PsychoPy - Psychology software for Python

Release 3.2.4

Jonathan Peirce

Oct 07, 2019

CONTENTS

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).

- Peirce, J. W., Gray, J. R., Simpson, S., MacAskill, M. R., Höchenberger, R., Sogo, H., Kastman, E., Lindeløv, J. (2019). PsychoPy2: experiments in behavior made easy. *Behavior Research Methods.* 10.3758/s13428-018-01193-y
- Peirce, J. W., & MacAskill, M. R. (2018). Building Experiments in PsychoPy. London: Sage.
- Peirce J. W. (2009). Generating stimuli for neuroscience using PsychoPy. *Frontiers in Neuroinformatics,* **2** (10), 1-8. doi:10.3389/neuro.11.010.2008
- Peirce, J. W. (2007). PsychoPy - Psychophysics software in Python. *Journal of Neuroscience Methods,* **162** (1-2):8-13 doi:10.1016/j.jneumeth.2006.11.017

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.

CHAPTER

TWO

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.:

```
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:

Units	Requires
norm (normalised to width/height)	n/a
pix (pixels)	Screen width in pixels
cm (centimeters on the screen)	Screen width in pixels and screen width in cm
deg (degrees of visual angle)	Screen width (pixels), screen width (cm) and distance (cm)

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 dont 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 its 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 isnt 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 guns 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 device-specific. 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.

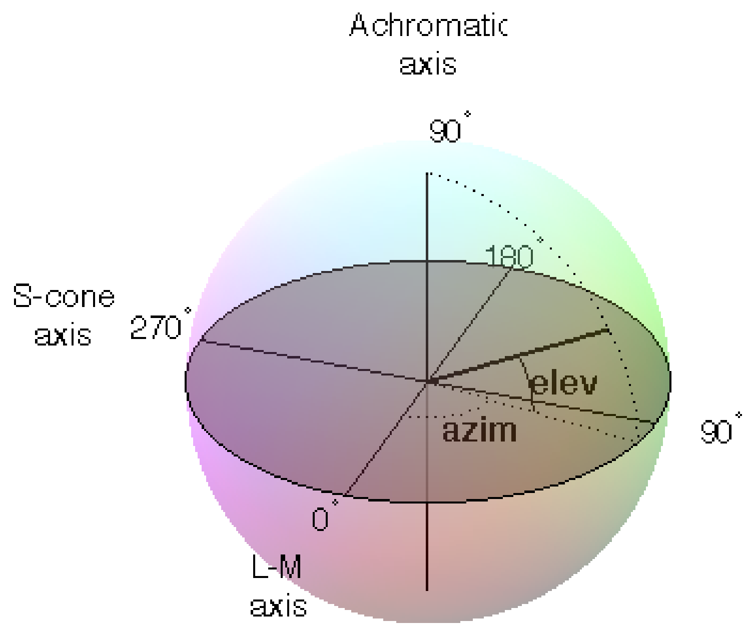
2.3.5 DKL color space

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

In the Derrington, Krauskopf and Lennie[1] color space (based on the Macleod and Boynton[2] 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).

[1] Derrington, A.M., Krauskopf, J., & Lennie, P. (1984). Chromatic Mechanisms in Lateral Geniculate Nucleus of Macaque. Journal of Physiology, 357, 241-265.

[2] MacLeod, D. I. A. & Boynton, R. M. (1979). Chromaticity diagram showing cone excitation by stimuli of equal luminance. Journal of the Optical Society of America, 69(8), 1183-1186.



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 = rn (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 PsychoPys audio engines provide the option not to sue portaudio but go directly to another lib (e.g. to coreaudio) but some dont 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.

These files are designed to be 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.

These 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 attributes of the Handler:

extraInfo the *extraInfo* dictionary provided to the Handler during its creation

trialList the list of dictionaries provided to the Handler during its creation

data a dictionary of 2D numpy arrays. Each entry in the dictionary represents a type of data (e.g. if you added rt data during your experiment using ~psychopy.data.TrialHandler.addData then rt will be a key). For each of those entries the 2D array represents the condition number and repeat number (remember that these start at 0 in python, unlike Matlab(TM) which starts at 1)

For example, to open a psydat file and examine some of its contents with:

```
from psychopy.misc import fromFile
datFile = fromFile('fileName.psydat')
#get info (added when the handler was created)
print datFile.extraInfo
#get data
print datFile.data
#get list of conditions
conditions = datFile.trialList
for condN, condition in enumerate(conditions):
    print condition, datFile.data['response'][condN], numpy.mean(datFile.data[
↪'response'][condN])
```

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 youre 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 sheetname) or a single file for multiple subjects (give the experiment name as the filename and the participant as the sheetname).

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 dont 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 (γ) 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:

$$L(V) = a + kV^{\gamma} \tag{2.1}$$

where L is the final luminance value, V is the requested intensity (ranging 0 to 1), a, k and γ 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:

$$LUT(V) = V^{1/\gamma} \tag{2.2}$$

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:

$$L(V) = a + (b + kV)^{\gamma} \tag{2.3}$$

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 γ.

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

$$LUT(V) = \frac{((1-V)b^{\gamma} + V(b+k)^{\gamma})^{1/\gamma} - b}{k} \tag{2.4}$$

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:

$$LUT(L) = \frac{((L-a)^{1/\gamma} - b)}{k} \tag{2.5}$$

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:

$$\begin{aligned} m &= a + b^{\gamma} \\ M &= a + (b+k)^{\gamma} \end{aligned} \tag{2.6}$$

Thus, the luminance value, *L* at any given point in the LUT, *V*, is given by

$$\begin{aligned} L(V) &= m + (M-m)V \\ &= a + b^{\gamma} + (a + (b+k)^{\gamma} - a - b^{\gamma})V \\ &= a + b^{\gamma} + ((b+k)^{\gamma} - b^{\gamma})V \\ &= a + (1-V)b^{\gamma} + V(b+k)^{\gamma} \end{aligned} \tag{2.7}$$

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):

$$LUT(L) = \frac{(L-a)^{1/\gamma} - b}{k} \tag{2.8}$$

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

$$\begin{aligned} LUT(V) &= \frac{(a + (1-V)b^{\gamma} + V(b+k)^{\gamma} - a)^{1/\gamma} - b}{k} \\ &= \frac{((1-V)b^{\gamma} + V(b+k)^{\gamma})^{1/\gamma} - b}{k} \end{aligned} \tag{2.9}$$

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 windows 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 computers 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 dont 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 č30. If youre 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 didnt 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 we each time we change the color of the stimulus we just 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.

blendMode = 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 opacity*stimRGB + (1-opacity)*backgroundRGB. This was the only mode available before PsychoPy version 1.80 and remains the default for the sake of backwards compatibility.

blendMode = add

If the window *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 *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 Projects

As of version 1.84 PsychoPy connects directly with the Open Science Framework website (http://OSF.io) allowing you to search for existing projects and upload your own experiments and data.

There are several reasons you may want to do this:

- sharing files with collaborators
- sharing files with the rest of the scientific community
- maintaining historical evidence of your work
- providing yourself with a simple version control across your different machines

2.8.1 Sharing with collaborators

You may find it simple to share files with your collaborators using dropbox but that means your data are stored by a commercial company over which you have no control and with no interest in scientific integrity. Check with your ethics committee how they feel about your data (e.g. personal details of participants?) being stored on dropbox. OSF, by comparison, is designed for scientists to stored their data securely and forever.

Once youve created a project on OSF you can add other contributors to it and when they log in via PsychoPy they will see the projects they share with you (as well as the project they have created themselves). Then they can sync with that project just like any other.

2.8.2 Sharing files/projects with others

Optionally, you can make your project (or subsets of it) publicly accessible so that others can view the files. This has various advantages, to the scientific field but also to you as a scientist.

Good for open science:

- Sharing your work allows scientists to work out why one experiment gave a different result to another; there are often subtleties in the exact construction of a study that didnt get described fully in the methods section. By sharing the actual experiment, rather than just a description of it, we can reduce the failings of replications
- Sharing your work helps others get up and running quickly. Thats good for the scientific community. We want science to progress faster and with fewer mistakes.

Some people feel that, having put in all that work to create their study, it would be giving up their advantage to let others simply use their work. Luckily, sharing is good for you as a scientist as well!

Good for the scientist:

- When you create a study you want others to base their work on yours (we call that academic impact)
- By giving people the exact materials from your work you increase the chance that they will work on your topic and base their next study on something of yours
- By making your project publicly available on OSF (or other sharing repository) you raise visibility of your work

You dont need to decide to share immediately. Probably you want your work to be private until the experiment is complete and the paper is under review (or has been accepted even). Thats fine. You can create your project and keep it private between you and your collaborators and then share it at a later date with the click of a button.

2.8.3 Maintaining a validated history of your work

In many areas of science researchers are very careful about maintaining a full documented history of what their work; what the

- you can preregister your plans for the next experiment (so that people cant later accuse you of p-hacking).
- all your files are timestamped so you can prove to others that they were collected on/by a certain date, removing any potential doubts about who collected data first
- your projects (and individual files) have a unique URL on OSF so you can cite/reference resources.

Additionally, Registrations (snapshots of your project at a fixed point in time) can be given a DOI, which guarantees they will exist permanently

2.8.4 Using PsychoPy to sync with OSF

PsychoPy doesnt currently have the facility to *create* user profiles or projects, so the first step is for you to do that yourself.

Login to OSF

From the *Projects* menu you can log in to OSF with your username and password (this is never stored; see *Security*). This user will stay logged in while the PsychoPy application remains open, or until you switch to a different user. If you select Remember me then your login will be stored and you can log in again without typing your password each time.

Projects that you have previously synchronised will try to use the stored details of the known users if possible and will revert to username and password if not. Project files (defining the details of the project to sync) can be stored wherever you choose; either in a private or shared location. User details are stored in the home space of the user currently logged in to the operating system so are not shared with other users by default.

Security

When you log in with your username and password these details are not stored by PsychoPy in any way. They are sent immediately to OSF using a secure (https) connection. OSF sends back an authorisation token identifying you as a valid user with authorised credentials. This is stored locally for future log in attempts. By visiting your user profile at http://OSF.io you can see what applications/computers have retrieved authorisation tokens for your account (and revoke them if you choose).

The auth token is stored in plain text on your computer, but a malicious attacker with access to your computer could only use this to log in to OSF.io. They could not use it to work out your password.

All files are sent by secure connection (https) to the server.

Searching for projects

Having logged in to OSF from the projects menu you can search for projects to work with using the *>Projects>Search* menu. This brings up a view that shows you all the current projects for the logged in user (owned or shared) and allows you to search for public projects using tags and/or words in the title.

When you select a project, either in your own projects or in the search box, then the details for that project come up on the right hand side, including a link to visit the project page on the web site.

On the web page for the project you can fork the project to your own username and then you can use PsychoPy to download/update/sync files with that project, just as with any other project. The project retains information about its history; the project from which it was forked gets its due credit.

Synchronizing projects

Having found your project online you can then synchronize a local folder with that set of files.

To do this the first time:

- select one of your projects in the project search window so the details appear on the right
- press the Sync button
- the Project Sync dialog box will appear
- set the location/name for a project file, which will store information about the state of files on the last sync
- set the location of the (root) folder locally that you want to be synchronised with the remote files
- press sync

The sync process and rules:

- on the first synchronisation all the files/folders will be merged: - the contents of the local folder will be uploaded to the server and vice versa - files that have the same name but different contents (irrespective of dates) will be flagged as conflicting (see below) and both copies kept
- on subsequent sync operations a two-way sync will be performed taking into account the previous state. **If you delete the files locally and then sync then they will be deleted remotely as well**
- files that are the same (according to an md5 checksum) and have the same location will be left as they are
- if a file is in conflict (it has been changed in both locations since the last sync) then both versions will be kept and will be tagged as conflicting
- if a file is deleted in one location but is also changed in the other (since the last sync) then it will be recreated on the side where it was deleted with the state of the side where is was not deleted.

Conflicting files will be labelled with their original filename plus the string _CONFLICT<datetimestamp> Deletion conflicts will be labelled with their original filename plus the string _DELETED

Limitations

- PsychoPy does not directly allow you to create a new project yet, nor create a user account. To start with you need to go to http://osf.io to create your username and/or project. You also cannot currently fork public projects to your own user space yet from within PsychoPy. If you find a project that is useful to you then fork it from the website (the link is available in the details panel of the project search window)
- The synchronisation routines are fairly basic right now and will not cater for all possible eventualities. For example, if you create a file locally but your colleague created a folder with the same name and synced that with the server, it isnt clear what will (or should ideally) happen when you now sync your project. You should be careful with this tool and always back up your data by an independent means in case damage to your files is caused
- This functionality is new and may well have bugs. **User beware!**

2.9 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 considerations for specific designs*.

Something that people seem to forget (not helped by the software manufacturers that keep talking about their sub-millisecond precision) is that the monitor, keyboard and human participant DO NOT have anything like this sort of precision. Your monitor updates every 10-20ms depending on frame rate. If you use a CRT screen then the top is drawn before the bottom of the screen by several ms. If you use an LCD screen the whole screen can take around 20ms to switch from one image to the next. Your keyboard has a latency of 4-30ms, depending on brand and system.

So, yes, PsychoPys temporal precision is as good as most other equivalent applications, for instance the duration for which stimuli are presented can be synchronised precisely to the frame, but the overall accuracy is likely to be severely limited by your experimental hardware. To get **very** precise timing of responses etc., you need to use specialised hardware like button boxes and you need to think carefully about the physics of your monitor.

Warning: The information about timing in PsychoPy assumes that your graphics card is capable of synchronising with the monitor frame rate. For integrated Intel graphics chips (e.g. GMA 945) under Windows, this is not true and the use of those chips is not recommended for serious experimental use as a result. Desktop systems can have a moderate graphics card added for around č30 which will be vastly superior in performance.

2.9.1 Specific considerations for specific designs

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 *arent 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).

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 dont need it, but it can be turned on any time after `Window` creation:

```
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 dont start recording frame times until that has ended. Obviously if you arent 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:

```
from __future__ import division, print_function

from psychopy import visual, logging
win = visual.Window([800,600])

win.recordFrameIntervals = True

# 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.
win.refreshThreshold = 1/85 + 0.004

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

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:

```
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:

```
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.

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 dont 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 mult**

    - anti-virus auto-updating (if youre 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 doesnt have to spend time working out what application is currently getting keyboard/mouse events.
2. dont 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 h**

   1. GratingStim.setTexture()
   2. RadialStim.setTexture()
   3. TextStim.setText()

4. if you dont 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 isnt 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 Holcombes page here
3. use a single screen rather than two (probably there is some graphics card overhead in managing double the number of pixels?)

Comparing Operating Systems under PsychoPy

This is an attempt to quantify the ability of PsychoPy draw without dropping frames on a variety of hardware/software. The following tests were conducted using the script at the bottom of the page. Note, of course that the hardware fully differs between the Mac and Linux/Windows systems below, but that both are standard off-the-shelf machines.

All of the below tests were conducted with normal systems rather than anything that had been specifically optimised:

- the machines were connected to network
- did not have anti-virus turned off (except Ubuntu had no anti-virus)
- they even all had dropbox clients running
- Linux was the standard (not realtime kernel)

No applications were actively being used by the operator while tests were run.

In order to test drawing under a variety of processing loads the test stimulus was one of:

- a single drifting Gabor
- 500 random dots continuously updating
- 750 random dots continuously updating
- 1000 random dots continuously updating

Common settings:

- Monitor was a CRT 1024x768 100Hz
- all tests were run in full screen mode with mouse hidden

System Differences:

- the iMac was lower spec than the Windows/Linux box and running across two monitors (necessary in order to connect to the CRT)

- the Windows/Linux box ran off a single monitor

Each run below gives the number of dropped frames out of a run of 10,000 (2.7 mins at 100Hz).

_	Windows XP	Windows 7	Mac OS X 10.6	Ubuntu 11.10
_	(SP3)	Enterprise	Snow Leopard	
Gabor	0	5	0	0
500-dot RDK	0	5	54	3
750-dot RDK	21	7	aborted	1174
1000-dot RDK	776	aborted	aborted	aborted
GPU	Radeon 5400	Radeon 5400	Radeon *2400*	Radeon 5400
GPU driver	Catalyst 11.11	Catalyst 11.11		Catalyst 11.11
CPU	Core Duo 3GHz	Core Duo 3GHz	Core Duo 2.4GHz	Core Duo 3GHz
RAM	4GB	4GB	2GB	4GB

Ill gradually try to update these tests to include:

- longer runs (one per night!)
- a faster Mac
- a real-time Linux kernel

2.9.2 Other questions about timing

Can PsychoPy deliver millisecond precision?

The simple answer is yes, given some additional hardware. 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 Cedrus Response Pad) and use PsychoPys serial port commands to retrieve information about responses and timing with a precision of around 1ms.

Before conducting your experiment in which effects might be on the order of 1 ms, do consider that;

- your screen has a temporal resolution of ~10 ms
- your visual system has a similar upper limit (or you would notice the flickering screen)
- human response times are typically in the range 200-400 ms and very variable
- USB keyboard latencies are variable, in the range 20-30ms

That said, PsychoPy does aim to give you as high a temporal precision as possible, and is likely not to be the limiting factor of your experiment.

Computer monitors

Monitors have fixed refresh rates, typically 60 Hz for a flat-panel display, higher for a CRT (85-100 Hz are common, up to 200 Hz is possible). For a refresh rate of 85 Hz there is a gap of 11.7 ms between frames and this limits the timing of stimulus presentation. You cannot have your stimulus appear for 100 ms, for instance; on an 85Hz monitor it can appear for either 94 ms (8 frames) or 105 ms (9 frames). There are further, less obvious, limitations however.

For CRT (cathode ray tube) screens, 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? For flat panel displays, or (or LCD projectors) your image will be presented simultaneously all over the screen, but it takes up to 20 ms(!!) for your pixels to go all the way from black to white (manufacturers of these panels quote values of 3 ms for the fastest panels, but they certainly dont mean 3 ms white-to-black, I assume they mean 3 ms half-life).

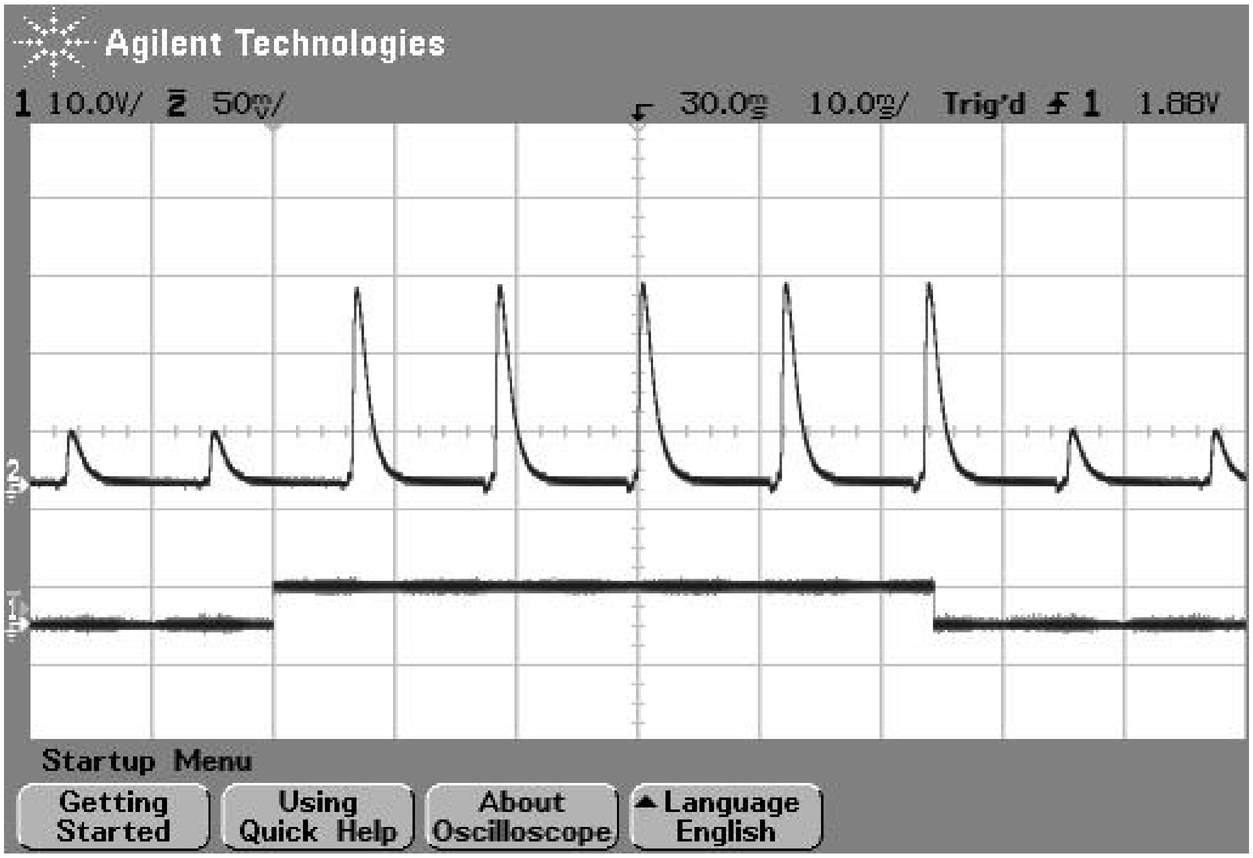


Fig. 1: Figure 1: photodiode trace at top of screen. The image above shows the luminance trace of a CRT recorded by a fast photo-sensitive diode at the top of the screen when a stimulus is requested (shown by the square wave). The square wave at the bottom is from a parallel port that indicates when 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.

Warning: If youre using a regular computer display, *you have a hardware-limited temporal precision of 10 ms irrespective of your response box or software clocks etc* and should bear that in mind when looking for effect sizes of less than that.

Can I have my stimulus to appear with a very precise rate?

Yes. Generally to do that you should time your stimulus (its onset/offset, its rate of change) using the frame refresh rather than a clock. e.g. you should write your code to say for 20 frames present this stimulus rather than for 300ms present this stimulus. Provided your graphics card is set to synchronise page-flips with the vertical blank, and provided that you arent *dropping frames* the frame rate will always be absolutely constant.

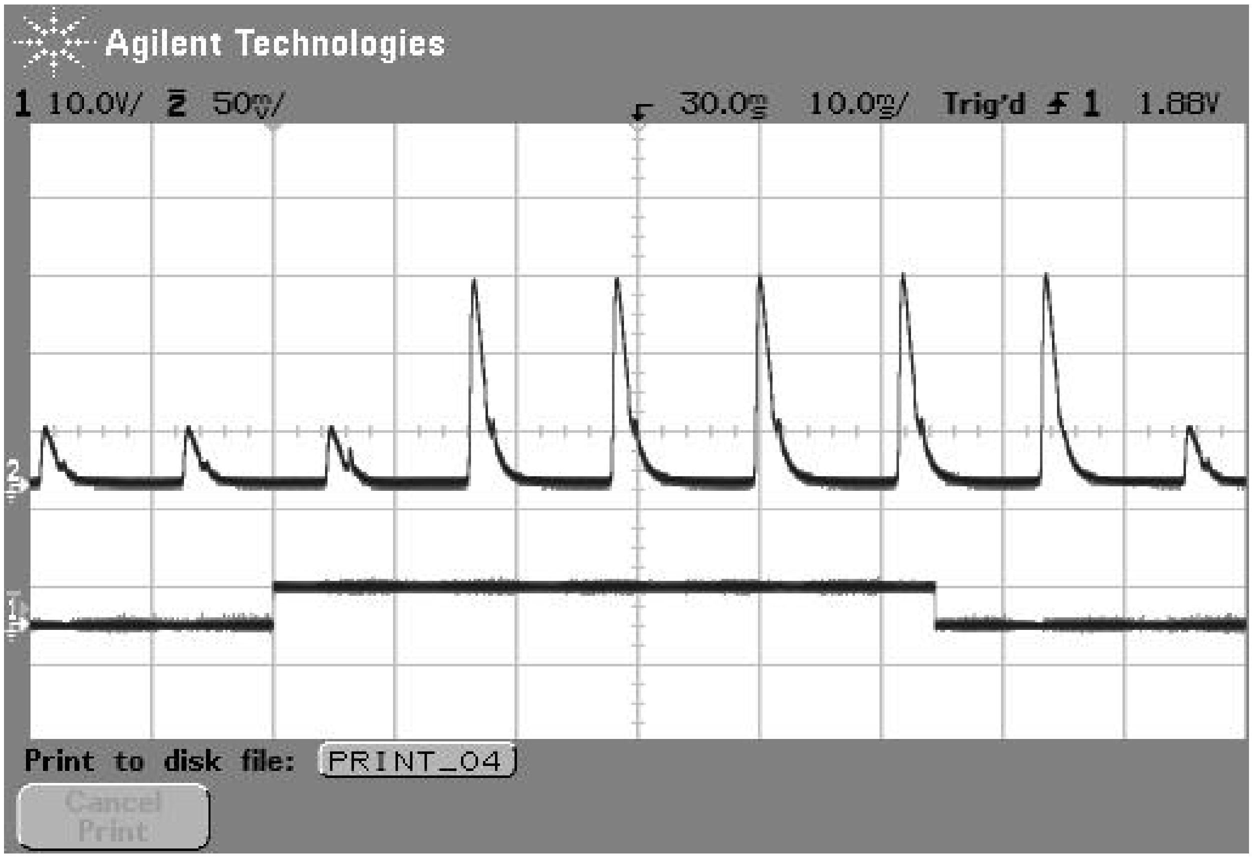


Fig. 2: Figure 2: photodiode trace of the same large stimulus at bottom of screen. The image above shows comes from exactly the same script as the above but the photodiode is positioned at the bottom of the screen. In this case, after the stimulus is requested the current frame (which is dark) finishes drawing and then, 10ms later than the above image, the screen goes bright at the bottom.

2.10 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 participants 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.

CHAPTER

THREE

INSTALLATION

3.1 Download

For the easiest installation download and install the Standalone package.

For all versions see the PsychoPy releases on github

3.2 Manual installations

See below for options if you dont want to use the Standalone releases:

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

3.2.1 pip install

Now that most python libraries can be install using *pip* its relatively easy to manually install PsychoPy and all its 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 youll 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 Linux

There used to be neurodebian and Gentoo packages for PsychoPy but these are both badly outdated. Wed 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:

https://extras.wxpython.org/wxPython4/extras/linux/gtk3/

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 doesnt 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 havent bee able to build universal wheels for these yet so you may have to build the pkg yourself. That should 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

3.2.3 Anaconda and Miniconda

With Python 3.6:

```
conda create -n psypy3 python=3.6
conda activate psypy3
conda install numpy scipy matplotlib pandas pyopengl pillow lxml openpyxl xlrd␣
↪configobj pyyaml gevent greenlet msgpack-python psutil pytables requests[security]␣
↪cffi seaborn wxpython cython pyzmq pyserial
conda install -c conda-forge pyglet pysoundfile python-bidi moviepy pyosf
pip install zmq json-tricks pyparallel sounddevice pygame pysoundcard psychopy_ext␣
↪psychopy
```

3.2.4 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 isnt very good). Try installing new drivers for your graphics card **from its manufacturers web page,** not from Microsoft. For example, NVIDIA provides drivers for its cards here: https://www.nvidia.com/Download/index.aspx

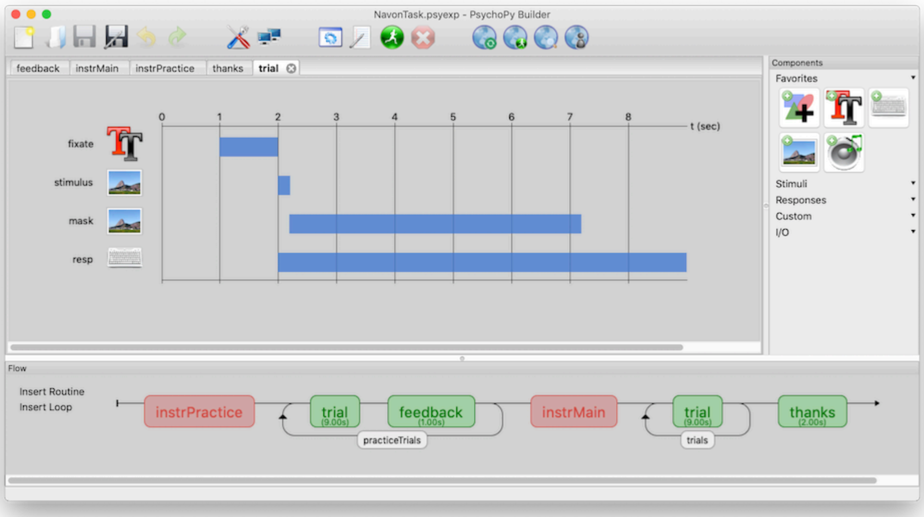
CHAPTER

FOUR

GETTING STARTED

As an application, PsychoPy has two main views: the *Builder* view, and the *Coder* view. It also has a underlying *API* that you can call directly.

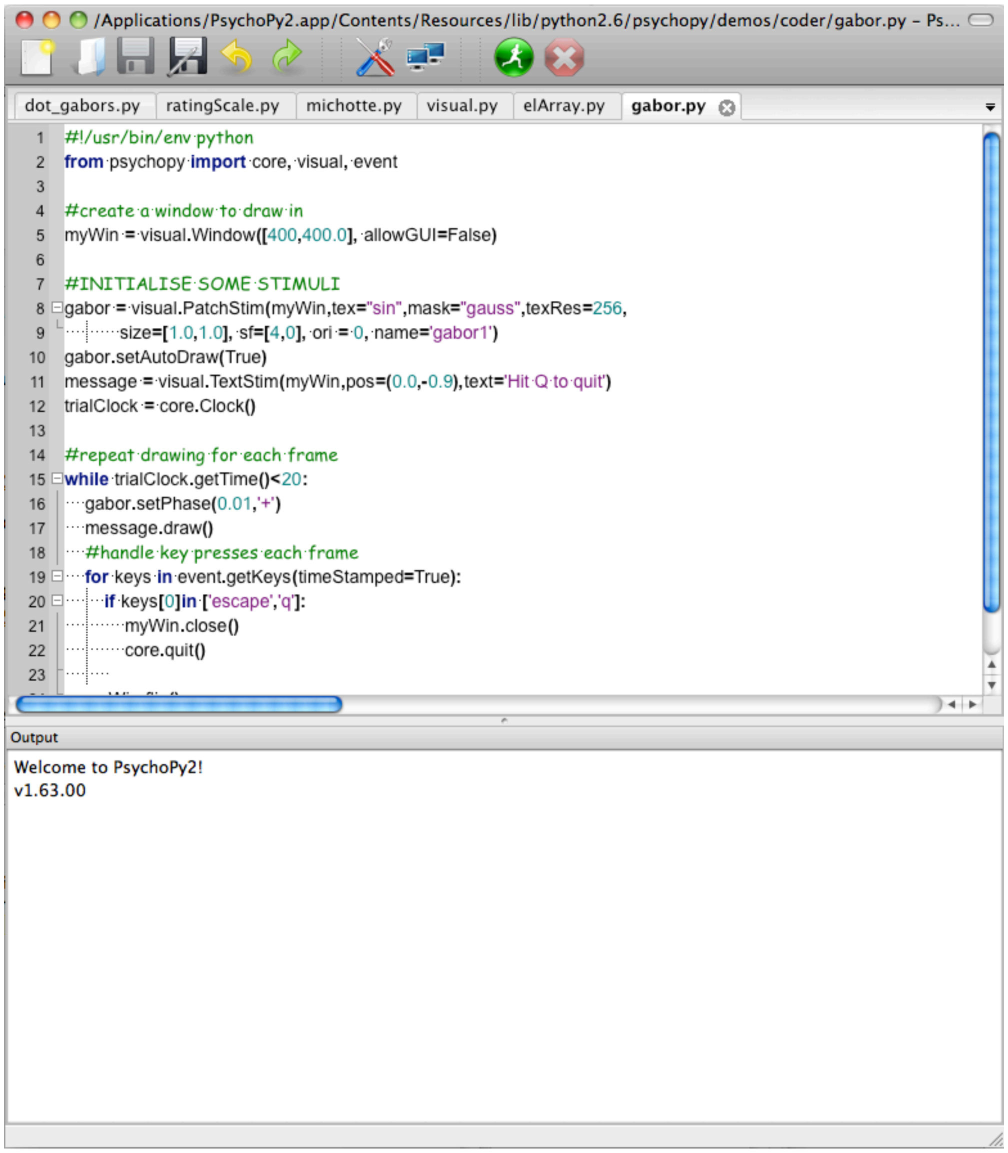
1. *Builder*. You can generate a wide range of experiments easily from the Builder using its intuitive, graphical user interface (GUI). This might be all you ever need to do. But you can always compile your experiment into a python script for fine-tuning, and this is a quick way for experienced programmers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

The Builder and Coder views are the two main aspects of the PsychoPy application. If youve 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, its 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.


/Applications/PsychoPy2.app/Contents/Resources/lib/python2.6/psychopy/demos/coder/gabor.py – Ps…
dot_gabors.py
ratingScale.py
michotte.py
visual.py
elArray.py
gabor.py
Output

```
#!/usr/bin/env python
from psychopy import core, visual, event

#create a window to draw in
myWin = visual.Window([400,400.0], allowGUI=False)

#INITIALISE SOME STIMULI
gabor = visual.PatchStim(myWin,tex="sin",mask="gauss",texRes=256,
        size=[1.0,1.0], sf=[4,0], ori = 0, name='gabor1')
gabor.setAutoDraw(True)
message = visual.TextStim(myWin,pos=(0.0,-0.9),text='Hit Q to quit')
trialClock = core.Clock()

#repeat drawing for each frame
while trialClock.getTime()<20:
    gabor.setPhase(0.01,'+')
    message.draw()
    #handle key presses each frame
    for keys in event.getKeys(timeStamped=True):
        if keys[0]in ['escape','q']:
            myWin.close()
            core.quit()
```

```
Welcome to PsychoPy2!
v1.63.00
```

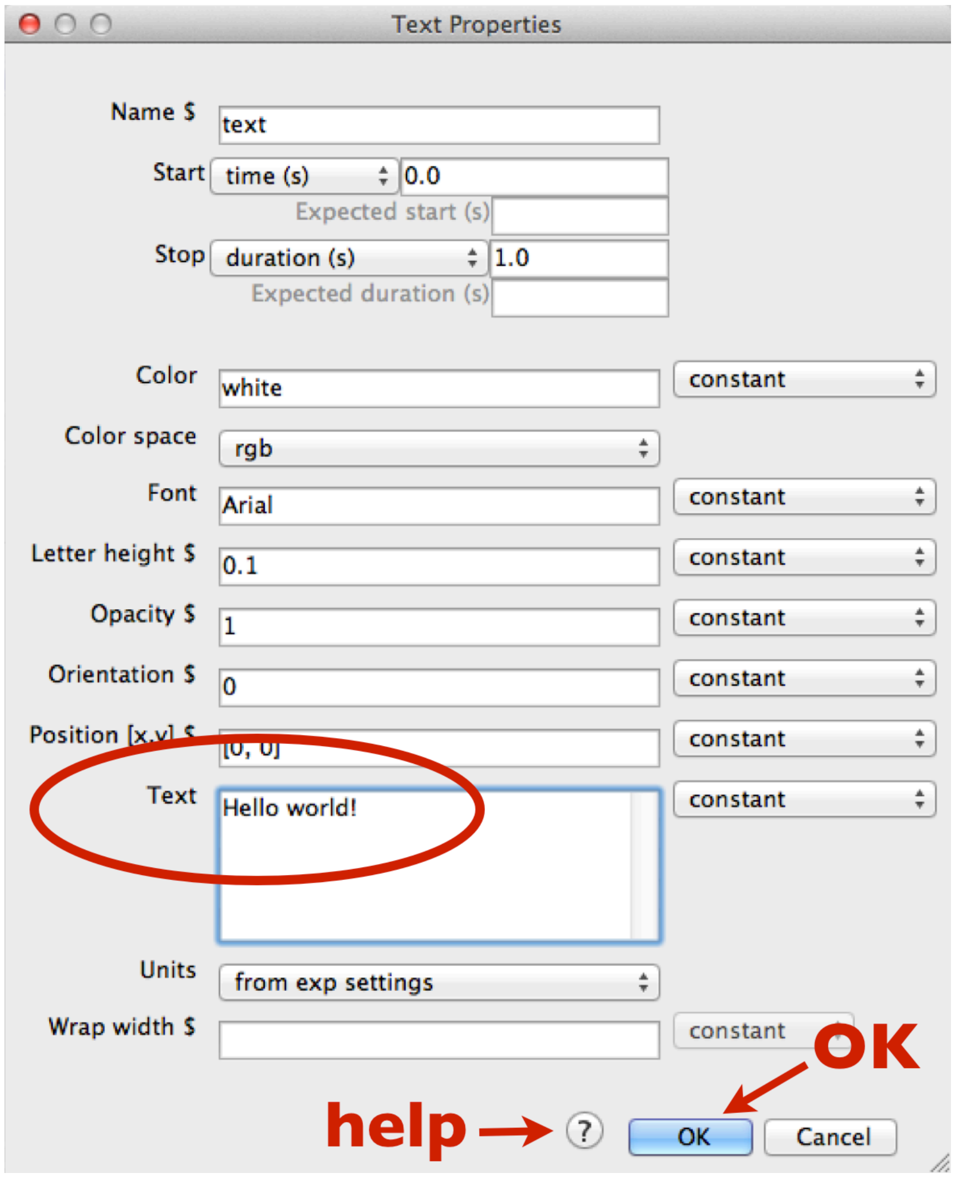
4.1 Builder

When learning a new computer language, the classic first program is simply to print or display Hello world!. Lets 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

Name $ text

Start time (s) 0.0

Expected start (s)

Stop duration (s) 1.0

Expected duration (s)

Color white constant

Color space rgb

Font Arial constant

Letter height $ 0.1 constant

Opacity $ 1 constant

Orientation $ 0 constant

Position [x,y] $ [0, 0] constant

Text Hello world! constant

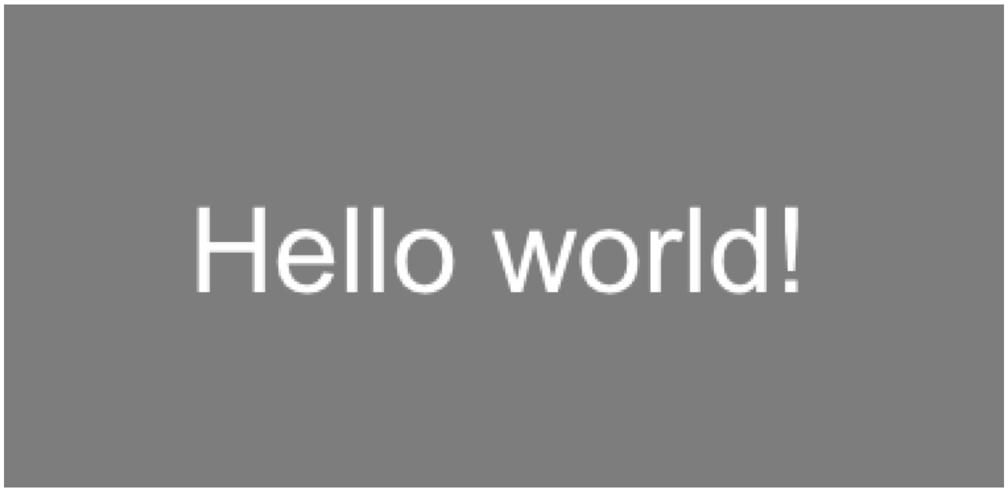
Units from exp settings

Wrap width $ constant

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 helpan 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!

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, youll 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 its 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 Unicode (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:

```
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 Coders output window is shown at the bottom of the Coder window. Click on the Shell tab, and you should see pythons 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 onwatch 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).

CHAPTER

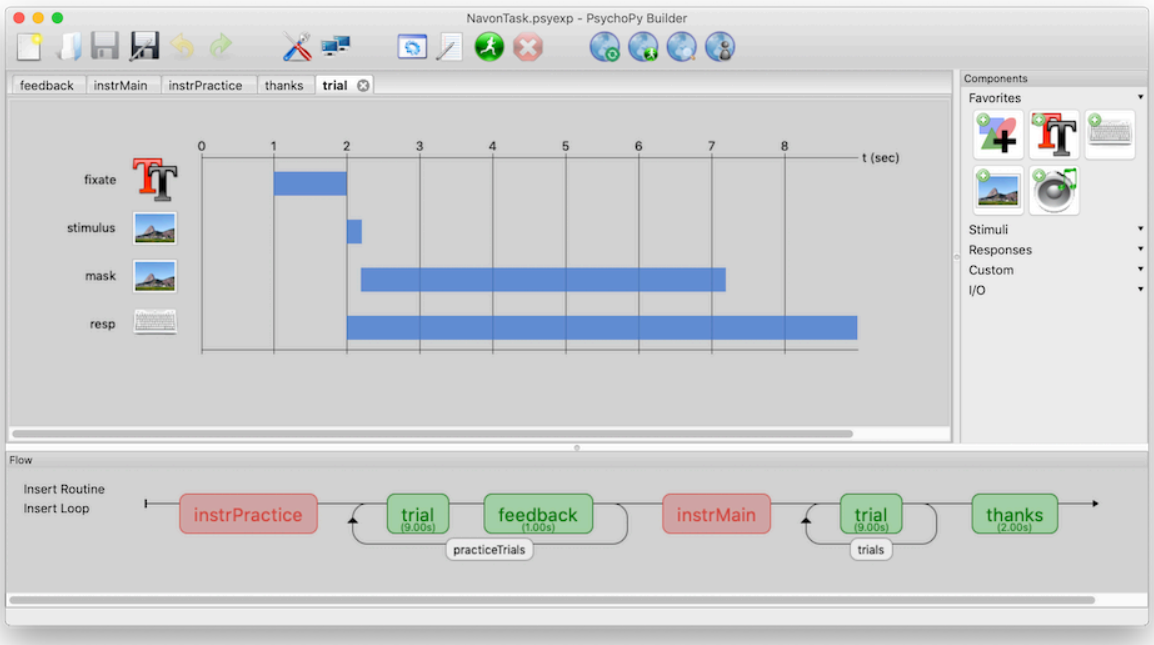
FIVE

BUILDER

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 experiments *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 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 *resp* Component in *trial* *Routine*)

[1] Stroop, J.R. (1935). Studies of interference in serial verbal reactions. Journal of Experimental Psychology 18: 643-662.

- How to change a stimulus parameter on each repetition of the loop. The text and rgb values of the *word* Component are based on *thisTrial*, which represents a single iteration of the *trials* loop. They have been set to change every repeat (dont forget that step!)
- How to present instructions: just have a long-lasting *TextStim* and then force end of the *Routine* when a key is pressed (but dont 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 *trialClock.getTime()*2*. Every *Routine* has a clock associated with it that gets reset at the beginning of the iteration through the *Routine*. There is also a *globalClock* that can be used in the same way. The phase of a *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 *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 *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. Thats 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 alphanumeric 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 dont 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:

```
ori   text    corrAns
0     aaa     left
90    aaa     left
0     bbb     right
90    bbb     right
```

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. Its 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:

```
letter
a
b
c
```

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 c a b]. 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 as 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 c 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 5th entry of a list - Python starts with index zero)
- *random(4)*10* gives 4 indices from 0 to 10 (so selects 4 out of 11 conditions)
- *5:10* selects the 6th to 9th rows
- *$myIndices* uses a variable that youve already created

Note in the last case that *5:8* isnt 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 Pythons built-in slicing syntax (where the first index is zero and the last entry of a slice doesnt get included). You might want to check the outputs of your selection in the Python shell (bottom of the Coder view) like this:

```
>>> 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 wont 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

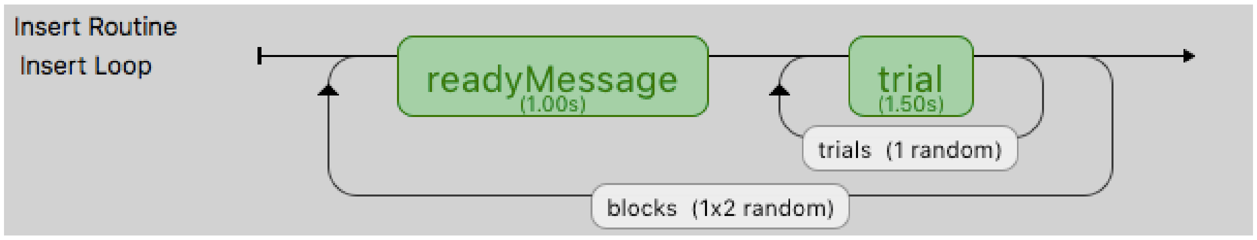
Many people ask how to create blocks of trials, how to randomise them, and how to counterbalance their order. This isnt all that hard, although it does require a bit of thinking!

5.4.1 Blocking

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, lets imagine youre 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.

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:

- facesBlock.xlsx
- housesBlock.xlsx
- chooseBlocks.xlsx

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 youd probably also have response keys andsuchlike as well.

So, how to switch between these files? Thats the trick and thats 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 were 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

Counterbalancing is simply an extension of blocking. Usually with a block design you would set the order of blocks to be set randomly. In the example above the blocks are set to occur randomly, but note that they could also be set to occur more than once if you want 2 repeats of the 2 blocks for a total of 4.

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 wed have something like *groupA.xlsx*:

condsFile
housesBlock.xlsx
facesBlock.xlsx

and *groupB.xlsx*:

condsFile
facesBlock.xlsx
housesBlock.xlsx

In this case 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 outside of PsychoPy and simply tells PsychoPy which group to assign to each participant.

The easiest way to do that is to add the field *group* to the initial dialog box, maybe with the default value of *A*. If you set the conditions file for the *blocks* loop to be ` ` $"group"+expInfo['group']+".xlsx" ` ` then this variable will be used from the dialog box to create the filename for the blocks file and you.

Also, if youre doing this, remember to set the *blocks* loop to use sequential rather than random sorting. Your inner loop still probably wants to be random (to shuffle the image order within a block) but your outer loop should now be using exactly the order that you specified in the blocks condition file.

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 cant 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.

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

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.

line settings: Control color and width of the line. The line width is always specified in pixels - it does not honour the *units* parameter.

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

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.

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.

Correct answer: button The correct answer, used by Store correct.

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

Advanced

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 isnt 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 `Cl`
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*:

Begin Experiment: Things that need to be done just once, like importing a supporting module, initialising a variable for later use.

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).:

```
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*.:

```
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:

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

and put this in the *Begin routine* field:

```
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 fields 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 whats 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 youve 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 numpys 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.

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.

nDots [int] number of dots to be generated

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.

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 als (1996) categories of RDK.

noiseDots [*direction*, position or walk] Determines the behaviour of the noise dots, taken directly from Scase et als (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.

See also:

API reference for `DotStim`

5.5.6 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

questionText The item question string

questionWidth 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

questionColor The question text font color

responseColor The response object color

Missing column headers will be replaced by default entries. The default entries are:

index 0 (increments for each item)

questionText Default question

questionWidth 0.7

type rating

responseWidth 0.3

options Yes, No

layout horiz

questionColor white

responseColor white

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

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.

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

randomize [bool] Randomize order of Form elements

See also:

API reference for `Form`

5.5.7 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 youre 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.

Color : See *Color spaces*

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

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

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*

Advanced Settings

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 is 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`

5.5.8 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.

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.

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

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

Advanced Settings

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

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

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`

5.5.9 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 boxs 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.

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).

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`.

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.

Correct answer: button The correct answer, used by Store correct.

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

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

See also:

API reference for `iolab`

5.5.10 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/joystic 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.

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*.

Advanced Settings

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

5.5.11 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 immage on each screen frame using the joystick coordinates: joystick.getX() and joystick.getY(). To ensure that the cursor image is drawon 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.

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.

Parameters Advanced

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

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

See also:

API reference for `Joystick`

5.5.12 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.

Allowed keys 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 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 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 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*

5.5.13 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**:

```
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.

See also:

API reference for `AdvAudioCapture`

5.5.14 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.

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?

See also:

API reference for `Mouse`

5.5.15 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.

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

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

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.

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

See also:

API reference for `MovieStim`

5.5.16 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.

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.

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.

See also:

API reference for `iolab`

5.5.17 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 is 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`

5.5.18 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

- 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).

nVertices : integer

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.

fill settings:

Control the color inside the shape. If you set this to *None* then you will have a transparent shape (the line will remain)

line settings:

Control color and width of the line. The line width is always specified in pixels - it does not honour the *units* parameter.

size [[w,h]] See note above

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.

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

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

5.5.19 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.

Pump index [int] The index of the pump: The first pumps index is 0, the second pumps 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 osition after the pump operation has finished. This can be used to ensure a sharp(er) stimulus offset.

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

5.5.20 RatingScale Component

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.

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.

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 : Brief instructions, reminding the subject how to interpret the numerical scale, default = 1 = not at all extremely = 7

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

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

Advanced settings

single click : If this box is checked the participant can only click the scale once and their response will be stored. If this box is not checked the participant must accept their rating before it is stored.

startTime [float or integer] The time (relative to the beginning of this Routine) that the rating scale should first appear.

forceEndTrial : If checked, when the subject makes a rating the routine will be ended.

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).

duration : The maximum duration in seconds for which the stimulus is presented. See duration for details. Typically, the subjects response should end the trial, not a duration. A blank or negative value means wait for a very long time.

storeRatingTime: Save the time from the beginning of the trial until the participant responds.

storeRating: Save the rating that was selected

lowAnchorText [str] Custom text to display at the low end of the scale, e.g., 0%; overrides low setting

highAnchorText [str] Custom text to display at the low end of the scale, e.g., 100%; overrides high setting

customize_everything [str] If this is not blank, it will be used when initializing the rating scale just as it would be in a code component (see `RatingScale`). This allows access to all the customizable aspects of a rating scale, and supersedes all of the other RatingScale settings in the dialog panel. (This does not affect: startTime, forceEndTrial, duration, storeRatingTime, storeRating.)

See also:

API reference for `RatingScale`

5.5.21 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. Csh Bf)
- a filename, which can be a relative or absolute path (mid, wav, and ogg are supported).

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

See also:

API reference for `SoundPyo`

5.5.22 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 isnt 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 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`

5.5.23 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.

color : See *Color spaces*

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

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

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.

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

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

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.

See also:

API reference for `TextStim`

5.5.24 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 you 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.

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

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

5.5.25 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)

5.5.26 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: (new in version 1.80.00): 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:

```
# all in data folder: 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
# on Windows you may need to replace ~ with your home directory
'~/dropbox/data/%s/%s-%s' %(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 that .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 doesnt 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 preceeding 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 Gotchas)

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 wont be printed)
- Have you entered your the settings for your *monitor*? If you are using degrees as a unit of measurement and have not 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 isnt appearing, theres only the grey background

- Have you checked the size of your stimulus? If it is 0.5x0.5 pixels you wont be able to see it!
- Have you checked the position of your stimulus? Is it positioned off the screen?

5.9.3 The loop isnt 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 its turning into a grating

- If you dont want your stimulus to have a texture, you need Image to be None

5.9.5 The code snippet Ive entered doesnt do anything

- Have you remembered to put a $ symbol at the beginning (this isnt 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 isnt 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 Im 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 Gauss
- 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

Its 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 were 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.

CHAPTER

SIX

CODER

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. Its 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')
```

(continues on next page)

(continued from previous page)

```
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 cant 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:

```
stim.attribute += value  # addition
stim.attribute -= value  # subtraction
stim.attribute *= value  # multiplication
stim.attribute /= value  # division
stim.attribute %= value  # modulus
stim.attribute **= value # power
```

They are easy to use and understand on scalars:

```
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:

```
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 cant be used while something is changing. For more flexible timing, you could use a `Clock()` object from the `core` module:

```
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 frame rate 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, dont 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:  # Present fixation for a subset of frames
        fixation.draw()
    if 50 <= frameN < 100:  # Present stim for a different subset
        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*:

```
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):

```
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:

```
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:

```
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):

```
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.

StairHandler

This generates the next trial using an *adaptive staircase*. The conditions are not predetermined and are generated based on the participants 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, lets ensure no global event keys are currently set by calling func:*event.globalKeys.clear*.

```
>>> 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.

```
>>> 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.

```
>>> event.globalKeys
<_GlobalEventKeys :
    [A] -> 'myfunc' <function myfunc at 0x10669ba28>
>
```

Your output should look similar. You may happen to spot We can take a closer look at the specific global key event we added.

```
>>> 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 wont 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, lets create the dummy function:

```
>>> 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 uples, and keyword arguments as dictionaries.

```
>>> 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.

```
>>> 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.

```
>>> 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.

```
>>> event.globalKeys
<_GlobalEventKeys :
    [A] -> 'myfunc' <function myfunc at 0x10669ba28>
    [CTRL] + [ALT] + [B] -> 'my name' <function myfunc2 at 0x112eecb90>
>
```

The key combination *[CTRL] + [ALT] + [B]* is now associated with the function *myfunc2*, which will be called in the following way:

```
myfunc2(1, 2, foo=2, bar=4)
```

Indexing

event.globalKeys can be accessed like an ordinary dictionary. The index keys are *(key, modifiers)* namedtuples.

```
>>> 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

```
>>> event.globalKeys['b', ['ctrl', 'alt']]
_GlobalEvent(func=<function myfunc2 at 0x112eecb90>, 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.

```
>>> event.globalKeys['a', ()]
_GlobalEvent(func=<function myfunc at 0x10669ba28>, func_args=(), func_kwargs={},␣
↪name='myfunc')
>>> 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.

```
>>> index = ('b', ['ctrl', 'alt'])
>>> event.globalKeys[index].func
<function myfunc2 at 0x112eecb90>
>>> 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.

```
>>> 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()`.

```
>>> event.globalKeys.remove(key='a')
```

A convenience method to quickly delete *all* global event keys is to pass *key=all*

```
>>> 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*.

```
>>> 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.

```
>>> 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).

```
>>> from psychopy.preferences import prefs
>>> prefs.general['shutdownKey'] = 'q'
```

We can now check if a global shutdown key has been automatically created.

```
>>> 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 scences? 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.

```
>>> 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'.

```
>>> from psychopy import core, event
>>> event.globalKeys.add(key='q', func=core.quit, name='shutdown')
```

Thats it!

A working example

In the above code snippets, our global event keys were not actually functional, as we didnt 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.

```
#!/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,
                       text='Press C to change color,\n CTRL + Q to quit.')

# 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 screen size 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. Heres an example. Type it into a coder window, save it somewhere and press run.

```python
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 arent needed.

Thats a bit easy though. Lets 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. Thats easy to add too. In the first line add `event` to the list of modules youll import. Then replace the line:

```
for frameN in range(200):
```

with the line:

```
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:

```
    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 youve collected the events you want (otherwise it gets full of events that we dont care about like the mouse moving around etc).

Your finished script should look something like this:

```
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 youre 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 isnt 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):

```
"""measure your JND in orientation using a staircase method"""
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 *Heres 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 youve written. For single-line comments, as youll 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 well 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, lets allow the user to change them in a dialogue box (which well 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 well use the updated values and save them straight to a parameters file (the one we try to load above).

If they hit *Cancel* then well 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

Well 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 were showing you the manual way to do it.

Well 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 well set it up to begin with an increment of 20deg (very detectable) and home in on the 80% threshold value. Well 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). Well 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.

```
# 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 theyre ready. The next two lines create a pair of messages, then draw them into the screen and then update the screen to show what weve drawn. Finally we issue the command event.waitKeys() which will wait for a keypress before continuing.

```
# 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
```

Then set the orientation of the foil to be the reference orientation plus `thisIncrement`, draw all the stimuli (including the fixation point) and update the window.

```python
    # set orientation of probe
    foil.setOri(expInfo['refOrientation'] + thisIncrement)

    # 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).

```python
    # 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 - youll 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 hasnt 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 were 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']:
```

(continues on next page)

(continued from previous page)

```
                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.

```
    # add the data to the staircase so it can calculate the next level
    staircase.addData(thisResp)
    dataFile.write('%i,%.3f,%i\n' %(targetSide, thisIncrement, thisResp))
    core.wait(1)
```

Output your data and clean up

OK! Were basically done! Weve 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.

```
# staircase has ended
dataFile.close()
staircase.saveAsPickle(fileName)  # special python binary file to save all the info
```

While were here, its quite nice to give some immediate feedback to the user. Lets 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!

```
# 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). In can be imported in different ways; the `import pylab` line at the beginning of the script is the way to import matploblib as well as a variety of other scientific tools (that arent 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
allIntensities, allResponses = [],[]
for thisFileName in files:
    thisDat = fromFile(thisFileName)
    allIntensities.append( thisDat.intensities )
    allResponses.append( thisDat.data )

#plot each staircase
pylab.subplot(121)
colors = 'brgkcmbrgkcm'
lines, names = [],[]
for fileN, thisStair in enumerate(allIntensities):
    #lines.extend(pylab.plot(thisStair))
    #names = files[fileN]
    pylab.plot(thisStair, label=files[fileN])
#pylab.legend()

#get combined data
combinedInten, combinedResp, combinedN = \
             data.functionFromStaircase(allIntensities, allResponses, 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)
```

(continues on next page)

(continued from previous page)

```
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()
```

CHAPTER

SEVEN

RUNNING AND SHARING STUDIES ONLINE

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 PsychoPys 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 cant 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 wont.

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*

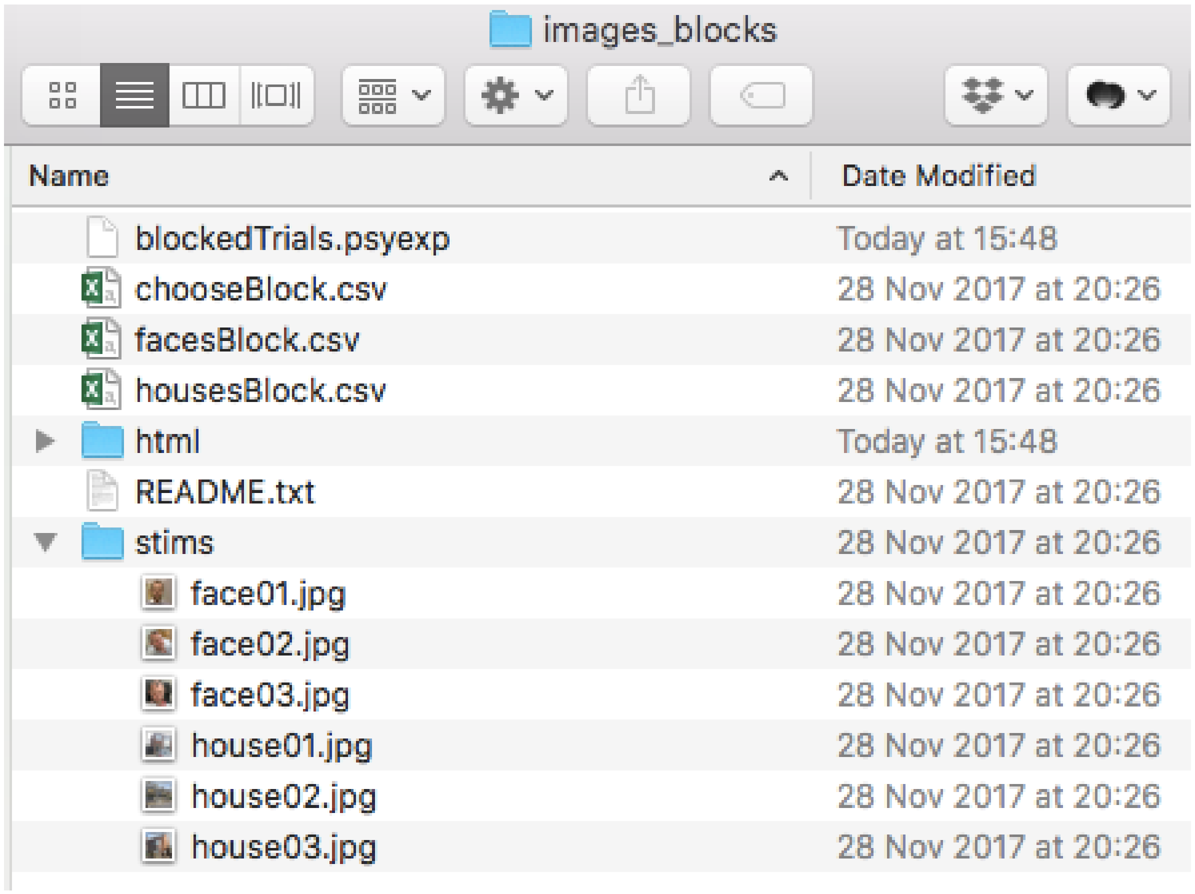
7.1.1 Check if your study is fully supported

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 youll know if we update/improve something and youll 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 youll be good to go! Usually you would have a folder structure something like this and sync that entire folder with pavlovia.org:



Name	Date Modified
blockedTrials.psyexp	Today at 15:48
chooseBlock.csv	28 Nov 2017 at 20:26
facesBlock.csv	28 Nov 2017 at 20:26
housesBlock.csv	28 Nov 2017 at 20:26
html	Today at 15:48
README.txt	28 Nov 2017 at 20:26
stims	28 Nov 2017 at 20:26
face01.jpg	28 Nov 2017 at 20:26
face02.jpg	28 Nov 2017 at 20:26
face03.jpg	28 Nov 2017 at 20:26
house01.jpg	28 Nov 2017 at 20:26
house02.jpg	28 Nov 2017 at 20:26
house03.jpg	28 Nov 2017 at 20:26

7.1.3 Export the HTML files

Once youve checked your settings you can simply go to *>File>Export HTML* from the Builder view with your experiment open.

That will generate all the necessary files (HTML and JS) that you need for your study

7.1.4 Uploading files to your own server

We really dont 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). Youll 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 Builders 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 starting point is that, as in Python, you need to be able to see the error messages (if there are any) being generated. To do this your browser you can hopefully show you the javascript console and you can see various logging messages and error messages there. If it doesnt make any sense to you then you could try sending it to the PsychoPy forum in the *Online* category.

7.1.6 Activate on Pavlovia

This is needed

7.1.7 Recruiting participants

Once youve 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 theyll 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 theyve 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 Academic'_**

7.1.8 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 wont be any psydat files though - that isnt possible from JavaScript).

You could just download the data folder or, if youve 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.

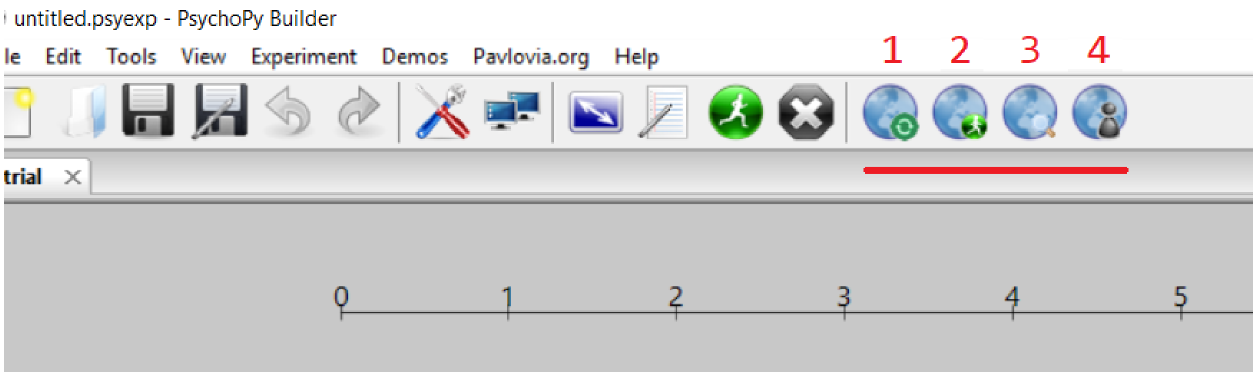
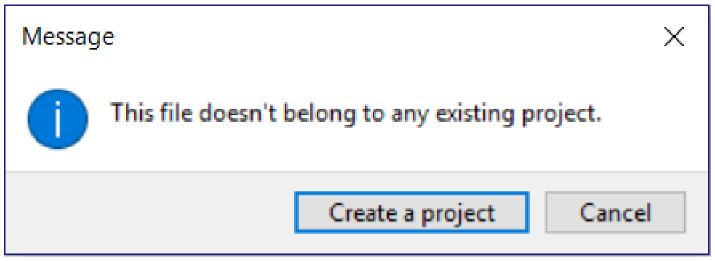

untitled.psyexp - PsychoPy Builder
le Edit Tools View Experiment Demos Pavlovia.org Help
1 2 3 4
trial
0 1 2 3 4 5

Figure 1. PsychoPy 3 Builder icons for building and running online studies

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.


Message

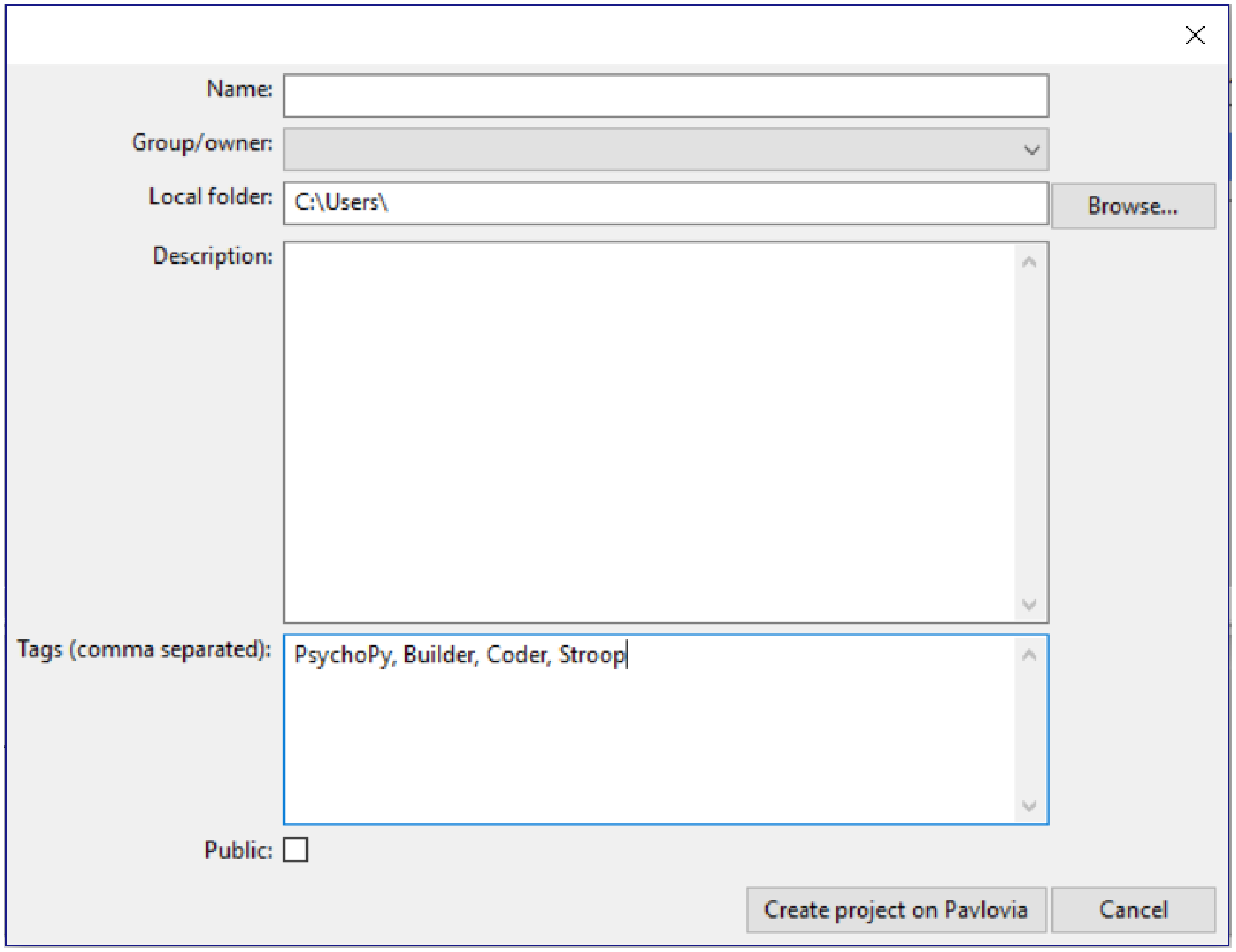
This file doesn't belong to any existing project.

Figure 2. The dialog that appears when an online project does not exist.

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

Use this window to add information to store your project on Pavlovia:



Name:

Group/owner:

Local folder: C:\Users\

Description:

Tags (comma separated): PsychoPy, Builder, Coder, Stroop

Public: ☐

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)

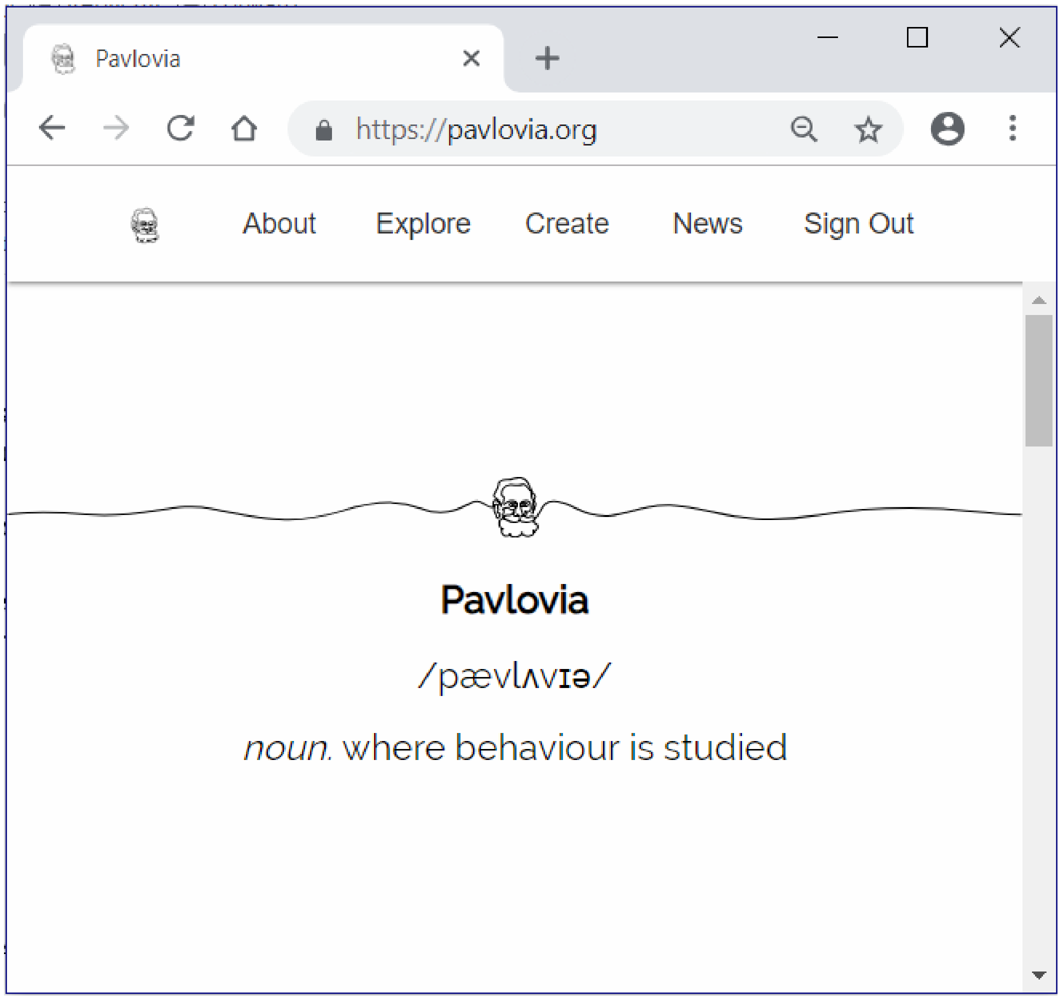
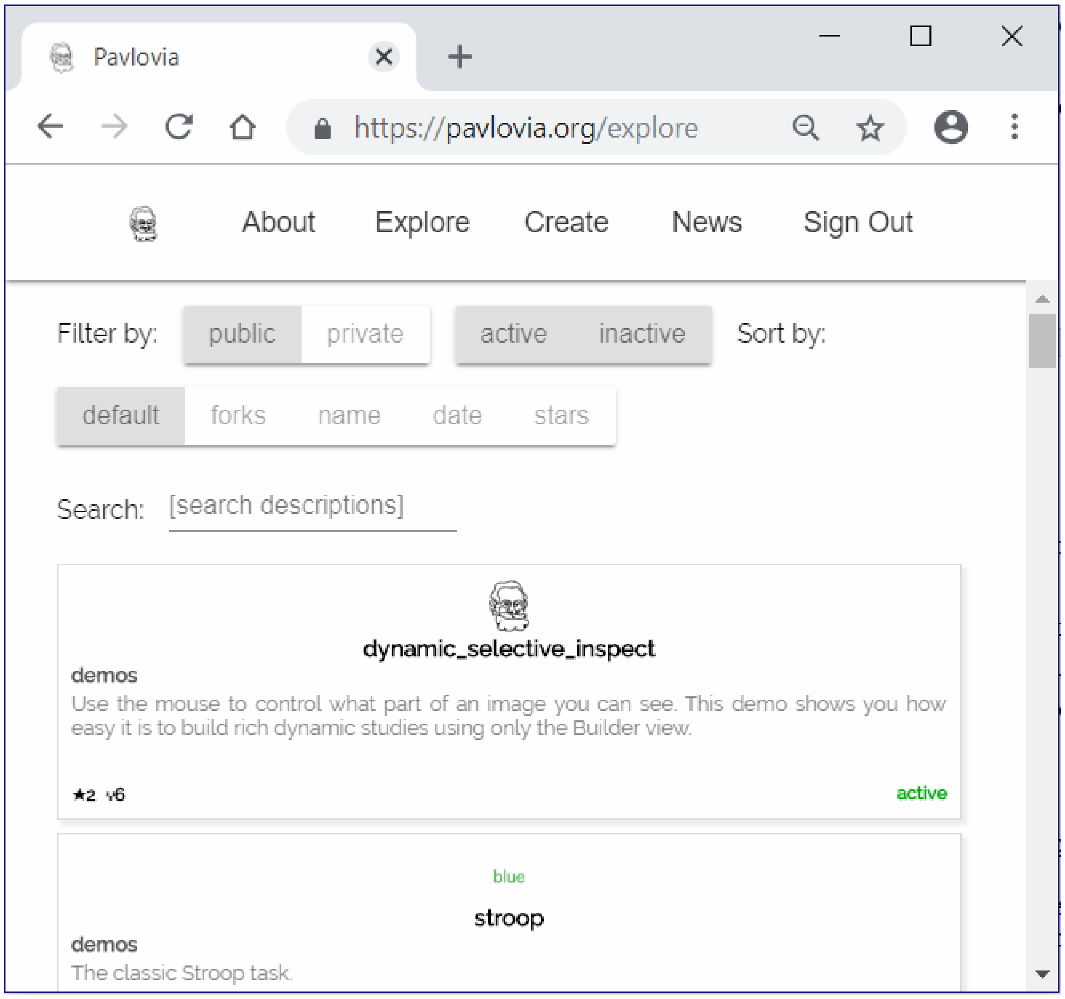


Figure 4. The *Pavlovia* home page

When exploring studies online, you are presented with a series of thumbnail images for all of the projects on *Pavlovia*. See Figure 5.



dynamic_selective_inspect

demos

Use the mouse to control what part of an image you can see. This demo shows you how easy it is to build rich dynamic studies using only the Builder view.

stroop

demos

The classic Stroop task.

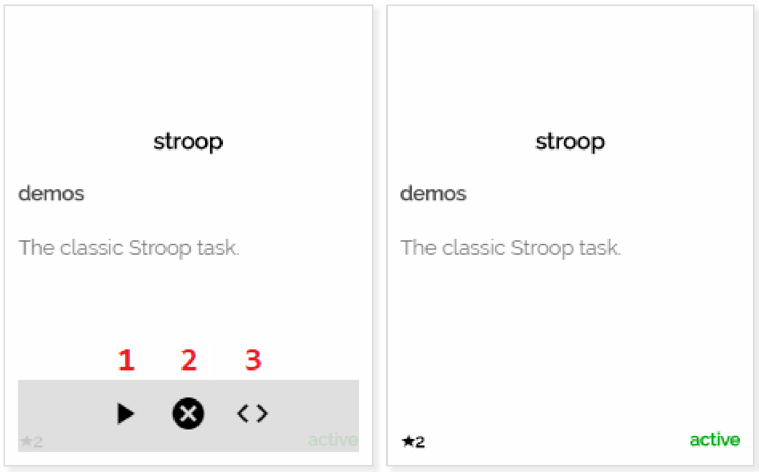
Figure 5. Exploring projects on *Pavlovia*

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.

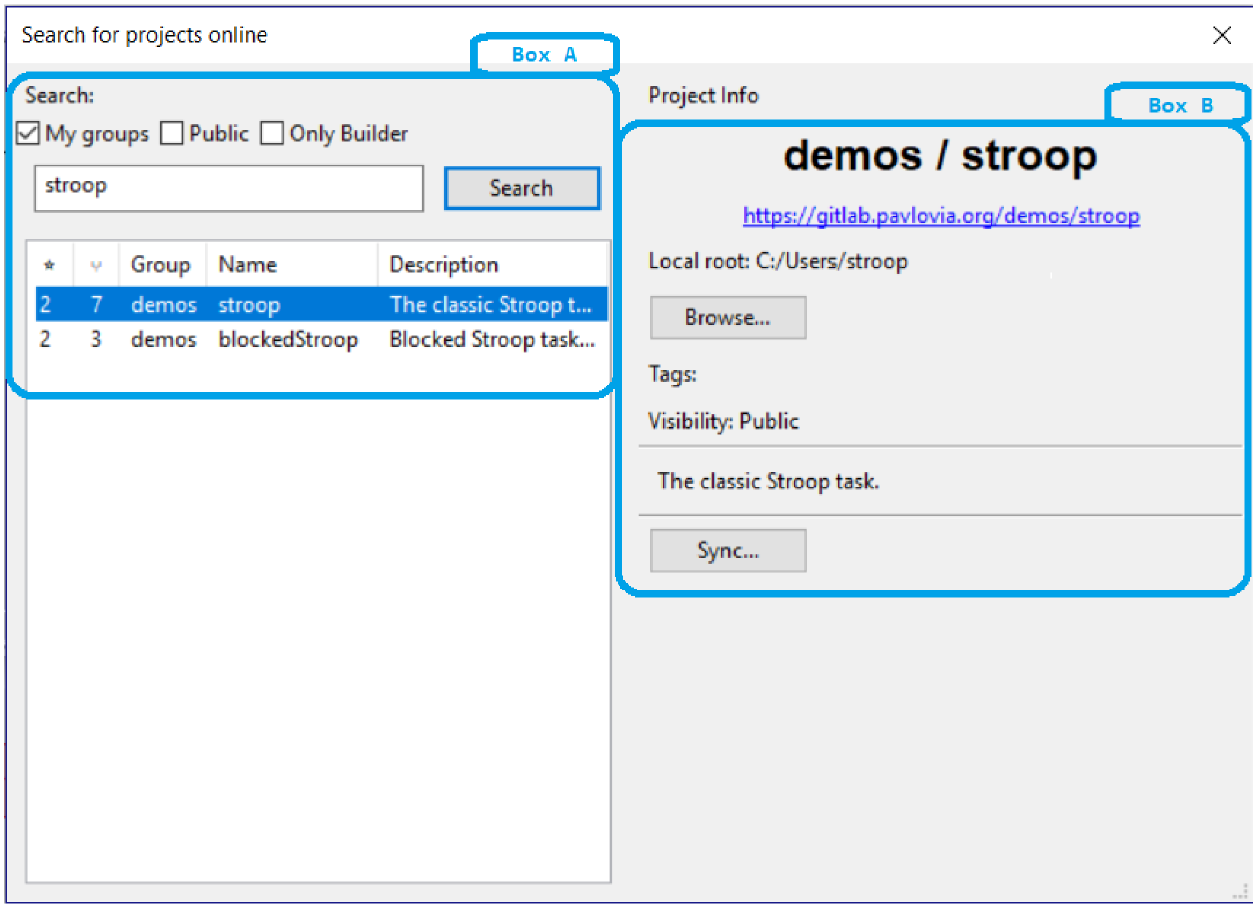
To fork and sync a project is to take your own copy of a project from *Pavlovia* (*fork*) and copy a version to your local desktop or laptop computer (*sync*). To fork a project, select the local folder to download the project using the Browse button, and then click Sync when you are ready - (see Fig 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*!

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.



☑ My groups ☐ Public ☐ Only Builder

Search: stroop

*	⑂	Group	Name	Description
2	7	demos	stroop	The classic Stroop t...
2	3	demos	blockedStroop	Blocked Stroop task...

demos / stroop

https://gitlab.pavlovia.org/demos/stroop

Local root: C:/Users/stroop

Tags:

Visibility: Public

The classic Stroop task.

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 youd 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 Routines 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 arent 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 hasnt 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 Academic

Prolific Academic 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 Academic 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%}}
```

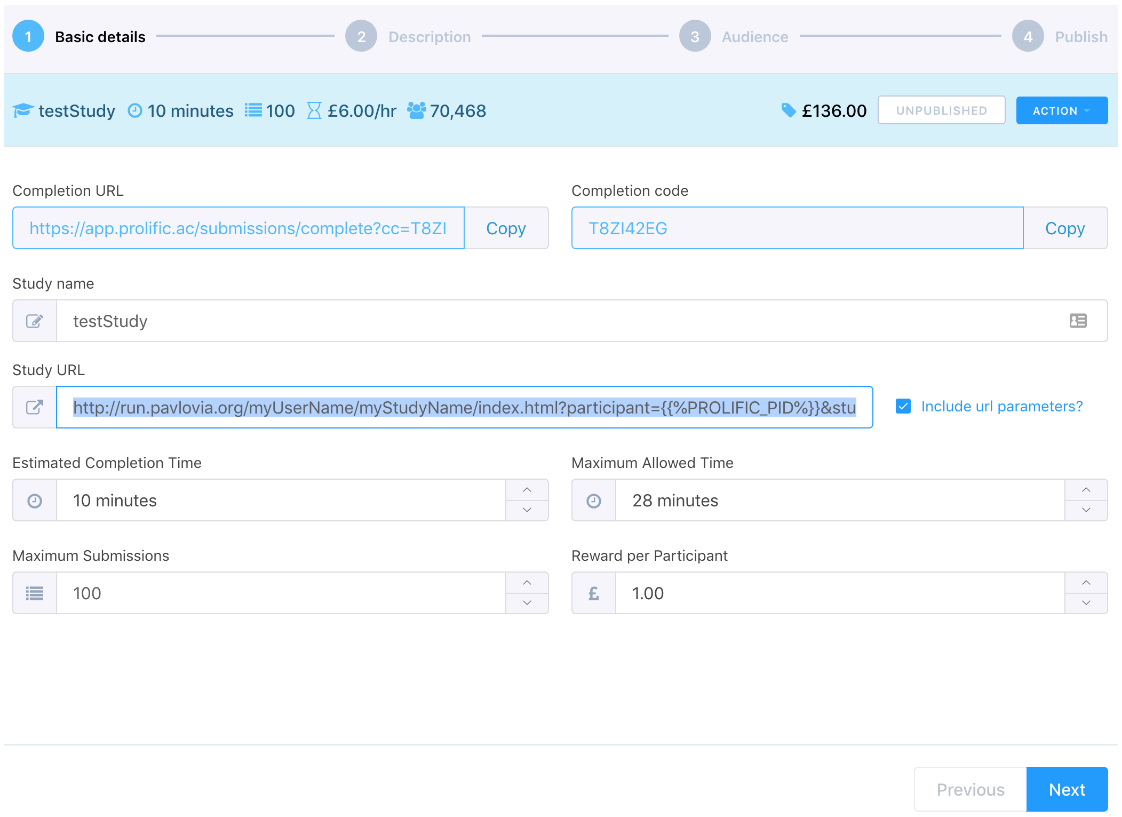
Example link to provide **to PsychoPy** as your completion URL (you will need to change your study ID number):

```
https://app.prolific.ac/submissions/complete?cc=T8ZI42EG
```

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 Academic

To recruit participants to your PsychoPy study you should see this screen while creating/modifying your study at https://prolific.ac:



Completion URL: https://app.prolific.ac/submissions/complete?cc=T8ZI

Completion code: T8ZI42EG

Study name: testStudy

Study URL: http://run.pavlovia.org/myUserName/myStudyName/index.html?participant={{%PROLIFIC_PID%}}&stu

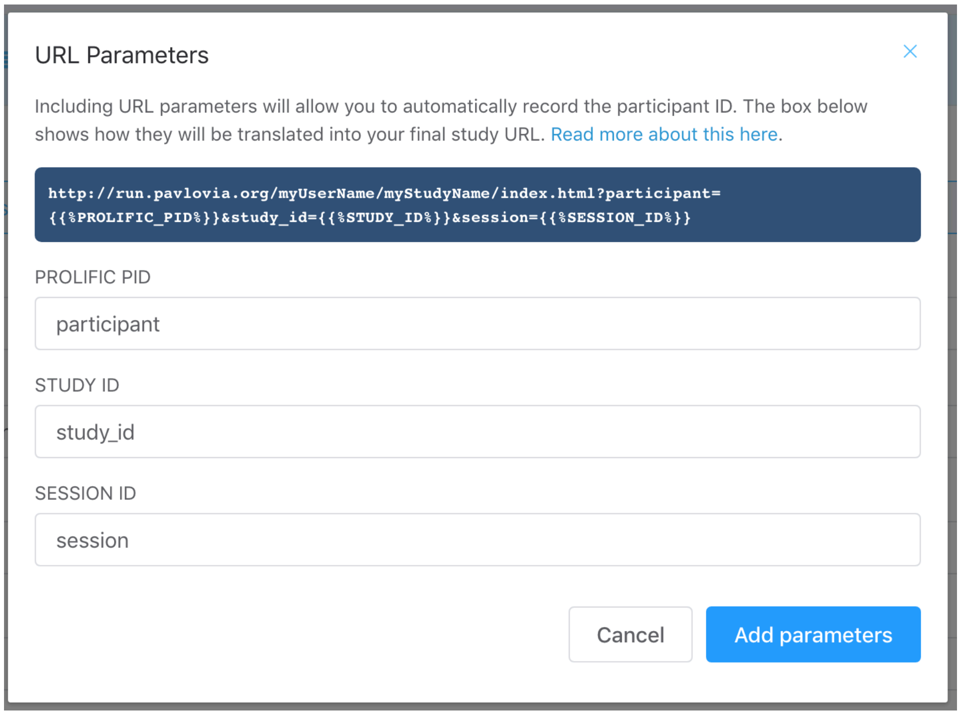
Estimated Completion Time: 10 minutes

Maximum Allowed Time: 28 minutes

Maximum Submissions: 100

Reward per Participant: £ 1.00

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. Ive 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):



URL Parameters

Including URL parameters will allow you to automatically record the participant ID. The box below shows how they will be translated into your final study URL. Read more about this here.

```
http://run.pavlovia.org/myUserName/myStudyName/index.html?participant=
{{%PROLIFIC_PID%}}&study_id={{%STUDY_ID%}}&session={{%SESSION_ID%}}
```

PROLIFIC PID: participant

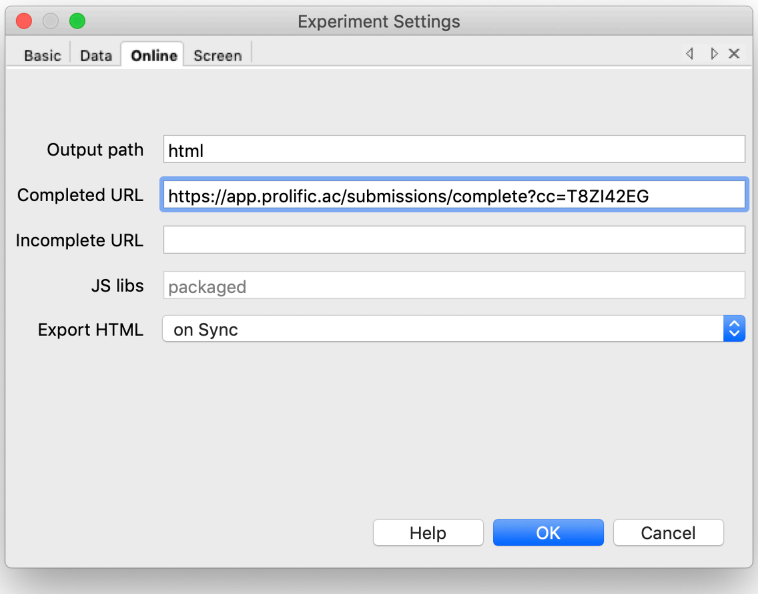
STUDY ID: study_id

SESSION ID: session

In each of the boxes in the figure above, you can see the name that Prolific gives to this value (e.g. *PROLIFIC_PID*) and the name that we want PsychoPy to use to store it (e.g. *participant*).

Setting the completion URL in PsychoPy

The last thing you need to do is copy the *Completion URL* from the main control panel above and paste that into the online tab for your PsychoPy *Experiment Settings* as below:



Output path: html
Completed URL: https://app.prolific.ac/submissions/complete?cc=T8ZI42EG
Incomplete URL:
JS libs: packaged
Export HTML: on Sync

7.4.3 Integrating with Amazons Mechanical Turk (MTurk)

Sorry, the documentation for this hasnt 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 truk (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 dont 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 <https://prolific.ac'_** instead.

7.4.4 Daisy-chaining with Qualtrics

Sorry, the documentation for this hasnt 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*:

```
https://run.pavlovia.org/yourUsername/yourStudyName/index.html?participant=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:

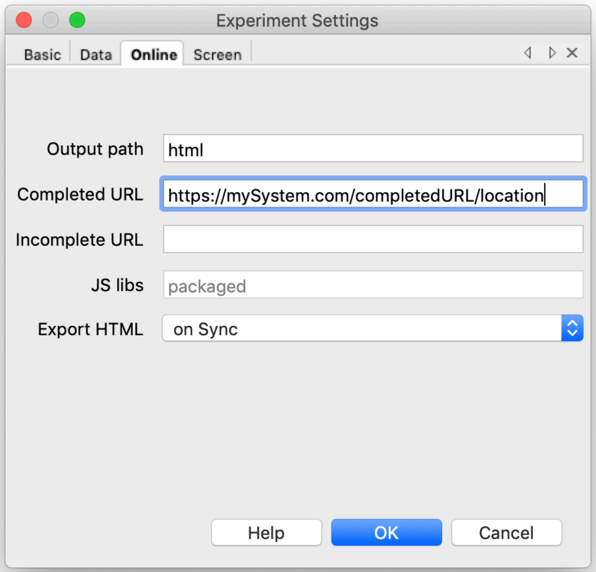
```
https://run.pavlovia.org/yourUsername/yourStudyName/index.html?participant=10101010&
↪group=A
```

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:



Output path: html
Completed URL: https://mySystem.com/completedURL/location
Incomplete URL:
JS libs: packaged
Export HTML: on Sync

but you should also be aware of the following:

7.5 Status of online options

The table below shows you the current state of play with capabilities of online studies.

Done		Not done (but could be)
	Inputs	
Keyboard		Mic
Mouse		Webcam
Rating Scales (Slider)		Free text (and similar)
		Multi-touch devices
	Stimuli	
Image		
Text		Gratings
Sounds		Apertures
Movies		Dots (RDKs)
	Data	
CSV files		XLSX files
MongoDB database		
Log files		
	Logic	
Loops (including nesting)		Staircases

Continued on next page

Table 1 – continued from previous page

Done		Not done (but could be)
Randomization		
Code Components		
Conditions files		
	Precision	
Frame-by-frame timing		
callOnFlip		
	External tools	
Synchronize with Pavlovia.org		
Sona Systems		
Mechanical Turk		
Prolific Academic		

Anything else we should add to the list above?

7.6 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 youve 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 arent 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 participants computer will be stored in your data file). When its 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.

7.6.1 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.

CHAPTER

EIGHT

REFERENCE MANUAL (API)

Contents:

8.1 `psychopy.core` - basic functions (clocks etc.)

Basic functions, including timing, rush (imported), quit

`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.`**`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.`**`wait`**`(`*secs*, *hogCPUperiod=0.2*`)`

Wait for a given time period.

If secs=10 and hogCPU=0.2 then for 9.8s pythons 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.

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 clocks 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.:

```
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:

```
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 zero. 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.

Dont 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.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 dont 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.

8.2 psychopy.visual - many visual stimuli

Window to display all stimuli below.

8.2.1 Aperture

8.2.2 BufferImageStim

Attributes

BufferImageStim
BufferImageStim.win
BufferImageStim.buffer
BufferImageStim.rect
BufferImageStim.stim
BufferImageStim.mask

Continued on next page

Table 1 – continued from previous page

`BufferImageStim.units`
`BufferImageStim.pos`
`BufferImageStim.ori`
`BufferImageStim.size`
`BufferImageStim.contrast`
`BufferImageStim.color`
`BufferImageStim.colorSpace`
`BufferImageStim.opacity`
`BufferImageStim.interpolate`
`BufferImageStim.name`
`BufferImageStim.autoLog`
`BufferImageStim.draw`
`BufferImageStim.autoDraw`

Details

8.2.3 `Circle`

8.2.4 `CustomMouse`

8.2.5 `DotStim`

8.2.6 `ElementArrayStim`

8.2.7 `GratingStim`

Attributes

`GratingStim`
`GratingStim.win`
`GratingStim.tex`
`GratingStim.mask`
`GratingStim.units`
`GratingStim.sf`
`GratingStim.pos`
`GratingStim.ori`
`GratingStim.size`
`GratingStim.contrast`
`GratingStim.color`
`GratingStim.colorSpace`
`GratingStim.opacity`
`GratingStim.interpolate`
`GratingStim.texRes`
`GratingStim.name`
`GratingStim.autoLog`
`GratingStim.draw`
`GratingStim.autoDraw`

Details

8.2.8 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 groups 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],]

8.2.9 `ImageStim`

As of PsychoPy version 1.79.00 *some* of the properties for this stimulus can be set using the syntax:

```
stim.pos = newPos
```

others need to be set with the older syntax:

```
stim.setImage(newImage)
```

Attributes

`ImageStim`
`ImageStim.win`
`ImageStim.setImage`
`ImageStim.setMask`

Continued on next page

Table 3 – continued from previous page

`ImageStim.units`
`ImageStim.pos`
`ImageStim.ori`
`ImageStim.size`
`ImageStim.contrast`
`ImageStim.color`
`ImageStim.colorSpace`
`ImageStim.opacity`
`ImageStim.interpolate`
`ImageStim.contains`
`ImageStim.overlaps`
`ImageStim.name`
`ImageStim.autoLog`
`ImageStim.draw`
`ImageStim.autoDraw`
`ImageStim.clearTextures`

Details

8.2.10 `Line`

8.2.11 `MovieStim`

Attributes

`MovieStim`
`MovieStim.win`
`MovieStim.units`
`MovieStim.pos`
`MovieStim.ori`
`MovieStim.size`
`MovieStim.opacity`
`MovieStim.name`
`MovieStim.autoLog`
`MovieStim.draw`
`MovieStim.autoDraw`
`MovieStim.loadMovie`
`MovieStim.play`
`MovieStim.seek`
`MovieStim.pause`
`MovieStim.stop`
`MovieStim.setFlipHoriz`
`MovieStim.setFlipVert`

Details

8.2.12 `NoiseStim`

Attributes

Details

8.2.13 `PatchStim` (deprecated)

8.2.14 `Polygon`

8.2.15 `RadialStim`

Attributes

`RadialStim`
`RadialStim.win`
`RadialStim.tex`
`RadialStim.mask`
`RadialStim.units`
`RadialStim.pos`
`RadialStim.ori`
`RadialStim.size`
`RadialStim.contrast`
`RadialStim.color`
`RadialStim.colorSpace`
`RadialStim.opacity`
`RadialStim.interpolate`
`RadialStim.setAngularCycles`
`RadialStim.setAngularPhase`
`RadialStim.setRadialCycles`
`RadialStim.setRadialPhase`
`RadialStim.name`
`RadialStim.autoLog`
`RadialStim.draw`
`RadialStim.autoDraw`
`RadialStim.clearTextures`

Details

8.2.16 `RatingScale`

8.2.17 `Rect`

8.2.18 `Rift`

Attributes

`Rift`
`Rift.productName`
`Rift.manufacturer`
`Rift.serialNumber`
`Rift.firmwareVersion`
`Rift.resolution`

Continued on next page

Table 6 – continued from previous page

`Rift.displayRefreshRate`
`Rift.trackingOriginType`
`Rift.getTrackingOriginType`
`Rift.setTrackinOrigin`
`Rift.recenterTrackingOrigin`
`Rift.shouldQuit`
`Rift.isVisible`
`Rift.isHmdMounted`
`Rift.isHmdPresent`
`Rift.shouldRecenter`
`Rift.setBuffer`
`Rift.absTime`
`Rift.viewMatrix`
`Rift.projectionMatrix`
`Rift.headLocked`
`Rift.pollControllers`
`Rift.flip`
`Rift.multiplyViewMatrixGL`
`Rift.multiplyProjectionMatrixGL`
`Rift.setRiftView`
`Rift.setDefaultView`
`Rift.controllerConnected`
`Rift.getConectedControllers`
`Rift.getThumbstickValues`
`Rift.getIndexTriggerValues`
`Rift.getHandTriggerValues`
`Rift.getButtons`
`Rift.getTouches`
`Rift.isIndexPointing`
`Rift.isThumbUp`
`Rift.raycastSphere`

Details

8.2.19 `EnvelopeGrating`

Attributes

Details

8.2.20 `ShapeStim`

Attributes

`ShapeStim`
`ShapeStim.win`
`ShapeStim.units`
`ShapeStim.vertices`
`ShapeStim.closeShape`

Continued on next page

Table 7 – continued from previous page

`ShapeStim.pos`
`ShapeStim.ori`
`ShapeStim.size`
`ShapeStim.contrast`
`ShapeStim.lineColor`
`ShapeStim.lineColorSpace`
`ShapeStim.fillColor`
`ShapeStim.fillColorSpace`
`ShapeStim.opacity`
`ShapeStim.interpolate`
`ShapeStim.name`
`ShapeStim.autoLog`
`ShapeStim.draw`
`ShapeStim.autoDraw`

Details

8.2.21 `SimpleImageStim`

8.2.22 `Slider`

Attributes

`Slider`
`Slider.getRating`
`Slider.getRT`
`Slider.markerPos`
`Slider.setReadOnly`
`Slider.contrast`
`Slider.style`
`Slider.getHistory`
`Slider.getMouseResponses`
`Slider.reset`

Details

8.2.23 `TextBox`

Attributes

`TextBox`

Note: The following *set______()* attributes all have equivalent *get______()* attributes:

`TextBox.setText`
`TextBox.setPosition`

Continued on next page

Table 10 – continued from previous page

`TextBox.setOri`
`TextBox.setHorzAlign`
`TextBox.setVertAlign`
`TextBox.setHorzJust`
`TextBox.setVertJust`
`TextBox.setFontColor`
`TextBox.setBorderColor`
`TextBox.setBackgroundColor`
`TextBox.setTextGridLineColor`
`TextBox.setTextGridLineWidth`
`TextBox.setInterpolated`
`TextBox.setOpacity`
`TextBox.setAutoLog`
`TextBox.draw`

Note: TextBox also provides the following read-only functions:

`TextBox.getSize`
`TextBox.getName`
`TextBox.getDisplayedText`
`TextBox.getValidStrokeWidths`
`TextBox.getLineSpacing`
`TextBox.getGlyphPositionForTextIndex`
`TextBox.getTextGridCellPlacement`

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

8.2.24 `TextStim`

8.2.25 `Window`

A class representing a window for displaying one or more stimuli.

8.2.26 `psychopy.visual.windowframepack` - Pack multiple monochrome images into RGB frame

`ProjectorFramePacker`

8.2.27 `psychopy.visual.windowwarp` - warping to spherical, cylindrical, or other projections

`Warper`

Windows and 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

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

`psychopy.clock.`**`getTime`**`()`

Copyright (c) 2018 Mario Kleiner. Licensed under MIT license.

For detailed help on a subfunction SUBFUNCTIONNAME, type GetSecs(SUBFUNCTIONNAME?) ie. 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.`**`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.`**`wait`**`(`*secs*, *hogCPUperiod=0.2*`)`

Wait for a given time period.

If secs=10 and hogCPU=0.2 then for 9.8s pythons 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.

`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 clocks 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.:

```
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.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 zero. 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.

Dont 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
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 dont 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.

8.4 `psychopy.data` - functions for storing/saving/analysing data

8.4.1 `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`] Containining 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.:

```
# 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=None*, *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: 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.

encoding: The encoding to use when saving a the file. Defaults to *utf-8-sig*.

fileCollisionMethod: Collision method passed to `handleFileCollision()`

sortColumns: will sort columns alphabetically by header name if True

8.4.2 `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:

```
from psychopy.tools.filetools import fromFile
dat = fromFile(path)
```

Then youll find that *dat* has the following attributes that

Parameters

trialList: a simple list (or flat array) of dictionaries specifying conditions. This can be imported from an excel/csv file using *`importConditions()`*

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 (or more strictly, a *DataHandler* sub- class 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

`_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 doesnt 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 0] [1 1 1 1 1] [2 2 2 2 2] [3 3 3 3 3] [4 4 4 4 4] [5 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

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

`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:

```
trials = data.TrialHandler(.......)
for eachTrial in trials:  # automatically stops when done
    # do stuff
```

or:

```
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 copys .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 trialHandlers 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:

```
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:

```
trials._exp = myExperiment
```

because it needs to be performed using the *weakref* module.

8.4.3 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.

stepType: specifies whether each step will be a jump of the given size in db, log or lin units (lin means this intensity will be added/subtracted)

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.

`_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

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*.

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
```

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 `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 `handleFileCollision()`. Can be either of *rename*, *overwrite*, or *fail*.

Notes

Currently, a copy of the object is created, and the copys .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 necess 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 delimitter 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.

8.4.4 MultiStairHandler

class psychopy.data.**MultiStairHandler**(*stairType='simple'*, *method='random'*, *conditions=None*, *nTrials=50*, *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 the 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 custom 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 or sequential The stairs are shuffled in each repeat but not randomised more than that (so you cant have 3 repeats of the same staircase in a row unless its the only one still running)

conditions: a list of dictionaries specifying conditions Can be used to control parameters for the different staicases. 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 hasnt also met its minimal reversals. See `StairHandler`

Example usage:

```
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
```

(continues on next page)

(continued from previous page)

```
# save data as multiple formats
stairs.saveDataAsExcel(fileName)  # easy to browse
stairs.saveAsPickle(fileName)  # contains more info
```

_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 wont 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:

```
staircase = data.MultiStairHandler(.......)
for eachTrial in staircase:  # automatically stops when done
    # do stuff here for the trial
```

or:

```
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-delimitted, , for csv files)

matrixOnly: True/False If True, prevents the output of the *extraInfo* provided at initialisation.

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 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 `handleFileCollision()`. Can be either of *rename*, *overwrite*, or *fail*.

Notes

Currently, a copy of the object is created, and the copys .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 fileName 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 delimitter 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.

8.4.5 QuestHandler

class psychopy.data.**QuestHandler**(*startVal*, *startValSd*, *pThreshold=0.82*, *nTrials=None*, *stopInterval=None*, *method='quantile'*, *beta=3.5*, *delta=0.01*, *gamma=0.5*, *grain=0.01*, *range=None*, *extraInfo=None*, *minVal=None*, *maxVal=None*, *staircase=None*, *originPath=None*, *name=''*, *autoLog=True*, ***kwargs*)

Class that implements the Quest algorithm for quick measurement of psychophysical thresholds.

Uses Andrew Straws QUEST, which is a Python port of Denis Pellis Matlab code.

Measures threshold using a Weibull psychometric function. Currently, it is not possible to use a different psychometric function.

Threshold t is measured on an abstract intensity scale, which usually corresponds to log10 contrast.

The Weibull psychometric function:

_e = -10**(beta * (x2 + xThreshold)) p2 = delta * gamma + (1-delta) * (1 - (1 - gamma) * exp(_e))

Example:

```
# 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)

(continued from previous 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 its 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.

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 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.

`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 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 wasnt 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:

```
staircase = data.QuestHandler(.......)
for eachTrial in staircase:  # automatically stops when done
    # do stuff
```

or:

```
staircase = data.QuestHandler(.......)
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='\t', 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(*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 `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 `handleFileCollision()`. Can be either of *rename*, *overwrite*, or *fail*.

Notes

Currently, a copy of the object is created, and the copys .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 necess 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 delimitter 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*.

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:

```
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

8.4.6 FitWeibull

class psychopy.data.**FitWeibull**(*xx*, *yy*, *sems=1.0*, *guess=None*, *display=1*, *expectedMin=0.5*, *optimize_kws=None*)

Fit a Weibull function (either 2AFC or YN) of the form:

```
y = chance + (1.0-chance)*(1-exp( -(xx/alpha)**(beta) ))
```

and with inverse:

```
x = alpha * (-log((1.0-y)/(1-chance)))**(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.

8.4.7 `FitLogistic`

class psychopy.data.**FitLogistic**(*xx*, *yy*, *sems=1.0*, *guess=None*, *display=1*, *expectedMin=0.5*, *optimize_kws=None*)

Fit a Logistic function (either 2AFC or YN) of the form:

```
y = chance + (1-chance)/(1+exp((PSE-xx)*JND))
```

and with inverse:

```
x = PSE - log((1-chance)/(yy-chance) - 1)/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]`)

_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.

8.4.8 `FitNakaRushton`

class psychopy.data.**FitNakaRushton**(*xx*, *yy*, *sems=1.0*, *guess=None*, *display=1*, *expectedMin=0.5*, *optimize_kws=None*)

Fit a Naka-Rushton function of the form:

```
yy = rMin + (rMax-rMin) * xx**n/(xx**n+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.

_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.

8.4.9 `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 = chance + (1-chance)*((special.erf((xx-xShift)/(sqrt(2)*sd))+1)*0.5)
```

and with inverse:

```
x = xShift+sqrt(2)*sd*(erfinv(((yy-chance)/(1-chance)-.5)*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.

`inverse``(`*yy*, *params=None*`)`
Evaluate yy for the current parameters of the model, or for arbitrary params if these are given.

8.4.10 `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`* *trialTypes* 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 (doesnt 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 0-8

8.4.11 `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:

```
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

8.4.12 `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

8.5 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

8.6 `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 neednt 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]*:

```
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 doesnt 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 doesnt 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

8.7 `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*

8.8 `psychopy.gui` - create dialogue boxes

8.8.1 `DlgFromDict`

class `psychopy.gui.`**`DlgFromDict`**(*dictionary*, *title=''*, *fixed=None*, *order=None*, *tip=None*, *screen=-1*, *sortKeys=True*, *copyDict=False*, *labels=None*, *show=True*, ***kwargs*)

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 ‘‘dictionaryt’t’. Not all keys in ‘‘dictionaryt’t’ 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 ‘‘dictionaryt’t’. If not all keys in ‘‘dictionaryt’t’ are contained in ‘‘ordert’t’, 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 ‘‘dictionaryt’t’. Not all keys in ‘‘dictionaryt’t’ 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.`** –
- **`::`** –

  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)
- **`the code above, the contents of info will be updated to the values`** (*`In`*) –
- **`by the dialogue box.`** (*`returned`*) –
- **`the user cancels (rather than pressing OK),`** (*`If`*) –
- **`the dictionary remains unchanged. If you want to check whether`** (*`then`*) –
- **`user hit OK, then check whether DlgFromDict.OK equals`** (*`the`*) –
- **`or False`** (*`True`*) –
- **`GUI.py for a usage demo, including order and tip (tooltip)`** (*`See`*) –

show()
Display the dialog.

8.8.2 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

```
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*)
Adds 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="*)
Adds 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

8.8.3 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.

8.8.4 `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.

8.9 `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:

8.9.1 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 Psychtoolbox kbQueue series of functions using the same compiled C code (available in python-psychtoolbox lib).

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 Psychtoolbox

Example usage

```
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'], waitDuration=True)
if 'quit' in keys:
    core.quit()
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 legacy systems.

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 well 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 wont 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) .tDown: 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:

```
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 cant make use of it.

Returns USB Info including with name, manufacturer, id, etc for each device

Return type A list of dicts

8.9.2 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:

```
    import 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:
```

(continues on next page)

(continued from previous page)

```
        dev.poll_for_response()
        if dev.response_queue_size() > 0:
            response = dev.get_next_response()
            # do something with the response
```

Useful functions

pyxid.**get_xid_device**(*device_number*)
returns device at a given index.

Raises ValueError if the device at the passed in index doesnt exist.

pyxid.**get_xid_devices**()
Returns a list of all Xid devices connected to your computer.

Device classes

class pyxid.**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 youd 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 devices 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 miliseconds when activate_line is called

8.9.3 Cambridge Research Systems Ltd.

For stimulus display

`BitsPlusPlus`

Control a CRS Bits# device. See typical usage in the class summary (and in the menu demos>hardware>BitsBox of PsychoPys Coder view).

Important: See note on *BitsPlusPlusIdentityLUT*

Attributes

`BitsPlusPlus`
`BitsPlusPlus.mode`
`BitsPlusPlus.setContrast`
`BitsPlusPlus.setGamma`
`BitsPlusPlus.setLUT`

Details

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:

```
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 PsychoPys Coder view).

Attributes

`BitsSharp`
`BitsSharp.mode`
`BitsSharp.isAwake`
`BitsSharp.getInfo`
`BitsSharp.checkConfig`
`BitsSharp.gammaCorrectFile`
`BitsSharp.temporalDithering`
`BitsSharp.monitorEDID`
`BitsSharp.beep`
`BitsSharp.getVideoLine`
`BitsSharp.start`
`BitsSharp.stop`

Direct communications with the serial port:

`BitsSharp.sendMessage`
`BitsSharp.getResponse`

Control the CLUT (Bits++ mode only):

`BitsSharp.setContrast`
`BitsSharp.setGamma`
`BitsSharp.setLUT`

Details

For display calibration

`ColorCAL`

Attributes

`ColorCAL`

Details

8.9.4 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

8.9.5 Launch an fMRI experiment: Test or Scan

8.9.6 fORP response box

fORP fibre optic (MR-compatible) response devices by CurrentDesigns: http://www.curdes.com/ This class is only useful when the fORP is connected via the serial port.

If youre 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()`.

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

classmethod **`_decodePress`**(*pressCode*)

Returns a list of buttons and whether theyre pressed, given a character code.

pressCode **:** A number with a bit set for every button currently pressed. Will be between 0 and 31.

`_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.

`clearBuffer`()

Empty the input buffer of all characters

`clearStatus`()

Resets the pressed statuses, so getEvents will return pressed buttons, even if they were already pressed in the last call.

`getEvents`(*returnRaw=False*, *asKeys=False*, *allowRepeats=False*)

Returns a list of unique events (one event per button pressed) and also stores a copy of the full list of events since last 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

8.9.7 iolab

This provides a basic ButtonBox class, and imports the ioLab python library.

class psychopy.hardware.iolab.**ButtonBox**

PsychoPys 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:

```
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.

_getTime(*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).

8.9.8 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)

8.9.9 labjacks (USB I/O devices)

PsychoPy provides an interface to the labjack U3 class with a couple of minor additions.

This is accessible by:

```
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(firstFound = False, 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

8.9.10 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:

```
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 doesnt respond first time how many attempts should be made? If youre 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:

```
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 youre 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 communating. 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 [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 theres 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 arent 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

8.9.11 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:

```
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:

```
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:

```
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 youre 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]

otherwise: The output will be the raw string from the PR650 and should then be passed to `.parseSpectrumOutput()`. Its 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()`. Its 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:

```
from psychopy import hardware
phot = hardware.findPhotometer()
print(phot.getLum())
```

Troubleshooting If the device isnt 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.

`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()`. Its 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

8.9.12 pylink (SR research)

For now the SR Research pylink module is packaged with the Standalone flavours of PsychoPy and can be imported with:

```
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:

https://www.sr-support.com/forums/forumdisplay.php?f=17

for pylink documentation see:

https://www.sr-support.com/forums/showthread.php?t=14

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 EyeLinkő 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 widthiheight - original image heightpixels - 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 positionys - crop y positionwidth - crop widthheight - crop heightfname - file name to savepath - path to savesvoptions - 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.`**`enableUTF8EyeLinkMessages`**`()`
If enabled, the utf8 encoding is used on message texts.

`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.`**`getDisplayInformation`**`()`
Returns the display configuration. Parameters None Return Value Instance of DisplayInfo class. The width, height, bits, and refresh rate of the display can be accessed from the returned value. For example. display = getDisplayInformation() print display.width, display.height, display.bits, display.refresh

`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 calibration, camera image display, and drift correction, the display background should match the brightness of the experimental 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 threeinteger (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 delectability) 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);

8.9.13 `psychopy.hardware.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:

CA Andersen, L Alfine, K Ohla, & R Hochenberger (2018): A new gustometer: Template for the construction of a portable and

modular stimulator for taste and lingual touch.

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 aspiation 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 didnt yield a valid response
- None if there were not even any valid ports (suggesting a driver not being installed)

e.g.:

```
# 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())
```

8.10 `psychopy.info` - functions for getting information about the system

8.11 `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.

8.11.1 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

`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 devices 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:

kwarg Name	Value Type	Description
experiment_code	str, <= 24 char	If experiment_code is provided, an ioHub HDF5 file will be created for the session.
session_code	str, <= 24 char	When specified, used as the name of the ioHub HDF5 file created for the session.
experiment_info	dict	Can be used to save the following experiment metadata fields:
		• code: str, <= 24 char • title: str, <= 48 char • description: str, < 256 char • version: str, <= 6 char
session_info	dict	Can be used to save the following session metadata fields:
		• code: str, <= 24 char • name: str, <= 48 char • comments: str, < 256 char • user_variables: dict
datastore_name	str	Used to provide an ioHub HDF5 file name different than the session_code.
psychopy_monitor_name	str	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.
iohub_config_name	str	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.
iohub.device.path Multiple Devices can be specified using separate kwarg entries.	dict	Add an ioHub Device by using the device class path as the key, and the devices configuration in a dict value.

Examples

A. Wait for the q key to be pressed:

```
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(['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 launchHubServer 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.

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:

```
# get the Mouse device, named mouse
mouse=hub.devices.mouse
mouse_position = mouse.getPosition()

print 'mouse position: ', mouse_position

# Returns something like:
# >> mouse position:  [-211.0, 371.0]
```

getDevice(*deviceName*)

Returns the ioHubDeviceView that has a matching name (based on the device : name property specified in the ioHub_config.yaml for the experiment). If no device with the given name is found, None is returned. Example, accessing a Keyboard device that was named kb

```
keyboard = self.getDevice('kb')
kb_events= keyboard.getEvent()
```

This is the same as using the natural naming approach supported by the .devices attribute, i.e:

```
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 events 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 Servers 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:

```
# 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 TriaHandler row / record to the iohub data file for future data analysis use.

Parameters **cv_row** –

Returns None

getTime()

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.

getPriority()

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 ids (from 0 to getSystemProcessorCount()-1). A Processs Affinity determines which CPUs 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 Processs Affinity determines which CPUs 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 ids; 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.

8.11.2 Supported Devices

psychopy.iohub 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 psychopy.iohub.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:

```
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):

```
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 )
```

(continues on next page)

(continued from previous page)

```
presses = keyboard.waitForPresses(maxWait=5.0)

print(presses)

# Stop the ioHub Server
io.quit()
```

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 KeyboardEvents 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:

```
# Read the reporting state of the keyboard.
is_reporting_keyboard_event = keyboard.reporting

# Stop the keyboard from reporting any new events.
keyboard.reporting = False
```

property state

time values. The key is taken from the originating press event .key field. The time value is time of the key press event.

Note that any pressed, or active, modifier keys are included in the return value.

Returns dict

Type 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

class psychopy.iohub.devices.mouse.**Mouse**(**args*, ***kwargs*)

The Mouse class and related events represent a standard computer mouse device and the events a standard mouse can produce.

Mouse position data is mapped to the coordinate space defined in the ioHub configuration file for the Display.

Examples

A. Print all mouse events received for 5 seconds:

```
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()

mouse = io.devices.mouse

# Check for and print any Mouse events received for 5 seconds.
stime = getTime()
while getTime()-stime < 5.0:
    for e in mouse.getEvents():
        print(e)

# Stop the ioHub Server
io.quit()
```

B. Print current mouse position for 5 seconds:

```
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()

mouse = io.devices.mouse

# Check for and print any Mouse events received for 5 seconds.
stime = getTime()
while getTime()-stime < 5.0:
    print(mouse.getPosition())

# 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 devices getEvents(), or clearEvents() methods.

Note that calling clearEvents() at the device level only clears the given devices event buffer. The ioHub Processs Global Event Buffer is unchanged.

Parameters **`None`** –

Returns None

`getEvents`(**args*, ***kwargs*)

Retrieve any DeviceEvents that have occurred since the last call to the devices getEvents() or clearEvents() methods.

Note that calling getEvents() at a device level does not change the Global Event Buffers 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 IDs 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)

`getPosition`(*return_display_index=False*)

Returns the current position of the ioHub Mouse Device. Mouse Position is in display coordinate units, with 0,0 being the center of the screen.

Parameters

- **`return_display_index`** – If True, the display index that is
- **`with the mouse position will also be returned.`** (*`associated`*) –

Returns If return_display_index is false (default), return (x, y) position of mouse. If return_display_index is True return ((x,y), display_index).

Return type tuple

`getPositionAndDelta`(*return_display_index=False*)

Returns a tuple of tuples, being the current position of the ioHub Mouse Device as an (x,y) tuple, and the amount the mouse position changed the last time it was updated (dx,dy). Mouse Position and Delta are in display coordinate units.

Parameters **`None`** –

Returns ((x,y), (dx,dy)) position of mouse, change in mouse position, both in Display coordinate space.

Return type tuple

`getScroll`()

Returns the current vertical scroll value for the mouse. The vertical scroll value changes when the scroll wheel on a mouse is moved up or down. The vertical scroll value is in an arbitrary value space ranging for -32648 to +32648. Scroll position is initialize to 0 when the experiment starts.

Parameters **`None`** –

Returns current vertical scroll value.

Return type int

setPosition(*pos*, *display_index=None*)
Sets the current position of the ioHub Mouse Device. Mouse position (pos) should be specified in Display coordinate units, with 0,0 being the center of the screen.

Parameters

- **pos** (*(x,y) list or tuple*) – The position, in Display
- **space, to set the mouse position too.** (*coordinate*) –
- **display_index** (*int*) – Optional arguement giving the display index
- **set the mouse pos within. If None, the active ioHub Display** (*to*) –
- **index is used.** (*device*) –

Returns new (x,y) position of mouse in Display coordinate space.

Return type tuple

setScroll(*s*)
Sets the current vertical scroll value for the mouse. The vertical scroll value changes when the scroll wheel on a mouse is moved up or down. The vertical scroll value is in an arbitrary value space ranging for -32648 to +32648. Scroll position is initialize to 0 when the experiment starts. This method allows you to change the scroll value to anywhere in the valid value range.

Args (int): The scroll position you want to set the vertical scroll to. Should be a number between -32648 to +32648.

Returns current vertical scroll value.

Return type int

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.

class psychopy.iohub.devices.mouse.**MouseMoveEvent**(*object*)
MouseMoveEvents occur when the mouse position changes. Mouse position is mapped to the coordinate space defined in the ioHub configuration file for the Display.

Event Type ID: EventConstants.MOUSE_MOVE

Event Type String: MOUSE_MOVE

time
time of event, in sec.msec format, using psychopy timebase.

x_position
x position of the Mouse when the event occurred; in display coordinate space.

y_position
y position of the Mouse when the event occurred; in display coordinate space.

scroll_x
Horizontal scroll wheel absolute position when the event occurred. macOS only. Always 0 on other OSs.

scroll_y
Vertical scroll wheel absolute position when the event occurred.

modifiers
List of the modifiers that were active when the mouse event occurred, provided as a list of the modifier constant labels.

display_id
The id of the display that the mouse was over when the event occurred. Only supported on Windows at this time. Always 0 on other OSs.

window_id
Window handle reference that the mouse was over when the event occurred (window does not need to have focus).

event_id
The id assigned to the device event. Every event generated by iohub during an experiment session is assigned a unique id, starting from 0.

type
The type id for the event. This is used to create DeviceEvent objects or dictionary representations of an event based on the data from an event list.

class `psychopy.iohub.devices.mouse.`**`MouseDragEvent`**(*object*)
MouseDragEvents occur when the mouse position changes and at least one mouse button is pressed. Mouse position is mapped to the coordinate space defined in the ioHub configuration file for the Display.

Event Type ID: EventConstants.MOUSE_DRAG

Event Type String: MOUSE_DRAG

time
x position of the Mouse when the event occurred; in display coordinate space.

x_position
x position of the Mouse when the event occurred; in display coordinate space.

button_state
1 if a mouse button press caused the event, 0 if button was released.

button_id
Index of the mouse button that caused the event. MouseConstants.MOUSE_BUTTON_LEFT, MouseConstants.MOUSE_BUTTON_RIGHT and MouseConstants.MOUSE_BUTTON_MIDDLE are int constants representing left, right, and middle buttons of the mouse.

pressed_buttons
All currently pressed button ids logically ORed together.

y_position
y position of the Mouse when the event occurred; in display coordinate space.

scroll_x
Horizontal scroll wheel absolute position when the event occurred. macOS only. Always 0 on other OSs.

scroll_y
Vertical scroll wheel absolute position when the event occurred.

modifiers
List of the modifiers that were active when the mouse event occurred, provided as a list of the modifier constant labels.

display_id
The id of the display that the mouse was over when the event occurred. Only supported on Windows at this time. Always 0 on other OSs.

window_id
Window handle reference that the mouse was over when the event occurred (window does not need to have focus).

event_id
The id assigned to the device event. Every event generated by iohub during an experiment session is assigned a unique id, starting from 0.

type
The type id for the event. This is used to create DeviceEvent objects or dictionary representations of an event based on the data from an event list.

class psychopy.iohub.devices.mouse.**MouseButtonPressEvent**(*object*)
MouseButtonPressEvents are created when a button on the mouse is pressed. The button_state of the event will equal MouseConstants.MOUSE_BUTTON_STATE_PRESSED, and the button that was pressed (button_id) will be MouseConstants.MOUSE_BUTTON_LEFT, MouseConstants.MOUSE_BUTTON_RIGHT, or MouseConstants.MOUSE_BUTTON_MIDDLE, assuming you have a 3 button mouse.

To get the current state of all three buttons on the Mouse Device, the pressed_buttons attribute can be read, which tracks the state of all three mouse buttons as an int that is equal to the sum of any pressed button ids (MouseConstants.MOUSE_BUTTON_LEFT, MouseConstants.MOUSE_BUTTON_RIGHT, or MouseConstants.MOUSE_BUTTON_MIDDLE).

To tell if a given mouse button was depressed when the event occurred, regardless of which button triggered the event, you can use the following:

```
isButtonPressed = event.pressed_buttons &
MouseConstants.MOUSE_BUTTON_xxx == MouseConstants.MOUSE_BUTTON_xxx
```

where xxx is LEFT, RIGHT, or MIDDLE.

For example, if at the time of the event both the left and right mouse buttons were in a pressed state:

```
buttonToCheck=MouseConstants.MOUSE_BUTTON_RIGHT
isButtonPressed = event.pressed_buttons & buttonToCheck ==
buttonToCheck

print isButtonPressed

>> True

buttonToCheck=MouseConstants.MOUSE_BUTTON_LEFT
isButtonPressed = event.pressed_buttons & buttonToCheck ==
buttonToCheck

print isButtonPressed

>> True

buttonToCheck=MouseConstants.MOUSE_BUTTON_MIDDLE
isButtonPressed = event.pressed_buttons & buttonToCheck ==
buttonToCheck

print isButtonPressed
```

(continues on next page)

(continued from previous page)

```
>> False
```

Event Type ID: EventConstants.MOUSE_BUTTON_PRESS

Event Type String: MOUSE_BUTTON_PRESS

:rtype : MouseButtonEvent :param args: :param kwargs:

time
x position of the Mouse when the event occurred; in display coordinate space.

x_position
x position of the Mouse when the event occurred; in display coordinate space.

button_state
1 if a mouse button press caused the event, 0 if button was released.

button_id
Index of the mouse button that caused the event. MouseConstants.MOUSE_BUTTON_LEFT, MouseConstants.MOUSE_BUTTON_RIGHT and MouseConstants.MOUSE_BUTTON_MIDDLE are int constants representing left, right, and middle buttons of the mouse.

pressed_buttons
All currently pressed button ids logically ORed together.

y_position
y position of the Mouse when the event occurred; in display coordinate space.

scroll_x
Horizontal scroll wheel absolute position when the event occurred. macOS only. Always 0 on other OSs.

scroll_y
Vertical scroll wheel absolute position when the event occurred.

modifiers
List of the modifiers that were active when the mouse event occurred, provided as a list of the modifier constant labels.

display_id
The id of the display that the mouse was over when the event occurred. Only supported on Windows at this time. Always 0 on other OSs.

window_id
Window handle reference that the mouse was over when the event occurred (window does not need to have focus).

event_id
The id assigned to the device event. Every event generated by iohub during an experiment session is assigned a unique id, starting from 0.

type
The type id for the event. This is used to create DeviceEvent objects or dictionary representations of an event based on the data from an event list.

class psychopy.iohub.devices.mouse.**MouseButtonReleaseEvent**(*object*)
MouseButtonUpEvents are created when a button on the mouse is released.

The button_state of the event will equal MouseConstants.MOUSE_BUTTON_STATE_RELEASED, and the button that was pressed (button_id) will be MouseConstants.MOUSE_BUTTON_LEFT, MouseConstants.MOUSE_BUTTON_RIGHT, or MouseConstants.MOUSE_BUTTON_MIDDLE, assuming you have a 3 button mouse.

Event Type ID: EventConstants.MOUSE_BUTTON_RELEASE

Event Type String: MOUSE_BUTTON_RELEASE

:rtype : MouseButtonEvent :param args: :param kwargs:

time
: x position of the Mouse when the event occurred; in display coordinate space.

x_position
: x position of the Mouse when the event occurred; in display coordinate space.

button_state
: 1 if a mouse button press caused the event, 0 if button was released.

button_id
: Index of the mouse button that caused the event. MouseConstants.MOUSE_BUTTON_LEFT, MouseConstants.MOUSE_BUTTON_RIGHT and MouseConstants.MOUSE_BUTTON_MIDDLE are int constants representing left, right, and middle buttons of the mouse.

pressed_buttons
: All currently pressed button ids logically ORed together.

y_position
: y position of the Mouse when the event occurred; in display coordinate space.

scroll_x
: Horizontal scroll wheel absolute position when the event occurred. macOS only. Always 0 on other OSs.

scroll_y
: Vertical scroll wheel absolute position when the event occurred.

modifiers
: List of the modifiers that were active when the mouse event occurred, provided as a list of the modifier constant labels.

display_id
: The id of the display that the mouse was over when the event occurred. Only supported on Windows at this time. Always 0 on other OSs.

window_id
: Window handle reference that the mouse was over when the event occurred (window does not need to have focus).

event_id
: The id assigned to the device event. Every event generated by iohub during an experiment session is assigned a unique id, starting from 0.

type
: The type id for the event. This is used to create DeviceEvent objects or dictionary representations of an event based on the data from an event list.

class psychopy.iohub.devices.mouse.**MouseScrollEvent**(*object*)

MouseScrollEvents are generated when the scroll wheel on the Mouse Device (if it has one) is moved. Vertical scrolling is supported on all operating systems, horizontal scrolling is only supported on OS X.

Each MouseScrollEvent provides the number of units the wheel was turned in each supported dimension, as well as the absolute scroll value for of each supported dimension.

Event Type ID: EventConstants.MOUSE_SCROLL

Event Type String: MOUSE_SCROLL

:rtype : MouseScrollEvent :param args: :param kwargs:

time
x position of the Mouse when the event occurred; in display coordinate space.

x_position
x position of the Mouse when the event occurred; in display coordinate space.

button_state
1 if a mouse button press caused the event, 0 if button was released.

button_id
Index of the mouse button that caused the event. MouseConstants.MOUSE_BUTTON_LEFT, MouseConstants.MOUSE_BUTTON_RIGHT and MouseConstants.MOUSE_BUTTON_MIDDLE are int constants representing left, right, and middle buttons of the mouse.

pressed_buttons
All currently pressed button ids logically ORed together.

y_position
y position of the Mouse when the event occurred; in display coordinate space.

scroll_x
Horizontal scroll wheel absolute position when the event occurred. macOS only. Always 0 on other OSs.

scroll_dx
Horizontal scroll wheel position change when the event occurred. macOS only. Always 0 on other OSs.

scroll_y
Vertical scroll wheel absolute position when the event occurred.

scroll_dy
Vertical scroll wheel position change when the event occurred.

modifiers
List of the modifiers that were active when the mouse event occurred, provided as a list of the modifier constant labels.

display_id
The id of the display that the mouse was over when the event occurred. Only supported on Windows at this time. Always 0 on other OSs.

window_id
Window handle reference that the mouse was over when the event occurred (window does not need to have focus).

event_id
The id assigned to the device event. Every event generated by iohub during an experiment session is assigned a unique id, starting from 0.

type
The type id for the event. This is used to create DeviceEvent objects or dictionary representations of an event based on the data from an event list.

ioHub Common Eye Tracker Interface

The iohub commmon 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 experiments iohub_config.yaml:

```
eyetracker.hw.gazepoint.gp3.EyeTracker
```

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:

```
# 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:

```
# 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 devices getEvents(), or clearEvents() methods.

Note that calling clearEvents() at the device level only clears the given devices event buffer. The ioHub Processs 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 devices getEvents() or clearEvents() methods.

Note that calling getEvents() at a device level does not change the Global Event Buffers 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 IDs 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 recordng 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 simple 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 recordng 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 trackers 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 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 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
    # 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)

(continued from previous 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 (not tested)
- macOS (not tested)

Required Python Version:

- Python 3.6 +

Supported Models:

- EyeLink 1000
- EyeLink 1000 Remote (not tested)
- EyeLink 1000 Plus (not tested)

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 PsychPy3, 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:

```
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:

```
# 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:

```
# 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 devices getEvents(), or clearEvents() methods.

Note that calling clearEvents() at the device level only clears the given devices event buffer. The ioHub Processs 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 devices getEvents() or clearEvents() methods.

Note that calling getEvents() at a device level does not change the Global Event Buffers 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 IDs 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 retieved 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 the relative to the head.

right_angle_y
The vertical angle of right eye the 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.

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 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.

Saccade Events

Successful eye tracker calibration must be performed prior to reading (meaningful) saccade event data.

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.

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 EyeTrackerConstants.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 EyeTrackerConstants.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.
    #
    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.
    #
```

(continues on next page)

(continued from previous page)

```
    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
```

(continues on next page)

(continued from previous page)

```
        # 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 asociated 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
            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
```

(continues on next page)

(continued from previous page)

```
    #   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␣
↪type,
        #   and each value is the accociated 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.
        #
        sample_filtering:
            FILTER_ALL: FILTER_LEVEL_OFF

        vog_settings:
```

(continues on next page)

(continued from previous page)

```
            # 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

            # 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 settngs on the eyelink Host and the model and mode␣
↪of
            #   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.
            #
            tracking_mode: PUPIL_CR_TRACKING

            # 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, or CENTROID_FIT
            #
            pupil_center_algorithm: ELLIPSE_FIT

    # model_name: The model_name setting allows the definition of the eye tracker␣
↪model 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

    # 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.

    # 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 isnstance 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 manufactiurer of the device
    # can be specified here. It is not used by the ioHub,
    # so is FYI only.
```

(continues on next page)

(continued from previous page)

```
    #
    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 tobii_research 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 experiments iohub_config.yaml:

```
eyetracker.hw.tobii.EyeTracker
```

Examples

A. Start ioHub with a Tobii device and run tracker calibration:

```
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:

```
# 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:

```
# 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 devices getEvents(), or clearEvents() methods.

Note that calling clearEvents() at the device level only clears the given devices event buffer. The ioHub Processs Global Event Buffer is unchanged.

Parameters **`None`** –

Returns None

`enableEventReporting`(*enabled=True*)

enableEventReporting is functionally identical to the eye tracker device specific enableEventReporting 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 devices getEvents() or clearEvents() methods.

Note that calling getEvents() at a device level does not change the Global Event Buffers 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 IDs 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 retieved 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 recordng 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 simple 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 recordng 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

```
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.
    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:
```

(continues on next page)

(continued from previous page)

```
    # 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:
```

(continues on next page)

(continued from previous page)

```
            # 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.
```

(continues on next page)

(continued from previous page)

```
                #
                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 defualt 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

8.11.3 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 constrains 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 Its 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.

8.12 `psychopy.logging` - control what gets logged

Provides functions for logging error and other messages to one or more files and/or the console, using pythons 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 doesnt use) using the commands:

```python
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 doesnt 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 t%(levelname)s t%(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.**critical**(*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 *log.DEBUG* or higher `psychopy.logging.`**`error`**(*message*)

Send the message to any receiver of logging info (e.g. a LogFile) of level *log.ERROR* or higher

`psychopy.logging.`**`exp`**(*msg*, *t=None*, *obj=None*)

Log a message about the experiment (e.g. a new trial, or end of a stimulus)

usage:: log.exp(message)

Sends the message to any receiver of logging info (e.g. a LogFile) of level *log.EXP* or higher

`psychopy.logging.`**`fatal`**(*msg*, *t=None*, *obj=None*)

log.critical(message) Send the message to any receiver of logging info (e.g. a LogFile) of level *log.CRITICAL* or higher

`psychopy.logging.`**`flush`**(*logger=<psychopy.logging._Logger object>*)

Send current messages in the log to all targets

`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.

`psychopy.logging.`**`info`**(*msg*, *t=None*, *obj=None*)

Log some information - maybe useful, maybe not

usage:: log.info(message)

Sends the message to any receiver of logging info (e.g. a LogFile) of level *log.INFO* or higher

`psychopy.logging.`**`log`**(*msg*, *level*, *t=None*, *obj=None*)

Log a message

usage:: log(level, msg, t=t, obj=obj)

Log the msg, at a given level on the root logger

`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.

`psychopy.logging.`**`warn`**(*msg*, *t=None*, *obj=None*)

log.warning(message)

Sends the message to any receiver of logging info (e.g. a LogFile) of level *log.WARNING* or higher

`psychopy.logging.`**`warning`**(*message*)

Sends the message to any receiver of logging info (e.g. a LogFile) of level *log.WARNING* or higher

8.12.1 `flush()`

`psychopy.logging.`**`flush`**(*logger=<psychopy.logging._Logger object>*)

Send current messages in the log to all targets

8.12.2 `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.

8.13 `psychopy.microphone` - Capture and analyze sound

(Available as of version 1.74.00; Advanced features available as of 1.77.00)

8.13.1 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.

8.13.2 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:

```
DVD or video = 48,000
CD-quality   = 44,100 / 24 bit
human hearing: ~15,000 (adult); children & young adult higher
human speech: 100-8,000 (useful for telephone: 100-3,300)
Google speech API: 16,000 or 8,000 only
Nyquist frequency: twice the highest rate, good to oversample a bit
```

pyos downsamp() function can reduce 48,000 to 16,000 in about 0.02s (uses integer steps 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 pyos 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=16 bits 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.

8.13.3 Speech recognition

Googles speech to text API is no longer available. AT&T, IBM, and wit.ai have a similar (paid) service.

8.13.4 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.

8.14 `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.

8.14.1 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.

8.14.2 From `psychopy.tools.colorspacetools`

`dkl2rgb`(dkl[, conversionMatrix])	Convert from DKL color space (Derrington, Krauskopf & Lennie) to RGB.
`dklCart2rgb`(LUM, LM, S[, conversionMatrix])	Like dkl2rgb except that it uses cartesian coords (LM,S,LUM) rather than spherical coords for DKL (elev, azim, contr).
`rgb2dklCart`(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).
`dkl2rgb`(dkl[, conversionMatrix])	Convert from DKL color space (Derrington, Krauskopf & Lennie) to RGB.

8.14.3 From `psychopy.tools.coordinatetools`

`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).

8.14.4 From `psychopy.tools.monitorunittools`

convertToPix(vertices, pos, units, win)	Takes vertices and position, combines and converts to pixels from any unit
cm2pix(cm, monitor)	Convert size in cm to size in pixels for a given Monitor object.
cm2deg(cm, monitor[, correctFlat])	Convert size in cm to size in degrees for a given Monitor object
deg2cm(degrees, monitor[, correctFlat])	Convert size in degrees to size in pixels for a given Monitor object.
deg2pix(degrees, monitor[, correctFlat])	Convert size in degrees to size in pixels for a given Monitor object
pix2cm(pixels, monitor)	Convert size in pixels to size in cm for a given Monitor object
pix2deg(pixels, monitor[, correctFlat])	Convert size in pixels to size in degrees for a given Monitor object

8.14.5 From `psychopy.tools.imagetools`

array2image(a)	Takes an array and returns an image object (PIL)
image2array(im)	Takes an image object (PIL) and returns a numpy array
makeImageAuto(inarray)	Combines float_uint8 and image2array operations ie.

8.14.6 From `psychopy.tools.plottools`

plotFrameIntervals(intervals)	Plot a histogram of the frame intervals.

8.14.7 From `psychopy.tools.typetools`

float_uint8(inarray)	Converts arrays, lists, tuples and floats ranging -1:1 into an array of Uint8s ranging 0:255
uint8_float(inarray)	Converts arrays, lists, tuples and UINTs ranging 0:255 into an array of floats ranging -1:1
float_uint16(inarray)	Converts arrays, lists, tuples and floats ranging -1:1 into an array of Uint16s ranging 0:2^16

8.14.8 From `psychopy.tools.unittools`

radians	radians(x, /, out=None, *, where=True, casting=same_kind, order=K, dtype=None, subok=True[, signature, extobj])
degrees	degrees(x, /, out=None, *, where=True, casting=same_kind, order=K, dtype=None, subok=True[, signature, extobj])

8.15 `psychopy.monitors` - for those that dont like Monitor Center

Most users wont 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:

```
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

8.15.1 `Monitor`

class psychopy.monitors.**Monitor**(*name*, *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:

`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)

_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. Wont be finalised unless monitor is saved

gammaIsDefault()

Determine whether were 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

getGamma()
Returns just the gamma value (not the whole grid)

getGammaGrid()
Gets the min,max,gamma values for the each gun

getLMS_RGB(*recompute=False*)
Returns the LMS->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.

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 witha a bits++ box

getWidth()
Of the viewable screen in cm, or None if not known

lineariseLums(*desiredLums*, *newInterpolators=False*, *overrideGamma=None*)
Equivalent of `linearizeLums()`.

linearizeLums(*desiredLums*, *newInterpolators=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%psychopy3monitors* 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)

8.15.2 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

8.15.3 `getAllMonitors()`

`psychopy.monitors.`**`getAllMonitors`**`()`

Find the names of all monitors for which calibration files exist

8.15.4 `findPR650()`

8.15.5 `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

8.15.6 `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)

8.15.7 `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)

8.15.8 `gammaInvFun()`

`psychopy.monitors.`**`gammaInvFun`**`(`*yy*, *minLum*, *maxLum*, *gamma*, *b=None*, *eq=1*`)`

Returns inverse gamma function for desired luminance values. x = gammaInvFun(y, 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)
- **eq determines the gamma equation used;** eq==1[default]: yy = a + (b * xx)**gamma eq==2: yy = (a + b*xx)**gamma

8.15.9 `makeDKL2RGB()`

`psychopy.monitors.`**`makeDKL2RGB`**`(`*nm*, *powerRGB*`)`

Creates a 3x3 DKL->RGB conversion matrix from the spectral input powers

8.15.10 `makeLMS2RGB()`

`psychopy.monitors.`**`makeLMS2RGB`**`(`*nm*, *powerRGB*`)`

Creates a 3x3 LMS->RGB conversion matrix from the spectral input powers

8.16 `psychopy.parallel` - functions for interacting with the parallel port

This module provides read / write access to the parallel port for Linux or Windows.

The `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 `ParallelPort` shown below you can use one of the following as drop-in replacements, forcing the use of a specific driver:

- *psychopy.parallel.PParallelInpOut*
- *psychopy.parallel.PParallelDLPortIO*
- *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.

8.16.1 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:

```
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 v easily in python:

```
parallel.setData(int("00000011", 2))  # pins 2 and 3 high
parallel.setData(int("00000101", 2))  # pins 2 and 4 high
```

`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:

```
parallel.setPin(3, 1)  # sets pin 3 high
parallel.setPin(3, 0)  # sets pin 3 low
```

`parallel.`**`readPin`**`()`

Determine whether a desired (input) pin is high(1) or low(0).

Pins 2-13 and 15 are currently read here

8.17 `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 dont want to touch your user settings in general, you can import preferences and set the option *audioLib* accordingly:

```
from psychopy import prefs
prefs.general['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 (dont do this in your actual experiment):

```
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*.

8.17.1 `Preferences`

Class for loading / saving prefs

`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:

```
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 thats running.

loadAll()
Load the user prefs and the application data

loadUserPrefs()
load user prefs, if any; dont 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

8.18 psychopy.serial - functions for interacting with the serial port

PsychoPy is compatible with Chris Liechtis pyserial package. You can use it like this:

```
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.

8.19 psychopy.sound - play various forms of sound

8.19.1 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 ften had issues (at least on macOS) with crashes and freezing of experiments, or causing them not to finish properly. If those issues arent 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). Its newer than *pyo* and so more prone to bugs and we havent 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 foryour audio dont mattter.

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:

```
from psychopy.sound import Sound
```

but the class that gets imported will then be an alias of one of the *Sound Classes* described below.

8.19.2 PTB audio latency

PTB brings a number of advantages in terms of latency.

The first is that is 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 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. Thats pretty good compared to the other options that have a lag of 20ms upwards and sevral ms variability.

BUT, on top of that, PTB allows you to *Preschedule your sound* 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* doesnt mean its 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 buit-in audio (not a fancy audio card):

8.19.3 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 arbitraty time (not in sync with a window flip)

For instance:

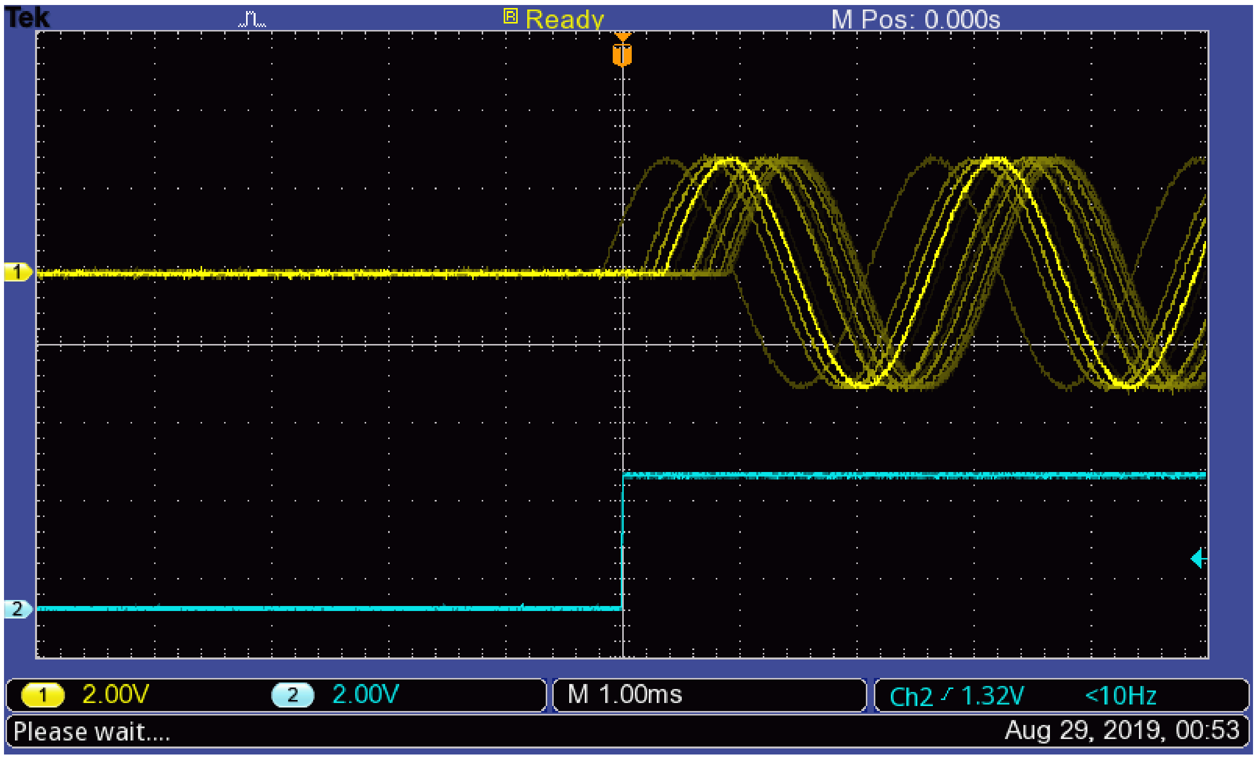


Fig. 1: Sub-ms audio timing with standard audio on Win10. Yellow trace is a 440 Hz tone played at 48 kHz with PTB engine. Cyan trace is the trigger (from a Labjack output). Gridlines are set to 1 ms.

```
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:

```
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_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.

8.19.4 PTB Audio Latency Modes

When using the PTB backend you get the option to choose the Latency Mode, referred to in PsychToolBox as the *reqlatencyclass*.

PsychoPy uses Mode 3 in as a default, assuming that you want low latency and you dont care if other applications cant play sound at the same time (dont listen to iTunes while running your study!)

The modes are as follows:

0 : Latency not important For when it really doesnt 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 isnt 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 youre using and dominates it. That can cause some problems for other apps if theyre 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.

8.19.5 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

_EOS(*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 its 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 wont 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

_EOS(*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 its 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 wont 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.

value: can be a number, string or an array:

- If its 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 wont 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 wont 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

`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 its 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 wont 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. Dont know why you would do this in psychophysics but its easy and fun to include as a possibility :)

`getDuration`()
Gets 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 wont 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() whiles 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

8.20 `psychopy.tools` - miscellaneous tools

Container for all miscellaneous functions and classes

8.20.1 `psychopy.tools.colorspacetools`

Functions and classes related to color space conversion.

dkl2rgb(dkl[, conversionMatrix])	Convert from DKL color space (Derrington, Krauskopf & Lennie) to RGB.

Continued on next page

Table 25 – continued from previous page

dklCart2rgb(LUM, LM, S[, conversionMatrix])	Like dkl2rgb except that it uses cartesian coords (LM,S,LUM) rather than spherical coords for DKL (elev, azim, contr).
rgb2dklCart(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).
dkl2rgb(dkl[, conversionMatrix])	Convert from DKL color space (Derrington, Krauskopf & Lennie) to RGB.
cielab2rgb(lab[, whiteXYZ,])	Transform CIE L*a*b* (1976) color space coordinates to RGB tristimulus values.
cielch2rgb(lch[, whiteXYZ,])	Transform CIE L*C*h* coordinates to RGB tristimulus values.
srgbTF(rgb[, reverse])	Apply sRGB transfer function (or gamma) to linear RGB values.
rec709TF(rgb, **kwargs)	Apply the Rec.

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:

```
rgb(Nx3) = dkl2rgb(dkl_Nx3(el,az,radius), conversionMatrix)
rgb(NxNx3) = dkl2rgb(dkl_NxNx3(el,az,radius), conversionMatrix)
```

psychopy.tools.colorspacetools.**dklCart2rgb**(*LUM*, *LM*, *S*, *conversionMatrix=None*)

Like dkl2rgb except that it uses cartesian coords (LM,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 CIE L*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 (its 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 displays gamut may not be valid!

Returns Array of RGB tristimulus values.

Return type ndarray

Example

Converting a CIE L*a*b* color to linear RGB:

```
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:

```
rgbColor = cst.cielab2rgb(cielabColor, transferFunc=cst.srgbTF)
```

`psychopy.tools.colorspacetools.`**`cielch2rgb`**(*lch*, *whiteXYZ=None*, *conversionMatrix=None*, *transferFunc=None*, *clip=False*, ***kwargs*)

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 (its 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`** (*`boolean`*) – Make all output values representable by the display. However, colors outside of the displays 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

- **`rgb`** (*`tuple, list or ndarray of floats`*) – Nx3 or NxNx3 array of linear RGB values, last dim must be size == 3 specifying RBG values.
- **`reverse`** (*`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`**(*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 **rgb** (*tuple, list or ndarray of floats*) – Nx3 or NxNx3 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

8.20.2 `psychopy.tools.coordinatetools`

Functions and classes related to coordinate system conversion

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).

Function details

`psychopy.tools.coordinatetools.`**`cart2pol`**(*x*, *y*, *units='deg'*)

Convert from cartesian to polar coordinates.

Usage theta, radius = cart2pol(x, y, units=deg)

units refers to the units (rad or deg) for theta that should be returned

`psychopy.tools.coordinatetools.`**`cart2sph`**(*z*, *y*, *x*)

Convert from cartesian coordinates (x,y,z) to spherical (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 = Luminance, y = S and x = LM

`psychopy.tools.coordinatetools.`**`pol2cart`**(*theta*, *radius*, *units='deg'*)

Convert from polar to cartesian coordinates.

usage:

```
x,y = pol2cart(theta, radius, units='deg')
```

`psychopy.tools.coordinatetools.`**`sph2cart`**(**args*)

Convert from spherical coordinates (elevation, azimuth, radius) to cartesian (x,y,z).

usage: array3xN[x,y,z] = sph2cart(array3xN[el,az,rad]) OR x,y,z = sph2cart(elev, azim, radius)

8.20.3 `psychopy.tools.filetools`

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.

8.20.4 `psychopy.tools.gltools`

Overview

`createProgram`
`createProgramObjectARB`
`compileShader`

Continued on next page

Table 27 – continued from previous page

compileShaderObjectARB
embedShaderSourceDefs
deleteObject
deleteObjectARB
attachShader
attachObjectARB
detachShader
detachObjectARB
linkProgram
linkProgramObjectARB
validateProgram
validateProgramARB
useProgram
useProgramObjectARB
getInfoLog
getUniformLocations
getAttribLocations
createQueryObject
beginQuery
endQuery
getQuery
getAbsTimeGPU
createFBO
attach
isComplete
deleteFBO
blitFBO
useFBO
createRenderbuffer
deleteRenderbuffer
createTexImage2D
createTexImage2DMultisample
deleteTexture
VertexArrayInfo
createVAO
drawVAO
deleteVAO
VertexBufferInfo
createVBO
bindVBO
unbindVBO
mapBuffer
unmapBuffer
deleteVBO
setVertexAttribPointer
enableVertexAttribArray
disableVertexAttribArray
createMaterial
useMaterial
createLight

Continued on next page

Table 27 – continued from previous page

useLights
setAmbientLight
ObjMeshInfo
loadObjFile
loadMtlFile
createUVSphere
createPlane
createMeshGrid
createBox
getIntegerv
getFloatv
getString
getOpenGLInfo

Details

Examples

Working with Framebuffer Objects (FBOs):

Creating an empty framebuffer with no attachments:

```
fbo = createFBO()  # invalid until attachments are added
```

Create a render target with multiple color texture attachments:

```
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 dont know the current binding state:

```
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:

```
fbo.userData['flags'] = ['left_eye', 'clear_before_use']
```

Binding an FBO for drawing/reading:

```
GL.glBindFramebuffer(GL.GL_FRAMEBUFFER, fb.id)
```

Depth-only framebuffers are valid, sometimes need for generating shadows:

```
depthTex = createTexImage2D(800, 600,
                            internalFormat=GL.GL_DEPTH_COMPONENT24,
                            pixelFormat=GL.GL_DEPTH_COMPONENT)
fbo = createFBO([(GL.GL_DEPTH_ATTACHMENT, depthTex)])
```

Deleting a framebuffer when done with it. This invalidates the framebuffers ID and makes it available for use:

```
deleteFBO(fbo)
```

8.20.5 `psychopy.tools.imagetools`

Functions and classes related to image handling

array2image(a)	Takes an array and returns an image object (PIL)
image2array(im)	Takes an image object (PIL) and returns a numpy array
makeImageAuto(inarray)	Combines float_uint8 and image2array operations ie.

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

8.20.6 `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:

```
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:

```
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 (eg. 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.

Overview

`length`(v[, squared, out, dtype])	Get the length of a vector.
`normalize`(v[, out, dtype])	Normalize a vector or quaternion.
`orthogonalize`(v, n[, out, dtype])	Orthogonalize a vector relative to a normal vector.
`reflect`(v, n[, out, dtype])	Reflection of a vector.
`dot`(v0, v1[, out, dtype])	Dot product of two vectors.
`cross`(v0, v1[, out, dtype])	Cross product of 3D vectors.
`project`(v0, v1[, out, dtype])	Project a vector onto another.
`perp`(v, n[, norm, out, dtype])	Project *v* to be a perpendicular axis of *n*.
`lerp`(v0, v1, t[, out, dtype])	Linear interpolation (LERP) between two vectors/coordinates.
`distance`(v0, v1[, out, dtype])	Get the distance between vectors/coordinates.
`angleTo`(v, point[, degrees, out, dtype])	Get the relative angle to a point from a vector.
`surfaceNormal`(tri[, norm, out, dtype])	Compute the surface normal of a given triangle.
`surfaceBitangent`(tri, uv[, norm, out, dtype])	Compute the bitangent vector of a given triangle.
`surfaceTangent`(tri, uv[, norm, out, dtype])	Compute the tangent vector of a given triangle.
`vertexNormal`(faceNorms[, norm, out, dtype])	Compute a vertex normal from shared triangles.
`intersectRayPlane`(orig, dir, planeOrig,)	Get the point which a ray intersects a plane.
`ortho3Dto2D`(p, orig, normal, up)	Get the planar coordinates of an orthogonal projection of a 3D point onto a 2D plane.
`slerp`(q0, q1, t[, shortest, out, dtype])	Spherical linear interpolation (SLERP) between two quaternions.
`quatToAxisAngle`(q[, degrees, dtype])	Convert a quaternion to *axis* and *angle* representation.
`quatFromAxisAngle`(axis, angle[, degrees, dtype])	Create a quaternion to represent a rotation about *axis* vector by *angle*.

Continued on next page

Table 29 – continued from previous page

`quatMagnitude`(q[, squared, out, dtype])	Get the magnitude of a quaternion.
`multQuat`(q0, q1[, out, dtype])	Multiply quaternion *q0* and *q1*.
`invertQuat`(q[, out, dtype])	Get tht multiplicative inverse of a quaternion.
`applyQuat`(q, points[, out, dtype])	Rotate points/coordinates using a quaternion.
`quatToMatrix`(q[, out, dtype])	Create a 4x4 rotation matrix from a quaternion.
`scaleMatrix`(s[, out, dtype])	Create a scaling matrix.
`rotationMatrix`(angle[, axis, out, dtype])	Create a rotation matrix.
`translationMatrix`(t[, out, dtype])	Create a translation matrix.
`invertMatrix`(m[, homogeneous, out, dtype])	Invert a 4x4 matrix.
`isOrthogonal`(m)	Check if a square matrix is orthogonal.
`isAffine`(m)	Check if a 4x4 square matrix describes an affine transformation.
`concatenate`(matrices[, out, dtype])	Concatenate matrix transformations.
`applyMatrix`(m, points[, out, dtype])	Apply a matrix over a 2D array of points.
`posOriToMatrix`(pos, ori[, out, dtype])	Convert a rigid body pose to a 4x4 transformation matrix.
`transform`(pos, ori, points[, out, dtype])	Transform points using a position and orientation.

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.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be float32 or float64. If *None* is specified, the data type of *out* is used. If *out* is not provided, float64 is used by default.

Returns Length of vector *v*.

Return type float or ndarray

`psychopy.tools.mathtools.`**`normalize`**(*v*, *out=None*, *dtype=None*)

Normalize a vector or quaternion.

v [array_like] Vector to normalize, can be Nx2, Nx3, or Nx4. If a 2D array is specified, rows are treated as separate vectors.

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 Normalized vector *v*.

Return type ndarray

Notes

- If the vector is degenerate (length is zero), a vector of all zeros is returned.

Examples

Normalize a vector:

```
v = [1., 2., 3., 4.]
vn = normalize(v)
```

The *normalize* function is vectorized. Its considerably faster to normalize large arrays of vectors than to call *normalize* separately for each one:

```
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!
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 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 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 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 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

- **v1** (`v0,`) – 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 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 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

- **v1** (`v0,`) – 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 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 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:

```
# create two 6x3 arrays with random numbers
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 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 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 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 Perpendicular axis of *n* from *v*.

Return type ndarray

Examples

Determine the local *up* (y-axis) of a surface or plane given *normal*:

```
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:

```
xaxis = cross(yaxis, normal)
```

psychopy.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 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 Vector at *t* with same shape as *v0* and *v1*.

Return type ndarray

Examples

Find the coordinate of the midpoint between two vectors:

```
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

- **v1** (*v0,*) – 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 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 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 *v0* and *v1* are 1D, the angle is returned as a scalar and *out* is ignored.
- If *v0* and *v1* are 2D, an array of angles between corresponding row vectors are returned.
- If either *v0* and *v1* 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].
- **point** (*array_like*) – Point(s) to compute angle to from vector *v*.
- **degrees** (*bool, optional*) – Return the resulting angles in degrees. If *False*, angles will be returned in 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 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 Distance between vectors *v0* and *v1*.

Return type ndarray

`psychopy.tools.mathtools.`**`surfaceNormal`**(*tri*, *norm=True*, *out=None*, *dtype=None*)

Compute the surface normal of a given triangle.

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.

- **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 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 Surface normal of triangle *tri*.

Return type ndarray

Examples

Compute the surface normal of a triangle:

```
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:

```
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 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 Surface bitangent of triangle *tri*.

Return type ndarray

Examples

Computing the bitangents for two triangles from vertex and texture coordinates (UVs):

```
# 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 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 Surface normal of triangle *tri*.

Return type ndarray

Examples

Compute surface normals, tangents, and bitangents for a list of triangles:

```
# triangle vertices (2x3x3)
vertices = [[[-1., 0., 0.], [0., 1., 0.], [1, 0, 0]],
            [[1., 0., 0.], [0., 1., 0.], [-1, 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)

(continued from previous 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 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 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.`**`intersectRayPlane`**(*orig*, *dir*, *planeOrig*, *planeNormal*)

Get the point which a ray intersects a plane.

Parameters

- **`orig`** (*array_like*) – Origin of the line in space [x, y, z].
- **`dir`** (*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].

Returns Position in space which the line intersects the plane. *None* is returned if the line does not intersect the plane at a single point or at all.

Return type ndarray

Examples

Find the point in the scene a ray intersects the plane:

```
# 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 in 3D world space
pnt = intersectRayPlane(rayOrigin, rayDir, planeOrigin, planeNormal)
```

psychopy.tools.mathtools.**ortho3Dto2D**(*p*, *orig*, *normal*, *up*)

Get the planar coordinates of an orthogonal projection of a 3D point onto a 2D 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 planes coordinate system. Must be perpendicular to *normal*.

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:

```
# 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.`**`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 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 Quaternion [x, y, z, w] at *t*.

Return type ndarray

Examples

Interpolate between two orientations:

```
q0 = quatFromAxisAngle(90.0, degrees=True)
q1 = quatFromAxisAngle(-90.0, degrees=True)
# halfway between 90 and -90 is 0.0 or quaternion [0. 0. 0. 1.]
qr = slerp(q0, q1, 0.5)
```

Example of smooth rotation of an object with fixed angular velocity:

```
degPerSec = 10.0  # rotate a stimulus at 10 degrees per second

# initial orientation, axis rotates in the Z direction
qr = quatFromAxisAngle([0., 0., -1.], 0.0, degrees=True)
# amount to rotate every second
qv = quatFromAxisAngle([0., 0., -1.], degPerSec, degrees=True)

# ---- within main experiment loop ----
# `frameTime` is the time elapsed in seconds from last `slerp`.
qr = multQuat(qr, slerp((0., 0., 0., 1.), qv, degPerSec * frameTime))
_, angle = quatToAxisAngle(qr)  # discard axis, only need angle

# myStim is a GratingStim or anything with an 'ori' argument which
# accepts angle in degrees
```

(continues on next page)

(continued from previous page)

```
myStim.ori = angle
myStim.draw()
```

psychopy.tools.mathtools.**quatToAxisAngle**(*q*, *degrees=True*, *dtype=None*)
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.
- **degrees** (*bool, optional*) – Indicate *angle* is to be returned in degrees, otherwise *angle* will be returned in radians.
- **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 Axis and angle of quaternion in form ([ax, ay, az], angle). If *degrees* is *True*, the angle returned is in degrees, radians if *False*.

Return type tuple

Examples

Using a quaternion to rotate a stimulus a fixed angle each frame:

```
# 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()
```

psychopy.tools.mathtools.**quatFromAxisAngle**(*axis*, *angle*, *degrees=True*, *dtype=None*)
Create a quaternion to represent a rotation about *axis* vector by *angle*.

Parameters

- **axis** (*tuple, list or ndarray, optional*) – Axis of rotation [x, y, z].
- **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 arrays, can either be float32 or float64. If *None* is specified, float64 is used.

Returns Quaternion [x, y, z, w].

Return type ndarray

Examples

Create a quaternion from specified *axis* and *angle*:

```
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.`**`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 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 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

- **q1** (*q0,*) – 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 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 Combined orientations of *q0* amd *q1*.

Return type ndarray

Notes

- Quaternions are normalized prior to multiplication.

Examples

Combine the orientations of two quaternions:

```
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.`**`invertQuat`**(*q*, *out=None*, *dtype=None*)

Get tht 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 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 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]:

```
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 points/coordinates 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 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 Transformed points.

Return type ndarray

Examples

Rotate points using a quaternion:

```
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:

```
axis = [0., 0., -1.]
angle = -90.0
rotMat = rotationMatrix(axis, angle)[:3, :3]  # rotation sub-matrix only
rotQuat = quatFromAxisAngle(axis, angle, 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*:

```
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 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 rotation matrix in row-major order.

Return type ndarray or None

Examples

Convert a quaternion to a rotation matrix:

```
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):

```
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.`**`scaleMatrix`**(*s*, *out=None*, *dtype=None*)

Create a scaling matrix.

The resulting matrix is the same as a generated by a *glScale* call.

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 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 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** (*ndarray, list, or tuple of float*) – Axis vector 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 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 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 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 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*, *homogeneous=False*, *out=None*, *dtype=None*)

Invert a 4x4 matrix.

Parameters

- **m** (*array_like*) – 4x4 matrix to invert.
- **homogeneous** (*bool, optional*) – Set as `True` if the input matrix specifies affine (homogeneous) transformations (scale, rotation, and translation). This will use a faster inverse method which handles such cases. Default is `False`.
- **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 specified, the default is float64.

Returns 4x4 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.**concatenate**(*matrices*, *out=None*, *dtype=None*)

Concatenate matrix transformations.

Combine 4x4 transformation matrices into a single matrix. This is similar to what occurs when building a matrix stack in OpenGL using *glRotate*, *glTranslate*, and *glScale* calls. Matrices are multiplied together from right-to-left, or the last item to first. Note that changing the order of the input matrices changes the final result.

The data types of input matrices are coerced to match that of *out* or *dtype* if *out* is *None*. For performance reasons, it is best that all arrays passed to this function have matching data types.

Parameters

- **matrices** (*list or tuple*) – List of matrices to concatenate. All matrices must be 4x4.
- **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 Concatenation of input matrices as a 4x4 matrix in row-major order.

Return type ndarray

Examples

Create an SRT (scale, rotate, and translate) matrix to convert model-space coordinates to world-space:

```
S = scaleMatrix([2.0, 2.0, 2.0])  # scale model 2x
R = rotationMatrix(-90., [0., 0., -1])  # rotate -90 about -Z axis
T = translationMatrix([0., 0., -5.])  # translate point 5 units away
SRT = concatenate([S, R, T])

# transform a point in model-space coordinates to world-space
pointModel = np.array([0., 1., 0., 1.])
pointWorld = np.matmul(SRT, pointModel.T)  # point in WCS
# ... or ...
pointWorld = 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:

```
# stimulus pose as quaternion and vector
stimOri = quatFromAxisAngle([0., 0., -1.], -45.0)
stimPos = [0., 1.5, -5.]
```

(continues on next page)

(continued from previous page)

```
# 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:

```
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:

```
# 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.`**`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:

```
points[:, :] = 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 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 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:

```
# 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:

```
S = scaleMatrix([5.0, 5.0, 5.0])  # scale 5x
R = rotationMatrix(180., [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:

```
sRGBMatrix = [[3.2404542, -1.5371385, -0.4985314],
              [-0.969266,  1.8760108,  0.041556 ],
              [0.0556434, -0.2040259,  1.0572252]]

colorsRGB = applyMatrix(sRGBMatrix, colorsXYZ)
```

`psychopy.tools.mathtools.`**`transform`**(*pos*, *ori*, *points*, *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 *None* is specified, the data type of *out* is used. 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:

```
# 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:

```
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.

8.20.7 `psychopy.tools.monitorunittools`

Functions and classes related to unit conversion respective to a particular monitor

`convertToPix`(vertices, pos, units, win)	Takes vertices and position, combines and converts to pixels from any unit
`cm2deg`(cm, monitor[, correctFlat])	Convert size in cm to size in degrees for a given Monitor object

Continued on next page

Table 30 – continued from previous page

cm2pix(cm, monitor)	Convert size in cm to size in pixels for a given Monitor object.
deg2cm(degrees, monitor[, correctFlat])	Convert size in degrees to size in pixels for a given Monitor object.
deg2pix(degrees, monitor[, correctFlat])	Convert size in degrees to size in pixels for a given Monitor object
pix2cm(pixels, monitor)	Convert size in pixels to size in cm for a given Monitor object
pix2deg(pixels, monitor[, correctFlat])	Convert size in pixels to size in degrees for a given Monitor object

Function details

psychopy.tools.monitorunittools.**convertToPix**(*vertices*, *pos*, *units*, *win*)

Takes vertices and position, combines and converts to pixels from any unit

The reason that *pos* and *vertices* are provided separately is that it allows the conversion from deg to apply flat-screen correction to each separately.

The reason that these use function args rather than relying on self.pos is that some stimuli use other terms (e.g. ElementArrayStim uses fieldPos).

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

8.20.8 `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

8.20.9 `psychopy.tools.rifttools`

Tools for the Oculus Rift.

Copyright (C) 2018 - Matthew D. Cutone, The Centre for Vision Research, Toronto, Ontario, Canada

`psychopy.tools.rifttools.`**`ovrSizei = <class 'psychxr.ovr.math.ovrSizei'>`**

`psychopy.tools.rifttools.`**`ovrRect = <class 'psychxr.ovr.math.ovrRecti'>`**

ovrRecti(*args, **kwargs)

`psychopy.tools.rifttools.`**`ovrVector3f = <class 'psychxr.ovr.math.ovrVector3f'>`**

ovrVector3f(*args, **kwargs)

`psychopy.tools.rifttools.`**`ovrMatrix4f = <class 'psychxr.ovr.math.ovrMatrix4f'>`**

ovrMatrix4f

4x4 Matrix typically used for 3D transformations. By default, all matrices are right handed. Values are stored in row-major order. Transformations are applied left-to-right.

`psychopy.tools.rifttools.`**`ovrQuat = <class 'psychxr.ovr.math.ovrQuatf'>`**

`psychopy.tools.rifttools.`**`ovrPosef = <class 'psychxr.ovr.math.ovrPosef'>`**

ovrPosef

Class to represent a pose using a vector and quaternion.

`psychopy.tools.rifttools.`**`ovrFovPort = <class 'psychxr.ovr.math.ovrFovPort'>`**

`psychopy.tools.rifttools.`**`OVR_EYE_LEFT = 0`**

int(x=0) -> integer int(x, base=10) -> integer

Convert a number or string to an integer, or return 0 if no arguments are given. If x is a number, return x.__int__(). For floating point numbers, this truncates towards zero.

If x is not a number or if base is given, then x must be a string, bytes, or bytearray instance representing an integer literal in the given base. The literal can be preceded by + or - and be surrounded by whitespace. The base defaults to 10. Valid bases are 0 and 2-36. Base 0 means to interpret the base from the string as an integer literal. >>> int(0b100, base=0) 4

`psychopy.tools.rifttools.`**`OVR_EYE_RIGHT = 1`**

int(x=0) -> integer int(x, base=10) -> integer

Convert a number or string to an integer, or return 0 if no arguments are given. If x is a number, return x.__int__(). For floating point numbers, this truncates towards zero.

If x is not a number or if base is given, then x must be a string, bytes, or bytearray instance representing an integer literal in the given base. The literal can be preceded by + or - and be surrounded by whitespace. The base defaults to 10. Valid bases are 0 and 2-36. Base 0 means to interpret the base from the string as an integer literal. >>> int(0b100, base=0) 4

`psychopy.tools.rifttools.`**`OVR_HAND_LEFT = 0`**

int(x=0) -> integer int(x, base=10) -> integer

Convert a number or string to an integer, or return 0 if no arguments are given. If x is a number, return x.__int__(). For floating point numbers, this truncates towards zero.

If x is not a number or if base is given, then x must be a string, bytes, or bytearray instance representing an integer literal in the given base. The literal can be preceded by + or - and be surrounded by whitespace. The base defaults to 10. Valid bases are 0 and 2-36. Base 0 means to interpret the base from the string as an integer literal. >>> int(0b100, base=0) 4 `psychopy.tools.rifttools.`**`OVR_HAND_RIGHT = 1`**

int(x=0) -> integer int(x, base=10) -> integer

Convert a number or string to an integer, or return 0 if no arguments are given. If x is a number, return x.__int__(). For floating point numbers, this truncates towards zero.

If x is not a number or if base is given, then x must be a string, bytes, or bytearray instance representing an integer literal in the given base. The literal can be preceded by + or - and be surrounded by whitespace. The base defaults to 10. Valid bases are 0 and 2-36. Base 0 means to interpret the base from the string as an integer literal. >>> int(0b100, base=0) 4

8.20.10 `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

```
>>> 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

```
>>> 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

```
>>> float_uint16(-1)
0
>>> float_uint16(0)
32768
```

8.20.11 `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*, *extobj*])

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*) – Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

- ****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

```
>>> 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])
```

```
>>> 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*) – Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.
- ****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

```
>>> rad = np.arange(12.)*np.pi/6
>>> np.degrees(rad)
array([   0.,   30.,   60.,   90.,  120.,  150.,  180.,  210.,  240.,
        270.,  300.,  330.])
```

```
>>> out = np.zeros((rad.shape))
>>> r = degrees(rad, out)
>>> np.all(r == out)
True
```

8.20.12 `psychopy.tools.viewtools`

Tools for working with view projections for 2- and 3-D rendering.

computeFrustum(scrWidth, scrAspect, scrDist)	Calculate frustum parameters.
generalizedPerspectiveProjection([,])	Generalized derivation of projection and view matrices based on the physical configuration of the display system.
orthoProjectionMatrix(left, right, bottom,)	Compute an orthographic projection matrix with provided frustum parameters.
perspectiveProjectionMatrix(left, right,)	Compute an perspective projection matrix with provided frustum parameters.
lookAt(eyePos, centerPos[, upVec, out, dtype])	Create a transformation matrix to orient a view towards some point.
pointToNdc(wcsPos, viewMatrix, projectionMatrix)	Map the position of a point in world space to normalized device coordinates/space.

Function details

psychopy.tools.viewtools.**computeFrustum**(*scrWidth*, *scrAspect*, *scrDist*, *convergeOffset=0.0*, *eyeOffset=0.0*, *nearClip=0.01*, *farClip=100.0*)

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 displays 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.

Returns Namedtuple with frustum parameters. Can be directly passed to glFrustum (e.g. glFrustum(*f)).

Return type Frustum

Notes

- The view point 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 accountfor 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 windows projection matrix:

```
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:

```
left, right, bottom, top, nearVal, farVal = frustum
# ... or ...
left = frustum.left
```

Off-axis frustums for stereo rendering:

```
# 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:

```
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
```

`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 Kooimas 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

Electron. Eng. Comput. Sci.

Examples

Computing a projection and view matrices for a window:

[1] Kooima, R. (2009). Generalized perspective projection. J. Sch.

```
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):

```
# 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(
    posBottomLeft, posBottomRight, posTopLeft, posLeftEye)
rightProjMatrix, rightViewMatrix = generalizedPerspectiveProjection(
    posBottomLeft, posBottomRight, posTopLeft, posRightEye)
```

`psychopy.tools.viewtools.`**`orthoProjectionMatrix`**(*left*, *right*, *bottom*, *top*, *nearClip*, *farClip*, *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*, *farClip*, *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 observers 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:

```
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:

```
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])
```

(continues on next page)

(continued from previous page)

```
pixelY = ((y + 1.0) / 2.0) * scrRes[1])
# object at point will appear at (pixelX, pixelY)
```

8.21 `psychopy.voicekey` - Real-time sound processing

(Available as of version 1.83.00)

8.21.1 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 is 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.

8.21.2 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

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 cant 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.

8.21.3 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.**zero_crossings**(*data*)
Return a vector of length n-1 of zero-crossings within vector *data*.

1 if the adjacent values switched sign, or 0 if they stayed the same sign.

psychopy.voicekey.**tone**(*freq=440*, *sec=2*, *rate=44100*, *vol=0.99*)
Return a np.array suitable for use as a tone (pure sine wave).

psychopy.voicekey.**apodize**(*data*, *ms=5*, *rate=44100*)
Apply a Hanning window (5ms) to reduce a sounds click onset / offset.

8.21.4 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.`**`samples_from_table`**(*table*, *start=0*, *stop=-1*, *rate=44100*)
Return samples as a np.array read from a pyo table.

A (start, stop) selection in seconds may require a non-default rate.

`psychopy.voicekey.`**`table_from_samples`**(*samples*, *start=0*, *stop=-1*, *rate=44100*)
Return a pyo DataTable constructed from samples.

A (start, stop) selection in seconds may require a non-default rate.

`psychopy.voicekey.`**`table_from_file`**(*file_in*, *start=0*, *stop=-1*)
Read data from files, any pyo format, returns (rate, pyo SndTable)

`psychopy.voicekey.`**`samples_from_file`**(*file_in*, *start=0*, *stop=-1*)
Read data from files, returns tuple (rate, np.array(.float64))

`psychopy.voicekey.`**`samples_to_file`**(*samples*, *rate*, *file_out*, *fmt=''*, *dtype='int16'*)
Write data to file, using requested format or infer from file .ext.

Only integer *rate* values are supported.

See http://ajaxsoundstudio.com/pyodoc/api/functions/sndfile.html

`psychopy.voicekey.`**`table_to_file`**(*table*, *file_out*, *fmt=''*, *dtype='int16'*)
Write data to file, using requested format or infer from file .ext.

8.22 `psychopy.web` - Web methods

8.22.1 Test for access

`psychopy.web.`**`haveInternetAccess`**(*forceCheck=False*)
Detect active internet connection or fail quickly.

If forceCheck is False, will rely on a cached value if possible.

`psychopy.web.`**`requireInternetAccess`**(*forceCheck=False*)
Checks for access to the internet, raise error if no access.

8.22.2 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:

CHAPTER

NINE

TROUBLESHOOTING

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.

9.1 The application doesnt 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 doesnt help then we need to get to an error message in order to work out why the application isnt 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 doesnt 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

9.2 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 dont get an error message but the application still doesnt start then manually turn off the viewing of the Output (as below) and try the above again.

9.3 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).

9.4 Use the source (Luke?)

PsychoPy comes with all the source code included. You may not think youre 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*

9.5 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 isnt working so that the underlying cause of the problem can be investigated (google first to see if its a known issue).

9.6 Errors with getting/setting the Gamma ramp

There are two common causes for errors getting/setting gamma ramps depending on whether youre running Windows or Linux (we havent seen these problems on Mac).

9.6.1 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 dont 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 isnt much that the PsychoPy team can do until Microsoft fixes the underlying bug. Youll 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 cant use dual independent displays for vision science studies until Microsoft fix it.

9.6.2 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 weve 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

CHAPTER

TEN

RECIPES (HOW-TOS)

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?).

10.1 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 its 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.

10.1.1 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.

10.1.2 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/python2.6. Put your .pth file here, next to the various libraries.

The advantage of this method is that you dont need to do the import psychopy step. The downside is that when you update PsychoPy to a new major release youll need to repeat this step (patch updates wont affect it though).

10.2 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 peoples names and a working script.)

10.3 Scrolling text

Key idea: Vary the **position** of the stimulus across frames.

Question: How can I produce scrolling text (like htmls <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.

10.4 Fade-in / fade-out effects

Key idea: vary the **opacity** of the stimulus over frames.

Question: Id 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 doesnt 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.

10.5 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 wont be provided so they cant 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 dont 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 youve done that then feel free to contribute the necessary script or instructions.

10.5.1 Using py2exe to build an executable

Instructions:

1. Download and install py2exe (http://www.py2exe.org/)
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:

```
#   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 = []
```

(continues on next page)

(continued from previous page)

```
my_data_files=matplotlib.get_py2exe_datafiles()

#define which files you want to copy for data_files
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
#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)

#all_files = [("psychopy\\preferences", preference_files),("psychopy\\app", app_
↪files), my_data_files[0]]

#combine the files
all_files = [("psychopy\\preferences", preference_files), my_data_files[0]]

#define the setup
setup(
                console=[progName],
                data_files = all_files,
                options = {
                    "py2exe":{
                        "skip_archive": True,
                        "optimize": 2
                    }
                }
)
```

10.6 Builder - providing feedback

If youre 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 arent used to them (they contain odd characters like %.2f). See *Generating formatted strings* for more info.

10.6.1 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:

```
key_resp.keys #a python list of keys pressed
key_resp.rt #the time to the first key press
key_resp.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 key_resp.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"
```

10.6.2 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['key_resp.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['key_resp.corr'].sum() #.std(), .mean() also available
meanRt = trials.data['key_resp.rt'].mean()
msg = "You got %i trials correct (rt=%.2f)" %(nCorr,meanRt)
```

10.6.3 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*.

10.7 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*.

10.8 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.

10.8.1 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\*

10.8.2 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:

> *<NETWORK_LOC>\psychopyLib\psychopy*

10.8.3 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:

```
>>> 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.

10.8.4 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 icons location to:

```
"<NETWORK_LOC>\psychopyLib\psychopy\app\Resources\psychopy.ico"
```

10.8.5 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.

10.9 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:

```
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:

```
longMessage = 'Well done %s that took %0.3f seconds' %(info['name'], time)
```

Some of the handy formatted string types:

```
>>> x=5
>>> x1=5124
>>> z='someText'
>>> 'show %s' %(z)
'show someText'
>>> '%0.1f' %(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.

10.10 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.

```
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=[]
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
```

(continues on next page)

(continued from previous page)

```
    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)
```

10.11 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: Theres a difference in luminance between my stimuli. How could I correct for that?

Im running an experiment where I manipulate color chromatic saturation, keeping luminance constant. Ive 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 dont possess spectrophotometers such as PR650 in our lab to calibrate each color gun. Ive calibrated the gamma of my monitor psychophysically. Gamma was set to 1.7 (threshold) for gamm(lum), gamma(R), gamma(G), gamma(B). Then Ive measured the luminance of each stimuli with a Brontes colorimeter. But theres a difference in luminance between my stimuli. How could I correct for that?

A2: Without a spectroradiometer you wont be able to use the color spaces like DKL which are designed to help this sort of thing.

If you dont 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=46cd/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 youre 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.

10.12 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.

```
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)
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)
```

(continues on next page)

(continued from previous page)

```
    myStim.setTex(pi)
    mywin.flip()
    theKey = event.getKeys()
    if len(theKey) != 0:
        break
```

CHAPTER

ELEVEN

FREQUENTLY ASKED QUESTIONS (FAQS)

11.1 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 isnt 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 youre 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 thats 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 dont worry so much about it).
- youve got a Mac thats 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.

11.2 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*

CHAPTER

TWELVE

RESOURCES (E.G. FOR TEACHING)

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!

12.1 Workshops

At Nottingham we run an annual workshop on Python/PsychoPy (ie. programming, not Builder). Please see the page on officialWorkshops for further details.

12.2 Youtube tutorials

- Youtube PsychoPy tutorial showing how to build a basic experiment in the *Builder* interface. Thats a great way to get started; build your own complete experiment in 15 minutes flat!
- Theres also a subtitled version of the stroop video tutorial (Thanks Kevin Cole for doing that!)
- 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

12.3 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).

- Theres a set of tools for teaching psychophysics using PsychoPy and a PsychoPysics poster from VSS. Thanks James Ferwerda

12.4 Materials for Coder

- Please see the page on officialWorkshops for further details on coming to an intensive residential Python workshop in Nottingham.
- Marco Bertamimis 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).

12.5 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
- Yale, 21-23 July : The first ever dedicated PsychoPy workshop/conference was at Yale, 21-23 July 2011. Thanks Jeremy for organising!
- EPS Satellite workshop, 8 July 2011
- BPS Maths Stats and Computing Section workshop (Dec 2010):

For developers:

CHAPTER

THIRTEEN

FOR DEVELOPERS

There is a separate mailing list to discuss development ideas and issues.

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 suggestedPackages for developers.

Dont *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.

13.1 Using the repository

Note: Much of the following is explained with more detail in the nitime documentation, and then in further detail in numerous online tutorials.

13.1.1 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.

- Create a fork of the central psychopy/psychopy repository
- Create a local clone of that fork
- **For small changes**
  - make the changes directly in the master branch
  - push back to your fork
  - submit a pull request to the central repository
- **For substantial changes (new features)**
  - create a branch
  - when finished run unit tests
  - when the unit tests pass merge changes back into the *master* branch
  - submit a pull request to the central repository

13.1.2 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.

13.1.3 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.

13.1.4 Fetching the latest version

Periodically its 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
```

13.1.5 Run PsychoPy using your local copy

Now that youve 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!

13.1.6 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
```

13.1.7 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 hasnt 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.

13.1.8 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? (Its 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 its not possible, its 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 code base 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 – well eventually move to the underscore style, but for now keep everything consistent within the code base). In your own code, write however you like of course. This is just about when contributing to the project.

13.1.9 Add a new feature branch

For more substantial work, 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
```

13.1.10 Completing work on a feature

When youre done run the unit tests for your feature branch. Set the *debug* preference setting (in the app section) to True, and restart psychopy. This will enable access to the test-suite. In debug mode, from the Coder (not Builder) you can now do Ctrl-T / Cmd-T (see Tools menu, Unit Testing) to bring up the unit test window. You can select a subset of tests to run, or run them all.

Its also possible to run just selected tests, such as doctests within a single file. From a terminal window:

```
cd psychopy/tests/  #eg /Users/jgray/code/psychopy/psychopy/tests
./run.py path/to/file_with_doctests.py
```

If the tests pass you hopefully havent damaged other parts of PsychoPy (!?). If possible add a unit test for your new feature too, so that if other people make changes they dont 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 youll 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 youve folded your new code back into your master and pushed it back to your github fork then its time to *Share your improvement with others*.

13.2 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 easy_install sphinx if its 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.

13.3 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 Components 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 doesnt mean anything to you then see *Using the repository*)

Youll 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.

13.3.1 1. Create the file defining the component: newcomp.py

Its 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 some of these methods, to override default behavior:

```
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*, *startType*, etc. Some of these fields may need to be overridden (e.g., *durationEstim* in *sound.py*). Inheriting from *VisualComponent* (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:

```
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):

```
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.)

Preliminary testing suggests that there are 600-ish names from numpy or numpy.random, plus the following:

```
['KeyResponse', '__builtins__', '__doc__', '__file__', '__name__', '__package__',
↪'buttons', 'core', 'data', 'dlg', '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 dont 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:

```
def writeInitCode(self,buff):
    # for debugging during Component development:
    buff.writeIndented("# self.params for aperture:\n")
    for p in self.params:
        try: buff.writeIndented("# %s: %s <type %s>\n" % (p, self.params[p].val, self.
↪params[p].valType))
        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.

13.3.2 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()s), you almost always want *%s % self.param[my_flag]*, without *.val*. Note that its 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 *componsTemplate.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.

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*.

13.3.3 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 Œ 48. Put it in the components directory.

In *newcomp.py*, have a line near the top:

```
iconFile = path.join(thisFolder, 'newcomp.png')
```

13.3.4 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*

13.4 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.
- Standardize the top stuff to have 1) a shebang with python, 2) utf-8 encoding, and 3) a comment:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

"""Demo name, purpose, description (1-2 sentences, although some demos need more␣
↪explanation).
"""
```

For the comment / description, its a good idea to read and be informed by the relevant parts of the API (see http://psychopy.org/api/api.html), but theres 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 dont 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 theres no user response and a timeout is appropriate for the demo (and a longer time-out might be needed, e.g., for *ratingScale.py*):

```
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, heres 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 its not strictly necessary

13.5 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. Youll need a strong understanding of PsychoPy, English, and the other language. 2) In some cases it will be necessary to adjust PsychoPys 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`.

13.5.1 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, youll 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 poedits 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 youve 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. Dont 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*).

13.5.2 Adjust PsychoPys 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 apps code to make it compatible with translations. If you are only making a translation, you can skip this section.

- In PsychoPys 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.

13.5.3 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).

13.6 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.

13.6.1 1. makeMenus()

The code that constructs the menus for the Builder is within a method named *makeMenus()*, within class builder.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 View menu, by editing the *makeMenus()* method of class *BuilderFrame* within *psychopy/app/builder/builder.py* (similar for Coder):

```
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).

13.6.2 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
```

Its possible that, instead of hard-coding it like this, its better to make a call to *wx.NewIdRef()* – wx will take care of avoiding duplicate IDs, presumably.

13.6.3 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.

13.6.4 4. Your new method

The second line within *makeMenus()* adds the key-binding definition into wxs 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).

13.6.5 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.

Happy Coding Folks!!

CHAPTER

FOURTEEN

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*.

14.1 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).

14.2 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.

14.3 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.

14.4 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*.

14.5 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.

14.6 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.

```
<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>
```

(continues on next page)

(continued from previous page)

```
    <Routine name="trial">
      <TextComponent name="word">
        <Param name="name" val="word" valType="code" updates="constant"/>
        <Param name="text" val="thisTrial.text" valType="code" updates="set every␣
↪repeat"/>
        <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="name" 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
↪"/>
        <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="name" val="instrText" valType="code" updates="constant"/>
        <Param name="text" val="&quot;Please press;&#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>
      <KeyboardComponent name="ready">
        <Param name="storeCorrect" val="False" valType="bool" updates="constant"/>
        <Param name="name" val="ready" 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"/
↪>
        <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">
```

(continues on next page)

(continued from previous page)

```
      <TextComponent name="thanksText">
        <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>
    <Routine name="instruct"/>
    <LoopInitiator loopType="TrialHandler" name="trials">
      <Param name="endPoints" val="[0, 1]" valType="num" updates="None"/>
      <Param name="name" val="trials" valType="code" updates="None"/>
      <Param name="loopType" val="random" valType="str" updates="None"/>
      <Param name="nReps" val="5" valType="num" updates="None"/>
      <Param name="trialList" val="[{'text': 'red', 'rgb': [1, -1, -1], 'congruent':␣
↪1, 'corrAns': 1}, {'text': 'red', 'rgb': [-1, 1, -1], 'congruent': 0, 'corrAns': 1},
↪ {'text': 'green', 'rgb': [-1, 1, -1], 'congruent': 1, 'corrAns': 2}, {'text':
↪'green', 'rgb': [-1, -1, 1], 'congruent': 0, 'corrAns': 2}, {'text': 'blue', 'rgb':␣
↪[-1, -1, 1], 'congruent': 1, 'corrAns': 3}, {'text': 'blue', 'rgb': [1, -1, -1],
↪'congruent': 0, 'corrAns': 3}]" valType="str" updates="None"/>
      <Param name="trialListFile" val="/Users/jwp...troop/trialTypes.csv" valType="str
↪" updates="None"/>
    </LoopInitiator>
    <Routine name="trial"/>
    <LoopTerminator name="trials"/>
    <Routine name="thanks"/>
  </Flow>
</PsychoPy2experiment>
```

PYTHON MODULE INDEX