)

psychopy.tools.colorspacetools.rgb2lms(rgb_Nx3, conversionMatrix=None)
Convert from RGB to cone space (LMS).

Requires a conversion matrix, which will be generated from generic Sony Trinitron phosphors if not supplied (note that you will not get an accurate representation of the color space unless you supply a conversion matrix)

usage:

lms_Nx3 = rgb2lms(rgb_Nx3(el,az,radius), conversionMatrix)

psychopy.tools.colorspacetools.dkl2rgb(dkl, conversionMatrix=None)
Convert from DKL color space (Derrington, Krauskopf & Lennie) to RGB.

Requires a conversion matrix, which will be generated from generic Sony Trinitron phosphors if not supplied (note that this will not be an accurate representation of the color space unless you supply a conversion matrix).

usage:

rgb(Nx3) = dkl2rgb(dkl_Nx3(el,az,radius), conversionMatrix)
rgb(NxNx3) = dkl2rgb(dkl_NxNx3(el,az,radius), conversionMatrix)

psychopy.tools.colorspacetools.cielab2rgb(lab, whiteXYZ=None, conversionMatrix=None, transferFunc=None, clip=False, **kwargs)
Transform CIEL*a*b* (1976) color space coordinates to RGB tristimulus values.

CIE L*a*b* are first transformed into CIE XYZ (1931) color space, then the RGB conversion is applied. By default, the sRGB conversion matrix is used with a reference D65 white point. You may specify your own RGB conversion matrix and white point (in CIE XYZ) appropriate for your display.

Parameters

- **lab** (tuple, list or ndarray) – 1-, 2-, 3-D vector of CIE L*a*b* coordinates to convert. The last dimension should be length-3 in all cases specifying a single coordinate.
- **whiteXYZ** (tuple, list or ndarray) – 1-D vector coordinate of the white point in CIE-XYZ color space. Must be the same white point needed by the conversion matrix. The default white point is D65 if None is specified, defined as X, Y, Z = 0.9505, 1.0000, 1.0890.
- **conversionMatrix** (tuple, list or ndarray) – 3x3 conversion matrix to transform CIE-XYZ to RGB values. The default matrix is sRGB with a D65 white point if None is specified. Note that values must be gamma corrected to appear correctly according to the sRGB standard.
• `transferFunc(pyfunc or None)` – Signature of the transfer function to use. If None, values are kept as linear RGB (it’s assumed your display is gamma corrected via the hardware CLUT). The TF must be appropriate for the conversion matrix supplied (default is sRGB). Additional arguments to ‘transferFunc’ can be passed by specifying them as keyword arguments. Gamma functions that come with PsychoPy are ‘srgbTF’ and ‘rec709TF’, see their docs for more information.

• `clip(bool)` – Make all output values representable by the display. However, colors outside of the display’s gamut may not be valid!

Returns  Array of RGB tristimulus values.

Return type  ndarray

Example

Converting a CIEL*a*b* color to linear RGB:

```python
import psychopy.tools.colorspacetools as cst
cielabColor = (53.0, -20.0, 0.0)  # greenish color (L*, a*, b*)
rgbColor = cst.cielab2rgb(cielabColor)
```

Using a transfer function to convert to sRGB:

```python
rgbColor = cst.cielab2rgb(cielabColor, transferFunc=cst.srgbTF)
```

Transform CIE L*C*h* coordinates to RGB tristimulus values.

Parameters

• `lch(tuple, list or ndarray)` – 1-, 2-, 3-D vector of CIE L*C*h* coordinates to convert. The last dimension should be length-3 in all cases specifying a single coordinate. The hue angle *h* is expected in degrees.

• `whiteXYZ(tuple, list or ndarray)` – 1-D vector coordinate of the white point in CIE-XYZ color space. Must be the same white point needed by the conversion matrix. The default white point is D65 if None is specified, defined as X, Y, Z = 0.9505, 1.0000, 1.0890

• `conversionMatrix(tuple, list or ndarray)` – 3x3 conversion matrix to transform CIE-XYZ to RGB values. The default matrix is sRGB with a D65 white point if None is specified. Note that values must be gamma corrected to appear correctly according to the sRGB standard.

• `transferFunc(pyfunc or None)` – Signature of the transfer function to use. If None, values are kept as linear RGB (it’s assumed your display is gamma corrected via the hardware CLUT). The TF must be appropriate for the conversion matrix supplied. Additional arguments to ‘transferFunc’ can be passed by specifying them as keyword arguments. Gamma functions that come with PsychoPy are ‘srgbTF’ and ‘rec709TF’, see their docs for more information.

• `clip(bool)` – Make all output values representable by the display. However, colors outside of the display’s gamut may not be valid!

Returns  array of RGB tristimulus values

Return type  ndarray

psychopy.tools.colorspacetools.srgbTF(rgb, reverse=False, **kwargs)

Apply sRGB transfer function (or gamma) to linear RGB values.
Input values must have been transformed using a conversion matrix derived from sRGB primaries relative to D65.

Parameters

- \( \text{rgb} (\text{tuple, list or ndarray of floats}) \) – \(\text{N} \times 3\) or \(\text{N} \times \text{Nx}3\) array of linear RGB values, last dim must be size \(== 3\) specifying RBG values.
- \( \text{reverse} (\text{boolean}) \) – If True, the reverse transfer function will convert sRGB -> linear RGB.

Returns Array of transformed colors with same shape as input.

Return type ndarray

psychopy.tools.colorspacetools.rec709TF \(\text{rgb, **kwargs}\)

Apply the Rec. 709 transfer function (or gamma) to linear RGB values.

This transfer function is defined in the ITU-R BT.709 (2015) recommendation document (http://www.itu.int/rec/R-REC-BT.709-6-201506-I/en) and is commonly used with HDTV televisions.

Parameters \( \text{rgb} (\text{tuple, list or ndarray of floats}) \) – \(\text{N} \times 3\) or \(\text{N} \times \text{Nx}3\) array of linear RGB values, last dim must be size \(== 3\) specifying RBG values.

Returns Array of transformed colors with same shape as input.

Return type ndarray

8.6.2 psychopy.tools.coordinatetools

Functions and classes related to coordinate system conversion

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{cart2pol}(\text{x, y[, units]}) )</td>
<td>Convert from cartesian to polar coordinates.</td>
</tr>
<tr>
<td>( \text{cart2sph}(\text{z, y, x}) )</td>
<td>Convert from cartesian coordinates ((\text{x,y,z})) to spherical ((\text{elevation, azimuth, radius})).</td>
</tr>
<tr>
<td>( \text{pol2cart}(\text{theta, radius[, units]}) )</td>
<td>Convert from polar to cartesian coordinates.</td>
</tr>
<tr>
<td>( \text{sph2cart(*args)} )</td>
<td>Convert from spherical coordinates ((\text{elevation, azimuth, radius})) to cartesian ((\text{x,y,z})).</td>
</tr>
</tbody>
</table>

Function details

psychopy.tools.coordinatetools.\texttt{cart2pol}(\text{x, y, \textit{units}=deg'})

Convert from cartesian to polar coordinates.

Usage \(\theta, \text{radius} = \text{cart2pol}(\text{x, y, \textit{units}=deg'})\)

units refers to the units (rad or deg) for theta that should be returned

psychopy.tools.coordinatetools.\texttt{cart2sph}(\text{z, y, x})

Convert from cartesian coordinates \((\text{x,y,z})\) to spherical \((\text{elevation, azimuth, radius})\). Output is in degrees.

Usage: array3xN[el,az,rad] = cart2sph(array3xN[x,y,z]) OR elevation, azimuth, radius = cart2sph(x,y,z)

If working in DKL space, \(z = \text{Luminance, y} = S\) and \(x = \text{LM}\)

psychopy.tools.coordinatetools.\texttt{pol2cart}(\text{theta, radius, \textit{units}=deg'})

Convert from polar to cartesian coordinates.

Usage:
To convert from spherical coordinates (elevation, azimuth, radius) to cartesian (x, y, z), you can use the function `sph2cart` from the `psychopy.tools.coordinate_tools` module.

```python
x, y = pol2cart(theta, radius, units='deg')
```

The `sph2cart` function is defined as follows:

```python
psychopy.tools.coordinate_tools.sph2cart(*args)
```

This function converts from spherical coordinates to cartesian coordinates. The function takes an arbitrary number of arguments, allowing for flexibility in input parameters.

**Usage:**
- `array3xN[x,y,z] = sph2cart(array3xN[el,az,rad])` OR `x,y,z = sph2cart(elev, azim, radius)`

### 8.6.3 psychopy.tools.filetools

This section is about functions and classes related to file and directory handling.

**psychopy.tools.filetools.toFile(filename, data)**
- **Save data (of any sort) as a pickle file.**
- **Simple wrapper of the cPickle module in core python.**

**psychopy.tools.filetools.fromFile(filename, encoding='utf-8-sig')**
- **Load data from a pickle or JSON file.**
  - **Parameters**
    - `encoding` (str) – The encoding to use when reading a JSON file. This parameter will be ignored for any other file type.

**psychopy.tools.filetools.mergeFolder(src, dst, pattern=None)**
- **Merge a folder into another.**
  - **Existing files in dst folder with the same name will be overwritten. Non-existent files/folders will be created.**

**psychopy.tools.filetools.openOutputFile(fileName=None, append=False, fileCollisionMethod='rename', encoding='utf-8-sig')**
- **Open an output file (or standard output) for writing.**
  - **Parameters**
    - `fileName` [None, 'stdout', or str] – The desired output file name. If None or stdout, return `sys.stdout`. Any other string will be considered a filename.
    - `append` [bool, optional] – If True, append data to an existing file; otherwise, overwrite it with new data. Defaults to True, i.e. appending.
    - `fileCollisionMethod` [string, optional] – How to handle filename collisions. Valid values are ‘rename’, ‘overwrite’, and ‘fail’. This parameter is ignored if `append` is set to True. Defaults to rename.
    - `encoding` [string, optional] – The encoding to use when writing the file. This parameter will be ignored if `append` is False and `fileName` ends with .psydat or .npy (i.e. if a binary file is to be written). Defaults to ‘utf-8’.

**Returns**
- `f` [file] – A writable file handle.

**psychopy.tools.filetools.genDelimiter(fileName)**
- **Return a delimiter based on a filename.**
  - **Parameters**
    - `fileName` [string] – The output file name.
  - **Returns**
delim [string] A delimiter picked based on the supplied filename. This will be , if the filename extension is .csv, and a tabulator character otherwise.

8.6.4 psychopy.tools.gltools

Overview

createProgram() Create an empty program object for shaders.
createProgramObjectARB() Create an empty program object for shaders.
compileShader(shaderSrc, shaderType) Compile shader GLSL code and return a shader object.
compileShaderObjectARB(shaderSrc, shaderType) Compile shader GLSL code and return a shader object.
embedShaderSourceDefs(shaderSrc, defs) Embed preprocessor definitions into GLSL source code.
deleteObject(obj) Delete a shader or program object.
deleteObjectARB(obj) Delete a program or shader object.
attachShader(program, shader) Attach a shader to a program.
attachObjectARB(program, shader) Attach a shader object to a program.
detachShader(program, shader) Detach a shader object from a program.
detachObjectARB(program, shader) Detach a shader object from a program.
linkProgram(program) Link a shader program.
linkProgramObjectARB(program) Link a shader program object.
validateProgram(program) Check if the program can execute given the current OpenGL state.
validateProgramARB(program) Check if the program can execute given the current OpenGL state.
useProgram(program) Use a program object’s executable shader attachments in the current OpenGL rendering state.
useProgramObjectARB(program) Use a program object’s executable shader attachments in the current OpenGL rendering state.
getInfoLog(obj) Get the information log from a shader or program object.
getUniformLocations(program[, builtins]) Get uniform names and locations from a given shader program object.
getAttribLocations(program[, builtins]) Get attribute names and locations from the specified program object.
createQueryObject([target]) Create a GL query object.
beginQuery(query) Begin query.
endQuery(query) End a query.
getQuery(query) Get the value stored in a query object.
getAbsTimeGPU() Get the absolute GPU time in nanoseconds.
createFBO([attachments]) Create a Framebuffer Object.
attach(attachPoint, imageBuffer) Attach an image to a specified attachment point on the presently bound FBO.
isComplete() Check if the currently bound framebuffer is complete.
deleteFBO(fbo) Delete a framebuffer.
blitFBO(srcRect[, dstRect, filter]) Copy a block of pixels between framebuffers via blitting.
useFBO(fbo) Context manager for Framebuffer Object bindings.

continues on next page
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>createRenderbuffer(width, height[, ...])</code></td>
<td>Create a new Renderbuffer Object with a specified internal format.</td>
</tr>
<tr>
<td><code>deleteRenderbuffer(renderBuffer)</code></td>
<td>Free the resources associated with a renderbuffer.</td>
</tr>
<tr>
<td><code>createTexImage2D(width, height, target[, ...])</code></td>
<td>Create a 2D texture in video memory.</td>
</tr>
<tr>
<td><code>createTexImage2DMultisample(width, height[, ...])</code></td>
<td>Create a 2D multisampled texture.</td>
</tr>
<tr>
<td><code>deleteTexture(texture)</code></td>
<td>Free the resources associated with a texture.</td>
</tr>
<tr>
<td><code>VertexArrayInfo([name, count, ...])</code></td>
<td>Vertex array object (VAO) descriptor.</td>
</tr>
<tr>
<td><code>createVBO(data, target, dataType, usage)</code></td>
<td>Create an array buffer object (VBO).</td>
</tr>
<tr>
<td><code>bindVBO(vbo)</code></td>
<td>Bind a VBO to the current GL state.</td>
</tr>
<tr>
<td><code>unbindVBO(vbo)</code></td>
<td>Unbind a vertex buffer object (VBO).</td>
</tr>
<tr>
<td><code>mapBuffer(vbo[, start, length, read, write, ...])</code></td>
<td>Map a vertex buffer object to client memory.</td>
</tr>
<tr>
<td><code>unmapBuffer(vbo)</code></td>
<td>Unmap a previously mapped buffer.</td>
</tr>
<tr>
<td><code>deleteVBO(vbo)</code></td>
<td>Delete a vertex buffer object (VBO).</td>
</tr>
<tr>
<td><code>setVertexAttribPointer(index, vbo[, size, ...])</code></td>
<td>Define an array of vertex attribute data with a VBO descriptor.</td>
</tr>
<tr>
<td><code>enableVertexAttribArray(index[, legacy])</code></td>
<td>Enable a vertex attribute array.</td>
</tr>
<tr>
<td><code>disableVertexAttribArray(index[, legacy])</code></td>
<td>Disable a vertex attribute array.</td>
</tr>
<tr>
<td><code>createMaterial([params, textures, face])</code></td>
<td>Create a new material.</td>
</tr>
<tr>
<td><code>useMaterial(material[, useTextures])</code></td>
<td>Use a material for proceeding vertex draws.</td>
</tr>
<tr>
<td><code>createLight([params])</code></td>
<td>Create a point light source.</td>
</tr>
<tr>
<td><code>useLights(lights[, setupOnly])</code></td>
<td>Use specified lights in successive rendering operations.</td>
</tr>
<tr>
<td><code>setAmbientLight(color)</code></td>
<td>Set the global ambient lighting for the scene when lighting is enabled.</td>
</tr>
<tr>
<td><code>ObjMeshInfo([vertexPos, texCoords, normals, ...])</code></td>
<td>Descriptor for mesh data loaded from a Wavefront OBJ file.</td>
</tr>
<tr>
<td><code>loadObjFile(objFile)</code></td>
<td>Load a Wavefront OBJ file (*.obj).</td>
</tr>
<tr>
<td><code>loadMtlFile(mtlLib[, texParams])</code></td>
<td>Load a material library file (*.mtl).</td>
</tr>
<tr>
<td><code>createUVSphere(radius, sectors, stacks[, ...])</code></td>
<td>Create a UV sphere.</td>
</tr>
<tr>
<td><code>createPlane(size)</code></td>
<td>Create a plane.</td>
</tr>
<tr>
<td><code>createMeshGridFromArrays(xvals, yvals[, ...])</code></td>
<td>Create a mesh grid using coordinates from arrays.</td>
</tr>
<tr>
<td><code>createMeshGrid([size, subdiv, tessMode])</code></td>
<td>Create a grid mesh.</td>
</tr>
<tr>
<td><code>createBox([size, flipFaces])</code></td>
<td>Create a box mesh.</td>
</tr>
<tr>
<td><code>transformMeshPosOri(vertiates, normals[, ...])</code></td>
<td>Transform a mesh.</td>
</tr>
<tr>
<td><code>calculateVertexNormals(vertiates, faces[, ...])</code></td>
<td>Calculate vertex normals given vertices and triangle faces.</td>
</tr>
<tr>
<td><code>getIntegerv(parName)</code></td>
<td>Get a single integer parameter value, return it as a Python integer.</td>
</tr>
<tr>
<td><code>getFloatv(parName)</code></td>
<td>Get a single float parameter value, return it as a Python float.</td>
</tr>
<tr>
<td><code>getString(parName)</code></td>
<td>Get a single string parameter value, return it as a Python UTF-8 string.</td>
</tr>
<tr>
<td><code>getOpenGLInfo()</code></td>
<td>Get general information about the OpenGL implementation on this machine.</td>
</tr>
</tbody>
</table>
**Details**

`psychopy.tools.gltools.createProgram()`  
Create an empty program object for shaders.

**Returns**  
OpenGL program object handle retrieved from a `glCreateProgram` call.

**Return type**  
int

**Examples**

Building a program with vertex and fragment shader attachments:

```python
myProgram = createProgram()  # new shader object

# compile vertex and fragment shader sources
vertexShader = compileShader(vertShaderSource, GL.GL_VERTEX_SHADER)
fragmentShader = compileShader(fragShaderSource, GL.GL_FRAGMENT_SHADER)

# attach shaders to program
attachShader(myProgram, vertexShader)
attachShader(myProgram, fragmentShader)

# link the shader, makes `myProgram` attachments executable by their respective processors and available for use
linkProgram(myProgram)

# optional, validate the program
validateProgram(myProgram)

# optional, detach and discard shader objects
detachShader(myProgram, vertexShader)
detachShader(myProgram, fragmentShader)

deleteObject(vertexShader)
deleteObject(fragmentShader)
```

You can install the program for use in the current rendering state by calling:

```python
useProgram(myShader)  # OR `glUseProgram(myShader)`
# set uniforms/attributes and start drawing here ...
```

`psychopy.tools.gltools.createProgramObjectARB()`  
Create an empty program object for shaders.

This creates an *Architecture Review Board* (ARB) program variant which is compatible with older GLSL versions and OpenGL coding practices (eg. immediate mode) on some platforms. Use *ARB variants of shader helper functions* (eg. `compileShaderObjectARB` instead of `compileShader`) when working with these ARB program objects. This was included for legacy support of existing PsychoPy shaders. However, it is recommended that you use `createShader()` and follow more recent OpenGL design patterns for new code (if possible of course).

**Returns**  
OpenGL program object handle retrieved from a `glCreateProgramObjectARB` call.

**Return type**  
int
Examples

Building a program with vertex and fragment shader attachments:

```python
myProgram = createProgramObjectARB()  # new shader object

# compile vertex and fragment shader sources
vertexShader = compileShaderObjectARB(vertShaderSource, GL.GL_VERTEX_SHADER_ARB)
fragmentShader = compileShaderObjectARB(fragShaderSource, GL.GL_FRAGMENT_SHADER_ARB)

# attach shaders to program
attachObjectARB(myProgram, vertexShader)
attachObjectARB(myProgram, fragmentShader)

# link the shader, makes 'myProgram' attachments executable by their respective processors and available for use
linkProgramObjectARB(myProgram)

# optional, validate the program
validateProgramARB(myProgram)

# optional, detach and discard shader objects
detachObjectARB(myProgram, vertexShader)
detachObjectARB(myProgram, fragmentShader)

deleteObjectARB(vertexShader)
deleteObjectARB(fragmentShader)

Use the program in the current OpenGL state:

useProgramObjectARB(myProgram)
```

psychoPy.tools.gltools.compileShader(shaderSrc, shaderType)

Compile shader GLSL code and return a shader object. Shader objects can then be attached to programs and made executable on their respective processors.

Parameters

- **shaderSrc** *(str, list of str)* - GLSL shader source code.
- **shaderType** *(GLuint)* - Shader program type (e.g. `GL_VERTEX_SHADER`, `GL_FRAGMENT_SHADER`, `GL_GEOMETRY_SHADER`, etc.)

Returns OpenGL shader object handle retrieved from a `glCreateShader` call.

Return type int
Examples

Compiling GLSL source code and attaching it to a program object:

```python
# GLSL vertex shader source
vertexSource = '''
#version 330 core
layout (location = 0) in vec3 vertexPos;

void main()
{
    gl_Position = vec4(vertexPos, 1.0);
}
'''

# compile it, specifying `GL_VERTEX_SHADER`
vertexShader = compileShader(vertexSource, GL.GL_VERTEX_SHADER)
attachShader(myProgram, vertexShader) # attach it to `myProgram`
```

`psychopy.tools.gltools.compileShaderObjectARB(shaderSrc, shaderType)`

Compile shader GLSL code and return a shader object. Shader objects can then be attached to programs an made executable on their respective processors.

Parameters

- **shaderSrc** (str, list of str) – GLSL shader source code text.
- **shaderType** (GLenum) – Shader program type. Must be *_ARB enums such as GL_VERTEX_SHADER_ARB, GL_FRAGMENT_SHADER_ARB, GL_GEOMETRY_SHADER_ARB, etc.

Returns OpenGL shader object handle retrieved from a `glCreateShaderObjectARB` call.

Return type int

`psychopy.tools.gltools.embedShaderSourceDefs(shaderSrc, defs)`

Embed preprocessor definitions into GLSL source code.

This function generates and inserts #define statements into existing GLSL source code, allowing one to use GLSL preprocessor statements to alter program source at compile time.

Passing {'MAX_LIGHTS': 8, 'NORMAL_MAP': False} to `defs` will create and insert the following #define statements into `shaderSrc`:

```python
#define MAX_LIGHTS 8
#define NORMAL_MAP 0
```

As per the GLSL specification, the #version directive must be specified at the top of the file before any other statement (with the exception of comments). If a #version directive is present, generated #define statements will be inserted starting at the following line. If no #version directive is found in `shaderSrc`, the statements will be prepended to `shaderSrc`.

Using preprocessor directives, multiple shader program routines can reside in the same source text if enclosed by #ifdef and #endif statements as shown here:

```python
#ifdef VERTEX
    // vertex shader code here ...
#endif

#ifdef FRAGMENT
```

(continues on next page)
Both the vertex and fragment shader can be built from the same GLSL code listing by setting either VERTEX or FRAGMENT as True:

```c
vertexShader = gltools.compileShaderObjectARB(
    gltools.embedShaderSourceDefs(glslSource, {'VERTEX': True}),
    GL.GL_VERTEX_SHADER_ARB)
fragmentShader = gltools.compileShaderObjectARB(
    gltools.embedShaderSourceDefs(glslSource, {'FRAGMENT': True}),
    GL.GL_FRAGMENT_SHADER_ARB)
```

In addition, #ifdef blocks can be used to prune render code paths. Here, this GLSL snippet shows a shader having diffuse color sampled from a texture is conditional on DIFFUSE_TEXTURE being True, if not, the material color is used instead:

```c
#ifdef DIFFUSE_TEXTURE
    uniform sampler2D diffuseTexture;
#endif

#ifdef DIFFUSE_TEXTURE
    // sample color from texture
    vec4 diffuseColor = texture2D(diffuseTexture, gl_TexCoord[0].st);
#else
    // code path for no textures, just output material color
    vec4 diffuseColor = gl_FrontMaterial.diffuse;
#endif
```

This avoids needing to provide two separate GLSL program sources to build shaders to handle cases where a diffuse texture is or isn’t used.

**Parameters**

- `shaderSrc (str)` – GLSL shader source code.
- `defs (dict)` – Names and values to generate #define statements. Keys must all be valid GLSL preprocessor variable names of type str. Values can only be int, float, str, bytes, or bool types. Boolean values True and False are converted to integers 1 and 0, respectively.

**Returns**  GLSL source code with #define statements inserted.

**Return type**  str

**Examples**

Defining MAX_LIGHTS as 8 in a fragment shader program at runtime:

```c
fragSrc = embedShaderSourceDefs(fragSrc, {'MAX_LIGHTS': 8})
fragShader = compileShaderObjectARB(fragSrc, GL_FRAGMENT_SHADER_ARB)
```

psychopy.tools.gltools.deleteObject (obj)
Delete a shader or program object.

**Parameters**  obj (int) – Shader or program object handle. Must have originated from a createProgram(), compileShader(), glCreateProgram or glCreateShader call.
psychoPy.tools.gltools.deleteObjectARB(obj)
Delete a program or shader object.

Parameters

- **obj (int)** – Program handle to attach shader to. Must have originated from a createProgramObjectARB(), compileShaderObjectARB, `glCreateProgramObjectARB()` or `glCreateShaderObjectARB` call.

psychoPy.tools.gltools.attachShader(program, shader)
Attach a shader to a program.

Parameters

- **program (int)** – Program handle to attach shader to. Must have originated from a createProgram() or `glCreateProgram` call.
- **shader (int)** – Handle of shader object to attach. Must have originated from a compileShader() or `glCreateShader` call.

psychoPy.tools.gltools.attachObjectARB(program, shader)
Attach a shader object to a program.

Parameters

- **program (int)** – Program handle to attach shader to. Must have originated from a createProgramObjectARB() or `glCreateProgramObjectARB` call.
- **shader (int)** – Handle of shader object to attach. Must have originated from a compileShaderObjectARB() or `glCreateShaderObjectARB` call.

psychoPy.tools.gltools.detachShader(program, shader)
Detach a shader object from a program.

Parameters

- **program (int)** – Program handle to detach shader from. Must have originated from a createProgram() or `glCreateProgram` call.
- **shader (int)** – Handle of shader object to detach. Must have been previously attached to program.

psychoPy.tools.gltools.detachObjectARB(program, shader)
Detach a shader object from a program.

Parameters

- **program (int)** – Program handle to detach shader from. Must have originated from a createProgramObjectARB() or `glCreateProgramObjectARB` call.
- **shader (int)** – Handle of shader object to detach. Must have been previously attached to program.

psychoPy.tools.gltools.linkProgram(program)
Link a shader program. Any attached shader objects will be made executable to run on associated GPU processor units when the program is used.

Parameters

- **program (int)** – Program handle to link. Must have originated from a createProgram() or `glCreateProgram` call.

Raises

- **ValueError** – Specified program handle is invalid.
- **RuntimeError** – Program failed to link. Log will be dumped to `stderr`.

8.6. psychoPy.tools - miscellaneous tools

407
psychopy.tools.gltools.linkProgramObjectARB \((\text{program})\)

Link a shader program object. Any attached shader objects will be made executable to run on associated GPU processor units when the program is used.

**Parameters** \(\text{program} (\text{int})\) – Program handle to link. Must have originated from a \texttt{createProgramObjectARB()} or \texttt{glCreateProgramObjectARB} call.

**Raises**

- **ValueError** – Specified \texttt{program} handle is invalid.
- **RuntimeError** – Program failed to link. Log will be dumped to \texttt{stderr}.

psychopy.tools.gltools.validateProgram \((\text{program})\)

Check if the program can execute given the current OpenGL state.

**Parameters** \(\text{program} (\text{int})\) – Handle of program to validate. Must have originated from a \texttt{createProgram()} or \texttt{glCreateProgram} call.

psychopy.tools.gltools.validateProgramARB \((\text{program})\)

Check if the program can execute given the current OpenGL state. If validation fails, information from the driver is dumped giving the reason.

**Parameters** \(\text{program} (\text{int})\) – Handle of program object to validate. Must have originated from a \texttt{createProgramObjectARB()} or \texttt{glCreateProgramObjectARB} call.

psychopy.tools.gltools.useProgram \((\text{program})\)

Use a program object’s executable shader attachments in the current OpenGL rendering state.

In order to install the program object in the current rendering state, a program must have been successfully linked by calling \texttt{linkProgram()} or \texttt{glLinkProgram}.

**Parameters** \(\text{program} (\text{int})\) – Handle of program to use. Must have originated from a \texttt{createProgram()} or \texttt{glCreateProgram} call and was successfully linked. Passing \texttt{0} or \texttt{None} disables shader programs.

**Examples**

Install a program for use in the current rendering state:

```python
useProgram(myShader)
```

Disable the current shader program by specifying \(0\):

```python
useProgram(0)
```

psychopy.tools.gltools.useProgramObjectARB \((\text{program})\)

Use a program object’s executable shader attachments in the current OpenGL rendering state.

In order to install the program object in the current rendering state, a program must have been successfully linked by calling \texttt{linkProgramObjectARB()} or \texttt{glLinkProgramObjectARB}.

**Parameters** \(\text{program} (\text{int})\) – Handle of program object to use. Must have originated from a \texttt{createProgramObjectARB()} or \texttt{glCreateProgramObjectARB} call and was successfully linked. Passing \texttt{0} or \texttt{None} disables shader programs.
**Examples**

Install a program for use in the current rendering state:

```python
useProgramObjectARB(myShader)
```

Disable the current shader program by specifying 0:

```python
useProgramObjectARB(0)
```

**Notes**

Some drivers may support using `glUseProgram` for objects created by calling `createProgramObjectARB()` or `glCreateProgramObjectARB`. The `getInfoLog` function retrieves a text log from the driver pertaining to the shader or program. For instance, a log can report shader compiler output or validation results. The verbosity and formatting of the logs are platform-dependent, where one driver may provide more information than another.

This function works with both standard and ARB program object variants.

**Parameters**

- `obj (int)` – Program or shader to retrieve a log from. If a shader, the handle must have originated from a `compileShader()`, `glCreateShader`, `createProgramObjectARB()` or `glCreateProgramObjectARB` call. If a program, the handle must have came from a `createProgram()`, `createProgramObjectARB()`, `glCreateProgram` or `glCreateProgramObjectARB` call.

**Returns**

Information log data. Logs can be empty strings if the driver has no information available.

**Return type**

`str`

The `getUniformLocations` function retrieves uniform names and locations from a given shader program object.

**Parameters**

- `program (int)` – Handle of program to retrieve uniforms. Must have originated from a `createProgram()`, `createProgramObjectARB()`, `glCreateProgram` or `glCreateProgramObjectARB` call.

- `builtins` (bool, optional) – Include built-in GLSL uniforms (eg. `gl_ModelViewProjectionMatrix`). Default is `False`.

**Returns**

Uniform names and locations.

**Return type**

`dict`

The `getAttribLocations` function retrieves attribute names and locations from the specified program object.

**Parameters**

- `program (int)` – Handle of program to retrieve attributes. Must have originated from a `createProgram()`, `createProgramObjectARB()`, `glCreateProgram` or `glCreateProgramObjectARB` call.
• **bultins**(bool, optional) – Include built-in GLSL attributes (e.g. *gl_Vertex*). Default is *False*.

**Returns** Attribute names and locations.

**Return type** dict

psychopy.tools.gltools.**createQueryObject**(target=35007)

Create a GL query object.

**Parameters** target (Glenum or int) – Target for the query.

**Returns** Query object.

**Return type** QueryObjectInfo

### Examples

Get GPU time elapsed executing rendering/GL calls associated with some stimuli (this is not the difference in absolute time between consecutive *beginQuery* and *endQuery* calls!):

```python
# create a new query object
qGPU = createQueryObject(GL_TIME_ELAPSED)

beginQuery(query)
myStim.draw()  # OpenGL calls here
endQuery(query)

# get time elapsed in seconds spent on the GPU
timeRendering = getQueryValue(qGPU) * 1e-9
```

You can also use queries to test if vertices are occluded, as their samples would be rejected during depth testing:

```python
drawVAO(shape0, GL_TRIANGLES)  # draw the first object

# check if the object was completely occluded
qOcclusion = createQueryObject(GL_ANY_SAMPLES_PASSED)

# draw the next shape within query context
beginQuery(qOcclusion)
drawVAO(shape1, GL_TRIANGLES)  # draw the second object
endQuery(qOcclusion)

isOccluded = getQueryValue(qOcclusion) == 1
```

This can be leveraged to perform occlusion testing/culling, where you can render a *cheap* version of your mesh/shape, then the more expensive version if samples were passed.

psychopy.tools.gltools.**beginQuery**(query)

Begin query.

**Parameters** query (QueryObjectInfo) – Query object descriptor returned by *createQueryObject()*.

psychopy.tools.gltools.**endQuery**(query)

End a query.

**Parameters** query (QueryObjectInfo) – Query object descriptor returned by *createQueryObject()* (previously passed to *beginQuery()*).
psychoPy.tools.gltools.getQuery(query)
Get the value stored in a query object.

Parameters
query (QueryObjectInfo) – Query object descriptor returned by createQueryObject(), previously passed to endQuery().

psychoPy.tools.gltools.getAbsTimeGPU()
Get the absolute GPU time in nanoseconds.

Returns
Time elapsed in nanoseconds since the OpenGL context was fully realized.

Return type
int

Examples
Get the current GPU time in seconds:

```python
timeInSeconds = getAbsTimeGPU() * 1e-9
```

Get the GPU time elapsed:

```python
t0 = getAbsTimeGPU()
# some drawing commands here ...
t1 = getAbsTimeGPU()
timeElapsed = (t1 - t0) * 1e-9  # take difference, convert to seconds
```

psychoPy.tools.gltools.createFBO(attachments=())
Create a Framebuffer Object.

Parameters
attachments (list or tuple of tuple) – Optional attachments to initialize the Framebuffer with. Attachments are specified as a list of tuples. Each tuple must contain an attachment point (e.g. GL_COLOR_ATTACHMENT0, GL_DEPTH_ATTACHMENT, etc.) and a buffer descriptor type (Renderbuffer or TexImage2D). If using a combined depth/stencil format such as GL_DEPTH24_STENCIL8, GL_DEPTH_ATTACHMENT and GL_STENCIL_ATTACHMENT must be passed the same buffer. Alternatively, one can use GL_DEPTH_STENCIL_ATTACHMENT instead. If using multisample buffers, all attachment images must use the same number of samples!. As an example, one may specify attachments as ‘attachments=(( GL.GL_COLOR_ATTACHMENT0, frameTexture), (GL.GL_DEPTH_STENCIL_ATTACHMENT, depthRenderBuffer))’. 

Returns
Framebuffer descriptor.

Return type
Framebuffer

Notes

• All buffers must have the same number of samples.
• The ‘userData’ field of the returned descriptor is a dictionary that can be used to store arbitrary data associated with the FBO.
• Framebuffers need a single attachment to be complete.
Examples

Create an empty framebuffer with no attachments:

```python
defo = createFBO()  # invalid until attachments are added
```

Create a render target with multiple color texture attachments:

```python
colorTex = createTexImage2D(1024, 1024)  # empty texture
depthRb = createRenderbuffer(800, 600, internalFormat=GL.GL_DEPTH24_STENCIL8)

# attach images
GL.glBindFramebuffer(GL.GL_FRAMEBUFFER, fbo.id)
attach(GL.GL_COLOR_ATTACHMENT0, colorTex)
attach(GL.GL_DEPTH_ATTACHMENT, depthRb)
attach(GL.GL_STENCIL_ATTACHMENT, depthRb)

# or attach(GL.GL_DEPTH_STENCIL_ATTACHMENT, depthRb)
GL.glBindFramebuffer(GL.GL_FRAMEBUFFER, 0)
```

Examples of userData some custom function might access:

```python
defo.userData['flags'] = ['left_eye', 'clear_before_use']
```

Using a depth only texture (for shadow mapping?):

```python
depthTex = createTexImage2D(800, 600,
    internalFormat=GL.GL_DEPTH_COMPONENT24,
    pixelFormat=GL.GL_DEPTH_COMPONENT)
```

```python
defo = createFBO([(GL.GL_DEPTH_ATTACHMENT, depthTex)])  # is valid
```

psychopy.tools.gltools.attach(attachPoint, imageBuffer)

Attach an image to a specified attachment point on the presently bound FBO.

:param attachPoint int: Attachment point for ‘imageBuffer’ (e.g. GL.GL_COLOR_ATTACHMENT0).
:param imageBuffer: Framebuffer-attachable buffer descriptor. :type imageBuffer:TexImage2D or Renderbuffer

Examples

Attach an image to attachment points on the framebuffer:

```python
GL.glBindFramebuffer(GL.GL_FRAMEBUFFER, fbo)
attach(GL.GL_COLOR_ATTACHMENT0, colorTex)
attach(GL.GL_DEPTH_STENCIL_ATTACHMENT, depthRb)
GL.glBindFramebuffer(GL.GL_FRAMEBUFFER, lastBoundFbo)
```

```python
# same as above, but using a context manager
with useFBO(fbo):
```

(continues on next page)
attachment(GL.GL_COLOR_ATTACHMENT0, colorTex)
attachment(GL.GL_DEPTH_STENCIL_ATTACHMENT, depthRb)

psychopy.tools.gltools.isComplete()
Check if the currently bound framebuffer is complete.

Returns True if the presently bound FBO is complete.
Return type bool

psychopy.tools.gltools.deleteFBO(fbo)
Delete a framebuffer.

psychopy.tools.gltools.blitFBO(srcRect, dstRect=None, filter=9729)
Copy a block of pixels between framebuffers via blitting. Read and draw framebuffers must be bound prior to
calling this function. Beware, the scissor box and viewport are changed when this is called to dstRect.

Parameters
- **srcRect** (list of int) – List specifying the top-left and bottom-right coordinates of the
  region to copy from (<X0>, <Y0>, <X1>, <Y1>).
- **dstRect** (list of int or None) – List specifying the top-left and bottom-right coordinates
  of the region to copy to (<X0>, <Y0>, <X1>, <Y1>). If None, srcRect is used for dstRect.
- **filter** (int) – Interpolation method to use if the image is stretched, default is
  GL_LINEAR, but can also be GL_NEAREST.

Returns
Return type None

Examples
Blitting pixels from on FBO to another:

```python
# bind framebuffer to read pixels from
GL.glBindFramebuffer(GL.GL_READ_FRAMEBUFFER, srcFbo)

# bind framebuffer to draw pixels to
GL.glBindFramebuffer(GL.GL_DRAW_FRAMEBUFFER, dstFbo)

gltools.blitFBO((0, 0, 800, 600), (0, 0, 800, 600))

# unbind both read and draw buffers
GL.glBindFramebuffer(GL.GL_FRAMEBUFFER, 0)
```

psychopy.tools.gltools.useFBO(fbo)
Context manager forFramebuffer Object bindings. This function yields the framebuffer name as an integer.

:param fbo int or Framebuffer: OpenGL Framebuffer Object name/ID or descriptor.

Yields int – OpenGL name of the framebuffer bound in the context.
Examples

Using a framebuffer context manager:

```python
# FBO bound somewhere deep in our code
GL.glBindFramebuffer(GL.GL_FRAMEBUFFER, someOtherFBO)

...  

# create a new FBO, but we have no idea what the currently bound FBO is
fbo = createFBO()

# use a context to bind attachments
with bindFBO(fbo):
    attach(GL.GL_COLOR_ATTACHMENT0, colorTex)
    attach(GL.GL_DEPTH_ATTACHMENT, depthRb)
    attach(GL.GL_STENCIL_ATTACHMENT, depthRb)
    isComplete = gltools.isComplete()

# someOtherFBO is still bound!
```

`psychopy.tools.gltools.createRenderbuffer(width, height, internalFormat=32856, samples=1)`

Create a new Renderbuffer Object with a specified internal format. A multisample storage buffer is created if samples > 1.

Renderbuffers contain image data and are optimized for use as render targets. See https://www.khronos.org/opengl/wiki/Renderbuffer_Object for more information.

**Parameters**

- `width` (int) – Buffer width in pixels.
- `height` (int) – Buffer height in pixels.
- `internalFormat` (int) – Format for renderbuffer data (e.g. GL_RGBA8, GL_DEPTH24_STENCIL8).
- `samples` (int) – Number of samples for multi-sampling, should be >1 and power-of-two. Work with one sample, but will raise a warning.

**Returns** A descriptor of the created renderbuffer.

**Return type** Renderbuffer

**Notes**

The ‘userData’ field of the returned descriptor is a dictionary that can be used to store arbitrary data associated with the buffer.

`psychopy.tools.gltools.deleteRenderbuffer(renderBuffer)`

Free the resources associated with a renderbuffer. This invalidates the renderbuffer’s ID.

`psychopy.tools.gltools.createTexImage2D(width, height, target=3553, level=0, internalFormat=32856, pixelFormat=6408, dataType=5126, data=None, unpackAlignment=4, texParams=None)`

Create a 2D texture in video memory. This can only create a single 2D texture with targets GL_TEXTURE_2D or GL_TEXTURE_RECTANGLE.

**Parameters**
PsychoPy - Psychology software for Python, Release 2020.2.10

• **width** (int) – Texture width in pixels.
• **height** (int) – Texture height in pixels.
• **target** (int) – The target texture should only be either GL_TEXTURE_2D or GL_TEXTURE_RECTANGLE.
• **level** (int) – LOD number of the texture, should be 0 if GL_TEXTURE_RECTANGLE is the target.
• **internalFormat** (int) – Internal format for texture data (e.g. GL_RGBA8, GL_R11F_G11F_B10F).
• **PixelFormat** (int) – Pixel data format (e.g. GL_RGBA, GL_DEPTH_STENCIL)
• **dataType** (int) – Data type for pixel data (e.g. GL_FLOAT, GL_UNSIGNED_BYTE).
• **data** (ctypes or None) – Ctypes pointer to image data. If None is specified, the texture will be created but pixel data will be uninitialized.
• **unpackAlignment** (int) – Alignment requirements of each row in memory. Default is 4.
• **texParams** (dict) – Optional texture parameters specified as dict. These values are passed to glTexParameteri. Each tuple must contain a parameter name and value. For example, texParameters={GL.GL_TEXTURE_MIN_FILTER: GL.GL_LINEAR, GL.GL_TEXTURE_MAG_FILTER: GL.GL_LINEAR}.

Returns A TexImage2D descriptor.

Return type TexImage2D

Notes

The ‘userData’ field of the returned descriptor is a dictionary that can be used to store arbitrary data associated with the texture.

Previous textures are unbound after calling ‘createTexImage2D’.

Examples

Creating a texture from an image file:

```python
import pyglet.gl as GL  # using Pyglet for now

# empty texture
textureDesc = createTexImage2D(1024, 1024, internalFormat=GL.GL_RGBA8)

# load texture data from an image file using Pillow and NumPy
from PIL import Image
import numpy as np
im = Image.open(imageFile)  # 8bpp!
im = im.transpose(Image.FLIP_TOP_BOTTOM)  # OpenGL origin is at bottom
im = im.convert("RGBA")
pixelData = np.array(im).ctypes  # convert to ctypes!

width = pixelData.shape[1]
height = pixelData.shape[0]
textureDesc = gltools.createTexImage2D(
```

(continues on next page)
Create a 2D multisampled texture.

**Parameters**

- `width (int)` – Texture width in pixels.
- `height (int)` – Texture height in pixels.
- `target (int)` – The target texture (e.g. `GL_TEXTURE_2D_MULTISAMPLE`).
- `samples (int)` – Number of samples for multi-sampling, should be >1 and power-of-two. Work with one sample, but will raise a warning.
- `internalFormat (int)` – Internal format for texture data (e.g. `GL_RGBA8`, `GL_R11F_G11F_B10F`).
- `texParameters (list of tuple of int)` – Optional texture parameters specified as a list of tuples. These values are passed to ‘glTexParameter’. Each tuple must contain a parameter name and value. For example, `texParameters=[(GL.GL_TEXTURE_MIN_FILTER, GL.GL_LINEAR), (GL.GL_TEXTURE_MAG_FILTER, GL.GL_LINEAR)]`

**Returns** A `TexImage2DMultisample` descriptor.

**Return type** `TexImage2DMultisample`

### psychopy.tools.gltools.deleteTexture (texture)

Free the resources associated with a texture. This invalidates the texture’s ID.

### psychopy.tools.gltools.createVAO (attribBuffers, indexBuffer=None, attribDivisors=None, legacy=False)

Create a Vertex Array object (VAO). VAOs store buffer binding states, reducing CPU overhead when drawing objects with vertex data stored in VBOs.

Define vertex attributes within a VAO state by passing a mapping for generic attribute indices and VBO buffers.

**Parameters**

- `attribBuffers (dict)` – Attributes and associated VBOs to add to the VAO state. Keys are vertex attribute pointer indices, values are VBO descriptors to define. Values can be `tuples` where the first value is the buffer descriptor, the second is the number of attribute components (int, either 2, 3 or 4), the third is the offset (int), and the last is whether to normalize the array (bool).
- `indexBuffer (VertexBufferInfo)` – Optional index buffer.
• **attribDivisors** *(dict)* – Attribute divisors to set. Keys are vertex attribute pointer indices, values are the number of instances that will pass between updates of an attribute. Setting attribute divisors is only permitted if **legacy** is **False**.

• **legacy** *(bool, optional)* – Use legacy attribute pointer functions when setting the VAO state. This is for compatibility with older GL implementations. Key specified to **attribBuffers** must be **GLenum** types such as **GL_VERTEX_ARRAY** to indicate the capability to use.

**Examples**

Create a vertex array object and enable buffer states within it:

```python
vao = createVAO({0: vertexPos, 1: texCoords, 2: vertexNormals})
```

Using an interleaved vertex buffer, all attributes are in the same buffer *(vertexAttr)*. We need to specify offsets for each attribute by passing a buffer in a *tuple* with the second value specifying the offset:

```python
# buffer with interleaved layout `00011222` per-attribute
vao = createVAO(
    {0: (vertexAttr, 3),  # size 3, offset 0
     1: (vertexAttr, 2, 3),  # size 2, offset 3
     2: (vertexAttr, 3, 5, True)})  # size 3, offset 5, normalize
```

You can mix interleaved and single-use buffers:

```python
vao = createVAO(
    {0: (vertexAttr, 3, 0), 1: (vertexAttr, 3, 3), 2: vertexColors})
```

Specifying an optional index array, this is used for indexed drawing of primitives:

```python
vao = createVAO({0: vertexPos}, indexBuffer=indices)
```

The returned **VertexArrayInfo** instance will have attribute **isIndexed==True**.

Drawing vertex arrays using a VAO, will use the **indexBuffer** if available:

```python
# draw the array
drawVAO(vao, mode=GL.GL_TRIANGLES)
```

Use legacy attribute pointer bindings when building a VAO for compatibility with the fixed-function pipeline and older GLSL versions:

```python
attribBuffers = {GL_VERTEX_ARRAY: vertexPos, GL_NORMAL_ARRAY: normals}
vao = createVAO(attribBuffers, legacy=True)
```

If you wish to used instanced drawing, you can specify attribute divisors this way:

```python
vao = createVAO(
    {0: (vertexAttr, 3, 0), 1: (vertexAttr, 3, 3), 2: vertexColors},
    attribDivisors=(2: 1))
```

```python
psychopy.tools.gltools.drawVAO(vao, mode=4, start=0, count=None, instanceCount=None, flush=False)
```

Draw a vertex array object. Uses **glDrawArrays** or **glDrawElements** if **instanceCount** is **None**, or else **glDrawArraysInstanced** or **glDrawElementsInstanced** is used.

**Parameters**

8.6. **psychopy.tools** - miscellaneous tools
- **vao** (*VertexArrayObject*) – Vertex Array Object (VAO) to draw.
- **mode** (*int, optional*) – Drawing mode to use (e.g. GL_TRIANGLES, GL_QUADS, GL_POINTS, etc.)
- **start** (*int, optional*) – Starting index for array elements. Default is 0 which is the beginning of the array.
- **count** (*int, optional*) – Number of indices to draw from *start*. Must not exceed *vao.*count - *start*.
- **instanceCount** (*int or None*) – Number of instances to draw. If >0 and not *None*, instanced drawing will be used.
- **flush** (*bool, optional*) – Flush queued drawing commands before returning.

### Examples

Creating a VAO and drawing it:

```python
# draw the VAO, renders the mesh
drawVAO(vaoDesc, GL.GL_TRIANGLES)
```

**psychopy.tools.gltools.deleteVAO(vao)**

Delete a Vertex Array Object (VAO). This does not delete array buffers bound to the VAO.

**Parameters**

- **vao** (*VertexArrayInfo*) – VAO to delete. All fields in the descriptor except *user-Data* will be reset.

**psychopy.tools.gltools.createVBO(data, target=34962, dataType=5126, usage=35044)**

Create an array buffer object (VBO).

Creates a VBO using input data, usually as a *ndarray* or *list*. Attributes common to one vertex should occupy a single row of the *data* array.

**Parameters**

- **data** (*array_like*) – A 2D array of values to write to the array buffer. The data type of the VBO is inferred by the type of the array. If the input is a Python list or tuple type, the data type of the array will be GL_FLOAT.
- **target** (*int*) – Target used when binding the buffer (e.g. GL_VERTEX_ARRAY or GL_ELEMENT_ARRAY_BUFFER). Default is GL_VERTEX_ARRAY.
- **dataType** (*Glenum, optional*) – Data type of array. Input data will be recast to an appropriate type if necessary. Default is GL_FLOAT.
- **usage** (*GLenum or int, optional*) – Usage type for the array (i.e. GL_STATIC_DRAW).

**Returns**

A descriptor with vertex buffer information.

**Return type**

VertexBufferInfo
Examples

Creating a vertex buffer object with vertex data:

```python
# vertices of a triangle
verts = [[1.0, 1.0, 0.0], # v0
         [0.0, -1.0, 0.0], # v1
         [-1.0, 1.0, 0.0]] # v2

# load vertices to graphics device, return a descriptor
vboDesc = createVBO(verts)
```

Drawing triangles or quads using vertex buffer data:

```python
nIndices, vSize = vboDesc.shape # element size
bindVBO(vboDesc)
setVertexAttribPointer(
    GL_VERTEX_ARRAY, vSize, vboDesc.dataType, legacy=True)
enableVertex_attribArray(GL_VERTEX_ARRAY, legacy=True)

if vSize == 3:
    drawMode = GL_TRIANGLES
elif vSize == 4:
    drawMode = GL_QUADS

glDrawArrays(drawMode, 0, nIndices)
glFlush()

disableVertex_attribArray(GL_VERTEX_ARRAY, legacy=True)
unbindVBO()
```

Custom data can be associated with this vertex buffer by specifying `userData`:

```python
myVBO = createVBO(data)
myVBO.userData['startIdx'] = 14 # first index to draw with

# use it later
nIndices, vSize = vboDesc.shape # element size
startIdx = myVBO.userData['startIdx']
endIdx = nIndices - startIdx
glDrawArrays(GL_TRIANGLES, startIdx, endIdx)
glFlush()
```

`psychopy.tools.gltools.bindVBO(vbo)`

Bind a VBO to the current GL state.

- **Parameters**  
  `vbo` (`VertexBufferInfo`) – VBO descriptor to bind.

- **Returns**  
  `True` if the binding state was changed. Returns `False` if the state was not changed due to the buffer already being bound.

- **Return type**  
  `bool`

`psychopy.tools.gltools.unbindVBO(vbo)`

Unbind a vertex buffer object (VBO).

- **Parameters**  
  `vbo` (`VertexBufferInfo`) – VBO descriptor to unbind.
psychopy.tools.gltools.mapBuffer(vbo, start=0, length=None, read=True, write=True, noSync=False)

Map a vertex buffer object to client memory. This allows you to modify its contents.

If planning to update VBO vertex data, make sure the VBO usage types are GL_DYNAMIC_* or GL_STREAM_* or else serious performance issues may arise.

**Warning:** Modifying buffer data must be done carefully, or else system stability may be affected. Do not use the returned view ndarray outside of successive *mapBuffer()* and *unmapBuffer()* calls. Do not use the mapped buffer for rendering until after *unmapBuffer()* is called.

**Parameters**

- **vbo (VertexBufferInfo)** – Vertex buffer to map to client memory.
- **start (int)** – Initial index of the sub-range of the buffer to modify.
- **length (int or None)** – Number of elements of the sub-array to map from offset. If None, all elements to from offset to the end of the array are mapped.
- **read (bool, optional)** – Allow data to be read from the buffer (sets GL_MAP_READ_BIT). This is ignored if noSync is True.
- **write (bool, optional)** – Allow data to be written to the buffer (sets GL_MAP_WRITE_BIT).
- **noSync (bool, optional)** – If True, GL will not wait until the buffer is free (i.e. not being processed by the GPU) to map it (sets GL_MAP_UNSYNCHRONIZED_BIT). The contents of the previous storage buffer are discarded and the driver returns a new one. This prevents the CPU from stalling until the buffer is available.

**Returns** View of the data. The type of the returned array is one which best matches the data type of the buffer.

**Return type** ndarray

**Examples**

Map a buffer and edit it:

```python
arr = mapBuffer(vbo)
arr[:, :] += 2.0  # add 2 to all values
unmapBuffer(vbo)  # call when done
# Don't ever modify `arr` after calling `unmapBuffer`. Delete it if
# necessary to prevent it form being used.
del arr
```

Modify a sub-range of data by specifying start and length, indices correspond to values, not byte offsets:

```python
arr = mapBuffer(vbo, start=12, end=24)
arr[:, :] *= 10.0
unmapBuffer(vbo)
```

psychopy.tools.gltools.unmapBuffer(vbo)

Unmap a previously mapped buffer. Must be called after *mapBuffer()* is called and before any drawing operations which use the buffer are called. Failing to call this before using the buffer could result in a system error.

**Parameters** **vbo (VertexBufferInfo)** – Vertex buffer descriptor.
**Returns**  *True* if the buffer has been successfully modified. If *False*, the data was corrupted for some reason and needs to be resubmitted.

**Return type**  `bool`

```python
def deleteVBO(vbo):
    # Delete a vertex buffer object (VBO).
```

**Parameters**  `vbo (VertexBufferInfo)` – Descriptor of VBO to delete.

```python
setVertexAttribPointer(index, vbo, size=None, offset=0, normalize=False, legacy=False)
```

Define an array of vertex attribute data with a VBO descriptor.

In modern OpenGL implementations, attributes are ‘generic’, where an attribute pointer index does not correspond to any special vertex property. Usually the usage for an attribute is defined in the shader program. It is recommended that shader programs define attributes using the `layout` parameters:

```plaintext
layout (location = 0) in vec3 position;
layout (location = 1) in vec2 texCoord;
layout (location = 2) in vec3 normal;
```

Setting attribute pointers can be done like this:

```python
setVertexAttribPointer(0, posVbo)
setVertexAttribPointer(1, texVbo)
setVertexAttribPointer(2, normVbo)
```

For compatibility with older OpenGL specifications, some drivers will alias vertex pointers unless they are explicitly defined in the shader. This allows VAOs to be used with the fixed-function pipeline or older GLSL versions.

On nVidia graphics drivers (and maybe others), the following attribute pointers indices are aliased with reserved GLSL names:

- `gl_Vertex` - 0
- `gl_Normal` - 2
- `gl_Color` - 3
- `gl_SecondaryColor` - 4
- `gl_FogCoord` - 5
- `gl_MultiTexCoord0` - 8
- `gl_MultiTexCoord1` - 9
- `gl_MultiTexCoord2` - 10
- `gl_MultiTexCoord3` - 11
- `gl_MultiTexCoord4` - 12
- `gl_MultiTexCoord5` - 13
- `gl_MultiTexCoord6` - 14
- `gl_MultiTexCoord7` - 15

Specifying `legacy` as *True* will allow for old-style pointer definitions. You must specify the capability as a `GLenum` associated with the pointer in this case:
Parameters

- **index** (int) – Index of the attribute to modify. If `legacy=True`, this value should be a `GLenum` type corresponding to the capability to bind the buffer to, such as `GL_VERTEX_ARRAY`, `GL_TEXTURE_COORD_ARRAY`, `GL_NORMAL_ARRAY`, etc.
- **vbo** (VertexBufferInfo) – VBO descriptor.
- **size** (int, optional) – Number of components per vertex attribute, can be either 1, 2, 3, or 4. If `None` is specified, the component size will be inferred from the `shape` of the VBO. You must specify this value if the VBO is interleaved.
- **offset** (int, optional) – Starting index of the attribute in the buffer.
- **normalize** (bool, optional) – Normalize fixed-point format values when accessed.
- **legacy** (bool, optional) – Use legacy vertex attributes (ie. `GL_VERTEX_ARRAY`, `GL_TEXTURE_COORD_ARRAY`, etc.) for backwards compatibility.

Examples

Define a generic attribute from a vertex buffer descriptor:

```python
# set the vertex location attribute
setVertexAttribPointer(0, vboDesc)  # 0 is vertex in our shader
GL.glColor3f(1.0, 0.0, 0.0)  # red triangle

# draw the triangle
nIndices, vSize = vboDesc.shape  # element size
GL.glDrawArrays(GL.GL_TRIANGLES, 0, nIndices)
```

If our VBO has interleaved attributes, we can specify `offset` to account for that:

```python
# define interleaved vertex attributes
# |
# | Position | Texture | Normals |
# vQuad = [[ -1.0, -1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0], # v0
#          [-1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0], # v1
#          [ 1.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0], # v2
#          [ 1.0, -1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0]] # v3

# create a VBO with interleaved attributes
vboInterleaved = createVBO(np.asarray(vQuad, dtype=np.float32))

# ... before rendering, set the attribute pointers
GL.glBindBuffer(vboInterleaved.target, vboInterleaved.name)
gtools.setVertexAttribPointer(0, vboInterleaved, size=3, offset=0)  # vertex pointer
gtools.setVertexAttribPointer(8, vboInterleaved, size=2, offset=3)  # texture pointer
gtools.setVertexAttribPointer(3, vboInterleaved, size=3, offset=5)  # normals pointer

# Note, we specified 'bind=False' since we are managing the binding
```

(continues on next page)
# state. It is recommended that you do this when setting up interleaved buffers to avoid re-binding the same buffer.

```python
# draw red, full screen quad
GL.glColor3f(1.0, 0.0, 0.0)
GL.glDrawArrays(GL.GL_QUADS, 0, vboInterleaved.shape[1])

# call these when done if `enable=True`
gltools.disableVertexAttribArray(0)
gltools.disableVertexAttribArray(8)
gltools.disableVertexAttribArray(1)

# unbind the buffer
GL.glBindBuffer(vboInterleaved.target, 0)
```

`psychopy.tools.gltools.enableVertexAttribArray(index, legacy=False)`

Enable a vertex attribute array. Attributes will be used for use by subsequent draw operations. Be sure to call `disableVertexAttribArray()` on the same attribute to prevent currently enabled attributes from affecting later rendering.

**Parameters**

- **index (int)** – Index of the attribute to enable. If `legacy=True`, this value should be a `GLenum` type corresponding to the capability to bind the buffer to, such as `GL_VERTEX_ARRAY`, `GL_TEXTURE_COORD_ARRAY`, `GL_NORMAL_ARRAY`, etc.

- **legacy (bool, optional)** – Use legacy vertex attributes (i.e. `GL_VERTEX_ARRAY`, `GL_TEXTURE_COORD_ARRAY`, etc.) for backwards compatibility.

`psychopy.tools.gltools.disableVertexAttribArray(index, legacy=False)`

Disable a vertex attribute array.

**Parameters**

- **index (int)** – Index of the attribute to enable. If `legacy=True`, this value should be a `GLenum` type corresponding to the capability to bind the buffer to, such as `GL_VERTEX_ARRAY`, `GL_TEXTURE_COORD_ARRAY`, `GL_NORMAL_ARRAY`, etc.

- **legacy (bool, optional)** – Use legacy vertex attributes (i.e. `GL_VERTEX_ARRAY`, `GL_TEXTURE_COORD_ARRAY`, etc.) for backwards compatibility.

`psychopy.tools.gltools.createMaterial(params=(), textures=(), face=1032)`

Create a new material.

**Parameters**

- **params (list of tuple, optional)** – List of material modes and values. Each mode is assigned a value as (mode, color). Modes can be `GL_AMBIENT`, `GL_DIFFUSE`, `GL_SPECULAR`, `GL_EMISSION`, `GL_SHININESS` or `GL_AMBIENT_AND_DIFFUSE`. Colors must be a tuple of 4 floats which specify reflectance values for each RGBA component. The value of `GL_SHININESS` should be a single float. If no values are specified, an empty material will be created.

**:param textures list of tuple:** List of texture units and TexImage2D descriptors. These will be written to the ‘textures’ field of the returned descriptor. For example, `[GL.GL_TEXTURE0, texDesc0), (GL.GL_TEXTURE1, texDesc1)]`. The number of texture units per-material is `GL_MAX_COMBINED_TEXTURE_IMAGE_UNITS`.

**Parameters**
• **optional** – List of texture units and TexImage2D descriptors. These will be written to the ‘textures’ field of the returned descriptor. For example, [(GL.GL_TEXTURE0, texDesc0), (GL.GL_TEXTURE1, texDesc1)]. The number of texture units per-material is GL_MAX_COMBINED_TEXTURE_IMAGE_UNITS.

• **face**(int, optional) – Faces to apply material to. Values can be GL_FRONT_AND_BACK, GL_FRONT and GL_BACK. The default is GL_FRONT_AND_BACK.

**Returns** A descriptor with material properties.

**Return type** Material

### Examples

Creating a new material with given properties:

```python
# The values for the material below can be found at
# http://devernay.free.fr/cours/opengl/materials.html

# create a gold material
gold = createMaterial([  
   (GL.GL_AMBIENT, (0.24725, 0.19950, 0.07450, 1.0)),  
   (GL.GL_DIFFUSE, (0.75164, 0.60648, 0.22648, 1.0)),  
   (GL.GL_SPECULAR, (0.628281, 0.555802, 0.366065, 1.0)),  
   (GL.GL_SHININESS, 0.4 * 128.0)])
```

Use the material when drawing:

```python
useMaterial(gold)
drawVAO( ... )  # all meshes will be gold
useMaterial(None)  # turn off material when done
```

Create a red plastic material, but define reflectance and shine later:

```python
red_plastic = createMaterial()

# you need to convert values to ctypes!
red_plastic.values[GL_AMBIENT] = (GLfloat * 4)(0.0, 0.0, 0.0, 1.0)
red_plastic.values[GL_DIFFUSE] = (GLfloat * 4)(0.5, 0.0, 0.0, 1.0)
red_plastic.values[GL_SPECULAR] = (GLfloat * 4)(0.7, 0.6, 0.6, 1.0)
red_plastic.values[GL_SHININESS] = 0.25 * 128.0

# set and draw
useMaterial(red_plastic)
drawVertexbuffers( ... )  # all meshes will be red plastic
useMaterial(None)
```

psychopy.tools.gltools.useMaterial(material, useTextures=True)

Use a material for proceeding vertex draws.

**Parameters**

• **material** (Material or None) – Material descriptor to use. Default material properties are set if None is specified. This is equivalent to disabling materials.

• **useTextures** (bool) – Enable textures. Textures specified in a material descriptor’s ‘texture’ attribute will be bound and their respective texture units will be enabled. Note, when disabling materials, the value of useTextures must match the previous call. If there are no textures attached to the material, useTexture will be silently ignored.
Returns

Return type  None

Notes

1. If a material mode has a value of None, a color with all components 0.0 will be assigned.

2. Material colors and shininess values are accessible from shader programs after calling `useMaterial`. Values can be accessed via built-in `gl_FrontMaterial` and `gl_BackMaterial` structures (e.g. `gl_FrontMaterial.diffuse`).

Examples

Use a material when drawing:

```
useMaterial(metalMaterials.gold)
drawVAO(...)
# all meshes drawn will be gold

useMaterial(None)
# turn off material when done
```

```
psychopy.tools.gltools.createLight(params=())
Create a point light source.

psychopy.tools.gltools.useLights(lights, setupOnly=False)
Use specified lights in successive rendering operations. All lights will be transformed using the present modelview matrix.

Parameters

• **lights** (List of Light or None) – Descriptor of a light source. If None, lighting is disabled.

• **setupOnly** (bool, optional) – Do not enable lighting or lights. Specify True if lighting is being computed via fragment shaders.
```

```
psychopy.tools.gltools.setAmbientLight(color)
Set the global ambient lighting for the scene when lighting is enabled. This is equivalent to GL.glLightModelfv(GL.GL_LIGHT_MODEL_AMBIENT, color) and does not contribute to the GL_MAX_LIGHTS limit.

Parameters **color** (tuple) – Ambient lighting RGBA intensity for the whole scene.
```

Notes

If unset, the default value is (0.2, 0.2, 0.2, 1.0) when GL_LIGHTING is enabled.

```
psychopy.tools.gltools.loadObjFile(objFile)
Load a Wavefront OBJ file (*.obj).

Loads vertex, normals, and texture coordinates from the provided *.obj file into arrays. These arrays can be processed then loaded into vertex buffer objects (VBOs) for rendering. The *.obj file must at least specify vertex position data to be loaded successfully. Normals and texture coordinates are optional.

Faces can be either triangles or quads, but not both. Faces are grouped by their materials. Index arrays are generated for each material present in the file.

Data from the returned `ObjMeshInfo` object can be used to create vertex buffer objects and arrays for rendering. See Examples below for details on how to do this.

8.6. psychopy.tools - miscellaneous tools
Parameters **objFile** *(str)* – Path to the *.OBJ file to load.

**Returns** Mesh data.

**Return type** ObjMeshInfo

**See also:**

*loadMtlFile* Load a *.mtl* file.

**Notes**

1. This importer should work fine for most sanely generated files. Export your model with Blender for best results, even if you used some other package to create it.
2. The mesh cannot contain both triangles and quads.

**Examples**

Loading a *.OBJ* mode from file:

```python
objModel = loadObjFile('/path/to/file.obj')
# load the material (*.mtl) file, textures are also loaded
mtllib = loadMtlFile('/path/to/' + objModel.mtlFile)
```

Creating separate vertex buffer objects (VBOs) for each vertex attribute:

```python
vertexPosVBO = createVBO(objModel.vertexPos)
texCoordVBO = createVBO(objModel.texCoords)
normalsVBO = createVBO(objModel.normals)
```

Create vertex array objects (VAOs) to draw the mesh. We create VAOs for each face material:

```python
objVAOs = {}  # dictionary for VAOs
# for each material create a VAO
# keys are material names, values are index buffers
for material, faces in objModel.faces.items():
    # convert index buffer to VAO
    indexBuffer = gltools.createVBO(
        faces.flatten(),  # flatten face index for element array
        target=GL.GL_ELEMENT_ARRAY_BUFFER,
        dataType=GL.GL_UNSIGNED_INT)
    objVAOs[material] = gltools.createVAO(
        {0: vertexPosVBO,  # 0 = gl_Vertex
         8: texCoordVBO,  # 8 = gl_MultiTexCoord0
         2: normalsVBO},  # 2 = gl_Normal
        indexBuffer=indexBuffer)

# if using legacy attribute pointers, do this instead ...
# objVAOs[key] = createVAO({GL_VERTEX_ARRAY: vertexPosVBO},
#                          {GL_TEXTURE_COORD_ARRAY: texCoordVBO},
#                          {GL_NORMAL_ARRAY: normalsVBO},
#                          indexBuffer=indexBuffer,
#                          legacy=True)  # this needs to be 'True'
```
To render the VAOs using `objVAOs` created above, do the following:

```python
for material, vao in objVAOs.items():
    useMaterial(mtllib[material])
    drawVAO(vao)
useMaterial(None)  # disable materials when done
```

Optionally, you can create a single-storage, interleaved VBO by using `numpy.hstack`. On some GL implementations, using single-storage buffers offers better performance:

```python
interleavedData = numpy.hstack((objModel.vertexPos, objModel.texCoords, objModel.normals))
vertexData = createVBO(interleavedData)
```

Creating VAOs with interleaved, single-storage buffers require specifying additional information, such as `size` and `offset`:

```python
objVAOs = {}
for key, val in objModel.faces.items():
    indexBuffer = gltools.createVBO(faces.flatten(),
        target=GL.GL_ELEMENT_ARRAY_BUFFER,
        dataType=GL.GL_UNSIGNED_INT)

    objVAOs[key] = createVAO({
        0: (vertexData, 3, 0),  # size=3, offset=0
        8: (vertexData, 2, 3),  # size=2, offset=3
        2: (vertexData, 3, 5),  # size=3, offset=5
        indexBuffer=val})
```

Drawing VAOs with interleaved buffers is exactly the same as shown before with separate buffers.

```python
psychopy.tools.gltools.loadMtlFile(mtllib, texParams=None)
```

Load a material library file (*.mtl).

#### Parameters

- **mtllib (str)** – Path to the material library file.
- **texParams (list or tuple)** – Optional texture parameters for loaded textures. Texture parameters are specified as a list of tuples. Each item specifies the option and parameter. For instance, `[(GL.GL_TEXTURE_MAG_FILTER, GL.GL_LINEAR), ...]`. By default, linear filtering is used for both the minifying and magnification filter functions. This is adequate for most uses.

#### Returns

Dictionary of materials. Where each key is the material name found in the file, and values are `Material` namedtuple objects.

#### Return type

`dict`

#### See also:

**Examples**

Load material associated with an *.OBJ file:

```python
objModel = loadObjFile('/path/to/file.obj')
# load the material (*.mtl) file, textures are also loaded
mtllib = loadMtl('/path/to/' + objModel.mt1File)
```

Use a material when rendering vertex arrays:

```python
useMaterial(mtllib[material])
drawVAO(vao)
useMaterial(None) # disable materials when done
```

The `psychopy.tools.gltools.createUVSphere` function:

Procedurally generate a UV sphere by specifying its radius, and number of stacks and sectors. The poles of the resulting sphere will be aligned with the Z-axis.

Surface normals and texture coordinates are automatically generated. The returned normals are computed to produce smooth shading.

**Parameters**

- `radius (float, optional)` – Radius of the sphere in scene units (usually meters). Default is 0.5.
- `sectors (int, optional)` – Number of longitudinal and latitudinal sub-divisions. Default is 16 for both.
- `stacks (int, optional)` – Number of longitudinal and latitudinal sub-divisions. Default is 16 for both.
- `flipFaces (bool, optional)` – If True, normals and face windings will be set to point inward towards the center of the sphere. Texture coordinates will remain the same. Default is False.

**Returns** Vertex attribute arrays (position, texture coordinates, and normals) and triangle indices.

**Return type** tuple

**Examples**

Create a UV sphere and VAO to render it:

```python
vertices, textureCoords, normals, faces = gltools.createUVSphere(sectors=32, stacks=32)
vertexVBO = gltools.createVBO(vertices)
texCoordVBO = gltools.createVBO(textureCoords)
normalsVBO = gltools.createVBO(normals)
indexBuffer = gltools.createVBO(faces.flatten(),
                         target=GL.GL_ELEMENT_ARRAY_BUFFER,
                         dataType=GL.GL_UNSIGNED_INT)

vao = gltools.createVAO({0: vertexVBO, 8: texCoordVBO, 2: normalsVBO},
                        indexBuffer=indexBuffer)
```

(continues on next page)
The color of the sphere can be changed by calling `glColor*`:

```python
gColor4f(1.0, 0.0, 0.0, 1.0)  # red
gtools.drawVAO(vao, GL.GL_TRIANGLES)
```

Raw coordinates can be transformed prior to uploading to VBOs. Here we can rotate vertex positions and normals so the equator rests on Z-axis:

```python
r = mt.rotationMatrix(90.0, (1.0, 0, 0.0))  # 90 degrees about +X axis
vertices = mt.applyMatrix(r, vertices)
normals = mt.applyMatrix(r, normals)
```

### `psychopy.tools.gltools.createPlane(size=(1.0, 1.0))`

Create a plane.

Procedurally generate a plane (or quad) mesh by specifying its size. Texture coordinates are computed automatically, with origin at the bottom left of the plane. The generated plane is perpendicular to the +Z axis, origin of the plane is at its center.

**Parameters**
- `size` *(tuple or float)* – Dimensions of the plane. If a single value is specified, the plane will be square. Provide a tuple of floats to specify the width and length of the plane (eg. `size=(0.2, 1.3)`).

**Returns**
- Vertex attribute arrays (position, texture coordinates, and normals) and triangle indices.

**Return type**
- *tuple*

### Examples

Create a plane mesh and draw it:

```python
vertices, textureCoords, normals, faces = gltools.createPlane()

vertexVBO = gltools.createVBO(vertices)
texCoordVBO = gltools.createVBO(textureCoords)
normalsVBO = gltools.createVBO(normals)
indexBuffer = gltools.createVBO(  
    faces.flatten(),  
    target=GL.GL_ELEMENT_ARRAY_BUFFER,  
    dataType=GL.GL_UNSIGNED_INT)

vao = gltools.createVAO({0: vertexVBO, 8: texCoordVBO, 2: normalsVBO},  
    indexBuffer=indexBuffer)

# in the rendering loop

gtools.drawVAO(vao, GL.GL_TRIANGLES)
```

### `psychopy.tools.gltools.createMeshGridFromArrays(xvals, yvals, zvals=None, tessMode='diag', computeNormals=True)`

Create a mesh grid using coordinates from arrays.

Generates a mesh using data in provided in 2D arrays of vertex coordinates. Triangle faces are automatically computed by this function by joining adjacent vertices at neighbouring indices in the array. Texture coordinates
are generated covering the whole mesh, with origin at the bottom left.

**Parameters**

- **xvals (array_like)** – NxM arrays of X and Y coordinates. Both arrays must have the same shape. The resulting mesh will have a single vertex for each X and Y pair. Faces will be generated to connect adjacent coordinates in the array.

- **yvals (array_like)** – NxM arrays of X and Y coordinates. Both arrays must have the same shape. The resulting mesh will have a single vertex for each X and Y pair. Faces will be generated to connect adjacent coordinates in the array.

- **zvals (array_like, optional)** – NxM array of Z coordinates for each X and Y. Must have the same shape as X and Y. If not specified, the Z coordinates will be filled with zeros.

- **tessMode (str, optional)** – Tessellation mode. Specifies how faces are generated. Options are ‘center’, ‘radial’, and ‘diag’. Default is ‘diag’. Modes ‘radial’ and ‘center’ work best with odd numbered array dimensions.

- **computeNormals (bool, optional)** – Compute normals for the generated mesh. If False, all normals are set to face in the +Z direction. Presently, computing normals is a slow operation and may not be needed for some meshes.

**Returns** Vertex attribute arrays (position, texture coordinates, and normals) and triangle indices.

**Return type** tuple

**Examples**

Create a 3D sine grating mesh using 2D arrays:

```python
x = np.linspace(0, 1.0, 32)
y = np.linspace(1.0, 0.0, 32)
xx, yy = np.meshgrid(x, y)
zz = np.tile(np.sin(np.linspace(0.0, 32., 32)) * 0.02, (32, 1))
vertices, textureCoords, normals, faces =
gltools.createMeshGridFromArrays(xx, yy, zz)
```

psychopy.tools.gltools.createMeshGrid(size=(1.0, 1.0), subdiv=0, tessMode='diag')

Create a grid mesh.

Procedurally generate a grid mesh by specifying its size and number of sub-divisions. Texture coordinates are computed automatically. The origin is at the center of the mesh. The generated grid is perpendicular to the +Z axis, origin of the grid is at its center.

**Parameters**

- **size (tuple or float)** – Dimensions of the mesh. If a single value is specified, the plane will be square. Provide a tuple of floats to specify the width and length of the plane (eg. size=(0.2, 1.3)).

- **subdiv (int, optional)** – Number of subdivisions. Zero subdivisions are applied by default, and the resulting mesh will only have vertices at the corners.

- **tessMode (str, optional)** – Tessellation mode. Specifies how faces are subdivided. Options are ‘center’, ‘radial’, and ‘diag’. Default is ‘diag’. Modes ‘radial’ and ‘center’ work best with an odd number of subdivisions.

**Returns** Vertex attribute arrays (position, texture coordinates, and normals) and triangle indices.
Return type  tuple

Examples

Create a grid mesh and draw it:

```python
def createPlane():
    vertices = np.array([[0, 0, 0], [0, 1, 0], [1, 0, 0], [1, 1, 0]])
    textureCoords = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
    normals = np.array([[0, 0, 1], [0, 0, 1], [0, 0, 1], [0, 0, 1]])
    faces = np.array([[0, 1, 2, 3]])

    return vertices, textureCoords, normals, faces
```

```python
vertices, textureCoords, normals, faces = gltools.createPlane()
vertexVBO = gltools.createVBO(vertices)
texCoordVBO = gltools.createVBO(textureCoords)
normalsVBO = gltools.createVBO(normals)
indexBuffer = gltools.createVBO(faces.flatten(),
            target=GL.GL_ELEMENT_ARRAY_BUFFER,
            dataType=GL.GL_UNSIGNED_INT)

vao = gltools.createVAO([0: vertexVBO, 8: texCoordVBO, 2: normalsVBO],
            indexBuffer=indexBuffer)

# in the rendering loop
gltools.drawVAO(vao, GL.GL_TRIANGLES)
```

Randomly displace vertices off the plane of the grid by setting the Z value per vertex:

```python
def createMeshGrid(subdiv=11):
    vertices = np.array([[0, 0, 0], [0, 1, 0], [1, 0, 0], [1, 1, 0]])
    textureCoords = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
    normals = np.array([[0, 0, 1], [0, 0, 1], [0, 0, 1], [0, 0, 1]])
    faces = np.array([[0, 1, 2, 3]])

    return vertices, textureCoords, normals, faces
```

```python
vertices, textureCoords, normals, faces = gltools.createMeshGrid(subdiv=11)
numVerts = vertices.shape[0]
vertices[:, 2] = np.random.uniform(-0.02, 0.02, (numVerts,))

# you must recompute surface normals to get correct shading!
normals = gltools.calculateVertexNormals(vertices, faces)

# create a VAO as shown in the previous example here to draw it ...
```

`psychopy.tools.gltools.createBox(size=(1.0, 1.0, 1.0), flipFaces=False)`

Create a box mesh.

Create a box mesh by specifying its size in three dimensions (x, y, z), or a single value (float) to create a cube. The resulting box will be centered about the origin. Texture coordinates and normals are automatically generated for each face.

Setting `flipFaces=True` will make faces and normals point inwards, this allows boxes to be viewed and lit correctly from the inside.

Parameters

- **size (tuple or float)** – Dimensions of the mesh. If a single value is specified, the box will be a cube. Provide a tuple of floats to specify the width, length, and height of the box (eg. size=(0.2, 1.3, 2.1)).

- **flipFaces (bool, optional)** – If True, normals and face windings will be set to point inward towards the center of the box. Texture coordinates will remain the same. Default is False.

Returns  Vertex attribute arrays (position, texture coordinates, and normals) and triangle indices.

Return type  tuple
Examples

Create a box mesh and draw it:

```python
vertices, textureCoords, normals, faces = gltools.createBox()
vertexVBO = gltools.createVBO(vertices)
texCoordVBO = gltools.createVBO(textureCoords)
normalsVBO = gltools.createVBO(normals)
indexBuffer = gltools.createVBO(faces.flatten(),
    target=GL.GL_ELEMENT_ARRAY_BUFFER,
    dataType=GL.GL_UNSIGNED_INT)

vao = gltools.createVAO({0: vertexVBO, 8: texCoordVBO, 2: normalsVBO},
    indexBuffer=indexBuffer)

# in the rendering loop
gltools.drawVAO(vao, GL.GL_TRIANGLES)
```

Transform a mesh.

Transform mesh vertices and normals to a new position and orientation using a position coordinate and rotation quaternion. Values `vertices` and `normals` must be the same shape. This is intended to be used when editing raw vertex data prior to rendering. Do not use this to change the configuration of an object while rendering.

**Parameters**

- `pos` ([array_like](https://numpy.org/doc/stable/reference/generated/numpy.array.html), optional) – Position vector to transform mesh vertices. If Nx3, `vertices` will be transformed by corresponding rows of `pos`.
- `ori` ([array_like](https://numpy.org/doc/stable/reference/generated/numpy.array.html), optional) – Orientation quaternion in form [x, y, z, w]. If Nx4, `vertices` and `normals` will be transformed by corresponding rows of `ori`.

**Returns** Transformed vertices and normals.

**Return type** tuple

**Examples**

Create and re-orient a plane to face upwards:

```python
vertices, textureCoords, normals, faces = createPlane()
# rotation quaternion
qr = quatFromAxisAngle((1., 0., 0.), -90.0)  # -90 degrees about +X axis

# transform the normals and points
vertices, normals = transformMeshPosOri(vertices, normals, ori=qr)
```

Any `create*` primitive generating function can be used in place of `createPlane`.

```python
psychopy.tools.gltools.calculateVertexNormals(vertices, faces, shading='smooth')
```

Calculate vertex normals given vertices and triangle faces.
Finds all faces sharing a vertex index and sets its normal to either the face normal if `shading='flat'` or the average normals of adjacent faces if `shading='smooth'`. Flat shading only works correctly if each vertex belongs to exactly one face.

The direction of the normals are determined by the winding order of triangles, assumed counter clockwise (OpenGL default). Most model editing software exports using this convention. If not, winding orders can be reversed by calling:

```python
faces = np.fliplr(faces)
```

In some case, creases may appear if vertices are at the same location, but do not share the same index.

**Parameters**
- `vertices(array_like)` – Nx3 vertex positions.
- `faces(array_like)` – N x 3 vertex indices.
- `shading(str, optional)` – Shading mode. Options are ‘smooth’ and ‘flat’. Flat only works with meshes where no vertex index is shared across faces.

**Returns**
Vertex normals array with the same shape as `vertices`. Computed normals are normalized.

**Return type**
ndarray

**Examples**

Recomputing vertex normals for a UV sphere:

```python
# create a sphere and discard normals
vertices, textureCoords, _, faces = gltools.createUVSphere()
normals = gltools.calculateVertexNormals(vertices, faces)
```

**psychopy.tools.gltools.getIntegerv (parName)**
Get a single integer parameter value, return it as a Python integer.

**Parameters**
- `parName(int)` – OpenGL property enum to query (e.g. GL_MAJOR_VERSION).

**Returns**

**Return type**
int

**psychopy.tools.gltools.getFloatv (parName)**
Get a single float parameter value, return it as a Python float.

**Parameters**
- `parName(float)` – OpenGL property enum to query.

**Returns**

**Return type**
float

**psychopy.tools.gltools.getString (parName)**
Get a single string parameter value, return it as a Python UTF-8 string.

**Parameters**
- `parName(int)` – OpenGL property enum to query (e.g. GL_VENDOR).

**Returns**

**Return type**
str

**psychopy.tools.gltools.getOpenGLInfo ()**
Get general information about the OpenGL implementation on this machine. This should provide a consistent means of doing so regardless of the OpenGL interface we are using.
Returns are dictionary with the following fields:

- vendor, renderer, version, majorVersion, minorVersion, doubleBuffer,
- maxTextureSize, stereo, maxSamples, extensions

Supported extensions are returned as a list in the ‘extensions’ field. You can check if a platform supports an extension by checking the membership of the extension name in that list.

**Returns**

- **Return type**: OpenGLInfo

**Examples**

**Working with Framebuffer Objects (FBOs):**

Creating an empty framebuffer with no attachments:

```python
fbo = createFBO()  # invalid until attachments are added
```

Create a render target with multiple color texture attachments:

```python
colorTex = createTexImage2D(1024, 1024)  # empty texture
depthRb = createRenderbuffer(800, 600, internalFormat=GL.GL_DEPTH24_STENCIL8)
GL.glBindFramebuffer(GL.GL_FRAMEBUFFER, fbo.id)
attach(GL.GL_COLOR_ATTACHMENT0, colorTex)
attach(GL.GL_DEPTH_ATTACHMENT, depthRb)
attach(GL.GL_STENCIL_ATTACHMENT, depthRb)
# or attach(GL.GL_DEPTH_STENCIL_ATTACHMENT, depthRb)
GL.glBindFramebuffer(GL.GL_FRAMEBUFFER, 0)
```

Attach FBO images using a context. This automatically returns to the previous FBO binding state when complete. This is useful if you don’t know the current binding state:

```python
with useFBO(fbo):
    attach(GL.GL_COLOR_ATTACHMENT0, colorTex)
    attach(GL.GL_DEPTH_ATTACHMENT, depthRb)
    attach(GL.GL_STENCIL_ATTACHMENT, depthRb)
```

How to set userData some custom function might access:

```python
fbo.userData['flags'] = ['left_eye', 'clear_before_use']
```

Binding an FBO for drawing/reading:

```python
GL.glBindFramebuffer(GL.GL_FRAMEBUFFER, fb.id)
```

Depth-only framebuffers are valid, sometimes need for generating shadows:

```python
depthTex = createTexImage2D(800, 600,
    internalFormat=GL.GL_DEPTH_COMPONENT24,
    pixelFormat=GL.GL_DEPTH_COMPONENT)
```

Deleting a framebuffer when done with it. This invalidates the framebuffer’s ID and makes it available for use:
8.6.5 `psychopy.tools.imagetools`

Functions and classes related to image handling

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>array2image(a)</code></td>
<td>Takes an array and returns an image object (PIL)</td>
</tr>
<tr>
<td><code>image2array(im)</code></td>
<td>Takes an image object (PIL) and returns a numpy array</td>
</tr>
<tr>
<td><code>makeImageAuto(inarray)</code></td>
<td>Combines float_uint8 and <code>image2array</code> operations ie.</td>
</tr>
</tbody>
</table>

**Function details**

- `psychopy.tools.imagetools.array2image(a)`
  Takes an array and returns an image object (PIL)

- `psychopy.tools.imagetools.image2array(im)`
  Takes an image object (PIL) and returns a numpy array

- `psychopy.tools.imagetools.makeImageAuto(inarray)`
  Combines float\_uint8 and `image2array` operations ie. scales a numeric array from -1:1 to 0:255 and converts to PIL image format

8.6.6 `psychopy.tools.mathtools`

Assorted math functions for working with vectors, matrices, and quaternions. These functions are intended to provide basic support for common mathematical operations associated with displaying stimuli (e.g. animation, posing, rendering, etc.)

For tools related to view transformations, see `viewtools`.

**Performance and Optimization**

Most functions listed here are very fast, however they are optimized to work on arrays of values (vectorization). Calling functions repeatedly (for instance within a loop), should be avoided as the CPU overhead associated with each function call (not to mention the loop itself) can be considerable.

For example, one may want to normalize a bunch of randomly generated vectors by calling `normalize()` on each row:

```python
v = np.random.uniform(-1.0, 1.0, (1000, 4))  # 1000 length 4 vectors
vn = np.zeros((1000, 4))  # place to write values

# don't do this!
for i in range(1000):
    vn[i, :] = normalize(v[i, :])
```

The same operation is completed in considerably less time by passing the whole array to the function like so:

```python
normalize(v, out=vn)  # very fast!
vn = normalize(v)  # also fast if 'out' is not provided
```

Specifying an output array to `out` will improve performance by reducing overhead associated with allocating memory to store the result (functions do this automatically if `out` is not provided). However, `out` should only be provided if the output
array is reused multiple times. Furthermore, the function still returns a value if `out` is provided, but the returned value is a reference to `out`, not a copy of it. If `out` is not provided, the function will return the result with a freshly allocated array.

### Data Types

Sub-routines used by the functions here will perform arithmetic using 64-bit floating-point precision unless otherwise specified via the `dtype` argument. This functionality is helpful in certain applications where input and output arrays demand a specific type (e.g. when working with data passed to and from OpenGL functions).

If a `dtype` is specified, input arguments will be coerced to match that type and all floating-point arithmetic will use the precision of the type. If input arrays have the same type as `dtype`, they will automatically pass-through without being recast as a different type. As a performance consideration, all input arguments should have matching types and `dtype` set accordingly.

Most functions have an `out` argument, where one can specify an array to write values to. The value of `dtype` is ignored if `out` is provided, and all input arrays will be converted to match the `dtype` of `out` (if not already). This ensures that the type of the destination array is used for all arithmetic.

### Overview

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>length(v[, squared, out, dtype])</code></td>
<td>Get the length of a vector.</td>
</tr>
<tr>
<td><code>normalize(v[, out, dtype])</code></td>
<td>Normalize a vector or quaternion.</td>
</tr>
<tr>
<td><code>orthogonalize(v, n[, out, dtype])</code></td>
<td>Orthogonalize a vector relative to a normal vector.</td>
</tr>
<tr>
<td><code>reflect(v, n[, out, dtype])</code></td>
<td>Reflection of a vector.</td>
</tr>
<tr>
<td><code>dot(v0, v1[, out, dtype])</code></td>
<td>Dot product of two vectors.</td>
</tr>
<tr>
<td><code>cross(v0, v1[, out, dtype])</code></td>
<td>Cross product of 3D vectors.</td>
</tr>
<tr>
<td><code>project(v0, v1[, out, dtype])</code></td>
<td>Project a vector onto another.</td>
</tr>
<tr>
<td><code>perp(v, n[, norm, out, dtype])</code></td>
<td>Project v to be a perpendicular axis of n.</td>
</tr>
<tr>
<td><code>lerp(v0, v1, t[, out, dtype])</code></td>
<td>Linear interpolation (LERP) between two vectors/coordinates.</td>
</tr>
<tr>
<td><code>distance(v0, v1[, out, dtype])</code></td>
<td>Get the distance between vectors/coordinates.</td>
</tr>
<tr>
<td><code>angleTo(v, point[, degrees, out, dtype])</code></td>
<td>Get the relative angle to a point from a vector.</td>
</tr>
<tr>
<td><code>bisector(v0, v1[, norm, out, dtype])</code></td>
<td>Get the angle bisector.</td>
</tr>
<tr>
<td><code>surfaceNormal(tri[, norm, out, dtype])</code></td>
<td>Compute the surface normal of a given triangle.</td>
</tr>
<tr>
<td><code>surfaceBitangent(tri, uv[, norm, out, dtype])</code></td>
<td>Compute the bitangent vector of a given triangle.</td>
</tr>
<tr>
<td><code>surfaceTangent(tri, uv[, norm, out, dtype])</code></td>
<td>Compute the tangent vector of a given triangle.</td>
</tr>
<tr>
<td><code>vertexNormal(faceNorms[, norm, out, dtype])</code></td>
<td>Compute a vertex normal from shared triangles.</td>
</tr>
<tr>
<td><code>fixTangentHandedness(tangents, normals, …)</code></td>
<td>Ensure the handedness of tangent vectors are all the same.</td>
</tr>
<tr>
<td><code>fitBBox(points[, dtype])</code></td>
<td>Fit an axis-aligned bounding box around points.</td>
</tr>
<tr>
<td><code>computeBBoxCorners(extents[, dtype])</code></td>
<td>Get the corners of an axis-aligned bounding box.</td>
</tr>
<tr>
<td><code>intersectRayPlane(rayOrig, rayDir,…[, dtype])</code></td>
<td>Get the point which a ray intersects a plane.</td>
</tr>
<tr>
<td><code>intersectRaySphere(rayOrig, rayDir[, …])</code></td>
<td>Calculate the points which a ray/line intersects a sphere (if any).</td>
</tr>
<tr>
<td><code>intersectRayAABB(rayOrig, rayDir,…[, dtype])</code></td>
<td>Find the point a ray intersects an axis-aligned bounding box (AABB).</td>
</tr>
<tr>
<td><code>intersectRayOBB(rayOrig, rayDir,…[, dtype])</code></td>
<td>Find the point a ray intersects an oriented bounding box (OBB).</td>
</tr>
<tr>
<td><code>intersectRayTriangle(rayOrig, rayDir, tri,…)</code></td>
<td>Get the intersection of a ray and triangle(s).</td>
</tr>
<tr>
<td><code>ortho3Dto2D(p, orig, normal, up[, right, dtype])</code></td>
<td>Get the planar coordinates of an orthogonal projection of a 3D point onto a 2D plane.</td>
</tr>
<tr>
<td><code>articulate(boneVecs, boneOris[, dtype])</code></td>
<td>Articulate an armature.</td>
</tr>
</tbody>
</table>
Table 30 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>slerp(q0, q1, t[, shortest, out, dtype])</td>
<td>Spherical linear interpolation (SLERP) between two quaternions.</td>
</tr>
<tr>
<td>quatToAxisAngle(q[, degrees, dtype])</td>
<td>Convert a quaternion to axis and angle representation.</td>
</tr>
<tr>
<td>quatFromAxisAngle(axis, angle[, degrees, dtype])</td>
<td>Create a quaternion to represent a rotation about axis vector by angle.</td>
</tr>
<tr>
<td>quatYawPitchRoll(q[, degrees, out, dtype])</td>
<td>Get the yaw, pitch, and roll of a quaternion’s orientation relative to the world -Z axis.</td>
</tr>
<tr>
<td>alignTo(v, t[, out, dtype])</td>
<td>Compute a quaternion which rotates one vector to align with another.</td>
</tr>
<tr>
<td>quatMagnitude(q[, squared, out, dtype])</td>
<td>Get the magnitude of a quaternion.</td>
</tr>
<tr>
<td>multQuat(q0, q1[, out, dtype])</td>
<td>Multiply quaternion q0 and q1.</td>
</tr>
<tr>
<td>accumQuat(qlist[, out, dtype])</td>
<td>Accumulate quaternion rotations.</td>
</tr>
<tr>
<td>invertQuat(q[, out, dtype])</td>
<td>Get the multiplicative inverse of a quaternion.</td>
</tr>
<tr>
<td>applyQuat(q, points[, out, dtype])</td>
<td>Rotate points/coordinates using a quaternion.</td>
</tr>
<tr>
<td>quatToMatrix(q[, out, dtype])</td>
<td>Create a 4x4 rotation matrix from a quaternion.</td>
</tr>
<tr>
<td>matrixToQuat(m[, out, dtype])</td>
<td>Convert a rotation matrix to a quaternion.</td>
</tr>
<tr>
<td>matrixFromEulerAngles(rx, ry, rz[, degrees, …])</td>
<td>Construct a 4x4 rotation matrix from Euler angles.</td>
</tr>
<tr>
<td>scaleMatrix(s[, out, dtype])</td>
<td>Create a scaling matrix.</td>
</tr>
<tr>
<td>rotationMatrix(angle[, axis, out, dtype])</td>
<td>Create a rotation matrix.</td>
</tr>
<tr>
<td>translationMatrix(t[, out, dtype])</td>
<td>Create a translation matrix.</td>
</tr>
<tr>
<td>invertMatrix(m[, out, dtype])</td>
<td>Invert a square matrix.</td>
</tr>
<tr>
<td>isOrthogonal(m)</td>
<td>Check if a square matrix is orthogonal.</td>
</tr>
<tr>
<td>isAffine(m)</td>
<td>Check if a 4x4 square matrix describes an affine transformation.</td>
</tr>
<tr>
<td>multMatrix(matrices[, reverse, out, dtype])</td>
<td>Chain multiplication of two or more matrices.</td>
</tr>
<tr>
<td>concatenate(matrices[, out, dtype])</td>
<td>Concatenate matrix transformations.</td>
</tr>
<tr>
<td>normalMatrix(modelMatrix[, out, dtype])</td>
<td>Get the normal matrix from a model matrix.</td>
</tr>
<tr>
<td>forwardProject(objPos, modelView, proj[…])</td>
<td>Project a point in a scene to a window coordinate.</td>
</tr>
<tr>
<td>reverseProject(winPos, modelView, proj[…])</td>
<td>Unproject window coordinates into object or scene coordinates.</td>
</tr>
<tr>
<td>applyMatrix(m, points[, out, dtype])</td>
<td>Apply a matrix over a 2D array of points.</td>
</tr>
<tr>
<td>posOriToMatrix(pos, ori[, out, dtype])</td>
<td>Convert a rigid body pose to a 4x4 transformation matrix.</td>
</tr>
<tr>
<td>transform(pos, ori, points[, out, dtype])</td>
<td>Transform points using a position and orientation.</td>
</tr>
<tr>
<td>scale(sf, points[, out, dtype])</td>
<td>Scale points by a factor.</td>
</tr>
<tr>
<td>lensCorrection(xys[, coefK, distCenter, …])</td>
<td>Lens correction (or distortion) using the division model with even polynomial terms.</td>
</tr>
<tr>
<td>zeroFix(a[, inplace, threshold])</td>
<td>Fix zeros in an array.</td>
</tr>
</tbody>
</table>

**Details**

`psychopy.tools.mathtools.length(v, squared=False, out=None, dtype=None)`

Get the length of a vector.

**Parameters**

- **v (array_like)** – Vector to normalize, can be Nx2, Nx3, or Nx4. If a 2D array is specified, rows are treated as separate vectors.
- **squared (bool, optional)** – If True the squared length is returned. The default is False.
- **out (ndarray, optional)** – Optional output array. Must be same shape and dtype as the expected output if out was not specified.
psychopy.tools.mathtools.normalize(v, out=None, dtype=None)

Normalize a vector or quaternion.

Parameters

- **v** (array_like) – Vector to normalize, can be Nx2, Nx3, or Nx4. If a 2D array is specified, rows are treated as separate vectors. All vectors should have nonzero length.
- **out** [ndarray, optional] Optional output array. Must be same shape and dtype as the expected output if out was not specified.
- **dtype** [dtype or str, optional] Data type for computations can either be ‘float32’ or ‘float64’. If out is specified, the data type of out is used and this argument is ignored. If out is not provided, ‘float64’ is used by default.

Returns

Normalized vector v.

Return type
ndarray

Notes

- If the vector has length is zero, a vector of all zeros is returned after normalization.

Examples

Normalize a vector:

```python
v = [1., 2., 3., 4.]
vn = normalize(v)
```

The normalize function is vectorized. It’s considerably faster to normalize large arrays of vectors than to call normalize separately for each one:

```python
v = np.random.uniform(-1.0, 1.0, (1000, 4))  # 1000 length 4 vectors
vn = np.zeros((1000, 4))  # place to write values
normalize(v, out=vn)  # very fast!
```

# don't do this!
```python
for i in range(1000):
    vn[i, :] = normalize(v[i, :])
```

psychopy.tools.mathtools.orthogonalize(v, n, out=None, dtype=None)

Orthogonalize a vector relative to a normal vector.

This function ensures that v is perpendicular (or orthogonal) to n.

Parameters

- **v** (array_like) – Vector to orthogonalize, can be Nx2, Nx3, or Nx4. If a 2D array is specified, rows are treated as separate vectors.
- **n** (array_like) – Normal vector, must have same shape as v.
- **out** (ndarray, optional) – Optional output array. Must be same shape and dtype as the expected output if out was not specified.
• dtype (dtype or str, optional) – Data type for computations can either be ‘float32’ or ‘float64’. If `out` is specified, the data type of `out` is used and this argument is ignored. If `out` is not provided, ‘float64’ is used by default.

Returns Orthogonalized vector v relative to normal vector n.

Return type ndarray

Warning: If v and n are the same, the direction of the perpendicular vector is indeterminate. The resulting vector is degenerate (all zeros).

psychopy.tools.mathtools.reflect (v, n, out=None, dtype=None)

Reflection of a vector.

Get the reflection of v relative to normal n.

Parameters

• v (array_like) – Vector to reflect, can be Nx2, Nx3, or Nx4. If a 2D array is specified, rows are treated as separate vectors.
• n (array_like) – Normal vector, must have same shape as v.
• out (ndarray, optional) – Optional output array. Must be same shape and dtype as the expected output if out was not specified.
• dtype (dtype or str, optional) – Data type for computations can either be ‘float32’ or ‘float64’. If `out` is specified, the data type of `out` is used and this argument is ignored. If `out` is not provided, ‘float64’ is used by default.

Returns Reflected vector v off normal n.

Return type ndarray

psychopy.tools.mathtools.dot (v0, v1, out=None, dtype=None)

Dot product of two vectors.

The behaviour of this function depends on the format of the input arguments:

• If v0 and v1 are 1D, the dot product is returned as a scalar and out is ignored.
• If v0 and v1 are 2D, a 1D array of dot products between corresponding row vectors are returned.
• If either v0 and v1 are 1D and 2D, an array of dot products between each row of the 2D vector and the 1D vector are returned.

Parameters

• v0 (array_like) – Vector(s) to compute dot products of (e.g. [x, y, z]). v0 must have equal or fewer dimensions than v1.
• v1 (array_like) – Vector(s) to compute dot products of (e.g. [x, y, z]). v0 must have equal or fewer dimensions than v1.
• out (ndarray, optional) – Optional output array. Must be same shape and dtype as the expected output if out was not specified.
• dtype (dtype or str, optional) – Data type for computations can either be ‘float32’ or ‘float64’. If `out` is specified, the data type of `out` is used and this argument is ignored. If `out` is not provided, ‘float64’ is used by default.

Returns Dot product(s) of v0 and v1.
Return type  ndarray

psychopy.tools.mathtools.cross(v0, v1, out=None, dtype=None)

Cross product of 3D vectors.

The behavior of this function depends on the dimensions of the inputs:
• If v0 and v1 are 1D, the cross product is returned as 1D vector.
• If v0 and v1 are 2D, a 2D array of cross products between corresponding row vectors are returned.
• If either v0 and v1 are 1D and 2D, an array of cross products between each row of the 2D vector and the 1D vector are returned.

Parameters

• v0 (array_like) – Vector(s) in form [x, y, z] or [x, y, z, 1].
• v1 (array_like) – Vector(s) in form [x, y, z] or [x, y, z, 1].
• out (ndarray, optional) – Optional output array. Must be same shape and dtype as the expected output if out was not specified.
• dtype (dtype or str, optional) – Data type for computations can either be ‘float32’ or ‘float64’. If out is specified, the data type of out is used and this argument is ignored. If out is not provided, ‘float64’ is used by default.

Returns  Cross product of v0 and v1.

Return type  ndarray

Notes

• If input vectors are 4D, the last value of cross product vectors is always set to one.
• If input vectors v0 and v1 are Nx3 and out is Nx4, the cross product is computed and the last column of out is filled with ones.

Examples

Find the cross product of two vectors:

```
a = normalize([1, 2, 3])
b = normalize([3, 2, 1])
c = cross(a, b)
```

If input arguments are 2D, the function returns the cross products of corresponding rows:

```
# create two 6x3 arrays with random numbers
shape = (6, 3,)
a = normalize(np.random.uniform(-1.0, 1.0, shape))
b = normalize(np.random.uniform(-1.0, 1.0, shape))
cprod = np.zeros(shape)  # output has the same shape as inputs
cross(a, b, out=cprod)
```

If a 1D and 2D vector are specified, the cross product of each row of the 2D array and the 1D array is returned as a 2D array:
a = normalize([1, 2, 3])
b = normalize(np.random.uniform(-1.0, 1.0, (6, 3)))
cprod = np.zeros(a.shape)
cross(a, b, out=cprod)

psychopy.tools.mathtools.project(v0, v1, out=None, dtype=None)
Project a vector onto another.

Parameters

- **v0** (array_like) – Vector can be Nx2, Nx3, or Nx4. If a 2D array is specified, rows are treated as separate vectors.
- **v1** (array_like) – Vector to project onto v0.
- **out** (ndarray, optional) – Optional output array. Must be same shape and dtype as the expected output if out was not specified.
- **dtype** (dtype or str, optional) – Data type for computations can either be ‘float32’ or ‘float64’. If out is specified, the data type of out is used and this argument is ignored. If out is not provided, ‘float64’ is used by default.

Returns Projection of vector v0 on v1.

Return type ndarray or float

psychopy.tools.mathtools.perp(v, n, norm=True, out=None, dtype=None)
Project v to be a perpendicular axis of n.

Parameters

- **v** (array_like) – Vector to project [x, y, z], may be Nx3.
- **n** (array_like) – Normal vector [x, y, z], may be Nx3.
- **norm** (bool) – Normalize the resulting axis. Default is True.
- **out** (ndarray, optional) – Optional output array. Must be same shape and dtype as the expected output if out was not specified.
- **dtype** (dtype or str, optional) – Data type for computations can either be ‘float32’ or ‘float64’. If out is specified, the data type of out is used and this argument is ignored. If out is not provided, ‘float64’ is used by default.

Returns Perpendicular axis of n from v.

Return type ndarray

Examples

Determine the local up (y-axis) of a surface or plane given normal:

```python
normal = [0., 0.70710678, 0.70710678]
up = [1., 0., 0.]
yaxis = perp(up, normal)
```

Do a cross product to get the x-axis perpendicular to both:

```python
xaxis = cross(yaxis, normal)
```
psychoPy - Psychology software for Python, Release 2020.2.10

psycho.py.tools.mathtools.lerp(v0, v1, t, out=None, dtype=None)

Linear interpolation (LERP) between two vectors/coordinates.

**Parameters**

- **v0 (array_like)** – Initial vector/coordinate. Can be 2D where each row is a point.
- **v1 (array_like)** – Final vector/coordinate. Must be the same shape as v0.
- **t (float)** – Interpolation weight factor \([0, 1]\).
- **out (ndarray, optional)** – Optional output array. Must be same shape and dtype as the expected output if out was not specified.
- **dtype (dtype or str, optional)** – Data type for computations can either be ‘float32’ or ‘float64’. If out is specified, the data type of out is used and this argument is ignored. If out is not provided, ‘float64’ is used by default.

**Returns**

Vector at \(t\) with same shape as \(v0\) and \(v1\).

**Return type**

ndarray

**Examples**

Find the coordinate of the midpoint between two vectors:

```python
u = [0., 0., 0.]
v = [0., 0., 1.]
midpoint = lerp(u, v, 0.5)  # 0.5 to interpolate half-way between points
```

psychoPy.tools.mathtools.distance(v0, v1, out=None, dtype=None)

Get the distance between vectors/coordinates.

The behaviour of this function depends on the format of the input arguments:

- If \(v0\) and \(v1\) are 1D, the distance is returned as a scalar and out is ignored.
- If \(v0\) and \(v1\) are 2D, an array of distances between corresponding row vectors are returned.
- If either \(v0\) and \(v1\) are 1D and 2D, an array of distances between each row of the 2D vector and the 1D vector are returned.

**Parameters**

- **v0 (array_like)** – Vectors to compute the distance between.
- **v1 (array_like)** – Vectors to compute the distance between.
- **out (ndarray, optional)** – Optional output array. Must be same shape and dtype as the expected output if out was not specified.
- **dtype (dtype or str, optional)** – Data type for computations can either be ‘float32’ or ‘float64’. If out is specified, the data type of out is used and this argument is ignored. If out is not provided, ‘float64’ is used by default.

**Returns**

Distance between vectors \(v0\) and \(v1\).

**Return type**

ndarray

psychoPy.tools.mathtools.angleTo(v, point, degrees=True, out=None, dtype=None)

Get the relative angle to a point from a vector.

The behaviour of this function depends on the format of the input arguments:
• If \( v_0 \) and \( v_1 \) are 1D, the angle is returned as a scalar and \( \text{out} \) is ignored.

• If \( v_0 \) and \( v_1 \) are 2D, an array of angles between corresponding row vectors are returned.

• If either \( v_0 \) and \( v_1 \) are 1D and 2D, an array of angles between each row of the 2D vector and the 1D vector are returned.

**Parameters**

- \( v \) (array_like) – Direction vector [x, y, z].
- \( \text{point} \) (array_like) – Point(s) to compute angle to from vector \( v \).
- \( \text{degrees} \) (bool, optional) – Return the resulting angles in degrees. If False, angles will be returned in radians. Default is True.
- \( \text{out} \) (ndarray, optional) – Optional output array. Must be same shape and dtype as the expected output if \( \text{out} \) was not specified.
- \( \text{dtype} \) (dtype or str, optional) – Data type for computations can either be ‘float32’ or ‘float64’. If \( \text{out} \) is specified, the data type of \( \text{out} \) is used and this argument is ignored. If \( \text{out} \) is not provided, ‘float64’ is used by default.

**Returns** Distance between vectors \( v_0 \) and \( v_1 \).

**Return type** *ndarray*

psychopy.tools.mathtools.bisector \((v_0, v_1, \text{norm=False, out=None, dtype=None})\)

Get the angle bisector.

Computes a vector which bisects the angle between \( v_0 \) and \( v_1 \). Input vectors \( v_0 \) and \( v_1 \) must be non-zero.

**Parameters**

- \( v_0 \) (array_like) – Vectors to bisect \([x, y, z]\). Must be non-zero in length and have the same shape. Inputs can be Nx3 where the bisector for corresponding rows will be returned.
- \( v_1 \) (array_like) – Vectors to bisect \([x, y, z]\). Must be non-zero in length and have the same shape. Inputs can be Nx3 where the bisector for corresponding rows will be returned.
- \( \text{norm} \) (bool, optional) – Normalize the resulting bisector. Default is False.
- \( \text{out} \) (ndarray, optional) – Optional output array. Must be same shape and dtype as the expected output if \( \text{out} \) was not specified.
- \( \text{dtype} \) (dtype or str, optional) – Data type for computations can either be ‘float32’ or ‘float64’. If \( \text{out} \) is specified, the data type of \( \text{out} \) is used and this argument is ignored. If \( \text{out} \) is not provided, ‘float64’ is used by default.

**Returns** Bisecting vector \([x, y, z]\).

**Return type** *ndarray*

psychopy.tools.mathtools.surfaceNormal \((\text{tri}, \text{norm=True, out=None, dtype=None})\)

Compute the surface normal of a given triangle.

**Parameters**

- \( \text{tri} \) (array_like) – Triangle vertices as 2D (3x3) array \([p0, p1, p2]\) where each vertex is a length 3 array \([vx, xy, vz]\). The input array can be 3D \((Nx3x3)\) to specify multiple triangles.
- \( \text{norm} \) (bool, optional) – Normalize computed surface normals if True, default is True.
• **out**(ndarray, optional) – Optional output array. Must have one fewer dimensions than *tri*. The shape of the last dimension must be 3.

• **dtype**(dtype or str, optional) – Data type for computations can either be ‘float32’ or ‘float64’. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, ‘float64’ is used by default.

**Returns** Surface normal of triangle *tri*.

**Return type** ndarray

**Examples**

Compute the surface normal of a triangle:

```python
vertices = [[-1., 0., 0.], [0., 1., 0.], [1., 0., 0]]
norm = surfaceNormal(vertices)
```

Find the normals for multiple triangles, and put results in a pre-allocated array:

```python
vertices = [[[[-1., 0., 0.], [0., 1., 0.], [1., 0., 0]], # 2x3x3
            [[1., 0., 0.], [0., 1., 0.], [-1., 0., 0]]]]
normals = np.zeros((2, 3)) # normals from two triangles
triangles = surfaceNormal(vertices, out=normals)
```

psychopy.tools.mathtools.surfaceBitangent(*tri*, *uv*, norm=True, out=None, dtype=None)

**Compute the bitangent vector of a given triangle.**

This function can be used to generate bitangent vertex attributes for normal mapping. After computing bitangents, one may orthogonalize them with vertex normals using the orthogonalize() function, or within the fragment shader. Uses texture coordinates at each triangle vertex to determine the direction of the vector.

**Parameters**

• **tri**(array_like) – Triangle vertices as 2D (3x3) array [p0, p1, p2] where each vertex is a length 3 array [vx, xy, vz]. The input array can be 3D (Nx3x3) to specify multiple triangles.

• **uv**(array_like) – Texture coordinates associated with each face vertex as a 2D array (3x2) where each texture coordinate is length 2 array [u, v]. The input array can be 3D (Nx3x2) to specify multiple texture coordinates if multiple triangles are specified.

• **norm**(bool, optional) – Normalize computed bitangents if True, default is True.

• **out**(ndarray, optional) – Optional output array. Must have one fewer dimensions than *tri*. The shape of the last dimension must be 3.

• **dtype**(dtype or str, optional) – Data type for computations can either be ‘float32’ or ‘float64’. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, ‘float64’ is used by default.

**Returns** Surface bitangent of triangle *tri*.

**Return type** ndarray
Examples

Computing the bitangents for two triangles from vertex and texture coordinates (UVs):

```python
# array of triangle vertices (2x3x3)
tri = np.asarray(
    [([-1.0, 1.0, 0.0], [-1.0, -1.0, 0.0], [1.0, -1.0, 0.0]),  # 1
     [([-1.0, 1.0, 0.0], [-1.0, -1.0, 0.0], [1.0, -1.0, 0.0])])  # 2

# array of triangle texture coordinates (2x3x2)
uv = np.asarray(
    [[(0.0, 1.0), (0.0, 0.0), (1.0, 0.0)],  # 1
     [(0.0, 1.0), (0.0, 0.0), (1.0, 0.0)])  # 2

bitangents = surfaceBitangent(tri, uv, norm=True)  # bitangets (2x3)
```

psychopy.tools.mathtools.surfaceTangent (tri, uv, norm=True, out=None, dtype=None)

Compute the tangent vector of a given triangle.

This function can be used to generate tangent vertex attributes for normal mapping. After computing tangents, one may orthogonalize them with vertex normals using the orthogonalize() function, or within the fragment shader. Uses texture coordinates at each triangle vertex to determine the direction of the vector.

Parameters

- **tri** (array_like) – Triangle vertices as 2D (3x3) array [p0, p1, p2] where each vertex is a length 3 array [vx, xy, vz]. The input array can be 3D (Nx3x3) to specify multiple triangles.

- **uv** (array_like) – Texture coordinates associated with each face vertex as a 2D array (3x2) where each texture coordinate is length 2 array [u, v]. The input array can be 3D (Nx3x2) to specify multiple texture coordinates if multiple triangles are specified. If so N must be the same size as the first dimension of tri.

- **norm** (bool, optional) – Normalize computed tangents if True, default is True.

- **out** (ndarray, optional) – Optional output array. Must have one fewer dimensions than tri. The shape of the last dimension must be 3.

- **dtype** (dtype or str, optional) – Data type for computations can either be ‘float32’ or ‘float64’. If out is specified, the data type of out is used and this argument is ignored. If out is not provided, ‘float64’ is used by default.

Returns

Surface normal of triangle tri.

Return type

ndarray

Examples

Compute surface normals, tangents, and bitangents for a list of triangles:

```python
# triangle vertices (2x3x3)
vertices = [[[-1.0, 0.0, 0.0], [0.0, 1.0, 0.0], [1.0, 0.0, 0.0]],
            [[1.0, 0.0, 0.0], [0.0, 1.0, 0.0], [-1.0, 0.0, 0.0]]]

# array of triangle texture coordinates (2x3x2)
uv = np.asarray(
    [[(0.0, 1.0), (0.0, 0.0), (1.0, 0.0)],  # 1
     [(0.0, 1.0), (0.0, 0.0), (1.0, 0.0)])  # 2
```

(continues on next page)
normals = surfaceNormal(vertices)
tangents = surfaceTangent(vertices, uv)
bitangents = cross(normals, tangents)  # or use 'surfaceBitangent'

Orthogonalize a surface tangent with a vertex normal vector to get the vertex tangent and bitangent vectors:

vertexTangent = orthogonalize(faceTangent, vertexNormal)
vertexBitangent = cross(vertexTangent, vertexNormal)

Ensure computed vectors have the same handedness, if not, flip the tangent vector (important for applications like normal mapping):

# tangent, bitangent, and normal are 2D
tangent[dot(cross(normal, tangent), bitangent) < 0.0, :] *= -1.0

psychopy.tools.mathtools.vertexNormal(faceNorms, norm=True, out=None, dtype=None)
Compute a vertex normal from shared triangles.

This function computes a vertex normal by averaging the surface normals of the triangles it belongs to. If model has no vertex normals, first use surfaceNormal() to compute them, then run vertexNormal() to compute vertex normal attributes.

While this function is mainly used to compute vertex normals, it can also be supplied triangle tangents and bitangents.

Parameters

- **faceNorms** (array_like) – An array (Nx3) of surface normals.
- **norm** (bool, optional) – Normalize computed normals if True, default is True.
- **out** (ndarray, optional) – Optional output array.
- **dtype** (dtype or str, optional) – Data type for computations can either be ‘float32’ or ‘float64’. If out is specified, the data type of out is used and this argument is ignored. If out is not provided, ‘float64’ is used by default.

Returns  Vertex normal.

Return type  ndarray

Examples

Compute a vertex normal from the face normals of the triangles it belongs to:

normals = [[1., 0., 0.], [0., 1., 0.]]  # adjacent face normals
vertexNorm = vertexNormal(normals)

psychopy.tools.mathtools.fixTangentHandedness(tangents, normals, bitangents, out=None, dtype=None)
Ensure the handedness of tangent vectors are all the same.

Often 3D computed tangents may not have the same handedness due to how texture coordinates are specified. This function takes input surface vectors are ensures that tangents have the same handedness. Use this function if you notice that normal mapping shading appears reversed with respect to the incident light direction. The output array of corrected tangents can be used inplace of the original.

Parameters
• **tangents** (`array_like`) – Input Nx3 arrays of triangle tangents, normals and bitangents. All arrays must have the same size.

• **normals** (`array_like`) – Input Nx3 arrays of triangle tangents, normals and bitangents. All arrays must have the same size.

• **bitangents** (`array_like`) – Input Nx3 arrays of triangle tangents, normals and bitangents. All arrays must have the same size.

• **out** (`ndarray`, *optional*) – Optional output array for tangents. If not specified, a new array of tangents will be allocated.

• **dtype** (*dtype or str, optional*) – Data type for computations can either be ‘float32’ or ‘float64’. If `out` is specified, the data type of `out` is used and this argument is ignored. If `out` is not provided, ‘float64’ is used by default.

    **Returns** Array of tangents with handedness corrected.

    **Return type** `ndarray`

psychopy.tools.mathtools.fitBBox (*points*, *dtype=None*)

Fit an axis-aligned bounding box around points.

This computes the minimum and maximum extents for a bounding box to completely enclose `points`. Keep in mind the the output in bounds areaxis-aligned and may not optimally fits the points (i.e. fits the points with the minimum required volume). However, this should work well enough for applications such as visibility testing (see ~psychopy.tools.viewtools.volumeVisible for more information).

    **Parameters**

• **points** (`array_like`) – Nx3 or Nx4 array of points to fit the bounding box to.

• **dtype** (*dtype or str, optional*) – Data type for computations can either be ‘float32’ or ‘float64’. If `out` is specified, the data type of `out` is used and this argument is ignored. If `out` is not provided, ‘float64’ is used by default.

    **Returns** Extents (mins, maxs) as a 2x3 array.

    **Return type** `ndarray`

    **See also:**

    ~psychopy.tools.mathtools.computeBBoxCorners

psychopy.tools.mathtools.computeBBoxCorners (*extents*, *dtype=None*)

Get the corners of an axis-aligned bounding box.

    **Parameters**

• **extents** (`array_like`) – 2x3 array indicating the minimum and maximum extents of the bounding box.

• **dtype** (*dtype or str, optional*) – Data type for computations can either be ‘float32’ or ‘float64’. If `out` is specified, the data type of `out` is used and this argument is ignored. If `out` is not provided, ‘float64’ is used by default.

    **Returns** 8x4 array of points defining the corners of the bounding box.

    **Return type** `ndarray`
Examples

Compute the corner points of a bounding box:

```python
minExtent = [-1, -1, -1]
maxExtent = [1, 1, 1]
corners = computeBBoxCorners([minExtent, maxExtent])

# [[ 1. 1. 1. 1.]
# [-1. 1. 1. 1.]
# [ 1. -1. 1. 1.]
# [-1. -1. 1. 1.]
# [ 1. 1. -1. 1.]
# [-1. 1. -1. 1.]
# [ 1. -1. -1. 1.]
# [-1. -1. -1. 1.]]
```

psychopy.tools.mathtools.intersectRayPlane(rayOrig, rayDir, planeOrig, planeNormal, dtype=None)

Get the point which a ray intersects a plane.

**Parameters**

- **rayOrig** (*array_like*) – Origin of the line in space [x, y, z].
- **rayDir** (*array_like*) – Direction vector of the line [x, y, z].
- **planeOrig** (*array_like*) – Origin of the plane to test [x, y, z].
- **planeNormal** (*array_like*) – Normal vector of the plane [x, y, z].
- **dtype** (*dtype or str, optional*) – Data type for computations can either be ‘float32’ or ‘float64’. If `out` is specified, the data type of `out` is used and this argument is ignored. If `out` is not provided, ‘float64’ is used by default.

**Returns** Position (*ndarray*) in space which the line intersects the plane and the distance the intersect occurs from the origin (*float*). *None* is returned if the line does not intersect the plane at a single point or at all.

**Return type** tuple or *None*

Examples

Find the point in the scene a ray intersects the plane:

```python
# plane information
planeOrigin = [0, 0, 0]
planeNormal = [0, 0, 1]
planeUpAxis = perp([0, 1, 0], planeNormal)

# ray
rayDir = [0, 0, -1]
rayOrigin = [0, 0, 5]

# get the intersect and distance in 3D world space
pnt, dist = intersectRayPlane(rayOrigin, rayDir, planeOrigin, planeNormal)
```

psychopy.tools.mathtools.intersectRaySphere(rayOrig, rayDir, sphereOrig=(0.0, 0.0, 0.0), sphereRadius=1.0, dtype=None)

Calculate the points which a ray/line intersects a sphere (if any).
Get the 3D coordinate of the point which the ray intersects the sphere and the distance to the point from \textit{orig}. The nearest point is returned if the line intersects the sphere at multiple locations. All coordinates should be in world/scene units.

\textbf{Parameters}

- \texttt{rayOrig} (\text{array_like}) – Origin of the ray in space \([x, y, z]\).
- \texttt{rayDir} (\text{array_like}) – Direction vector of the ray \([x, y, z]\), should be normalized.
- \texttt{sphereOrig} (\text{array_like}) – Origin of the sphere to test \([x, y, z]\).
- \texttt{sphereRadius} (\text{float}) – Sphere radius to test in scene units.
- \texttt{dtype} (\text{dtype or str, optional}) – Data type for computations can either be ‘float32’ or ‘float64’. If \texttt{out} is specified, the data type of \texttt{out} is used and this argument is ignored. If \texttt{out} is not provided, ‘float64’ is used by default.

\textbf{Returns} Coordinate in world space of the intersection and distance in scene units from \textit{orig}. Returns \text{None} if there is no intersection.

\textbf{Return type} \text{tuple}

\texttt{psychopy.tools.mathtools.intersectRayAABB} (\texttt{rayOrig}, \texttt{rayDir}, \texttt{boundsOffset}, \texttt{boundsExtents}, \texttt{dtype=\text{None}})

Find the point a ray intersects an axis-aligned bounding box (AABB).

\textbf{Parameters}

- \texttt{rayOrig} (\text{array_like}) – Origin of the ray in space \([x, y, z]\).
- \texttt{rayDir} (\text{array_like}) – Direction vector of the ray \([x, y, z]\), should be normalized.
- \texttt{boundsOffset} (\text{array_like}) – Offset of the bounding box in the scene \([x, y, z]\).
- \texttt{boundsExtents} (\text{array_like}) – Minimum and maximum extents of the bounding box.
- \texttt{dtype} (\text{dtype or str, optional}) – Data type for computations can either be ‘float32’ or ‘float64’. If \texttt{out} is specified, the data type of \texttt{out} is used and this argument is ignored. If \texttt{out} is not provided, ‘float64’ is used by default.

\textbf{Returns} Coordinate in world space of the intersection and distance in scene units from \texttt{rayOrig}. Returns \text{None} if there is no intersection.

\textbf{Return type} \text{tuple}

\textbf{Examples}

Get the point on an axis-aligned bounding box that the cursor is over and place a 3D stimulus there. The eye location is defined by \textit{RigidBodyPose} object \texttt{camera}:

```python
# get the mouse position on-screen
mx, my = mouse.getPos()

# find the point which the ray intersects on the box
result = intersectRayAABB(
    camera.pos,
    camera.transformNormal(win.coordToRay((mx, my))),
    myStim.pos,
    myStim.thePose.bounds.extents)

# if the ray intersects, set the position of the cursor object to it
```

(continues on next page)
if result is not None:
    cursorModel.thePose.pos = result[0]
cursorModel.draw()  # don't draw anything if there is no intersect

Note that if the model is rotated, the bounding box may not be aligned anymore with the axes. Use intersectRayOBB if your model rotates.

psychopy.tools.mathtools.intersectRayOBB(rayOrig, rayDir, modelMatrix, boundsExtents, dtype=None)

Find the point a ray intersects an oriented bounding box (OBB).

**Parameters**

- **rayOrig** (array_like) – Origin of the ray in space [x, y, z].
- **rayDir** (array_like) – Direction vector of the ray [x, y, z], should be normalized.
- **modelMatrix** (array_like) – 4x4 model matrix of the object and bounding box.
- **boundsExtents** (array_like) – Minimum and maximum extents of the bounding box.
- **dtype** (dtype or str, optional) – Data type for computations can either be ‘float32’ or ‘float64’. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, ‘float64’ is used by default.

**Returns**

Coordinate in world space of the intersection and distance in scene units from *rayOrig*. Returns *None* if there is no intersection.

**Return type**

tuple

**Examples**

Get the point on an oriented bounding box that the cursor is over and place a 3D stimulus there. The eye location is defined by *RigidBodyPose* object *camera*:

```python
# get the mouse position on-screen
mx, my = mouse.getPos()

# find the point which the ray intersects on the box
result = intersectRayOBB(
    camera.pos,
    camera.transformNormal(win.coordToRay((mx, my))],
    myStim.thePose.getModelMatrix(),
    myStim.thePose.bounds.extents)

# if the ray intersects, set the position of the cursor object to it
if result is not None:
    cursorModel.thePose.pos = result[0]
cursorModel.draw()  # don't draw anything if there is no intersect
```

psychopy.tools.mathtools.intersectRayTriangle(rayOrig, rayDir, tri, dtype=None)

Get the intersection of a ray and triangle(s).

This function can be used to achieve 'pixel-perfect' ray picking/casting on meshes defined with triangles. However, high-poly meshes may lead to performance issues.

**Parameters**

- **rayOrig** (array_like) – Origin of the ray in space [x, y, z].
• **rayDir** (*array_like*) – Direction vector of the ray [x, y, z], should be normalized.

• **tri** (*array_like*) – Triangle vertices as 2D (3x3) array [p0, p1, p2] where each vertex is a length 3 array [vx, xy, vz]. The input array can be 3D (Nx3x3) to specify multiple triangles.

• **dtype** (*dtype or str, optional*) – Data type for computations can either be ‘float32’ or ‘float64’. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, ‘float64’ is used by default.

**Returns** Coordinate in world space of the intersection, distance in scene units from *rayOrig*, and the barycentric coordinates on the triangle [x, y]. Returns *None* if there is no intersection.

**Return type** tuple

psychopy.tools.mathtools.**ortho3Dto2D**(p, *orig*, normal, *up*, right=None, *dtype=None*)

Get the planar coordinates of an orthogonal projection of a 3D point onto a 2D plane.

This function gets the nearest point on the plane which a 3D point falls on the plane.

**Parameters**

• **p** (*array_like*) – Point to be projected on the plane.

• **orig** (*array_like*) – Origin of the plane to test [x, y, z].

• **normal** (*array_like*) – Normal vector of the plane [x, y, z], must be normalized.

• **up** (*array_like*) – Normalized up (+Y) direction of the plane’s coordinate system. Must be perpendicular to normal.

• **right** (*array_like, optional*) – Perpendicular right (+X) axis. If not provided, the axis will be computed via the cross product between normal and up.

• **dtype** (*dtype or str, optional*) – Data type for computations can either be ‘float32’ or ‘float64’. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, ‘float64’ is used by default.

**Returns** Coordinates on the plane [X, Y] where the 3D point projects towards perpendicularly.

**Return type** ndarray

**Examples**

This function can be used with **intersectRayPlane()** to find the location on the plane the ray intersects:

```python
# plane information
planeOrigin = [0, 0, 0]
planeNormal = [0, 0, 1]  # must be normalized
planeUpAxis = perp([0, 1, 0], planeNormal)  # must also be normalized

# ray
rayDir = [0, 0, -1]
rayOrigin = [0, 0, 5]

# get the intersect in 3D world space
pnt = intersectRayPlane(rayOrigin, rayDir, planeOrigin, planeNormal)

# get the 2D coordinates on the plane the intersect occurred
planeX, planeY = ortho3Dto2D(pnt, planeOrigin, planeNormal, planeUpAxis)
```

psychopy.tools.mathtools.**articulate**(boneVecs, boneOris, *dtype=None*)

Articulate an armature.
This function is used for forward kinematics and posing by specifying a list of ‘bones’. A bone has a length and orientation, where sequential bones are linked end-to-end. Returns the transformed origins of the bones in scene coordinates and their orientations.

There are many applications for forward kinematics such as posing armatures and stimuli for display (eg. mocap data). Another application is for getting the location of the end effector of coordinate measuring hardware, where encoders measure the joint angles and the length of linking members are known. This can be used for computing pose from “Sword of Damocles”[1]_ like hardware or some other haptic input devices which the participant wears (eg. a glove that measures joint angles in the hand). The computed pose of the joints can be used to interact with virtual stimuli.

Parameters

- **boneVecs** (*array_like*) – Bone lengths [x, y, z] as an Nx3 array.
- **boneOris** (*array_like*) – Orientation of the bones as quaternions in form [x, y, z, w], relative to the previous bone.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be ‘float32’ or ‘float64’. If `out` is specified, the data type of `out` is used and this argument is ignored. If `out` is not provided, ‘float64’ is used by default.

Returns

Array of bone origins and orientations. The first origin is root position which is always at [0, 0, 0]. Use `transform()` to reposition the armature, or create a transformation matrix and use `applyMatrix` to translate and rotate the whole armature into position.

Return type: tuple

References

Examples

Compute the orientations and origins of segments of an arm:

```python
# bone lengths
boneLengths = [[0., 1., 0.], [0., 1., 0.], [0., 1., 0.]]

# create quaternions for joints
shoulder = mt.quatFromAxisAngle(’-y’, 45.0)
elbow = mt.quatFromAxisAngle(’+z’, 45.0)
wrist = mt.quatFromAxisAngle(’+z’, 45.0)

# articulate the parts of the arm
boxPos, boxOri = mt.articulate(pos, [shoulder, elbow, wrist])

# assign positions and orientations to 3D objects
shoulderModel.thePose.posOri = (boxPos[0, :], boxOri[0, :])
elbowModel.thePose.posOri = (boxPos[1, :], boxOri[1, :])
wristModel.thePose.posOri = (boxPos[2, :], boxOri[2, :])
```

psychopy.tools.mathtools.slerp (q0, q1, t, shortest=True, out=None, dtype=None)

Spherical linear interpolation (SLERP) between two quaternions.

The behaviour of this function depends on the types of arguments:

- If `q0` and `q1` are both 1-D and `t` is scalar, the interpolation at `t` is returned.
- If `q0` and `q1` are both 2-D Nx4 arrays and `t` is scalar, an Nx4 array is returned with each row containing the interpolation at `t` for each quaternion pair at matching row indices in `q0` and `q1`.
Parameters

- **q0** (*array_like*) – Initial quaternion in form \([x, y, z, w]\) where \(w\) is real and \(x, y, z\) are imaginary components.
- **q1** (*array_like*) – Final quaternion in form \([x, y, z, w]\) where \(w\) is real and \(x, y, z\) are imaginary components.
- **t** (*float*) – Interpolation weight factor within interval 0.0 and 1.0.
- **shortest** (*bool, optional*) – Ensure interpolation occurs along the shortest arc along the 4-D hypersphere (default is True).
- **out** (*ndarray, optional*) – Optional output array. Must be same shape and dtype as the expected output if out was not specified.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be ‘float32’ or ‘float64’. If out is specified, the data type of out is used and this argument is ignored. If out is not provided, ‘float64’ is used by default.

**Returns**  Quaternion \([x, y, z, w]\) at \(t\).

**Return type**  ndarray

Examples

Interpolate between two orientations:

```python
def q0 = quatFromAxisAngle(90.0, degrees=True)
def q1 = quatFromAxisAngle(-90.0, degrees=True)
# halfway between 90 and -90 is 0.0 or quaternion [0. 0. 0. 1.]
def qr = slerp(q0, q1, 0.5)
```

Example of smooth rotation of an object with fixed angular velocity:

```python
def degPerSec = 10.0  # rotate a stimulus at 10 degrees per second

def initial orientation, axis rotates in the Z direction
def qr = quatFromAxisAngle([0., 0., -1.], 0.0, degrees=True)
def amount to rotate every second
def qv = quatFromAxisAngle([0., 0., -1.], degPerSec, degrees=True)

# ---- within main experiment loop ----
# 'frameTime' is the time elapsed in seconds from last 'slerp'.
def qr = multQuat(qr, slerp([0., 0., 0., 1.], qv, degPerSec * frameTime))
_, angle = quatToAxisAngle(qr)  # discard axis, only need angle
def myStim is a GratingStim or anything with an 'ori' argument which
# accepts angle in degrees
myStim.ori = angle
myStim.draw()
```

convert a quaternion to axis and angle representation.

This allows you to use quaternions to set the orientation of stimuli that have an ori property.

**Parameters**

- **q** (*tuple, list or ndarray of float*) – Quaternion in form \([x, y, z, w]\) where \(w\) is real and \(x, y, z\) are imaginary components.
**quatFromAxisAngle**

Create a quaternion to represent a rotation about `axis` vector by `angle`.

**Parameters**

- **axis** (tuple, list, ndarray or str) – Axis vector components or axis name. If a vector, input must be length 3 [x, y, z]. A string can be specified for rotations about world axes (e.g. ‘+x’, ‘-z’, ‘+y’, etc.)

- **angle** (float) – Rotation angle in radians (or degrees if `degrees` is `True`). Rotations are right-handed about the specified `axis`.

- **degrees** (bool, optional) – Indicate `angle` is in degrees, otherwise `angle` will be treated as radians.

- **dtype** (dtype or str, optional) – Data type for computations can either be ‘float32’ or ‘float64’. If `out` is specified, the data type of `out` is used and this argument is ignored. If `out` is not provided, ‘float64’ is used by default.

**Returns** Quat [x, y, z, w].

**Return type** ndarray

**Examples**

Using a quaternion to rotate a stimulus a fixed angle each frame:

```python
# initial orientation, axis rotates in the Z direction
qr = quatFromAxisAngle([0., 0., -1.], 0.0, degrees=True)
# rotation per-frame, here it's 0.1 degrees per frame
qf = quatFromAxisAngle([0., 0., -1.], 0.1, degrees=True)

# ---- within main experiment loop ----
# myStim is a GratingStim or anything with an 'ori' argument which
# accepts angle in degrees
qr = multQuat(qr, qf)  # cumulative rotation
_, angle = quatToAxisAngle(qr)  # discard axis, only need angle
myStim.ori = angle
myStim.draw()
```

**Examples**

Using a quaternion to rotate a stimulus a fixed angle each frame:

```python
# initial orientation, axis rotates in the Z direction
qr = quatFromAxisAngle([0., 0., -1.], 0.0, degrees=True)
# rotation per-frame, here it's 0.1 degrees per frame
qf = quatFromAxisAngle([0., 0., -1.], 0.1, degrees=True)

# ---- within main experiment loop ----
# myStim is a GratingStim or anything with an 'ori' argument which
# accepts angle in degrees
qr = multQuat(qr, qf)  # cumulative rotation
_, angle = quatToAxisAngle(qr)  # discard axis, only need angle
myStim.ori = angle
myStim.draw()
```

**Examples**

Using a quaternion to rotate a stimulus a fixed angle each frame:

```python
# initial orientation, axis rotates in the Z direction
qr = quatFromAxisAngle([0., 0., -1.], 0.0, degrees=True)
# rotation per-frame, here it's 0.1 degrees per frame
qf = quatFromAxisAngle([0., 0., -1.], 0.1, degrees=True)

# ---- within main experiment loop ----
# myStim is a GratingStim or anything with an 'ori' argument which
# accepts angle in degrees
qr = multQuat(qr, qf)  # cumulative rotation
_, angle = quatToAxisAngle(qr)  # discard axis, only need angle
myStim.ori = angle
myStim.draw()
```
Examples

Create a quaternion from specified axis and angle:

```python
axis = [0., 0., -1.]  # rotate about -Z axis
angle = 90.0  # angle in degrees
ori = quatFromAxisAngle(axis, angle, degrees=True)  # using degrees!
```

`psychopy.tools.mathtools.quatYawPitchRoll(q, degrees=True, out=None, dtype=None)`

Get the yaw, pitch, and roll of a quaternion’s orientation relative to the world -Z axis.

You can multiply the quaternion by the inverse of some other one to make the returned values referenced to a local coordinate system.

**Parameters**

- `q` (tuple, list or ndarray of float) – Quaternion in form [x, y, z, w] where w is real and x, y, z are imaginary components.
- `degrees` (bool, optional) – Indicate angles are to be returned in degrees, otherwise they will be returned in radians.
- `out` (ndarray) – Optional output array. Must have same shape and dtype as what is expected to be returned by this function of out was not specified.
- `dtype` (dtype or str, optional) – Data type for computations can either be ‘float32’ or ‘float64’. If out is specified, the data type of out is used and this argument is ignored. If out is not provided, ‘float64’ is used by default.

**Returns** Yaw, pitch and roll [yaw, pitch, roll] of quaternion q.

**Return type** ndarray

`psychopy.tools.mathtools.alignTo(v, t, out=None, dtype=None)`

Compute a quaternion which rotates one vector to align with another.

**Parameters**

- `v` (array_like) – Vector [x, y, z] to rotate. Can be Nx3, but must have the same shape as t.
- `t` (array_like) – Target [x, y, z] vector to align to. Can be Nx3, but must have the same shape as v.
- `out` (ndarray, optional) – Optional output array. Must be same shape and dtype as the expected output if out was not specified.
- `dtype` (dtype or str, optional) – Data type for computations can either be ‘float32’ or ‘float64’. If out is specified, the data type of out is used and this argument is ignored. If out is not provided, ‘float64’ is used by default.

**Returns** Quaternion which rotates v to t.

**Return type** ndarray
Examples

Rotate some vectors to align with other vectors, inputs should be normalized:

```python
vec = [[[1, 0, 0], [0, 1, 0], [1, 0, 0]]
targets = [[[0, 1, 0], [0, -1, 0], [-1, 0, 0]]
qr = alignTo(vec, targets)
vecRotated = applyQuat(qr, vec)
numpy.allclose(vecRotated, targets)  # True
```

Get matrix which orients vertices towards a point:

```python
point = [5, 6, 7]
vec = [0, 0, -1]  # initial facing is -Z (forward in GL)
targetVec = normalize(point - vec)
qr = alignTo(vec, targetVec)  # get rotation to align
M = quatToMatrix(qr)  # 4x4 transformation matrix
```

psychopy.tools.mathtools.quatMagnitude(q, squared=False, out=None, dtype=None)
Get the magnitude of a quaternion.

A quaternion is normalized if its magnitude is 1.

**Parameters**

- `q (array_like)` – Quaternion(s) in form [x, y, z, w] where w is real and x, y, z are imaginary components.
- `squared (bool, optional)` – If True return the squared magnitude. If you are just checking if a quaternion is normalized, the squared magnitude will suffice to avoid the square root operation.
- `out (ndarray, optional)` – Optional output array. Must be same shape and dtype as the expected output if out was not specified.
- `dtype (dtype or str, optional)` – Data type for computations can either be ‘float32’ or ‘float64’. If out is specified, the data type of out is used and this argument is ignored. If out is not provided, ‘float64’ is used by default.

**Returns**

Magnitude of quaternion q.

**Return type**

float or ndarray

psychopy.tools.mathtools.multQuat(q0, q1, out=None, dtype=None)
Multiply quaternion q0 and q1.

The orientation of the returned quaternion is the combination of the input quaternions.

**Parameters**

- `q0 (array_like)` – Quaternions to multiply in form [x, y, z, w] where w is real and x, y, z are imaginary components. If 2D (Nx4) arrays are specified, quaternions are multiplied row-wise between each array.
- `q1 (array_like)` – Quaternions to multiply in form [x, y, z, w] where w is real and x, y, z are imaginary components. If 2D (Nx4) arrays are specified, quaternions are multiplied row-wise between each array.
• **out** (*ndarray, optional*) – Optional output array. Must be same shape and dtype as the expected output if **out** was not specified.

• **dtype** (*dtype or str, optional*) – Data type for computations can either be ‘float32’ or ‘float64’. If **out** is specified, the data type of **out** is used and this argument is ignored. If **out** is not provided, ‘float64’ is used by default.

**Returns** Combined orientations of **q0** and **q1**.

**Return type** *ndarray*

**Notes**

• Quaternions are normalized prior to multiplication.

**Examples**

Combine the orientations of two quaternions:

```python
a = quatFromAxisAngle([0, 0, -1], 45.0, degrees=True)
b = quatFromAxisAngle([0, 0, -1], 90.0, degrees=True)
c = multQuat(a, b)  # rotates 135 degrees about -Z axis
```

psychopy.tools.mathtools.accumQuat (*qlist, out=None, dtype=None*)

Accumulate quaternion rotations.

Chain multiplies an Nx4 array of quaternions, accumulating their rotations. This function can be used for computing the orientation of joints in an armature for forward kinematics. The first quaternion is treated as the ‘root’ and the last is the orientation of the end effector.

**Parameters**

• **q** (*array_like*) – Nx4 array of quaternions to accumulate, where each row is a quaternion.

• **out** (*ndarray, optional*) – Optional output array. Must be same shape and dtype as the expected output if **out** was not specified. In this case, the same shape as **qlist**.

• **dtype** (*dtype or str, optional*) – Data type for computations can either be ‘float32’ or ‘float64’. If **out** is specified, the data type of **out** is used and this argument is ignored. If **out** is not provided, ‘float64’ is used by default.

**Returns** Nx4 array of quaternions.

**Return type** *ndarray*

**Examples**

Get the orientation of joints in an armature if we know their relative angles:

```python
shoulder = quatFromAxisAngle('-x', 45.0)  # rotate shoulder down 45 deg
elkow = quatFromAxisAngle('+x', 45.0)  # rotate elbow up 45 deg
wrist = quatFromAxisAngle('-x', 45.0)  # rotate wrist down 45 deg
finger = quatFromAxisAngle('+x', 0.0)  # keep finger in-line with wrist
armRotations = accumQuat([[shoulder, elbow, wrist, finger]])
```
Get the multiplicative inverse of a quaternion.

This gives a quaternion which rotates in the opposite direction with equal magnitude. Multiplying a quaternion by its inverse returns an identity quaternion as both orientations cancel out.

**Parameters**

- **q** (ndarray, list, or tuple of float) – Quaternion to invert in form \([x, y, z, w]\) where \(w\) is real and \(x, y, z\) are imaginary components. If \(q\) is 2D (Nx4), each row is treated as a separate quaternion and inverted.

- **out** (ndarray, optional) – Optional output array. Must be same shape and dtype as the expected output if \(out\) was not specified.

- **dtype** (dtype or str, optional) – Data type for computations can either be ‘float32’ or ‘float64’. If \(out\) is specified, the data type of \(out\) is used and this argument is ignored. If \(out\) is not provided, ‘float64’ is used by default.

**Returns**

Inverse of quaternion \(q\).

**Return type**

ndarray

**Examples**

Show that multiplying a quaternion by its inverse returns an identity quaternion where \([x=0, y=0, z=0, w=1]\):

```python
angle = 90.0
axis = [0., 0., -1.]
q = quatFromAxisAngle(axis, angle, degrees=True)
qinv = invertQuat(q)
qr = multQuat(q, qinv)
qi = np.array([0., 0., 0., 1.])  # identity quaternion
print(np.allclose(qi, qr))  # True
```

**Notes**

- Quaternions are normalized prior to inverting.

**psychopy.tools.mathtools.applyQuat** (q, points, out=None, dtype=None)

Rotate points/coordinates using a quaternion.

This is similar to using applyMatrix with a rotation matrix. However, it is computationally less intensive to use applyQuat if one only wishes to rotate points.

**Parameters**

- **q** (array_like) – Quaternion to invert in form \([x, y, z, w]\) where \(w\) is real and \(x, y, z\) are imaginary components.

- **points** (array_like) – 2D array of vectors or points to transform, where each row is a single point. Only the \(x, y,\) and \(z\) components (the first three columns) are rotated. Additional columns are copied.

- **out** (ndarray, optional) – Optional output array. Must be same shape and dtype as the expected output if \(out\) was not specified.
• **dtype** *(dtype or str, optional)* – Data type for computations can either be ‘float32’ or ‘float64’. If `out` is specified, the data type of `out` is used and this argument is ignored. If `out` is not provided, ‘float64’ is used by default.

**Returns** Transformed points.

**Return type** ndarray

**Examples**

Rotate points using a quaternion:

```python
points = [[1., 0., 0.], [0., -1., 0.]]
quat = quatFromAxisAngle(-90.0, [0., 0., -1.], degrees=True)
pointsRotated = applyQuat(quat, points)
# [[0. 1. 0.]
# [1. 0. 0.]]
```

Show that you get the same result as a rotation matrix:

```python
axis = [0., 0., -1.]
angle = -90.0
rotMat = rotationMatrix(axis, angle)[:3, :3]  # rotation sub-matrix only
rotQuat = quatFromAxisAngle(axis, axis, degrees=True)
points = [[1., 0., 0.], [0., -1., 0.]]
isClose = np.allclose(applyMatrix(rotMat, points),  # True
                      applyQuat(rotQuat, points))
```

Specifying an array to `q` where each row is a quaternion transforms points in corresponding rows of `points`:

```python
points = [[1., 0., 0.], [0., -1., 0.]]
quats = [quatFromAxisAngle(-90.0, [0., 0., -1.], degrees=True),
         quatFromAxisAngle(45.0, [0., 0., -1.], degrees=True)]
applyQuat(quats, points)
```

**psychopy.tools.mathtools**.**quatToMatrix**(q, out=None, dtype=None)

Create a 4x4 rotation matrix from a quaternion.

**Parameters**

- **q** *(tuple, list or ndarray of float)* – Quaternion to convert in form [x, y, z, w] where w is real and x, y, z are imaginary components.

- **out** *(ndarray or None)* – Optional output array. Must be same shape and dtype as the expected output if `out` was not specified.

- **dtype** *(dtype or str, optional)* – Data type for computations can either be ‘float32’ or ‘float64’. If `out` is specified, the data type of `out` is used and this argument is ignored. If `out` is not provided, ‘float64’ is used by default.

**Returns** 4x4 rotation matrix in row-major order.

**Return type** ndarray or None
Examples

Convert a quaternion to a rotation matrix:

```python
point = [0., 1., 0., 1.]  # 4-vector form [x, y, z, 1.0]
ori = [0., 0., 0., 1.]
rotMat = quatToMatrix(ori)
# rotate 'point' using matrix multiplication
newPoint = np.matmul(rotMat.T, point)  # returns [-1., 0., 0., 1.]
```

Rotate all points in an array (each row is a coordinate):

```python
points = np.asarray([[0., 0., 0., 1.],
                     [0., 1., 0., 1.],
                     [1., 1., 0., 1.]])
newPoints = points.dot(rotMat)
```

Notes

• Quaternions are normalized prior to conversion.

psychopy.tools.mathtools.matrixToQuat (m, out=None, dtype=None)

Convert a rotation matrix to a quaternion.

Parameters

• m (array_like) – 3x3 rotation matrix (row-major). A 4x4 affine transformation matrix may be provided, assuming the top-left 3x3 sub-matrix is orthonormal and is a rotation group.

• out (ndarray, optional) – Optional output array. Must be same shape and dtype as the expected output if out was not specified.

• dtype (dtype or str, optional) – Data type for computations can either be ‘float32’ or ‘float64’. If out is specified, the data type of out is used and this argument is ignored. If out is not provided, ‘float64’ is used by default.

Returns
Rotation quaternion.

Return type
ndarray

Notes

• Depending on the input, returned quaternions may not be exactly the same as the one used to construct the rotation matrix (i.e. by calling quatToMatrix), typically when a large rotation angle is used. However, the returned quaternion should result in the same rotation when applied to points.
Examples

Converting a rotation matrix from the OpenGL matrix stack to a quaternion:

```python
glRotatef(45., -1, 0, 0)

m = np.zeros((4, 4), dtype='float32')  # store the matrix
GL.glGetFloatv(
    GL.GL_MODELVIEW_MATRIX,
    m.ctypes.data_as(ctypes.POINTER(ctypes.c_float)))

qr = matrixToQuat(m.T)  # must be transposed
```

Interpolation between two 4x4 transformation matrices:

```python
interpWeight = 0.5

posStart = mStart[:, 3]
oriStart = matrixToQuat(mStart)

posEnd = mEnd[:, 3]
oriEnd = matrixToQuat(mEnd)

oriInterp = slerp(qStart, qEnd, interpWeight)
posInterp = lerp(posStart, posEnd, interpWeight)
mInterp = posOriToMatrix(posInterp, oriInterp)
```

`psychopy.tools.mathtools.matrixFromEulerAngles(rx, ry, rz, degrees=True, out=None, dtype=None)`

Construct a 4x4 rotation matrix from Euler angles.

Rotations are combined by first rotating about the X axis, then Y, and finally Z.

Parameters

- `rx` (*float*) – Rotation angles (pitch, yaw, and roll).
- `ry` (*float*) – Rotation angles (pitch, yaw, and roll).
- `rz` (*float*) – Rotation angles (pitch, yaw, and roll).
- `degrees` (*bool, optional*) – Rotation angles are specified in degrees. If `False`, they will be assumed as radians. Default is `True`.
- `out` (*ndarray, optional*) – Optional output array. Must be same shape and `dtype` as the expected output if `out` was not specified.
- `dtype` (*dtype or str, optional*) – Data type for computations can either be ‘float32’ or ‘float64’. If `out` is specified, the data type of `out` is used and this argument is ignored. If `out` is not provided, ‘float64’ is used by default.

Returns 4x4 rotation matrix.

Return type `ndarray`
Examples

Demonstration of how a combination of axis-angle rotations is equivalent to a single call of `matrixFromEulerAngles`:

```python
m1 = matrixFromEulerAngles(90., 45., 135.))

# construct rotation matrix from 3 orthogonal rotations
rx = rotationMatrix(90., (1, 0, 0)) # x-axis
ry = rotationMatrix(45., (0, 1, 0)) # y-axis
rz = rotationMatrix(135., (0, 0, 1)) # z-axis
m2 = concatenate([rz, ry, rx]) # note the order
print(numpy.allclose(m1, m2)) # True
```

Not only does `matrixFromEulerAngles` require less code, it also is considerably more efficient than constructing and multiplying multiple matrices.

`psychopy.tools.mathtools.scaleMatrix(s, out=None, dtype=None)`

Create a scaling matrix.

Parameters

- **s** (array_like, float or int) – Scaling factor(s). If s is scalar (float), scaling will be uniform. Providing a vector of scaling values [sx, sy, sz] will result in an anisotropic scaling matrix if any of the values differ.
- **out** (ndarray, optional) – Optional output array. Must be same shape and dtype as the expected output if out was not specified.
- **dtype** (dtype or str, optional) – Data type for computations can either be ‘float32’ or ‘float64’. If out is specified, the data type of out is used and this argument is ignored. If out is not provided, ‘float64’ is used by default.

Returns

4x4 scaling matrix in row-major order.

Return type

ndarray

`psychopy.tools.mathtools.rotationMatrix(angle, axis=(0.0, 0.0, -1.0), out=None, dtype=None)`

Create a rotation matrix.

The resulting matrix will rotate points about `axis` by `angle`. The resulting matrix is similar to that produced by a `glRotate` call.

Parameters

- **angle** (float) – Rotation angle in degrees.
- **axis** (array_like or str) – Axis vector components or axis name. If a vector, input must be length 3. A string can be specified for rotations about world axes (eg. ‘+x’, ‘+z’, ‘+y’, etc.)
- **out** (ndarray, optional) – Optional output array. Must be same shape and dtype as the expected output if out was not specified.
- **dtype** (dtype or str, optional) – Data type for computations can either be ‘float32’ or ‘float64’. If out is specified, the data type of out is used and this argument is ignored. If out is not provided, ‘float64’ is used by default.

Returns

4x4 scaling matrix in row-major order. Will be the same array as out if specified, if not, a new array will be allocated.
Return type  ndarray

Notes

• Vector axis is normalized before creating the matrix.

psychopy.tools.mathtools.translationMatrix(t, out=None, dtype=None)
Create a translation matrix.

The resulting matrix is the same as generated by a glTranslate call.

Parameters

• t (ndarray, tuple, or list of float) – Translation vector [tx, ty, tz].

• out (ndarray, optional) – Optional output array. Must be same shape and dtype as the expected output if out was not specified.

• dtype (dtype or str, optional) – Data type for computations can either be ‘float32’ or ‘float64’. If out is specified, the data type of out is used and this argument is ignored. If out is not provided, ‘float64’ is used by default.

Returns 4x4 translation matrix in row-major order. Will be the same array as out if specified, if not, a new array will be allocated.

Return type  ndarray

psychopy.tools.mathtools.invertMatrix(m, out=None, dtype=None)
Invert a square matrix.

Parameters

• m (array_like) – Square matrix to invert. Inputs can be 4x4, 3x3 or 2x2.

• out (ndarray, optional) – Optional output array. Must be same shape and dtype as the expected output if out was not specified.

• dtype (dtype or str, optional) – Data type for computations can either be ‘float32’ or ‘float64’. If out is specified, the data type of out is used and this argument is ignored. If out is not provided, ‘float64’ is used by default.

Returns Matrix which is the inverse of m

Return type  ndarray

psychopy.tools.mathtools.isOrthogonal(m)
Check if a square matrix is orthogonal.

If a matrix is orthogonal, its columns form an orthonormal basis and is non-singular. An orthogonal matrix is invertible by simply taking the transpose of the matrix.

Parameters m (array_like) – Square matrix, either 2x2, 3x3 or 4x4.

Returns True if the matrix is orthogonal.

Return type  bool

psychopy.tools.mathtools.isAffine(m)
Check if a 4x4 square matrix describes an affine transformation.

Parameters m (array_like) – 4x4 transformation matrix.

Returns True if the matrix is affine.

Return type  bool
psychopy.tools.mathtools.multMatrix(matrices, reverse=False, out=None, dtype=None)

Chain multiplication of two or more matrices.

Multiply a sequence of matrices together, reducing to a single product matrix. For instance, specifying matrices the sequence of matrices (A, B, C, D) will return the product (((AB)C)D). If reverse=True, the product will be (A(B(CD))).

Alternatively, a 3D array can be specified to matrices as a stack, where an index along axis 0 references a 2D slice storing matrix values. The product of the matrices along the axis will be returned. This is a bit more efficient than specifying separate matrices in a sequence, but the difference is negligible when only a few matrices are being multiplied.

Parameters

- **matrices** *(list, tuple or ndarray)* – Sequence or stack of matrices to multiply. All matrices must have the same dimensions.

- **reverse** *(bool, optional)* – Multiply matrices right-to-left. This is useful when dealing with transformation matrices, where the order of operations for transforms will appear the same as the order the matrices are specified. Default is ‘False’. When True, this function behaves similarly to concatenate().

- **out** *(ndarray, optional)* – Optional output array. Must be same shape and dtype as the expected output if out was not specified.

- **dtype** *(dtype or str, optional)* – Data type for computations can either be ‘float32’ or ‘float64’. If out is specified, the data type of out is used and this argument is ignored. If out is not provided, ‘float64’ is used by default.

Returns Matrix product.

Return type ndarray

Notes

- You may use numpy.matmul when dealing with only two matrices instead of multMatrix.

- If a single matrix is specified, the returned product will have the same values.

Examples

Chain multiplication of SRT matrices:

```python
translate = translationMatrix((0.035, 0, -0.5))
rotate = rotationMatrix(90.0, (0, 1, 0))
scale = scaleMatrix(2.0)
SRT = multMatrix((translate, rotate, scale))
```

Same as above, but matrices are in a 3x4x4 array:

```python
matStack = np.array((translate, rotate, scale))
# or ...
# matStack = np.zeros((3, 4, 4))
# matStack[0, :, :] = translate
# matStack[1, :, :] = rotate
# matStack[2, :, :] = scale
```

(continues on next page)
SRT = multMatrix(matStack)

Using `reverse=True` allows you to specify transformation matrices in the order which they will be applied:

\[
\text{SRT} = \text{multMatrix}(\text{np.array((scale, rotate, translate)), reverse=True})
\]

`psychopy.tools.mathtools.concatenate(matrices, out=None, dtype=None)`

Concatsenate matrix transformations.

Chain multiply matrices describing transform operations into a single matrix product, that when applied, transforms points and vectors with each operation in the order they’re specified.

**Parameters**

- `matrices (list or tuple)` – List of matrices to concatenate. All matrices must all have the same size, usually 4x4 or 3x3.
- `out (ndarray, optional)` – Optional output array. Must be same shape and dtype as the expected output if `out` was not specified.
- `dtype (dtype or str, optional)` – Data type for computations can either be ‘float32’ or ‘float64’. If `out` is specified, the data type of `out` is used and this argument is ignored. If `out` is not provided, ‘float64’ is used by default.

**Returns** Matrix product.

**Return type** ndarray

**See also:**

- `multMatrix` : Chain multiplication of matrices.

**Notes**

- This function should only be used for combining transformation matrices. Use `multMatrix` for general matrix chain multiplication.

**Examples**

Create an SRT (scale, rotate, and translate) matrix to convert model-space coordinates to world-space:

\[
S = \text{scaleMatrix}([2.0, 2.0, 2.0]) \quad \# \text{scale model 2x}
R = \text{rotationMatrix}(-90., [0., 0., -1]) \quad \# \text{rotate -90 about -Z axis}
T = \text{translationMatrix}([0., 0., -5.]) \quad \# \text{translate point 5 units away}
\]

\[
\# \text{product matrix when applied to points will scale, rotate and transform}
\# \text{in that order.}
\text{SRT} = \text{concatenate}([S, R, T])
\]

\[
\# \text{transform a point in model-space coordinates to world-space}
\text{pointModel} = \text{np.array}([0., 1., 0., 1.])
\text{pointWorld} = \text{np.matmul(SRT, pointModel.T)} \quad \# \text{point in WCS}
\]

\[
\# \ldots \text{or} \ldots
\text{pointWorld} = \text{matrixApply(SRT, pointModel)}
\]
Create a model-view matrix from a world-space pose represented by an orientation (quaternion) and position (vector). The resulting matrix will transform model-space coordinates to eye-space:

```python
# eye pose as quaternion and vector
stimOri = quatFromAxisAngle([0., 0., -1.], -45.0)
stimPos = [0., 1.5, -5.]

# create model matrix
R = quatToMatrix(stimOri)
T = translationMatrix(stimPos)
M = concatenate(R, T)  # model matrix

# create a view matrix, can also be represented as 'pos' and 'ori'
eyePos = [0., 1.5, 0.]
eyeFwd = [0., 0., -1.]
eyeUp = [0., 1., 0.]
V = lookAt(eyePos, eyeFwd, eyeUp)  # from viewtools

# modelview matrix
MV = concatenate([M, V])
```

You can put the created matrix in the OpenGL matrix stack as shown below. Note that the matrix must have a 32-bit floating-point data type and needs to be loaded transposed since OpenGL takes matrices in column-major order:

```python
GL.glMatrixMode(GL.GL_MODELVIEW)

# pyglet
MV = np.asarray(MV, dtype='float32')  # must be 32-bit float!
ptrMV = MV.ctypes.data_as(ctypes.POINTER(ctypes.c_float))
GL.glLoadTransposeMatrixf(ptrMV)

# PyOpenGL
MV = np.asarray(MV, dtype='float32')
GL.glLoadTransposeMatrixf(MV)
```

Furthermore, you can convert a point from model-space to homogeneous clip-space by concatenating the projection, view, and model matrices:

```python
# compute projection matrix, functions here are from 'viewtools'
screenWidth = 0.52
screenAspect = w / h
scrDistance = 0.55
frustum = computeFrustum(screenWidth, screenAspect, scrDistance)
P = perspectiveProjectionMatrix(*frustum)

# multiply model-space points by MVP to convert them to clip-space
MVP = concatenate([M, V, P])
pointModel = np.array([0., 1., 0., 1.])
pointClipSpace = np.matmul(MVP, pointModel.T)
```

`psychopy.tools.mathtools.normalMatrix(modelMatrix, out=None, dtype=None)`

Get the normal matrix from a model matrix.

**Parameters**

- `modelMatrix (array_like)` – 4x4 homogeneous model matrix.
- `out (ndarray, optional)` – Optional output array. Must be same shape and dtype as...
the expected output if `out` was not specified.

- **dtype (dtype or str, optional)** – Data type for computations can either be ‘float32’ or ‘float64’. If `out` is specified, the data type of `out` is used and this argument is ignored. If `out` is not provided, ‘float64’ is used by default.

Returns Normal matrix.

Return type ndarray

psychopy.tools.mathtools.forwardProject (objPos, modelView, proj, viewport=None, out=None, dtype=None)

Project a point in a scene to a window coordinate.

This function is similar to `gluProject` and can be used to find the window coordinate which a point projects to.

Parameters

- **objPos (array_like)** – Object coordinates (x, y, z). If an Nx3 array of coordinates is specified, where each row contains a window coordinate this function will return an array of projected coordinates with the same size.

- **modelView (array_like)** – 4x4 combined model and view matrix for returned value to be object coordinates. Specify only the view matrix for a coordinate in the scene.

- **proj (array_like)** – 4x4 projection matrix used for rendering.

- **viewport (array_like)** – Viewport rectangle for the window [x, y, w, h]. If not specified, the returned values will be in normalized device coordinates.

- **out (ndarray, optional)** – Optional output array. Must be same shape and dtype as the expected output if `out` was not specified.

- **dtype (dtype or str, optional)** – Data type for computations can either be ‘float32’ or ‘float64’. If `out` is specified, the data type of `out` is used and this argument is ignored. If `out` is not provided, ‘float64’ is used by default.

Returns Normalized device or viewport coordinates [x, y, z] of the point. The z component is similar to the depth buffer value for the object point.

Return type ndarray

psychopy.tools.mathtools.reverseProject (winPos, modelView, proj, viewport=None, out=None, dtype=None)

Unproject window coordinates into object or scene coordinates.

This function works like `gluUnProject` and can be used to find to an object or scene coordinate at the point on-screen (mouse coordinate or pixel). The coordinate can then be used to create a direction vector from the viewer’s eye location. Another use of this function is to convert depth buffer samples to object or scene coordinates. This is the inverse operation of `forwardProject()`.

Parameters

- **winPos (array_like)** – Window coordinates (x, y, z). If `viewport` is not specified, these should be normalized device coordinates. If an Nx3 array of coordinates is specified, where each row contains a window coordinate this function will return an array of unprojected coordinates with the same size. Usually, you only need to specify the x and y coordinate, leaving z as zero. However, you can specify z if sampling from a depth map or buffer to convert a depth sample to an actual location.

- **modelView (array_like)** – 4x4 combined model and view matrix for returned value to be object coordinates. Specify only the view matrix for a coordinate in the scene.

- **proj (array_like)** – 4x4 projection matrix used for rendering.
• **viewport** ([array_like]) – Viewport rectangle for the window [x, y, w, h]. Do not specify one if `winPos` is in already in normalized device coordinates.

• **out** ([ndarray, optional]) – Optional output array. Must be same `shape` and `dtype` as the expected output if `out` was not specified.

• **dtype** ([dtype or str, optional]) – Data type for computations can either be ‘float32’ or ‘float64’. If `out` is specified, the data type of `out` is used and this argument is ignored. If `out` is not provided, ‘float64’ is used by default.

**Returns** Object or scene coordinates.

**Return type** `ndarray`

```python
psychopy.tools.mathtools.applyMatrix(m, points, out=None, dtype=None)
```

Apply a matrix over a 2D array of points.

This function behaves similarly to the following `Numpy` statement:

```python
points[:, i] = points.dot(m.T)
```

Transformation matrices specified to `m` must have dimensions 4x4, 3x4, 3x3 or 2x2. With the exception of 4x4 matrices, input `points` must have the same number of columns as the matrix has rows. 4x4 matrices can be used to transform both Nx4 and Nx3 arrays.

**Parameters**

• **m** ([array_like]) – Matrix with dimensions 2x2, 3x3, 3x4 or 4x4.

• **points** ([array_like]) – 2D array of points/coordinates to transform. Each row should have length appropriate for the matrix being used.

• **out** ([ndarray, optional]) – Optional output array. Must be same `shape` and `dtype` as the expected output if `out` was not specified.

• **dtype** ([dtype or str, optional]) – Data type for computations can either be ‘float32’ or ‘float64’. If `out` is specified, the data type of `out` is used and this argument is ignored. If `out` is not provided, ‘float64’ is used by default.

**Returns** Transformed coordinates.

**Return type** `ndarray`

**Notes**

• Input (`points`) and output (`out`) arrays cannot be the same instance for this function.

• In the case of 4x4 input matrices, this function performs optimizations based on whether the input matrix is affine, greatly improving performance when working with Nx3 arrays.
Examples

Construct a matrix and transform a point:

```python
# identity 3x3 matrix for this example
M = [
    [1.0, 0.0, 0.0],
    [0.0, 1.0, 0.0],
    [0.0, 0.0, 1.0]]
pnt = [1.0, 0.0, 0.0]
pntNew = applyMatrix(M, pnt)
```

Construct an SRT matrix (scale, rotate, transform) and transform an array of points:

```python
S = scaleMatrix([5.0, 5.0, 5.0])  # scale 5x
R = rotationMatrix([180.0, [0., 0., -1]])  # rotate 180 degrees
T = translationMatrix([0., 1.5, -3.])  # translate point up and away
M = concatenate([S, R, T])  # create transform matrix

# points to transform
points = np.array([[0., 1., 0., 1.], [-1., 0., 0., 1.]])  # [x, y, z, w]
newPoints = applyMatrix(M, points)  # apply the transformation
```

Convert CIE-XYZ colors to sRGB:

```python
sRGBMatrix = [[3.2404542, -1.5371385, -0.4985314],
              [-1.969266, 1.8760108, 0.041556],
              [0.0556434, -0.2040259, 1.0572252]]

colorsRGB = applyMatrix(sRGBMatrix, colorsXYZ)
```

**psychopy.tools.mathtools.posOriToMatrix** *(pos, ori=None, out=None, dtype=None)*

Convert a rigid body pose to a 4x4 transformation matrix.

A pose is represented by a position coordinate `pos` and orientation quaternion `ori`.

**Parameters**

- **pos** *(ndarray, tuple, or list of float)* – Position vector [x, y, z].
- **ori** *(tuple, list or ndarray of float)* – Orientation quaternion in form [x, y, z, w] where w is real and x, y, z are imaginary components.
- **out** *(ndarray, optional)* – Optional output array. Must be same shape and dtype as the expected output if `out` was not specified.
- **dtype** *(dtype or str, optional)* – Data type for computations can either be ‘float32’ or ‘float64’. If `out` is specified, the data type of `out` is used and this argument is ignored. If `out` is not provided, ‘float64’ is used by default.

**Returns** 4x4 transformation matrix.

**Return type** ndarray

**psychopy.tools.mathtools.transform** *(pos, ori, points=None, out=None, dtype=None)*

Transform points using a position and orientation. Points are rotated then translated.

**Parameters**

- **pos** *(array_like)* – Position vector in form [x, y, z] or [x, y, z, 1].
• **ori** (*array_like*) – Orientation quaternion in form [x, y, z, w] where w is real and x, y, z are imaginary components.

• **points** (*array_like*) – Point(s) [x, y, z] to transform.

• **out** (*ndarray, optional*) – Optional output array. Must be same shape and dtype as the expected output if `out` was not specified.

• **dtype** (*dtype or str, optional*) – Data type for computations can either be ‘float32’ or ‘float64’. If `out` is specified, the data type of `out` is used and this argument is ignored. If `out` is not provided, ‘float64’ is used by default.

**Returns**

Transformed points.

**Return type** *ndarray*

**Examples**

Transform points by a position coordinate and orientation quaternion:

```python
# rigid body pose
ori = quatFromAxisAngle([0., 0., -1.], 90.0, degrees=True)
pos = [0., 1.5, -3.]
# points to transform
points = np.array([[0., 1., 0., 1.], [-1., 0., 0., 1.]])  # [x, y, z, 1]
outPoints = np.zeros_like(points)  # output array
transform(pos, ori, points, out=outPoints)  # do the transformation
```

You can get the same results as the previous example using a matrix by doing the following:

```python
R = rotationMatrix(90., [0., 0., -1])
T = translationMatrix([0., 1.5, -3.])
M = concatenate([R, T])
applyMatrix(M, points, out=outPoints)
```

If you are defining transformations with quaternions and coordinates, you can skip the costly matrix creation process by using `transform`.

**Notes**

• In performance tests, `applyMatrix` is noticeably faster than `transform` for very large arrays, however this is only true if you are applying the same transformation to all points.

• If the input arrays for `points` or `pos` is Nx4, the last column is ignored.

**psychopy.tools.mathtools.scale** (*sf, points, out=None, dtype=None*)

Scale points by a factor.

This is useful for converting points between units, and to stretch or compress points along a given axis. Scaling can be uniform which the same factor is applied along all axes, or anisotropic along specific axes.

**Parameters**

• **sf** (*array_like or float*) – Scaling factor. If scalar, all points will be scaled uniformly by that factor. If a vector, scaling will be anisotropic along an axis.

• **points** (*array_like*) – Point(s) [x, y, z] to scale.
• **out** (*ndarray*, *optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.

• **dtype** (*dtype or str, optional*) – Data type for computations can either be ‘float32’ or ‘float64’. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, ‘float64’ is used by default.

Returns Scaled points.

Return type *ndarray*

---

**Examples**

Apply uniform scaling to points, here we scale to convert points in centimeters to meters:

```python
CM_TO_METERS = 1.0 / 100.0
pointsCM = [[1, 2, 3], [4, 5, 6], [-1, 1, 0]]
pointsM = scale(CM_TO_METERS, pointsCM)
```

Anisotropic scaling along the X and Y axis:

```python
pointsM = scale((SCALE_FACTOR_X, SCALE_FACTOR_Y), pointsCM)
```

Scale only on the X axis:

```python
pointsM = scale((SCALE_FACTOR_X,), pointsCM)
```

Apply scaling on the Z axis only:

```python
pointsM = scale((1.0, 1.0, SCALE_FACTOR_Z), pointsCM)
```

---

**psychopy.tools.mathtools.lensCorrection** (*xys*, *coefK=(1.0)*, *distCenter=(0.0, 0.0)*, *out=None*, *dtype=None*)

Lens correction (or distortion) using the division model with even polynomial terms.

Calculate new vertex positions or texture coordinates to apply radial warping, such as ‘pincushion’ and ‘barrel’ distortion. This is to compensate for optical distortion introduced by lenses placed in the optical path of the viewer and the display (such as in an HMD).

Parameters

• **xys** (*array_like*) – Nx2 list of vertex positions or texture coordinates to distort. Works correctly only if input values range between -1.0 and 1.0.

• **coefK** (*array_like or float*) – Distortion coefficients K_n. Specifying multiple values will add more polynomial terms to the distortion formula. Positive values will produce ‘barrel’ distortion, whereas negative will produce ‘pincushion’ distortion. In most cases, two or three coefficients are adequate, depending on the degree of distortion.

• **distCenter** (*array_like, optional*) – X and Y coordinate of the distortion center (eg. (0.2, -0.4)).

• **out** (*ndarray, optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.

• **dtype** (*dtype or str, optional*) – Data type for computations can either be ‘float32’ or ‘float64’. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, ‘float64’ is used by default.

Returns Array of distorted vertices.
Return type  ndarray

Notes

• At this time tangential distortion (i.e. due to a slant in the display) cannot be corrected for.

References

Examples

Creating a lens correction mesh with barrel distortion (eg. for HMDs):

```python
vertices, textureCoords, normals, faces = gltools.createMeshGrid(
    subdiv=11, tessMode='center')

# recompute vertex positions
vertices[:, :2] = mt.lensCorrection(vertices[:, :2], coefK=(5., 5.))
```

`psychopy.tools.mathtools.zeroFix(a, inplace=False, threshold=None)`

Fix zeros in an array.

This function truncates very small numbers in an array to zero and removes any negative zeros.

Parameters

• `a (ndarray)` – Input array, must be a Numpy array.

• `inplace (bool)` – Fix an array inplace. If True, the input array will be modified, otherwise a new array will be returned with same dtype and shape with the fixed values.

• `threshold (float or None)` – Threshold for truncation. If None, the machine epsilon value for the input array dtype will be used. You can specify a custom threshold as a float.

Returns  Output array with zeros fixed.

Return type  ndarray

8.6.7 `psychopy.tools.monitorunittools`

Functions and classes related to unit conversion respective to a particular monitor

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>convertToPix(vertices, pos, units, win)</code></td>
<td>Takes vertices and position, combines and converts to pixels from any unit</td>
</tr>
<tr>
<td><code>cm2deg(cm, monitor[, correctFlat])</code></td>
<td>Convert size in cm to size in degrees for a given Monitor object</td>
</tr>
<tr>
<td><code>cm2pix(cm, monitor)</code></td>
<td>Convert size in cm to size in pixels for a given Monitor object</td>
</tr>
<tr>
<td><code>deg2cm(degrees, monitor[, correctFlat])</code></td>
<td>Convert size in degrees to size in cm for a given Monitor object</td>
</tr>
<tr>
<td><code>deg2pix(degrees, monitor[, correctFlat])</code></td>
<td>Convert size in degrees to size in pixels for a given Monitor object</td>
</tr>
<tr>
<td><code>pix2cm(pixels, monitor)</code></td>
<td>Convert size in pixels to size in cm for a given Monitor object</td>
</tr>
</tbody>
</table>

continues on next page
Table 31 – continued from previous page

<table>
<thead>
<tr>
<th>Function details</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>pix2deg</strong>(pixels, monitor[, correctFlat])</td>
</tr>
</tbody>
</table>

Function details

**psychopy.tools.monitorunittools.cm2deg** *(cm, monitor, correctFlat=False)*
Convert size in cm to size in degrees for a given Monitor object.

**psychopy.tools.monitorunittools.cm2pix** *(cm, monitor)*
Convert size in cm to size in pixels for a given Monitor object.

**psychopy.tools.monitorunittools.deg2cm** *(degrees, monitor, correctFlat=False)*
Convert size in degrees to size in pixels for a given Monitor object.

- If `correctFlat == False` then the screen will be treated as if all points are equal distance from the eye. This means that each “degree” will be the same size irrespective of its position.
- If `correctFlat == True` then the `degrees` argument must be an Nx2 matrix for X and Y values (the two cannot be calculated separately in this case).
- With `correctFlat == True` the positions may look strange because more eccentric vertices will be spaced further apart.

**psychopy.tools.monitorunittools.deg2pix** *(degrees, monitor, correctFlat=False)*
Convert size in degrees to size in pixels for a given Monitor object.

**psychopy.tools.monitorunittools.pix2cm** *(pixels, monitor)*
Convert size in pixels to size in cm for a given Monitor object.

**psychopy.tools.monitorunittools.pix2deg** *(pixels, monitor, correctFlat=False)*
Convert size in pixels to size in degrees for a given Monitor object.

8.6.8 **psychopy.tools.plottools**

Functions and classes related to plotting

**psychopy.tools.plottools.plotFrameIntervals** *(intervals)*
Plot a histogram of the frame intervals.
Where `intervals` is either a filename to a file, saved by Window.saveFrameIntervals, or simply a list (or array) of frame intervals.
8.6.9 `psychopy.tools.rifttools`

Various tools for working with the `Rift` class. The documentation for classes in on this page originate from PsychXR and may make references to functions and objects not included with PsychoPy.

**Overview**

**Classes**

These classes are included with PsychXR to use with the LibOVR interface. They can be accessed from this module to avoid needing to explicitly import PsychXR. If PsychXR is not available on the system, these classes will have values `None`.

- `LibOVRPose`
- `LibOVRPoseState`
- `LibOVRHapticsBuffer`
- `LibOVRBounds`

**Functions**

These functions can be called without first starting a VR session (initializing a `Rift` instance) to check if the drivers/services are running on this computer or if an HMD is connected.

- `isHmdConnected([timeout])` Check if an HMD is connected.
- `isOculusServiceRunning([timeout])` Check if the Oculus(tm) service is currently running.

**Details**

- `psychopy.tools.rifttools.LibOVRPose`
- `psychopy.tools.rifttools.LibOVRPoseState`
- `psychopy.tools.rifttools.LibOVRBounds`
- `psychopy.tools.rifttools.LibOVRHapticsBuffer`
- `psychopy.tools.rifttools.isHmdConnected(timeout=0)` Check if an HMD is connected.
  - **Parameters** `timeout` *(int)* – Timeout in milliseconds.
  - **Returns** `True` if an HMD is connected.
  - **Return type** `bool`
- `psychopy.tools.rifttools.isOculusServiceRunning(timeout=0)` Check if the Oculus(tm) service is currently running.
  - **Parameters** `timeout` *(int)* – Timeout in milliseconds.
  - **Returns** `True` if the service is loaded and running.
  - **Return type** `bool`
8.6.10 psychopy.tools.typetools

Functions and classes related to variable type conversion

psychopy.tools.typetools.float_uint8(inarray)
Converts arrays, lists, tuples and floats ranging -1:1 into an array of Uint8s ranging 0:255

```python
>>> float_uint8(-1)
0
>>> float_uint8(0)
128
```

psychopy.tools.typetools.uint8_float(inarray)
Converts arrays, lists, tuples and UINTs ranging 0:255 into an array of floats ranging -1:1

```python
>>> uint8_float(0)
-1.0
>>> uint8_float(128)
0.0
```

psychopy.tools.typetools.float_uint16(inarray)
Converts arrays, lists, tuples and floats ranging -1:1 into an array of Uint16s ranging 0:2^16

```python
>>> float_uint16(-1)
0
>>> float_uint16(0)
32768
```

8.6.11 psychopy.tools.unittools

Functions and classes related to unit conversion

psychopy.tools.unittools.radians(x, /, out=None, *, where=True, casting='same_kind', order='K',
dtype=None, subok=True, signature=None, extobj=None)

Convert angles from degrees to radians.

Parameters

- `x (array_like)` – Input array in degrees.
- `out (ndarray, None, or tuple of ndarray and None, optional)` – A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.
- `where (array_like, optional)` – This condition is broadcast over the input. At locations where the condition is True, the out array will be set to the ufunc result. Elsewhere, the out array will retain its original value. Note that if an uninitialized out array is created via the default out= None, locations within it where the condition is False will remain uninitialized.
- **kwargs – For other keyword-only arguments, see the ufunc docs.

Returns `y` – The corresponding radian values. This is a scalar if x is a scalar.

Return type `ndarray`

See also:

deg2rad equivalent function
Examples

Convert a degree array to radians

```python
>>> deg = np.arange(12.) * 30.
>>> np.radians(deg)
array([ 0. , 0.52359878, 1.04719755, 1.57079633, 2.0943951 ,
      2.61799388, 3.14159265, 3.66519143, 4.1887902 , 4.71238898,
      5.23598776, 5.75958653])
```

```python
>>> out = np.zeros((deg.shape))
>>> ret = np.radians(deg, out)
>>> ret is out
True
```

```
psychopy.tools.unittools.degrees(x, /, out=None, *, where=True, casting='same_kind', order='K',
dtype=None, subok=True, signature, extobj)
```

Convert angles from radians to degrees.

**Parameters**

- `x (array_like)` – Input array in radians.
- `out (ndarray, None, or tuple of ndarray and None, optional)` – A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.
- `where (array_like, optional)` – This condition is broadcast over the input. At locations where the condition is True, the `out` array will be set to the ufunc result. Elsewhere, the `out` array will retain its original value. Note that if an uninitialized `out` array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.
- `**kwargs` – For other keyword-only arguments, see the ufunc docs.

**Returns** `y` – The corresponding degree values; if `out` was supplied this is a reference to it. This is a scalar if `x` is a scalar.

**Return type** ndarray of floats

**See also:**

`rad2deg` equivalent function

Examples

Convert a radian array to degrees

```python
>>> rad = np.arange(12.)*np.pi/6
>>> np.degrees(rad)
array([ 0. , 30. , 60. , 90. , 120. , 150. , 180. , 210. , 240. ,
       270. , 300. , 330. ])
```

```python
>>> out = np.zeros((rad.shape))
>>> r = np.degrees(rad, out)
>>> np.all(r == out)
True
```
### 8.6.12 psychopy.tools.viewtools

Math functions for working with view transformations and performing visibility testing (see also `mathtools`).
Tools for working with view projections for 2- and 3-D rendering.

#### Overview

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>visualAngle(size, distance[, degrees, out, ...])</code></td>
<td>Get the visual angle for an object of <code>size</code> at <code>distance</code>. Object is assumed to be fronto-parallel with the viewer.</td>
</tr>
<tr>
<td><code>computeFrustum(scrWidth, scrAspect, scrDist)</code></td>
<td>Calculate frustum parameters.</td>
</tr>
<tr>
<td><code>computeFrustumFOV(scrFOV, scrAspect, scrDist)</code></td>
<td>Compute a frustum for a given field-of-view (FOV).</td>
</tr>
<tr>
<td><code>projectFrustum(frustum, dist[,dtype])</code></td>
<td>Project a frustum on a fronto-parallel plane and get the width and height of the required drawing area.</td>
</tr>
<tr>
<td><code>projectFrustumToPlane(frustum, planeOrig[, ...])</code></td>
<td>Project a frustum on a fronto-parallel plane and get the coordinates of the corners in physical space.</td>
</tr>
<tr>
<td><code>generalizedPerspectiveProjection(...[, ...])</code></td>
<td>Generalized derivation of projection and view matrices based on the physical configuration of the display system.</td>
</tr>
<tr>
<td><code>orthoProjectionMatrix(left,right,bottom,top)</code></td>
<td>Compute an orthographic projection matrix with provided frustum parameters.</td>
</tr>
<tr>
<td><code>perspectiveProjectionMatrix(left,right,...)</code></td>
<td>Compute an perspective projection matrix with provided frustum parameters.</td>
</tr>
<tr>
<td><code>lookAt(eyePos, centerPos[, upVec, out, dtype])</code></td>
<td>Create a transformation matrix to orient a view towards some point.</td>
</tr>
<tr>
<td><code>pointToNdc(wcsPos, viewMatrix, projectionMatrix)</code></td>
<td>Map the position of a point in world space to normalized device coordinates/space.</td>
</tr>
<tr>
<td><code>cursorToRay(cursorX, cursorY, winSize, ...)</code></td>
<td>Convert a 2D mouse coordinate to a 3D ray.</td>
</tr>
<tr>
<td><code>visible(points, mvp[, mode, dtype])</code></td>
<td>Test if points are visible.</td>
</tr>
<tr>
<td><code>visibleBBox(extents, mvp[, dtype])</code></td>
<td>Check if a bounding box is visible.</td>
</tr>
</tbody>
</table>

#### Details

**psychopy.tools.viewtools.visualAngle** *(size, distance, degrees= True, out= None, dtype= None)*

Get the visual angle for an object of `size` at `distance`. Object is assumed to be fronto-parallel with the viewer.

This function supports vector inputs. Values for `size` and `distance` can be arrays or single values. If both inputs are arrays, they must have the same size.

**Parameters**

- `size (float or array_like)` – Size of the object in meters.
- `distance (float or array_like)` – Distance to the object in meters.
- `degrees (bool)` – Return result in degrees, if `False` result will be in radians.
- `out (ndarray, optional)` – Optional output array. Must be same shape and dtype as the expected output if `out` was not specified.
- `dtype (dtype or str, optional)` – Data type for arrays, can either be ‘float32’ or ‘float64’. If `None` is specified, the data type is inferred by `out`. If `out` is not provided, the default is ‘float64’.

**Returns** Visual angle.

**Return type** float
Examples

Calculating the visual angle (vertical FOV) of a monitor screen:

```python
monDist = 0.5  # monitor distance, 50cm
monHeight = 0.45  # monitor height, 45cm
vertFOV = visualAngle(monHeight, monDist)
```

Compute visual angle at multiple distances for objects with the same size:

```python
va = visualAngle(0.20, [1.0, 2.0, 3.0])  # returns
# [11.42118627  5.72481045  3.81830487]
```

`psychopy.tools.viewtools.computeFrustum(scrWidth, scrAspect, scrDist, convergeOffset=0.0, eyeOffset=0.0, nearClip=0.01, farClip=100.0, dtype=None)`

Calculate frustum parameters. If an eye offset is provided, an asymmetric frustum is returned which can be used for stereoscopic rendering.

**Parameters**

- `scrWidth (float)` – The display's width in meters.
- `scrAspect (float)` – Aspect ratio of the display (width / height).
- `scrDist (float)` – Distance to the screen from the view in meters. Measured from the center of their eyes.
- `convergeOffset (float)` – Offset of the convergence plane from the screen. Objects falling on this plane will have zero disparity. For best results, the convergence plane should be set to the same distance as the screen (0.0 by default).
- `eyeOffset (float)` – Half the inter-ocular separation (i.e. the horizontal distance between the nose and center of the pupil) in meters. If eyeOffset is 0.0, a symmetric frustum is returned.
- `nearClip (float)` – Distance to the near clipping plane in meters from the viewer. Should be at least less than `scrDist`.
- `farClip (float)` – Distance to the far clipping plane from the viewer in meters. Must be >nearClip.
- `dtype (dtype or str, optional)` – Data type for arrays, can either be 'float32' or 'float64'. If `None` is specified, the data type is inferred by `out`. If `out` is not provided, the default is 'float64'.

**Returns** Array of frustum parameters. Can be directly passed to `glFrustum` (e.g. `glFrustum(*f)`).

**Return type** `ndarray`
Notes

• The viewpoint must be transformed for objects to appear correctly. Offsets in the X-direction must be applied +/- eyeOffset to account for inter-ocular separation. A transformation in the Z-direction must be applied to account for screen distance. These offsets MUST be applied to the GL_MODELVIEW matrix, not the GL_PROJECTION matrix! Doing so may break lighting calculations.

Examples

Creating a frustum and setting a window’s projection matrix:

```python
scrWidth = 0.5  # screen width in meters
scrAspect = win.size[0] / win.size[1]
scrDist = win.scrDistCM * 100.0  # monitor setting, can be anything
frustum = viewtools.computeFrustum(scrWidth, scrAspect, scrDist)
```

Accessing frustum parameters:

```python
left, right, bottom, top, nearVal, farVal = frustum
# ... or ...
left = frustum.left
```

Off-axis frustums for stereo rendering:

```python
# compute view matrix for each eye, these value usually don’t change
eyeOffset = (-0.035, 0.035)  # +/- IOD / 2.0
scrDist = 0.50  # 50cm
scrWidth = 0.53  # 53cm
scrAspect = 1.778
leftFrustum = viewtools.computeFrustum(
    scrWidth, scrAspect, scrDist, eyeOffset[0])
rightFrustum = viewtools.computeFrustum(
    scrWidth, scrAspect, scrDist, eyeOffset[1])
# make sure your view matrix accounts for the screen distance and eye
# offsets!
```

Using computed view frustums with a window:

```python
win.projectionMatrix = viewtools.perspectiveProjectionMatrix(frustum)
# generate a view matrix looking ahead with correct viewing distance,
# origin is at the center of the screen. Assumes eye is centered with
# the screen.
eyePos = [0.0, 0.0, scrDist]
screenPos = [0.0, 0.0, 0.0]  # look at screen center
eyeUp = [0.0, 1.0, 0.0]
win.viewMatrix = viewtools.lookAt(eyePos, screenPos, eyeUp)
win.applyViewTransform()  # call before drawing
```

Compute a frustum for a given field-of-view (FOV).

```python
psychopy.tools.computeFrustumFOV(scrFOV, scrAspect, scrDist, convergeOffset=0.0,
    eyeOffset=0.0, nearClip=0.01, farClip=100.0, dtype=None)
```

Similar to `computeFrustum`, but computes a frustum based on FOV rather than screen dimensions.

Parameters:

- `scrFOV` (float) – Vertical FOV in degrees (fovY).
• `scrAspect (float)` – Aspect between the horizontal and vertical FOV (i.e. fovX / fovY).

• `scrDist (float)` – Distance to the screen from the view in meters. Measured from the center of the viewer’s eye(s).

• `convergeOffset (float)` – Offset of the convergence plane from the screen. Objects falling on this plane will have zero disparity. For best results, the convergence plane should be set to the same distance as the screen (0.0 by default).

• `eyeOffset (float)` – Half the inter-ocular separation (i.e. the horizontal distance between the nose and center of the pupil) in meters. If eyeOffset is 0.0, a symmetric frustum is returned.

• `nearClip (float)` – Distance to the near clipping plane in meters from the viewer. Should be at least less than `scrDist`. Never should be 0.

• `farClip (float)` – Distance to the far clipping plane from the viewer in meters. Must be >`nearClip`.

• `dtype (dtype or str, optional)` – Data type for arrays, can either be ‘float32’ or ‘float64’. If `None` is specified, the data type is inferred by `out`. If `out` is not provided, the default is ‘float64’.

**Examples**

Equivalent to `gluPerspective`:

```python
frustum = computeFrustumFOV(45.0, 1.0, 0.5)
projectionMatrix = perspectiveProjectionMatrix(frustum)
```

`psychopy.tools.viewtools.projectFrustum (frustum, dist, dtype=None)`

Project a frustum on a fronto-parallel plane and get the width and height of the required drawing area.

This function can be used to determine the size of the drawing area required for a given frustum on a screen. This is useful for cases where the observer is viewing the screen through a physical aperture that limits the FOV to a sub-region of the display. You must convert the size in meters to units of your screen and apply any offsets.

**Parameters**

• `frustum (array_like)` – Frustum parameters (left, right, bottom, top, near, far), you can exclude `far` since it is not used in this calculation. However, the function will still succeed if given.

• `dist (float)` – Distance to project points to in meters.

• `dtype (dtype or str, optional)` – Data type for arrays, can either be ‘float32’ or ‘float64’. If `None` is specified, the data type is inferred by `out`. If `out` is not provided, the default is ‘float64’.

**Returns** Width and height (w, h) of the area intersected by the given frustum at `dist`.

**Return type** `ndarray`
Examples

Compute the viewport required to draw in the area where the frustum intersects the screen:

```python
# needed information
scrWidthM = 0.52
scrDistM = 0.72
scrWidthPIX = 1920
scrHeightPIX = 1080
scrAspect = scrWidthPIX / float(scrHeightPIX)
pixPerMeter = scrWidthPIX / scrWidthM

# Compute a frustum for 20 degree vertical FOV at distance of the # screen.
frustum = computeFrustumFOV(20., scrAspect, scrDistM)

# get the dimensions of the frustum
w, h = projectFrustum(frustum, scrDistM) * pixPerMeter

# get the origin of the viewport, relative to center of screen.
x = (scrWidthPIX - w) / 2.
y = (scrHeightPIX - h) / 2.

# if there is an eye offset ...
# x = (scrWidthPIX - w + eyeOffsetM * pixPerMeter) / 2.

# viewport rectangle
rect = np.asarray((x, y, w, h), dtype=int)
```

You can then set the viewport/scissor rectangle of the buffer to restrict drawing to `rect`.

psychopy.tools.viewtools.projectFrustumToPlane(frustum, planeOrig, dtype=None)

Project a frustum on a fronto-parallel plane and get the coordinates of the corners in physical space.

**Parameters**

- `frustum (array_like)` – Frustum parameters (left, right, bottom, top, near, far), you can exclude `far` since it is not used in this calculation. However, the function will still succeed if given.

- `planeOrig (float)` – Distance of plane to project points on in meters.

- `dtype (dtype or str, optional)` – Data type for arrays, can either be `float32` or `float64`. If `None` is specified, the data type is inferred by `out`. If `out` is not provided, the default is `float64`.

**Returns** 4x3 array of coordinates in the physical reference frame with origin at the eye.

**Return type** ndarray

psychopy.tools.viewtools.generalizedPerspectiveProjection(posBottomLeft, posBottomRight, posTopLeft, eyePos, nearClip=0.01, farClip=100.0, dtype=None)

Generalized derivation of projection and view matrices based on the physical configuration of the display system.

This implementation is based on Robert Kooima’s ‘Generalized Perspective Projection’ method\(^1\).

**Parameters**

---

• **posBottomLeft** *(list of float or ndarray)* – Bottom-left 3D coordinate of the screen in meters.

• **posBottomRight** *(list of float or ndarray)* – Bottom-right 3D coordinate of the screen in meters.

• **posTopLeft** *(list of float or ndarray)* – Top-left 3D coordinate of the screen in meters.

• **eyePos** *(list of float or ndarray)* – Coordinate of the eye in meters.

• **nearClip** *(float)* – Near clipping plane distance from viewer in meters.

• **farClip** *(float)* – Far clipping plane distance from viewer in meters.

• **dtype** *(dtype or str, optional)* – Data type for arrays, can either be ‘float32’ or ‘float64’. If *None* is specified, the data type is inferred by *out*. If *out* is not provided, the default is ‘float64’.

**Returns**  The 4x4 projection and view matrix.

**Return type**  tuple

**See also:**

*computeFrustum* Compute frustum parameters.

**Notes**

• The resulting projection frustums are off-axis relative to the center of the display.

• The returned matrices are row-major. Values are floats with 32-bits of precision stored as a contiguous (C-order) array.

**References**

**Examples**

Computing a projection and view matrices for a window:

```python
projMatrix, viewMatrix = viewtools.generalizedPerspectiveProjection(
    posBottomLeft, posBottomRight, posTopLeft, eyePos)
```

# set the window matrices
win.projectionMatrix = projMatrix
win.viewMatrix = viewMatrix

# before rendering
win.applyEyeTransform()

Stereo-pair rendering example from Kooima (2009):

```python
# configuration of screen and eyes
posBottomLeft = [-1.5, -0.75, -18.0]
posBottomRight = [1.5, -0.75, -18.0]
posTopLeft = [-1.5, 0.75, -18.0]
posLeftEye = [-1.25, 0.0, 0.0]
posRightEye = [1.25, 0.0, 0.0]

# create projection and view matrices
leftProjMatrix, leftViewMatrix = generalizedPerspectiveProjection(
```

(continues on next page)
posBottomLeft, posBottomRight, posTopLeft, posLeftEye)
rightProjMatrix, rightViewMatrix = generalizedPerspectiveProjection(
posBottomLeft, posBottomRight, posTopLeft, posRightEye)

psychopy.tools.viewtools.orthoProjectionMatrix(left, right, bottom, top, nearClip=0.01, farClip=100.0, out=None, dtype=None)

Compute an orthographic projection matrix with provided frustum parameters.

Parameters

- **left** *(float)* – Left clipping plane coordinate.
- **right** *(float)* – Right clipping plane coordinate.
- **bottom** *(float)* – Bottom clipping plane coordinate.
- **top** *(float)* – Top clipping plane coordinate.
- **nearClip** *(float)* – Near clipping plane distance from viewer.
- **farClip** *(float)* – Far clipping plane distance from viewer.
- **out** *(ndarray, optional)* – Optional output array. Must be same shape and dtype as the expected output if out was not specified.
- **dtype** *(dtype or str, optional)* – Data type for arrays, can either be ‘float32’ or ‘float64’. If None is specified, the data type is inferred by out. If out is not provided, the default is ‘float64’.

Returns 4x4 projection matrix

Return type ndarray

See also:

**perspectiveProjectionMatrix** Compute a perspective projection matrix.

Notes

- The returned matrix is row-major. Values are floats with 32-bits of precision stored as a contiguous (C-order) array.

psychopy.tools.viewtools.perspectiveProjectionMatrix(left, right, bottom, top, nearClip=0.01, farClip=100.0, out=None, dtype=None)

Compute an perspective projection matrix with provided frustum parameters. The frustum can be asymmetric.

Parameters

- **left** *(float)* – Left clipping plane coordinate.
- **right** *(float)* – Right clipping plane coordinate.
- **bottom** *(float)* – Bottom clipping plane coordinate.
- **top** *(float)* – Top clipping plane coordinate.
- **nearClip** *(float)* – Near clipping plane distance from viewer.
- **farClip** *(float)* – Far clipping plane distance from viewer.
• **out** (*ndarray*, *optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.

• **dtype** (*dtype* or *str*, *optional*) – Data type for arrays, can either be ‘float32’ or ‘float64’. If *None* is specified, the data type is inferred by *out*. If *out* is not provided, the default is ‘float64’.

**Returns** 4x4 projection matrix

**Return type** *ndarray*

**See also:**

*orthoProjectionMatrix* Compute a orthographic projection matrix.

**Notes**

• The returned matrix is row-major. Values are floats with 32-bits of precision stored as a contiguous (C-order) array.

`psychopy.tools.viewtools.lookAt(eyePos, centerPos, upVec=(0.0, 1.0, 0.0), out=None, dtype=None)`  
Create a transformation matrix to orient a view towards some point. Based on the same algorithm as ‘gluLookAt’. This does not generate a projection matrix, but rather the matrix to transform the observer’s view in the scene.

For more information see: https://www.khronos.org/registry/OpenGL-Refpages/gl2.1/xhtml/gluLookAt.xml

**Parameters**

• **eyePos** (*list of float* or *ndarray*) – Eye position in the scene.

• **centerPos** (*list of float* or *ndarray*) – Position of the object center in the scene.

• **upVec** (*list of float* or *ndarray*, *optional*) – Vector defining the up vector. Default is +Y is up.

• **out** (*ndarray*, *optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.

• **dtype** (*dtype* or *str*, *optional*) – Data type for arrays, can either be ‘float32’ or ‘float64’. If *None* is specified, the data type is inferred by *out*. If *out* is not provided, the default is ‘float64’.

**Returns** 4x4 view matrix

**Return type** *ndarray*

**Notes**

• The returned matrix is row-major. Values are floats with 32-bits of precision stored as a contiguous (C-order) array.

`psychopy.tools.viewtools.pointToNdc(wcsPos, viewMatrix, projectionMatrix, out=None, dtype=None)`  
Map the position of a point in world space to normalized device coordinates/space.

**Parameters**
• **wcsPos***(tuple, list or ndarray)* – Nx3 position vector(s) (xyz) in world space coordinates.

• **viewMatrix***(ndarray)* – 4x4 view matrix.

• **projectionMatrix***(ndarray)* – 4x4 projection matrix.

• **out***(ndarray, optional)* – Optional output array. Must be same shape and dtype as the expected output if out was not specified.

• **dtype***(dtype or str, optional)* – Data type for arrays, can either be ‘float32’ or ‘float64’. If None is specified, the data type is inferred by out. If out is not provided, the default is ‘float64’.

**Returns**

3x1 vector of normalized device coordinates with type ‘float32’

**Return type**

ndarray

**Notes**

• The point is not visible, falling outside of the viewing frustum, if the returned coordinates fall outside of -1 and 1 along any dimension.

• In the rare instance the point falls directly on the eye in world space where the frustum converges to a point (singularity), the divisor will be zero during perspective division. To avoid this, the divisor is ‘bumped’ to 1e-5.

• This function assumes the display area is rectilinear. Any distortion or warping applied in normalized device or viewport space is not considered.

**Examples**

Determine if a point is visible:

```python
point = (0.0, 0.0, 10.0)  # behind the observer
ndc = pointToNdc(point, win.viewMatrix, win.projectionMatrix)
isVisible = not np.any((ndc > 1.0) | (ndc < -1.0))
```

Convert NDC to viewport (or pixel) coordinates:

```python
scrRes = (1920, 1200)
point = (0.0, 0.0, -5.0)  # forward -5.0 from eye
x, y, z = pointToNdc(point, win.viewMatrix, win.projectionMatrix)
pixelX = ((x + 1.0) / 2.0) * scrRes[0]
pixelY = ((y + 1.0) / 2.0) * scrRes[1]
# object at point will appear at (pixelX, pixelY)
```

psychopy.tools.viewtools.cursorToRay*(cursorX, cursorY, winSize, viewport, projectionMatrix, normalize=True, out=None, dtype=None)*

Convert a 2D mouse coordinate to a 3D ray.

Takes a 2D window/mouse coordinate and transforms it to a 3D direction vector from the viewpoint in eye space (vector origin is [0, 0, 0]). The center of the screen projects to vector [0, 0, -1].

**Parameters**

- **cursorX***(float or int)* – Window coordinates. These need to be scaled if you are using a framebuffer that does not have 1:1 pixel mapping (i.e. retina display).
- **cursorY** *(float or int)* – Window coordinates. These need to be scaled if you are using a framebuffer that does not have 1:1 pixel mapping (i.e. retina display).
- **winSize** *(array_like)* – Size of the window client area [w, h].
- **viewport** *(array_like)* – Viewport rectangle [x, y, w, h] being used.
- **projectionMatrix** *(ndarray)* – 4x4 projection matrix being used.
- **normalize** *(bool)* – Normalize the resulting vector.
- **out** *(ndarray, optional)* – Optional output array. Must be same shape and dtype as the expected output if *out* was not specified.
- **dtype** *(dtype or str, optional)* – Data type for arrays, can either be ‘float32’ or ‘float64’. If *None* is specified, the data type is inferred by *out*. If *out* is not provided, the default is ‘float64’.

**Returns**  Direction vector (x, y, z).

**Return type**  ndarray

**Examples**

Place a 3D stim at the mouse location 5.0 scene units (meters) away:

```python
# define camera
camera = RigidBodyPose((-3.0, 5.0, 3.5))
camera.alignTo((0, 0, 0))

# in the render loop
dist = 5.0
mouseRay = vt.cursorToRay(x, y, win.size, win.viewport, win.projectionMatrix)
mouseRay *= dist  # scale the vector

# set the sphere position by transforming vector to world space
sphere.thePose.pos = camera.transform(mouseRay)
```

**psychopy.tools.viewtools.visible** *(points, mvp, mode='discrete', dtype=None)*

Test if points are visible.

This function is useful for visibility culling, where objects are only drawn if a portion of them are visible. This test can avoid costly drawing calls and OpenGL state changes if the object is not visible.

**Parameters**

- **points** *(array_like)* – Point(s) or bounding box to test. Input array must be N×3 or N×4, where each row is a point. It is recommended that the input be N×4 since the w component will be appended if the input is N×3 which adds overhead.
- **mvp** *(array_like)* – 4x4 MVP matrix.
- **mode** *(str)* – Test mode. If ‘discrete’, rows of *points* are treated as individual points. This function will return an array of boolean values with length equal to the number of rows in *points*, where the value at each index corresponds to the visibility test results for points at the matching row index of *points*. If ‘group’ a single boolean value is returned, which is *False* if all points fall to one side of the frustum.
• **dtype** *(dtype or str, optional)* – Data type for arrays, can either be ‘float32’ or ‘float64’. If *None* is specified, the data type is inferred by *out*. If *out* is not provided, the default is ‘float64’.

**Returns** Test results. The type returned depends on *mode*.

**Return type** bool or ndarray

**Examples**

Visibility culling, only a draw line connecting two points if visible:

```python
linePoints = [[-1.0, -1.0, -1.0, 1.0],
              [ 1.0,  1.0,  1.0, 1.0]]
mvp = np.matmul(win.projectionMatrix, win.viewMatrix)
if visible(linePoints, mvp, mode='group'):
    # drawing commands here ...
```

**psychopy.tools.viewtools.visibleBBox** *(extents, mvp, dtype=None)*

Check if a bounding box is visible.

This function checks if a bounding box intersects a frustum defined by the current projection matrix, after being transformed by the model-view matrix.

**Parameters**

- **extents** *(array_like)* – Bounding box minimum and maximum extents as a 2x3 array. The first row if the minimum extents along each axis, and the second row the maximum extents (eg. [[minX, minY, minZ], [maxX, maxY, maxZ]]).
- **mvp** *(array_like)* – 4x4 MVP matrix.
- **dtype** *(dtype or str, optional)* – Data type for arrays, can either be ‘float32’ or ‘float64’. If *None* is specified, the data type is inferred by *out*. If *out* is not provided, the default is ‘float64’.

**Returns** Visibility test results.

**Return type** ndarray or bool

---

**8.7 psychopy.data - functions for storing/saving/analysing data**

**Contents:**

- **ExperimentHandler** - to combine multiple loops in one study
- **TrialHandler** - basic predefined trial matrix
- **TrialHandler2** - similar to TrialHandler but with ability to update mid-run
- **TrialHandlerExt** - similar to TrialHandler but with ability to run oddball designs
- **StairHandler** - for basic up-down (fixed step) staircases
- **QuestHandler** - for traditional QUEST algorithm
- **QuestPlusHandler** - for the updated QUEST+ algorithm (Watson, 2017)
- **PsiHandler** - the Psi staircase of Kontsevich & Tyler (1999)
• MultiStairHandler - a wrapper to combine interleaved staircases of any sort

Utility functions:
• importConditions() - to load a list of dicts from a csv/excel file
• functionFromStaircase() - to convert a staircase into its psychometric function
• bootStraps() - generate a set of bootstrap resamples from a dataset

Curve Fitting:
• FitWeibull
• FitLogistic
• FitNakaRushton
• FitCumNormal

8.7.1 ExperimentHandler

class psychopy.data.ExperimentHandler(name='', version='', extraInfo=None, runtimeInfo=None, originPath=None, savePickle=True, saveWideText=True, dataFileName='', autoLog=True, appendFiles=False)

A container class for keeping track of multiple loops/handlers
Useful for generating a single data file from an experiment with many different loops (e.g. interleaved staircases or loops within loops

Usage exp = data.ExperimentHandler(name="Face Preference",version='0.1.0')

Parameters

- name [a string or unicode] As a useful identifier later
- version [usually a string (e.g. ‘1.1.0’)] To keep track of which version of the experiment was run
- extraInfo [a dictionary] Containing useful information about this run (e.g. {'participant':'jwp', 'gender': 'm', 'orientation': 90} )
- runtimeInfo [psychopy.info.RunTimeInfo] Containing information about the system as detected at runtime
- originPath [string or unicode] The path and filename of the originating script/experiment If not provided this will be determined as the path of the calling script.
- dataFileName [string] This is defined in advance and the file will be saved at any point that the handler is removed or discarded (unless .abort() had been called in advance). The handler will attempt to populate the file even in the event of a (not too serious) crash!

- savePickle : True (default) or False
- saveWideText : True (default) or False
- autoLog : True (default) or False

_getAllParamNames ()
Returns the attribute names of loop parameters (trialN etc) that the current set of loops contain, ready to build a wide-format data file.
_getExtraInfo()
Get the names and vals from the extraInfo dict (if it exists)

_getLoopInfo(loop)
Returns the attribute names and values for the current trial of a particular loop. Does not return data inputs
from the subject, only info relating to the trial execution.

abort()
Inform the ExperimentHandler that the run was aborted.

Experiment handler will attempt automatically to save data (even in the event of a crash if possible). So if
you quit your script early you may want to tell the Handler not to save out the data files for this run. This is
the method that allows you to do that.

addData(name, value)
Add the data with a given name to the current experiment.

Typically the user does not need to use this function; if you added your data to the loop and had already added
the loop to the experiment then the loop will automatically inform the experiment that it has received data.

Multiple data name/value pairs can be added to any given entry of the data file and is considered part of the
same entry until the nextEntry() call is made.

e.g.:

```python
# add some data for this trial
exp.addData('resp.rt', 0.8)
exp.addData('resp.key', 'k')
# end of trial - move to next line in data output
exp.nextEntry()
```

addLoop(loopHandler)
Add a loop such as a TrialHandler or StairHandler Data from this loop will be included in the
resulting data files.

close()

getAllEntries()
Fetches a copy of all the entries including a final (orphan) entry if that exists. This allows entries to be saved
even if nextEntry() is not yet called.

Returns copy (not pointer) to entries

loopEnded(loopHandler)
Informs the experiment handler that the loop is finished and not to include its values in further entries of the
experiment.

This method is called by the loop itself if it ends its iterations, so is not typically needed by the user.

nextEntry()
Calling nextEntry indicates to the ExperimentHandler that the current trial has ended and so further addData()
calls correspond to the next trial.

saveAsPickle(fileName, fileCollisionMethod='rename')
Basically just saves a copy of self (with data) to a pickle file.

This can be reloaded if necessary and further analyses carried out.

Parameters fileCollisionMethod: Collision method passed to handleFileCollision()

saveAsWideText(fileName, delim='auto', matrixOnly=False, appendFile=None, encoding='utf-8-sig',
fileCollisionMethod='rename', sortColumns=False)
Saves a long, wide-format text file, with one line representing the attributes and data for a single trial. Suitable
for analysis in R and SPSS.

If `appendFile=True` then the data will be added to the bottom of an existing file. Otherwise, if the file exists already it will be overwritten.

If `matrixOnly=True` then the file will not contain a header row, which can be handy if you want to append data to an existing file of the same format.

**Parameters**

- `fileName` (str): if extension is not specified, `.csv` will be appended if the delimiter is ``, else `.tsv` will be appended. Can include path info.
- `delim` (str): allows the user to use a delimiter other than the default tab (````) is popular with file extension `.csv`)
- `matrixOnly` (bool): outputs the data with no header row.
- `appendFile` (bool): will add this output to the end of the specified file if it already exists.
- `encoding` (str): The encoding to use when saving a the file. Defaults to `utf-8-sig`.
- `sortColumns` (bool): will sort columns alphabetically by header name if True.

### 8.7.2 TrialHandler

**class** `psychopy.data.TrialHandler` *(trialList, nReps, method='random', dataTypes=None, extraInfo=None, seed=None, originPath=None, name='', autoLog=True)*

Class to handle trial sequencing and data storage.

Calls to `.next()` will fetch the next trial object given to this handler, according to the method specified (random, sequential, fullRandom). Calls will raise a `StopIteration` error if trials have finished.

See demo_trialHandler.py

The psydat file format is literally just a pickled copy of the TrialHandler object that saved it. You can open it with:

```python
from psychopy.tools.filetools import fromFile
dat = fromFile(path)
```

Then you'll find that `dat` has the following attributes that

**Parameters**

- `trialList` (list): a simple list (or flat array) of dictionaries specifying conditions. This can be imported from an excel/csv file using `importConditions()`.
- `nReps` (int): number of repeats for all conditions
- `method` (str): `random`, `sequential`, or `fullRandom` ‘sequential’ obviously presents the conditions in the order they appear in the list. ‘random’ will result in a shuffle of the conditions on each repeat, but all conditions occur once before the second repeat etc. ‘fullRandom’ fully randomises the trials across repeats as well, which means you could potentially run all trials of one condition before any trial of another.
- `dataTypes` (optional): list of names for data storage. e.g. `[corr,’rt’,’resp’]`. If not provided then these will be created as needed during calls to `addData()`.
- `extraInfo` (dict): This will be stored alongside the data and usually describes the experiment and subject ID, date etc.
seed: an integer  If provided then this fixes the random number generator to use the same pattern of trials, by seeding its startpoint

originPath: a string describing the location of the  script / experiment file path. The psydat file format will store a copy of the experiment if possible. If originPath==None is provided here then the TrialHandler will still store a copy of the script where it was created. If OriginPath=-1 then nothing will be stored.

Attributes (after creation)

.data - a dictionary (or more strictly, a DataHandler subclass of a dictionary) of numpy arrays, one for each data type stored
.trialList - the original list of dicts, specifying the conditions
.thisIndex - the index of the current trial in the original conditions list
.nTotal - the total number of trials that will be run
.nRemaining - the total number of trials remaining
.thisN - total trials completed so far
.thisRepN - which repeat you are currently on
.thisTrialN - which trial number within that repeat
.thisTrial - a dictionary giving the parameters of the current trial
.finished - True/False for have we finished yet
.extraInfo - the dictionary of extra info as given at beginning
.origin - the contents of the script or builder experiment that created the handler

_makeIndices (inputArray)
Creates an array of tuples the same shape as the input array where each tuple contains the indices to itself in the array.
Useful for shuffling and then using as a reference.

30 trials will be returned by .next() in the order: 0, 1, 2, 3, 4, 5, 0, 1, 2, …… 3, 4, 5
To add a new type of sequence (as of v1.65.02): - add the sequence generation code here - adjust “if self.method in [ …]:” in both __init__ and .next() - adjust allowedVals in experiment.py -> shows up in DlgLoopProperties Note that users can make any sequence whatsoever outside of PsychoPy, and specify sequential order; any order is possible this way.
_terminate()
Remove references to ourself in experiments and terminate the loop

addData (thisType, value, position=None)
Add data for the current trial

getEarlierTrial (n=-1)
Returns the condition information from n trials previously. Useful for comparisons in n-back tasks. Returns
‘None’ if trying to access a trial prior to the first.

getExp()
Return the ExperimentHandler that this handler is attached to, if any. Returns None if not attached

getFutureTrial (n=1)
Returns the condition for n trials into the future, without advancing the trials. A negative n returns a previous
(past) trial. Returns ‘None’ if attempting to go beyond the last trial.

getOriginPathAndFile (originPath=None)
Attempts to determine the path of the script that created this data file and returns both the path to that script
and its contents. Useful to store the entire experiment with the data.
If originPath is provided (e.g. from Builder) then this is used otherwise the calling script is the originPath
(fine from a standard python script).

next()
Advances to next trial and returns it. Updates attributes; thisTrial, thisTrialN and thisIndex If the trials have
ended this method will raise a StopIteration error. This can be handled with code such as:

```python
trials = data.TrialHandler(........)
for eachTrial in trials:  # automatically stops when done
    # do stuff
```
or:

```python
trials = data.TrialHandler(........)
while True:  # ie forever
    try:
        thisTrial = trials.next()
    except StopIteration:  # we got a StopIteration error
        break  # break out of the forever loop
    # do stuff here for the trial
```

printAsText (stimOut=None, dataOut=('all_mean', 'all_std', 'all_raw'), delim='\t', matrixOnly=False)
Exactly like saveAsText() except that the output goes to the screen instead of a file

saveAsExcel (fileName, sheetName='rawData', stimOut=None, dataOut=('n', 'all_mean', 'all_std', 'all_raw'), matrixOnly=False, appendFile=True, fileCollisionMethod='rename')
Save a summary data file in Excel OpenXML format workbook (.xlsx) for processing in most spreadsheet
packages. This format is compatible with versions of Excel (2007 or greater) and and with OpenOffice
(>=3.0).

It has the advantage over the simpler text files (see TrialHandler.saveAsText() ) that data can be
stored in multiple named sheets within the file. So you could have a single file named after your experiment
and then have one worksheet for each participant. Or you could have one file for each participant and then
multiple sheets for repeated sessions etc.

The file extension .xlsx will be added if not given already.

Parameters
fileName: string  the name of the file to create or append. Can include relative or absolute path

sheetName: string  the name of the worksheet within the file

stimOut: list of strings  the attributes of the trial characteristics to be output. To use this you need to have provided a list of dictionaries specifying to trialList parameter of the TrialHandler and give here the names of strings specifying entries in that dictionary

dataOut: list of strings  specifying the dataType and the analysis to be performed, in the form dataType_analysis. The data can be any of the types that you added using trialHandler.data.add() and the analysis can be either ‘raw’ or most things in the numpy library, including ‘mean’, ‘std’, ‘median’, ‘max’, ‘min’. e.g. rt_max will give a column of max reaction times across the trials assuming that rt values have been stored. The default values will output the raw, mean and std of all datatypes found.

appendFile: True or False  If False any existing file with this name will be overwritten. If True then a new worksheet will be appended. If a worksheet already exists with that name a number will be added to make it unique.

fileCollisionMethod: string  Collision method passed to handleFileCollision(). This is ignored if append is True.

saveAsJson (fileName=None, encoding='utf-8', fileCollisionMethod='rename')
Serialize the object to the JSON format.

Parameters

• fileName (string, or None) – the name of the file to create or append. Can include a relative or absolute path. If None, will not write to a file, but return an in-memory JSON object.

• encoding (string, optional) – The encoding to use when writing the file.

• fileCollisionMethod (string) – Collision method passed to handleFileCollision(). Can be either of ‘rename’, ‘overwrite’, or ‘fail’.

Notes

Currently, a copy of the object is created, and the copy’s .origin attribute is set to an empty string before serializing because loading the created JSON file would sometimes fail otherwise.

saveAsPickle (fileName, fileCollisionMethod='rename')
Basically just saves a copy of the handler (with data) to a pickle file.

This can be reloaded if necessary and further analyses carried out.

Parameters  fileCollisionMethod: Collision method passed to handleFileCollision()

saveAsText (fileName, stimOut=None, dataOut=('n', 'all_mean', 'all_std', 'all_raw'), delim=None, matrixOnly=False, appendFile=True, summarised=True, fileCollisionMethod='rename', encoding='utf-8-sig')
Write a text file with the data and various chosen stimulus attributes

Parameters

• fileName:  will have .tsv appended and can include path info.

• stimOut:  the stimulus attributes to be output. To use this you need to use a list of dictionaries and give here the names of dictionary keys that you want as strings
**dataOut**: a list of strings specifying the dataType and the analysis to be performed, in the form `dataType_analysis`. The data can be any of the types that you added using `trialHandler.data.add()` and the analysis can be either ‘raw’ or most things in the numpy library, including: ‘mean’, ‘std’, ‘median’, ‘max’, ‘min’… The default values will output the raw, mean and std of all datatypes found.

**delim**: allows the user to use a delimiter other than tab (“,” is popular with file extension “.csv”)

**matrixOnly**: outputs the data with no header row or extraInfo attached

**appendFile**: will add this output to the end of the specified file if it already exists

**fileCollisionMethod**: Collision method passed to `handleFileCollision()`

**encoding**: The encoding to use when saving a the file. Defaults to utf-8-sig.

---

**saveAsWideText** *(fileName, delim=None, matrixOnly=False, appendFile=True, encoding='utf-8-sig', fileCollisionMethod='rename')*

Write a text file with the session, stimulus, and data values from each trial in chronological order. Also, return a pandas DataFrame containing same information as the file.

That is, unlike ‘saveAsText’ and ‘saveAsExcel’:

- each row comprises information from only a single trial.
- no summarizing is done (such as collapsing to produce mean and standard deviation values across trials).

This ‘wide’ format, as expected by R for creating dataframes, and various other analysis programs, means that some information must be repeated on every row.

In particular, if the trialHandler’s ‘extraInfo’ exists, then each entry in there occurs in every row. In builder, this will include any entries in the ‘Experiment info’ field of the ‘Experiment settings’ dialog. In Coder, this information can be set using something like:

```python
myTrialHandler.extraInfo = {'SubjID': 'Joan Smith', 'Group': 'Control'}
```

---

**Parameters**

- **fileName**: if extension is not specified, ‘.csv’ will be appended if the delimiter is ‘,’ else ‘.tsv’ will be appended. Can include path info.
- **delim**: allows the user to use a delimiter other than the default tab (“,” is popular with file extension “.csv”)
- **matrixOnly**: outputs the data with no header row.
- **appendFile**: will add this output to the end of the specified file if it already exists.
- **fileCollisionMethod**: Collision method passed to `handleFileCollision()`
- **encoding**: The encoding to use when saving a the file. Defaults to utf-8-sig.

---

**setExp**(exp)

Sets the ExperimentHandler that this handler is attached to

Do NOT attempt to set the experiment using:

```python
trials._exp = myExperiment
```

because it needs to be performed using the `weakref` module.
8.7.3 TrialHandler2

class psychopy.data.TrialHandler2(trialList, nReps, method='random', dataTypes=None, extraInfo=None, seed=None, originPath=None, name='', autoLog=True)

Class to handle trial sequencing and data storage.

Calls to .next() will fetch the next trial object given to this handler, according to the method specified (random, sequential, fullRandom). Calls will raise a StopIteration error if trials have finished.

See demo_trialHandler.py

The psydat file format is literally just a pickled copy of the TrialHandler object that saved it. You can open it with:

```python
from psychopy.tools.filetools import fromFile
dat = fromFile(path)
```

Then you’ll find that `dat` has the following attributes that

Parameters

- **trialList**: filename or a simple list (or flat array) of dictionaries specifying conditions
- **nReps**: number of repeats for all conditions
- **method**: `'random'`, `'sequential'`, or `'fullRandom'` ‘sequential’ obviously presents the conditions in the order they appear in the list. ‘random’ will result in a shuffle of the conditions on each repeat, but all conditions occur once before the second repeat etc. ‘fullRandom’ fully randomises the trials across repeats as well, which means you could potentially run all trials of one condition before any trial of another.
- **dataTypes**: (optional) list of names for data storage. e.g. `[‘corr’,’rt’,’resp’]`. If not provided then these will be created as needed during calls to `addData()`.
- **extraInfo**: A dictionary This will be stored alongside the data and usually describes the experiment and subject ID, date etc.
- **seed**: an integer If provided then this fixes the random number generator to use the same pattern of trials, by seeding its startpoint.
- **originPath**: a string describing the location of the script / experiment file path. The psydat file format will store a copy of the experiment if possible. If `originPath==None` is provided here then the TrialHandler will still store a copy of the script where it was created. If `OriginPath==-1` then nothing will be stored.

Attributes (after creation)

- **.data** - a dictionary of numpy arrays, one for each data type stored
- **.trialList** - the original list of dicts, specifying the conditions
- **.thisIndex** - the index of the current trial in the original conditions list
- **.nTotal** - the total number of trials that will be run
- **.nRemaining** - the total number of trials remaining
- **.thisN** - total trials completed so far
- **.thisRepN** - which repeat you are currently on
- **.thisTrialN** - which trial number within that repeat
- **.thisTrial** - a dictionary giving the parameters of the current trial
PsychoPy - Psychology software for Python, Release 2020.2.10

.finished - True/False for have we finished yet
.extraInfo - the dictionary of extra info as given at beginning
.origin - the contents of the script or builder experiment that created the handler

__terminate__
Remove references to ourself in experiments and terminate the loop

addData__(thisType, value)__
Add a piece of data to the current trial

property data
Returns a pandas DataFrame of the trial data so far Read only attribute - you can't directly modify TrialHandler.data

Note that data are stored internally as a list of dictionaries, one per trial. These are converted to a DataFrame on access.

gETEarlierTrial__(n=-1)__
Returns the condition information from n trials previously. Useful for comparisons in n-back tasks. Returns 'None' if trying to access a trial prior to the first.

gETExp__
Return the ExperimentHandler that this handler is attached to, if any. Returns None if not attached

gETFUTuRETrial__(n=1)__
Returns the condition for n trials into the future, without advancing the trials. Returns 'None' if attempting to go beyond the last trial.

gETOriGInPaTHAndFiLE__(originPath=None)__
Attempts to determine the path of the script that created this data file and returns both the path to that script and its contents. Useful to store the entire experiment with the data.

If originPath is provided (e.g. from Builder) then this is used otherwise the calling script is the originPath (fine from a standard python script).

next__
Advances to next trial and returns it. Updates attributes; thisTrial, thisTrialN and thisIndex If the trials have ended this method will raise a StopIteration error. This can be handled with code such as:

```python
trials = data.TrialHandler(........)
for eachTrial in trials:  # automatically stops when done
    # do stuff
```

or:

```python
trials = data.TrialHandler(........)
while True:  # ie forever
    try:
        thisTrial = trials.next()
    except StopIteration:  # we got a StopIteration error
        break  # break out of the forever loop
    # do stuff here for the trial
```

printAsText__(stimOut=None, dataOut=('all_mean', 'all_std', 'all_raw'), delim='\t', matrixOnly=False)__
Exactly like saveAsText() except that the output goes to the screen instead of a file

saveAsExcel__(fileName, sheetName='rawData', stimOut=None, dataOut=('n', 'all_mean', 'all_std', 'all_raw'), matrixOnly=False, appendFile=True, fileCollisionMethod='rename')
Save a summary data file in Excel OpenXML format workbook (.xlsx) for processing in most spreadsheet

496 Chapter 8. Reference Manual (API)
packages. This format is compatible with versions of Excel (2007 or greater) and and with OpenOffice (>=3.0).

It has the advantage over the simpler text files (see TrialHandler.saveAsText() ) that data can be stored in multiple named sheets within the file. So you could have a single file named after your experiment and then have one worksheet for each participant. Or you could have one file for each participant and then multiple sheets for repeated sessions etc.

The file extension .xlsx will be added if not given already.

Parameters

- **fileName**: string the name of the file to create or append. Can include relative or absolute path
- **sheetName**: string the name of the worksheet within the file
- **stimOut**: list of strings the attributes of the trial characteristics to be output. To use this you need to have provided a list of dictionaries specifying to trialList parameter of the TrialHandler and give here the names of strings specifying entries in that dictionary
- **dataOut**: list of strings specifying the dataType and the analysis to be performed, in the form dataType_analysis. The data can be any of the types that you added using trialHandler.data.add() and the analysis can be either 'raw' or most things in the numpy library, including 'mean', 'std', 'median', 'max', 'min'. e.g. **rt_max** will give a column of max reaction times across the trials assuming that rt values have been stored. The default values will output the raw, mean and std of all datatypes found.
- **appendFile**: True or False If False any existing file with this name will be overwritten. If True then a new worksheet will be appended. If a worksheet already exists with that name a number will be added to make it unique.
- **fileCollisionMethod**: string Collision method passed to handleFileCollision()

This is ignored if append is True.

**saveAsJson** (fileName=None, encoding='utf-8', fileCollisionMethod='rename')

Serialize the object to the JSON format.

Parameters

- **fileName** (string, or None) – the name of the file to create or append. Can include a relative or absolute path. If None, will not write to a file, but return an in-memory JSON object.
- **encoding** (string, optional) – The encoding to use when writing the file.
- **fileCollisionMethod** (string) – Collision method passed to handleFileCollision(). Can be either of 'rename', 'overwrite', or 'fail'.

Notes

Currently, a copy of the object is created, and the copy’s .origin attribute is set to an empty string before serializing because loading the created JSON file would sometimes fail otherwise.

The RNG self._rng cannot be serialized as-is, so we store its state in self._rng_state so we can restore it when loading.

**saveAsPickle** (fileName, fileCollisionMethod='rename')

Basically just saves a copy of the handler (with data) to a pickle file.

This can be reloaded if necessary and further analyses carried out.
Parameters

fileCollisionMethod: Collision method passed to handleFileCollision()

saveAsText (fileName, stimOut=None, dataOut=('n', 'all_mean', 'all_std', 'all_raw'), delim=None, matrixOnly=False, appendFile=True, summarised=True, fileCollisionMethod='rename', encoding='utf-8-sig')

Write a text file with the data and various chosen stimulus attributes

Parameters

fileName: will have .tsv appended and can include path info.

stimOut: the stimulus attributes to be output. To use this you need to use a list of dictionaries and give here the names of dictionary keys that you want as strings

dataOut: a list of strings specifying the dataType and the analysis to be performed, in the form dataType_analysis. The data can be any of the types that you added using trialHandler.data.add() and the analysis can be either ‘raw’ or most things in the numpy library, including: ‘mean’, ‘std’, ‘median’, ‘max’, ‘min’… The default values will output the raw, mean and std of all datatypes found

delim: allows the user to use a delimiter other than tab (”, “ is popular with file extension “.csv”)

matrixOnly: outputs the data with no header row or extraInfo attached

appendFile: will add this output to the end of the specified file if it already exists

fileCollisionMethod: Collision method passed to handleFileCollision()

encoding: The encoding to use when saving a the file. Defaults to utf-8-sig.

saveAsWideText (fileName, delim=None, matrixOnly=False, appendFile=True, encoding='utf-8-sig', fileCollisionMethod='rename')

Write a text file with the session, stimulus, and data values from each trial in chronological order. Also, return a pandas DataFrame containing same information as the file.

That is, unlike ‘saveAsText’ and ‘saveAsExcel’:

• each row comprises information from only a single trial.
• no summarising is done (such as collapsing to produce mean and standard deviation values across trials).

This 'wide' format, as expected by R for creating dataframes, and various other analysis programs, means that some information must be repeated on every row.

In particular, if the trialHandler's 'extraInfo' exists, then each entry in there occurs in every row. In builder, this will include any entries in the 'Experiment info' field of the 'Experiment settings' dialog. In Coder, this information can be set using something like:

```python
myTrialHandler.extraInfo = {'SubjID': 'Joan Smith', 'Group': 'Control'}
```

Parameters

fileName: if extension is not specified, `.csv` will be appended if the delimiter is ‘,’; else `.tsv` will be appended. Can include path info.

delim: allows the user to use a delimiter other than the default tab (”, “ is popular with file extension “.csv”)

matrixOnly: outputs the data with no header row.

appendFile: will add this output to the end of the specified file if it already exists.
**fileCollisionMethod:** Collision method passed to `handleFileCollision()`

**encoding:** The encoding to use when saving a the file. Defaults to `utf-8-sig`.

```python
setExp (exp)
Sets the ExperimentHandler that this handler is attached to
Do NOT attempt to set the experiment using:

```trials._exp = myExperiment```

because it needs to be performed using the `weakref` module.

### 8.7.4 TrialHandlerExt

**class** `psychopy.data.TrialHandlerExt` (trialList, nReps, method='random', DataTypes=None, extraInfo=None, seed=None, originPath=None, name='', autoLog=True)

A class for handling trial sequences in a *non-counterbalanced design* (i.e. oddball paradigms). Its functions are a superset of the class TrialHandler, and as such, can also be used for normal trial handling.

TrialHandlerExt has the same function names for data storage facilities.

To use non-counterbalanced designs, all TrialType dict entries in the trial list must have a key called “weight”. For example, if you want trial types A, B, C, and D to have 10, 5, 3, and 2 repetitions per block, then the trialList can look like:

```json
[{Name:'A', ..., weight:10}, {Name:'B', ..., weight:5}, {Name:'C', ..., weight:3}, {Name:'D', ..., weight:2}]
```

For experimenters using an excel or csv file for trial list, a column called weight is appropriate for this purpose.

Calls to `.next()` will fetch the next trial object given to this handler, according to the method specified (random, sequential, fullRandom). Calls will raise a StopIteration error when all trials are exhausted.

*Authored by Suddha Sourav at BPN, Uni Hamburg - heavily borrowing from the TrialHandler class*

**Parameters**

- **trialList:** a simple list (or flat array) of dictionaries specifying conditions. This can be imported from an excel / csv file using `importConditions()` For non-counterbalanced designs, each dict entry in trialList must have a key called weight!

- **nReps:** number of repeats for all conditions. When using a non-counterbalanced design, nReps is analogous to the number of blocks.

- **method:** `'random'`, `'sequential'`, or `'fullRandom'` When the weights are not specified: `sequential` presents the conditions in the order they appear in the list. `random` will result in a shuffle of the conditions on each repeat, but all conditions occur once before the second repeat etc. `fullRandom` fully randomises the trials across repeats as well, which means you could potentially run all trials of one condition before any trial of another.

In the presence of weights: `sequential` presents each trial type the number of times specified by its weight, before moving on to the next type. `random` randomizes the presentation order within block. `fullRandom` shuffles trial order across weights an nRep, that is, a full shuffling.

- **dataTypes:** (optional) list of names for data storage. e.g. ['corr', 'rt', 'resp']. If not provided then these will be created as needed during calls to `addData()`.

- **extraInfo:** A dictionary This will be stored alongside the data and usually describes the experiment and subject ID, date etc.
seed: an integer  If provided then this fixes the random number generator to use the same pattern of trials, by seeding its startpoint

originPath: a string describing the location of the script / experiment file path. The psydat file format will store a copy of the experiment if possible. If originPath=None is provided here then the TrialHandler will still store a copy of the script where it was created. If OriginPath=-1 then nothing will be stored.

Attributes (after creation)

.data - a dictionary of numpy arrays, one for each data type  stored
.trialList - the original list of dicts, specifying the conditions
.thisIndex - the index of the current trial in the original  conditions list
.nTotal - the total number of trials that will be run
.nRemaining - the total number of trials remaining
.thisN - total trials completed so far
.thisRepN - which repeat you are currently on
.thisTrialN - which trial number within that repeat
.thisTrial - a dictionary giving the parameters of the current  trial
.finished - True/False for have we finished yet
.extraInfo - the dictionary of extra info as given at beginning
.origin - the contents of the script or builder experiment that  created the handler
.trialWeights - None if all weights are not specified. If all weights are specified, then a list containing the weights of the trial types.

_createOutputArray (stimOut, dataOut, delim=None, matrixOnly=False)
  Does the leg-work for saveAsText and saveAsExcel. Combines stimOut with _parseDataOutput()

_createOutputArrayData (dataOut)
  This just creates the dataOut part of the output matrix. It is called by _createOutputArray() which creates the header line and adds the stimOut columns

_createSequence ()
  Pre-generates the sequence of trial presentations (for non-adaptive methods). This is called automatically when the TrialHandler is initialised so doesn’t need an explicit call from the user.

  The returned sequence has form indices[stimN][repN] Example: sequential with 6 trialtypes (rows), 5 reps (cols), returns:

  [[0 0 0 0] [1 1 1 1] [2 2 2 2] [3 3 3 3] [4 4 4 4] [5 5 5 5]]

  These 30 trials will be returned by .next() in the order: 0, 1, 2, 3, 4, 5, 0, 1, 2, … … 3, 4, 5

  Example: random, with 3 trialtypes, where the weights of conditions 0,1, and 2 are 3,2, and 1 respectively, and a rep value of 5, might return:

  [[0 1 2 0] [1 0 1 1] [0 2 0 0] [0 0 0 1] [2 0 1 0] [2 1 0 2]]

  These 30 trials will be returned by .next() in the order: 0, 1, 0, 0, 2, 1, 1, 0, 2, 0, 0, 1, … … 0, 2, 0

stopIteration
To add a new type of sequence (as of v1.65.02): - add the sequence generation code here - adjust “if self.method in [ ...]:” in both __init__ and .next() - adjust allowedVals in experiment.py -> shows up in DlgLoopProperties Note that users can make any sequence whatsoever outside of PsychoPy, and specify sequential order; any order is possible this way.

_makeIndices(inputArray)
Creates an array of tuples the same shape as the input array where each tuple contains the indices to itself in the array.
Useful for shuffling and then using as a reference.

_terminate()
Remove references to ourself in experiments and terminate the loop

addData(thisType, value, position=None)
Add data for the current trial

getCurrentTrialPosInDataHandler()

getPreviousTrial(n=-1)
Returns the condition information from n trials previously. Useful for comparisons in n-back tasks. Returns ‘None’ if trying to access a trial prior to the first.

getExp()
Return the ExperimentHandler that this handler is attached to, if any. Returns None if not attached

getFutureTrial(n=1)
Returns the condition for n trials into the future, without advancing the trials. A negative n returns a previous (past) trial. Returns ‘None’ if attempting to go beyond the last trial.

getNextTrialPosInDataHandler()

getOriginPathAndFile(originPath=None)
Attempts to determine the path of the script that created this data file and returns both the path to that script and its contents. Useful to store the entire experiment with the data.

If originPath is provided (e.g. from Builder) then this is used otherwise the calling script is the originPath (fine from a standard python script).

next()
Advances to next trial and returns it. Updates attributes; thisTrial, thisTrialN and thisIndex. If the trials have ended this method will raise a StopIteration error. This can be handled with code such as:

```python
trials = data.TrialHandler(.........)
for eachTrial in trials:  # automatically stops when done
    # do stuff
```

or:

```python
trials = data.TrialHandler(.........)
while True:  # ie forever
    try:
        thisTrial = trials.next()
    except StopIteration:  # we got a StopIteration error
        break  # break out of the forever loop
    # do stuff here for the trial
```

printAsText(stimOut=None, dataOut=('all_mean', 'all_std', 'all_raw'), delim='\t', matrixOnly=False)
Exactly like saveAsText() except that the output goes to the screen instead of a file.
saveAsExcel(fileName, sheetName='rawData', stimOut=None, dataOut=('n', 'all_mean', 'all_std', 'all_raw'), matrixOnly=False, appendFile=True, fileCollisionMethod='rename')

Save a summary data file in Excel OpenXML format workbook (.xlsx) for processing in most spreadsheet packages. This format is compatible with versions of Excel (2007 or greater) and and with OpenOffice (>=3.0).

It has the advantage over the simpler text files (see TrialHandler.saveAsText() ) that data can be stored in multiple named sheets within the file. So you could have a single file named after your experiment and then have one worksheet for each participant. Or you could have one file for each participant and then multiple sheets for repeated sessions etc.

The file extension .xlsx will be added if not given already.

Parameters

- fileName: string the name of the file to create or append. Can include relative or absolute path
- sheetName: string the name of the worksheet within the file
- stimOut: list of strings the attributes of the trial characteristics to be output. To use this you need to have provided a list of dictionaries specifying to trialList parameter of the TrialHandler and give here the names of strings specifying entries in that dictionary
- dataOut: list of strings specifying the dataType and the analysis to be performed, in the form dataType_analysis. The data can be any of the types that you added using trialHandler.data.add() and the analysis can be either ‘raw’ or most things in the numpy library, including ‘mean’,‘std’,‘median’,‘max’,‘min’. e.g. rt_max will give a column of max reaction times across the trials assuming that rt values have been stored. The default values will output the raw, mean and std of all datatypes found.
- appendFile: True or False If False any existing file with this name will be overwritten. If True then a new worksheet will be appended. If a worksheet already exists with that name a number will be added to make it unique.
- fileCollisionMethod: string Collision method passed to handleFileCollision()

This is ignored if append is True.

saveAsJson(fileName=None, encoding='utf-8', fileCollisionMethod='rename')

Serialize the object to the JSON format.

Parameters

- fileName (string, or None) – the name of the file to create or append. Can include a relative or absolute path. If None, will not write to a file, but return an in-memory JSON object.
- encoding (string, optional) – The encoding to use when writing the file.
- fileCollisionMethod (string) – Collision method passed to handleFileCollision(). Can be either of ‘rename’, ‘overwrite’, or ‘fail’.
Notes

Currently, a copy of the object is created, and the copy’s .origin attribute is set to an empty string before serializing because loading the created JSON file would sometimes fail otherwise.

**saveAsPickle** *(fileName, fileCollisionMethod='rename')*

Basically just saves a copy of the handler (with data) to a pickle file.

This can be reloaded if necessary and further analyses carried out.

**Parameters**

- **fileCollisionMethod**: Collision method passed to `handleFileCollision()`

**saveAsText** *(fileName, stimOut=None, dataOut=('n', 'all_mean', 'all_std', 'all_raw'), delim=None, matrixOnly=False, appendFile=True, summarised=True, fileCollisionMethod='rename', encoding='utf-8-sig')*

Write a text file with the data and various chosen stimulus attributes

**Parameters**

- **fileName**: will have .tsv appended and can include path info.
- **stimOut**: the stimulus attributes to be output. To use this you need to use a list of dictionaries and give here the names of dictionary keys that you want as strings
- **dataOut**: a list of strings specifying the dataType and the analysis to be performed, in the form `dataType_analysis`. The data can be any of the types that you added using trialHandler.data.add() and the analysis can be either ‘raw’ or most things in the numpy library, including: ‘mean’, ‘std’, ‘median’, ‘max’, ‘min’… The default values will output the raw, mean and std of all datatypes found
- **delim**: allows the user to use a delimiter other than tab (”,,” is popular with file extension “.csv”)
- **matrixOnly**: outputs the data with no header row or extraInfo attached
- **appendFile**: will add this output to the end of the specified file if it already exists
- **fileCollisionMethod**: Collision method passed to `handleFileCollision()`
- **encoding**: The encoding to use when saving a the file. Defaults to utf-8-sig.

**saveAsWideText** *(fileName, delim='
', matrixOnly=False, appendFile=True, encoding='utf-8-sig', fileCollisionMethod='rename')*

Write a text file with the session, stimulus, and data values from each trial in chronological order.

**That is, unlike ‘saveAsText’ and ‘saveAsExcel’:**

- each row comprises information from only a single trial.
- no summarizing is done (such as collapsing to produce mean and standard deviation values across trials).

This ‘wide’ format, as expected by R for creating dataframes, and various other analysis programs, means that some information must be repeated on every row.

In particular, if the trialHandler’s ‘extraInfo’ exists, then each entry in there occurs in every row. In builder, this will include any entries in the ‘Experiment info’ field of the ‘Experiment settings’ dialog. In Coder, this information can be set using something like:

```python
myTrialHandler.extraInfo = {'SubjID': 'Joan Smith', 'Group': 'Control'}
```

**Parameters**
fileName: if extension is not specified, `.csv` will be appended if the delimiter is ``,`, else `.txt` will be appended. Can include path info.

delim: allows the user to use a delimiter other than the default tab (`` ``` is popular with file extension ``.csv``)

matrixOnly: outputs the data with no header row.

appendFile: will add this output to the end of the specified file if it already exists.

fileCollisionMethod: Collision method passed to `handleFileCollision()`

encoding: The encoding to use when saving a the file. Defaults to `utf-8-sig`.

```python
setExp (exp)
```

Sets the ExperimentHandler that this handler is attached to

Do NOT attempt to set the experiment using:

```python
trials._exp = myExperiment
```

because it needs to be performed using the `weakref` module.

### 8.7.5 StairHandler

**class** `psychopy.data.StairHandler` (`startVal`, `nReversals=None`, `stepSizes=4`, `nTrials=0`, `nUp=1`, `nDown=3`, `applyInitialRule=True`, `extraInfo=None`, `method='2AFC'`, `stepType='db'`, `minVal=None`, `maxVal=None`, `originPath=None`, `name='', autoLog=True`, **kwargs)

Class to handle smoothly the selection of the next trial and report current values etc. Calls to `next()` will fetch the next object given to this handler, according to the method specified.

See Demos >> ExperimentalControl >> JND_staircase_exp.py

The staircase will terminate when `nTrials` AND `nReversals` have been exceeded. If `stepSizes` was an array and has been exceeded before `nTrials` is exceeded then the staircase will continue to reverse.

`nUp` and `nDown` are always considered as 1 until the first reversal is reached. The values entered as arguments are then used.

**Parameters**

- **startVal**: The initial value for the staircase.

- **nReversals**: The minimum number of reversals permitted. If `stepSizes` is a list, but the minimum number of reversals to perform, `nReversals`, is less than the length of this list, PsychoPy will automatically increase the minimum number of reversals and emit a warning.

- **stepSizes**: The size of steps as a single value or a list (or array). For a single value the step size is fixed. For an array or list the step size will progress to the next entry at each reversal.

- **nTrials**: The minimum number of trials to be conducted. If the staircase has not reached the required number of reversals then it will continue.

- **nUp**: The number of ‘incorrect’ (or 0) responses before the staircase level increases.

- **nDown**: The number of ‘correct’ (or 1) responses before the staircase level decreases.

- **applyInitialRule** [bool] Whether to apply a 1-up/1-down rule until the first reversal point (if `True`), before switching to the specified up/down rule.
**extraInfo:** A dictionary (typically) that will be stored along with collected data using `saveASpickle()` or `saveAsText()` methods.

**method:** Not used and may be deprecated in future releases.

**stepType:** `‘db’, ‘lin’, ‘log’` The type of steps that should be taken each time. ‘lin’ will simply add or subtract that amount each step, ‘db’ and ‘log’ will step by a certain number of decibels or log units (note that this will prevent your value ever reaching zero or less)

**minVal:** `None, or a number` The smallest legal value for the staircase, which can be used to prevent it reaching impossible contrast values, for instance.

**maxVal:** `None, or a number` The largest legal value for the staircase, which can be used to prevent it reaching impossible contrast values, for instance.

Additional keyword arguments will be ignored.

**Notes**

The additional keyword arguments `**kwargs` might for example be passed by the `MultiStairHandler`, which expects a `label` keyword for each staircase. These parameters are to be ignored by the StairHandler.

```python
_intensityDec()  
derectment the current intensity and reset counter

_intensityInc()  
increment the current intensity and reset counter

_terminate()  
Remove references to ourself in experiments and terminate the loop
```

**addData** *(result, intensity=None)*

Deprecated since 1.79.00: This function name was ambiguous. Please use one of these instead:

```python
.addResponse(result, intensity).addOtherData('dataName', value')
```

**addOtherData** *(dataName, value)*

Add additional data to the handler, to be tracked alongside the result data but not affecting the value of the staircase

**addResponse** *(result, intensity=None)*

Add a 1 or 0 to signify a correct / detected or incorrect / missed trial.

This is essential to advance the staircase to a new intensity level!

Supplying an `intensity` value here indicates that you did not use the recommended intensity in your last trial and the staircase will replace its recorded value with the one you supplied here.

**calculateNextIntensity()**

Based on current intensity, counter of correct responses, and current direction.

**getExp()**

Return the ExperimentHandler that this handler is attached to, if any. Returns None if not attached

**getOriginPathAndFile** *(originPath=None)*

Attempts to determine the path of the script that created this data file and returns both the path to that script and its contents. Useful to store the entire experiment with the data.

If `originPath` is provided (e.g. from Builder) then this is used otherwise the calling script is the `originPath` (fine from a standard python script).

**next()**

Advances to next trial and returns it. Updates attributes; `thisTrial`, `thisTrialN` and `thisIndex`.

---

8.7. **psychopy.data** - functions for storing/saving/analysing data
If the trials have ended, calling this method will raise a StopIteration error. This can be handled with code such as:

```python
staircase = data.StairHandler(........)
for eachTrial in staircase:  # automatically stops when done
    # do stuff
```
or:

```python
staircase = data.StairHandler(........)
while True:  # ie forever
    try:
        thisTrial = staircase.next()
    except StopIteration:  # we got a StopIteration error
        break  # break out of the forever loop
    # do stuff here for the trial
```

`printAsText(stimOut=None, dataOut=('all_mean', 'all_std', 'all_raw'), delim='\t', matrixOnly=False)`

Exactly like `saveAsText()` except that the output goes to the screen instead of a file.

`saveAsExcel(fileName, sheetName='data', matrixOnly=False, appendFile=True, fileCollisionMethod='rename')`

Save a summary data file in Excel OpenXML format workbook (.xlsx) for processing in most spreadsheet packages. This format is compatible with versions of Excel (2007 or greater) and and with OpenOffice (>=3.0).

It has the advantage over the simpler text files (see `TrialHandler.saveAsText()` ) that data can be stored in multiple named sheets within the file. So you could have a single file named after your experiment and then have one worksheet for each participant. Or you could have one file for each participant and then multiple sheets for repeated sessions etc.

The file extension .xlsx will be added if not given already.

The file will contain a set of values specifying the staircase level ('intensity') at each reversal, a list of reversal indices (trial numbers), the raw staircase / intensity level on every trial and the corresponding responses of the participant on every trial.

**Parameters**

- **fileName:** string the name of the file to create or append. Can include relative or absolute path.
- **sheetName:** string the name of the worksheet within the file
- **matrixOnly:** True or False If set to True then only the data itself will be output (no additional info)
- **appendFile:** True or False If False any existing file with this name will be overwritten. If True then a new worksheet will be appended. If a worksheet already exists with that name a number will be added to make it unique.
- **fileCollisionMethod:** string Collision method passed to `handleFileCollision()`
  
  This is ignored if appendFile is True.

`saveAsJson(fileName=None, encoding='utf-8-sig', fileCollisionMethod='rename')`

Serialize the object to the JSON format.

**Parameters**

- **fileName** (string, or None) – the name of the file to create or append. Can include a relative or absolute path. If None, will not write to a file, but return an in-memory JSON object.
• **encoding** *(string, optional)* – The encoding to use when writing the file.

• **fileCollisionMethod** *(string)* – Collision method passed to `handleFileCollision()`. Can be either of ‘rename’, ‘overwrite’, or ‘fail’.

### Notes

Currently, a copy of the object is created, and the copy’s .origin attribute is set to an empty string before serializing because loading the created JSON file would sometimes fail otherwise.

**`saveAsPickle(fileName, fileCollisionMethod='rename')`**

Basically just saves a copy of self (with data) to a pickle file.

This can be reloaded if necessary and further analyses carried out.

**Parameters**

- `fileCollisionMethod`: Collision method passed to `handleFileCollision()`

**`saveAsText(fileName, delim=None, matrixOnly=False, fileCollisionMethod='rename', encoding='utf-8-sig')`**

Write a text file with the data

**Parameters**

- **fileName**: a string The name of the file, including path if needed. The extension .tsv will be added if not included.
- **delim**: a string the delimiter to be used (e.g. ‘ ‘ for tab-delimitted, ‘,’ for csv files)
- **matrixOnly**: True/False If True, prevents the output of the `extraInfo` provided at initialisation.
- **fileCollisionMethod**: Collision method passed to `handleFileCollision()`
- **encoding**: The encoding to use when saving a the file. Defaults to utf-8-sig.

**`setExp(exp)`**

Sets the ExperimentHandler that this handler is attached to

Do NOT attempt to set the experiment using:

```
trials._exp = myExperiment
```

because it needs to be performed using the `weakref` module.

### 8.7.6 PsiHandler

**class psychopy.data.PsiHandler**(nTrials, intensRange, alphaRange, betaRange, intensPrecision, alphaPrecision, betaPrecision, delta, stepType='lin', expectedMin=0.5, prior=None, fromFile=False, extraInfo=None, name='')

Handler to implement the “Psi” adaptive psychophysical method (Kontsevich & Tyler, 1999).

This implementation assumes the form of the psychometric function to be a cumulative Gaussian. Psi estimates the two free parameters of the psychometric function, the location (alpha) and slope (beta), using Bayes’ rule and grid approximation of the posterior distribution. It chooses stimuli to present by minimizing the entropy of this grid. Because this grid is represented internally as a 4-D array, one must choose the intensity, alpha, and beta ranges carefully so as to avoid a Memory Error. Maximum likelihood is used to estimate Lambda, the most likely location/slope pair. Because Psi estimates the entire psychometric function, any threshold defined on the function may be estimated once Lambda is determined.
It is advised that Lambda estimates are examined after completion of the Psi procedure. If the estimated alpha or beta values equal your specified search bounds, then the search range most likely did not contain the true value. In this situation the procedure should be repeated with appropriately adjusted bounds.

Because Psi is a Bayesian method, it can be initialized with a prior from existing research. A function to save the posterior over Lambda as a Numpy binary file is included.

Kontsevich & Tyler (1999) specify their psychometric function in terms of $d'$. PsiHandler avoids this and treats all parameters with respect to stimulus intensity. Specifically, the forms of the psychometric function assumed for Yes/No and Two Alternative Forced Choice (2AFC) are, respectively:

\[ Y(x) = 0.5 \times \delta + (1 - \delta) \times \text{normCdf} \]
\[ Y(x) = 0.5 \times \delta + (1 - \delta) \times (0.5 + 0.5 \times \text{normCdf}) \]

Initializes the handler and creates an internal Psi Object for grid approximation.

**Parameters**

- **nTrials** (int) The number of trials to run.
- **intensRange** (list) Two element list containing the (inclusive) endpoints of the stimuli intensity range.
- **alphaRange** (list) Two element list containing the (inclusive) endpoints of the alpha (location parameter) range.
- **betaRange** (list) Two element list containing the (inclusive) endpoints of the beta (slope parameter) range.
- **intensPrecision** (float or int) If stepType == 'lin', this specifies the step size of the stimuli intensity range. If stepType == 'log', this specifies the number of steps in the stimuli intensity range.
- **alphaPrecision** (float) The step size of the alpha (location parameter) range.
- **betaPrecision** (float) The step size of the beta (slope parameter) range.
- **delta** (float) The guess rate.
- **stepType** (str) The type of steps to be used when constructing the stimuli intensity range. If 'lin' then evenly spaced steps are used. If 'log' then logarithmically spaced steps are used. Defaults to 'lin'.
- **expectedMin** (float) The expected lower asymptote of the psychometric function (PMF).
  
  For a Yes/No task, the PMF usually extends across the interval [0, 1]; here, expectedMin should be set to 0.

  For a 2-AFC task, the PMF spreads out across [0.5, 1.0]. Therefore, expectedMin should be set to 0.5 in this case, and the 2-AFC psychometric function described above going to be is used.

  Currently, only Yes/No and 2-AFC designs are supported.

  Defaults to 0.5, or a 2-AFC task.
- **prior** (numpy ndarray or str) Optional prior distribution with which to initialize the Psi Object.
  
  This can either be a numpy ndarray object or the path to a numpy binary file (.npy) containing the ndarray.

- **fromFile** (str) Flag specifying whether prior is a file pathname or not.
- **extraInfo** (dict) Optional dictionary object used in PsychoPy's built-in logging system.
- **name** (str) Optional name for the PsiHandler used in PsychoPy's built-in logging system.
Raises

* `NotImplementedError` If the supplied *minVal* parameter implies an experimental design other than Yes/No or 2-AFC.

__checkFinished__() checks if we are finished. Updates attribute: *finished*

__intensityDec__() decrement the current intensity and reset counter

__intensityInc__() increment the current intensity and reset counter

__terminate__() Remove references to ourself in experiments and terminate the loop

**addData** (*result, intensity=None*)

Deprecated since 1.79.00: This function name was ambiguous. Please use one of these instead:

`.addResponse(result, intensity) .addOtherData('dataName', value')`

**addOtherData** (*dataName, value*)

Add additional data to the handler, to be tracked alongside the result data but not affecting the value of the staircase

**addResponse** (*result, intensity=None*)

Add a 1 or 0 to signify a correct / detected or incorrect / missed trial. Supplying an *intensity* value here indicates that you did not use the recommended intensity in your last trial and the staircase will replace its recorded value with the one you supplied here.

**calculateNextIntensity()**

Based on current intensity, counter of correct responses, and current direction.

**estimateLambda()**

Returns a tuple of (location, slope)

**estimateThreshold** (*thresh, lamb=None*)

Returns an intensity estimate for the provided probability.

The optional argument ‘lamb’ allows thresholds to be estimated without having to recompute the maximum likelihood lambda.

**getExp()**

Return the ExperimentHandler that this handler is attached to, if any. Returns None if not attached

**getOriginPathAndFile** (*originPath=None*)

Attempts to determine the path of the script that created this data file and returns both the path to that script and its contents. Useful to store the entire experiment with the data.

If *originPath* is provided (e.g. from Builder) then this is used otherwise the calling script is the originPath (fine from a standard python script).

**next()**

Advances to next trial and returns it.

**printAsText** (*stimOut=None, dataOut=('all_mean', 'all_std', 'all_raw'), delim='\t', matrixOnly=False*)

Exactly like saveAsText() except that the output goes to the screen instead of a file

**saveAsExcel** (*fileName, sheetName='data', matrixOnly=False, appendFile=True, fileCollisionMethod='rename'*)

Save a summary data file in Excel OpenXML format workbook (.xlsx) for processing in most spreadsheet packages. This format is compatible with versions of Excel (2007 or greater) and and with OpenOffice (>=3.0).
It has the advantage over the simpler text files (see TrialHandler.saveAsText() ) that data can be stored in multiple named sheets within the file. So you could have a single file named after your experiment and then have one worksheet for each participant. Or you could have one file for each participant and then multiple sheets for repeated sessions etc.

The file extension .xlsx will be added if not given already.

The file will contain a set of values specifying the staircase level ('intensity') at each reversal, a list of reversal indices (trial numbers), the raw staircase / intensity level on every trial and the corresponding responses of the participant on every trial.

Parameters

- **fileName**: string the name of the file to create or append. Can include relative or absolute path.
- **sheetName**: string the name of the worksheet within the file
- **matrixOnly**: True or False If set to True then only the data itself will be output (no additional info)
- **appendFile**: True or False If False any existing file with this name will be overwritten. If True then a new worksheet will be appended. If a worksheet already exists with that name a number will be added to make it unique.
- **fileCollisionMethod**: string Collision method passed to handleFileCollision() This is ignored if appendFile is True.

saveAsJson (fileName=None, encoding='utf-8-sig', fileCollisionMethod='rename')

Serialize the object to the JSON format.

Parameters

- **fileName** (string, or None) – the name of the file to create or append. Can include a relative or absolute path. If None, will not write to a file, but return an in-memory JSON object.
- **encoding** (string, optional) – The encoding to use when writing the file.
- **fileCollisionMethod** (string) – Collision method passed to handleFileCollision(). Can be either of 'rename', 'overwrite', or 'fail'.

Notes

Currently, a copy of the object is created, and the copy's .origin attribute is set to an empty string before serializing because loading the created JSON file would sometimes fail otherwise.

saveAsPickle (fileName, fileCollisionMethod='rename')

Basically just saves a copy of self (with data) to a pickle file.

This can be reloaded if necessary and further analyses carried out.

Parameters fileCollisionMethod: Collision method passed to handleFileCollision()

saveAsText (fileName, delimiter=None, matrixOnly=False, fileCollisionMethod='rename', encoding='utf-8-sig')

Write a text file with the data

Parameters

- **fileName**: a string The name of the file, including path if needed. The extension .tsv will be added if not included.
**QuestHandler**

Class that implements the Quest algorithm for quick measurement of psychophysical thresholds.

Uses Andrew Straw's QUEST, which is a Python port of Denis Pelli's Matlab code.

Measures threshold using a Weibull psychometric function. Currently, it is not possible to use a different psychometric function.

The Weibull psychometric function is given by the formula

\[
\Psi(x) = \delta \gamma + (1 - \delta)[1 - (1 - \gamma) \exp(-10^{\beta(x - T + \epsilon)})]
\]

Here, \(x\) is an intensity or a contrast (in log10 units), and \(T\) is estimated threshold.

Quest internally shifts the psychometric function such that intensity at the user-specified threshold performance level \(p\text{Threshold}\) (e.g., 50% in a yes-no or 75% in a 2-AFC task) is equal to 0. The parameter \(\epsilon\) is responsible for this shift, and is determined automatically based on the specified \(p\text{Threshold}\) value. It is the parameter Watson & Pelli (1983) introduced to perform measurements at the "optimal sweat factor". Assuming your QuestHandler instance is called \(q\), you can retrieve this value via \(q.\epsilon\) ation.

Example:

```python
# setup display/window
...
# create stimulus
stimulus = visual.RadialStim(win=win, tex='sinXsin', size=1, pos=[0,0], units='deg')
...
# create staircase object
```

(continues on next page)
% trying to find out the point where subject's response is 50 / 50
% if wanted to do a 2AFC then the defaults for pThreshold and gamma
% are good. As start value, we'll use 50% contrast, with SD = 20%
staircase = data.QuestHandler(0.5, 0.2,
    pThreshold=0.63, gamma=0.01,
    nTrials=20, minVal=0, maxVal=1)
...
while thisContrast in staircase:
    # setup stimulus
    stimulus.setContrast(thisContrast)
    stimulus.draw()
    win.flip()
    core.wait(0.5)
    # get response
    ...
    # inform QUEST of the response, needed to calculate next level
    staircase.addResponse(thisResp)
...
# can now access 1 of 3 suggested threshold levels
staircase.mean()
staircase.mode()
staircase.quantile(0.5)   # gets the median

Typical values for pThreshold are:

- 0.82 which is equivalent to a 3 up 1 down standard staircase
- **0.63 which is equivalent to a 1 up 1 down standard staircase** (and might want gamma=0.01)

The variable(s) nTrials and/or stopSd must be specified.

*beta*, *delta*, and *gamma* are the parameters of the Weibull psychometric function.

**Parameters**

- **startVal:** Prior threshold estimate or your initial guess threshold.
- **startValSd:** Standard deviation of your starting guess threshold. Be generous with the sd as QUEST will have trouble finding the true threshold if it's more than one sd from your initial guess.
- **pThreshold** Your threshold criterion expressed as probability of response==1. An intensity offset is introduced into the psychometric function so that the threshold (i.e., the midpoint of the table) yields pThreshold.
- **nTrials:** *None or a number* The maximum number of trials to be conducted.
- **stopInterval:** *None or a number* The minimum 5-95% confidence interval required in the threshold estimate before stopping. If both this and nTrials is specified, whichever happens first will determine when Quest will stop.
- **method:** *'quantile', 'mean', 'mode'* The method used to determine the next threshold to test. If you want to get a specific threshold level at the end of your staircasing, please use the quantile, mean, and mode methods directly.
- **beta:** *3.5 or a number* Controls the steepness of the psychometric function.
- **delta:** *0.01 or a number* The fraction of trials on which the observer presses blindly.
- **gamma:** *0.5 or a number* The fraction of trials that will generate response 1 when intensity=-Inf.
**PsychoPy - Psychology software for Python, Release 2020.2.10**

- **grain**: *0.01 or a number*  
  The quantization of the internal table.

- **range**: *None, or a number*  
  The intensity difference between the largest and smallest intensity that the internal table can store. This interval will be centered on the initial guess tGuess. QUEST assumes that intensities outside of this range have zero prior probability (i.e., they are impossible).

- **extraInfo**: A dictionary (typically) that will be stored along with collected data using `saveAsPickle()` or `saveAsText()` methods.

- **minVal**: *None, or a number*  
  The smallest legal value for the staircase, which can be used to prevent it reaching impossible contrast values, for instance.

- **maxVal**: *None, or a number*  
  The largest legal value for the staircase, which can be used to prevent it reaching impossible contrast values, for instance.

- **staircase**: *None or StairHandler*  
  Can supply a staircase object with intensities and results. Might be useful to give the quest algorithm more information if you have it. You can also call the `importData` function directly.

  Additional keyword arguments will be ignored.

**Notes**

The additional keyword arguments `**kwargs` might for example be passed by the `MultiStairHandler`, which expects a `label` keyword for each staircase. These parameters are to be ignored by the StairHandler.

- `_checkFinished()`  
  checks if we are finished  
  Updates attribute: `finished`

- `_intensity()`  
  assigns the next intensity level

- `_intensityDec()`  
  decrement the current intensity and reset counter

- `_intensityInc()`  
  increment the current intensity and reset counter

- `_terminate()`  
  Remove reference to ourselves in experiments and terminate the loop

- **addData** *(result, intensity=None)*  
  Deprecated since 1.79.00: This function name was ambiguous. Please use one of these instead:

    .addResponse(result, intensity) .addOtherData(‘dataName’, value’)

- **addOtherData** *(dataName, value)*  
  Add additional data to the handler, to be tracked alongside the result data but not affecting the value of the staircase

- **addResponse** *(result, intensity=None)*  
  Add a 1 or 0 to signify a correct / detected or incorrect / missed trial

  Supplying an `intensity` value here indicates that you did not use the recommended intensity in your last trial and the staircase will replace its recorded value with the one you supplied here.

- **property beta**

- **calculateNextIntensity()**  
  based on current intensity and counter of correct responses

- **confInterval** *(getDifference=False)*  
  Return estimate for the 5%–95% confidence interval (CI).
Parameters

**getDifference** (bool) If True, return the width of the confidence interval (95% - 5% percentiles). If False, return an NumPy array with estimates for the 5% and 95% boundaries.

**Returns** scalar or array of length 2.

**property delta**

**property epsilon**

**property gamma**

**getExp()**

Return the ExperimentHandler that this handler is attached to, if any. Returns None if not attached

**getOriginPathAndFile**(originPath=None)

Attempts to determine the path of the script that created this data file and returns both the path to that script and its contents. Useful to store the entire experiment with the data.

If originPath is provided (e.g. from Builder) then this is used otherwise the calling script is the originPath (fine from a standard python script).

**property grain**

**importData** (intensities, results)

import some data which wasn’t previously given to the quest algorithm

**incTrials**(nNewTrials)

increase maximum number of trials Updates attribute: nTrials

**mean()**

mean of Quest posterior pdf

**mode()**

mode of Quest posterior pdf

**next()**

Advances to next trial and returns it. Updates attributes; thisTrial, thisTrialN, thisIndex, finished, intensities

If the trials have ended, calling this method will raise a StopIteration error. This can be handled with code such as:

```python
staircase = data.QuestHandler(.......)
for eachTrial in staircase:  # automatically stops when done
    # do stuff
```

or:

```python
while True:  # i.e. forever
    try:
        thisTrial = staircase.next()
    except StopIteration:  # we got a StopIteration error
        break  # break out of the forever loop
    # do stuff here for the trial
```

**printAsText**(stimOut=None, dataOut=('all_mean', 'all_std', 'all_raw'), delim='\n', matrixOnly=False)

Exactly like saveAsText() except that the output goes to the screen instead of a file

**quantile**(p=None)

quantile of Quest posterior pdf

**property range**
saveAsExcel(file=fileName, sheetName='data', matrixOnly=False, appendFile=True, fileCollisionMethod='rename')

Save a summary data file in Excel OpenXML format workbook (.xlsx) for processing in most spreadsheet packages. This format is compatible with versions of Excel (2007 or greater) and with OpenOffice (>=3.0).

It has the advantage over the simpler text files (see TrialHandler.saveAsText() ) that data can be stored in multiple named sheets within the file. So you could have a single file named after your experiment and then have one worksheet for each participant. Or you could have one file for each participant and then multiple sheets for repeated sessions etc.

The file extension .xlsx will be added if not given already.

The file will contain a set of values specifying the staircase level ('intensity') at each reversal, a list of reversal indices (trial numbers), the raw staircase / intensity level on every trial and the corresponding responses of the participant on every trial.

Parameters

- **fileName**: string the name of the file to create or append. Can include relative or absolute path.
- **sheetName**: string the name of the worksheet within the file
- **matrixOnly**: True or False If set to True then only the data itself will be output (no additional info)
- **appendFile**: True or False If False any existing file with this name will be overwritten. If True then a new worksheet will be appended. If a worksheet already exists with that name a number will be added to make it unique.
- **fileCollisionMethod**: string Collision method passed to handleFileCollision(). This is ignored if appendFile is True.

saveAsJson(file=None, encoding='utf-8-sig', fileCollisionMethod='rename')

Serialize the object to the JSON format.

Parameters

- **fileName**: (string, or None) – the name of the file to create or append. Can include a relative or absolute path. If None, will not write to a file, but return an in-memory JSON object.
- **encoding**: (string, optional) – The encoding to use when writing the file.
- **fileCollisionMethod**: (string) – Collision method passed to handleFileCollision(). Can be either of 'rename', 'overwrite', or 'fail'.

Notes

Currently, a copy of the object is created, and the copy's .origin attribute is set to an empty string before serializing because loading the created JSON file would sometimes fail otherwise.

saveAsPickle(file=fileName, fileCollisionMethod='rename')

Basically just saves a copy of self (with data) to a pickle file.

This can be reloaded if necessary and further analyses carried out.

Parameters fileCollisionMethod: Collision method passed to handleFileCollision()

saveAsText(file=fileName, delim=None, matrixOnly=False, fileCollisionMethod='rename', encoding='utf-8-sig')

Write a text file with the data
Parameters

**fileName: a string**  The name of the file, including path if needed. The extension .tsv will be added if not included.

**delim: a string**  the delimiter to be used (e.g. ‘ ‘ for tab-delimited, ‘,’ for csv files)

**matrixOnly: True/False**  If True, prevents the output of the extraInfo provided at initialisation.

**fileCollisionMethod:** Collision method passed to handleFileCollision()

**encoding:**  The encoding to use when saving a the file. Defaults to utf-8-sig.

\[sd()\]
standard deviation of Quest posterior pdf

\[setExp(exp)\]
Sets the ExperimentHandler that this handler is attached to
Do NOT attempt to set the experiment using:

```python
trials._exp = myExperiment
```
because it needs to be performed using the weakref module.

\[simulate(tActual)\]
returns a simulated user response to the next intensity level presented by Quest, need to supply the actual threshold level

### 8.7.8 QuestPlusHandler

**class psychopy.data.QuestPlusHandler**

\[nTrials, intensityVals, thresholdVals, slopeVals, lowerAsymptoteVals, lapseRateVals, responseVals=('Yes', 'No'), prior=None, startIntensity=None, psychometricFunc='weibull', stimScale='log10', stimSelectionMethod='minEntropy', stimSelectionOptions=None, paramEstimationMethod='mean', extraInfo=None, name='', label='', **kwargs\]

QUEST+ implementation. Currently only supports parameter estimation of a Weibull-shaped psychometric function.

The parameter estimates can be retrieved via the .paramEstimate attribute, which returns a dictionary whose keys correspond to the names of the estimated parameters (i.e., QuestPlusHandler.paramEstimate[‘threshold’] will provide the threshold estimate). Retrieval of the marginal posterior distributions works similarly: they can be accessed via the .posterior dictionary.

**Parameters**

- **nTrials (int)** – Number of trials to run.
- **intensityVals (collection of floats)** – The complete set of possible stimulus levels. Note that the stimulus levels are not necessarily limited to intensities (as the name of this parameter implies), but they could also be contrasts, durations, weights, etc.
- **thresholdVals (float or collection of floats)** – The complete set of possible threshold values.
• **slopeVals** *(float or collection of floats)* – The complete set of possible slope values.

• **lowerAsymptoteVals** *(float or collection of floats)* – The complete set of possible values of the lower asymptote. This corresponds to false-alarm rates in yes-no tasks, and to the guessing rate in n-AFC tasks. Therefore, when performing an n-AFC experiment, the collection should consists of a single value only (e.g., \([0.5]\) for 2-AFC, \([0.33]\) for 3-AFC, \([0.25]\) for 4-AFC, etc.).

• **lapseRateVals** *(float or collection of floats)* – The complete set of possible lapse rate values. The lapse rate defines the upper asymptote of the psychometric function, which will be at \(1 - \text{lapse rate}\).

• **responseVals** *(collection)* – The complete set of possible response outcomes. Currently, only two outcomes are supported: the first element must correspond to a successful response / stimulus detection, and the second one to an unsuccessful or incorrect response. For example, in a yes-no task, one would use \(['Yes', 'No']\), and in an n-AFC task, \(['Correct', 'Incorrect']\); or, alternatively, the less verbose \([1, 0]\) in both cases.

• **prior** *(dict of floats)* – The prior probabilities to assign to the parameter values. The dictionary keys correspond to the respective parameters: threshold, slope, lowerAsymptote, lapseRate.

• **startIntensity** *(float)* – The very first intensity (or stimulus level) to present.

• **psychometricFunc** *(\{'weibull'\})* – The psychometric function to fit. Currently, only the Weibull function is supported.

• **stimScale** *(\{'log10', 'dB', 'linear'\})* – The scale on which the stimulus intensities (or stimulus levels) are provided. Currently supported are the decadic logarithm, \(\log10\); decibels, \(dB\); and a linear scale, \(linear\).

• **stimSelectionMethod** *(\{'minEntropy', 'minNEntropy'\})* – How to select the next stimulus. \(\text{minEntropy}\) will select the stimulus that will minimize the expected entropy. \(\text{minNEntropy}\) will randomly pick pick a stimulus from the set of stimuli that will produce the smallest, 2nd-smallest, ..., N-smallest entropy. This can be used to ensure some variation in the stimulus selection (and subsequent presentation) procedure. The number \(N\) will then have to be specified via the **stimSelectionOption** parameter.

• **stimSelectionOptions** *(dict)* – This parameter further controls how to select the next stimulus in case **stimSelectionMethod=**\(\text{minNEntropy}\). The dictionary supports two keys: \(N\) and \(\text{maxConsecutiveReps}\). \(N\) defines the number of “best” stimuli (i.e., those which produce the smallest \(N\) expected entropies) from which to randomly select a stimulus for presentation in the next trial. \(\text{maxConsecutiveReps}\) defines how many times the exact same stimulus can be presented on consecutive trials. For example, to randomly pick a stimulus from those which will produce the \(4\) smallest expected entropies, and to allow the same stimulus to be presented on two consecutive trials \(\text{max}\), use **stimSelectionOptions=dict(N=4, maxConsecutiveReps=2)**. To achieve reproducible results, you may pass a seed to the random number generator via the **randomSeed** key.

• **paramEstimationMethod** *(\{'mean', 'mode'\})* – How to calculate the final parameter estimate. \(\text{mean}\) returns the mean of each parameter, weighted by their respective posterior probabilities. \(\text{mode}\) returns the the parameters at the peak of the posterior distribution.

• **extraInfo** *(dict)* – Additional information to store along the actual QUEST+ staircase data.

• **name** *(str)* – The name of the QUEST+ staircase object. This will appear in the PsychoPy logs.
• **label** (*str*) – Only used by `MultiStairHandler`, and otherwise ignored.

• **kwargs** (*dict*) – Additional keyword arguments. These might be passed, for example, through a `MultiStairHandler`, and will be ignored. A warning will be emitted whenever additional keyword arguments have been passed.

**Warns** `RuntimeWarning` – If an unknown keyword argument was passed.

**Notes**

The QUEST+ algorithm was first described by\(^1\).

```python
_intensityDec()
```

decrement the current intensity and reset counter

```python
_intensityInc()
```

increment the current intensity and reset counter

```python
_terminate()
```

Remove references to ourselves in experiments and terminate the loop

```python
addData (result, intensity=None)
```

Deprecated since 1.79.00: This function name was ambiguous. Please use one of these instead:

```python
...addResponse(result, intensity) .addOtherData('dataName', value')
```

```python
addOtherData (dataName, value)
```

Add additional data to the handler, to be tracked alongside the result data but not affecting the value of the staircase

```python
addResponse (response, intensity=None)
```

Add a 1 or 0 to signify a correct / detected or incorrect / missed trial.

This is essential to advance the staircase to a new intensity level!

Supplying an `intensity` value here indicates that you did not use the recommended intensity in your last trial and the staircase will replace its recorded value with the one you supplied here.

```python
calculateNextIntensity()
```

Based on current intensity, counter of correct responses, and current direction.

```python
getExp()
```

Return the ExperimentHandler that this handler is attached to, if any. Returns None if not attached

```python
getOriginPathAndFile (originPath=None)
```

Attempts to determine the path of the script that created this data file and returns both the path to that script and its contents. Useful to store the entire experiment with the data.

If `originPath` is provided (e.g. from Builder) then this is used otherwise the calling script is the `originPath` (fine from a standard python script).

```python
next()
```

Advances to next trial and returns it. Updates attributes; `thisTrial`, `thisTrialN` and `thisIndex`.

If the trials have ended, calling this method will raise a StopIteration error. This can be handled with code such as:

staircase = data.StairHandler(.......)
for eachTrial in staircase:  # automatically stops when done
    # do stuff

or:

staircase = data.StairHandler(.......)
while True:  # ie forever
    try:
        thisTrial = staircase.next()
    except StopIteration:  # we got a StopIteration error
        break  # break out of the forever loop
    # do stuff here for the trial

property paramEstimate
The estimated parameters of the psychometric function.

    Returns  A dictionary whose keys correspond to the names of the estimated parameters.

    Return type  dict of floats

property posterior
The marginal posterior distributions.

    Returns  A dictionary whose keys correspond to the names of the estimated parameters.

    Return type  dict of np.ndarrays

printAsText (stimOut=None, dataOut=('all_mean', 'all_std', 'all_raw'), delim='\t', matrixOnly=False)
Exactly like saveAsText() except that the output goes to the screen instead of a file

property prior
The marginal prior distributions.

    Returns  A dictionary whose keys correspond to the names of the parameters.

    Return type  dict of np.ndarrays

saveAsExcel (fileName, sheetName='data', matrixOnly=False, appendFile=True, fileCollision- Method='rename')
Save a summary data file in Excel OpenXML format workbook (.xlsx) for processing in most spreadsheet packages. This format is compatible with versions of Excel (2007 or greater) and and with OpenOffice (>=3.0).

It has the advantage over the simpler text files (see TrialHandler.saveAsText() ) that data can be stored in multiple named sheets within the file. So you could have a single file named after your experiment and then have one worksheet for each participant. Or you could have one file for each participant and then multiple sheets for repeated sessions etc.

The file extension .xlsx will be added if not given already.

The file will contain a set of values specifying the staircase level ('intensity') at each reversal, a list of reversal indices (trial numbers), the raw staircase / intensity level on every trial and the corresponding responses of the participant on every trial.

    Parameters

        fileName: string  the name of the file to create or append. Can include relative or absolute path.

        sheetName: string  the name of the worksheet within the file
matrixOnly: True or False  If set to True then only the data itself will be output (no additional info)

appendFile: True or False  If False any existing file with this name will be overwritten. If True then a new worksheet will be appended. If a worksheet already exists with that name a number will be added to make it unique.

fileCollisionMethod: string  Collision method passed to handleFileCollision()  This is ignored if appendFile is True.

saveAsJson (fileName=None, encoding='utf-8-sig', fileCollisionMethod='rename')  Serializes the object to the JSON format.

Parameters

- **fileName** (string, or None) – the name of the file to create or append. Can include a relative or absolute path. If None, will not write to a file, but return an in-memory JSON object.
- **encoding** (string, optional) – The encoding to use when writing the file.
- **fileCollisionMethod** (string) – Collision method passed to handleFileCollision(). Can be either of ‘rename’, ‘overwrite’, or ‘fail’.

Notes

Currently, a copy of the object is created, and the copy’s .origin attribute is set to an empty string before serializing because loading the created JSON file would sometimes fail otherwise.

saveAsPickle (fileName, fileCollisionMethod='rename')  Basically just saves a copy of self (with data) to a pickle file.

This can be reloaded if necessary and further analyses carried out.

Parameters  fileCollisionMethod: Collision method passed to handleFileCollision()

saveAsText (fileName, delim=None, matrixOnly=False, fileCollisionMethod='rename', encoding='utf-8-sig')  Write a text file with the data

Parameters

- **fileName**: a string  The name of the file, including path if needed. The extension .tsv will be added if not included.
- **delim**: a string  the delimiter to be used (e.g. ‘ ‘ for tab-delimited, ‘,’ for csv files)
- **matrixOnly**: True/False  If True, prevents the output of the extraInfo provided at initialisation.
- **fileCollisionMethod**: Collision method passed to handleFileCollision()
- **encoding**: The encoding to use when saving a the file. Defaults to utf-8-sig.

setExp (exp)  Sets the ExperimentHandler that this handler is attached to

Do NOT attempt to set the experiment using:

```python
trials._exp = myExperiment
```

because it needs to be performed using the weakref module.

property startIntensity
8.7.9 MultiStairHandler

class psychopy.data.MultiStairHandler(stairType='simple', method='random', conditions=None, nTrials=50, randomSeed=None, originPath=None, name='', autoLog=True)

A Handler to allow easy interleaved staircase procedures (simple or QUEST).

Parameters for the staircases, as used by the relevant StairHandler or QuestHandler (e.g. the startVal, minVal, maxVal...) should be specified in the conditions list and may vary between each staircase. In particular, the conditions must include a startVal (because this is a required argument to the above handlers), a label to tag the staircase and a startValSd (only for QUEST staircases). Any parameters not specified in the conditions file will revert to the default for that individual handler.

If you need to customize the behaviour further you may want to look at the recipe on Coder - interleave staircases.

Params

stairType: ‘simple’, ‘quest’, or ‘questplus’

Use a StairHandler, a QuestHandler, or a QuestPlusHandler.

method: ‘random’, ‘fullRandom’, or ‘sequential’ If random, stairs are shuffled in each repeat but not randomized more than that (so you can’t have 3 repeats of the same staircase in a row unless it’s the only one still running). If fullRandom, the staircase order is “fully” randomized, meaning that, theoretically, a large number of subsequent trials could invoke the same staircase repeatedly. If sequential, don’t perform any randomization.

conditions: a list of dictionaries specifying conditions Can be used to control parameters for the different staircases. Can be imported from an Excel file using psychopy.data.importConditions MUST include keys providing, ‘startVal’, ‘label’ and ‘startValSd’ (QUEST only). The ‘label’ will be used in data file saving so should be unique. See Example Usage below.

nTrials=50 Minimum trials to run (but may take more if the staircase hasn’t also met its minimal reversals. See StairHandler

randomSeed [int or None] The seed with which to initialize the random number generator (RNG). If None (default), do not initialize the RNG with a specific value.

Example usage:

```python
conditions=[
    {'label': 'low', 'startVal': 0.1, 'ori': 45},
    {'label': 'high', 'startVal': 0.8, 'ori': 45},
    {'label': 'low', 'startVal': 0.1, 'ori': 90},
    {'label': 'high', 'startVal': 0.8, 'ori': 90},
]
stairs = data.MultiStairHandler(conditions=conditions, nTrials=50)
for thisIntensity, thisCondition in stairs:
    thisOri = thisCondition['ori']
    # do something with thisIntensity and thisOri
    stairs.addResponse(correctIncorrect)  # this is ESSENTIAL

# save data as multiple formats
stairs.saveDataAsExcel(fileName)  # easy to browse
stairs.saveAsPickle(fileName)  # contains more info
```
Raises `ValueError` – If an unknown randomization option was passed via the `method` keyword argument.

__startNewPass()__
Create a new iteration of the running staircases for this pass.

This is not normally needed by the user - it gets called at `__init__` and every time that `next()` runs out of trials for this pass.

__terminate()__
Remove references to ourself in experiments and terminate the loop

`addData(result, intensity=None)`
Deprecated 1.79.00: It was ambiguous whether you were adding the response (0 or 1) or some other data concerning the trial so there is now a pair of explicit methods:

```
addResponse(corr, intensity) # some data that alters the next trial value
addOtherData('RT', reactionTime) # some other data that won't control staircase
```

`addOtherData(name, value)`
Add some data about the current trial that will not be used to control the staircase(s) such as reaction time data

`addResponse(result, intensity=None)`
Add a 1 or 0 to signify a correct/detected or incorrect/missed trial

This is essential to advance the staircase to a new intensity level!

`getExp()`
Return the ExperimentHandler that this handler is attached to, if any. Returns None if not attached

`getOriginPathAndFile(originPath=None)`
Attempts to determine the path of the script that created this data file and returns both the path to that script and its contents. Useful to store the entire experiment with the data.

If `originPath` is provided (e.g. from Builder) then this is used otherwise the calling script is the `originPath` (fine from a standard python script).

`next()`
Advances to next trial and returns it.

This can be handled with code such as:

```python
staircase = data.MultiStairHandler(........)
for eachTrial in staircase:  # automatically stops when done
    # do stuff here for the trial
```

or:

```python
staircase = data.MultiStairHandler(........)
while True:  # ie forever
    try:  
        thisTrial = staircase.next()
    except StopIteration:  # we got a StopIteration error
        break  # break out of the forever loop
    # do stuff here for the trial
```

`printAsText(delim='\t', matrixOnly=False)`
Write the data to the standard output stream

Parameters
delim: a string  the delimitter to be used (e.g. ‘‘ for tab-delimited, ‘,’ for csv files)

matrixOnly: True/False  If True, prevents the output of the extraInfo provided at initialisa-

saveAsExcel (fileName, matrixOnly=False, appendFile=False, fileCollisionMethod='rename')
Save a summary data file in Excel OpenXML format workbook (.xlsx) for processing in most spreadsheet
packages. This format is compatible with versions of Excel (2007 or greater) and with OpenOffice (>=3.0).
It has the advantage over the simpler text files (see TrialHandler.saveAsText()) that the data from
each staircase will be save in the same file, with the sheet name coming from the ‘label’ given in the dictionary
of conditions during initialisation of the Handler.
The file extension .xlsx will be added if not given already.
The file will contain a set of values specifying the staircase level (‘intensity’) at each reversal, a list of reversal
indices (trial numbers), the raw staircase/intensity level on every trial and the corresponding responses of the
participant on every trial.

Parameters

fileName: string  the name of the file to create or append. Can include relative or absolute
path

matrixOnly: True or False  If set to True then only the data itself will be output (no additional
info)

appendFile: True or False  If False any existing file with this name will be overwritten. If
True then a new worksheet will be appended. If a worksheet already exists with that name a
number will be added to make it unique.

fileCollisionMethod: string  Collision method passed to handleFileCollision()
This is ignored if append is True.

saveAsJson (fileName=None, encoding='utf-8-sig', fileCollisionMethod='rename')
Serialize the object to the JSON format.

Parameters

• fileName (string, or None) – the name of the file to create or append. Can include
a relative or absolute path. If None, will not write to a file, but return an in-memory JSON
object.

• encoding (string, optional) – The encoding to use when writing the file.

• fileCollisionMethod (string) – Collision method passed to handleFileCol-
ission(). Can be either of ‘rename’, ‘overwrite’, or ‘fail’.

Notes

Currently, a copy of the object is created, and the copy’s .origin attribute is set to an empty string before
serializing because loading the created JSON file would sometimes fail otherwise.

saveAsPickle (fileName, fileCollisionMethod='rename')
Saves a copy of self (with data) to a pickle file.
This can be reloaded later and further analyses carried out.

Parameters  fileCollisionMethod: Collision method passed to handleFileCollision()

saveAsText (fileName, delim=None, matrixOnly=False, fileCollisionMethod='rename', encoding='utf-8-
sig')
Write out text files with the data.
For MultiStairHandler this will output one file for each staircase that was run, with _label added to the file-name that you specify above (label comes from the condition dictionary you specified when you created the Handler).

**Parameters**

- **fileName**: a string The name of the file, including path if needed. The extension `.tsv` will be added if not included.
- **delim**: a string the delimiter to be used (e.g. `'` for tab-delimited, `','` for csv files)
- **matrixOnly**: True/False If True, prevents the output of the `extraInfo` provided at initialisation.
- **fileCollisionMethod**: Collision method passed to `handleFileCollision()`
- **encoding**: The encoding to use when saving a the file. Defaults to `utf-8-sig`.

**setExp** *(exp)*

Sets the ExperimentHandler that this handler is attached to

Do NOT attempt to set the experiment using:

```python
trials._exp = myExperiment
```

because it needs to be performed using the `weakref` module.

### 8.7.10 FitWeibull

**class** `psychopy.data.FitWeibull`(xx, yy, *sens=1.0, guess=None, display=1, expectedMin=0.5, *optimize_kws=None)*

Fit a Weibull function (either 2AFC or YN) of the form:

\[
y = \text{chance} + (1.0 - \text{chance}) \times (1 - \exp(-\frac{xx}{\alpha}\times\beta))
\]

and with inverse:

\[
x = \alpha \times (-\log(\frac{(1.0-y)}{(1-\text{chance}))}) \times \frac{1.0}{\beta}
\]

After fitting the function you can evaluate an array of x-values with `fit.eval(x)`, retrieve the inverse of the function with `fit.inverse(y)` or retrieve the parameters from `fit.params` (a list with `[alpha, beta]`)

**_doFit** ()

The Fit class that derives this needs to specify its `_evalFunction`

**eval** *(xx, params=None)*

Evaluate xx for the current parameters of the model, or for arbitrary params if these are given.

**inverse** *(yy, params=None)*

Evaluate yy for the current parameters of the model, or for arbitrary params if these are given.
8.7.11 FitLogistic

```python
class psychopy.data.FitLogistic(xx, yy, sens=1.0, guess=None, display=1, expectedMin=0.5, optimize_kws=None):
    Fit a Logistic function (either 2AFC or YN) of the form:
    \[
    y = \text{chance} + \frac{(1-\text{chance})}{1 + \exp((\text{PSE}-\text{xx}) \times \text{JND})}
    \]
    and with inverse:
    \[
    x = \text{PSE} - \text{log}(\frac{(1-\text{chance})}{\text{yy}-\text{chance}} - 1)/\text{JND}
    \]
    After fitting the function you can evaluate an array of x-values with `fit.eval(x)`, retrieve the inverse of the
    function with `fit.inverse(y)` or retrieve the parameters from `fit.params` (a list with `[PSE, JND]`)
```

8.7.12 FitNakaRushton

```python
class psychopy.data.FitNakaRushton(xx, yy, sens=1.0, guess=None, display=1, expectedMin=0.5, optimize_kws=None):
    Fit a Naka-Rushton function of the form:
    \[
    \text{yy} = \text{rMin} + (\text{rMax}-\text{rMin}) \times \frac{\text{xx}^n}{(\text{xx}^n+\text{c50}^n)}
    \]
    After fitting the function you can evaluate an array of x-values with `fit.eval(x)`, retrieve the inverse of the
    function with `fit.inverse(y)` or retrieve the parameters from `fit.params` (a list with `[rMin, rMax, c50, n]`)
    Note that this differs from most of the other functions in not using a value for the expected minimum. Rather, it
    fits this as one of the parameters of the model.
```

8.7. psychopy.data - functions for storing/saving/analysing data

525
8.7.13 FitCumNormal

class psychopy.data.FitCumNormal (xx, yy, sems=1.0, guess=None, display=1, expectedMin=0.5, optimize_kws=None)

Fit a Cumulative Normal function (aka error function or erf) of the form:

\[ y = \text{chance} + (1-\text{chance}) \times \left( \frac{\text{special.errf}((xx-xShift)/(\sqrt{2})*sd)}{2} + 1 \right) \times 0.5 \]

and with inverse:

\[ x = xShift + \sqrt{2} \times sd \times \text{erfinv}((yy-chance)/(1-chance)-0.5) \times 2) \]

After fitting the function you can evaluate an array of x-values with fit.eval(x), retrieve the inverse of the function with fit.inverse(y) or retrieve the parameters from fit.params (a list with [centre, sd] for the Gaussian distribution forming the cumulative).

NB: Prior to version 1.74 the parameters had different meaning, relating to xShift and slope of the function (similar to 1/sd). Although that is more in with the parameters for the Weibull fit, for instance, it is less in keeping with standard expectations of normal (Gaussian distributions) so in version 1.74.00 the parameters became the [centre, sd] of the normal distribution.

_doFit ()

The Fit class that derives this needs to specify its _evalFunction
eval (xx, params=None)

Evaluate xx for the current parameters of the model, or for arbitrary params if these are given.
iinverse (yy, params=None)

Evaluate yy for the current parameters of the model, or for arbitrary params if these are given.

8.7.14 importConditions ()

psychopy.data.importConditions (fileName, returnFieldNames=False, selection="")

Imports a list of conditions from an .xlsx, .csv, or .pkl file.

The output is suitable as an input to TrialHandler trialList or to MultiStairHandler as a conditions list.

If fileName ends with:

- .csv: import as a comma-separated-value file (header + row x col)
- .xlsx: import as Excel 2007 (xlsx) files. No support for older (.xls) is planned.
- .pkl: import from a pickle file as list of lists (header + row x col)

The file should contain one row per type of trial needed and one column for each parameter that defines the trial type. The first row should give parameter names, which should:

- be unique
- begin with a letter (upper or lower case)
- contain no spaces or other punctuation (underscores are permitted)

selection is used to select a subset of condition indices to be used. It can be a list/array of indices, a python slice object or a string to be parsed as either option e.g.:

- “1,2,4” or [1,2,4] or (1,2,4) are the same
- “2:5” # 2, 3, 4 (doesn’t include last whole value)
- “-10:2:” # tenth from last to the last in steps of 2
• slice(-10, 2, None) # the same as above
• random(5) * 8 # five random vals 0-7

8.7.15 functionFromStaircase()

`psychopy.data.functionFromStaircase(intensities, responses, bins=10)`
Create a psychometric function by binning data from a staircase procedure. Although the default is 10 bins Jon now always uses ‘unique’ bins (fewer bins looks pretty but leads to errors in slope estimation)

Usage:

```python
intensity, meanCorrect, n = functionFromStaircase(intensities, responses, bins)
```

where:
- `intensities` are a list (or array) of intensities to be binned
- `responses` are a list of 0,1 each corresponding to the equivalent intensity value
- `bins` can be an integer (giving that number of bins) or ‘unique’ (each bin is made from aa data for exactly one intensity value)
- `intensity` a numpy array of intensity values (where each is the center of an intensity bin)
- `meanCorrect` a numpy array of mean % correct in each bin
- `n` a numpy array of number of responses contributing to each mean

8.7.16 bootStraps()

`psychopy.data.bootStraps(dat, n=1)`
Create a list of n bootstrapped resamples of the data

SLOW IMPLEMENTATION (Python for-loop)

Usage: `out = bootStraps(dat, n=1)`

Where:

- `dat` an NxM or 1xN array (each row is a different condition, each column is a different trial)
- `n` number of bootstrapped resamples to create

out
- `dim[0]=conditions`
- `dim[1]=trials`
- `dim[2]=resamples`
8.8 Encryption

Some labs may wish to better protect their data from casual inspection or accidental disclosure. This is possible within PsychoPy using a separate python package, pyFileSec, which grew out of PsychoPy. pyFileSec is distributed with the StandAlone versions of PsychoPy, or can be installed using pip or easy_install via https://pypi.python.org/pypi/PyFileSec/

Some elaboration of pyFileSec usage and security strategy can be found here: http://pythonhosted.org/PyFileSec

Basic usage is illustrated in the Coder demo > misc > encrypt_data.py

8.9 psychopy.event - for keypresses and mouse clicks

class psychopy.event.Mouse (visible=True, newPos=None, win=None)

Easy way to track what your mouse is doing.

It needn’t be a class, but since Joystick works better as a class this may as well be one too for consistency

Create your visual.Window before creating a Mouse.

Parameters

visible [True or False] makes the mouse invisible if necessary

newPos [None or [x,y]] gives the mouse a particular starting position (pygame Window only)

win [None or Window] the window to which this mouse is attached (the first found if None provided)

clickReset (buttons=(0, 1, 2))

Reset a 3-item list of core.Clocks use in timing button clicks.

The pyglet mouse-button-pressed handler uses their clock.getLastResetTime() when a button is pressed so the user can reset them at stimulus onset or offset to measure RT. The default is to reset all, but they can be reset individually as specified in buttons list

getPos ()

Returns the current position of the mouse, in the same units as the Window (0,0) is at centre

getPressed (getTime=False)

Returns a 3-item list indicating whether or not buttons 0,1,2 are currently pressed.

If getTime=True (False by default) then getPressed will return all buttons that have been pressed since the last call to mouse.clickReset as well as their time stamps:

buttons = mouse.getPressed()
buttons, times = mouse.getPressed(getTime=True)

Typically you want to call mouse.clickReset() at stimulus onset, then after the button is pressed in reaction to it, the total time elapsed from the last reset to click is in mouseTimes. This is the actual RT, regardless of when the call to getPressed() was made.

getRel ()

Returns the new position of the mouse relative to the last call to getRel or getPos, in the same units as the Window.

getVisible ()

Gets the visibility of the mouse (1 or 0)
getWheelRel()
Returns the travel of the mouse scroll wheel since last call. Returns a numpy.array(x,y) but for most wheels y is the only value that will change (except Mac mighty mice?)

isPressedIn (shape, buttons=(0, 1, 2))
Returns True if the mouse is currently inside the shape and one of the mouse buttons is pressed. The default is that any of the 3 buttons can indicate a click; for only a left-click, specify buttons=[0]:

```python
if mouse.isPressedIn(shape):
    if mouse.isPressedIn(shape, buttons=[0]):  # left-clicks only
```

Ideally, shape can be anything that has a .contains() method, like ShapeStim or Polygon. Not tested with ImageStim.

mouseMoveTime()
mouseMoved (distance=None, reset=False)
Determine whether/how far the mouse has moved.

With no args returns true if mouse has moved at all since last getPos() call, or distance (x,y) can be set to pos or neg distances from x and y to see if moved either x or y that far from lastPos, or distance can be an int/float to test if new coordinates are more than that far in a straight line from old coords.

Retrieve time of last movement from self.mouseClock.getTime().

Reset can be to ‘here’ or to screen coords (x,y) which allows measuring distance from there to mouse when moved. If reset is (x,y) and distance is set, then prevPos is set to (x,y) and distance from (x,y) to here is checked, mouse.lastPos is set as current (x,y) by getPos(), mouse.prevPos holds lastPos from last time mouseMoved was called.

setExclusive (exclusivity)
Binds the mouse to the experiment window. Only works in Pyglet.

In multi-monitor settings, or with a window that is not fullscreen, the mouse pointer can drift, and thereby PsychoPy might not get the events from that window. setExclusive(True) works with Pyglet to bind the mouse to the experiment window.

Note that binding the mouse pointer to a window will cause the pointer to vanish, and absolute positions will no longer be meaningful getPos() returns [0, 0] in this case.

setPos (newPos=(0, 0))
Sets the current position of the mouse, in the same units as the Window. (0,0) is the center.

Parameters

newPos [(x,y) or [x,y]] the new position on the screen

setVisible (visible)
Sets the visibility of the mouse to 1 or 0

NB when the mouse is not visible its absolute position is held at (0, 0) to prevent it from going off the screen and getting lost! You can still use getRel() in that case.

property units
The units for this mouse (will match the current units for the Window it lives in)

psychopy.event.clearEvents (eventType=None)
Clears all events currently in the event buffer.

Optional argument, eventType, specifies only certain types to be cleared.

Parameters
eventType [None, ‘mouse’, ‘joystick’, ‘keyboard’] If this is not None then only events of the given type are cleared

psychopy.event.waitKeys(maxWait=inf, keyList=None, modifiers=False, timeStamped=False, clearEvents=True)
Same as ~psychopy.event.getKeys, but halts everything (including drawing) while awaiting input from keyboard.

Parameters

maxWait [any numeric value.] Maximum number of seconds period and which keys to wait for. Default is float("inf") which simply waits forever.

keyList [None or []] Allows the user to specify a set of keys to check for. Only keypresses from this set of keys will be removed from the keyboard buffer. If the keyList is None, all keys will be checked and the key buffer will be cleared completely. NB, pygame doesn’t return timestamps (they are always 0)

modifiers [False or True] If True will return a list of tuples instead of a list of keynames. Each tuple has (keyname, modifiers). The modifiers are a dict of keyboard modifier flags keyed by the modifier name (eg. ‘shift’, ‘ctrl’).

timeStamped [False, True, or Clock] If True will return a list of tuples instead of a list of keynames. Each tuple has (keyname, time). If a core.Clock is given then the time will be relative to the Clock’s last reset.

clearEvents [True or False] Whether to clear the keyboard event buffer (and discard preceding keypresses) before starting to monitor for new keypresses.

Returns None if times out.

psychopy.event.getKeys(keyList=None, modifiers=False, timeStamped=False)
Returns a list of keys that were pressed.

Parameters

keyList [None or []] Allows the user to specify a set of keys to check for. Only keypresses from this set of keys will be removed from the keyboard buffer. If the keyList is None, all keys will be checked and the key buffer will be cleared completely. NB, pygame doesn’t return timestamps (they are always 0)

modifiers [False or True] If True will return a list of tuples instead of a list of keynames. Each tuple has (keyname, modifiers). The modifiers are a dict of keyboard modifier flags keyed by the modifier name (eg. ‘shift’, ‘ctrl’).

timeStamped [False, True, or Clock] If True will return a list of tuples instead of a list of keynames. Each tuple has (keyname, time). If a core.Clock is given then the time will be relative to the Clock’s last reset.

Author

• 2003 written by Jon Peirce
• 2009 keyList functionality added by Gary Strangman
• 2009 timeStamped code provided by Dave Britton
• 2016 modifiers code provided by 5AM Solutions

psychopy.event.xydist(p1=(0.0, 0.0), p2=(0.0, 0.0))
Helper function returning the cartesian distance between p1 and p2
8.10 psychopy.filters - helper functions for creating filters

This module has moved to psychopy.visual.filters but you can still (currently) import it as psychopy.filters

8.11 psychopy.gui - create dialogue boxes

8.11.1 DlgFromDict

class psychopy.gui.DlgFromDict(dictionary, title='', fixed=None, order=None, tip=None, screen=-1, sortKeys=True, copyDict=False, labels=None, show=True, sort_keys=None, copy_dict=None)

Creates a dialogue box that represents a dictionary of values. Any values changed by the user are change (in-place) by this dialogue box.

Parameters

- **dictionary (dict)** – A dictionary defining the input fields (keys) and pre-filled values (values) for the user dialog
- **title (str)** – The title of the dialog window
- **labels (dict)** – A dictionary defining labels (values) to be displayed instead of key strings (keys) defined in "dictionary". Not all keys in "dictionary" need to be contained in labels.
- **fixed (list)** – A list of keys for which the values shall be displayed in non-editable fields
- **order (list)** – A list of keys defining the display order of keys in "dictionary". If not all keys in "dictionary" are contained in "order", those will appear in random order after all ordered keys.
- **tip (list)** – A dictionary assigning tooltips to the keys
- **screen (int)** – Screen number where the Dialog is displayed. If -1, the Dialog will be displayed on the primary screen.
- **sortKeys (bool)** – A boolean flag indicating that keys are to be sorted alphabetically.
- **copyDict (bool)** – If False, modify dictionary in-place. If True, a copy of the dictionary is created, and the altered version (after user interaction) can be retrieved from :attr:`psychopy.gui.DlgFromDict.dictionary`.
- **labels** – A dictionary defining labels (dict values) to be displayed instead of key strings (dict keys) defined in "dictionary". Not all keys in "dictionary" need to be contained in labels.
- **show (bool)** – Whether to immediately display the dialog upon instantiation. If False, it can be displayed at a later time by calling its show() method.

* e.g. *

: param :::

    info = {'Observer': 'jwp', 'GratingOri': 45, 'ExpVersion': 1.1, 'Group': ['Test', 'Control']}
    infoDlg = gui.DlgFromDict(dictionary=info, title='TestExperiment', fixed=['ExpVersion'])
    if infoDlg.OK:
        print(info)
    else:
        print('User Cancelled')
Parameters

- the code above (In) –
- contents of info will be updated to the values (the) –
- by the dialogue box. (returned) –
- the user cancels (rather than pressing OK) (If) –

:param : :param then the dictionary remains unchanged. If you want to check whether :param the user hit OK: :param then check whether DlgFromDict.OK equals :param True or False: :param See GUI.py for a usage demo: :param including order and tip (tooltip).:

show()
Display the dialog.

8.11.2 Dlg

class psychopy.gui.Dlg(title='PsychoPy Dialog', pos=None, size=None, style=None, labelButtonOK='OK', labelButtonCancel='Cancel', screen=-1)
A simple dialogue box. You can add text or input boxes (sequentially) and then retrieve the values.

see also the function dlgFromDict for an even simpler version

Example

```python
from psychopy import gui
myDlg = gui.Dlg(title="JWP's experiment")
myDlg.addText('Subject info')
myDlg.addField('Name:')
myDlg.addField('Age:', 21)
myDlg.addText('Experiment Info')
myDlg.addField('Grating Ori:', 45)
myDlg.addField('Group:', choices=['Test', 'Control'])
ok_data = myDlg.show() # show dialog and wait for OK or Cancel
if myDlg.OK: # or if ok_data is not None
    print(ok_data)
else:
    print('user cancelled')
```

addField (label="", initial="", color="", choices=None, tip="", enabled=True)
Add a (labelled) input field to the dialogue box, optional text color and tooltip.
If `initial` is a bool, a checkbox will be created. If `choices` is a list or tuple, a dropdown selector is created. Otherwise, a text line entry box is created.

Returns a handle to the field (but not to the label).

addFixedField (label="", initial="", color="", choices=None, tip="")
Add a field to the dialog box (like addField) but the field cannot be edited. e.g. Display experiment version.

show ()
Presents the dialog and waits for the user to press OK or CANCEL.
If user presses OK button, function returns a list containing the updated values coming from each of the input fields created. Otherwise, None is returned.

Returns self.data
8.11.3 fileOpenDlg()

psychopy.gui.fileOpenDlg(tryFilePath='', tryFileName='', prompt='Select file to open', allowed=None)
A simple dialogue allowing read access to the file system.

Parameters

tryFilePath: string default file path on which to open the dialog

tryFileName: string default file name, as suggested file

prompt: string (default “Select file to open”) can be set to custom prompts

allowed: string (available since v1.62.01) a string to specify file filters. e.g. “Text files (*.txt);; Image files (*.bmp *.gif)” See http://pyqt.sourceforge.net/Docs/PyQt4/qfiledialog.html #getOpenFileNames for further details

If tryFilePath or tryFileName are empty or invalid then current path and empty names are used to start search.
If user cancels, then None is returned.

8.11.4 fileSaveDlg()

psychopy.gui.fileSaveDlg(initFilePath='', initFileName='', prompt='Select file to save', allowed=None)
A simple dialogue allowing write access to the file system. (Useful in case you collect an hour of data and then try to save to a non-existent directory!!)

Parameters

initFilePath: string default file path on which to open the dialog

initFileName: string default file name, as suggested file

prompt: string (default “Select file to open”) can be set to custom prompts

allowed: string a string to specify file filters. e.g. “Text files (*.txt);; Image files (*.bmp *.gif)”
See http://pyqt.sourceforge.net/Docs/PyQt4/qfiledialog.html #getSaveFileName for further details

If initFilePath or initFileName are empty or invalid then current path and empty names are used to start search.
If user cancels the None is returned.

8.12 psychopy.info - functions for getting information about the system

This module has tools for fetching data about the system or the current Python process. Such info can be useful for understanding the context in which an experiment was run.

class psychopy.info.RunTimeInfo(author=None, version=None, win=None, refreshTest='grating', userProcsDetailed=False, verbose=False)

Returns a snapshot of your configuration at run-time, for immediate or archival use.

Returns a dict-like object with info about PsychoPy, your experiment script, the system & OS, your window and monitor settings (if any), python & packages, and openGL.

If you want to skip testing the refresh rate, use ‘refreshTest=None’

Example usage: see runtimeInfo.py in coder demos.
Author

- 2010 written by Jeremy Gray, input from Jon Peirce and Alex Holcombe

Parameters

- `win` [None, False, Window instance] what window to use for refresh rate testing (if any) and settings. None -> temporary window using defaults; False -> no window created, used, nor profiled; a Window() instance you have already created
- `author` [None, string] None = try to autodetect first __author__ in sys.argv[0]; string = user-supplied author info (of an experiment)
- `version` [None, string] None = try to autodetect first __version__ in sys.argv[0]; string = user-supplied version info (of an experiment)
- `verbose` : False, True; how much detail to assess
- `refreshTest` [None, False, True, 'grating'] True or 'grating' = assess refresh average, median, and SD of 60 win.flip()s, using visual.getMsPerFrame() 'grating' = show a visual during the assessment; True = assess without a visual
- `userProcsDetailed` : False, True get details about concurrent user’s processes (command, process-ID)

Returns a flat dict (but with several groups based on key names):

- `psychopy` [version, rush() availability] psychopyVersion, psychopyHaveExtRush, git branch and current commit hash if available
- `experiment` [author, version, directory, name, current time-stamp,] SHA1 digest, VCS info (if any, svn or hg only), experimentAuthor, experimentVersion, …
- `system` [hostname, platform, user login, count of users,] user process info (count, cmd + pid), flagged processes systemHostname, systemPlatform, …
- `window` [(see output; many details about the refresh rate, window,] and monitor; units are noted) windowWinType, windowWaitBlanking, …windowRefreshTimeSD_ms, … windowMonitor.<details>, …
- `python` [version of python, versions of key packages] (wx, numpy, scipy, matplotlib, pyglet, pygame) pythonVersion, pythonScipyVersion, …
- `openGL` [version, vendor, rendering engine, plus info on whether] several extensions are present openGLVersion, …, openGLextGL_EXT_framebuffer_object, …

_setCurrentProcessInfo (verbose=False, userProcsDetailed=False)
What other processes are currently active for this user?

_setExperimentInfo (author, version, verbose)
Auto-detect __author__ and __version__ in sys.argv[0] (= the # user’s script)

_setPythonInfo()
External python packages, python details

_setSystemInfo()
System info

_setWindowInfo (win, verbose=False, refreshTest='grating', usingTempWin=True)
Find and store info about the window: refresh rate, configuration info.

psychopy.info.__getHgVersion (filename)
Tries to discover the mercurial (hg) parent and id of a file.
Not thoroughly tested; untested on Windows Vista, Win 7, FreeBSD
psychopy.info._getSha1hexDigest(thing, isfile=False)
  Returns base64/hex encoded sha1 digest of str(thing), or of a file contents. Return None if a file is requested but no such file exists

Author
  • 2010 written by Jeremy Gray

psychopy.info._getSvnVersion(filename)
  Try to discover the svn version (revision #) for a file.

  Not thoroughly tested; untested on Windows Vista, Win 7, FreeBSD

Author
  • 2010 written by Jeremy Gray

psychopy.info._getUserNameUID()
  Return username, UID.

  UID values can be used to infer admin-level: -1=undefined, 0=full admin/root, >499=assume non-admin/root (>999 on debian-based)

Author
  • 2010 written by Jeremy Gray

psychopy.info.getMemoryUsage()
  Get the memory (RAM) currently used by this Python process, in M.

psychopy.info.getRAM()
  Return system's physical RAM & available RAM, in M.

8.13 psychopy.logging - control what gets logged

Provides functions for logging error and other messages to one or more files and/or the console, using python's own logging module. Some warning messages and error messages are generated by PsychoPy itself. The user can generate more using the functions in this module.

There are various levels for logged messages with the following order of importance: ERROR, WARNING, DATA, EXP, INFO and DEBUG.

When setting the level for a particular log target (e.g. LogFile) the user can set the minimum level that is required for messages to enter the log. For example, setting a level of INFO will result in INFO, EXP, DATA, WARNING and ERROR messages to be recorded but not DEBUG messages.

By default, PsychoPy will record messages of WARNING level and above to the console. The user can silence that by setting it to receive only CRITICAL messages, (which PsychoPy doesn’t use) using the commands:
from psychopy import logging
logging.console.setLevel(logging.CRITICAL)

class psychopy.logging.LogFile (f=None, level=30, filemode='a', logger=None, encoding='utf8'):
    A text stream to receive inputs from the logging system

    Create a log file as a target for logged entries of a given level

    Parameters

    • f: this could be a string to a path, that will be created if it doesn’t exist. Alternatively this
      could be a file object, sys.stdout or any object that supports .write() and .flush() methods

    • level: The minimum level of importance that a message must have to be logged by this target.

    • filemode: ‘a’, ‘w’ Append or overwrite existing log file

    setLevel (level)
    Set a new minimal level for the log file/stream

    write (txt)
    Write directly to the log file (without using logging functions). Useful to send messages that only this file
    receives

class psychopy.logging._Logger (format='%(t).4f 	%(levelname)s 	%(message)s'):
    Maintains a set of log targets (text streams such as files of stdout)

    self.targets is a list of dicts {'stream':stream, 'level':level}

    The string-formatted elements %(xxxx)f can be used, where each xxxx is an attribute of the LogEntry. e.g. t,
    t_ms, level,levelname, message

    addTarget (target)
    Add a target, typically a LogFile to the logger

    flush ()
    Process all current messages to each target

    log (message, level=t=None, obj=None)
    Add the message to the log stack at the appropriate level

    If no relevant targets (files or console) exist then the message is simply discarded.

    removeTarget (target)
    Remove a target, typically a LogFile from the logger

psychopy.logging.addLevel (level, levelName)
    Associate ‘levelName’ with ‘level’.

    This is used when converting levels to text during message formatting.

psychopy.logging.debug (message)
    Send the message to any receiver of logging info (e.g. a LogFile) of level log.CRITICAL or higher

psychopy.logging.data (msg, t=None, obj=None)
    Log a message about data collection (e.g. a key press)

    usage:: log.data(message)

    Sends the message to any receiver of logging info (e.g. a LogFile) of level log.DATA or higher

psychopy.logging.debug (msg, t=None, obj=None)
    Log a debugging message (not likely to be wanted once experiment is finalised)

    usage:: log.debug(message)
Sends the message to any receiver of logging info (e.g. a LogFile) of level \texttt{log.DEBUG} or higher

\texttt{psychopy.logging.error(message)}

Send the message to any receiver of logging info (e.g. a LogFile) of level \texttt{log.ERROR} or higher

\texttt{psychopy.logging.exp(msg, t=None, obj=None)}

Log a message about the experiment (e.g. a new trial, or end of a stimulus)

\textbf{usage::} log.exp(message)

Send the message to any receiver of logging info (e.g. a LogFile) of level \texttt{log.EXP} or higher

\texttt{psychopy.logging.fatal(msg, t=None, obj=None)}

\texttt{log.critical(message)}

Send the message to any receiver of logging info (e.g. a LogFile) of level \texttt{log.CRITICAL} or higher

\texttt{psychopy.logging.flush(logger=<psychopy.logging._Logger object>)}

Send current messages in the log to all targets

\texttt{psychopy.logging.getLevel(level)}

Return the textual representation of logging level 'level'.

- If the level is one of the predefined levels (CRITICAL, ERROR, WARNING, INFO, DEBUG) then you get the corresponding string. If you have associated levels with names using addLevelName then the name you have associated with 'level' is returned.

- If a numeric value corresponding to one of the defined levels is passed in, the corresponding string representation is returned.

- Otherwise, the string “Level \%s” \% level is returned.

\texttt{psychopy.logging.info(msg, t=None, obj=None)}

\texttt{log.info(message)}

Sends the message to any receiver of logging info (e.g. a LogFile) of level \texttt{log.INFO} or higher

\texttt{psychopy.logging.log(msg, level, t=None, obj=None)}

Log a message

\textbf{usage::} log(level, msg, t=t, obj=obj)

Log the msg, at a given level on the root logger

\texttt{psychopy.logging.setDefaultClock(clock)}

Set the default clock to be used to reference all logging times. Must be a \texttt{psychopy.core.Clock} object. Beware that if you reset the clock during the experiment then the resets will be reflected here. That might be useful if you want your logs to be reset on each trial, but probably not.

\texttt{psychopy.logging.warn(msg, t=None, obj=None)}

\texttt{log.warning(message)}

Sends the message to any receiver of logging info (e.g. a LogFile) of level \texttt{log.WARNING} or higher

\texttt{psychopy.logging.warning(message)}

Sends the message to any receiver of logging info (e.g. a LogFile) of level \texttt{log.WARNING} or higher
8.13.1 flush()

```
psychopy.logging.flush(logger=<psychopy.logging._Logger object>)
```

Send current messages in the log to all targets

8.13.2 setDefaultClock()

```
psychopy.logging.setDefaultClock(clock)
```

Set the default clock to be used to reference all logging times. Must be a `psychopy.core.Clock` object. Beware that if you reset the clock during the experiment then the resets will be reflected here. That might be useful if you want your logs to be reset on each trial, but probably not.

8.14 psychopy.microphone - Capture and analyze sound

(Available as of version 1.74.00; Advanced features available as of 1.77.00)

8.14.1 Overview

`AudioCapture()` allows easy audio recording and saving of arbitrary sounds to a file (wav format). `AudioCapture` will likely be replaced entirely by `AdvAudioCapture` in the near future.

`AdvAudioCapture()` can do everything `AudioCapture` does, and also allows onset-marker sound insertion and detection, loudness computation (RMS audio “power”), and lossless file compression (flac). The Builder microphone component now uses `AdvAudioCapture` by default.

8.14.2 Audio Capture

```
psychopy.microphone.switchOn(sampleRate=48000, outputDevice=None, bufferSize=None)
```

You need to switch on the microphone before use, which can take several seconds. The only time you can specify the sample rate (in Hz) is during `switchOn()`.

Considerations on the default sample rate 48kHz:

<table>
<thead>
<tr>
<th>Format</th>
<th>Sample Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>DVD or video</td>
<td>48,000</td>
</tr>
<tr>
<td>CD-quality</td>
<td>44,100 / 24 bit</td>
</tr>
<tr>
<td>Human hearing: adult</td>
<td>~15,000</td>
</tr>
<tr>
<td>Children &amp; young adult: higher</td>
<td>~30,000</td>
</tr>
<tr>
<td>Human speech: telephone</td>
<td>100-3,000</td>
</tr>
<tr>
<td>Google speech API</td>
<td>16,000 or 8,000 only</td>
</tr>
<tr>
<td>Nyquist frequency: twice the highest rate</td>
<td>Good to oversample a bit</td>
</tr>
</tbody>
</table>

`pyo`'s `downsamp()` function can reduce 48,000 to 16,000 in about 0.02s (uses integer step sizes). So recording at 48kHz will generate high-quality archival data, and permit easy downsampling.

`outputDevice, bufferSize`: set these parameters on the `pyoSndServer` before booting; None means use `pyo`'s default values

```
class psychopy.microphone.AdvAudioCapture(name='advMic', filename='', saveDir='', sampleType=0, buffering=16, chnl=0, stereo=True, autoLog=True)
```

Class extends `AudioCapture`, plays marker sound as a “start” indicator.

Has method for retrieving the marker onset time from the file, to allow calculation of vocal RT (or other sound-based RT).
See Coder demo > input > latencyFromTone.py

**Parameters**

**name**: Stem for the output file, also used in logging.

**filename**: Optional file name to use; default = 'name-onsetTimeEpoch.wav'

**saveDir**: Directory to use for output .wav files. If a saveDir is given, it will return 'saveDir/file'. If no saveDir, then return abspath(file)

**sampletype** [bit depth] pyo recording option: 0=16bits int, 1=24 bits int; 2=32 bits int

**buffering** : pyo argument chnl: which audio input channel to record (default=0) stereo: how many channels to record

  (default True, stereo; False = mono)

**compress (keep=False)** Compress using FLAC (lossless compression).

**getLoudness ()**

Return the RMS loudness of the saved recording.

**getMarkerInfo ()**

Returns (hz, duration, volume) of the marker sound. Custom markers always return 0 hz (regardless of the sound).

**getMarkerOnset (chunk=128, secs=0.5, filename='')**

Return (onset, offset) time of the first marker within the first secs of the saved recording.

Has approx ~1.33ms resolution at 48000Hz, chunk=64. Larger chunks can speed up processing times, at a sacrifice of some resolution, e.g., to pre-process long recordings with multiple markers.

If given a filename, it will first set that file as the one to work with, and then try to detect the onset marker.

**playMarker ()**

Plays the current marker sound. This is automatically called at the start of recording, but can be called anytime to insert a marker.

**playback (block=True, loops=0, stop=False, log=True)**

Plays the saved .wav file, as just recorded or resampled. Execution blocks by default, but can return immediately with block=False.

  **loops** : number of extra repetitions; 0 = play once

  **stop** : True = immediately stop ongoing playback (if there is one), and return

**record (sec, filename='', block=False)**

Starts recording and plays an onset marker tone just prior to returning. The idea is that the start of the tone in the recording indicates when this method returned, to enable you to sync a known recording onset with other events.

**resample (newRate=16000, keep=True, log=True)**

Re-sample the saved file to a new rate, return the full path.

Can take several visual frames to resample a 2s recording.

The default values for resample() are for Google-speech, keeping the original (presumably recorded at 48kHz) to archive. A warning is generated if the new rate not an integer factor / multiple of the old rate.

To control anti-aliasing, use pyo.downsamp() or upsamp() directly.
reset (log=True)
    Restores to fresh state, ready to record again

setFile (filename)
    Sets the name of the file to work with.

setMarker (tone=19000, secs=0.015, volume=0.03, log=True)
    Sets the onset marker, where tone is either in hz or a custom sound.

    The default tone (19000 Hz) is recommended for auto-detection, as being easier to isolate from speech sounds
    (and so reliable to detect). The default duration and volume are appropriate for a quiet setting such as a lab
    testing room. A louder volume, longer duration, or both may give better results when recording loud sounds
    or in noisy environments, and will be auto-detected just fine (even more easily). If the hardware microphone
    in use is not physically near the speaker hardware, a louder volume is likely to be required.

    Custom sounds cannot be auto-detected, but are supported anyway for presentation purposes. E.g., a recording
    of someone saying “go” or “stop” could be passed as the onset marker.

stop (log=True)
    Interrupt a recording that is in progress; close & keep the file.

    Ends the recording before the duration that was initially specified. The same file name is retained, with the
    same onset time but a shorter duration.

    The same recording cannot be resumed after a stop (it is not a pause), but you can start a new one.

uncompress (keep=False)
    Uncompress from FLAC to .wav format.

8.14.3 Speech recognition

Google’s speech to text API is no longer available. AT&T, IBM, and wit.ai have a similar (paid) service.

8.14.4 Misc

Functions for file-oriented Discrete Fourier Transform and RMS computation are also provided.

psychopy.microphone.wav2flac (path, keep=True, level=5)
    Lossless compression: convert .wav file (on disk) to .flac format.

    If path is a directory name, convert all .wav files in the directory.

    keep to retain the original .wav file(s), default True.

    level is compression level: 0 is fastest but larger, 8 is slightly smaller but much slower.

psychopy.microphone.flac2wav (path, keep=True)
    Uncompress: convert .flac file (on disk) to .wav format (new file).

    If path is a directory name, convert all .flac files in the directory.

    keep to retain the original .flac file(s), default True.

psychopy.microphone.getDft (data, sampleRate=None, wantPhase=False)
    Compute and return magnitudes of numpy.fft.fft() of the data.

    If given a sample rate (samples/sec), will return (magn, freq). If wantPhase is True, phase in radians is also returned
    (magn, freq, phase). data should have power-of-2 samples, or will be truncated.
psychoPy.microphone.getRMS(data)
Compute and return the audio power (“loudness”).
Uses numpy.std() as RMS. std() is same as RMS if the mean is 0, and .wav data should have a mean of 0. Returns an array if given stereo data (RMS computed within-channel).

data can be an array (1D, 2D) or filename; .wav format only. data from .wav files will be normalized to -1..+1 before RMS is computed.

8.15 psychopy.misc - miscellaneous routines for converting units etc

Wrapper for all miscellaneous functions and classes from psychopy.tools
psychopy.misc has gradually grown very large and the underlying code for its functions are distributed in multiple files. You can still (at least for now) import the functions here using from psychopy import misc but you can also import them from the tools sub-modules.

8.15.1 From psychopy.tools.filetools

toFile(filename, data) Save data (of any sort) as a pickle file.
fromFile(filename[, encoding]) Load data from a pickle or JSON file.
mergeFolder(src, dst[, pattern]) Merge a folder into another.

8.15.2 From psychopy.tools.colorspacetools
dk12rgb(dkl[, conversionMatrix]) Convert from DKL color space (Derrington, Krauskopf & Lennie) to RGB.
dk1Cart2rgb(LUM, LM, S[, conversionMatrix]) Like dk12rgb except that it uses cartesian coords (L,M,S,LUM) rather than spherical coords for DKL (elev, azim, contr).
rgb2dk1Cart(picture[, conversionMatrix]) Convert an RGB image into Cartesian DKL space.
hsv2rgb(hsv_Nx3) Convert from HSV color space to RGB gun values.
lms2rgb(lms_Nx3[, conversionMatrix]) Convert from cone space (Long, Medium, Short) to RGB.
rgb2lms(rgb_Nx3[, conversionMatrix]) Convert from RGB to cone space (LMS).
dk12rgb(dkl[, conversionMatrix]) Convert from DKL color space (Derrington, Krauskopf & Lennie) to RGB.

8.15.3 From psychopy.tools.coordinatestools
cart2pol(x, y[, units]) Convert from cartesian to polar coordinates.
cart2sph(z, y, x) Convert from cartesian coordinates (x,y,z) to spherical (elevation, azimuth, radius).
pol2cart(theta, radius[, units]) Convert from polar to cartesian coordinates.
sph2cart(*args) Convert from spherical coordinates (elevation, azimuth, radius) to cartesian (x,y,z).
### 8.15.4 From `psychopy.tools.monitorunittools`

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>convertToPix</code></td>
<td>Takes vertices and position, combines and converts to pixels from any unit.</td>
</tr>
<tr>
<td><code>cm2pix</code></td>
<td>Convert size in cm to size in pixels for a given Monitor object.</td>
</tr>
<tr>
<td><code>cm2deg</code></td>
<td>Convert size in cm to size in degrees for a given Monitor object.</td>
</tr>
<tr>
<td><code>deg2cm</code></td>
<td>Convert size in degrees to size in pixels for a given Monitor object.</td>
</tr>
<tr>
<td><code>deg2pix</code></td>
<td>Convert size in degrees to size in pixels for a given Monitor object.</td>
</tr>
<tr>
<td><code>pix2cm</code></td>
<td>Convert size in pixels to size in cm for a given Monitor object.</td>
</tr>
<tr>
<td><code>pix2deg</code></td>
<td>Convert size in pixels to size in degrees for a given Monitor object.</td>
</tr>
</tbody>
</table>

### 8.15.5 From `psychopy.tools.imagetools`

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>array2image</code></td>
<td>Takes an array and returns an image object (PIL).</td>
</tr>
<tr>
<td><code>image2array</code></td>
<td>Takes an image object (PIL) and returns a numpy array.</td>
</tr>
<tr>
<td><code>makeImageAuto</code></td>
<td>Combines float_uint8 and image2array operations ie.</td>
</tr>
</tbody>
</table>

### 8.15.6 From `psychopy.tools.plottools`

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>plotFrameIntervals</code></td>
<td>Plot a histogram of the frame intervals.</td>
</tr>
</tbody>
</table>

### 8.15.7 From `psychopy.tools.typetools`

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>float_uint8</code></td>
<td>Converts arrays, lists, tuples and floats ranging -1:1 into an array of Uint8s ranging 0:255</td>
</tr>
<tr>
<td><code>uint8_float</code></td>
<td>Converts arrays, lists, tuples and UINTs ranging 0:255 into an array of floats ranging -1:1</td>
</tr>
<tr>
<td><code>float_uint16</code></td>
<td>Converts arrays, lists, tuples and floats ranging -1:1 into an array of Uint16s ranging 0:2^16</td>
</tr>
</tbody>
</table>
8.15.8 From `psychopy.tools.unittools`

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>radians</code></td>
<td>radians(x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature, extobj])</td>
</tr>
<tr>
<td><code>degrees</code></td>
<td>degrees(x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature, extobj])</td>
</tr>
</tbody>
</table>

8.16 `psychopy.monitors` - for those that don’t like Monitor Center

Most users won’t need to use the code here. In general the Monitor Centre interface is sufficient and monitors setup that way can be passed as strings to `Window`s. If there is some aspect of the normal calibration that you wish to override, eg:

```python
from psychopy import visual, monitors
mon = monitors.Monitor('SonyG55') #fetch the most recent calib for this monitor
mon.setDistance(114) #further away than normal?
win = visual.Window(size=[1024,768], monitor=mon)
```

You might also want to fetch the `Photometer` class for conducting your own calibrations

8.16.1 Monitor

```python
class psychopy.monitors.Monitor (name=None, width=None, distance=None, gamma=None, notes=None, useBits=None, verbose=True, currentCalib=None, autoLog=True)
```

Creates a monitor object for storing calibration details. This will be loaded automatically from disk if the monitor name is already defined (see methods).

Many settings from the stored monitor can easily be overridden either by adding them as arguments during the initial call.

**arguments:**

- `width, distance, gamma` are details about the calibration
- `notes` is a text field to store any useful info
- `useBits` True, False, None
- `verbose` True, False, None
- `currentCalib is a dictionary object containing various` fields for a calibration. Use with caution since the dictionary may not contain all the necessary fields that a monitor object expects to find.

**eg:**

```python
myMon = Monitor('sony500', distance=114) #Fetches the info on the sony500 and overrides its usual distance to be 114cm for this experiment.
```

These can be saved to the monitor file using `save()` or not (in which case the changes will be lost)

```python
_loadAll() #Fetches the calibrations for this monitor from disk, storing them as self.calibs
```
copyCalib(calibName=None)
Stores the settings for the current calibration settings as new monitor.

delCalib(calibName)
Remove a specific calibration from the current monitor. Won't be finalised unless monitor is saved

gammaIsDefault()
Determine whether we're using the default gamma values

getCalibDate()
As a python date object (convert to string using calibTools.strFromDate

getDKL_RGB(RECOMPUTE=False)
Returns the DKL->RGB conversion matrix. If one has been saved this will be returned. Otherwise, if power
spectra are available for the monitor a matrix will be calculated.

getDistance()
Returns distance from viewer to the screen in cm, or None if not known

gammaGrid()  
Returns just the gamma value (not the whole grid)

getGammaGrid()
Gets the min, max, gamma values for the each gun

getLMS_RGB(recompute=False)
ReturnstheLMS->RGBconversionmatrix. Ifonehasbeensavedthiswillbereturned. Otherwise(ifpowerspectraareavailableforthemonitor)amatricewillbecalculated.

getLevelsPost()
Gets the measured luminance values from last calibration TEST

getLevelsPre()
Gets the measured luminance values from last calibration

getLinearizeMethod()
Gets the method that this monitor is using to linearize the guns

getLumsPost()
Gets the measured luminance values from last calibration TEST

getLumsPre()
Gets the measured luminance values from last calibration

getMeanLum()
Returns the mean luminance of the screen if explicitly stored

getNotes()
Notes about the calibration

getPsychopyVersion()
Returns the version of PsychoPy that was used to create this calibration

getSizePix()
Returns the size of the current calibration in pixels, or None if not defined

getSpectra()
Gets the wavelength values from the last spectrometer measurement (if available)

usage:

• nm, power = monitor.getSpectra()
getUseBits()
    Was this calibration carried out with a bits++ box

getWidth()
    Of the viewable screen in cm, or None if not known

lineariseLums(desiredLums, newInterpolator=False, overrideGamma=None)
    Equivalent of lineariseLums().

lineariseLums(desiredLums, newInterpolator=False, overrideGamma=None)
    lums should be uncalibrated luminance values (e.g. a linear ramp) ranging 0:1

newCalib(calibName=None, width=None, distance=None, gamma=None, notes=None, useBits=False, verbose=True)
    create a new (empty) calibration for this monitor and makes this the current calibration

save()
    Save the current calibrations to disk.
    This will write a json file to the monitors subfolder of your PsychoPy configuration folder (typically ~/.psychopy3 monitors on Linux and macOS, and %APPDATA%psychopy3 monitors on Windows).
    Additionally saves a pickle (.calib) file if you are running Python 2.7.

saveMon()
    Equivalent of save().

setCalibDate(date=None)
    Sets the current calibration to have a date/time or to the current date/time if none given. (Also returns the date as set)

setCurrent(calibration=-1)
    Sets the current calibration for this monitor. Note that a single file can hold multiple calibrations each stored under a different key (the date it was taken)
    The argument is either a string (naming the calib) or an integer eg:

    myMon.setCurrent('mainCalib') fetches the calibration named mainCalib. You can name calibrations what you want but PsychoPy will give them names of date/time by default. In Monitor Center you can ‘copy…’ a calibration and give it a new name to keep a second version.

    calibName = myMon.setCurrent(0) fetches the first calibration (alphabetically) for this monitor

    calibName = myMon.setCurrent(-1) fetches the last alphabetical calibration for this monitor (this is default). If default names are used for calibrations (ie date/time stamp) then this will import the most recent.

setDKL_RGB(dkl_rgb)
    Sets the DKL->RGB conversion matrix for a chromatically calibrated monitor (matrix is a 3x3 num array).

setDistance(distance)
    To the screen (cm)

setGamma(gamma)
    Sets the gamma value(s) for the monitor. This only uses a single gamma value for the three guns, which is fairly approximate. Better to use setGammaGrid (which uses one gamma value for each gun)

setGammaGrid(gammaGrid)
    Sets the min,max,gamma values for the each gun

setLMS_RGB(lms_rgb)
    Sets the LMS->RGB conversion matrix for a chromatically calibrated monitor (matrix is a 3x3 num array).
setLevelsPost (levels)
Sets the last set of luminance values measured AFTER calibration

setLevelsPre (levels)
Sets the last set of luminance values measured during calibration

setLineariseMethod (method)
Sets the method for linearising 0 uses $y=a+(bx)^\gamma$ 1 uses $y=(a+bx)^\gamma$ 2 uses linear interpolation over the curve

setLumsPost (lums)
Sets the last set of luminance values measured AFTER calibration

setLumsPre (lums)
Sets the last set of luminance values measured during calibration

setMeanLum (meanLum)
Records the mean luminance (for reference only)

setNotes (notes)
For you to store notes about the calibration

setPsychopyVersion (version)
To store the version of PsychoPy that this calibration used

setSizePix (pixels)
Set the size of the screen in pixels x,y

setSpectra (nm, rgb)
Sets the phosphor spectra measured by the spectrometer

setUseBits (usebits)
DEPRECATED: Use the new hardware classes to control these devices

setWidth (width)
Of the viewable screen (cm)

---

8.16.2 GammaCalculator

class psychopy.monitors.GammaCalculator (inputs=(), lums=(), gamma=None, bitsIN=8, bitsOUT=8, eq=1)

Class for managing gamma tables

Parameters:

- **inputs** (required)= values at which you measured screen luminance either in range 0.0:1.0, or range 0:255. Should include the min and max of the monitor

Then give EITHER “lums” or “gamma”:

- lums = measured luminance at given input levels
- gamma = your own gamma value (single float)
- bitsIN = number of values in your lookup table
- bitsOUT = number of bits in the DACs

myTable.gammaModel myTable.gamma
fitGammaErrFun (params, x, y, minLum, maxLum)
Provides an error function for fitting gamma function
(used by fitGammaFun)

fitGammaFun (x, y)
Fits a gamma function to the monitor calibration data.
Parameters: -xVals are the monitor look-up-table vals, either 0-255 or 0.0-1.0 -yVals are the measured luminances from a photometer/spectrometer

8.16.3 getAllMonitors()

psychopy.monitors.getAllMonitors()
Find the names of all monitors for which calibration files exist

8.16.4 getLumSeriesPR650()

psychopy.monitors.getLumSeriesPR650 (lumLevels=8, winSize=(800, 600), monitor=None, gamma=1.0, allGuns=True, useBits=False, autoMode='auto', stimSize=0.3, photometer='COM1')
DEPRECATED (since v1.60.01): Use psychopy.monitors.getLumSeries() instead

8.16.5 getRGBspectra()

psychopy.monitors.getRGBspectra (stimSize=0.3, winSize=(800, 600), photometer='COM1')
usage: getRGBspectra(stimSize=0.3, winSize=(800,600), photometer='COM1')

Params
- ‘photometer’ could be a photometer object or a serial port name on which a photometer might be found (not recommended)

8.16.6 gammaFun()

psychopy.monitors.gammaFun (xx, minLum, maxLum, gamma, eq=1, a=None, b=None, k=None)
Returns gamma-transformed luminance values. y = gammaFun(x, minLum, maxLum, gamma)
a and b are calculated directly from minLum, maxLum, gamma
Parameters:
- xx are the input values (range 0-255 or 0.0-1.0)
- minLum = the minimum luminance of your monitor
- maxLum = the maximum luminance of your monitor (for this gun)
- gamma = the value of gamma (for this gun)
8.16.7 \texttt{gammaInvFun}()

\texttt{psychopy.monitors.gammaInvFun(\textit{yy}, \textit{minLum}, \textit{maxLum}, \textit{gamma}, \textit{b=None}, \textit{eq=1})}

Returns inverse gamma function for desired luminance values. \(x = \text{gammaInvFun}(y, \text{minLum}, \text{maxLum}, \text{gamma})\)

\textit{a} and \textit{b} are calculated directly from \textit{minLum}, \textit{maxLum}, \textit{gamma}

Parameters:

- \textit{xx} are the input values (range 0-255 or 0.0-1.0)
- \textit{minLum} = the minimum luminance of your monitor
- \textit{maxLum} = the maximum luminance of your monitor (for this gun)
- \textit{gamma} = the value of gamma (for this gun)
- \textit{eq} determines thegamma equation used: \(eq==1\) [default]: \(yy = a + (b \times xx)^\gamma\) \(eq==2\): \(yy = (a + b \times xx)^\gamma\)

8.16.8 \texttt{makeDKL2RGB}()

\texttt{psychopy.monitors.makeDKL2RGB(\textit{nm}, \textit{powerRGB})}

Creates a 3x3 DKL->RGB conversion matrix from the spectral input powers

8.16.9 \texttt{makeLMS2RGB}()

\texttt{psychopy.monitors.makeLMS2RGB(\textit{nm}, \textit{powerRGB})}

Creates a 3x3 LMS->RGB conversion matrix from the spectral input powers

8.17 \texttt{psychopy.parallel} - functions for interacting with the parallel port

This module provides read / write access to the parallel port for Linux or Windows.

The \texttt{Parallel} class described below will attempt to load whichever parallel port driver is first found on your system and should suffice in most instances. If you need to use a specific driver then, instead of using \texttt{ParallelPort} shown below you can use one of the following as drop-in replacements, forcing the use of a specific driver:

- \texttt{psychopy.parallel.PParallelInpOut}
- \texttt{psychopy.parallel.PParallelDLPortIO}
- \texttt{psychopy.parallel.PParallelLinux}

Either way, each instance of the class can provide access to a different parallel port.

There is also a legacy API which consists of the routines which are directly in this module. That API assumes you only ever want to use a single parallel port at once.

\texttt{class psychopy.parallel.ParallelPort(\textit{address})}

Class for read/write access to the parallel port on Windows & Linux

Usage:
from psychopy import parallel
port = parallel.ParallelPort(address=0x0378)

port.setData(4)
port.readPin(2)
port.setPin(2, 1)

This is just a dummy constructor to avoid errors when the parallel port cannot be initiated

readData()
Return the value currently set on the data pins (2-9)

readPin(pinNumber)
Determine whether a desired (input) pin is high(1) or low(0).

Pins 2-13 and 15 are currently read here

setData(data)
Set the data to be presented on the parallel port (one ubyte). Alternatively you can set the value of each pin (data pins are pins 2-9 inclusive) using setPin()

Examples:

from psychopy import parallel
port = parallel.ParallelPort(address=0x0378)

port.setData(0)  # sets all pins low
port.setData(255) # sets all pins high
port.setData(2)  # sets just pin 3 high (pin2 = bit0)
port.setData(3)  # sets just pins 2 and 3 high

You can also convert base 2 to int easily in python:

port.setData( int("00000011", 2) )  # pins 2 and 3 high
port.setData( int("00000101", 2) )  # pins 2 and 4 high

8.17.1 Legacy functions

We would strongly recommend you use the class above instead: these are provided for backwards compatibility only.

parallel.setPortAddress()
Set the memory address or device node for your parallel port of your parallel port, to be used in subsequent commands

Common port addresses:

LPT1 = 0x0378 or 0x03BC
LPT2 = 0x0278 or 0x0378
LPT3 = 0x0278

or for Linux: /dev/parport0

This routine will attempt to find a usable driver depending on your platform

parallel.setData()
Set the data to be presented on the parallel port (one ubyte). Alternatively you can set the value of each pin (data pins are pins 2-9 inclusive) using setPin()
Examples:

```python
parallel.setData(0)  # sets all pins low
parallel.setData(255)  # sets all pins high
parallel.setData(2)  # sets just pin 3 high (remember that pin2=bit0)
parallel.setData(3)  # sets just pins 2 and 3 high
```

You can also convert base 2 to int easily in python:

```python
parallel.setData(int("00000011", 2))  # pins 2 and 3 high
parallel.setData(int("00000101", 2))  # pins 2 and 4 high
```

```python
parallel.setPin(state)
```

Set a desired pin to be high (1) or low (0).

Only pins 2-9 (incl) are normally used for data output:

```python
parallel.setPin(3, 1)  # sets pin 3 high
parallel.setPin(3, 0)  # sets pin 3 low
```

```python
parallel.readPin()
```

Determine whether a desired (input) pin is high(1) or low(0).

Pins 2-13 and 15 are currently read here

---

### 8.18 `psychopy.plugins` - utilities for extending PsychoPy with plugins

#### 8.18.1 Overview

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>loadPlugin(plugin, *args, **kwargs)</code></td>
<td>Load a plugin to extend PsychoPy.</td>
</tr>
<tr>
<td><code>listPlugins([which])</code></td>
<td>Get a list of installed or loaded PsychoPy plugins.</td>
</tr>
<tr>
<td><code>startUpPlugins(plugins[, add, verify])</code></td>
<td>Specify which plugins should be loaded automatically when a PsychoPy session starts.</td>
</tr>
<tr>
<td><code>computeChecksum(fpath[, method, writeOut])</code></td>
<td>Compute the checksum hash/key for a given package.</td>
</tr>
</tbody>
</table>

#### 8.18.2 Details

`psychopy.plugins.loadPlugin (plugin, *args, **kwargs)`

Load a plugin to extend PsychoPy.

Plugins are packages which extend upon PsychoPy’s existing functionality by dynamically importing code at runtime, without modifying the existing installation files. Plugins create or redefine objects in the namespaces of modules (eg. `psychopy.visual`) and unbound classes, allowing them to be used as if they were part of PsychoPy. In some cases, objects exported by plugins will be registered for a particular function if they define entry points into specific modules.

Plugins are simply Python packages, `loadPlugin` will search for them in directories specified in `sys.path`. Only packages which define entry points in their metadata which pertain to PsychoPy can be loaded with this function. This function also permits passing optional arguments to a callable object in the plugin module to run any initialization routines prior to loading entry points.

This function is robust, simply returning `True` or `False` whether a plugin has been fully loaded or not. If a plugin
fails to load, the reason for it will be written to the log as a warning or error, and the application will continue running. This may be undesirable in some cases, since features the plugin provides may be needed at some point and would lead to undefined behavior if not present. If you want to halt the application if a plugin fails to load, consider using `requirePlugin()`.

It is advised that you use this function only when using PsychoPy as a library. If using the builder or coder GUI, it is recommended that you use the plugin dialog to enable plugins for PsychoPy sessions spawned by the experiment runner. However, you can still use this function if you want to load additional plugins for a given experiment, having their effects isolated from the main application and other experiments.

**Parameters**

- `plugin (str)` – Name of the plugin package to load. This usually refers to the package or project name.
- `*args` – Optional arguments and keyword arguments to pass to the plugin’s `__register__` function.
- `**kwargs` – Optional arguments and keyword arguments to pass to the plugin’s `__register__` function.

**Returns** `True` if the plugin has valid entry points and was loaded successfully. Also returns `True` if the plugin was already loaded by a previous `loadPlugin` call this session, this function will have no effect in this case. `False` is returned if the plugin defines no entry points specific to PsychoPy or crashed (an error is logged).

**Return type** `bool`

**Warning:** Make sure that plugins installed on your system are from reputable sources, as they may contain malware! PsychoPy is not responsible for undefined behaviour or bugs associated with the use of 3rd party plugins.

**See also:**

- `listPlugins` Search for and list installed or loaded plugins.
- `requirePlugin` Require a plugin be previously loaded.

**Examples**

Load a plugin by specifying its package/project name:

```python
loadPlugin('psychopy-hardware-box')
```

You can give arguments to this function which are passed on to the plugin:

```python
loadPlugin('psychopy-hardware-box', switchOn=True, baudrate=9600)
```

You can use the value returned from `loadPlugin` to determine if the plugin is installed and supported by the platform:

```python
hasPlugin = loadPlugin('psychopy-hardware-box')
if hasPlugin:
    # initialize objects which require the plugin here ...
```

Get a list of installed or loaded PsychoPy plugins:

```python
psychopy.plugins.listPlugins( which='all' )
```
This function lists either all potential plugin packages installed on the system, those registered to be loaded automatically when PsychoPy starts, or those that have been previously loaded successfully this session.

**Parameters**

`which (str)` – Category to list plugins. If ‘all’, all plugins installed on the system will be listed, whether they have been loaded or not. If ‘loaded’, only plugins that have been previously loaded successfully this session will be listed. If ‘startup’, plugins registered to be loaded when a PsychoPy session starts will be listed, whether or not they have been loaded this session. If ‘unloaded’, plugins that have not been loaded but are installed will be listed. If ‘failed’, returns a list of plugin names that attempted to load this session but failed for some reason.

**Returns** Names of PsychoPy related plugins as strings. You can load all installed plugins by passing list elements to `loadPlugin`.

**Return type** list

See also:

`loadPlugin` Load a plugin into the current session.

**Examples**

Load all plugins installed on the system into the current session (assumes all plugins don’t require any additional arguments passed to them):

```python
for plugin in plugins.listPlugins():
    plugins.loadPlugin(plugin)
```

If certain plugins take arguments, you can do this give specific arguments when loading all plugins:

```python
pluginArgs = {'some-plugin': (('someArg',), {'setup': True, 'spam': 10})
for plugin in plugins.listPlugins():
    try:
        args, kwargs = pluginArgs[plugin]
        plugins.loadPlugin(plugin, *args, **kwargs)
    except KeyError:
        plugins.loadPlugin(plugin)
```

Check if a plugin package named `plugin-test` is installed on the system and has entry points into PsychoPy:

```python
if 'plugin-test' in plugins.listPlugins():
    print("Plugin installed!")
```

Check if all plugins registered to be loaded on startup are currently active:

```python
if not all([p in listPlugins('loaded') for p in listPlugins('startup')]):
    print('Please restart your PsychoPy session for plugins to take effect.')
```

`psychopy.plugins.startUpPlugins (plugins, add=True, verify=True)`

Specify which plugins should be loaded automatically when a PsychoPy session starts.

This function edits `psychopy.preferences.prefs.general['startUpPlugins']` and provides a means to verify if entries are valid. The PsychoPy session must be restarted for the plugins specified to take effect.

If using PsychoPy as a library, this function serves as a convenience to avoid needing to explicitly call `loadPlugin()` every time to use your favorite plugins.

**Parameters**

- `plugins (str, list or None)` – Name(s) of plugins to have load on startup.
• **add** *(bool)* – If *True* names of plugins will be appended to `startUpPlugins` unless a name is already present. If *False*, `startUpPlugins` will be set to `plugins`, overwriting the previous value. If `add=False` and `plugins=[]` or `plugins=None`, no plugins will be loaded in the next session.

• **verify** *(bool)* – Check if `plugins` are installed and have valid entry points to PsychoPy. Raises an error if any are not. This prevents undefined behavior arising from invalid plugins being loaded in the next session. If *False*, plugin names will be added regardless if they are installed or not.

**Raises** *RuntimeError* – If `verify=True`, any of `plugins` is not installed or does not have entry points to PsychoPy. This is raised to prevent issues in future sessions where invalid plugins are written to the config file and are automatically loaded.

**Warning:** Do not use this function within the builder or coder GUI! Use the plugin dialog to specify which plugins to load on startup. Only use this function when using PsychoPy as a library!

### Examples

Adding plugins to load on startup:

```python
startUpPlugins(['plugin1', 'plugin2'])
```

Clearing the startup plugins list, no plugins will be loaded automatically at the start of the next session:

```python
plugins.startUpPlugins([], add=False)
# or ..
plugins.startUpPlugins(None, add=False)
```

If passing `None` or an empty list with `add=True`, the present value of `prefs.general['startUpPlugins']` will remain as-is.

```python
psychopy.plugins.computeChecksum(fpath, method='sha256', writeOut=None)
```

Compute the checksum hash/key for a given package.

Authors of PsychoPy plugins can use this function to compute a checksum hash and users can use it to check the integrity of their packages.

**Parameters**

- **fpath** *(str)* – Path to the plugin package or file.
- **method** *(str)* – Hashing method to use, values are ‘md5’ or ‘sha256’. Default is ‘sha256’.
- **writeOut** *(str)* – Path to a text file to write checksum data to. If the file exists, the data will be written as a line at the end of the file.

**Returns** Checksum hash digested to hexadecimal format.

**Return type** *str*
Examples

Compute a checksum for a package and write it to a file:

```python
with open('checksum.txt', 'w') as f:
    f.write(computeChecksum('/path/to/plugin/psychopy_plugin-1.0-py3.6.egg'))
```

8.19 psychopy.preferences - getting and setting preferences

You can set preferences on a per-experiment basis. For example, if you would like to use a specific audio library, but don’t want to touch your user settings in general, you can import preferences and set the option audioLib accordingly:

```python
from psychopy import prefs
prefs.hardware['audioLib'] = ['pyo']
from psychopy import sound
```

!!IMPORTANT!! You must import the sound module AFTER setting the preferences. To check that you are getting what you want (don’t do this in your actual experiment):

```python
print sound.Sound
```

The output should be `<class 'psychopy.sound.SoundPyo'>` for pyo, or `<class 'psychopy.sound.SoundPygame'>` for pygame.

You can find the names of the preferences sections and their different options [here](#).

8.19.1 Preferences

Class for loading / saving prefs

```python
class psychopy.preferences.Preferences:
    Users can alter preferences from the dialog box in the application, by editing their user preferences file (which is what the dialog box does) or, within a script, preferences can be controlled like this:

```python
import psychopy
psychopy.prefs.hardware['audioLib'] = ['PTB', 'pyo', 'pygame']
print(prefs)
# prints the location of the user prefs file and all the current vals
```

Use the instance of `prefs`, as above, rather than the `Preferences` class directly if you want to affect the script that’s running.

loadAll()  
Load the user prefs and the application data

loadUserPrefs()  
load user prefs, if any; don’t save to a file because doing so will break easy_install. Saving to files within the psychopy/ is fine, eg for key-bindings, but outside it (where user prefs will live) is not allowed by easy_install (security risk)

resetPrefs()  
removes userPrefs.cfg, does not touch appData.cfg

restoreBadPrefs(cfg, result)  
result = result of `validate`
saveAppData()
    Save the various setting to the appropriate files (or discard, in some cases)

saveUserPrefs()
    Validate and save the various setting to the appropriate files (or discard, in some cases)

validate()
    Validate (user) preferences and reset invalid settings to defaults

8.20 psychopy.serial - functions for interacting with the serial port

PsychoPy is compatible with Chris Liechti’s pyserial package. You can use it like this:

```python
import serial
ser = serial.Serial(0, 19200, timeout=1) # open first serial port
#ser = serial.Serial('/dev/ttyS1', 19200, timeout=1)#or something like this for Mac/
→ Linux machines
ser.write('someCommand')
line = ser.readline() # read a '\n' terminated line
ser.close()
```

Ports are fully configurable with all the options you would expect of RS232 communications. See http://pyserial.sourceforge.net for further details and documentation.

pyserial is packaged in the Standalone (Windows and Mac distributions), for manual installations you should install this yourself.

8.21 psychopy.sound - play various forms of sound

8.21.1 Sound

PsychoPy currently supports a choice of sound engines: PTB, pyo, sounddevice or pygame. You can select which will be used via the audioLib preference. sound.Sound() will then refer to one of SoundPTB, SoundDevice, SoundPyo or SoundPygame. This preference can be set on a per-experiment basis by importing preferences, and setting the audioLib option to use.

- The PTB library has by far the lowest latencies and is strongly recommended (requires 64 bit Python3)
- The pyo library is, in theory, the highest performer, but in practice it has often had issues (at least on macOS) with crashes and freezing of experiments, or causing them not to finish properly. If those issues aren’t affecting your studies then this could be the one for you.
- The sounddevice library looks like the way of the future. The performance appears to be good (although this might be less so in cases where you have complex rendering being done as well because it operates from the same computer core as the main experiment code). It’s newer than pyo and so more prone to bugs and we haven’t yet added microphone support to record your participants.
- The pygame backend is the oldest and should work without errors, but has the least good performance. Use it if latencies for your audio don’t matter.

Sounds are actually generated by a variety of classes, depending on which “backend” you use (like pyo or sounddevice) and these different backends can have slightly different attributes, as below.

The user should typically do:
but the class that gets imported will then be an alias of one of the Sound Classes described below.

### 8.21.2 PTB audio latency

PTB brings a number of advantages in terms of latency.

The first is that it has been designed specifically with low-latency playback in mind (rather than, say, on-the-fly mixing and filtering capabilities). Mario Kleiner has worked very hard to get the best out of the drivers available on each operating system and, as a result, with the most aggressive low-latency settings you can get a sound to play in “immediate” mode with typically in the region of 5ms lag and maybe 1ms precision. That’s pretty good compared to the other options that have a lag of 20ms upwards and several ms variability.

BUT, on top of that, PTB allows you to preschedule your sound to occur at a particular point in time (e.g. when the trigger is due to be sent or when the screen is due to flip) and the PTB engine will then prepare all the buffers ready to go and will also account for the known latencies in the card. With this method the PTB engine is capable of sub-ms precision and even sub-ms lag!

Of course, capable doesn’t mean it’s happening in your case. It can depend on many things about the local operating system and hardware. You should test it yourself for your kit, but here is an example of a standard Win10 box using built-in audio (not a fancy audio card):

![Fig. 1: Sub-ms audio timing with standard audio on Win10. Yellow trace is a 440 Hz tone played at 48 kHz with PTB engine. Cyan trace is the trigger (from a Labjack output). Gridlines are set to 1 ms.](image-url)
8.21.3 Preschedule your sound

The most precise way to use the PTB audio backend is to preschedule the playing of a sound. By doing this PTB can actually take into account both the time taken to load the sound (it will preload ready) and also the time taken by the hardware to start playing it.

To do this you can call `play()` with an argument called `when`. The `when` argument needs to be in the PsychToolBox clock timebase which can be accessed by using `psychtoolbox.GetSecs()` if you want to play sound at an arbitrary time (not in sync with a window flip)

For instance:

```python
import psychtoolbox as ptb
from psychopy import sound

mySound = sound.Sound('A')
now = ptb.GetSecs()
mySound.play(when=now+0.5)  # play in EXACTLY 0.5s
```

or using `Window.getFutureFlipTime(clock='ptb')` if you want a synchronized time:

```python
import psychtoolbox as ptb
from psychopy import sound, visual

mySound = sound.Sound('A')
win = visual.Window()
win.flip()
nextFlip = win.getFutureFlipTime(clock='ptb')
mySound.play(when=nextFlip)  # sync with screen refresh
```

The precision of that timing is still dependent on the PTB Audio Latency Modes and can obviously not work if the delay before the requested time is not long enough for the requested mode (e.g. if you request that the sound starts on the next refresh but set the latency mode to be 0 (which has a lag of around 300 ms) then the timing will be very poor.

8.21.4 PTB Audio Latency Modes

When using the PTB backend you get the option to choose the Latency Mode, referred to in PsychToolBox as the `req_latency_class`.

PsychoPy uses Mode 3 in as a default, assuming that you want low latency and you don’t care if other applications can’t play sound at the same time (don’t listen to iTunes while running your study!)

The modes are as follows:

0 : **Latency not important**  For when it really doesn’t matter. Latency can easily be in the region of 300ms!

1 : **Share low-latency access**  Tries to use a low-latency setup in combination with other applications. Latency usually isn’t very good and in MS Windows the sound you play must be the same sample rate as any other application that is using the sound system (which means you usually get restricted to exactly 48000 instead of 44100).

2 : **Exclusive mode low-latency**  Takes control of the audio device you’re using and dominates it. That can cause some problems for other apps if they’re trying to play sounds at the same time.

3 : **Aggressive exclusive mode**  As Mode 2 but with more aggressive settings to prioritise our use of the card over all others. **This is the recommended mode for most studies**
4: Critical mode  As Mode 3 except that, if we fail to be totally dominant, then raise an error rather than just accepting our slightly less dominant status.

8.21.5 Sound Classes

PTB Sound

class psychopy.sound.backend_ptb.SoundPTB(value='C', secs=0.5, octave=4, stereo=-1, volume=1.0, loops=0, sampleRate=None, blockSize=128, preBuffer=-1, hamming=True, startTime=0, stopTime=-1, name='', autoLog=True, syncToWin=None)

Play a variety of sounds using the new PsychPortAudio library

Parameters

• value – note name (“C”, “Bfl”), filename or frequency (Hz)
• secs – duration (for synthesised tones)
• octave – which octave to use for note names (4 is middle)
• stereo – -1 (auto), True or False to force sounds to stereo or mono
• volume – float 0-1
• loops – number of loops to play (-1=forever, 0=single repeat)
• sampleRate – sample rate for synthesized tones
• blockSize – the size of the buffer on the sound card (small for low latency, large for stability)
• preBuffer – integer to control streaming/buffering - -1 means store all - 0 (no buffer) means stream from disk - potentially we could buffer a few secs(!?)
• hamming – boolean (default True) to indicate if the sound should be apodized (i.e., the onset and offset smoothly ramped up from down to zero). The function apodize uses a Hanning window, but arguments named ‘hamming’ are preserved so that existing code is not broken by the change from Hamming to Hanning internally. Not applied to sounds from files.
• startTime – for sound files this controls the start of snippet
• stopTime – for sound files this controls the end of snippet
• name – string for logging purposes
• autoLog – whether to automatically log every change
• syncToWin – if you want start/stop to sync with win flips add this

_EOL (reset=True, log=True)
  Function called on End Of Stream

_channelCheck (array)
  Checks whether stream has fewer channels than data. If True, ValueError

_getDefaultSampleRate ()
  Check what streams are open and use one of these

pause ()
  Stop the sound but play will continue from here if needed
play (loops=None, when=None, log=True)
Start the sound playing

setSound (value, secs=0.5, octave=4, hamming=None, log=True)
Set the sound to be played.

Often this is not needed by the user - it is called implicitly during initialisation.

Parameters

value: can be a number, string or an array:

- If it’s a number between 37 and 32767 then a tone will be generated at that frequency in Hz.
- It could be a string for a note (‘A’, ‘Bfl’, ‘B’, ‘C’, ‘Csh’. …). Then you may want to specify which octave.
- Or a string could represent a filename in the current location, or mediaLocation, or a full path combo
- Or by giving an Nx2 numpy array of floats (-1:1) you can specify the sound yourself as a waveform

secs: duration (only relevant if the value is a note name or a frequency value)

octave: is only relevant if the value is a note name. Middle octave of a piano is 4. Most computers won’t output sounds in the bottom octave (1) and the top octave (8) is generally painful

property status
status gives a simple value from psychopy.constants to indicate NOT_STARTED, STARTED, FINISHED, PAUSED

Psychtoolbox sounds also have a statusDetailed property with further info

stop (reset=True, log=True)
Stop the sound and return to beginning

property stream
Read-only property returns the the stream on which the sound will be played

property track
The track on the master stream to which we belong

SoundDevice Sound

class psychopy.sound.backend_sounddevice.SoundDeviceSound (value='C', secs=0.5, octave=4, stereo=-1, volume=1.0, loops=0, sampleRate=None, blockSize=128, preBuffer=-1, hamming=True, startTime=0, stopTime=-1, name='', autoLog=True)

Play a variety of sounds using the new SoundDevice library

Parameters

- value – note name (“C”,”Bfl”), filename or frequency (Hz)
- **secs** – duration (for synthesised tones)
- **octave** – which octave to use for note names (4 is middle)
- **stereo** – -1 (auto), True or False to force sounds to stereo or mono
- **volume** – float 0-1
- **loops** – number of loops to play (-1=forever, 0=single repeat)
- **sampleRate** – sample rate (for synthesized tones)
- **blockSize** – the size of the buffer on the sound card (small for low latency, large for stability)
- **preBuffer** – integer to control streaming/buffering - -1 means store all - 0 (no buffer) means stream from disk - potentially we could buffer a few secs(!?)
- **hamming** – boolean (default True) to indicate if the sound should be apodized (i.e., the onset and offset smoothly ramped up from down to zero). The function apodize uses a Hanning window, but arguments named ‘hamming’ are preserved so that existing code is not broken by the change from Hamming to Hanning internally. Not applied to sounds from files.
- **startTime** – for sound files this controls the start of snippet
- **stopTime** – for sound files this controls the end of snippet
- **name** – string for logging purposes
- **autoLog** – whether to automatically log every change

_EOF (reset=True)
Function called on End Of Stream

channelCheck (array)
Checks whether stream has fewer channels than data. If True, ValueError

pause()
Stop the sound but play will continue from here if needed

play(loops=None, when=None)
Start the sound playing

Parameters when (not used) – Included for compatibility purposes

setSound (value, secs=0.5, octave=4, hamming=None, log=True)
Set the sound to be played.

Often this is not needed by the user - it is called implicitly during initialisation.

Parameters

value: can be a number, string or an array:

- If it’s a number between 37 and 32767 then a tone will be generated at that frequency in Hz.
- It could be a string for a note (‘A’, ‘Bfl’, ‘B’, ‘C’, ‘Csh’. …). Then you may want to specify which octave.
- Or a string could represent a filename in the current location, or mediaLocation, or a full path combo
- Or by giving an Nx2 numpy array of floats (-1:1) you can specify the sound yourself as a waveform

secs: duration (only relevant if the value is a note name or a frequency value)
octave: is only relevant if the value is a note name. Middle octave of a piano is 4. Most computers won’t output sounds in the bottom octave (1) and the top octave (8) is generally painful

stop (reset=True)
Stop the sound and return to beginning

property stream
Read-only property returns the the stream on which the sound will be played

Pyo Sound

class psychopy.sound.backend_pyo.SoundPyo (value='C', secs=0.5, octave=4, stereo=True, volume=1.0, loops=0, sampleRate=44100, bits=16, hamming=True, start=0, stop=-1, name='', autoLog=True)

Create a sound object, from one of MANY ways.

type: can be a number, string or an array:

• If it’s a number between 37 and 32767 then a tone will be generated at that frequency in Hz.
• It could be a string for a note (‘A’, ‘Bfl’, ‘B’, ‘C’, ‘Csh’, …). Then you may want to specify which octave as well
• Or a string could represent a filename in the current location, or mediaLocation, or a full path combo
• Or by giving an Nx2 numpy array of floats (-1:1) you can specify the sound yourself as a waveform

By default, a Hanning window (5ms duration) will be applied to a generated tone, so that onset and offset are smoother (to avoid clicking). To disable the Hanning window, set hamming=False.

secs: Duration of a tone. Not used for sounds from a file.

start [float] Where to start playing a sound file; default = 0s (start of the file).

stop [float] Where to stop playing a sound file; default = end of file.

octave: is only relevant if the value is a note name. Middle octave of a piano is 4. Most computers won’t output sounds in the bottom octave (1) and the top octave (8) is generally painful

stereo: True (= default, two channels left and right), False (one channel)

volume: loudness to play the sound, from 0.0 (silent) to 1.0 (max). Adjustments are not possible during playback, only before.

loops [int] How many times to repeat the sound after it plays once. If loops == -1, the sound will repeat indefinitely until stopped.

sampleRate (= 44100): if the psychopy.sound.init() function has been called or if another sound has already been created then this argument will be ignored and the previous setting will be used

bits: has no effect for the pyo backend

hamming: boolean (default True) to indicate if the sound should be apodized (i.e., the onset and offset smoothly ramped up from down to zero). The function apodize uses a Hanning window, but arguments named ‘hamming’ are preserved so that existing code is not broken by the change from Hamming to Hanning internally. Not applied to sounds from files.

play (loops=None, autoStop=True, log=True, when=None)
Starts playing the sound on an available channel.
loops [int] How many times to repeat the sound after it plays once. If loops == -1, the sound will repeat indefinitely until stopped.

when: not used but included for compatibility purposes

For playing a sound file, you cannot specify the start and stop times when playing the sound, only when creating the sound initially.

Playing a sound runs in a separate thread i.e. your code won’t wait for the sound to finish before continuing. To pause while playing, you need to use a `psychopy.core.wait(mySound.getDuration())`. If you call `play()` while something is already playing the sounds will be played over each other.

stop (log=True)
Stops the sound immediately

**pygame Sound**

```python
class psychopy.sound.backend_pygame.SoundPygame(value='C', secs=0.5, octave=4, sampleRate=44100, bits=16, name='', autoLog=True, loops=0, stereo=True, hamming=False)
```

Create a sound object, from one of many ways.

**Parameters**

**value:** can be a number, string or an array:

- If it's a number between 37 and 32767 then a tone will be generated at that frequency in Hz.
- It could be a string for a note ('A', 'Bfl', 'B', 'C', 'Csh', ...). Then you may want to specify which octave as well
- Or a string could represent a filename in the current location, or mediaLocation, or a full path combo
- Or by giving an Nx2 numpy array of floats (-1:1) you can specify the sound yourself as a waveform

**secs:** duration (only relevant if the value is a note name or a frequency value)

**octave:** is only relevant if the value is a note name. Middle octave of a piano is 4. Most computers won't output sounds in the bottom octave (1) and the top octave (8) is generally painful

**sampleRate(=44100):** If a sound has already been created or if the

**bits(=16):** Pygame uses the same bit depth for all sounds once initialised

**fadeOut (mSecs)**
Fades out the sound (when playing) over mSecs. Don’t know why you would do this in psychophysics but it’s easy and fun to include as a possibility :)

**getDuration ()**
Get's the duration of the current sound in secs

**getVolume ()**
Returns the current volume of the sound (0.0:1.0)

**play (fromStart=True, log=True, loops=None, when=None)**
Starts playing the sound on an available channel.

**Parameters**

**fromStart** [bool] Not yet implemented.
log [bool] Whether or not to log the playback event.

loops [int] How many times to repeat the sound after it plays once. If loops == -1, the sound will repeat indefinitely until stopped.

when: not used but included for compatibility purposes

Notes If no sound channels are available, it will not play and return None. This runs off a separate thread i.e. your code won’t wait for the sound to finish before continuing. You need to use a psychopy.core.wait() command if you want things to pause. If you call play() while something is already playing the sounds will be played over each other.

setVolume (newVol, log=True)
Sets the current volume of the sound (0.0:1.0)

stop (log=True)
Stops the sound immediately

8.22 psychopy.voicekey - Real-time sound processing

(Available as of version 1.83.00)

8.22.1 Overview

Hardware voice-keys are used to detect and signal acoustic properties in real time, e.g., the onset of a spoken word in word-naming studies. PsychoPy provides two virtual voice-keys, one for detecting vocal onsets and one for vocal offsets. All PsychoPy voice-keys can take their input from a file or from a microphone. Event detection is typically quite similar in both cases.

The base class is very general, and is best thought of as providing a toolkit for developing a wide range of custom voice-keys. It would be possible to develop a set of voice-keys, each optimized for detecting different initial phonemes. Band-pass filtered data and zero-crossing counts are computed in real-time every 2ms.

8.22.2 Voice-Keys

class psychopy.voicekey.OnsetVoiceKey (sec=0, file_out='', file_in='', **config)
Class for speech onset detection.

Uses bandpass-filtered signal (100-3000Hz). When the voice key trips, the best voice-onset RT estimate is saved as self.event_onset, in sec.

Parameters

sec: duration to record in seconds

file_out: name for output filename (for microphone input)

file_in: name of input file for sound source (not microphone)

config: kwargs dict of parameters for configuration. defaults are:

‘msPerChunk’: 2; duration of each real-time analysis chunk, in ms

‘signaler’: default None

‘autosave’: True; False means manual saving to a file is still possible (by calling .save() but not called automatically upon stopping

8.22. psychopy.voicekey - Real-time sound processing
`chnl_in` [microphone channel:] see psychopy.sound.backend.get_input_devices()

`chnl_out`: not implemented; output device to use

`start`: 0, select section from a file based on (start, stop) time

`stop`: -1, end of file (default)

`vol`: 0.99, volume 0..1

`low`: 100, Hz, low end of bandpass; can vary for M/F speakers

`high`: 3000, Hz, high end of bandpass

`threshold`: 10

`baseline`: 0; 0 = auto-detect; give a non-zero value to use that

`more_processing`: True; compute more stats per chunk including bandpass;  try

    False if 32-bit python can’t keep up

    ‘zero_crossings’: True

_do_chunk ()
Core function to handle a chunk (= a few ms) of input.

There can be small temporal gaps between or within chunks, i.e., slippage. Adjust several parameters until this is small: msPerChunk, and what processing is done within _process().

A trigger (_chunktrig) signals that _chunktable has been filled and has set _do_chunk as the function to call upon triggering. _play() the trigger again to start recording the next chunk.

_process (chunk)
Calculate and store basic stats about the current chunk.

This gets called every chunk – keep it efficient, esp 32-bit python

_set_baseline ()
Set self.baseline = rms(silent period) using _baselinetable data.

Called automatically (via pyo trigger) when the baseline table is full. This is better than using chunks (which have gaps between them) or the whole table (which can be very large = slow to work with).

_set_defaults ()
Set remaining defaults, initialize lists to hold summary stats

_set_signaler ()
Set the signaler to be called by trip()

_set_source ()
Data source: file_in, array, or microphone

_set_tables ()
Set up the pyo tables (allocate memory, etc).

One source -> three pyo tables: chunk=short, whole=all, baseline. triggers fill tables from self._source; make triggers in .start()

detect ()
Trip if recent audio power is greater than the baseline.

join (sec=None)
Sleep for sec or until end-of-input, and then call stop().
save (ftype='', dtype='int16')
Save new data to file, return the size of the saved file (or None).

The file format is inferred from the filename extension, e.g., flac. This will be overridden by the ftype if one is provided; defaults to wav if nothing else seems reasonable. The optional dtype (e.g., int16) can be any of the sample types supported by pyo.

**property slippage**
Ratio of the actual (elapsed) time to the ideal time.

Ideal ratio = 1 = sample-perfect acquisition of msPerChunk, without any gaps between or within chunks. 1. / slippage is the proportion of samples contributing to chunk stats.

**Type** Diagnostic

start (silent=False)
Start reading and processing audio data from a file or microphone.

**property started**
Boolean property, whether .start() has been called.

stop()
Stop a voice-key in progress.

Ends and saves the recording if using microphone input.

wait_for_event (plus=0)
Start, join, and wait until the voice-key trips, or it times out.

Optionally wait for some extra time, plus, before calling stop().

class psychopy.voicekey.OffsetVoiceKey (sec=10, file_out='', file_in='', delay=0.3, **kwargs)
Class to detect the offset of a single-word utterance.

Record and ends the recording after speech offset. When the voice key trips, the best voice-offset RT estimate is saved as self.event_offset, in seconds.

**Parameters**

sec: duration of recording in the absence of speech or other sounds.

delay: extra time to record after speech offset, default 0.3s.

The same methods are available as for class OnsetVoiceKey.

### 8.22.3 Signal-processing functions

Several utility functions are available for real-time sound analysis.

psychopy.voicekey.smooth (data, win=16, tile=True)
Running smoothed average, via convolution over win window-size.

tile with the mean at start and end by default; otherwise replace with 0.

psychopy.voicekey.bandpass (data, low=80, high=1200, rate=44100, order=6)
Return bandpass filtered data.

psychopy.voicekey.rms (data)
Basic audio-power measure: root-mean-square of data.

Identical to std when the mean is zero; faster to compute just rms.

psychopy.voicekey.std (data)
Like rms, but also subtracts the mean (= slower).
psychoPy.voicekey.

8.22.4 Sound file I/O

Several helper functions are available for converting and saving sound data from several data formats (numpy arrays, pyo tables) and file formats. All file formats that pyo supports are available, including wav, flac for lossless compression. mp3 format is not supported (but you can convert to .wav using another utility).

psychoPy.voicekey.

8.23 psychoPy.web - Web methods

8.23.1 Test for access

psychoPy.web.


8.23.2 Proxy set-up and testing

psychopy.web.setupProxy (log=True)

Set up the urllib proxy if possible.

The function will use the following methods in order to try and determine proxies:

1. standard urllib.request.urlopen (which will use any statically-defined http-proxy settings)
2. previous stored proxy address (in prefs)
3. proxy.pac files if these have been added to system settings
4. auto-detect proxy settings (WPAD technology)

Returns True (success) or False (failure)

Further information:
TIMING INFORMATION FOR PSYCHOpy

There is documentation about how to optimize timing in PsychoPy at *Timing Issues and synchronisation*.

We recently ran a study testing the timing on a wide range of software packages, online and offline. The data for that study are available below:

### 9.1 Mega-timing study data

Here are the data summaries for our paper, *The timing mega-study: comparing a range of experiment generators, both lab-based and online*.

You can read the full preprint of the paper at [https://peerj.com/articles/9414/](https://peerj.com/articles/9414/)

#### 9.1.1 Table2: lab-based timing data

This is a sortable version of Table2 from *The timing mega-study: comparing a range of experiment generators, both lab-based and online*.

**Timing summaries of lab-based software** by package and platform. The Var(iability) measures are the inter-trial standard deviations of the various latencies for that configuration. The table is sorted by the mean of those variabilities (Mean Var). The Lag/Bias measures are the mean latency values, for that configuration. In the case of audiovisual sync, a negative bias indicates the audio lead the visual stimulus, a positive bias means the visual lead the audio. Each of the values with a hyperlink will lead to a plot of the distribution of values leading to that summary value.

#### 9.1.2 Table3: online timing data

This is a sortable version of Table3 from *The timing mega-study: comparing a range of experiment generators, both lab-based and online*.

**Timing summaries of web-based software** by package, platform, and browser. The Var(iability) measures are the inter-trial standard deviations of the various latencies for that configuration. The table is sorted by the mean of those variabilities (Mean Var). The Lag/Bias measures are the mean latency values, for that configuration. In the case of audiovisual sync, a negative bias indicates the audio lead the visual stimulus, a positive bias means the visual lead the audio. Each of the values with a hyperlink will lead to a plot of the distribution of values leading to that summary value.
Regrettably, PsychoPy is not bug-free. Running on all possible hardware and all platforms is a big ask. That said, a huge number of bugs have been resolved by the fact that there are literally 1000s of people using the software that have contributed either bug reports and/or fixes.

Below are some of the more common problems and their workarounds, as well as advice on how to get further help.

### 10.1 The application doesn’t start

You may find that you try to launch the PsychoPy application, the splash screen appears and then goes away and nothing more happens. What this means is that an error has occurred during startup itself.

Commonly, the problem is that a preferences file is somehow corrupt. To fix that see *Cleaning preferences and app data*, below.

If resetting the preferences files doesn’t help then we need to get to an error message in order to work out why the application isn’t starting. The way to get that message depends on the platform (see below).

**Windows users** (starting from the Command Prompt):

1. Did you get an error message that “This application failed to start because the application configuration is incorrect. Reinstalling the application may fix the problem”? If so that indicates you need to update your .NET installation to SP1.

2. **open a Command Prompt (terminal):**
   1. go to the Windows Start menu
   2. select Run… and type in cmd <Return>

3. paste the following into that window (Ctrl-V doesn’t work in Cmd.exe but you can right-click and select Paste):

   ```
   "C:\Program Files\PsychoPy2\python.exe" -m psychopy.app.psychopyApp
   ```

4. when you hit <return> you will hopefully get a moderately useful error message that you can *Contribute to the Forum (mailing list)*

**Mac users:**

1. open the Console app (open spotlight and type console)

2. if there are a huge number of messages there you might find it easiest to clear them (the brush icon) and then start PsychoPy again to generate a new set of messages
10.2 I run a Builder experiment and nothing happens

An error message may have appeared in a dialog box that is hidden (look to see if you have other open windows somewhere).

An error message may have been generated that was sent to output of the Coder view:

1. go to the Coder view (from the Builder>View menu if not visible)
2. if there is no Output panel at the bottom of the window, go to the View menu and select Output
3. try running your experiment again and see if an error message appears in this Output view

If you still don’t get an error message but the application still doesn’t start then manually turn off the viewing of the Output (as below) and try the above again.

10.3 Manually turn off the viewing of output

Very occasionally an error will occur that crashes the application after the application has opened the Coder Output window. In this case the error message is still not sent to the console or command prompt.

To turn off the Output view so that error messages are sent to the command prompt/terminal on startup, open your appData.cfg file (see Cleaning preferences and app data), find the entry:

```
[coder]
showOutput = True
```

and set it to `showOutput = False` (note the capital ‘F’).

10.4 Use the source (Luke?)

PsychoPy comes with all the source code included. You may not think you’re much of a programmer, but have a go at reading the code. You might find you understand more of it than you think!

To have a look at the source code do one of the following:

- when you get an error message in the Coder click on the hyperlinked error lines to see the relevant code
- on Windows
  - go to `<location of PsychoPy app>\Lib\site-packages\psychopy`
  - have a look at some of the files there
- on Mac
  - right click the PsychoPy app and select Show Package Contents
  - navigate to `Contents/Resources/lib/pythonX.X/psychopy`
10.5 Cleaning preferences and app data

Every time you shut down PsychoPy (by normal means) your current preferences and the state of the application (the location and state of the windows) are saved to disk. If PsychoPy is crashing during startup you may need to edit those files or delete them completely.

The exact location of those files varies by machine but on windows it will be something like `%APPDATA%psychopy3` and on Linux/MacOS it will be something like `~/.psychopy3`. You can find it running this in the commandline (if you have multiple Python installations then make sure you change `python` to the appropriate one for PsychoPy:

```
python -c "from psychopy import prefs; print(prefs.paths['userPrefsDir'])"
```

Within that folder you will find `userPrefs.cfg` and `appData.cfg`. The files are simple text, which you should be able to edit in any text editor.

If the problem is that you have a corrupt experiment file or script that is trying and failing to load on startup, you could simply delete the `appData.cfg` file. Please also Contribute to the Forum (mailing list) a copy of the file that isn’t working so that the underlying cause of the problem can be investigated (google first to see if it’s a known issue).

10.6 Errors with getting/setting the Gamma ramp

There are two common causes for errors getting/setting gamma ramps depending on whether you’re running Windows or Linux (we haven’t seen these problems on Mac).

10.6.1 MS Windows bug in release 1903

In Windows release 1903 Microsoft added a bug that prevents getting/setting the gamma ramp. This only occurs in certain scenarios, like when the screen orientation is in portrait, or when it is extended onto a second monitor, but it does affect all versions of PsychoPy.

For the Windows bug the workarounds are as follows:

If you don’t need gamma correction then, as of PsychoPy 3.2.4, you can go to the preferences and set the `default-GammaFailPolicy` to be be ‘warn’ (rather than ‘abort’) and then your experiment will still at least run, just without gamma correction.

If you do need gamma correction then there isn’t much that the PsychoPy team can do until Microsoft fixes the underlying bug. You’ll need to do one of:

- Not using Window 1903 (e.g. revert the update) until a fix is listed on the status of the gamma bug
- Altering your monitor settings in Windows (e.g. turning off extended desktop) until it works. Unfortunately that might mean you can’t use dual independent displays for vision science studies until Microsoft fix it.
10.6.2 Linux missing xorg.conf

On Linux some systems appear to be missing a configuration file and adding this back in and restarting should fix things. Create the following file (including the folders as needed):

/etc/X11/xorg.conf.d/20-intel.conf

and put the following text inside (assuming you have an intel card, which is where we've typically seen the issue crop up):

```
Section "Device"
    Identifier "Intel Graphics"
    Driver "intel"
EndSection
```

For further information on the discussion of this (Linux) issue see https://github.com/psychopy/psychopy/issues/2061
The PsychoPy Alerts system is designed to provide you information of varying levels about things that may be a concern in your study (but equally may not be - it depends on your study!). Alerts show up within PsychoPy with a code and that code should take you here for detailed information about the root cause of the problem, the workarounds, and the type of study that might be badly affected by this issue.

11.1 2xxx: Issues with units

These issue usually result in problems with stimuli not appearing (too fast, too small, off-screen) or being an unexpected size or color.

11.1.1 2115: Stimulus size is bigger than the window dimensions

Synopsis

Your stimulus size exceeds the X or Y window dimensions. Stimuli sized greater than the window size will not be completely visible.

Details

This issue is often caused by an inconsistency between the units of your stimulus and the values being requested. For instance a size of 3, when the units are deg is sensible (3 degrees would be within the screen dimensions) but a size of 3 when the units are height would not be sensible (3 times bigger than the height of the screen).

PsychoPy versions affected

All versions.
Solutions

Check the size and the :ref:`units of the stimulus <units>` carefully. You may also need to check the monitor calibration if you’re using units that depend on the monitor size and resolution (like \textit{cm} and \textit{deg}).

11.1.2 2120: Stimulus size is smaller than 1 pixel

Synopsis

Stimuli size requested is smaller than 1 pixel on X and/or Y dimensions. Stimuli sized smaller than 1 pixel will not be visible.

Details

This issue is often caused by an inconsistency between the units of your stimulus and the values being requested. For instance a size of 0.1, when the units are \textit{height} is sensible (1/10th the height of the screen) but a size of 0.1 when the units are \textit{pix} would not be sensible (1/10th of a pixel).

PsychoPy versions affected

All versions

Solutions

Check the size and the :ref:`units of the stimulus <units>` carefully. You may also need to check the monitor calibration if you’re using units that depend on the monitor size and resolution (like \textit{cm} and \textit{deg}).

11.1.3 2155: Stimulus position is beyond the bounds of the window

Synopsis

Your stimulus position exceeds the X or Y window dimensions. Stimuli centered beyond the window will not be completely visible.

Details

This issue is often caused by an inconsistency between the units of your stimulus and the values being requested. For instance a position of \((3, 0)\), when the units are \textit{deg} is sensible (3 degrees to the right of the screen center) but a position of \((3, 0)\) when the units are \textit{height} would not be sensible (3 times the height of the screen to the right of the center).
PsychoPy versions affected

All versions

Solutions

Check the position and the :ref:`units of the stimulus <units>` carefully. You may also need to check the monitor calibration if you’re using units that depend on the monitor size and resolution (like cm and deg).

11.2 3xxx: Issues with timing

There are many ways for timing to go wrong, and these alerts will warn you about some of them (but we do always recommend that you test your timing carefully with hardware devices if precise timing is important to your study).

11.2.1 3110: Stimulus duration is less than one screen refresh

Synopsis

Your stimulus is scheduled to last for a duration that can’t be achieved with a normal 60 Hz monitor. Duration will implicitly be round up (probably) to the next frame duration.

Requested start or stop times of visual components cannot be presented for times requested. Accurate presentation times must be in increments of your screen refresh rate.

Details

Stimuli can only be presented for a fixed number of screen refreshes. If you screen has a refresh rate of 60 Hz (common for standard monitors) then each screen refresh period lasts 1/60 s (roughly 16.6667 ms). That means you can present your stimulus for 16.7 ms but not for, say, 5 ms because that would require the stimulus to be presented for half of one screen refresh period.

PsychoPy versions affected

All versions.

Solutions

We recommend for brief stimuli that you simply specify your stimulus duration in terms of the number of frames it should be presented (e.g. 1, 2, 3, for 16.7, 33.3 and 50 ms respectively). That reminds you of what is possible and means that PsychoPy won’t have to guess about what to do when the desired duration isn’t achievable.

If you need stimuli to be presented for briefer durations than 16.7 ms then you should look into high-frame-rate displays (100, 120 and 144 Hz displays are all available). There are also now variable-frame-rate monitors that can vary the duration of each frame within limits. If you are already using a high- or variable-rate monitor then this alert may not be relevant to you.
11.2.2 3115: Stimulus duration is not possible on a standard monitor refresh

Synopsis

If using a 60Hz or 100Hz monitor, then for accurate presentation of visual stimuli, components must be presented in valid multiples of screen refresh for 60 Hz or 100 Hz.

Details

When presenting stimuli at, say, 60 Hz you stimulus can be presented for 1 frame (1/60 s = 16.667 ms), 2 frames (2/60 s = 33.333 ms) but not for intervening periods (20 ms is not possible because the stimulus would have to be presented for a little more than 1 frame, which isn’t physically possible on standard fixed-framerate monitors.

PsychoPy versions affected

All versions.

Solutions

We recommend for brief stimuli that you simply specify your stimulus duration in terms of the number of frames it should be presented (e.g. 1, 2, 3, for 16.7, 33.3 and 50 ms respectively). That reminds you of what is possible and means that PsychoPy won’t have to guess about what to do when the desired duration isn’t achievable.

If you need stimuli to be presented for briefer durations than 16.7 ms then you should look into high-frame-rate displays (100, 120 and 144 Hz displays are all available). There are also now variable-frame-rate monitors that can vary the duration of each frame within limits. If you are already using a high- or variable-rate monitor then this alert may not be relevant to you.

11.3 4xxx: Issues with Builder experiments

Issues specifically around Builder experiments doing unexpected things.

11.3.1 4105: Component start time exceeds its stop time

Synopsis

A component start time/frame exceeds the stop time/frame. This means that the component starts after it is due to finish and the stimulus will most likely not be shown at all.
PsychoPy versions affected

All versions.

Solutions

Check your start and stop time carefully, including the units being used. For example your stimulus might be set to start at a certain time (say 36 seconds) rather than frame number 36.

11.3.2 4115: Component start/stop in units of frames must be whole numbers

Synopsis

Component start and stop times/durations in frames must be given as whole numbers.

Details

Since it isn’t possible to start or stop a stimulus part-way through a screen refresh it would be unwise to request that PsychoPy attempts that.

PsychoPy versions affected

All versions.

Solutions

Check your stimulus start/stop time and set to an integer value instead of a decimal value.

11.3.3 4205: Probable syntax error detected in your Python code

Synopsis

Python syntax error found in your code component.

Details

This may be spurious in that the code check may have failed to understand something that is a valid syntax (syntax in Python can change according to version) but this will become clear if you run your experiment and it fails to run.
PsychoPy versions affected

All versions.

Solutions

Check the code carefully at the indicated line and on the few lines above. Check especially for things like un-matched parentheses or quote symbols.

11.3.4 4210: Probable syntax error detected in your JavaScript code

Synopsis

JavaScript syntax error found in your code component.

Details

This alert may be spurious in that the code check may have failed to understand something that is a valid syntax (syntax in JavaScript can change according to version) but this will become clear if you run your experiment and it fails to run.

PsychoPy versions affected

All versions.

Solutions

Check the code carefully at the indicated line and on the few lines above. Check especially for things like un-matched parentheses or quote symbols.

11.3.5 4305: Component is currently disabled in your experiment

Synopsis

You have a disabled Component in your experiment. This alert is created to inform users that disabled components will not be written to your experiment script. This may not be intentional, and will therefore affect your desired outcome.

Details

Most likely you disabled the Component deliberately while testing things out, but this Alert is making sure you remember that it isn't currently operational.
PsychoPy versions affected

>= 3.1.0

Solutions

Re-enable the Component in the Builder view or just ignore this Alert if you intended it to be disabled.

11.3.6 4310: Builder cannot check your parameter further

Synopsis

This alert is received when an integrity check has failed to calculate a parameter. Most commonly, this is because a variable from code, or a conditions file, has been encountered, and the value of the variable is not available to the integrity checking system.

Details

Longer description with optional links to further information

PsychoPy versions affected

>= 3.2.3

Solutions

This alert is for information only, and does not require any action.
Below are various tips/tricks/recipes/how-tos for PsychoPy. They involve something that is a little more involved than you would find in FAQs, but too specific for the manual as such (should they be there?).

12.1 Adding external modules to Standalone PsychoPy

You might find that you want to add some additional Python module/package to your Standalone version of PsychoPy. To do this you need to:

- download a copy of the package (make sure it's for Python 2.7 on your particular platform)
- unzip/open it into a folder
- add that folder to the path of PsychoPy by one of the methods below

Avoid adding the entire path (e.g. the site-packages folder) of separate installation of Python, because that may contain conflicting copies of modules that PsychoPy is also providing.

12.1.1 Using preferences

As of version 1.70.00 you can do this using the PsychoPy preferences/general. There you will find a preference for paths which can be set to a list of strings e.g. ['/Users/jwp/code', '~/code/thirdParty']

These only get added to the Python path when you import psychopy (or one of the psychopy packages) in your script.

12.1.2 Adding a .pth file

An alternative is to add a file into the site-packages folder of your application. This file should be pure text and have the extension .pth to indicate to Python that it adds to the path.

On win32 the site-packages folder will be something like:

C:/Program Files/PsychoPy2/lib/site-packages

On macOS you need to right-click the application icon, select 'Show Package Contents' and then navigate down to Contents/Resources/lib/pythonX.X. Put your .pth file here, next to the various libraries.

The advantage of this method is that you don’t need to do the import psychopy step. The downside is that when you update PsychoPy to a new major release you'll need to repeat this step (patch updates won't affect it though).
12.2 Animation

General question: How can I animate something?

Conceptually, animation just means that you vary some aspect of the stimulus over time. So the key idea is to draw something slightly different on each frame. This is how movies work, and the same principle can be used to create scrolling text, or fade-in / fade-out effects, and the like.

(copied & pasted from the email list; see the list for people's names and a working script.)

12.3 Scrolling text

Key idea: Vary the position of the stimulus across frames.

Question: How can I produce scrolling text (like html's <marquee behavior="scroll"> directive)?

Answer: PsychoPy has animation capabilities built-in (it can even produce and export movies itself (e.g. if you want to show your stimuli in presentations)). But here you just want to animate stimuli directly.

e.g. create a text stimulus. In the 'pos' (position) field, type:

[frameN, 0]

and select “set every frame” in the popup button next to that field.

Push the Run button and your text will move from left to right, at one pixel per screen refresh, but stay at a fixed y-coordinate. In essence, you can enter an arbitrary formula in the position field and the stimulus will be-redrawn at a new position on each frame. frameN here refers to the number of frames shown so far, and you can extend the formula to produce what you need.

You might find performance issues (jittering motion) if you try to render a lot of text in one go, in which case you may have to switch to using images of text.

I wanted my text to scroll from right to left. So if you keep your eyes in the middle of the screen the next word to read would come from the right (as if you were actually reading text). The original formula posted above scrolls the other way. So, you have to put a negative sign in front of the formula for it to scroll the other way. You have to change the units to pixel. Also, you have to make sure you have an end time set, otherwise it just flickers. I also set my letter height to 100 pixels. The other problem I had was that I wanted the text to start blank and scroll into the screen. So, I wrote

[2000-frameN, 0]

and this worked really well.

12.4 Fade-in / fade-out effects

Key idea: vary the opacity of the stimulus over frames.

Question: I'd like to present an image with the image appearing progressively and disappearing progressively too. How to do that?

Answer: The Patch stimulus has an opacity field. Set the button next to it to be “set every frame” so that its value can be changed progressively, and enter an equation in the box that does what you want.

e.g. if your screen refresh rate is 60 Hz, then entering:

frameN/120
would cycle the opacity linearly from 0 to 1.0 over 2s (it will then continue incrementing but it doesn’t seem to matter if the value exceeds 1.0).

Using a code component might allow you to do more sophisticated things (e.g. fade in for a while, hold it, then fade out). Or more simply, you just create multiple successive Patch stimulus components, each with a different equation or value in the opacity field depending on their place in the timeline.

### 12.5 Building an application from your script

A lot of people ask how they can build a standalone application from their Python script. Usually this is because they have a collaborator and want to just send them the experiment.

In general this is not advisable - the resulting bundle of files (single file on macOS) will be on the order of 100Mb and will not provide the end user with any of the options that they might need to control the task (for example, Monitor Center won’t be provided so they can’t to calibrate their monitor). A better approach in general is to get your collaborator to install the Standalone PsychoPy on their own machine, open your script and press run. (You don’t send a copy of Microsoft Word when you send someone a document - you expect the reader to install it themself and open the document).

Nonetheless, it is technically possible to create exe files on Windows, and Ricky Savjani (savjani at bcm.edu) has kindly provided the following instructions for how to do it. A similar process might be possible on macOS using py2app - if you’ve done that then feel free to contribute the necessary script or instructions.

#### 12.5.1 Using py2exe to build an executable

Instructions:

2. Develop your PsychoPy script as normal
3. Copy this setup.py file into the same directory as your script
4. Change the Name of progName variable in this file to the Name of your desired executable program name
5. Use cmd (or bash, terminal, etc.) and run the following in the directory of your the two files: python setup.py py2exe
6. Open the ‘dist’ directory and run your executable

A example setup.py script:

```python
# Created 8-09-2011
# Ricky Savjani
# (savjani at bcm.edu)

# import necessary packages
from distutils.core import setup
import os, matplotlib
import py2exe

# the name of your .exe file
progName = 'MultipleSchizophrenia.py'

# Initialize Holder Files
preference_files = []
app_files = []
my_data_files=matplotlib.get_py2exe_datafiles()
```

(continues on next page)
#define which files you want to copy for data_files

```python
for files in os.listdir('C:\Program Files\PsychoPy2\Lib\site-packages\PsychoPy-1.65.00-py2.6.egg\psychopy\preferences\'):  
    f1 = 'C:\Program Files\PsychoPy2\Lib\site-packages\PsychoPy-1.65.00-py2.6.egg\psychopy\preferences\' + files  
    preference_files.append(f1)
```

#if you might need to import the app files

```python
# for files in os.listdir('C:\Program Files\PsychoPy2\Lib\site-packages\PsychoPy-1.65.00-py2.6.egg\psychopy\app\'):
#    f1 = 'C:\Program Files\PsychoPy2\Lib\site-packages\PsychoPy-1.65.00-py2.6.egg\psychopy\app\' + files
#    app_files.append(f1)
```

#combine the files

```python
all_files = ["psychopy\preferences", preference_files], ("psychopy\app", app_files), my_data_files[0]]
```

#define the setup

```python
setup(
    console=[progName],
    data_files = all_files,
    options = {
        "py2exe":{
            "skip_archive": True,
            "optimize": 2
        }
    }
)
```

## 12.6 Builder - providing feedback

If you’re using the Builder then the way to provide feedback is with a Code Component to generate an appropriate message (and then a text to present that message). PsychoPy will be keeping track of various aspects of the stimuli and responses for you throughout the experiment and the key is knowing where to find those.

The following examples assume you have a Loop called trials, containing a Routine with a Keyboard Component called key_resp. Obviously these need to be adapted in the code below to fit your experiment.

**Note:** The following generate strings use python ‘formatted strings’. These are very powerful and flexible but a little strange when you aren’t used to them (they contain odd characters like %.2f). See Generating formatted strings for more info.
12.6.1 Feedback after a trial

This is actually demonstrated in the demo, *ExtendedStroop* (in the Builder>demos menu, unpack the demos and then look in the menu again, tada!)

If you have a Keyboard Component called *key_resp* then, after every trial you will have the following variables:

```
keyResp.keys  # a python list of keys pressed
keyResp.rt    # the time to the first key press
keyResp.corr  # None, 0 or 1, if you are using 'store correct'
```

To create your *msg*, insert the following into the ‘start experiment’ section of the *Code Component*:

```
msg='doh!' # if this comes up we forgot to update the msg!
```

and then insert the following into the *Begin Routine* section (this will get run every repeat of the routine):

```
if not keyResp.keys :
    msg="Failed to respond"
elif resp.corr: # stored on last run routine
    msg="Correct! RT=%.3f" %(resp.rt)
else:
    msg="Oops! That was wrong"
```

12.6.2 Feedback after a block

In this case the feedback routine would need to come after the loop (the block of trials) and the message needs to use the stored data from the loop rather than the *key_resp* directly. Accessing the data from a loop is not well documented but totally possible.

In this case, to get all the keys pressed in a *numpy* array:

```
trials.data['keyResp.keys']  # numpy array with size=[ntrials,ntypes]
```

If you used the ‘Store Correct’ feature of the Keyboard Component (and told psychopy what the correct answer was) you will also have a variable:

```
# numpy array storing whether each response was correct (1) or not (0)
trials.data['resp.corr']
```

So, to create your *msg*, insert the following into the ‘start experiment’ section of the *Code Component*:

```
msg='doh!' # if this comes up we forgot to update the msg!
```

and then insert the following into the *Begin Routine* section (this will get run every repeat of the routine):

```
nCorr = trials.data['keyResp.corr'].sum()  # std(), mean() also available
meanRt = trials.data['keyResp.rt'].mean()
msg = "You got %i trials correct (rt=%.2f)" %(nCorr,meanRt)
```
12.6.3 Draw your message to the screen

Using one of the above methods to generate your msg in a Code Component, you then need to present it to the participant by adding a text to your feedback Routine and setting its text to $msg.

**Warning:** The Text Component needs to be below the Code Component in the Routine (because it needs to be updated after the code has been run) and it needs to *set every repeat*.

12.7 Builder - terminating a loop

People often want to terminate their Loops before they reach the designated number of trials based on subjects’ responses. For example, you might want to use a Loop to repeat a sequence of images that you want to continue until a key is pressed, or use it to continue a training period, until a criterion performance is reached.

To do this you need a Code Component inserted into your routine. All loops have an attribute called finished which is set to True or False (in Python these are really just other names for 1 and 0). This finished property gets checked on each pass through the loop. So the key piece of code to end a loop called trials is simply:

```
trials.finished=True  # or trials.finished=1 if you prefer
```

Of course you need to check the condition for that with some form of if statement.

**Example 1:** You have a change-blindness study in which a pair of images flashes on and off, with intervening blanks, in a loop called presentationLoop. You record the key press of the subject with a Keyboard Component called resp1. Using the ‘ForceEndTrial’ parameter of resp1 you can end the current cycle of the loop but to end the loop itself you would need a Code Component. Insert the following two lines in the End Routine parameter for the Code Component, which will test whether more than zero keys have been pressed:

```
if resp1.keys is not None and len(resp1.keys)>0 :
    trials.finished=1
```

or:

```
if resp1.keys :
    presentationLoop.finished=1
```

**Example 2:** Sometimes you may have more possible trials than you can actually display. By default, a loop will present all possible trials (nReps * length-of-list). If you only want to present the first 10 of all possible trials, you can use a code component to count how many have been shown, and then finish the loop after doing 10.

This example assumes that your loop is named ‘trials’. You need to add two things, the first to initialize the count, and the second to update and check it.

**Begin Experiment:**

```
myCount = 0
```

**Begin Routine:**

```
myCount = myCount + 1
if myCount > 10:
    trials.finished = True
```
Note: In Python there is no end to finish an if statement. The content of the if or of a for-loop is determined by the indentation of the lines. In the above example only one line was indented so that one line will be executed if the statement evaluates to True.

12.8 Installing PsychoPy in a classroom (administrators)

For running PsychoPy in a classroom environment it is probably preferable to have a ‘partial’ network installation. The PsychoPy library features frequent new releases, including bug fixes and you want to be able to update machines with these new releases. But PsychoPy depends on many other python libraries (over 200Mb in total) that tend not to change so rapidly, or at least not in ways critical to the running of experiments. If you install the whole PsychoPy application on the network then all of this data has to pass backwards and forwards, and starting the app will take even longer than normal.

The basic aim of this document is to get to a state whereby;

- Python and the major dependencies of PsychoPy are installed on the local machine (probably a disk image to be copied across your lab computers)
- PsychoPy itself (only ~2Mb) is installed in a network location where it can be updated easily by the administrator
- a file is created in the installation that provides the path to the network drive location
- Start-Menu shortcuts need to be set to point to the local Python but the remote PsychoPy application launcher

Once this is done, the vast majority of updates can be performed simply by replacing the PsychoPy library on the network drive.

12.8.1 1. Install dependencies locally

Download the latest version of the Standalone PsychoPy distribution, and run as administrator. This will install a copy of Python and many dependencies to a default location of

C:\Program Files\PsychoPy2\

12.8.2 2. Move the PsychoPy to the network

You need a network location that is going to be available, with read-only access, to all users on your machines. You will find all the contents of PsychoPy itself at something like this (version dependent obviously):

C:\Program Files\PsychoPy2\Lib\site-packages\PsychoPy-1.70.00-py2.6.egg

Move that entire folder to your network location and call it psychopyLib (or similar, getting rid of the version-specific part of the name). Now the following should be a valid path:

<NEXTWORK_LOC>\psychopyLib\psychopy
12.8.3 3. Update the Python path

The Python installation (in C:\Program Files\PsychoPy2) needs to know about the network location. If Python finds a text file with extension .pth anywhere on its existing path then it will add to the path any valid paths it finds in the file. So create a text file that has one line in it:

\<NETWORK_LOC>\psychopyLib

You can test if this has worked. Go to C:\Program Files\PsychoPy2 and double-click on python.exe. You should get a Python terminal window come up. Now try:

```python
>>> import psychopy
```

If psychopy is not found on the path then there will be an import error. Try adjusting the .pth file, restarting python.exe and importing again.

12.8.4 4. Update the Start Menu

The shortcut in the Windows Start Menu will still be pointing to the local (now non-existent) PsychoPy library. Right-click it to change properties and set the shortcut to point to something like:

"C:\Program Files\PsychoPy2\pythonw.exe" "\<NETWORK_LOC>\psychopyLib\psychopy\app\--psychopyApp.py"

You probably spotted from this that the PsychoPy app is simply a Python script. You may want to update the file associations too, so that .psyexp and .py are opened with:

"C:\Program Files\PsychoPy2\pythonw.exe" "\<NETWORK_LOC>\psychopyLib\psychopy\app\--psychopyApp.py" "%1"

Lastly, to make the shortcut look pretty, you might want to update the icon too. Set the icon’s location to:

"\<NETWORK_LOC>\psychopyLib\psychopy\app\Resources\psychopy.ico"

12.8.5 5. Updating to a new version

Fetch the latest .zip release. Unpack it and replace the contents of \<NETWORK_LOC>\psychopyLib\ with the contents of the zip file.

12.9 Generating formatted strings

A formatted string is a variable which has been converted into a string (text). In python the specifics of how this is done is determined by what kind of variable you want to print.

Example 1: You have an experiment which generates a string variable called text. You want to insert this variable into a string so you can print it. This would be achieved with the following code:

```python
message = 'The result is %s' % (text)
```

This will produce a variable message which if used in a text object would print the phrase ‘The result is’ followed by the variable text. In this instance %s is used as the variable being entered is a string. This is a marker which tells the script where the variable should be entered. %text tells the script which variable should be entered there.

Multiple formatted strings (of potentially different types) can be entered into one string object:
Some of the handy formatted string types:

```python
>>> x=5
>>> x1=5124
>>> z='someText'
>>> 'show %s' % (z)
'show someText'
>>> '%.1f' % (x)  # will show as a float to one decimal place
'5.0'
>>> '%3i' % (x)  # an integer, at least 3 chars wide, padded with spaces
'  5'
>>> '%03i' % (x)  # as above but pad with zeros (good for participant numbers)
'005'
```

See the python documentation for a more complete list.

## 12.10  Coder - Interleave Staircases

Often psychophysicists using staircase procedures want to interleave multiple staircases, either with different start points, or for different conditions.

There is now a class, `psychopy.data.MultiStairHandler` to allow simple access to interleaved staircases of either basic or QUEST types. That can also be used from the `Loops` in the `Builder`. The following method allows the same to be created in your own code, for greater options.

The method works by nesting a pair of loops, one to loop through the number of trials and another to loop across the staircases. The staircases can be shuffled between trials, so that they do not simply alternate.

**Note:** Note the need to create a *copy* of the info. If you simply do `thisInfo=info` then all your staircases will end up pointing to the same object, and when you change the info in the final one, you will be changing it for all.

```python
from __future__ import print_function
from builtins import next
from builtins import range
from psychopy import visual, core, data, event
from numpy.random import shuffle
import copy, time  # from the std python libs

# create some info to store with the data
info={}
info['startPoints']=[1.5,3,6]
info['nTrials']=10
info['observer']='jwp'

win=visual.Window([400,400])
#---------------------
# create the stimuli
#---------------------

# create staircases
stairs=[]
(continues on next page)
for thisStart in info['startPoints']:
    # we need a COPY of the info for each staircase
    # (or the changes here will be made to all the other staircases)
    thisInfo = copy.copy(info)
    # now add any specific info for this staircase
    thisInfo['thisStart'] = thisStart  # we might want to keep track of this
    thisStair = data.StairHandler(startVal = thisStart,
        extraInfo = thisInfo,
        nTrials = 50, nUp = 1, nDown = 3,
        minVal = 0.5, maxVal = 8,
        stepSizes = [4, 4, 2, 2, 1, 1])
    stairs.append(thisStair)

for trialN in range(info['nTrials']):
    shuffle(stairs)  # this shuffles 'in place' (ie stairs itself is changed, nothing returned)
    # then loop through our randomised order of staircases for this repeat
    for thisStair in stairs:
        thisIntensity = next(thisStair)
        print('start=%.2f, current=%.4f ' % (thisStair.extraInfo['thisStart'],
    thisIntensity))

        #---------------------
        # run your trial and get an input
        #---------------------
        keys = event.waitKeys()  # (we can simulate by pushing left for 'correct')
        if 'left' in keys: wasCorrect = True
        else: wasCorrect = False

        thisStair.addData(wasCorrect)  # so that the staircase adjusts itself

        # this trial (of all staircases) has finished
        # all trials finished

    # save data (separate pickle and txt files for each staircase)
    dateStr = time.strftime('%b_%d_%H%M', time.localtime())  # add the current time
    for thisStair in stairs:
        # create a filename based on the subject and start value
        filename = '%s start%.2f %s' % (thisStair.extraInfo['observer'], thisStair.
        extraInfo['thisStart'], dateStr)
        thisStair.saveAsPickle(filename)
        thisStair.saveAsText(filename)

12.11 Making isoluminant stimuli

From the mailing list (see there for names, etc):

Q1: How can I create colours (RGB) that are isoluminant?
A1: The easiest way to create isoluminant stimuli (or control the luminance content) is to create the stimuli in DKL space and then convert them into RGB space for presentation on the monitor.

More details on DKL space can be found in the section about Color spaces and conversions between DKL and RGB can be found in the API reference for psychopy.misc

Q2: There’s a difference in luminance between my stimuli. How could I correct for that?
I'm running an experiment where I manipulate color chromatic saturation, keeping luminance constant. I've coded the colors (red and blue) in rgb255 for 6 saturation values (10%, 20%, 30%, 40%, 50%, 60%, 90%) using a conversion from HSL to RGB color space.

Note that we don't possess spectrophotometers such as PR650 in our lab to calibrate each color gun. I've calibrated the gamma of my monitor psychophysically. Gamma was set to 1.7 (threshold) for gamm(lum), gamm(R), gamm(G), gamm(B). Then I've measured the luminance of each stimuli with a Brontes colorimeter. But there's a difference in luminance between my stimuli. How could I correct for that?

A2: Without a spectroradiometer you won't be able to use the color spaces like DKL which are designed to help this sort of thing.

If you don't care about using a specific colour space though you should be able to deduce a series of isoluminant colors manually, because the luminance outputs from each gun should sum linearly. e.g. on my monitor:

| maxR=46 cd/m2 |
| maxG=114 |
| maxB=15 |

(note that green is nearly always brightest)

So I could make a 15cd/m2 stimulus using various appropriate fractions of those max values (requires that the screen is genuinely gamma-corrected):

| R=0, G=0, B=255 |
| R=255*15/46, G=0, B=0 |
| R=255*7.5/46, G=255*15/114, B=0 |

Note that, if you want a pure fully-saturated blue, then you're limited by the monitor to how bright you can make your stimulus. If you want brighter colours your blue will need to include some of the other guns (similarly for green if you want to go above the max luminance for that gun).

A2.1. You should also consider that even if you set appropriate RGB values to display your pairs of chromatic stimuli at the same luminance that they might still appear different, particularly between observers (and even if your light measurement device says the luminance is the same, and regardless of the colour space you want to work in). To make the pairs perceptually isoluminant, each observer should really determine their own isoluminant point. You can do this with the minimum motion technique or with heterochromatic flicker photometry.

### 12.12 Adding a web-cam

From the mailing list (see there for names, etc):

“I spent some time today trying to get a webcam feed into my psychopy proj, inside my visual.window. The solution involved using the opencv module, capturing the image, converting that to PIL, and then feeding the PIL into a SimpleImageStim and looping and win.flipping. Also, to avoid looking like an Avatar in my case, you will have to change the default decoder used in PIL fromstring to utilize BGR instead of RGB in the decoding. I thought I would save some time for people in the future who might be interested in using a webcam feed for their psychopy project. All you need to do is import the opencv module into psychopy (importing modules was well documented by psychopy online) and integrate something like this into your psychopy script.”

```python
from __future__ import print_function
from psychopy import visual, event, core
import Image, time, pylab, cv, numpy

mywin = visual.Window(allowGUI=False, monitor='testMonitor', units='norm', colorSpace='rgb', color=[-1,-1,-1], fullscr=True)
```

(continues on next page)
mywin.setMouseVisible(False)

capture = cv.CaptureFromCAM(0)
img = cv.QueryFrame(capture)
pi = Image.fromstring("RGB", cv.GetSize(img), img.tostring(), "raw", "BGR", 0, 1)
print(pi.size)
myStim = visual.GratingStim(win=mywin, tex=pi, pos=[0, 0.5], size=[0.6, 0.6], opacity=1.0, units='norm')
myStim.setAutoDraw(True)

while True:
    img = cv.QueryFrame(capture)
    pi = Image.fromstring("RGB", cv.GetSize(img), img.tostring(), "raw", "BGR", 0, 1)
    myStim.setTex(pi)
    mywin.flip()
    theKey = event.getKeys()
    if len(theKey) != 0:
        break
13.1 Why is the bits++ demo not working?

So far PsychoPy supports bits++ only in the bits++ mode (rather than mono++ or color++). In this mode, a code (the T-lock code) is written to the lookup table on the bits++ device by drawing a line at the top of the window. The most likely reason that the demo isn’t working for you is that this line is not being detected by the device, and so the lookup table is not being modified. Most of these problems are actually nothing to do with PsychoPy /per se/, but to do with your graphics card and the CRS bits++ box itself.

There are a number of reasons why the T-lock code is not being recognised:

- the bits++ device is in the wrong mode. Open the utility that CRS supply and make sure you're in the right mode. Try resetting the bits++ (turn it off and on).
- the T-lock code is not fully on the screen. If you create a window that's too big for the screen or badly positioned then the code will be broken/not visible to the device.
- the T-lock code is on an ‘odd’ pixel.
- the graphics card is doing some additional filtering (win32). Make sure you turn off any filtering in the advanced display properties for your graphics card.
- the gamma table of the graphics card is not set to be linear (but this should normally be handled by PsychoPy, so don’t worry so much about it).
- you’ve got a Mac that’s performing temporal dithering (new Macs, around 2009). Apple have come up with a new, very annoying idea, where they continuously vary the pixel values coming out of the graphics card every frame to create additional intermediate colours. This will break the T-lock code on 1/2-2/3rds of frames.

13.2 Can PsychoPy run my experiment with sub-millisecond timing?

This question is common enough and complex enough to have a section of the manual all of its own. See Timing Issues and synchronisation.
13.3 How do I run experiments written with older versions of PsychoPy?

If you perform experiments on computers shared by different research groups (e. g. at a shared experimental facility), it’s possible that their experiments and yours are written using different versions of PsychoPy. Or maybe you yourself have some older and some newer experiments. In such a situation, it’s important to ensure that the right version of PsychoPy is used for the right experiment.

In PsychoPy standalone, there is an easy-to-use system for controlling what version of PsychoPy is used.

13.3.1 Version control using Builder View

1. Open up the PsychoPy experiment file (a ‘.psyexp’ file) that contains the experiment you want to use.
2. Go to experiment settings by clicking the icon with a cogwheel.
3. Under the “Basic” settings tab, there is an option named “Use PsychoPy version”. Set it to the PsychoPy version you want to emulate.
4. Click “OK” to save the settings.
5. Run the experiment by clicking the green ‘run’ button.

13.3.2 Version control using Coder View

1. Open up the PsychoPy script file (a ‘.py’ file) that contains the experiment you want to use.
2. Add `import psychopy` at the top of your script, above all other import statements.
3. Add the function call `psychopy.useVersion('<version_no>')` (e. g. `psychopy.useVersion('1.90.2')`) directly below `import psychopy`, but still above the other import statements. Here’s an example:

```python
import psychopy
psychopy.useVersion('1.90.2')
from psychopy import visual, core, event
# the rest of your script follows
```
4. Run the script.

13.3.3 NOTE: Internet connection needed (the first time)

You need to have a working internet connection the first time that you run an experiment using a particular version (e. g. 1.90.2) on a computer, so that PsychoPy can download some info about the version for you. Once you’ve used a version once, your computer has saved the information it needs for emulating it. This means that after the first time, you don’t need an internet connection if you run the same or another experiment using that version (e. g. 1.90.2).
13.3.4 Compatibility

Using either of the above methods, you should often only need the latest version of PsychoPy standalone to run older experiments. However, if you have an experiment designed with a very old version of PsychoPy (say, version 1.77.01) you might have to install an older version of standalone PsychoPy. Since these things change over time, you probably want to search for help in the PsychoPy forums.
There are a number of further resources to help learn/teach about PsychoPy.
If you also have PsychoPy materials/course then please let us know so that we can link to them from here too!

14.1 Workshops

We are currently running virtual workshops in several formats, see our workshops pages for details

14.2 Youtube tutorials

• There is our YouTube PsychoPy playlist showing how to build basic experiments in the Builder interface.
• Jason Ozubko has added a series of great PsychoPy Builder video tutorials too
• Damien Mannion added a similarly great series of PsychoPy programming videos on YouTube
• … and a searching YouTube for PsychoPy reveals many more!
14.3 Materials for Builder

The most comprehensive guide is the book *Building Experiments in PsychoPy* by Peirce and MacAskill. The book is suitable for a wide range of needs and skill sets, with 3 sections for:

- The Beginner (suitable for undergraduate teaching)
- The Professional (more detail for creating more precise studies)
- The Specialist (with info about specialist needs such as studies in fMRI, EEG, …)

At School of Psychology, University of Nottingham, PsychoPy is now used for all first year practical class teaching. The classes that comprise that first year course are provided below. They were created partially with funding from the former Higher Education Academy Psychology Network. Note that the materials here will be updated frequently as they are further developed (e.g. to update screenshots etc) so make sure you have the latest version of them!

- PsychoPy_pracs_2011v2.zip (21MB) (last updated: 15 Dec 2011)

The GestaltReVision group (University of Leuven) wiki covering PsychoPy (some Builder info and great tutorials for Python/PsychoPy coding of experiments).

There’s a set of tools for teaching psychophysics using PsychoPy and a PsychoPhysics poster from VSS. Thanks James Ferwerda

14.4 Materials for Coder

- Please see the page on officialWorkshops for further details on coming to an intensive residential Python workshop in Nottingham.
- Marco Bertamini’s book, *Programming Illusions for Everyone* is a fun way to learn about stimulus rendering in PsychoPy by learning how to create visual illusions
- Gary Lupyan runs a class on programming experiments using Python/PsychoPy and makes his lecture materials available on this wiki
- The GestaltReVision group (University of Leuven) offers a three-day crash course to Python and PsychoPy on a IPython Notebook, and has lots of great information taking you from basic programming to advanced techniques.
- Radboud University, Nijmegen also has a PsychoPy programming course
- Programming for Psychology in Python - Vision Science has lessons and screencasts on PsychoPy (by Damien Mannion, UNSW Australia).

14.5 Previous events

- ECEM, August 2013: Python for eye-tracking workshop with (Sol Simpson, Michael MacAskill and Jon Peirce). Download Python-for-eye-tracking materials
- VSS

For developers:
The best place to discuss ideas in depth is probably the dedicated developers section of the forum.

For developers the best way to use PsychoPy is to install a version to your own copy of python (preferably 3.6 but we try to support a reasonable range). Make sure you have all the dependencies, including the extra suggested packages for developers.

Don't install PsychoPy. Instead fetch a copy of the git repository and add this to the python path using a .pth file. Other users of the computer might have their own standalone versions installed without your repository version touching them.

15.1 Using the repository

Any code that you want to be included into PsychoPy is done via Git in the GitHub repository. There's something of a learning curve to this, but it's common to development in many other packages.

15.1.1 Workflow

The use of git and the following workflow allows people to contribute changes that can easily be incorporated back into the project, while (hopefully) maintaining order and consistency in the code. All changes should be tracked and reversible.

1. Create a fork of the psychopy/psychopy repository, usually referred to as “upstream”

2. For very small changes (e.g. to a single file):
   - you might work directly on GitHub

3. Normally you would clone your fork to your local development machine:
   - Create a local clone of that fork, with “upstream” as an additional remote
   - Before doing some coding, fetch updates from _upstream_
   - Do your work in a _branch_
   - Push that branch back to your fork
   - Submit a pull request there
15.1.2 Create your own fork of the central repository

Go to github, create an account and make a fork of the psychopy repository. You can change your fork in any way you choose without it affecting the central project. You can also share your fork with others, including the central project.

15.1.3 Fetch a local copy

Install git on your computer. Create and upload an ssh key to your github account - this is necessary for you to push changes back to your fork of the project at github.

Then, in a folder of your choosing fetch your fork:

```
$ git clone git@github.com:USER/psychopy.git
$ cd psychopy
$ git remote add upstream git://github.com/psychopy/psychopy.git
```

The last line connects your copy (with read access) to the central server so you can easily fetch any updates to the central repository.

15.1.4 Fetching the latest version

Periodically it’s worth fetching any changes to the central psychopy repository (into your master branch, more on that below):

```
$ git checkout master
$ git pull upstream master  # here 'master' is the desired branch of psychopy to fetch
```

15.1.5 Run PsychoPy using your local copy

Now that you’ve fetched the latest version of psychopy using git, you should run this version in order to try out yours/others latest improvements. See this guide on how to permanently run your git version of psychopy instead of the version you previously installed.

*Run git version for just one session (Linux and Mac only):* If you want to switch between the latest-and-greatest development version from git and the stable version installed on your system, you can choose to only temporarily run the git version. Open a terminal and set a temporary python path to your psychopy git folder:

```
$ export PYTHONPATH=/path/to/local/git/folder/
```

To check that worked you should open python in the terminal and try to import psychopy:

```
$ python
Python 2.7.6 (default, Mar 22 2014, 22:59:56)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import psychopy
```

PsychoPy depends on a lot of other packages and you may get a variety of failures to import them until you have them all installed in your custom environment!
15.1.6 Fixing bugs and making minor improvements

You can make minor changes directly in the master branch of your fork. After making a change you need to commit a set of changes to your files with a message. This enables you to group together changes and you will subsequently be able to go back to any previous commit, so your changes are reversible.

I (Jon) usually do this by opening the graphical user interface that comes with git:

```
$ git gui
```

From the GUI you can select (or stage in git terminology) the files that you want to include in this particular commit and give it a message. Give a clear summary of the changes for the first line. You can add more details about the changes on lower lines if needed.

If you have internet access then you could also push your changes back up to your fork (which is called your origin by default), either by pressing the push button in the GUI or by closing that and typing:

```
$ git push
```

15.1.7 Commit messages

Informative commit messages are really useful when we have to go back through the repository finding the time that a particular change to the code occurred. Precede your message with one or more of the following to help us spot easily if this is a bug fix (which might need pulling into other development branches) or new feature (which we might want to avoid pulling in if it might disrupt existing code).

- **BF**: bug fix
- **FF**: ‘feature’ fix. This is for fixes to code that hasn’t been released
- **RF**: refactoring
- **NF**: new feature
- **ENH**: enhancement (improvement to existing code)
- **DOC**: for all kinds of documentation-related commits
- **TEST**: for adding or changing tests

When making commits that fall into several commit categories (e.g., BF and TEST), please make separate commits for each category and avoid concatenating commit message prefixes. E.g., please do not use BF/TEST, because this will affect how commit messages are sorted when we pull in fixes for each release.

NB: The difference between BF and FF is that BF indicates a fix that is appropriate for back-porting to earlier versions, whereas FF indicates a fix to code that has not been released, and so cannot be back-ported.

15.1.8 Share your improvement with others

Only a couple of people have direct write-access to the psychopy repository, but you can get your changes included in upstream by pushing your changes back to your github fork and then submitting a pull request. Communication is good, and hopefully you have already been in touch (via the user or dev lists) about your changes.

When adding an improvement or new feature, consider how it might impact others. Is it likely to be generally useful, or is it something that only you or your lab would need? (It’s fun to contribute, but consider: does it actually need to be part of PsychoPy?) Including more features has a downside in terms of complexity and bloat, so try to be sure that there is a “business case” for including it. If there is, try at all times to be backwards compatible, e.g., by adding a new
keyword argument to a method or function (not always possible). If it's not possible, it's crucial to get wider input about the possible impacts. Flag situations that would break existing user scripts in your commit messages.

Part of sharing your code means making things sensible to others, which includes good coding style and writing some documentation. You are the expert on your feature, and so are in the best position to elaborate nuances or gotchas. Use meaningful variable names, and include comments in the code to explain non-trivial things, especially the intention behind specific choices. Include or edit the appropriate doc-string, because these are automatically turned into API documentation (via sphinx). Include doc-tests if that would be meaningful. The existing codebase has a comment / code ratio of about 28%, which earns it high marks.

For larger changes and especially new features, you might need to create some usage examples, such as a new Coder demo, or even a Builder demo. These can be invaluable for being a starting point from which people can adapt things to the needs of their own situation. This is a good place to elaborate usage-related gotchas.

In terms of style, try to make your code blend in with and look like the existing code (e.g., using about the same level of comments, use camelCase for var names, despite the conflict with the usual PEP – we’ll eventually move to the underscore style, but for now keep everything consistent within the codebase). In your own code, write however you like of course. This is just about when contributing to the project.

### 15.1.9 Add a new feature branch

For most jobs on the repository, you should create a new branch in your repository. Often while working on a new feature other aspects of the code will get broken and the *master* branch should always be in a working state. To create a new branch:

```
$ git branch feature-somethingNew
```

You can now switch to your new feature branch with:

```
$ git checkout feature-somethingNew
```

And get back to your *master* branch with:

```
$ git checkout master
```

You can push your new branch back to your fork (*origin*) with:

```
$ git push origin feature-somethingNew
```

### 15.1.10 Completing work on a feature

It's also possible to run just selected tests, such as doctests within a single file. From a terminal window:

```
cd psychopy/tests/  # e.g. /Users/jgray/code/psychopy/psychopy/tests
pytest
```

If the tests pass you hopefully haven’t damaged other parts of PsychoPy (!?). If possible add a unit test for your new feature too, so that if other people make changes they don’t break your work!

You can merge your changes back into your master branch with:

```
$ git checkout master
$ git merge feature-somethingNew
```

Merge conflicts happen, and need to be resolved. If you configure your git preferences (~/gitconfig) to include:
[merge]
  summary = true
  log = true
  tool = opendiff

then you’ll be able to use a handy GUI interface (opendiff) for reviewing differences and conflicts, just by typing:

git mergetool

from the command line after hitting a merge conflict (such as during a `git pull upstream master`).

Once you’ve folded your new code back into your master and pushed it back to your github fork then it’s time to Share your improvement with others.

## 15.2 Adding documentation

There are several ways to add documentation, all of them useful: doc strings, comments in the code, and demos to show an example of actual usage. To further explain something to end-users, you can create or edit a .rst file that will automatically become formatted for the web, and eventually appear on www.psychopy.org.

You make a new file under psychopy/docs/source/, either as a new file or folder or within an existing one.

To test that your doc source code (.rst file) does what you expect in terms of formatting for display on the web, you can simply do something like (this is my actual path, unlikely to be yours):

```
$ cd /Users/jgray/code/psychopy/docs/
$ make html
```

Do this within your docs directory (requires sphinx to be installed, try “pip install sphinx” if it’s not working). That will add a build/html sub-directory.

Then you can view your new doc in a browser, e.g., for me:

```
file:///Users/jgray/code/psychopy/docs/build/html/
```

Push your changes to your github repository (using a “DOC:” commit message) and let Jon know, e.g. with a pull request.

## 15.3 Adding a new Builder Component

Builder Components are auto-detected and displayed to the experimenter as icons (in the right-most panel of the Builder interface panel). This makes it straightforward to add new ones.

All you need to do is create a list of parameters that the Component needs to know about (that will automatically appear in the Component’s dialog) and a few pieces of code specifying what code should be called at different points in the script (e.g. beginning of the Routine, every frame, end of the study etc…). Many of these will come simply from subclassing the _base or _visual Components.

To get started, Add a new feature branch for the development of this component. (If this doesn't mean anything to you then see Using the repository)

You’ll mainly be working in the directory …/psychopy/experiment/components/. Take a look at several existing Components (such as image.py), and key files including _base.py and _visual.py.

There are three main steps, the first being by far the most involved.
15.3.1 1. Create the file defining the component: newcomp.py

It’s most straightforward to model a new Component on one of the existing ones. Be prepared to specify what your Component needs to do at several different points in time: the first trial, every frame, at the end of each routine, and at the end of the experiment. In addition, you may need to sacrifice some complexity in order to keep things streamlined enough for a Builder (see e.g., ratingscale.py).

Your new Component class (in your file newcomp.py) should inherit from BaseComponent (in _base.py), VisualComponent (in _visual.py), or KeyboardComponent (in keyboard.py). You may need to rewrite some or all of these methods, to override default behavior:

```python
class NewcompComponent(BaseComponent): # or (VisualComponent)  
    def __init__(...):
        super(NewcompComponent, self).__init__(...)
        
        def writeInitCode(self, buff):
        def writeRoutineStartCode(self, buff):
        def writeFrameCode(self, buff):
        def writeRoutineEndCode(self, buff):
```

Calling super() will create the basic default set of params that almost every component will need: name, startVal, start- Type, etc. Some of these fields may need to be overridden (e.g., durationEstim in sound.py). Inheriting from Visual- Component (which in turn inherits from BaseComponent) adds default visual params, plus arranges for Builder scripts to import psychopy.visual. If your component will need other libs, call self.exp.requirePsychopyLib(['neededLib']) (see e.g., parallelPort.py).

At the top of a component file is a dict named _localized. It contains mappings that allow a strict separation of internal string values (= used in logic, never displayed) from values used for display in the Builder interface (= for display only, possibly translated, never used in logic). The .hint and .label fields of params['someParam'] should always be set to a localized value, either by using a dict entry such as _localized['message'], or via the globally available translation function, _('message'). Localized values must not be used elsewhere in a component definition.

Very occasionally, you may also need to edit settings.py, which writes out the set-up code for the whole experiment (e.g., to define the window). For example, this was necessary for the ApertureComponent, to pass allowStencil=True to the window creation.

Your new Component writes code into a buffer that becomes an executable python file, xxx_lastrun.py (where xxx is whatever the experimenter specifies when saving from the Builder, xxx.psyexp). You will do a bunch of this kind of call in your newcomp.py file:

```python
buff.writeIndented(your_python_syntax_string_here)
```

You have to manage the indentation level of the output code, see experiment.IndentingBuffer().

xxx_lastrun.py is the file that gets built when you run xxx.psyexp from the Builder. So you will want to look at xxx_lastrun.py frequently when developing your component.

Name-space

There are several internal variables (i.e. names of Python objects) that have a specific, hardcoded meaning within xxx_lastrun.py. You can expect the following to be there, and they should only be used in the original way (or something will break for the end-user, likely in a mysterious way):

```python
win  # the window
T    # time within the trial loop, referenced to `trialClock`
x, y # mouse coordinates, but only if the experimenter uses a mouse component
```

Handling of variable names is under active development, so this list may well be out of date. (If so, you might consider updating it or posting a note to the PsychoPy Discourse developer forum.)
PsychoPy - Psychology software for Python, Release 2020.2.10

Preliminary testing suggests that there are 600-ish names from numpy or numpy.random, plus the following:

```python
['KeyResponse', '__builtins__', '__doc__', '__file__', '__name__', '__package__',
 '__buttons__', 'core', 'data', 'dig', 'event', 'expInfo', 'expName', 'filename', 'gui',
 '__logFile__', 'os', 'psychopy', 'sound', 't', 'visual', 'win', 'x', 'y']
```

Yet other names get derived from user-entered names, like trials → thisTrial.

**Params**

`self.params` is a key construct that you build up in `__init__`. You need name, startTime, duration, and several other params to be defined or you get errors. `name` should be of type 'code'.

The `Param()` class is defined in `psychopy.app.builder.experiment.Param()`. A very useful thing that Params know is how to create a string suitable for writing into the .py script. In particular, the `__str__` representation of a Param will format its value (.val) based on its type (.valType) appropriately. This means that you don’t need to check or handle whether the user entered a plain string, a string with a code trigger character ($), or the field was of type 'code' in the first place. If you simply request the `str()` representation of the param, it is formatted correctly.

To indicate that a param (eg, `thisParam`) should be considered as an advanced feature, set its category to advanced: `self.params['thisParam'].categ = 'Advanced'`. Then the GUI shown to the experimenter will automatically place it on the 'Advanced' tab. Other categories work similarly (Custom, etc).

During development, it can sometimes be helpful to save the params into the `xxx_lastrun.py` file as comments, so you can see what is happening:

```python
def writeInitCode(self, buff):
    # for debugging during Component development:
    buff.writelnIndented("# self.params for aperture:
    for p in self.params:
        try: buff.writelnIndented("# $s: $s <type %s>
        except: pass
```

A lot more detail can be inferred from existing components.

Making things loop-compatible looks interesting – see `keyboard.py` for an example, especially code for saving data at the end.

### 15.3.2 Notes & gotchas

**syntax errors in new_comp.py:** The PsychoPy app will fail to start if there are syntax error in any of the components that are auto-detected. Just correct them and start the app again.

**param[].val:** If you have a boolean variable (e.g., `my_flag`) as one of your params, note that `self.param['my_flag']` is always True (the param exists --> True). So in a boolean context you almost always want the .val part, e.g., `if self.param['my_flag'].val`.

However, you do not always want .val. Specifically, in a string/unicode context (= to trigger the self-formatting features of Param()), you almost always want “%s” % self.param['my_flag'], without .val. Note that it’s better to do this via “%s” than str() because str(self.param['my_flag']) coerces things to type str (squashing unicode) whereas %s works for both str and unicode.

**Travis testing** Before submitting a pull request with the new component, you should regenerate the `componentsTemplate.txt` file. This is a text file that lists the attributes of all of the user interface settings and options in the various components. It is used during the Travis automated testing process when a pull request is submitted to GitHub, allowing the detection of errors that may have been caused in refactoring. Your new component needs to have entries added to this file if the Travis testing is going to pass successfully.

15.3. Adding a new Builder Component 607
To re-generate the file, cd to this directory …/psychopy/tests/test_app/test_builder/ and run:

```
python genComponsTemplate.py --out
```

This will over-write the existing file so you might want to make a copy in case the process fails. **Compatibility issues:** As at May 2018, that script is not yet Python 3 compatible, and on a Mac you might need to use `pythonw`.

### 15.3.3 2. Icon: newcomp.png

Using your favorite image software, make an icon for your Component with a descriptive name, e.g., `newcomp.png`. Dimensions = 48 × 48. Put it in the components directory.

In `newcomp.py`, have a line near the top:

```
iconFile = path.join(thisFolder, 'newcomp.png')
```

### 15.3.4 3. Documentation: newcomp.rst

Just make a descriptively-named text file that ends in `.rst` (“restructured text”), and put it in `psychopy/docs/source/builder/components/`. It will get auto-formatted and end up at http://www.psychopy.org/builder/components/newcomp.html

### 15.4 Style-guide for coder demos

Each coder demo is intended to illustrate a key PsychoPy feature (or two), especially in ways that show usage in practice, and go beyond the description in the API. The aim is not to illustrate every aspect, but to get people up to speed quickly, so they understand how basic usage works, and could then play around with advanced features.

As a newcomer to PsychoPy, you are in a great position to judge whether the comments and documentation are clear enough or not. If something is not clear, you may need to ask a PsychoPy contributor for a description; email psychopy-dev@googlegroups.com.

Here are some style guidelines, written for the OpenHatch event(s) but hopefully useful after that too. These are intended specifically for the coder demos, not for the internal code-base (although they are generally quite close).

The idea is to have clean code that looks and works the same way across demos, while leaving the functioning mostly untouched. Some small changes to function might be needed (e.g., to enable the use of ‘escape’ to quit), but typically only minor changes like this.

- Generally, when you run the demo, does it look good and help you understand the feature? Where might there be room for improvement? You can either leave notes in the code in a comment, or include them in a commit message.

  ```
  #!/usr/bin/env python
  # -*- coding: utf-8 -*-
  
  """Demo name, purpose, description (1-2 sentences, although some demos need more... → explanation)."
  """
  ```

- Standardize the top stuff to have 1) a shebang with python, 2) utf-8 encoding, and 3) a comment:
For the comment / description, it’s a good idea to read and be informed by the relevant parts of the API (see http://psychopy.org/api/api.html), but there’s no need to duplicate that text in your comment. If you are unsure, please post to the dev list psychopy-dev@googlegroups.com.

- Follow PEP-8 mostly, some exceptions:
  - current PsychoPy convention is to use camelCase for variable names, so don’t convert those to underscores
  - 80 char columns can spill over a little. Try to keep things within 80 chars most of the time.
  - do allow multiple imports on one line if they are thematically related (e.g., import os, sys, glob).
  - inline comments are ok (because the code demos are intended to illustrate and explain usage in some detail, more so than typical code).
- Check all imports:
  - remove any unnecessary ones
  - replace import time with from psychopy import core. Use core.getTime() (= ms since the script started) or core.getAbsTime() (= seconds, unix-style) instead of time.time(), for all time-related functions or methods not just time().
  - add from __future__ import division, even if not needed. And make sure that doing so does not break the demo!
- Fix any typos in comments; convert any lingering British spellings to US, e.g., change colour to color
- Prefer if <boolean>: as a construct instead of if <boolean> == True: (There might not be any to change).
- If you have to choose, opt for more verbose but easier-to-understand code instead of clever or terse formulations. This is for readability, especially for people new to python. If you are unsure, please add a note to your commit message, or post a question to the dev list psychopy-dev@googlegroups.com.
- Standardize variable names:
  - use win for the visual.Window(), and so win.flip()
- Provide a consistent way for a user to exit a demo using the keyboard, ideally enable this on every visual frame: use if len(event.getKeys([‘escape’]): core.quit(). Note: if there is a previous event.getKeys() call, it can slurp up the ‘escape’ keys. So check for ‘escape’ first.
- Time-out after 10 seconds, if there’s no user response and a timeout is appropriate for the demo (and a longer time-out might be needed, e.g., for ratingScale.py):

```python
demoClock = core.clock()  # is demoClock’s time is 0.000s at this point
...
if demoClock.getTime() > 10.:
    core.quit()
```

- Most demos are not full screen. For any that are full-screen, see if it can work without being full screen. If it has to be full-screen, add some text to say that pressing ‘escape’ will quit.
- If displaying log messages to the console seems to help understand the demo, here’s how to do it:

```python
from psychopy import logging
...
logging.console.setLevel(logging.INFO)  # or logging.DEBUG for even more stuff
```

- End a script with win.close() (assuming the script used a visual.Window), and then core.quit() even though it’s not strictly necessary
15.5 Localization (I18N, translation)

PsychoPy is used worldwide. Starting with v1.81, many parts of PsychoPy itself (the app) can be translated into any language that has a unicode character set. A translation affects what the experimenter sees while creating and running experiments; it is completely separate from what is shown to the subject. Translations of the online documentation will need a completely different approach.

In the app, translation is handled by a function, \_translate(), which takes a string argument. (The standard name is \_(), but unfortunately this conflicts with \_ as used in some external packages that PsychoPy depends on.) The _translate() function returns a translated, unicode version of the string in the locale / language that was selected when starting the app. If no translation is available for that locale, the original string is returned (= English).

A locale setting (e.g., ‘ja_JP’ for Japanese) allows the end-user (= the experimenter) to control the language that will be used for display within the app itself. (It can potentially control other display conventions as well, not just the language.) PsychoPy will obtain the locale from the user preference (if set), or the OS.

Workflow: 1) Make a translation from English (en_US) to another language. You’ll need a strong understanding of PsychoPy, English, and the other language. 2) In some cases it will be necessary to adjust PsychoPy's code, but only if new code has been added to the app and that code displays text. Then re-do step 1 to translate the newly added strings.

See notes in psychopy/app/localization/readme.txt.

15.5.1 Make a translation (.po file)

As a translator, you will likely introduce many new people to PsychoPy, and your translations will greatly influence their experience. Try to be completely accurate; it is better to leave something in English if you are unsure how PsychoPy is supposed to work.

To translate a given language, you’ll need to know the standard 5-character code (see psychopy/app/localization/mappings). E.g., for Japanese, wherever LANG appears in the documentation here, you should use the actual code, i.e., “ja_JP” (without quotes).

A free app called poedit is useful for managing a translation. For a given language, the translation mappings (from en_US to LANG) are stored in a .po file (a text file with extension .po); after editing with poedit, these are converted into binary format (with extension .mo) which are used when the app is running.

- Start translation (do these steps once):
  - Start a translation by opening psychopy/app/locale/LANG/LC_MESSAGE/messages.po in Poedit. If there is no such .po file, create a new one:
    - make a new directory psychopy/app/locale/LANG/LC_MESSAGE/ if needed. Your LANG will be auto-detected within PsychoPy only if you follow this convention. You can copy metadata (such as the project name) from another .po file.
  - Set your name and e-mail address from “Preferences…” of “File” menu. Set translation properties (such as project name, language and charset) from Catalog Properties Dialog, which can be opened from “Properties…” of “Catalog” menu.
  - In poedit’s properties dialog, set the “source keywords” to include ‘_translate’. This allows poedit to find the strings in PsychoPy that are to be translated.
  - To add paths where Poedit scans .py files, open “Sources paths” tab on the Catalog Properties Dialog, and set “Base path:” to “../../../” (= psychopy/psychopy/). Nothing more should be needed. If you’ve created new catalog, save your catalog to psychopy/app/locale/LANG/LC_MESSAGE/messages.po.
  - Probably not needed, but check anyway: Edit the file containing language code and name mappings, psychopy/app/localization/mappings, and fill in the name for your language. Give a name that should be familiar to
people who read that language (i.e., use the name of the language as written in the language itself, not in en_US). About 25 are already done.

• Edit a translation:
  Open the .po file with Poedit and press “Update” button on the toolbar to update newly added / removed strings that need to be translated. Select a string you want to translate and input your translation to “Translation:” box. If you are unsure where string is used, point on the string in “Source text” box and right-click. You can see where the string is defined.

• Technical terms should not be translated: Builder, Coder, PsychoPy, Flow, Routine, and so on. (See the Japanese translation for guidance.)

• If there are formatting arguments in the original string (%s, % (first) i), the same number of arguments must also appear in the translation (but their order is not constrained to be the original order). If they are named (e.g., % (first) i), that part should not be translated—here first is a python name.

• If you think your translation might have room for improvement, indicate that it is “fuzzy”. (Saving Notes does not work for me on Mac, seems like a bug in poedit.)

• After making a new translation, saving it in poedit will save the .po file and also make an associated .mo file. You need to update the .mo file if you want to see your changes reflected in PsychoPy.

• The start-up tips are stored in separate files, and are not translated by poedit. Instead:
  • copy the default version (named psychopy/app/Resources/tips.txt) to a new file in the same directory, named tips_LANG.txt. Then replace English-language tips with translated tips. Note that some of the humor might not translate well, so feel free to leave out things that would be too odd, or include occasional mild humor that would be more appropriate. Humor must be respectful and suitable for using in a classroom, laboratory, or other professional situation. Don’t get too creative here. If you have any doubt, best leave it out. (Hopefully it goes without saying that you should avoid any religious, political, disrespectful, or sexist material.)

  • in poedit, translate the file name: translate “tips.txt” as “tips_LANG.txt”
  • Commit both the .po and .mo files to github (not just one or the other), and any changed files (e.g., tips_LANG, localization/mappings).

### 15.5.2 Adjust PsychoPy’s code

This is mostly complete (as of 1.81.00), but will be needed for new code that displays text to users of the app (experimenters, not study participants).

There are a few things to keep in mind when working on the app’s code to make it compatible with translations. If you are only making a translation, you can skip this section.

• In PsychoPy’s code, the language to be used should always be English with American spellings (e.g., “color”).

• Within the app, the return value from _translate() should be used only for display purposes: in menus, tooltips, etc. A translated value should never be used as part of the logic or internal functioning of PsychoPy. It is purely a “skin”. Internally, everything must be in en_US.

• Basic usage is exactly what you expect: _translate("hello") will return a unicode string at run-time, using mappings for the current locale as provided by a translator in a .mo file. (Not all translations are available yet, see above to start a new one.) To have the app display a translated string to the experimenter, just display the return value from the underscore translation function.

• The strings to be translated must appear somewhere in the app code base as explicit strings within _translate(). If you need to translate a variable, e.g., named str_var using the expression _translate(str_var), somewhere else you need to explicitly give all the possible values that str_var can take, and enclose each of them within the translate function. It is okay for that to be elsewhere, even in another file, but
not in a comment. This allows poedit to discover of all the strings that need to be translated. (This is one of the purposes of the _localized dict at the top of some modules.)

• `_translate()` should not be given a null string to translate; if you use a variable, check that it is not '' to avoid invoking `_translate('')`.

• Strings that contain formatting placeholders (e.g., %d, %s, %.4f) require a little more thought. Single placeholders are easy enough: `_translate("hello, %s") % name.

• Strings with multiple formatting placeholders require named arguments, because positional arguments are not always sufficient to disambiguate things depending on the phrase and the language to be translated into: `_translate("hello, %(first)s %(last)s") % {'first': firstname, 'last': lastname}

• Localizing drop-down menus is a little more involved. Such menus should display localized strings, but return selected values as integers (GetSelection() returns the position within the list). Do not use GetStringSelection(), because this will return the localized string, breaking the rule about a strict separation of display and logic. See Builder ParamDialogs for examples.

15.5.3 Other notes

When there are more translations (and if they make the app download large) we might want to manage things differently (e.g., have translations as a separate download from the app).

15.6 Adding a new Menu Item

Adding a new menu-item to the Builder (or Coder) is relatively straightforward, but there are several files that need to be changed in specific ways.

15.6.1 1. makeMenus()

The code that constructs the menus for the Builder is within a method named `makeMenus()`, within class `BuilderFrame()`. Decide which submenu your new command fits under, and look for that section (e.g., File, Edit, View, and so on). For example, to add an item for making the Routine panel items larger, I added two lines within the `self.viewMenu.Append()` method, and replacing:

```python
self.viewMenu.Append(self.IDs.tbIncrRoutineSize, _("&Routine Larger\t%s") %self.app._keys['largerRoutine'], _("Larger routine items"))
wx.EVT_MENU(self, self.IDs.tbIncrRoutineSize, self.routinePanel.increaseSize)
```

Note the use of the translation function, `_( )`, for translating text that will be displayed to users (menu listing, hint).
15.6.2 2. wxIDs.py

A new item needs to have a (numeric) ID so that wx can keep track of it. Here, the number is `self.IDs.tbIncrRoutineSize`, which I had to define within the file `psychopy/app/wxIDs.py`:

```
tbIncrRoutineSize=180
```

It's possible that, instead of hard-coding it like this, it's better to make a call to `wx.NewIdRef()` – wx will take care of avoiding duplicate IDs, presumably.

15.6.3 3. Key-binding prefs

I also defined a key to use to as a keyboard short-cut for activating the new menu item:

```
self.app.keys['largerRoutine']
```

The actual key is defined in a preference file. Because psychopy is multi-platform, you need to add info to four different .spec files, all of them being within the `psychopy/preferences/` directory, for four operating systems (Darwin, FreeBSD, Linux, Windows). For `Darwin.spec` (meaning macOS), I added two lines. The first line is not merely a comment: it is also automatically used as a tooltip (in the preferences dialog, under key-bindings), and the second being the actual short-cut key to use:

```
# increase display size of Routines
largerRoutine = string(default='Ctrl++') # on Mac Book Pro this is good
```

This means that the user has to hold down the _Ctrl_ key and then press the _+ _ key. Note that on Macs, ‘Ctrl’ in the spec is automatically converted into ‘Cmd’ for the actual key to use; in the .spec, you should always specify things in terms of ‘Ctrl’ (and not ‘Cmd’). The default value is the key-binding to use unless the user defines another one in her or his preferences (which then overrides the default). Try to pick a sensible key for each operating system, and update all four .spec files.

15.6.4 4. Your new method

The second line within `makeMenus()` adds the key-binding definition into wx’s internal space, so that when the key is pressed, wx knows what to do. In the example, it will call the method `self.routinePanel.increaseSize`, which I had to define to do the desired behavior when the method is called (in this case, increment an internal variable and redraw the routine panel at the new larger size).

15.6.5 5. Documentation

To let people know that your new feature exists, add a note about your new feature in the CHANGELOG.txt, and appropriate documentation in .rst files.
15.7 Creating Plugins for PsychoPy

Plugins provide a means for developers to extend PsychoPy, adding new features and customizations without directly modifying the PsychoPy installation. Read usingplugins for more information about plugins before proceeding on this page.

15.7.1 How plugins work

The plugin system in PsychoPy functions as a dynamic importer, which imports additional executable code from plugin packages then patches them into an active PsychoPy session at runtime. This is done by calling the `psychopy.plugins.loadPlugin()` function and passing the project name of the desired plugin to it. Once `loadPlugin()` returns, imported objects are immediately accessible. Any changes made to PsychoPy with plugins do not persist across sessions, meaning if Python is restarted, PsychoPy will return to its default behaviour unless `loadPlugin()` is called again.

Installed plugins for PsychoPy are discoverable on the system using package metadata. The metadata of the package defines “entry points” which tells the plugin loader where within PsychoPy’s namespace to place objects exported by the plugin. The loader also ensures plugins are compatible with the Python environment (ie. operating system, CPU architecture, and Python version). Any Python package can define entry points, allowing developers to add functionality to PsychoPy without needing to create a separate plugin project.

15.7.2 Plugin packages

A plugin has a similar structure to Python package, see the official Packaging Python Projects (https://packaging.python.org/tutorials/packaging-projects) guide for details.

Naming plugin packages

Standalone plugins, which are packages that exist only to extend PsychoPy should adhere to the following naming convention to make PsychoPy plugins discernible from any other package in public repositories. Plugin project names should always be prefixed with `psychopy` with individual words separated with a - or _ symbol (i.e. `psychopy-quest-procedure` or `psychopy_quest_procedure` are valid). What you chose to name the package is up to you, but keep it concise and informative.

Note: The plugin system does not use project names to identify plugins, rather relying on package metadata to identify if a package has entry points pertinent to PsychoPy. Therefore, projects do not need to be named a particular way to still be used as plugins. This allows packages which are not primarily used with PsychoPy to extend it, without the need for a separate plugin package. It also allows a single package to be used as a plugin for multiple projects unrelated to PsychoPy.

The module or sub-package which defines the objects which entry points refer to should be some variant of the name to prevent possible namespace conflicts. For instance, we would name our module `psychopy_quest_procedure` if our project was called `psychopy-quest-procedure`.
Specifying entry points

Entry points reference objects in a plugin module that PsychoPy will attach to itself. Packages advertise their entry points by having them in their metadata. How entry points are defined and added to package metadata is described in the section Dynamic Discovery of Services and Plugins of the documentation for `setuptools`.

When loading a specified plugin, the plugin loader searches for a distribution matching the given project name, then gets the entry point mapping from its metadata. Any entry point belonging to groups whose names start with `psychopy` is loaded. Group names are fully-qualified names of modules or unbound classes within PsychoPy’s namespace to create links to the associated entry points in the plugin module/package.

As an example, using entry point groups and specifiers, we can add a class called `MyStim` defined in the plugin module `psychopy_plugin` to appear in `psychopy.visual` when the plugin is loaded. To do this, we use the following dictionary when defining entry point metadata with the `setup()` function in the plugin project’s `setup.py` file:

```python
setup(
    ...,
    entry_points={
        'psychopy.visual': 'MyStim = psychopy_plugin:MyStim',
        ...,
    }
)
```

**Note:** Plugins can load and assign entry points to names anywhere in PsychoPy’s namespace. However, plugin developers should place them where they make most sense. In the last example, we put `MyStim` in `psychopy.visual` because that’s where users would expect to find it if it was part of the base PsychoPy installation.

If we have additional classes we’d like to add to `psychopy.visual`, entry entry points for that group can be given as a list of specifiers:

```python
setup(
    ...,
    entry_points={
        'psychopy.visual': ['MyStim = psychopy_plugin:MyStim',
                            'MyStim2 = psychopy_plugin:MyStim2'],
        ...,
    }
)
```

For more complex (albeit contrived) example to demonstrate how to modify unbound class attributes (ie. methods and properties), say we have a plugin which provides a custom interface to some display hardware called `psychopy-display` that needs to alter the existing `flip()` method of the `psychopy.visual.Window` class to work. Furthermore, we want to add a class to `psychopy.hardware` called `DisplayControl` to give the user a way of setting up and configuring the display. Entry points for both objects are defined in the plugin’s `psychopy_display` module. To get the effect we want, we specify entry points using the following:

```python
setup(
    ...,
    entry_points={
        'psychopy.version.Window': ['flip = psychopy_display:flip'],
        'psychopy.hardware': ['DisplayControl = psychopy_display:DisplayControl'],
        ...,
    }
)
```

After calling `loadPlugin('psychopy-display')`, the user will be able to create instances of `psychopy.hardware.DisplayControl` and new instances of `psychopy.visual.Window` will have the modified `flip()` method.
The __register__ attribute

Plugin modules can define a optional attribute named __register__ which specifies a callable object. The purpose of __register__ is to allow the module to perform tasks before loading entry points based on arguments passed to it by the plugin loader. The arguments passed to the target of __register__, come from the **kwargs given to loadPlugins(). The value of this attribute can be a string of the name or a reference to a callable object (ie. function or method).

Note: The __register__ attribute should only ever be used for running routines pertinent to setting up entry points. The referenced object is only called on a module once per session.

As an example, consider a case where an entry point is defined as doThis in plugin python-foobar. There are two possible behaviors which are foo and bar that doThis can have. We can implement both behaviors in separate functions, and use arguments passed to the __register__ target to assign which to use as the entry point:

```python
__register__ = 'register'
doThis = None
def foo():
    return 'foo'
def bar():
    return 'bar'
def register(**kwargs):
    global dothis
    option = kwargs.get('option', 'foo')
    if option == 'bar':
        dothis = bar
    else:
        dothis = foo
```

When the user calls loadPlugin('python-foobar', option='bar'), the plugin will assign function bar() to doThis. If option is not specified or given as 'foo', the behavior of doThis will be that of foo().

15.7.3 Plugin example project

This section will demonstrate how to create a plugin project and package it for distribution. For this example, we will create a plugin called psychopy-rect-area which adds a method to the psychopy.visual.Rect stimulus class called getArea() that returns the area of the shape when called.

Project files

First, we need to create a directory called psychopy-rect-area which all our Python packages and code will reside. Inside that directory, we create the following files and directories:
PsychoPy - Psychology software for Python, Release 2020.2.10

The implementation for the getArea() method will be deﬁned in a ﬁle called psychopy_rect_area/__init__.
py, it should contain the following:
#!/usr/bin/env python
# -*- coding: utf-8 -*"""Plugin entry points for `psychopy-rect-area`."""
def get_area(self):
"""Compute the area of a `Rect` stimulus in `units`.
Returns
------float
Area in units^2.
"""
return self.size[0] * self.size[1]

Note: The get_area() function needs to have self as the ﬁrst argument because were are going to assign it as class method.
All class methods get a reference to the class as the ﬁrst argument. You can name this whatever you like (eg. cls).
The setup.py script is used to generate an installable plugin package. This should contain something like the following:
#!/usr/bin/env python
# -*- coding: utf-8 -*from setuptools import setup
setup(name='psychopy-rect-area',
version='1.0',
description='Compute the area of a Rect stimulus.',
long_description='',
url='http://repo.example.com',
author='Nobody',
author_email='nobody@example.com',
license='GPL3',
classifiers=[
'Development Status :: 4 - Beta',
'License :: OSI Approved :: GLP3 License',
'Programming Language :: Python :: 2.7',
'Programming Language :: Python :: 3'
],
keywords='psychopy stimulus',
packages=['psychopy_rect_area'],
install_requires=['psychopy'],
include_package_data=True,
entry_points={
'psychopy.visual.Rect': ['getArea = psychopy_rect_area:get_area']
},
zip_safe=False)

Looking at entry_points we can see that were assigning psychopy_rect_area.get_area to psychopy.
visual.Rect.getArea. Attributes assigned to entry points should follow the naming conventions of PsychoPy
(camel case), however plugins are free to use internally whatever style the author chooses (eg. PEP8). You should also
use appropriate classiﬁers for your plugin, a full list can be found here (https://pypi.org/pypi?%3Aaction=list_classifiers).
You can also specify install_requires to indicate which versions of PsychPy are compatible with your plugin.
Visit https://packaging.python.org/discussions/install-requires-vs-requirements/ for more information.
15.7. Creating Plugins for PsychoPy

617


One should also include a README.md file which provides detailed information about the plugin. This file can be read and passed to the long_description argument of setup() in setup.py if desired by inserting the following into the setup script:

```python
from setuptools import setup
def get_readme_text():
    with open('README.md') as f:
        return f.read()

setup(
    ...,
    long_description=get_readme_text(),
    ...
)
```

Finally, we need specify README.md in our MANIFEST.in file to tell the packaging system to include the file when packaging. Simply put the following line in MANIFEST.in:

```
README.md
```

## Building packages

PsychoPy plugin packages are built like any other Python package. We can build a wheel distribution by calling the following console command:

```
python setup.py sdist bdist_wheel
```

The resulting .whl files will appear in directory psychopy-rect-area/dist. The generated packages can be installed with pip or uploaded to the Python Package Index. For more information about building and uploading packages, visit: https://packaging.python.org/tutorials/packaging-projects/

If uploaded to PyPI, other PsychoPy users can install your plugin by entering the following into their command prompt:

```
python -m pip install psychopy-rect-area
```

## Using the plugin

Once installed the plugin can be activated by using the psychopy.plugins.loadPlugin() function. This function should be called after the import statements in your script:

```
from psychopy import visual, core, plugins
plugins.loadPlugin('psychopy-demo-plugin')  # load the plugin
```

After calling loadPlugin(), all instances of Rect will have the method getArea():

```
rectStim = visual.Rect(win)
rectArea = rectStim.getArea()
```
15.7.4 Plugins as patches

A special use case of plugins is to apply and distribute “patches”. Using entry points to override module and class attributes, one can create patches to fix minor bugs in extant PsychoPy installations between releases, or backport fixes and features to older releases (that support plugins) that cannot be upgraded for some reason. Patches can be distributed like any other Python package, and can be installed and applied uniformly across multiple PsychoPy installations.

Plugins can also patch other plugins that have been previously loaded by loadPlugin() calls. This is done by defining entry points to module and class attributes that have been created by a previously loaded plugin.

Creating patches

As an example, consider a fictional scenario where a bug was introduced in a recent release of PsychoPy by a hardware vendor updating their drivers. As a result, PsychoPy’s built-in support for their devices provided by the psychopy.hardware.Widget class is now broken. You notice that it has been fixed in a pending release of PsychoPy, and that it involves a single change to the getData() method of the psychopy.hardware.Widget class to get it working exactly as before. However, you cannot wait for the next release because you are in the middle of running scheduled experiments, even worse, you have dozens of test stations using the hardware.

In this case, you can create a plugin to not only fix the bug, but apply it across multiple existing installations to save the day. Creating a package for our patch is no different than a regular plugin (see the Plugin example project section for more information), so you go about creating a project for a plugin called psychopy-hotfix which defines the working version of the getData() method in a sub-module called psychopy_hotfix like this:

```
# method copy and pasted from the bug fix commit
def getData(self):
    """This function reads data from the device."""
    # code here ...
```

In the setup.py file of the plugin package, specify the entry points like this to override the defective method in our installations:

```
setup(
    name='psychopy-hotfix',
    ...
    entry_points={
        'psychopy.hardware.Widget': ['getData = psychopy_patch:getData'],
    },
    ...
)
```

That’s it, just build a distributable package and install it on all the systems affected by the bug.

Applying patches

Whether you create your own patch, or obtain one provided by the PsychoPy community, they are applied using the loadPlugin() function after installing them. Experiment scripts will need to have the following lines added under the import statements at the top of the file for the plugin to take effect (but it’s considerably less work than manually patching in the code across many separate installations):

```
import psychopy.plugin as plugin
plugin.loadPlugin('psychopy-patch')
```

After loadPlugin is called, the behaviour of the getData() method of any instances of the psychopy.hardware.Widget class will change to the correct one.
Once a new release of PsychoPy comes out with the patch incorporated into it and your installations are upgraded, you can remove the above lines.

### 15.7.5 Creating window backends

Custom backends for the `Window` class can be implemented in plugins, allowing one to create windows using frameworks other than Pyglet, GLFW, and PyGame that can be enabled using the appropriate `winType` argument.

A plugin can add a `winType` by specifying class and module entry points for `psychopy.visual.backends`. If the entry point is a subclass of `psychopy.visual.backends.BaseBackend` and has `winTypeName` defined, it will be automatically registered and can be used as a `winType` by instances of `psychopy.visual.Window`.

**Note:** If a module is given as an entry point, the whole module will be added to `backends` and any class within it that is a subclass of `BaseBackend` and defines `winTypeName` will be registered. This allows one to add multiple window backends to PsychoPy with a single plugin module.

### Example

For example, say we have a backend class called `CustomBackend` defined in module `custom_backend` in the plugin package `psychopy-custom-backend`. We can tell the plugin loader to register it to be used when a `Window` instance is created with `winType='custom'` by adding the `winTypeName` class attribute to `CustomBackend`:

```python
class CustomBackend(BaseBackend):
    winTypeName = 'custom'
    ...
```

**Note:** If `winTypeName` is not defined, the entry points will still get added to `backends` but users will not be able to use it directly by specifying `winType`.

We define the entry point for our custom backend in `setup.py` as:

```python
setup(
    ...
    entry_points={
        'psychopy.visual.backends': 'custom_backend = custom_backend',
        ...
    }
)
```

Optionally, we can point to the backend class directly:

```python
setup(
    ...
    entry_points={
        'psychopy.visual.backends':
            'custom_backend = custom_backend:CustomBackend',
        ...
    }
)
```

After the plugin is installed and loaded, we can use our backend for creating windows by specifying `winType` as `win-TypeName`:
loadPlugin('psychopy-custom-backend')
win = Window(winType='custom')

Happy Coding Folks!!
PSYCHOPY EXPERIMENT FILE FORMAT (.PSYEXP)

The file format used to save experiments constructed in PsychoPy builder was created especially for the purpose, but is an open format, using a basic xml form, that may be of use to other similar software. Indeed the builder itself could be used to generate experiments on different backends (such as Vision Egg, PsychToolbox or PyEPL). The xml format of the file makes it extremely platform independent, as well as moderately easy to read by humans. There was a further suggestion to generate an XSD (or similar) schema against which psyexp files could be validated. That is a low priority but welcome addition if you wanted to work on it(!) There is a basic XSD (XML Schema Definition) available in psychopy/app/builder/experiment.xsd.

The simplest way to understand the file format is probably simply to create an experiment, save it and open the file in an xml-aware editor/viewer (e.g. change the file extension from .psyexp to .xml and then open it in Firefox). An example (from the stroop demo) is shown below.

The file format maps fairly obviously onto the structure of experiments constructed with the Builder interface, as described here. There are general Settings for the experiment, then there is a list of Routines and a Flow that describes how these are combined.

As with any xml file the format contains object nodes which can have direct properties and also child nodes. For instance the outermost node of the .psyexp file is the experiment node, with properties that specify the version of PsychoPy that was used to save the file most recently and the encoding of text within the file (ascii, unicode etc.), and with child nodes Settings, Routines and Flow.

16.1 Parameters

Many of the nodes described within this xml description of the experiment contain Param entries, representing different parameters of that Component. Nearly all parameter nodes have a name property and a val property. The parameter node with the name “advancedParams” does not have them. Most also have a valType property, which can take values ‘bool’, ‘code’, ‘extendedCode’, ‘num’, ‘str’ and an updates property that specifies whether this parameter is changing during the experiment and, if so, whether it changes ‘every frame’ (of the monitor) or ‘every repeat’ (of the Routine).

16.2 Settings

The Settings node contains a number of parameters that, in PsychoPy, would normally be set in the Experiment settings dialog, such as the monitor to be used. This node contains a number of Parameters that map onto the entries in that dialog.
16.3 Routines

This node provides a sequence of xml child nodes, each of which describes a Routine. Each Routine contains a number of children, each specifying a Component, such as a stimulus or response collecting device. In the Builder view, the Routines obviously show up as different tabs in the main window and the Components show up as tracks within that tab.

16.4 Components

Each Component is represented in the .psyexp file as a set of parameters, corresponding to the entries in the appropriate component dialog box, that completely describe how and when the stimulus should be presented or how and when the input device should be read from. Different Components have slightly different nodes in the xml representation which give rise to different sets of parameters. For instance the TextComponent nodes has parameters such as colour and font, whereas the KeyboardComponent node has parameters such as forceEndTrial and correctIf.

16.5 Flow

The Flow node is rather more simple. Its children simply specify objects that occur in a particular order in time. A Routine described in this flow must exist in the list of Routines, since this is where it is fully described. One Routine can occur once, more than once or not at all in the Flow. The other children that can occur in a Flow are LoopInitiators and LoopTerminators which specify the start and endpoints of a loop. All loops must have exactly one initiator and one terminator.

16.6 Names

For the experiment to generate valid PsychoPy code the name parameters of all objects (Components, Loops and Routines) must be unique and contain no spaces. That is, an experiment can not have two different Routines called ‘trial’, nor even a Routine called ‘trial’ and a Loop called ‘trial’.

The Parameter names belonging to each Component (or the Settings node) must be unique within that Component, but can be identical to parameters of other Components or can match the Component name themselves. A TextComponent should not, for example, have multiple ‘pos’ parameters, but other Components generally will, and a Routine called ‘pos’ would also be also permissible.

```xml
<PsychoPy2experiment version="1.50.04" encoding="utf-8">
  <Settings>
    <Param name="Monitor" val="testMonitor" valType="str" updates="None"/>
    <Param name="Window size (pixels)" val="[1024, 768]" valType="code" updates="None"/>
    <Param name="Full-screen window" val="True" valType="bool" updates="None"/>
    <Param name="Save log file" val="True" valType="bool" updates="None"/>
    <Param name="Experiment info" val="{'participant':"s_001", 'session':001}" valType="code" updates="None"/>
    <Param name="Show info dlg" val="True" valType="bool" updates="None"/>
    <Param name="logging level" val="warning" valType="code" updates="None"/>
    <Param name="Units" val="norm" valType="str" updates="None"/>
    <Param name="Screen" val="1" valType="num" updates="None"/>
  </Settings>
  <Routines>
    <Routine name="trial">
      (continues on next page)
    </Routine>
  </Routines>
</PsychoPy2experiment>
```
<TextComponent name="word">
  <Param name="word" val="word" valType="code" updates="constant"/>
  <Param name="text" val="thisTrial.text" valType="code" updates="set every...
  repeat"/>
</TextComponent>

<Param name="colour" val="thisTrial.rgb" valType="code" updates="set every...
  repeat"/>

<Param name="ori" val="0" valType="code" updates="constant"/>
<Param name="pos" val="[0, 0]" valType="code" updates="constant"/>
<Param name="times" val="[0.5, 2.0]" valType="code" updates="constant"/>
<Param name="letterHeight" val="0.2" valType="code" updates="constant"/>
<Param name="colourSpace" val="rgb" valType="code" updates="constant"/>
<Param name="units" val="window units" valType="str" updates="None"/>
<Param name="font" val="Arial" valType="str" updates="constant"/>
</TextComponent>

<KeyboardComponent name="resp">
  <Param name="storeCorrect" val="True" valType="bool" updates="constant"/>
  <Param name="resp" val="resp" valType="code" updates="None"/>
  <Param name="forceEndTrial" val="True" valType="bool" updates="constant"/>
  <Param name="times" val="[0.5, 2.0]" valType="code" updates="constant"/>
  <Param name="allowedKeys" val="[1, 2, 3]" valType="code" updates="constant"/>
</KeyboardComponent>

<Param name="storeResponseTime" val="True" valType="bool" updates="constant"/>
<Param name="correctIf" val="resp.keys==str(thisTrial.corrAns)" valType="code...
  updates="constant"/>
<Param name="store" val="last key" valType="str" updates="constant"/>
</KeyboardComponent>
</Routine>

<Routine name="instruct">
  <TextComponent name="instrText">
    <Param name="val" val=""&quot;Please press;&quot;##10;1 for red ink,##10;2 for...
    green ink##10;3 for blue ink##10; (Esc will quit)##10;##10;Any key to continue&quot;" valType="code" updates="constant"/>
    <Param name="colour" val="[1, 1, 1]" valType="code" updates="constant"/>
    <Param name="ori" val="0" valType="code" updates="constant"/>
    <Param name="pos" val="[0, 0]" valType="code" updates="constant"/>
    <Param name="times" val="[0, 10000]" valType="code" updates="constant"/>
    <Param name="letterHeight" val="0.1" valType="code" updates="constant"/>
    <Param name="colourSpace" val="rgb" valType="code" updates="constant"/>
    <Param name="units" val="window units" valType="str" updates="None"/>
    <Param name="font" val="Arial" valType="str" updates="constant"/>
  </TextComponent>
</Routine>

<KeyboardComponent name="ready">
  <Param name="storeCorrect" val="False" valType="bool" updates="constant"/>
  <Param name="val" val=""&quot;ready&quot; valType="code" updates="None"/>
  <Param name="forceEndTrial" val="True" valType="bool" updates="constant"/>
  <Param name="times" val="[0, 10000]" valType="code" updates="constant"/>
  <Param name="allowedKeys" val="" valType="code" updates="constant"/>
  <Param name="storeResponseTime" val="False" valType="bool" updates="constant"/>
</KeyboardComponent>

<Param name="correctIf" val="resp.keys==str(thisTrial.corrAns)" valType="code...
  updates="constant"/>
<Param name="store" val="last key" valType="str" updates="constant"/>
</KeyboardComponent>
</Routine>

<Routine name="thanks">
  <TextComponent name="thanksText">
(continues from previous page)

16.6. Names 625

(continues on next page)
<Param name="name" val="thanksText" valType="code" updates="constant"/>
<Param name="text" val="&quot;Thanks!&quot;" valType="code" updates="constant"/>
<Param name="colour" val="[1, 1, 1]" valType="code" updates="constant"/>
<Param name="ori" val="0" valType="code" updates="constant"/>
<Param name="pos" val="[0, 0]" valType="code" updates="constant"/>
<Param name="times" val="[1.0, 2.0]" valType="code" updates="constant"/>
<Param name="letterHeight" val="0.2" valType="code" updates="constant"/>
<Param name="colourSpace" val="rgb" valType="code" updates="constant"/>
<Param name="units" val="window units" valType="str" updates="None"/>
<Param name="font" val="arial" valType="str" updates="constant"/>
</TextComponent>

</Routine>
</Routines>
<Flow>
</Flow>
</PsychoPy2experiment>
Python Module Index

p
psychopy.clock, 244
psychopy.core, 115
psychopy.data, 447
psychopy.hardware, 247
psychopy.hardware.brainproducts, 249
psychopy.hardware.crs, 254
psychopy.hardware.eqi, 289
psychopy.hardware.emulator, 289
psychopy.hardware.forp, 294
psychopy.hardware.iolab, 295
psychopy.hardware.joystick, 296
psychopy.hardware.keyboard, 247
psychopy.hardware.minolta, 300
psychopy.hardware.pr, 301
psychopy.hardware.qmix, 307
psychopy.info, 488
psychopy.iohub.client, 310
psychopy.iohub.client.keyboard, 316
psychopy.logging, 490
psychopy.misc, 496
psychopy.parallel, 503
psychopy.preferences, 509
psychopy.sound, 510
psychopy.tools, 355
psychopy.tools.colorspacetools, 355
psychopy.tools.coordinatestools, 358
psychopy.tools.filetools, 359
psychopy.tools.gltools, 360
psychopy.tools.imagetools, 394
psychopy.tools.mathtools, 395
psychopy.tools.monitorunittools, 431
psychopy.tools.plottools, 433
psychopy.tools.typetools, 434
psychopy.tools.unittools, 434
psychopy.tools.viewtools, 436
psychopy.visual.windowframepack, 241
psychopy.visual.windowwarp, 241
pylink, 304
pyxid2, 252