Chapter 4 Getting to know Pylink

A brief introduction to eye-tracking

Put your thumb on this page, keep looking at the thumbnail and you will see that words on either side of the thumb are rather challenging to read. Why? Because most light-sensitive cells (cones) concentrate in a tiny area on the retina, known as the fovea, and visual acuity reduces exponentially as the distance from the fovea increases. The fovea corresponds to about 1-2 degrees of visual angle. One degree is about the size of a thumbnail at arms-length. Because of this limitation of the retina, the eyes need to quickly rotate and change the direction of gaze around 2-3 times a second. These rapid movements of the eyes typically last for about 15-100 milliseconds, and are known as “saccades”. Between saccades are periods during which the eyes remain relatively still, so processing of visual details can take place. These periods of relative stability are known as “fixations”.

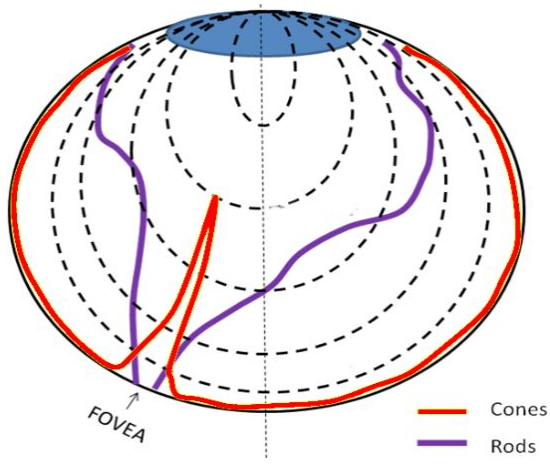
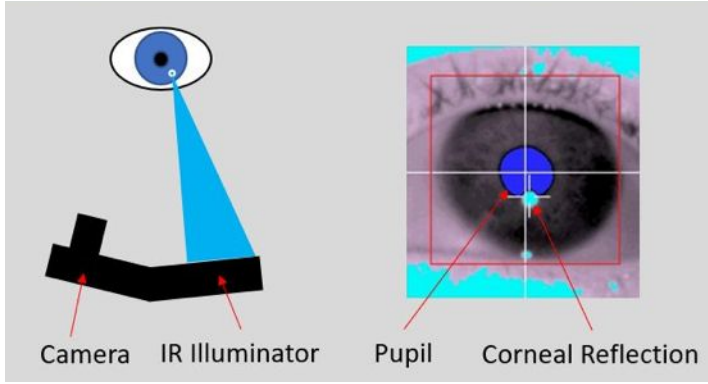


Figure 4-1 *The distribution of cones and rods on the retina (adapted from WikiMedia[1]).*

Since the 1870s, researchers have developed a wide range of methods to record eye movements, including both electromagnetic and optical techniques. Currently, most commercial eye-tracking devices have adopted a video-based solution, which has the advantages of being simple to use and delivering the high levels of accuracy and precision required for laboratory research.

The working principle of a video-based eye-tracker is rather straightforward. An infrared light source illuminates the eye area, and a camera images the eyes at extremely high speed (e.g., 1000 Hz). The camera lens filters out the visible light to obtain a clear image of the eyes in the infrared spectrum. In addition to illuminating the eyes, the infrared light source creates a bright "glint" on the cornea, known as the corneal reflection, or CR for short. When the eyes rotate, the position of the pupil changes relative to the CR. Image analysis software computes the vector between the center of the pupil and CR (Pupil minus CR) in the camera sensor coordinates. This vector changes when the eye rotates, but not when the head moves. As a result, small head movements (within one cubic inch space; see Merchant et al., 1974), can be tolerated, and it is not necessary to use a bite bar to keep the head perfectly still.



[1] Licensed to use in the public domain, https://commons.wikimedia.org/wiki/File:Density_rods_n_cones.png

Figure 4-2 *The working principle of Pupil-CR tracking. Adapted from a blog by Sam Hutton (with permission).*[2]

A calibration process creates a mapping function between the raw Pupil-CR data (which is based on the camera sensor pixel units) and the spatial location the subject is currently fixating (e.g., pixel the on screen). The result of the calibration is a complex regression model derived from the raw Pupil-CR values for a set of calibration targets that have known locations in space. The regression model allows the eye tracker to estimate where the participant is looking at (i.e., gaze position).

SR Research EyeLink® eye trackers are widely used in a range of research settings, including vision science. The use of the tracker typically involves launching an experimental script on a stimulus presentation PC (or Display PC, in EyeLink terminology), adjusting the camera for optimal tracking, calibrating the tracker, evaluating the calibration results, and then starting the task to record eye movement data. The operation of the tracker is straightforward, and I would recommend users to go through the short video tutorials available on the SR Support Forum and the EyeLink User Manual for detailed guidelines and instructions.

The EyeLink trackers come with an API (also known as the EyeLink Developer's Kit) that allows users to develop custom eye-tracking applications. The Python wrapper of this API is the Pylink library, and this chapter will focus on the fundamental functions available in this Python library. The Pylink library is compatible with both Python 2 and 3. The example scripts included in this book have been tested with Python 3.6.6, but they should also work in more recent versions of Python.

Install Pylink

To use Pylink, you need first to install the latest version of the EyeLink Developer's Kit, which works on all major platforms (Windows, macOS, and Ubuntu). Windows and macOS installers for the EyeLink Developer's Kit are freely available from the SR Support Forum (https://sr-support.com). Once the

[2] https://www.sr-research.com/eye-tracking-blog/how-does-eye-tracking-work/

Developer's Kit is installed, you will find the latest version of Pylink in the installation folder:

Windows: C:\Program Files (x86)\SR Research\EyeLink\SampleExperiments\Python
macOS: /Applications/EyeLink/pylink

These folders contain multiple versions of the Pylink library; please rename the "pylink*" folder that matches your Python version to "pylink" and copy it to the Python "site-packages" folder.

Windows: C:\Users\{Username}\AppData\Local\Programs\Python\Python3.6\lib\site-packages
macOS: /Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/site-packages

If unsure, open a Python shell, then type in the following commands to show the Python paths.

```
>>> import sys
>>> sys.path
```

For Ubuntu, SR Research has set up an online software repository for users, and the detailed installation instructions can be found on the SR Support Forum. Once you have configured the online repository on your Ubuntu machine, you can run the following commands to install the EyeLink Developer's Kit, and of course, the Pylink library.

```
sudo apt-get install eyelink-display-software
```

On Windows or macOS, the latest Pylink is already included in the Standalone version of PsychoPy. If you use PsychoPy, you only need to install the EyeLink Developer's Kit; there is no need to manually copy the Pylink library to the site-package folder.

An overview of an eye-tracking experiment

The EyeLink® eye-tracking system has a unique design that features a Host PC, dedicated to eye movement recording, and a Display PC, responsible for stimulus presentation. This two-PC setup allows real-time data collection on the Host PC, which is not interrupted by the experimental task itself. The Host and

Display PCs are linked with an Ethernet cable, which allows the Display PC to send over commands to control data collection and to retrieve real-time eye movement data if needed (e.g., for gaze-contingent tasks such as moving window / boundary crossing paradigms).

The Pylink library is a wrapper of core EyeLink API functions, which, among other things, allow users to open a connection to the Host PC, to display the camera image, and to calibrate the tracker on the Display PC side. We will give an overview of the frequently used functions of Pylink in this chapter, but first, we will outline the operations typically involved in running an EyeLink® experiment.

A typical experiment with an EyeLink® tracker involves the following operations.

1. Initialize a connection to the EyeLink tracker.
2. Open a full-screen window for camera setup and calibration.
3. Send commands to the EyeLink Host PC to set tracking parameters.
4. Open an EDF (EyeLink Data Format) data file on the Host PC.
5. Calibrate the tracker and then run a set of experimental trials. Repeat this process if you need to test multiple blocks of trials.
6. Close the EDF data file on the Host PC and transfer a copy to the Display PC, if needed.
7. Terminate the connection to the Host PC.

We will begin with a minimalist script that contains just a few lines of code while still implementing all the above-noted functions. Please bear in mind that this script is for illustration purposes only; the later sections of this chapter will detail the additional commands that would be required for an actual eye-tracking task. The code below is heavily commented, and there is no need to explain the parameters of each of the functions called for now.

```
# Filename: minimal_example.py

import pylink

# Step 1: connect to the tracker
tk = pylink.EyeLink('100.1.1.1')
```

```
# Step 2: open calibration window
pylink.openGraphics()

# Step 3: set some tracking parameters
tk.sendCommand("sample_rate 1000")

# Step 4: open an EDF data file on the EyeLink Host PC
tk.openDataFile('test.edf')

# Step 5: calibrate the tracker, then run five trials
tk.doTrackerSetup()

for i in range(5):
    # start recording
    tk.startRecording(1,1,1,1)
    # record data for 2 seconds
    pylink.msecDelay(2000)
    # stop recording
    tk.stopRecording()

# Step 6: close the EDF data file and download it
tk.closeDataFile()
tk.receiveDataFile('test.edf', 'test.edf')

# Step 7: close the graphics and the link to the tracker
pylink.closeGraphics()
tk.close()
```

Connect to the tracker

The Display PC connects to the EyeLink Host PC through an Ethernet connection. At the beginning of an eye-tracking experiment, an active (or simulated) connection needs to be established with the Host PC. Without this connection, the experimental script cannot send commands to control the tracker and nor can it receive gaze data from the eye tracker. The command for initializing a connection is *pylink.EyeLink()*. This command takes just one parameter, the IP address of the Host PC. If you omit the IP address parameter, this function will use the default IP address of the EyeLink Host PC, which is 100.1.1.1.

The command *pylink.EyeLink()* returns an EyeLink instance (i.e., '*tk*' in the example below), which has a set of methods that we can use to control the tracker. Once there is an active connection, the status of the EyeLink Host PC will switch to "Link Open" (in the top-right corner of the screen).

```
tk = pylink.EyeLink('100.1.1.1')
```

One frequently seen mistake when connecting to the tracker is that the IP address of the Display PC is wrong. For the Display PC to communicate with the Host PC, it has to be on the same network as the Host PC. Following the EyeLink user manual, the IP address for the Display PC should typically be set to 100.1.1.2, and the subnet mask should be 255.255.255.0.

The eye-tracker in your lab may not always be available, or you may find it more convenient, to debug your experimental script on a computer that is not physically connected to the EyeLink Host PC. If your script does not rely on real-time eye movement data, you can open a simulated connection to the tracker with the following command:

```
tk = pylink.EyeLink(None)
```

Open a camera setup / calibration screen

As noted, calibration is needed for the tracker to accurately report which spatial location the participant is looking at. For a screen-based task, calibration usually involves the presentation of a series of targets at different screen locations. To draw the calibration targets, a graphics window should be opened. We can open a calibration window with the Pylink function *pylink.openGraphics()*, but we can also open a Pygame or PsychoPy window and use it for calibration (see Chapter 7 for information on custom calibration graphics).

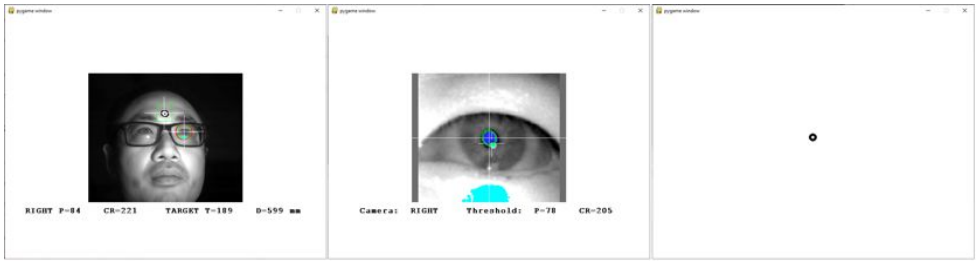


Figure 4-3 *A calibration window is showing the camera image (left), the zoomed-in view of the eye (middle), and the calibration target (right).*

In addition to the calibration targets, you can also show the camera image on the Display PC prior to the actual calibration (see Figure 4-3). For now, it is sufficient to know that opening a window to show the camera image and calibration targets only requires a single line of code.

```
pylink.openGraphics()
```

To close the calibration window after calibration is done (or when a testing session ends), call *pylink.closeGraphics()*.

Configure the tracker

You can configure the tracker parameters (e.g., sampling rate) on the EyeLink Host PC by clicking the relevant buttons on the software interface. Importantly, you can also set these same tracker parameters by sending commands over the link, with the *sendCommand()* function. For instance, the command shown in the example script above sets the sampling rate to 1000 Hz.

```
tk.sendCommand("sample_rate 1000")
```

The following command restricts the calibration target to the central 80% of the screen in both horizontal and vertical directions.

```
tk.sendCommand('calibration_area_proportion 0.8 0.8')
```

Please see Chapter 9 for a list of frequently used commands, and the corresponding GUI buttons on the EyeLink Host PC software.

Open an EDF file

The EyeLink Host PC is a machine dedicated to eye-tracking and data logging. At the beginning of a new testing session, we need to open a data file on the Host PC to save the eye movement data. The data file can be retrieved from the Host PC at the end of a testing session after the data file is closed. For backward compatibility, the name of the EDF data files should not exceed ***8*** characters. To open an EDF data file on the Host PC, use the following command.

```
tk.openDataFile('test.edf')
```

To close an EDF data file that is currently open on the Host PC, call the following command.

```
tk.closeDataFile()
```

Calibration

With the EyeLink tracker, calibration is a two-step process: calibrating the tracker and evaluating the calibration results. Both stages involve presenting a series of visual targets at different known screen positions. The participant is required to move his/her eyes to follow the calibration target. The calibration process can use either 3, 5, 9, or 13 screen positions. A 9-point calibration will give you the best results if you use a chin rest to stabilize the head of the participants. A 5-point or 13-point calibration will give you better results when the participant's head is free to move (i.e., tracking in Remote mode). Following calibration, there is a validation process in which a visual target appears at known screen positions, and the participant moves his/her eyes to follow the target. By comparing the gaze position recorded and the physical position of the target, the tracker will give us an estimation of the gaze errors at different screen positions.

This two-step procedure (calibration and validation) requires a single Pylink command.

```
tk.doTrackerSetup()
```

Once the script has entered the calibration mode, the experimenter can press Enter to show the camera image on the Display PC (see Figure 4-3, left panel), press C to calibrate, press V to validate, and press O (short for output) twice to move on to the next stage of the task.

Data recording

The recommended protocol is to start data recording at the beginning of each trial and to stop recording at the end of each trial. By doing so, the tracker skips the inter-trial intervals, reducing the size of the EDF data file, and simplifying things at the analysis stage. For situations in which a single continuous recording is preferred or required (e.g., in an fMRI or EEG study), one can also start data recording at the beginning of a session (or run), and stop data recording at the end of a session (or run).

To start data recording, call *startRecording()*. This command takes four parameters specifying what types of data (event and/or sample) are recorded in the EDF data file, and what types of data are available over the link during testing. With four 1's, the tracker will record both events and samples in the data file and also make these two types of data available over the link.

```
tk.startRecording(1,1,1,1)
```

Retrieve the EDF data file

As noted, the EDF data file is open on the Host PC, and we need to close it at the end of a testing session by calling *closeDataFile()*. All EDF data files are stored on the Host PC unless they are deleted or overwritten. After closing the file, it is usually transferred to the Display PC when the testing session finishes. The example command below does not alter the file name on the Display PC; you can, of course, rename the downloaded EDF file and save it to the desired folder, if needed.

```
tk.receiveDataFile('test.edf', 'test.edf')
```

Disconnect from the tracker

Once a task finishes, we close the link to the EyeLink Host PC with the *close()* command.

```
tk.close()
```

A real example: Free viewing

The example script in the previous section illustrates the key steps in eye tracker integration, but does not even present any stimuli to the participants. In this section, I will present a complete script to illustrate the frequently used commands and the coding protocol recommended for the EyeLink trackers. To keep things simple, we will use the Pygame library to show the visual stimuli and to register participant responses.

The task implemented in this example script is straightforward. We present two pictures in sequence and request the participant to report whether the picture is clear or blurry, by pressing the left and right arrow keys, respectively. The complete script is listed below; I will explain the critical bits of the script in detail.

```
# Filename: free_view.py

import pylink
import pygame
from pygame.locals import *

# window dimension
scn_w, scn_h = (1920, 1080)

# images and correct keys for all trials
t_pars = [['lake.png', 'c'],
          ['lake_blur.png', 'b'],
          ['train.png', 'c'],
          ['train_blur.png', 'b']]

# Step 1: connect to the tracker
tk = pylink.EyeLink('100.1.1.1')
```

```
# Step 2: open a Pygame window first; then call openGraphics()
# to let Pylink use the Pygame window for calibration
pygame.display.set_mode((scn_w, scn_h), DOUBLEBUF|FULLSCREEN)
pylink.openGraphics()

# Step 3: set some tracking parameters, e.g., sampling rate to 1000 Hz
# put the tracker in offline mode before we change its parameters
tk.setOfflineMode()
# give the tracker an extra 100 ms to switch the operation mode
pylink.msecDelay(100)

# set the sampling rate to 1000 Hz
tk.sendCommand("sample_rate 1000")

# Send screen resolution to the tracker
tk.sendCommand("screen_pixel_coords = 0 0 %d %d" % (scn_w-1, scn_h-1))

# request the tracker to perform a standard 9-point calibration
tk.sendCommand("calibration_type = HV9")

#calibrate the central 80% of the screen
tk.sendCommand('calibration_area_proportion 0.8 0.8')
tk.sendCommand('validation_area_proportion 0.8 0.8')

# Step 4: open EDF data file
tk.openDataFile('freeview.edf')
# optional file header to identify the experimental task
tk.sendCommand("add_file_preamble_text 'Free Viewing'")

# Step 5: start calibration and switch to the camera setup screen
tk.doTrackerSetup()

# run through all four trials
for t in t_pars:

    # unpacking the picture and correct response key
    pic_name, cor_key = t

    # load the picture
    img = pygame.image.load('images/' + pic_name).convert()

    # record_status_message : show some info on the Host PC
    tk.sendCommand("record_status_message 'Current Picture: %s'"% pic_name)
```

```
    # drift check; parameters: x, y, draw_target, allow_setup
    # draw_target (1-default, 0-user draw the target then call this
function)
    # allow_setup (1-allow pressing ESCAPE to recalibrate, 0-not allowed)
    try:
        err = tk.doDriftCorrect(int(scn_w/2), int(scn_h/2), 1, 1)
    except:
        tk.doTrackerSetup()

    # start recording
    # parameters: event_in_file, sample_in_file,
    # event_over_link, sample_over_link (1-yes, 0-no)
    tk.startRecording(1,1,1,1)
    # wait for 100 ms to cache some samples
    pylink.msecDelay(100)

    # present the image
    surf = pygame.display.get_surface()
    surf.blit(img, (0,0))
    pygame.display.flip()

    # log a message to mark image onset
    tk.sendMessage('Image_onset')

    # get key response in a while loop
    pygame.event.clear() # clear all cached events
    got_key = False
    while not got_key:
        for ev in pygame.event.get():
            if ev.type == KEYDOWN:
                if ev.key in [K_c, K_b]:
                    got_key = True

    # stop recording
    tk.stopRecording()
    pylink.pumpDelay(50)

# Step 6: close the EDF data file and download it
tk.closeDataFile()
tk.receiveDataFile('freeview.edf', 'freeview.edf')

# Step 7: close the link to the tracker
tk.close()
```

```
# quit pygame
pygame.quit()
```

As noted, this script uses Pygame to present the pictures and to register the keyboard responses. The first thing we need is to import Pygame and also the Pygame local constants (see Chapter 3) at the beginning of the script. We then specify the screen dimensions, by unpacking a two-item tuple (1920, 1080) onto two variables, *scn_w*, and *scn_h*. One other constant we initialized is the picture and correct key for each of the trials, encapsulated in lists, just like we did with the example scripts in Chapter 2 and Chapter 3. For the commands relevant to eye-tracking, I will highlight a few important ones here.

Calibration graphics

As noted, the *pylink.openGraphics()* command will open a window to present the calibration target. It does more than presenting the calibration target; it also draws the camera image and logs keyboard events on the Display PC. There is no need to detail all the functions encapsulated in this function, but I would note here that the graphics backend is SDL; yes, the same graphics backend as Pygame. As clearly stated in the Pylink manual, if a display mode is already set, this function will use the current display mode. That is, Pylink will use an already open Pygame window as the calibration window when we call *pylink.openGraphics()*.

Idle mode

The *setOfflineMode()* command is frequently called in EyeLink scripts. The name of the command is self-explaining, i.e., putting the tracker into the offline (idle) mode. Every time we call this command, the EyeLink Host PC will switch to the Offline screen, in which the tracker is ready to accept various configuration commands. Following this command, we allocate about 100 ms for the tracker to get ready for additional operations.

EyeLink Host commands

The EyeLink Host PC can accept lots of commands for tracker configuration. You can send these commands over the link to control the tracker. This example script illustrates a few frequently used commands, e.g., *sample_rate*,

calibration_type, *screen_pixel_coords*. The *screen_pixel_coords* command is particularly important, as it informs the tracker about the resolution of the screen used for stimulus presentation. This command is essential because the tracker needs this information to estimate eye movement velocity in degrees of visual angle. The velocity data is necessary for accurate online detection of eye events (saccades and fixations).

```
# set the sampling rate to 1000 Hz
tk.sendCommand("sample_rate 1000")

# Send screen resolution to the tracker
tk.sendCommand("screen_pixel_coords = 0 0 %d %d" % (scn_w-1, scn_h-1))

# request the tracker to perform a standard 9-point calibration
tk.sendCommand("calibration_type = HV9")

#calibrate the central 80% of the screen
tk.sendCommand('calibration_area_proportion 0.8 0.8')
tk.sendCommand('validation_area_proportion 0.8 0.8')
```

Preamble text in EDF

For the EDF data file, it is good practice to put some header information there. Otherwise, it can be difficult to tell which EDF data file belongs to which research project. Please note that the header text, i.e., "*Free Viewing Task in Chapter 4*", needs to be in a pair of single quotes (see example below).

```
# optional file header to identify the experimental task
tk.sendCommand("add_file_preamble_text 'Free Viewing Task in Chapter 4'")
```

Record status message

At the beginning of each trial, we can show some trial information in the bottom-right corner on the Host PC screen (see Figure 4-4), by sending over a "*record_status_message*" command. The message sent with this command should be informative, so the experimenter can tell which trial is running, how many trials have been completed, etc.

```
# record_status_message : show some info on the Host PC
tk.sendCommand("record_status_message 'Current Picture: %s'"% pic_name)
```

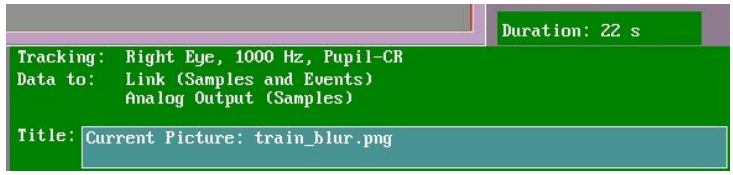


Figure 4-4 *Recording status message shown in the gray box in the bottom-right corner on the EyeLink Host PC.*

Drift-check/drift-correction

One of the most important commands illustrated in this example script is *doDriftCorrect()*. Following this command, a target appears on the screen, and the participant needs to look at the target and the experimenter then confirms gaze position by pressing a key. The tracker can thus estimate the current tracking accuracy with the reported gaze position and the physical position of the target.

This feature is known as drift-correction or drift-check, and one can think of it as a 1-point validation of tracking accuracy. The concept of drift correction is inherited from earlier head-mounted versions of the EyeLink eye tracker. In these models it was possible for the headband to slip and cause a systematic drift in the gaze data (when tracking in Pupil-only mode). This issue was tackled by a linear correction of the gaze data based on the gaze error reported by the drift-correction procedure.

```
# drift check; parameters: x, y, draw_target, allow_setup
# draw_target (1-default, 0-user draw the target then call this function)
# allow_setup (1-allow pressing ESCAPE to recalibrate, 0-not allowed)
try:
    err = tk.doDriftCorrect(int(scn_w/2), int(scn_h/2), 1, 1)
except:
    tk.doTrackerSetup()
```

In all recent EyeLink systems, by default, the gaze error is no longer used to correct the gaze data, as the Pupil-CR tracking algorithm is resilient to small head or camera movements. Instead, the drift-correction routine allows the experimenter to check the tracking accuracy, and recalibrate if necessary.

The *doDriftCorrect*() command takes four parameters. The first two are x, y pixel coordinates for the drift-check target. Note that x, y must be integers, e.g., 512, 384. The third parameter specifies whether Pylink should draw the target. If set to 0, we need to first draw a target at the (x, y) pixel coordinates, then call the *doDriftCorrect*() command. The fourth parameter controls whether Pylink should allow the experimenter to recalibrate the tracker. This option should be enabled.

Logging messages

Messages are sent to the tracker whenever a critical event occurs; for instance, a picture appears on the screen. With these messages in the EDF data file, we can tell what events occurred at what time and segment the recording for meaningful analysis. For instance, in the example script, the "Image_onset" message is sent to the tracker immediately after each picture is shown on the screen.

```
# send a message to mark image onset
tk.sendMessage('Image_onset')
```

In addition to user-defined messages, we can also log messages that are useful for offline data analysis and visualization in Data Viewer. As shown in Figure 4-5, Data Viewer will locate and load the background image when it reads in a message that contains the "IMGLOAD" keyword (command). We will cover the various commands supported by Data Viewer in more detail in Chapter 5.

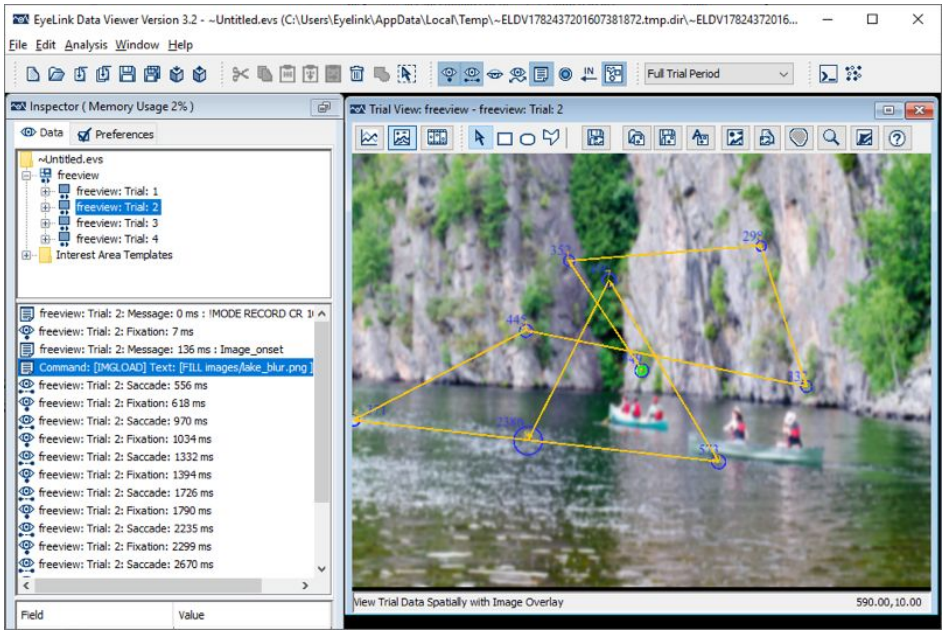


Figure 4-5 *A screenshot of the Data Viewer software showing the background image used for gaze data overlay. The left-side panel highlights the IMGLOAD command.*

To briefly summarize, this chapter introduced video-based eye-tracking and discussed some of the frequently used Pylink functions.

When programming an experimental task it is very important to bear in mind data analysis and visualization needs. A small amount of time invested in ensuring that the script contains critical event marking messages and Data Viewer integration messages can save a great deal of time at the analysis stage. In Chapter 5, I will focus on the programming protocols that allow seamless integration with Data Viewer analysis software.