Chapter 6 Retrieving gaze data over the link

The types of eye-tracking applications discussed in Chapter 4 and Chapter 5 involve “passive recording”. That is, eye movements are recorded while the participant is performing a task, but we do not use the eye movement data in any way during the task. In many research scenarios, however, it is necessary to retrieve real-time gaze data to drive the presentation of the stimuli. For instance, in a moving window task, the subject is allowed to view a small screen region around the current gaze position; in a saccadic adaptation task, the target is shifted immediately after saccade onset. This chapter will focus on ways of retrieving eye movement data during recording. We will also discuss recording playback, which allows users to retrieve the gaze data from a previous recording (trial).

Samples and events

Two types of data can be retrieved from the EyeLink Host PC during recording - samples, and events. The samples are the “raw” eye movement data and comprise the x, y gaze position, and pupil size. They are provided every 1 or 2 ms (depending on the sampling rate). Events are “parsed data” - the result of the online parsing algorithm employed by EyeLink eye trackers, notably, the onset and offset of fixations, saccades, and blinks. The parsing of gaze data into fixations and saccades is a complicated matter, and different research fields have different conventions and approaches.

The eye event identification algorithm used by the EyeLink trackers is velocity-based. A saccade is detected if the velocity or acceleration of the gaze movement exceeds predefined thresholds, and the same saccade terminates when both the velocity and acceleration fall below these thresholds. The

termination of the previous saccade defines the onset of a new fixation; the start of a saccade defines the end of the preceding fixation (see Figure 6-1).

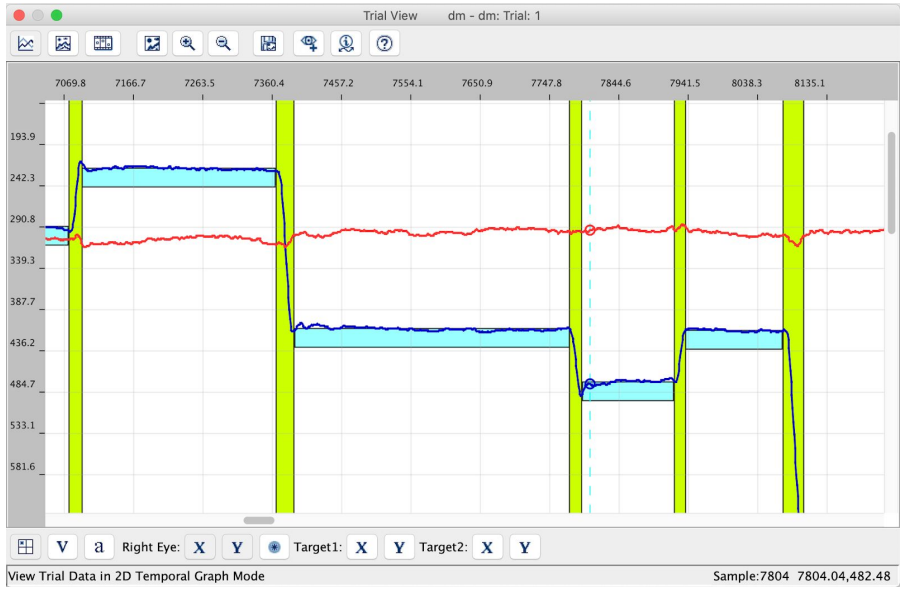

Figure 6-1 *The EyeLink trackers detect eye events based on instantaneous velocity and acceleration. In this temporal graph, the X, Y gaze traces are plotted in blue and red, respectively. Blue boxes represent fixations, and vertical stripes represent saccades.*

The high sampling rate of the EyeLink trackers (minimum 250 Hz) allows the identification of eye events during recording, and the tracker can make these events available over the link. Bear in mind that we can access sample data in a near real-time fashion, with a delay in the 1-2 sample range (depending on the sampling rate and filter setting). Eye events, on the other hand, require some time to identify and cannot be accessed in a real-time manner. For instance, for the detection of saccade onset, the tracker by default requires an additional 4-ms verification period to be sure that velocity or acceleration is indeed above the thresholds. For the detection of the saccade offset, the verification period is 20 ms. Consequently, when a saccade END event is available over the link, the event happened at least 20 ms ago.

In addition to gaze data, the samples may also include HREF or RAW data, depending on the tracker configuration. HREF is head-referenced data, i.e., eye rotations in relation to the head; it can be useful for physiological studies examining the actual movement of the eye. RAW data is the raw data from the camera sensor; it is helpful in studies where researchers would like to calibrate the tracker with their calibration routines. For instance, in MWorks[1], the RAW data is read to perform a simple calibration by adjusting signal gain and offset to map the RAW signal to a screen region.

```
tk.sendCommand("file_sample_data  =
LEFT,RIGHT,GAZE,GAZERES,PUPIL,HREF,AREA,STATUS,HTARGET,INPUT")
tk.sendCommand("link_sample_data  =
LEFT,RIGHT,GAZE,GAZERES,PUPIL,HREF,AREA,STATUS,HTARGET,INPUT")
```

Users need to specify what types of data are included in the samples. Assuming that you have instantiated a connection to the tracker (*tk*), the above commands will request the tracker to save all types of sample data in the EDF data file (*file_sample_data*) and to make all types of sample data available over the link (*link_sample_data*).

Users can also use the following commands to specify what types of event data should be saved in the EDF data file or made available over the link. I will not go through the different sample (e.g., GAZERES) and event (e.g., FIXUPDATE) flags you can include in these commands, please see Section 4 of the EyeLink User Manual for details.

```
tk.sendCommand("file_event_filter =
LEFT,RIGHT,FIXATION,SACCADE,BLINK,MESSAGE,BUTTON,INPUT")
tk.sendCommand("link_event_filter =
LEFT,RIGHT,FIXATION,FIXUPDATE,SACCADE,BLINK,BUTTON,INPUT")
```

When you call the *startRecording()* command to start recording, it is mandatory to specify four parameters to let the tracker know whether sample and event data should be saved to file and made available over the link. These four parameters are:

[1] An open-source tool for experimental task design and data collection, https://mworks.github.io/.

```
<file_samples> --  1, writes samples to EDF; 0, disables sample recording
<file_events> --  1, writes events to EDF; 0, disables event recording
<link_samples> --  1, sends samples through link; 0, disables link sample
<link_events> -- 1, sends events through link; 0, disables link event
```

With the following command, both events and samples will be recorded in the EDF data file and made available over the link. Turning the <*link_samples*> and <*link_events*> flags to 0 will disable sample and event accessing during recording.

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

Access sample data over the link

Once we have started recording and made the eye movement data available over the link, we can use various commands to retrieve either samples or events during testing. The retrieval of samples is usually useful in various gaze-contingent tasks, for instance, the classic moving window task by McConkie and Rayner (1975; see Rayner, 2014, for a review). Fast online retrieval of gaze samples is also necessary for boundary-crossing tasks, in which the stimuli are manipulated the moment gaze crosses an invisible boundary on the screen.

We use the *getNewestSample()* command to retrieve the latest sample. This command will return a *sample*, which has lots of properties (data) that we can extract. For instance, the following code snippet helps to examine whether the latest sample comes from the left, right eye, or both eyes, and to get the resolution data (i.e., how many pixels correspond to one degree of visual angle at the current gaze position) and the timestamp of the sample.

```
smp = tk.getNewestSample() # retrieve the latest sample
is_left = smp.isLeftSample() # left eye sample
is_right = smp.isRightSample() # right eye sample
is_bino = smp.isBinocular() # binocular recording is on
res = smp.getPPD() # 1 deg = how many pixels at current gaze position
time_stamp = smp.getTime() # timestamp on tracker timer
```

To get the current gaze position, we need to call the *getLeftEye()* or *getRightEye()* command to get the *SampleData* out of the sample. For instance, you can use the following code snippet to check if data is available from the left, right, or both eyes. If the left eye data is available, we retrieve the gaze position from the left eye data stream; otherwise, we extract data from the right eye data stream.

```
while True:
    # poll the newest sample
    smp = tk.getNewestSample()
    if smp is not None:
        if smp.getEye()==0: # left eye
            smp_data = smp.getLeftEye()
        else: # right eye or binocular
            smp_data = smp.getRightEye()

        gaze_pos = smp_data.getGaze()
        print(gaze_pos)
```

In addition to gaze position (in screen pixel coordinates), HREF data can be retrieved with *getHREF()*, raw data with *getRawPupil()*, and pupil size data with *getPupilSize()*.

I have included a short script below to illustrate the various type of sample data that we can retrieve with Pylink. Running this short script should not require any other library, except for Pylink. In this example script, we extract all the available data from the samples, including gaze, HREF, raw, and pupil size data. We open a calibration window after we connect to the tracker, we calibrate the tracker and then close the calibration window. We also send two commands to the tracker to make sure the tracker is operating at 1000 Hz, and importantly, to make the HREF and raw PUPIL data available over the link. We then open a plain text file to save the sample data.

Please note that, in the while-loop, we check the timestamp of the latest available sample and save it to file only if the timestamp of a retrieved sample is new. Three seconds later, we close the data files and terminate the link to the tracker. Bear in mind that in this example we log the samples into a plain text file, just for illustration purposes only. Samples are stored in the EDF data files

on the Host, so there is no need to log them in this way on the stimulus presentation computer.

```
# Filename: retrieve_samples.py

import pylink

# connect to the tracker and open an EDF
tk = pylink.EyeLink('100.1.1.1')
tk.openDataFile('smp_test.edf')

# open a window to calibrate the tracker
pylink.openGraphics()
tk.doTrackerSetup()
pylink.closeGraphics()

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

# make sure gaze, HREF, and raw (PUPIL) data is available over the link
tk.sendCommand('link_sample_data  =
LEFT,RIGHT,GAZE,GAZERES,PUPIL,HREF,AREA,STATUS,INPUT')

# start recording
error = tk.startRecording(1,1,1,1)
pylink.pumpDelay(100) # cache some samples for event parsing

# open a plain text file to write the sample data
text_file = open('sample_data.csv', 'w')

t_start = tk.trackerTime() # current tracker time
smp_time = -1
while True:
    # break after 10 seconds have elapsed
    if tk.trackerTime() - t_start > 3000:
        break

    # poll the latest samples
    dt = tk.getNewestSample()
    if dt is not None:
        if dt.isRightSample():
            gaze = dt.getRightEye().getGaze()
            href = dt.getRightEye().getHREF()
            raw  = dt.getRightEye().getRawPupil()
```

```
            pupil= dt.getRightEye().getPupilSize()
        elif dt.isLeftSample():
            gaze = dt.getLeftEye().getGaze()
            href = dt.getLeftEye().getHREF()
            raw  = dt.getLeftEye().getRawPupil()
            pupil= dt.getLeftEye().getPupilSize()

        timestamp = dt.getTime()
        if timestamp > smp_time:
            smp = map(str,[dt.getTime(),gaze,href,raw,pupil])
            text_file.write('\t'.join(smp) + '\n')
            smp_time = timestamp

tk.stopRecording() # stop recording
tk.closeDataFile() # close EDF data file on the Host
text_file.close()
tk.close() #close the link to the tracker
```

An excerpt from the output text file is shown below. The columns are sample timestamp, gaze position, HREF x/y, raw data, and pupil size (area or diameter in arbitrary units).

```
3577909.0    (872.0999755859375, 392.70001220703125)      (1631.0, 7100.0)
       (-1924.0, -3891.0)    366.0
3577910.0    (872.2000122070312, 393.3999938964844)       (1632.0, 7093.0)
       (-1924.0, -3889.0)    365.0
3577911.0    (872.2999877929688, 393.79998779296875)      (1632.0, 7090.0)
       (-1924.0, -3888.0)    365.0
3577912.0    (872.2000122070312, 397.20001220703125)      (1631.0, 7060.0)
       (-1924.0, -3876.0)    365.0
3577913.0    (872.0999755859375, 397.29998779296875)      (1630.0, 7059.0)
       (-1925.0, -3875.0)    365.0
3577914.0    (870.7000122070312, 397.3999938964844)       (1619.0, 7059.0)
       (-1930.0, -3875.0)    362.0
3577915.0    (870.7000122070312, 397.3999938964844)       (1619.0, 7058.0)
       (-1930.0, -3875.0)    362.0
```

The following example script uses the technique illustrated above to implement a gaze-contingent task in PsychoPy. It also makes use of the *Aperture* feature of PsychoPy, which allows users to show or hide a portion of the screen. As shown

in Figure 6-2, an Aperture reveals a circular shaped region centered at the current gaze position.

```
# Filename: gc_window.py

import pylink
from EyeLinkCoreGraphicsPsychoPy import EyeLinkCoreGraphicsPsychoPy
from psychopy import visual, core, event, monitors

# established a link to the tracker
tk = pylink.EyeLink('100.1.1.1')

# Open an EDF data file
tk.openDataFile('psychopy.edf')

# Initialize custom graphics in Psychopy
scnWidth, scnHeight = (800, 600)
# set monitor parameters
mon = monitors.Monitor('myMac15', width=53.0, distance=70.0)
mon.setSizePix((scnWidth, scnHeight))
win = visual.Window((scnWidth, scnHeight), monitor=mon, fullscr=False,
color=[0,0,0],
                    units='pix', allowStencil=True)
genv = EyeLinkCoreGraphicsPsychoPy(tk, win)
pylink.openGraphicsEx(genv)

# make sure sample data is available over the link
tk.sendCommand("link_sample_data  =
LEFT,RIGHT,GAZE,GAZERES,PUPIL,HREF,AREA,STATUS,INPUT")

# show instructions and calibrate the tracker
instructions = visual.TextStim(win, text='Press ENTER twice to calibrate the
tracker')
instructions.draw()
win.flip()
event.waitKeys()
tk.doTrackerSetup()

# set up a circular aperture as the gaze-contingent window
gaze_window = visual.Aperture(win, size=200)
gaze_window.enabled=True

# load and stretch the background image to fill full screen
```

```
img = visual.ImageStim(win, image='sacrmeto.jpg', size=(scnWidth,
scnHeight))

# start recording
tk.startRecording(1,1,1,1)

# show the image indefinitely until a key is pressed
gaze_pos =  (-32768, -32768)
terminate = False
event.clearEvents() # clear cached (keyboard/mouse etc.) events
while not terminate:
    # check for keypress to terminate a trial
    if event.getKeys():
        terminate = True

    # check for new samples
    dt = tk.getNewestSample()
    if dt is not None:
        if dt.isRightSample():
            gaze_pos = dt.getRightEye().getGaze()
        elif dt.isLeftSample():
            gaze_pos = dt.getLeftEye().getGaze()

    # draw background image with the aperture (window)
    img.draw()
    gaze_window.pos = (gaze_pos[0]-scnWidth/2, scnHeight/2-gaze_pos[1])
    win.flip()

# stop recording
tk.stopRecording()

# close EDF and the link
tk.closeDataFile()

#close the link to the tracker
tk.close()

# close the graphics
win.close()
core.quit()
```

The code relevant to online sample retrieval is much like that in the previous example. The only thing new here is the use of a custom calibration graphics library built with native PsychoPy functions (see Chapter 4 and Chapter 7).

```
from EyeLinkCoreGraphicsPsychoPy import EyeLinkCoreGraphicsPsychoPy

# Initialize custom graphics in Psychopy
scnWidth, scnHeight = (800, 600)
# set monitor parameters
mon = monitors.Monitor('myMac15', width=53.0, distance=70.0)
mon.setSizePix((scnWidth, scnHeight))
win = visual.Window((scnWidth, scnHeight), monitor=mon, fullscr=False,
color=[0,0,0],
                    units='pix', allowStencil=True)
genv = EyeLinkCoreGraphicsPsychoPy(tk, win)
pylink.openGraphicsEx(genv)
```

As shown in Figure 6-2, when running this script, a circular-shaped window will move with the current gaze position to reveal the image covered below.

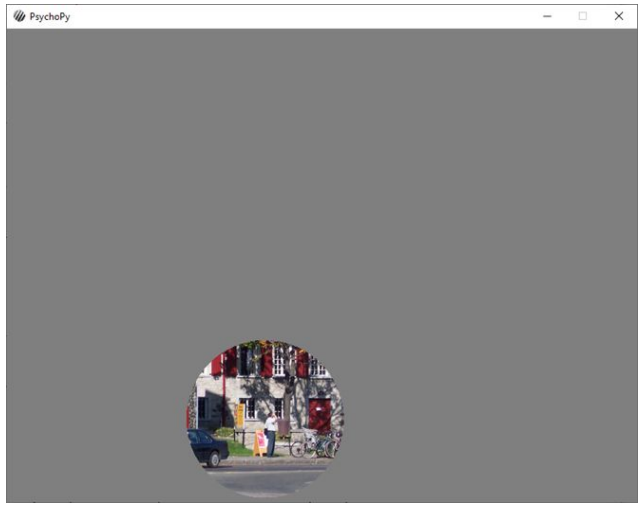

Figure 6-2 *A simple moving window task implemented in PsychoPy.*

Access eye events over the link

As noted previously, sample data is parsed online by the Host PC during recording to detect the eye events, e.g., the onset and offset of saccades, fixations, and blinks. Depending on the tracker settings, both events and sample data can be saved in the EDF data files and made available over the link during testing. It is possible to convert the EDF data file into plain text files (see Chapter 8), so we can quickly examine the various eye events parsed online by the Host PC. In the example data lines shown below, we see two pairs of events: SSACC (start of saccade) and ESACC (end of saccade), and SFIX (start of fixation), and EFIX (end of fixation). The letter "L" (or "R") following the event type label indicates whether the data comes from the left or right eye. Note that for a "start" event, the timestamp is the only data recorded. For an "end" event, summary data is available, e.g., in the case of the EFIX event, the duration of fixation (2368 ms), the average x and y gaze position of the samples within the fixation (755,447) and the average pupil size during the fixation (5161).

```
SSACC L  659818
ESACC L  659818       659962 148      749.0  413.0  752.0  437.0   0.35
571
SFIX L   659966
EFIX L   659966       662330 2368     755.0  447.0   5161
```

The Pylink library has lots of predefined constants to represent the types of eye events that can be retrieved online. For instance, enter "pylink.STARTBLINK" (without quotes) in the Python shell (after importing Pylink), the shell will return the corresponding constant "3". The Pylink constants for all eye events are listed in Table 6-1.

```
>>> import pylink
>>> pylink.STARTBLINK
3
```

Table 6-1 *Pylink constants for various eye events.*

Variables	Constants	Description
STARTBLINK	3	Start of blink (pupil disappear)
ENDBLINK	4	End of blink (pupil reappear)
STARTSACC	5	Start of saccade

ENDSACC	6	End of saccade
STARTFIX	7	Start of fixation
ENDFIX	8	End of fixation
FIXUPDATE	9	Update fixation summary data during a fixation
MESSAGEEVENT	24	Message received by tracker
BUTTONEVENT	25	Button press
INPUTEVENT	28	Change of pin status on the Host parallel port
SAMPLE_TYPE	200	Sample data

When talking about eye events, people are generally referring to fixations and saccades. However, as you can see from the list above, these are not the only types of events. For example, the Host PC also flags the start and end of blinks. The onset of blinks is flagged when the Host PC cannot compute a valid X, Y location for gaze. Because the EyeLink tracker uses a velocity-based algorithm to detect the onset and offset of saccades, you will observe that these blink events are typically "wrapped" in a (false) saccade. The sequence of events generally is SSACC, SBLINK, EBLINK, ESACC. The reason for this is that as the eyelids descend over the pupil, the center of the pupil (from the camera perspective) is rapidly changing (shifting downwards as the eyelid descends and obscures the upper part). The tracker, of course, will see a fast "movement" of the eyeball and report a saccade start event. When the eyelids reopen, the tracker sees the pupil center quickly stabilizes. The velocity of the "eye movement" falls below the threshold, leading to the detection of a saccade offset event.

As noted previously, the access of eye events during testing is not real-time. When an eye event (e.g., STARTSACC and ENDFIX) is available in the link buffer, it actually occurred some milliseconds ago. Despite this, being able to access events online can be useful in a number of different experimental scenarios. Here we will present a short script to show how to retrieve the start and end events and then examine the event parsing delay offline with the EDF data file. This example script is similar to the one in which we showed how to retrieve the samples.

We first call the *getNextData()* command to see whether any events are available over the link; this command will return a code representing an event (see Table 6-1). Please note that this function treats samples as special "events"--SAMPLE_DATA, represented by the constant 200. If there is no data

(samples or events) available, the command will return 0. The script also calls *getFloatData()* to retrieve the currently available data in the buffer so that we can get additional information (e.g., the timestamp of the event). In the code below, we continuously examine the start and end events for fixations and saccades in a while loop. If an event occurs, we log a message that contains the actual timestamp of the event. In the EDF data file, this message will have a timestamp, which is the time at which the stimulus presentation PC retrieved this event over the link.

```
# Filename: retrieve_events.py

import pylink

# connect to the tracker and open an EDF
tk = pylink.EyeLink('100.1.1.1')
tk.openDataFile('ev_test.edf')

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

# make sure all types of event data are available over the link
tk.sendCommand("link_event_filter =
LEFT,RIGHT,FIXATION,FIXUPDATE,SACCADE,BLINK,BUTTON,INPUT")

# open a window to calibrate the tracker
pylink.openGraphics()
tk.doTrackerSetup()
pylink.closeGraphics()

# start recording
error = tk.startRecording(1,1,1,1)
pylink.pumpDelay(100) # cache some samples for event parsing

t_start = tk.trackerTime() # current tracker time
while True:
    # break after 10 seconds have elapsed
    if tk.trackerTime() - t_start > 5000:
        break

    # retrieve the oldest event in the buffer
    dt = tk.getNextData()
    if dt > 0:
        ev = tk.getFloatData()
```

```
        if dt == pylink.STARTSACC:
            tk.sendMessage('STARTSACC %d'% ev.getTime())
        if dt == pylink.ENDSACC:
            tk.sendMessage('ENDSACC %d'% ev.getTime())
        if dt == pylink.STARTFIX:
            tk.sendMessage('STARTFIX %d'% ev.getTime())
        if dt == pylink.ENDFIX:
            tk.sendMessage('ENDFIX %d'% ev.getTime())

tk.stopRecording() # stop recording
tk.closeDataFile() # close EDF data file on the Host
tk.receiveDataFile('ev_test.edf', 'ev_test.edf') # retrieve EDF from Host
tk.close() #close the link to the tracker
```

Before closing the link to the tracker, the EDF data file stored on the Host PC can be retrieved and saved on the Display PC with the *receiveDataFile()* command. The EDF2ASC converter provided by SR Research can be used to convert the EDF data file into an ASCII (plain text) file (see Chapter 8 for details). The EDF2ASC converter allows users to determine what types of data are saved into this plain text file. For simplicity, we will only retrieve the various eye events and messages, ignoring the samples.

The first data line starts with the keyword “EFIX”. which marks the end of a fixation and gives us some summary data i.e., the fixation’s start time, end time, duration, average gaze position, and average pupil size. The end time of this fixation is 96007959, and the following timestamp of 96007960, which is the onset time of a saccade, i.e., the “SSACC” event. The third line of data is a message received from the Display PC, which starts with the keyword “MSG”. The timestamp of this message is 96007986, i.e., the time at which the stimulus presentation script received the EFIX event over the link. Note that this EFIX event has a timestamp of 96007959 ms. It is the same fixation we mentioned above; so, the link delay in retrieving this EFIX event is 96007986 - 96007959 = 27 ms. With the same calculation, we can estimate that, in my testing setup, the average event access delay for saccade onset events (SSACC) is about 26 ms. For saccade offset events (ESACC) the average delay was about 39 ms, and for fixation onset events (SFIX) the average delay was about 38 ms. Be cautious if you plan to use these online detected eye events to drive your gaze-contingent display.

```
EFIX R   96007475     96007959      485       569.8   555.2    392
SSACC R  96007960
MSG    96007986 ENDFIX 96007959
MSG    96007986 STARTSACC 96007960
ESACC R  96007960     96008003      44        553.9   568.9   331.8   495.0
5.21       210
SFIX R   96008004
MSG    96008042 ENDSACC 96008003
MSG    96008042 STARTFIX 96008004
```

It is, of course, possible to retrieve more information from the various eye events, in addition to the timestamp of the events (see example script "*retrieve_events_2.py*"). For instance, the saccade start and end time, saccade amplitude, peak velocity, and start and end positions can be retrieved from ESACC events.

```
# make sure all types of event data are available over the link
tk.sendCommand("link_event_filter =
LEFT,RIGHT,FIXATION,FIXUPDATE,SACCADE,BLINK,BUTTON,INPUT")

    dt = tk.getNextData()
    if dt > 0:
        ev = tk.getFloatData()
        if dt == pylink.ENDSACC:
            print('ENDSACC Event: \n',
                  'Amplitude',       ev.getAmplitude(),'\n',
                  'Angle',           ev.getAngle(),'\n',
                  'AverageVelocity',   ev.getAverageVelocity(),'\n',
                  'PeakVelocity',    ev.getPeakVelocity(),'\n',
                  'StartTime',       ev.getStartTime(),'\n',
                  'StartGaze',       ev.getStartGaze(),'\n',
                  'StartHREF',       ev.getStartHREF(),'\n',
                  'StartPPD',        ev.getStartPPD(),'\n',
                  'StartVelocity',   ev.getStartVelocity(),'\n',
                  'EndTime',         ev.getEndTime(),'\n',
                  'EndGaze',         ev.getEndGaze(),'\n',
                  'EndHREF',         ev.getEndHREF(),'\n',
                  'EndPPD',          ev.getEndPPD(),'\n',
                  'EndVelocity',     ev.getEndVelocity(),'\n',
                  'Eye',             ev.getEye(),'\n',
                  'Time',            ev.getTime(),'\n',
```

```
                    'Type',           ev.getType(), '\n')
```

The output of the above scripts is listed below. Note the saccade amplitude in horizontal and vertical directions are reported as a tuple.

```
ENDSACC Event:
 Amplitude (4.537407467464733, 0.5766124065721234)
 Angle -7.242339197477229
 AverageVelocity 102.0999984741211
 PeakVelocity 175.3000030517578
 StartTime 52627.0
 StartGaze (417.79998779296875, 1739.0999755859375)
 StartHREF (-1450.0, -2441.0)
 StartPPD (48.0, 49.70000076293945)
 StartVelocity 5.800000190734863
 EndTime 52671.0
 EndGaze (633.0999755859375, 1767.699951171875)
 EndHREF (-236.0, -2669.0)
 StartPPD (48.0, 49.70000076293945)
 EndVelocity 37.599998474121094
 Eye 1
 Time 52671.0
 Type 6
```

Table 6-2 gives a summary of the Pylink commands that can be used to retrieve the various eye event data. These commands are carefully grouped in Table 6-2, based on their functionality.

1. Applicable to all eye events. *getTime()* returns the time point at which an event occurred, *getEye()* returns the eye being tracked (0-left, 1-right, 2-binocular), and *getType()* returns the event type, i.e., the constants listed in Table 6-1.
2. These commands return the start and end time of an event.
3. These commands return eye position data in the GAZE coordinates.
4. These commands return eye position data in the HREF coordinates.
5. These commands return the resolution data (PPD, pixels per degree), i.e., how many screen pixels correspond to 1 degree of visual angle
6. These commands return the velocity data.
7. These commands return the pupil size data.

8. These commands return are only applicable to the ENDSACC event; they return saccade amplitude and direction.

One thing worth mentioning here is that the fixation update (FIXUPDATE) event reports the same set of data as the fixation end event, at predefined intervals (50 ms by default) following the onset of a fixation. In this way it is possible to access the states of an (ongoing) fixation (e.g., the average gaze position) before it ends. This feature can be useful in HCI applications, e.g., using gaze to drive the mouse cursor.

Table 6-2 *The Pylink commands for retrieving event data. Here we use "Y" to mark the properties available for each eye event.*

		STARTSACC	ENDSACC	STARTFIX	ENDFIX	STARTBLINK	ENDBLINK	FIXUPDATE
1	getTime()	Y	Y	Y	Y	Y	Y	Y
	getEye()	Y	Y	Y	Y	Y	Y	Y
	getType()	Y	Y	Y	Y	Y	Y	Y
	getStatus()	Y	Y	Y	Y	Y	Y	Y
2	getStartTime()	Y	Y	Y	Y	Y	Y	Y
	getEndTime()		Y		Y		Y	Y
3	getStartGaze()	Y	Y	Y	Y			Y
	getEndGaze()		Y		Y			Y
	getAverageGaze()				Y			Y
4	getStartHREF()	Y	Y	Y	Y			Y
	getEndHREF()		Y		Y			Y
	getAverageHREF()				Y			Y
5	getStartPPD()	Y	Y	Y	Y			Y
	getEndPPD()		Y		Y			Y
6	getStartVelocity()	Y	Y	Y	Y			Y
	getEndVelocity()		Y		Y			Y
	getAverageVelocity()		Y		Y			Y
	getPeakVelocity()		Y		Y			Y
7	getStartPupilSize()			Y	Y			Y
	getEndPupilSize()				Y			Y
	getAveragePupilSize()				Y			Y

8	getAmplitude()		Y					
	getAngle()		Y					

Recording playback

EyeLink eye trackers also allow users to access eye movement data stored in an open EDF data file on the Host, so we can access the samples and events from a previous trial as if the Host PC is streaming the data over the link. This feature can be useful for a demonstration of an interesting behavioral effect, or for a neuro-feedback type study in which physiological responses evoked by a task is fed back to the subject to shape behavior.

Playback will rewind the previous data block marked by the START and END of a recording. Please bear in mind that the EDF data file needs to be open on the Host PC for playback to work. A typical setup would involve:

1. Open an EDF data file on the Host at the beginning of a session
2. Start recording at the beginning of a new trial
3. Present the visual stimuli and collect responses
4. Stop recording at the end of a trial
5. Call *startPlayBack()* and wait for the tracker switch to playback mode
6. Retrieve samples or (and) events over the link, and plot the data, if needed
7. Call *stopPlayBack()*
8. After completing all experimental trials, close the EDF data file.

We present a short script below to demonstrate this feature. The task implemented in the script is a simple visual distractor task. The participant initiates a saccade to a visual target (green disk) while a visually salient (bright) diamond-shaped distractor is present (or absent). The distractor will distort the trajectory of the primary saccade, causing it to deviate away from the straight path to the target.

```
# Filename: recording_playback.py

import pylink
```

```
import pygame
from math import hypot

# connect to the tracker
tk = pylink.EyeLink()

# send screen coordinates to the tracker
tk.sendCommand('screen_pixel_coords 0 0 800 600')

# open a Pygame window to force openGraphics() to use a non-fullscreen
window
win = pygame.display.set_mode((800, 600))
pylink.openGraphics()

# open a data file, so we can playback the recorded data
tk.openDataFile('playback.edf')

# calibrate the tracker; one can run in mouse simulation mode
tk.doTrackerSetup()

# run 5 trials of a simple visual distractor task
for i in range(5):

    # start recording
    tk.startRecording(1,1,1,1)

    # a grey fixation, a green target, and a bright distractor on a black
background
    win.fill((0,0,0))
    pygame.draw.circle(win, (128,128,128), (400,300), 10)
    pygame.draw.circle(win, (0,255,0), (400,100), 20)
    pygame.draw.polygon(win, (255,255,255),
[(521,159),(541,139),(561,159),(541,179)])
    pygame.display.flip()

    # wait for a saccadic response
    got_SAC = False
    while not got_SAC:
        dt = tk.getNextData()
        if dt is not None:
            ev_data = tk.getFloatData()
            if dt == pylink.ENDSACC:
                amp_x, amp_y =  ev_data.getAmplitude()
                # jump out of the loop if a saccade >2 deg is detected
```

```
                if hypot(amp_x, amp_y) > 2.0:
                    got_SAC = True

    tk.stopRecording() # stop recording

    #start playback and draw the saccade trajectory
    tk.startPlayBack()
    pylink.pumpDelay(50) # wait for 50 ms so the Host can switch to playback
mode

    smp_pos = []
    smp_timestamp = -32768
    while True:
        smp = tk.getNewestSample()
        if smp is not None:
            if smp.getEye() == 0:
                gaze_pos = smp.getLeftEye().getGaze()
            else:
                gaze_pos = smp.getRightEye().getGaze()
            if smp.getTime() > smp_timestamp:
                smp_pos = smp_pos + [(int(gaze_pos[0]), int(gaze_pos[1]))]
                smp_timestamp = smp.getTime()

                # plot the tracjectory
                if len(smp_pos) > 1:
                    pygame.draw.lines(win, (255,255,255), False, smp_pos, 3)
                    pygame.display.update()

        if tk.getCurrentMode() == pylink.IN_IDLE_MODE:
            break

    # keep the trajectory on screen for a while
    pylink.pumpDelay(1500)
    # clear up the screen
    win.fill((0,0,0))
    pygame.display.flip()
    pylink.pumpDelay(1000)

    # stop playback
    tk.stopPlayBack()

tk.closeDataFile()
tk.close()
pygame.quit()
```

Following the recording start, we draw the visual stimuli (central fixation, target, and distractor) on the screen. Then, we wait for the occurrence of a large (> 2 degrees) saccade. Please note that *getAmplitude()* returns the eye movement displacement in both horizontal and vertical directions. Here we use the *hypot()* function to get the length of the vector.

```
# wait for a saccadic response
got_SAC = False
while not got_SAC:
    dt = tk.getNextData()
    if dt is not None:
        ev_data = tk.getFloatData()
        if dt == pylink.ENDSACC:
            amp_x, amp_y =  ev_data.getAmplitude()
            # jump out of the loop if a saccade >2 deg is detected
            if hypot(amp_x, amp_y) > 2.0:
                got_SAC = True
```

We then call *startPlayBack()* to trigger recording playback and initialize an empty list to store the samples retrieved from the link. In the while loop, we check if there is new sample data; if so, we plot the saccade trajectory up to the current time.

```
smp_pos = []
smp_timestamp = -32768
while True:
    smp = tk.getNewestSample()
    if smp is not None:
        if smp.getEye() == 0:
            gaze_pos = smp.getLeftEye().getGaze()
        else:
            gaze_pos = smp.getRightEye().getGaze()
        if smp.getTime() > smp_timestamp:
            smp_pos = smp_pos + [(int(gaze_pos[0]), int(gaze_pos[1]))]
            smp_timestamp = smp.getTime()

            # plot the trajectory
            if len(smp_pos) > 1:
                pygame.draw.lines(win, (255,255,255), False, smp_pos, 3)
                pygame.display.update()
```

We also check if the playback finishes, and the tracker has switched to the idle mode (*IN_IDLE_MODE*). When this mode switch occurs, we break out the

while-loop and clear the screen for the next trial. Then, we stop the playback mode.

```
if tk.getCurrentMode() == pylink.IN_IDLE_MODE:
    break
```

Figure 6-3 shows a screen generated by this short script. The playback of the samples shows a saccade trajectory that slightly curves away from the diamond distractor.

Figure 6-3 *Playback of the saccade trajectory in a visual distractor task.*

To briefly recap, the EyeLink tracker images the eye(s) at up to 2000 times per second, and each eye image is analyzed to estimate the eye (or gaze) position. In addition to the raw eye position data (samples), the tracker can also detect and record eye events during recording. This chapter illustrates how to access the sample and event data over the Ethernet link during recording. This chapter also explains how to playback a previous recording (trial) while the EDF data file is still open on the EyeLink Host PC. In the following chapters, we will cover

some more advanced topics about the Pylink library (Chapter 7) and discuss data analysis and visualization in Python (Chapter 8).