Behav Res (2014) 46:913–921
DOI 10.3758/s13428-013-0422-2

PyGaze: An open-source, cross-platform toolbox for minimal-effort programming of eyetracking experiments

Edwin S. Dalmaijer · Sebastiaan Mathôt · Stefan Van der Stigchel

Published online: 21 November 2013

Abstract The PyGaze toolbox is an open-source software package for Python, a high-level programming language. It is designed for creating eyetracking experiments in Python syntax with the least possible effort, and it offers programming ease and script readability without constraining functionality and flexibility. PyGaze can be used for visual and auditory stimulus presentation; for response collection via keyboard, mouse, joystick, and other external hardware; and for the online detection of eye movements using a custom algorithm. A wide range of eyetrackers of different brands (EyeLink, SMI, and Tobii systems) are supported. The novelty of PyGaze lies in providing an easy-to-use layer on top of the many different software libraries that are required for implementing eyetracking experiments. Essentially, PyGaze is a software bridge for eyetracking research.

Keywords Eyetracking · Open-source · Software · Python · PsychoPy · Gaze contingency

Computers are an indispensable part of any (cognitive) neuroscientist's toolbox, not only for analysis purposes, but also for experiment presentation. Creating experiments has rapidly become easier over the past few years, especially with the introduction of graphical experiment builders (GEBs; Forster & Forster, 2003; Mathôt, Schreij, & Theeuwes, 2012; Peirce, 2007; Schneider, 1988; Stahl, 2006). These software packages provide users with a graphical interface to create experiments, a technique often referred to as "drag 'n drop" or "point 'n click." Although these tools increase productivity by decreasing the amount of time that a researcher has to invest in creating experiments, they are generally limited when it comes to complex experimental designs. In contrast, a programming language provides a researcher with almost unlimited flexibility, but requires considerable knowledge and skill.

In the present article, a new toolbox for creating eyetracking experiments using Python is introduced. The aim of the present project was to introduce the ease of GEBs into actual programming, using a mildly object-oriented approach. The result is PyGaze, a package that allows users to create experiments using short and readable code, without compromising flexibility. The package is largely platform and eyetracker independent, as it supports multiple operating systems, and eyetrackers of different manufacturers. In essence, the individual functionality of a number of existing Python libraries is combined within one package, making stimulus presentation and communication with multiple brands of eyetrackers possible using a unified set of routines. PyGaze contains functions for easy implementation of complex paradigms such as forced retinal locations, areas of interest, and other gaze-contingent experiments that can be created by obtaining and processing gaze samples in real time. These are notoriously difficult to implement using a GEB, although it is technically possible to use PyGaze scripting within a GEB (see the Usability section in the Results below).

Method

Python

Python (Van Rossum & Drake, 2011) is an interpreted programming language that does not need pre-compiling, but executes code by statement. For scientific use, a number of external packages, which are not part of the Python standard

E. S. Dalmaijer · S. Van der Stigchel
Experimental Psychology, Helmholtz Institute, Utrecht University, Utrecht, The Netherlands

S. Mathôt
Laboratoire de Psychologie Cognitive, Aix-Marseille Université & CNRS, Marseille, France

E. S. Dalmaijer (✉)
Department of Experimental Psychology, Utrecht University, Heidelberglaan 2, 3584 CS Utrecht, Netherlands
e-mail: e.s.dalmaijer@uu.nl

library, but could be regarded as "add-ons," are available. These include the NumPy and SciPy libraries (Oliphant, 2007) for scientific computing, Matplotlib (Hunter, 2007) for plotting, and PsychoPy (Peirce, 2007, 2009) for stimulus presentation.

With the addition of these packages, Python is a viable alternative to MATLAB (The Mathworks Inc.), a proprietary programming language that is widely used for scientific computing. In combination with the Psychophysics Toolbox (Brainard, 1997) and the Eyelink Toolbox (Cornelissen, Peters, & Palmer, 2002), MATLAB can be used for stimulus presentation and eyetracking using an EyeLink system (SR Research). Although both the Psychophysics and EyeLink toolboxes are freely available, MATLAB itself is expensive software, of which the source code is not available. Python, along with the aforementioned external packages, is completely open-source and might therefore be preferred over MATLAB.

It should be noted that PyGaze runs on Python 2.7, of which the most recent stable version at the time of writing stems from May 15, 2013. Although Python 3 is already available, and will have the focus of attention for future development, version 2 is still supported by the Python community. The reason PyGaze is based on version 2.7, is that most of the dependencies are not (yet) compatible with Python 3. It will be fairly straightforward to convert the PyGaze source to Python 3 once this becomes the standard.

Dependencies

For a complete eyetracking experiment, at least two external packages are required: one for communication with the eyetracker and one for experiment processes. The latter is either PsychoPy (Peirce, 2007, 2009) or PyGame, whereas the former depends on a user's preferred setup.

Both PyGame and PsychoPy are complete libraries for controlling computer displays, keyboards, mouses, joysticks, and other external devices, as well as internal timing. The main difference between the two is that PsychoPy supports hardware-accelerated graphics through OpenGL. In practice, this means that a great number of complicated stimuli, such as drifting Gabors, can be created within the time needed for a single frame refresh. In addition, PsychoPy retrieves millisecond-accurate information on the actual refresh time. This makes PsychoPy the package of choice for complex paradigms that require heavy processing or a high degree of temporal precision—for instance, dot motion displays. The drawback is that PsychoPy requires a graphics card that supports OpenGL drivers and multi-texturing (Peirce, 2007). This should not be a problem for most modern computers, but on some systems this functionality is not available—think of rather old computers or the Raspberry Pi. To provide support for these systems, non-OpenGL PyGame functions have been built into PyGaze as well. To switch between PyGame and PsychoPy, all a user has to do is change one constant.

Depending on a user's brand of choice, eyetracker communication is dealt with by custom libraries built on top of pylink (SR Research), the Tobii Software Development Kit, or the iViewX API, which is a part of the iViewX Software Development Kit by SensoMotoric Instruments. A dummy mode is available as well. It uses the mouse to simulate eye movements and requires no further external packages beyond either PyGame or PsychoPy. This means that PyGaze experiments can be developed and tested on a computer without an eyetracker attached, which is useful in labs where tracker time is scarce.

Although PsychoPy and PyGame are excellent for creating experiments, using them in combination with an eyetracker requires additional external libraries, not to mention additional effort. Pylink, the Tobii SDK, and the iViewX API are relatively difficult to use for novice programmers and scripts that use these APIs directly are often complicated. PyGaze acts as a wrapper for all of the aforementioned libraries.

For novice Python users, it might prove difficult to find and install all of the necessary packages. Therefore, a full list of the dependencies and installation instructions, as well as a complete Python distribution for Windows are available from the PyGaze website.

Hardware requirements

PyGaze has been developed and tested on a range of Windows versions (2000, XP, and 7). Additional tests have been performed on Mac OS X (Snow Leopard) and Linux (Ubuntu 12.04, Debian "wheezy," and Raspbian). Since PyGaze is written solely in Python and uses no compiled code of its own, its portability depends whether all dependencies are available on a specific system.

The two main dependencies for stimulus presentation, PyGame and PsychoPy, each come with different hardware requirements. PsychoPy requires a graphics card that supports OpenGL drivers and multitexturing (for details, see Peirce, 2007), whereas PyGame runs on practically any computer. The libraries used to communicate with EyeLink and Tobii devices are available on a broad range of operating systems, including most Windows, OS X, and Linux versions. Regrettably, SMI's iView X SDK is compatible with Windows XP, Vista, and 7 (32- and 64-bit) only. In sum, PyGaze is versatile and compatible with a large variety of systems, albeit on certain systems only a subset of functionality is available.

Results

Usability

The PyGaze package consists of a number of libraries (modules, in Python terminology) that contain several object

definitions (classes). The advantage of working with objects, which are specific instances of a class, is that script length is reduced and script readability is enhanced.

The philosophy of object oriented programming (OOP) is that a programmer should only solve a particular, complicated problem once. A class consists of properties (variables) and methods (functions) that contain the code to deal with a certain problem. An example of a class in PyGaze is the EyeTracker class, which contains high-level methods to start calibration, retrieve gaze position, and so forth. After constructing the necessary classes, a programmer can introduce these into a script without having to deal with the inside workings of the class, so that the programmer can focus on the overview and logic of the experiment. For example, in PyGaze, the programmer can call the calibration routine of an EyeTracker object, without being concerned with the details of how calibration is performed on a particular system. This approach makes a lot of sense in real life: A car company does not reinvent the wheel every time a new car is developed. Rather, cars are built using a number of existing objects (among which the wheel) and new parts are developed only when necessary. The same approach makes as much sense in programming as it does in the real world.

Another advantage of OOP is that scripts become more compact and require less typing. To illustrate this point: A regular Python script for initializing and calibrating an EyeLink system contains over 50 lines of code when the pylink library is used, whereas the same could be achieved with two lines of code when using PyGaze (as is illustrated in Listing 1, lines 15 and 24).

More advanced Python users will find it easy to incorporate PyGaze classes into their own scripts to use functions from PyGame, PsychoPy or any other external package, or even create additional libraries for PyGaze. Code samples for this kind of approach—for example, for using PsychoPy's GratingStim class on a PyGaze Screen object—are available on the PyGaze website. As a consequence of this flexibility, PyGaze might even be used within GEBs, for example by using OpenSesame's (Mathôt et al., 2012) Python inline scripting possibilities. This is useful for researchers that do want to harness PyGaze's capabilities, but have a personal preference for using a graphical environment over scripting.

Basic functionality

To display visual stimuli, Display and Screen classes are provided. Screen objects should be viewed as blank sheets on which a user draws stimuli. Functions are provided for drawing lines, rectangles, circles, ellipses, polygons, fixation marks, text, and images. The Display object contains the information that is to be shown on the computer monitor and can be filled with a Screen object. After this, the monitor is updated by showing the Display. See Listing 1, lines 11–12 and 28–34 for a code example. Custom code that is written using PsychoPy or PyGame functions can be used as well.

```
# imports
from constants import *
from pygaze import libtime
from pygaze.libscreen import Display, Screen
from pygaze.eyetracker import EyeTracker
from pygaze.libinput import Keyboard
from pygaze.liblog import Logfile
from pygaze.libgazecon import FRL

# visuals
disp = Display(disptype='psychopy', dispsize=(1024,768))
scr = Screen(disptype='psychopy', dispsize=(1024,768))

# eye tracking
tracker = EyeTracker(disp)
frl = FRL(pos='center', dist=125, size=200)

# input collection and storage
kb = Keyboard(keylist=['escape','space'], timeout=None)
log = Logfile()
log.write(["trialnr", "trialstart", "trialend", "image"])

# calibrate eye tracker

tracker.calibrate()

# run trials
for trialnr in range(len(IMAGES)):
	# blank display
	disp.fill()
	disp.show()
	libtime.pause(1000)
	# prepare stimulus
	scr.clear()
	scr.draw_image(IMAGES[trialnr])
	# start recording gaze data
	tracker.drift_correction()
	tracker.start_recording()
	tracker.status_msg("trial %d" % trialnr)
	tracker.log("start trial %d" % trialnr)
	# present stimulus
	response = None
	trialstart = libtime.get_time()
	while not response:
		gazepos = tracker.sample()
		frl.update(disp, scr, gazepos)
		response, presstime = kb.get_key(timeout=1)
	# stop tracking and process input
	tracker.stop_recording()
	tracker.log("stop trial %d" % trialnr)
	log.write([trialnr, trialstart, presstime, IMAGES[trialnr]])

# close experiment
log.close()
tracker.close()
disp.close()
libtime.expend()
```

Listing 1 Code example of a PyGaze experiment script that records eye movements while showing images that are obscured outside of a small cutout around a participant's gaze position. The full experiment, including a short script for the

constants and the website screenshots referred to as IMAGES, is provided on the PyGaze website.

Using the Sound class, it is possible to play sounds from sound files and to create sine, square, and saw waves, as well as white noise. With the Keyboard, Mouse, and Joystick classes, a user can collect input (see Listing 1, lines 19 and 46). A Logfile object is used to store variables in a text file, in which values are tab-separated (see Listing 1, lines 20, 21 and 50). Eyetrackers can be controlled using the EyeTracker class (see Listing 1, lines 15, 24, 36–39, 44, 48–49, and 54).

Apart from this basic functionality, PyGaze comes with a number of classes for more advanced, gaze-contingent functionality. At the moment of writing, the available functionality is for forced retinal locations (used in Lingnau, Schwarzbach, & Vorberg, 2008, 2010), gaze-contingent cursors and areas of interest (AOIs). A code implementation of a forced retinal location (FRL) paradigm is provided in listing 1 (see lines 16 and 45). The AOI class provides a method to check if a gaze position is within a certain area (rectangle, ellipse or circle shaped). The aim for this is to provide users with a readymade way to check whether a subject is looking at a certain stimulus, allowing for direct interaction with the display. Further paradigms may be implemented in the future by the developers, but could be created by users as well, using the EyeTracker class's sample method.

The classes referred to in this paragraph do require some settings—for example, to set the eyetracker brand and display size. The default settings are stored in a single file within the PyGaze package that can be adjusted by the user. Another option, which does not require readjusting the defaults for every new experiment, is adding a constants file to each new experiment or hard coding the constants within the experiment script.

Support for multiple eyetracking brands

PyGaze currently is compatible with SR Research's EyeLink systems, all SMI products that run via iViewX, and Tobii devices, as long as the software for these systems is installed. This software is usually provided along with the eyetrackers, together with installation instructions. The classes for EyeLink, SMI, and Tobii use the same methods (albeit with different inner workings), meaning that a PyGaze script can be used for all three types of systems, without having to adjust the code.

Data storage and communication with an eyetracker are handled by software provided by the manufacturers (i.e., their software development kits, abbreviated SDKs). Therefore, gaze-data collection is always performed as intended by the manufacturer. There are differences in how PyGaze works between manufacturers, and even between eyetrackers. Some eyetrackers use a second computer to gather gaze data (e.g., EyeLink), whereas others work via a parallel process on the same computer that runs the experiment (e.g., Tobii and the SMI RED-m). A consequence of these differences is that gaze data is not stored in a single manner: EyeLink devices produce an EDF file, SMI devices an IDF file, and Tobii data is stored in a simple text file (with a .txt extension). PyGaze does include a Logfile class for creating and writing to a text file, which can be used to store e.g., trial information (trial number, stimulus type, condition, etc.) and response data (key name, response time, etc.). See Listing 1, lines 20, 21, and 50, for an example on the Logfile class. There is no automatic synchronization between both types of data files, or with the monitor refresh. However, this can easily be implemented by using the EyeTracker class's log method. This method allows a user to include any string in the gaze data file—for example, directly after a call to the Display class's show method, to pass a string containing the display refresh time (similar to Listing 1, line 39, where the trial number is written to the gaze data file).

Although the same PyGaze code works for these three types of systems, there are differences in the ways that PyGaze works between the three. These include the obvious difference between the GUIs and setup of the EyeLink, iViewX, and Tobii controllers, as well as more subtle dissimilarities in the software created by the manufacturers. An example is the lack of event detection functions in the iViewX API and the Tobii SDK. To compensate for this, algorithms for online event detection were developed. These are described below.

Online saccade detection algorithm

In a gaze-contingent paradigm, the experimental flow is dependent on a participant's eye movement behavior. For this purpose, it is important that a toolbox for creating eyetracking experiments provides means to assess this behavior in real time. Apart from information on the current gaze position, information on certain events could be useful as well. For example, online saccade detection is required in an experiment in which a target shifts in position after a saccade has been initiated (Sharika et al., 2013). Some producers of eyetrackers (e.g., SR Research) offer functions to detect events online, but not all libraries offer this kind of functionality. To compensate for this, custom event detection algorithms are implemented in PyGaze, of which the most elaborate is the detection of saccades. By specifying the value of a single constant, users can select either the PyGaze event detection or the event detection that is native to the underlying library (if this is provided by the manufacturer of their eyetracker of choice).

The PyGaze algorithm for online saccade detection resembles the Kliegl algorithm for (micro)saccade detection (Engbert & Kliegl, 2003) in that it identifies saccades by calculating eye movement velocity on the basis of multiple samples. However, since the present algorithm is developed for online saccade detection, it should detect events as soon as possible. Therefore, eye movement velocity is calculated using the smallest possible number of samples, which is two: the newest and the previous sample. By comparison,

the Kliegl algorithm "looks ahead" two samples and uses a total of five samples. In addition, the present algorithm takes into account eye movement acceleration as a saccade indicator. Note that the acceleration in the previous sample is based on the speed in that sample and in its preceding sample, and therefore the sample window for acceleration calculation is actually 3. The exact process is described below. This algorithm has been designed with speed and responsiveness in mind, and is consequently less reliable than more advanced algorithm for offline event detection. Therefore, researchers are advised to analyze the raw eye-movement data using an algorithm that is designed for offline event detection—for example, those described by Engbert and Kliegl (2003) or Nyström and Holmqvist (2010)—as one would normally do when analyzing eye movement data.

After calibration, parameters for the horizontal and vertical precision for both the left and right eye are derived through computation of the RMS noise on the basis of a collection of samples obtained continuously during a short central fixation. Second, user-defined or default values for saccade speed (in deg/s) and acceleration (in deg/s^2) thresholds are transformed to values in pixels per second and pixels per second^2, respectively. This transformation is done on the basis of information on the physical display size in centimeters and the display resolution (defined by the user) and information on the distance between the participant and the display, which is either supplied by the user, or obtained through the iViewX API that allows for estimating the distance between the eyetracker and the participant.

During the detection of saccade starts, samples are continuously obtained. For each new sample, the distance from the previous sample is obtained (see Fig. 1). Equation 1 is based on the formula for an ellipse in a Cartesian grid and produces the weighted distance between samples. It is used to check whether this distance is larger than the maximal measurement-error distance, on the basis of the obtained precision values. If the outcome for this formula is greater than the precision threshold, the distance between samples is higher than the maximal measurement error and thus the distance between the current sample and the previous sample is likely due to factors other than measurement error. In this case, the threshold is set to one, but higher values may be used for a more conservative approach. The speed of the movement between these two samples expressed in pixels per second is equal to the distance between both samples in pixels divided by the time that elapsed between obtaining the samples. The acceleration expressed in pixels per second^2 is calculated by subtracting the speed in the previous sample from the current speed. If either the current speed or the current acceleration exceeds the corresponding threshold value, a saccade is detected.

$$\left(\frac{sx}{tx}\right)^2 + \left(\frac{sy}{ty}\right)^2 > 1 \qquad (1)$$

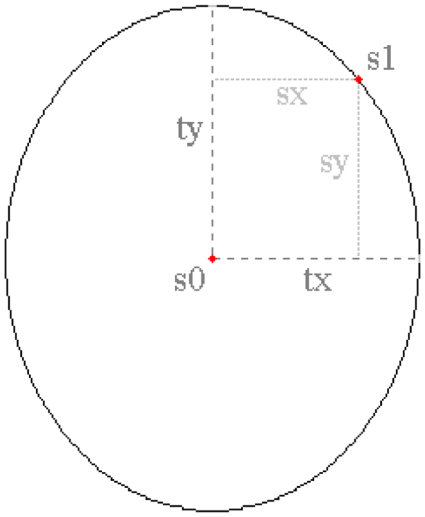

Fig. 1 Two gaze position samples s0 and s1, with their corresponding horizontal and vertical intersample distances sx and sy; tx and ty represent the horizontal and vertical thresholds for RMS noise

$$tx = \sqrt{\frac{\sum_{i=2}^{n}(X_i - X_{i-1})^2}{n-1}} \text{ and } ty = \sqrt{\frac{\sum_{i=2}^{n}(Y_i - Y_{i-1})^2}{n-1}} \qquad (2)$$

sx	horizontal distance between current and previous sample
sy	vertical distance between current and previous sample
tx	threshold for the horizontal distance, based on the horizontal precision
ty	threshold for the vertical distance, based on the horizontal precision
X	x value of a sample
Y	y value of a sample
n	amount of samples
i	index number of a sample

The detection of saccade endings is very similar to that of saccade starts, with the obvious difference that a saccade end is detected if both the eye movement speed and acceleration fall *below* the corresponding thresholds. The precision values do not play a role in saccade ending detection, since their only use in the current saccade detection is to prevent differences in sample position due to measurement errors to be mistaken for actual saccade starts.

To complement libraries that do not provide means for blink detection, a custom blink detection has been implemented as well. Blinks are detected when during a period of at least 150 ms, no gaze position can be derived. At a low sampling frequency of 60 Hz, this equals nine consecutive samples, reducing the chance of a false positive blink detection when a small amount of samples would be dropped for reasons other than a blink.

Availability

PyGaze is freely available via www.fss.uu.nl/psn/pygaze/. Documentation and instructions on downloading all the relevant dependencies can be found there as well. The source code of PyGaze is accessible on GitHub, a website that allows for easily following the "best practices for scientific computing" as formulated by Wilson et al. (2012)—that is, programming in small steps with frequent feedback, version control and collaboration on a large scale. Theoretically, every individual with access to the internet and the relevant knowledge could do a code review or contribute to the project.

As Peirce (2007) has pointed out, open-source software for visual neuroscience offers considerable advantages over proprietary software, since it is free and its users are able to determine the exact inner workings of their tools. Open-source software may be freely used and modified, not only by researchers, but by companies that could benefit from eyetracking as well; it is a way of giving back scientific work to the public. For this reason, PyGaze is released under the GNU General Public License (version 3; Free Software Foundation, 2007), which ensures that users are free to use, share, and modify their version of PyGaze.

Place among existing software

The aim of the present project is to provide an umbrella for all of the existing Python packages that are useful in eyetracking software, unifying their functionality within one interface. As compared to current alternatives, PyGaze provides a more user-friendly and less time-consuming way to harness the functionality of its underlying packages.

Most, if not all, of the existing software for Python could be integrated within PyGaze by programmers with reasonable experience in Python. Among these are the Python APIs for different brands of eyetrackers (e.g., Interactive Minds) as well as other stimulus generators (e.g., VisionEgg; Straw, 2008). An interesting option to explore for researchers in the open-source community is the use of a (Web) camera with infrared illumination and the ITU Gaze Tracker (San Agustin et al., 2010; San Agustin, Skovsgaard, Hansen, & Hansen, 2009) software in combination with PyGaze. Although an out-of-the-box library for this system does not (yet) exist, it should not be difficult to implement. The same principal applies to a combination of PyGaze and GazeParser (Sogo, 2013), another open-source library for video-based eyetracking, for both data gathering and analysis. These options allow for easy and low-cost eyetracking that is completely open-source from stimulus presentation to analysis.

Benchmark experiment

A benchmark experiment was conducted to test the temporal precision (defined as the lack of variation) and accuracy (defined as the closeness to true values) of display presentation with PyGaze. A photodiode was placed against a monitor that alternately presented black and white displays. This photodiode was triggered, for all intents and purposes instantaneously, by the luminosity of the white display. The response time (RT) of the photodiode to the presentation of the white display ($RT = T_{resp} - T_{display}$) was used as a measure of the temporal accuracy of display presentation. Ideally, the RT should be 0 ms, indicating that the moment at which the white display actually appears matches the display timestamp. In practice, various sources of error will lead to an RT that is higher than 0 ms, but this error should be constant and low.

The experiment was conducted on a desktop system (HP Compaq dc7900, Intel Core 2 Quad Q9400, 2.66 Ghz, 3Gb) connected to a CRT monitor (21-in. ViewSonic P227f) running Windows XP. The experiment was repeated with photodiode placement at the top left and the bottom right of the monitor, and using PsychoPy as well as PyGame. $N = 1{,}000$ for each test.

Figure 2a shows the results when PsychoPy was used and the photodiode was held to the top left of the monitor. Here, the observed RT distribution was clearly bimodal, consisting of fast ($M = 0.27$ ms, $SD = 0.01$) and slower ($M = 10.28$ ms, $SD = 0.02$) responses. The reason for this bimodality is that the photodiode was polled only after the white display had been presented. Because we used a CRT monitor, which uses a phasic refresh,[1] the white display started to fade immediately after presentation, and therefore frequently failed to trigger the photodiode, which was calibrated to respond to the peak luminance of the monitor. When this happened, the photodiode was invariably triggered on the next refresh cycle (i.e., after 10 ms on a 100-Hz display). The point to note here is that the RTs are extremely constant and accurately reflect the physical properties of the display, which shows that the display timestamps provided by PyGaze/PsychoPy are very accurate. Holding the photodiode to the bottom right of the display served as a sanity check (Fig. 2b). Because monitors are refreshed from the top down, there was a delay between the moment that the white display came on and the moment that the refresh reached the bottom of the monitor. This delay was just short of one refresh cycle. Therefore, when holding the photodiode to the bottom right, a consistent response on the first refresh cycle, with an RT that is just short of 10 ms, was expected. This prediction was borne out by the results ($M = 8.70$ ms, $SD = 0.06$), confirming that PyGaze provides highly accurate display timestamps when using PsychoPy.

Higher and more variable RTs were expected when using PyGame (Fig. 2c and d), which is known to offer less temporal precision (Mathôt et al., 2012). Indeed, RTs were relatively high and variable. Holding the photodiode to the bottom right

[1] A video showing the difference between the refreshing of TFT and CRT monitors can be found here: https://vimeo.com/24216910.

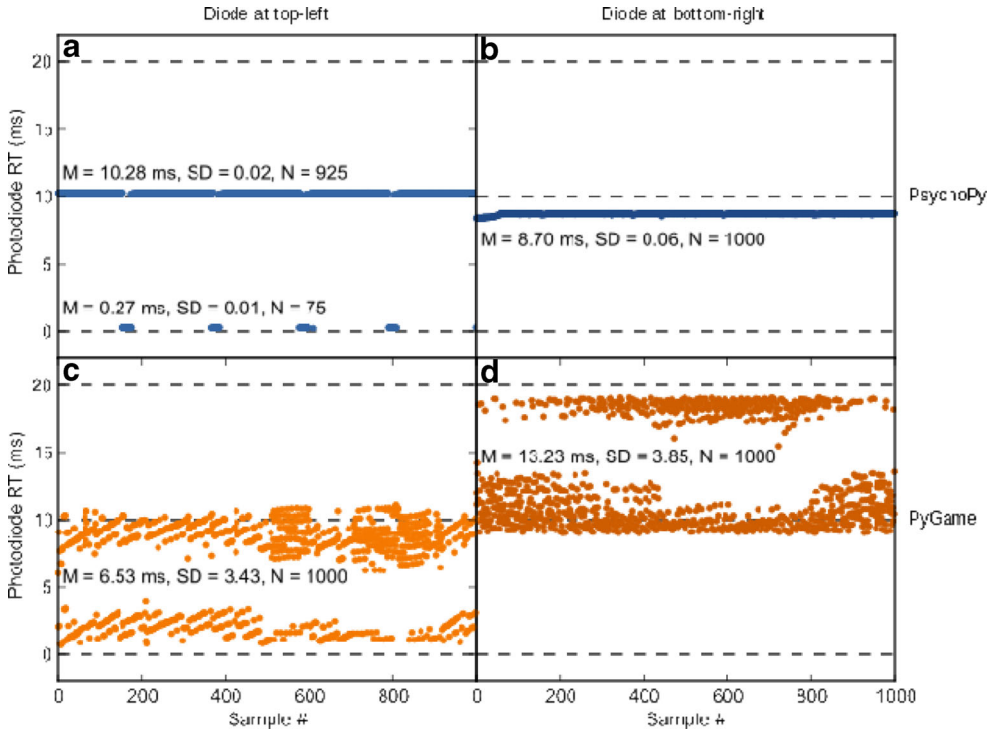

Fig. 2 Results of a benchmark experiment for temporal precision and accuracy of the display presentation times obtained with PyGaze, using either PsychoPy or PyGame

of the monitor shifted the RT distribution (M = 13.23 ms, SD = 3.85), but did not alter the basic pattern, as compared to when the photodiode was held to the top left (M = 6.52 ms, SD = 3.43), as expected.

In our experience, these results generalize to most modern systems. But one should always keep in mind that a wide variety of factors can interfere with timing, and that it is recommended to test an experimental setup when temporal accuracy is important. This is especially true for gaze-contingent experiments in which the computer display should respond to gaze behavior with as little delay as possible.

To assess how quickly new samples could be obtained using the sample method from the EyeTracker class, a second benchmark test was performed on three different setups. Each setup used an eyetracker of a different brand: an EyeLink 1000, an SMI RED-m, and a Tobii TX300 (for details, see Table 1). The benchmark test was a short but full PyGaze script that initialized an experiment (displaying to a monitor,

Table 1 Results for a benchmark test to assess sampling speed without further processing (no display) and sampling speed in a gaze-contingent experiment (display), using both the PsychoPy and PyGame display types, for three different types of eyetrackers

		PsychoPy		PyGame	
		No Display	Display	No Display	Display
EyeLink 1000[a]	IST	0.018 (0.003)	16.661 (0.096)	0.015 (0.002)	16.652 (0.272)
	dropped	0	0	0	0
SMI RED-m[b]	IST	0.003 (0.001)	16.664 (0.113)	0.003 (0.001)	11.774 (0.914)
	dropped	0	0	0	0
Tobii TX300[c]	IST	0.003 (0.001)	16.714 (1.371)	0.003 (0.003)	23.887 (2.067)
	dropped	0	3	0	1,000

The displayed values are means (standard deviation appear between parentheses) of 1,000 intersample times (IST), specified in milliseconds. The number of dropped frames (where the intersample time surpassed the display refresh time) is displayed below the mean intersample time. The monitor refresh cycle duration for each setup was 16.667 ms. [a] desktop: Dell Precision PWS 390, Intel Core 2 6600, 2.40 GHz, 3 GB, Windows XP; monitor: Philips Brilliance 202P7, 1,024 × 768, 60 Hz. [b] laptop: Clevo W150ER, Intel Core i7-3610QM, 2.30 GHz, 16 GB, Windows 7; monitor: LG-Philips LP156WF1 (built-in), 1,920 × 1,080, 60 Hz. [c] desktop: custom build, Intel Core 2 4300, 1.80 GHz, 2 GB, Windows XP; monitor: Tobii TX Display, 1,280 × 1,024, 60 Hz

preparing a keyboard for response input, and starting communication with an eyetracker), calibrated an eyetracker, and consecutively called the EyeTracker class's sample method for 1,001 samples. In a second version, a dot was displayed at gaze position directly after obtaining each sample. After calling the sample method, a timestamp was obtained using the get_time function from PyGaze's time library (which is based on either PsychoPy or PyGame timing functions, depending on the display type). The intersample time was calculated for all samples, resulting in 1,000 intersample times per measurement. Although the setups differ, the software environment was kept constant. The experiment scripts were always run from an external hard drive, using the portable Python distribution for Windows (mentioned under Dependencies in the Method section of this article). The results of the second benchmark are summarized in Table 1.

The results show that the sample method can be called consecutively at a high pace, ranging from 50 000 to over 300 000 Hz.[2] This is *well* under the refresh rate of all currently existing monitors (most run at refresh rates of 60 or 100 Hz), and allows for almost all of the time within a single refresh to be used on drawing operations. This is reflected in the intersample times produced using a PsychoPy display type. Since PsychoPy waits for the vertical refresh before a timestamp is recorded, intersample times are very close to the duration of one monitor refresh cycle, which indicates that drawing operations are completed within a single refresh. PyGame does not wait for the next vertical refresh, meaning that after sending the update information to the monitor, it runs without delay. Therefore, the intersample time reflects the iteration time of the part of the script that obtains a sample and a timestamp, clears the old screen information, draws a new dot, and sends the new display information to the monitor. The lack of deviation in the intersample times indicates a consistent duration of the process of displaying the gaze contingent dot. The relatively high intersample time obtained when using a PyGame display type together with the Tobii TX300 is curious, and it might be explained by a lack of general processing capacity for that particular setup (PyGame uses the CPU for drawing operations, whereas PsychoPy uses the GPU).

Other latencies will occur outside of PyGaze software. However, this *system latency* is produced by setup-specific sources outside of PyGaze's influence—for example, communication delays between an eyetracker and a computer, or between a computer and a monitor. Since the system latency will differ from setup to setup, researchers are advised to benchmark their own system. An excellent method for measuring the total system latency of gaze contingent displays is described by Saunders and Woods (2013).

Discussion

Although many options are available for creating experiments for eyetracking research, these can be divided into two categories: those that require (advanced) programming skills and those that do not, but offer limited functionality. There is no denying that excellent and very complicated paradigms can be implemented using programming languages as C, MATLAB, and Python, but this requires advanced programming skills.

Those who do not have these programming skills may turn to graphical experiment builders (GEBs), such as Presentation (Neurobehavioural Systems Inc.), E-Prime (Psychology Software Tools), Experiment Builder (SR Research), and OpenSesame (Mathôt et al., 2012). However, configuring E-Prime and Presentation to work with an eyetracker requires additional scripting. Most of the required scripting is provided by the producer of either the software or the eyetracker, but it lacks the intuitiveness that GEBs advertise with. Experiment Builder provides a more user-friendly way of creating eyetracking experiments, but is limited to the EyeLink platform and proprietary software. Arguably, OpenSesame provides the most intuitive and broad platform for the development of eye-movement experiments, particularly for relatively simple designs. Although it is possible to create more elaborate paradigms (e.g., a reading task with a forced retinal location) in OpenSesame, this requires additional Python scripting.

Using Python syntax outside of a GEB allows for a less complicated experiment structure than using code within a GEB (see also Krause & Lindemann, 2013). An experiment created with a code editor is located within a single script, whereas an experiment created with a GEB is divided over GUI elements and several inline scripts. PyGaze provides means for creating experiments in a single script, using functions that allow for quicker and clearer programming than would have been the case using the aforementioned packages. Therefore, PyGaze is potentially a valuable resource for every researcher that uses eyetrackers and is familiar with Python, or is willing to invest some time in learning the basics of Python.

In conclusion, PyGaze fills the gap between complicated and time-consuming programming and restrictive graphical experiment builders by combining the flexibility of the former with the user-friendliness and comprehensibility of the latter. It provides an ideal package for creating eyetracking and other neuroscientific experiments.

[2] None of the currently available eyetrackers generate new samples at this pace. The sample method provides the *most recent* sample, which is not necessarily a *newly obtained* sample.

Author note Many thanks to Richard Bethlehem for his help with testing, to Ignace Hooge for his advice on saccade detection, and to Daniel Schreij and Wouter Kruijne for their contributions to the EyeLink code. S.M. was funded by ERC Grant No. 230313 to Jonathan Grainger.

References

Brainard, D. H. (1997). The Psychophysics Toolbox. *Spatial Vision, 10,* 433–436. doi:10.1163/156856897X00357

Cornelissen, F. W., Peters, E. M., & Palmer, J. (2002). The Eyelink Toolbox: Eye tracking with MATLAB and the Psychophysics Toolbox. *Behavior Research Methods, Instruments, & Computers, 34,* 613–617. doi:10.3758/BF03195489

Engbert, R., & Kliegl, R. (2003). Microsaccades uncover the orientation of covert attention. *Vision Research, 43,* 1035–1045. doi:10.1016/S0042-6989(03)00084-1

Forster, K. I., & Forster, J. C. (2003). DMDX: A Windows display program with millisecond accuracy. *Behavior Research Methods, Instruments, & Computers, 35,* 116–124. doi:10.3758/BF03195503

Free Software Foundation. (2007). GNU General Public License. *The GNU General Public License v3.0 - GNU Project - Free Software Foundation (FSF)*. Retrieved July 28, 2013, from https://gnu.org/licenses/gpl.html

Hunter, J. D. (2007). Matplotlib: A 2D Graphics Environment. *Computing in Science & Engineering, 9,* 90–95. doi:10.1109/MCSE.2007.55

Krause, F., & Lindemann, O. (2013). Expyriment: A Python library for cognitive and neuroscientific experiments. *Behavior Research Methods*. doi:10.3758/s13428-013-0390-6

Lingnau, A., Schwarzbach, J., & Vorberg, D. (2008). Adaptive strategies for reading with a forced retinal location. *Journal of Vision*, *8*(5):6, 1–18. doi:10.1167/8.5.6

Lingnau, A., Schwarzbach, J., & Vorberg, D. (2010). (Un-)coupling gaze and attention outside central vision. *Journal of Vision, 10*(11), 13. doi:10.1167/10.11.13

Mathôt, S., Schreij, D., & Theeuwes, J. (2012). OpenSesame: An open-source, graphical experiment builder for the social sciences. *Behavior Research Methods, 44,* 314–324. doi:10.3758/s13428-011-0168-7

Nyström, M., & Holmqvist, K. (2010). An adaptive algorithm for fixation, saccade, and glissade detection in eyetracking data. *Behavior Research Methods, 42,* 188–204. doi:10.3758/BRM.42.1.188

Oliphant, T. E. (2007). Python for Scientific Computing. *Computing in Science & Engineering, 9,* 10–20. doi:10.1109/MCSE.2007.58

Peirce, J. W. (2007). PsychoPy—Psychophysics software in Python. *Journal of Neuroscience Methods, 162,* 8–13. doi:10.1016/j.jneumeth.2006.11.017

Peirce, J. W. (2009). Generating stimuli for neuroscience using PsychoPy. *Frontiers in Neuroinformatics, 2,* 10. doi:10.3389/neuro.11.010.2008

San Agustin, J., Skovsgaard, H., Hansen, J. P., & Hansen, D. W. (2009). Low-cost gaze interaction: ready to deliver the promises. In *Proceedings of the 27th international Conference Extended Abstracts on Human Factors in Computing Systems* (pp. 4453–4458). New York, NY: ACM Press. doi:10.1145/1520340.1520682

San Agustin, J., Skovsgaard, H., Mollenbach, E., Barret, M., Tall, M., Hansen, D. W., & Hansen, J. P. (2010). Evaluation of a low-cost open-source gaze tracker. In *Proceedings of the 2010 Symposium on Eye-Tracking Research and Applications* (pp. 77–80). New York, NY: ACM Press. doi:10.1145/1743666.1743685

Saunders, D. R., & Woods, R. L. (2013). Direct measurement of the system latency of gaze-contingent displays. *Behavior Research Methods*. doi:10.3758/s13428-013-0375-5

Schneider, W. (1988). Micro Experimental Laboratory: An integrated system for IBM PC compatibles. *Behavior Research Methods, Instruments, & Computers, 20,* 206–217. doi:10.3758/BF03203833

Sharika, K. M., Neggers, S. F. W., Gutteling, T. P., Van der Stigchel, S., Dijkerman, H. C., & Murthy, A. (2013). Proactive control of sequential saccades in the human supplementary eye field. *Proceedings of the National Academy of Sciences, 110,* E1311–E1320. doi:10.1073/pnas.1210492110

Sogo, H. (2013). GazeParser: an open-source and multiplatform library for low-cost eye tracking and analysis. *Behavior Research Methods, 45,* 684–695. doi:10.3758/s13428-012-0286-x

Stahl, C. (2006). Software for generating psychological experiments. *Experimental Psychology, 53,* 218–232. doi:10.1027/1618-3169.53.3.218

Straw, A. D. (2008). Vision Egg: An open-source library for realtime visual stimulus generation. *Frontiers in Neuroinformatics, 2,* 4. doi:10.3389/neuro.11.004.2008

Van Rossum, G., & Drake, F. L. (2011). *Python Language reference manual*. Bristol, UK: Network Theory Ltd.

Wilson, G., Aruliah, D. A., Brown, C. T., Chue Hong, N. P., Davis, M., Guy, R. T., . . . Wilson, P. (2012). Best practices for scientific computing. *arXiv*, 1210.0530v3. Retrieved January 20, 2013, from http://arxiv.org/abs/1210.0530v3