Chapter 5 Preparing scripts for analysis and visualization in Data Viewer

Data Viewer is a software package developed by SR Research to ease the analysis and visualization of EyeLink data. It is, of course, possible to write a Python script to retrieve the eye movement data needed for a particular analysis (see Chapter 8). Data Viewer is nevertheless the most convenient tool for quick examination and analysis of EyeLink data, as frequently used dependent measures (like dwell time and fixation count for interest areas) are just a few mouse-clicks away.

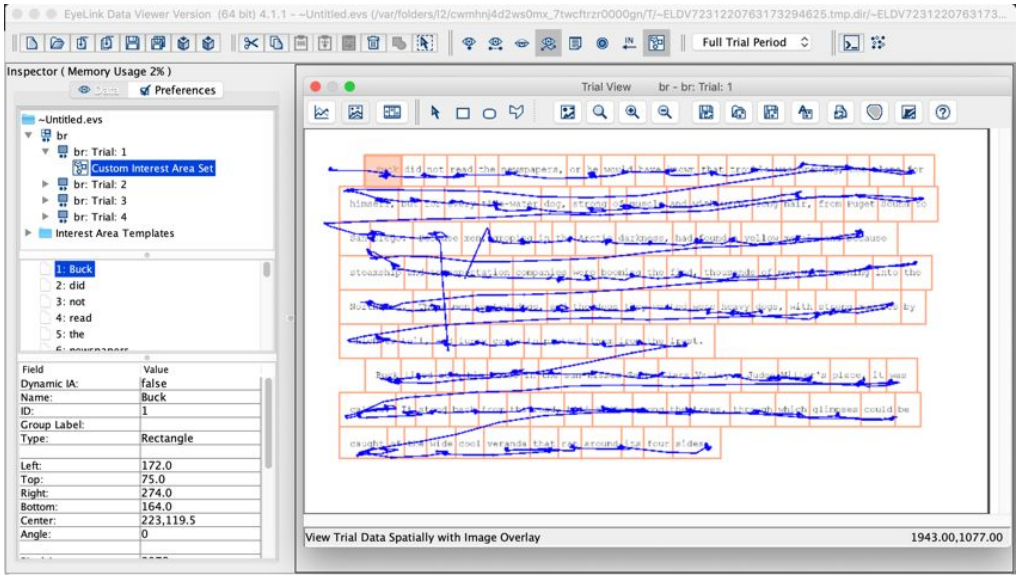

Figure 5-1 *The Data Viewer software developed by SR Research. It is a convenient tool for analyzing and visualizing EyeLink data.*

When analyzing gaze data, it is usually preferable to segment the eye movement recordings into “trials”, i.e., short periods during which we manipulate a particular stimulus or experimental condition. An experiment usually contains tens, if not hundreds of trials, allowing the researcher to estimate the typical (average) pattern of results over trials of any given type. To perform this type of analysis, we need to record some specific information in the EyeLink data file, so that Data Viewer knows: a) the start and end time of each trial, b) which condition or conditions a trial belongs to, c) the onset / offset of critical trial events such as stimulus presentation, d) the locations and onset times of any interest areas associated with a trial, and e) the participant’s responses in a trial (response time, accuracy, etc.). For visualization purposes, it is useful to be able to make the images or videos we displayed during the task available to Data Viewer. The gaze data can then be plotted over the images or videos to facilitate data interpretation and to create meaningful heatmaps, etc. In order to facilitate data analysis in Data Viewer special “Data Viewer Integration” messages can be written by the script into the eye movement data file.

The following sections explain the various Data Viewer integration messages and illustrate their usage with minimal scripts. These scripts are for illustration purposes only. If you need a template for your experimental task, use the example script presented in the final section of this chapter.

Trial segmentation

Data Viewer needs to know the start and end of each trial to segment recordings into trials, much like epoching in EEG data analysis. The start and end of trials are marked by messages that start with the reserved keywords TRIALID, which marks the start of a trial, and TRIAL_RESULT, which marks the end of a trial. Both keywords are in capitals, and the format of these messages is listed below. Note that the flags <Trial ID value list> and <possible trial result values> are optional. As far as the keywords “TRIALID” and “TRIAL_RESULT” are seen, Data Viewer knows they mark the start and end of a trial.

```
TRIALID < Trial ID value list >
TRIAL_RESULT < possible trial result values >
```

Assuming "*tk*" is the tracker connection we have initiated (as in the example scripts in previous chapters), we can include the following line of code at the beginning of each trial.

```
tk.sendMessage('TRIALID 1')
```

In the above message, the optional <Trial ID value list> is "1", which provides additional information about the current trial in addition to the TRIALID keyword.

At the end of a trial, we can include the following line of code to send the "TRIAL_RESULT" message.

```
tk.sendMessage('TRIAL_RESULT 0')
```

Again, I added a flag "0" to the TRIAL_RESULT message, to indicate the trial completed successfully.

As mentioned previously, in most EyeLink experiments the eye tracker is instructed to start / stop recording for each trial. Typically the TRIALID message would be sent before the recording starts and the TRIAL_RESULT message after the recording ends. However, this is not mandatory; in fMRI and EEG studies, the TRIALID and TRIAL_RESULT messages can be sent to the tracker while data recording is ongoing. In this way, you will get a single continuous recording, which contains multiple TRIALID and TRIAL_RESULT messages that allow the Data Viewer software to segment the recording into trials for analysis and visualization purposes.

```
# Filename: trial_segmentation.py
import pylink

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

# open an EDF on the Host
tk.openDataFile('seg.edf')

# run through five trials
```

```
for trial in range(1,6):
    #print a message to show the current trial #
    print("Trial #: %d" % trial)

    # log a TRIALID message to mark trial start
    tk.sendMessage('TRIALID %d' % trial)

    tk.startRecording(1,1,1,1) # start recording
    pylink.pumpDelay(2000) # record for 2-sec
    tk.stopRecording() # stop recording

    # log a TRIAL_RESULT message to mark trial ends
    tk.sendMessage('TRIAL_RESULT 0')

# close data file and retrieve
tk.closeDataFile()
tk.receiveDataFile('seg.edf', 'seg.edf')

# close the link
tk.close()
```

The above script illustrates the usage of trial segmentation messages TRIALID and TRIAL_RESULT. The script is straightforward. For the TRIALID message, we used the % operator to include the current trial number (1, 2, 3, etc.) in the message. When we load the resulting data file ('seg.edf') into Data Viewer, there will be five trials. As is clear from the screenshot shown in Figure 5-2, these five trials are correctly segmented, with the TRIALID and TRIAL_RESULT messages marking the start and end of each trial.

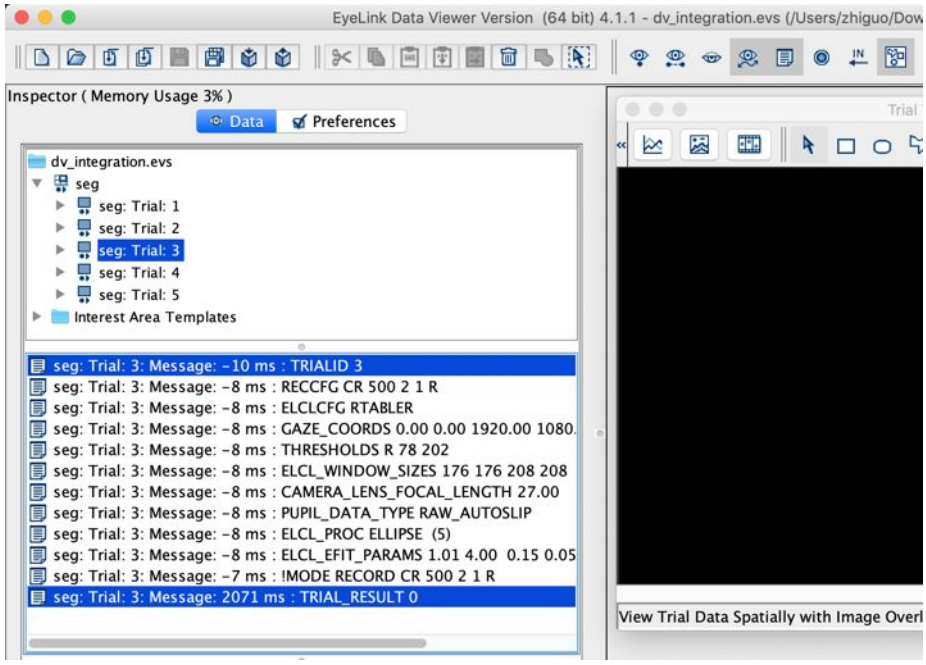

Figure 5-2 *Data Viewer by default uses the keyword "TRIALID" and "TRIAL_RESULT" to segment the eye movement recordings into trials.*

Trial variables

An experimental task usually involves the manipulation of independent variables. For instance, in a Stroop task, we will have trials on which the color of the word is congruent or incongruent with the meaning of the same word (showing the word "RED" in a blue font gives us an incongruent trial). It is generally a good idea to record all variables manipulated in a task in the EDF data file so that we can group the eye movement data based on these variables for visualization or statistical purposes at the analysis stage. In addition to variables manipulated by the experimenter, a trial may also give rise to additional variables like response time and accuracy. These variables can also be useful for eye movement data analysis, e.g., trials can be grouped by correct and incorrect responses, or trials can be excluded if a response was too fast or too slow.

Trial-variable messages should be sent at the end of each trial, allowing variables such as response time, accuracy, etc., to be included. They should be

sent before the TRIAL_RESULT message, which marks the end of the trial. The format of the trial-variable messages is listed below. The prefix "!V" in the message informs Data Viewer that the message is for a particular purpose. Following the prefix is the keyword "TRIAL_VAR" and two additional parameters—<trial_var_label> and <trial_var_value>, i.e., the name of the variable and the value we assigned to the variable in the current trial.

```
!V TRIAL_VAR <trial_var_label > <trial_var_value >
```

We will show how these messages work in the example script at the end of this chapter; please see below for a few example messages, in which the values of the variables "condition," "gap_duration," and "acc" are recorded in the EDF data file. More variables can be recorded in the EDF data file if needed.

```
tk.sendMessage('!V TRIAL_VAR condition step')
tk.sendMessage('!V TRIAL_VAR gap_duration 200')
tk.sendMessage('!V TRIAL_VAR direction Right')
```

After loading the EDF data file into Data Viewer, you can examine the recorded trial variables from the Trial Variable Value Editor. The timestamps of the trial-variable messages do not matter. The variables will be correctly parsed by Data Viewer as long as they are logged between the TRIALID and TRIAL_RESULT messages.

For illustration purposes, we insert the above lines of code into the above example script, just before we send the TRIAL_RESULT message.

```
for trial in range(1,6):
    #print a message to show the current trial #
    print("Trial #: %d" % trial)

    # the TRIALID message marks the start of a new trial
    tk.sendMessage('TRIALID %d' % trial)

    tk.startRecording(1,1,1,1) # start recording
    pylink.pumpDelay(2000) # record for 2-sec
    tk.stopRecording() # stop recording

    # store trial variables in the EDF data file
```

```
    tk.sendMessage('!V TRIAL_VAR condition step')
    tk.sendMessage('!V TRIAL_VAR gap_duration 200')
    tk.sendMessage('!V TRIAL_VAR direction Right')

    # send the TRIAL_RESULT message to mark the end of a trial
    tk.sendMessage('TRIAL_RESULT 0')
```

In Data Viewer, we can examine the recorded variables in the Trial Variable Value Editor (see Figure 5-3). The values of the variables did not vary from trial to trial in this code snippet. This is seldom the case in a real experiment. In the final section of this chapter, we will show a few actual experimental scripts that you can use as templates.

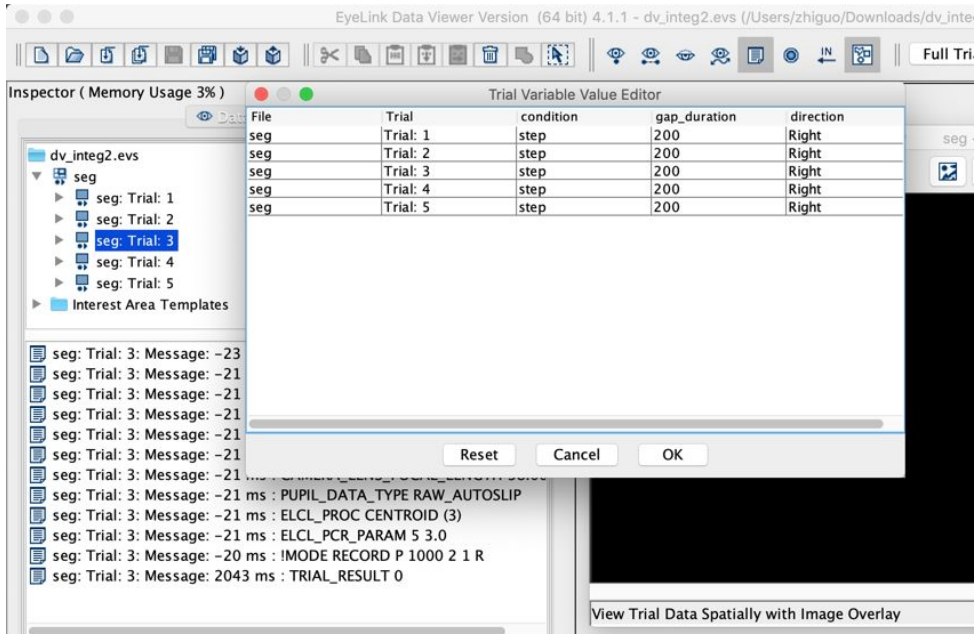

Figure 5-3 *Data Viewer automatically parses the trial-variable messages when importing EDF data files, and the variables / values can be viewed / edited in the messages in the Trial Variable Value Editor.*

Interest Areas

Interest areas (IAs), also known as Areas of Interest (AOIs) or Regions of Interest (ROIs), are indispensable in many analysis protocols. In Data Viewer,

an IA can be rectangular, elliptical, or a “freehand” shape. Each IA has a unique ID (e.g., 1, 2, 3, …) and a meaningful label (optional). While interest areas can be defined manually in Data Viewer after data collection, they can also be defined in experimental scripts. Creating IAs in the experimental script itself has several advantages. It is particularly appropriate for tasks such as Visual Search / Visual World in which the location of Targets / Distractors may change over trials. Special “Interest Area” messages can be written into the EDF files, and these messages will allowData Viewer to display the IAs during analysis. The interest area messages logged during testing can take one of the following formats. The message always starts with the prefix “!V”, followed by the “IAREA” command and a flag specifying the shape of the interest area, i.e., “RECTANGLE,” “ELLIPSE,” and “FREEHAND.” The interest area <id> option is mandatory, and it should be unique for each interest area, but the interest area [label string] is optional.

```
!V IAREA RECTANGLE <id> <left> <top> <right> <bottom> [label string]
!V IAREA ELLIPSE <id> <left> <top> <right> <bottom> [label string]
!V IAREA FREEHAND <id> <x1, y1> <x2, y2 > ... <xn, yn> [label string]
```

For illustration purposes, we insert the following interest area messages into the minimalist style script we have used so far, just following the *tk.startRecording(1,1,1,1)* command. Please bear in mind that Data Viewer will use the timestamps of the interest area messages to determine when to show them during trial playback. In Data Viewer, the details of each interest area can be examined by selecting the “Custom Interest Area Set” field under each trial (see Figure 5-4).

```
tk.sendMessage('!V IAREA ELLIPSE 1 0 0 100 100 head')
tk.sendMessage('!V IAREA RECTANGLE 2 85 85 285 185 body')
tk.sendMessage('!V IAREA FREEHAND 3 285,125 385,50 335,125 tail')
```

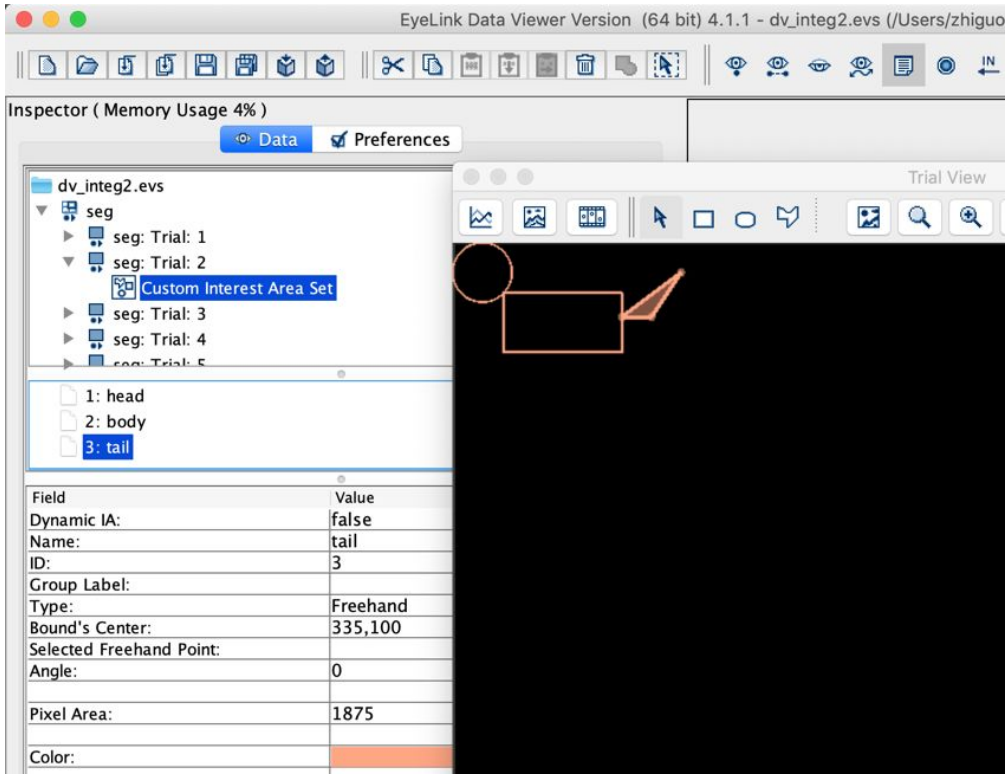

Figure 5-4 *Data Viewer reconstructs the interest areas from the messages in the EDF data files.*

If the experimental task involves a large number of interest areas (e.g., a reading task), all of the interest area definitions can be placed in a single plain text file, known as an interest area set (.ias) file. In the .ias file, multiple interest areas are defined, with each in a single line.

```
RECTANGLE     1     51     29     146     82     Buck
RECTANGLE     2     146    29     195     82     did
RECTANGLE     3     195    29     244     82     not
RECTANGLE     4     244    29     311     82     read
RECTANGLE     5     311    29     360     82     the
RECTANGLE     6     360    29     536     82     newspapers
```

To point Data Viewer to read an interest area set file in the experimental script, send a message in the following format:.

```
!V IAREA FILE <ias_file>
```

Note that interest area set files may also contain the start and end timestamps of each interest area, which can be useful for creating “dynamic” interest areas. For more information on the format of interest area set files and other interest area related issues, please see section 7 of the Data Viewer user manual.

Background graphics

For visualization and data inspection purposes, it is helpful to show the background graphics (e.g., images) together with the gaze data (samples, fixations, saccades, etc.). Some studies may involve the use of hundreds of different screens, so it would be impractical to store all of the images in the EDF data file. A more elegant solution is to embed messages in the EDF data files that contain “pointers” or paths to where the image resources are stored. In that way, when importing the data files, Data Viewer knows where to find the background graphics for each trial. The background graphics can be images, simple drawings, or even videos.

Size of the screen

As noted in Chapter 4, it is crucial to let the tracker know the size of the screen (in pixels), so the tracker can use this information to estimate the velocity of the eye movements (and store gaze in the appropriate coordinate space). To overlay gaze on background graphics, we need to first write a DISPLAY_SCREEN message in the EDF data file. This message informs Data Viewer of the size of the screen, and its format is listed below.

```
DISPLAY_SCREEN <left> <top> <right> <bottom>
```

This message is usually sent to the tracker before data recording begins. Note in the example code below; the DISPLAY_SCREEN message is a reserved keyword and does not have the “!V” prefix, as with the TRIALID and TRIAL_RESULT messages.

```
tk.sendMessage('DISPLAY_SCREEN 1 0 0 100 100 head')
```

Image

For messages specifying the background images, the format can be one of the following. The command "!V IMGLOAD" is followed by a keyword, which specifies the reference point for the image. The "FILL" command will stretch the image to fill the whole screen; the "TOP_LEFT" and "CENTER" commands tell Data Viewer to present the top left corner or the center of the image at the specified screen pixel coordinates, i.e., <x_position> and <y_position>. The [width] and [height] arguments are optional.

```
!V IMGLOAD FILL < image >
!V IMGLOAD TOP_LEFT <image > <x_position> <y_position> [width] [height]
!V IMGLOAD CENTER <image > <x_position> <y_position> [width] [height]
```

Bear in mind that the <image> file name should contain the full path to the image file, relative to the location of the EDF data file itself. Assuming our script folder has the following structure, and the EDF data files are saved in the "*data_files*" folder whereas the images used by the script are saved in the "images" folder:

```
Script_folder
    |__experiment_script.py
    |__images
    |  |__ image_1.png
    |  |__ image_2.png
    |__data_files
       |__test.edf
```

The message sent to the tracker immediately after "*image_1.png*" appears on the screen should be:

```
tk.sendMessage('!V IMGLOAD FILL ./../images/image_1.png')
```

In the line of code above, "." (one dot) represents the current EDF data folder (i.e., "data_files"), ".." (two dots) means to move up one level in the folder hierarchy.

It is important to note that, unlike the trial-variable messages, the timestamps of the background image messages control when the image appears during trial

playback in Data Viewer. As such, these messages should be sent straight after the code that actually draws the image to the screen. For an illustration, please see the free viewing script presented in Chapter 4.

Video

For tasks involving videos, it is crucial to correctly load the video in Data Viewer so that we can appropriately define dynamic interest areas, and assign eye movement data to the correct video frames. To use videos as background graphics, Data Viewer needs to know the exact timing of each frame and also the position of the videos on the screen, of course. For this purpose, we need to write !V VFRAME messages in the experimental scripts. The format of the VFRAME message is as follows.

```
!V VFRAME <frame_number> <pos_x> <pos_y> <movie_file>
```

Note that, for the position of the videos, we need to specify the position of the top-left corner of the video in screen pixel coordinates. Please see the “Example scripts in PsychoPy” section for an example script illustrating the usage of the VFRAME messages.

Simple drawing

Data Viewer also recognizes messages that request it to draw simple graphics as background for gaze overlay. Consider a screen like the one shown below, where two circles are flanking a fixation dot, and the red one is the saccade target. Every time we show this screen to the participant, we could take a screenshot and send the appropriate image loading messages to the tracker so that the image can be loaded into Data Viewer automatically. An alternative and light-weight solution would be to draw the shapes directly in Data Viewer with special messages in predefined formats.

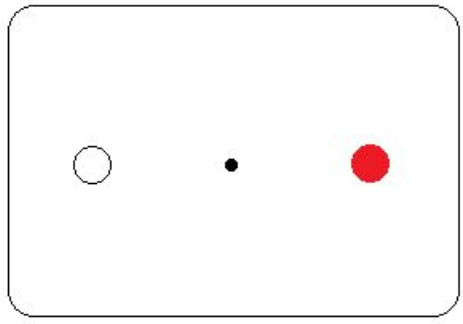

Figure 5-5 *A simple screen used in a saccade task.*

Data Viewer supports multiple drawing commands. The CLEAR command clears the screen; the DRAWLINE command draws a line; the DRAWBOX command draws a rectangle, whereas the FILLBOX command draws a filled box. The FIXPOINT can be used to draw an empty (or full) circle.

```
!V CLEAR <red> <green> <blue>
!V DRAWLINE <red> <green> <blue> <x_start> <y_start> <x_end> <y_end>
!V DRAWBOX <red> <green> <blue> <left> <top> <right> <bottom>
!V FILLBOX <red> <green> <blue> <left> <top> <right> <bottom>
!V FIXPOINT <target_red> <target_green> <target_blue> <erase_red>
<erase_green> <erase_blue> <x> <y> <outer_diameter> <inner_diameter>
```

For the drawing commands, we need to provide the color (in RGB format) and the screen pixel coordinates required for the drawing. For the FIXPOINT command, the target color (<target_*>) is the color for the outline, and the erase color (<erase_*>) is the color for the fixation center.

```
# Filename: simple_drawing.py
import pylink

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

# log the screen size for Data Viewer to draw graphics
tk.sendMessage('DISPLAY_SCREEN 0 0 1024 768')

# open an EDF on the Host
tk.openDataFile('seg.edf')
```

```
# run through five trials of 2-second recordings
for trial in range(1,6):
    #print a message to show the current trial #
    print("Trial #: %d" % trial)

    # the TRIALID message marks the start of a new trial
    tk.sendMessage('TRIALID %d' % trial)

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

    # draw a central fixation flanked by two possible targets
    tk.sendMessage('!V CLEAR 255 255 255') # clear the screen to show white
background
    tk.sendMessage('!V FIXPOINT 0 0 0 0 0 0 512 384 25 0') # central
fixation dot
    tk.sendMessage('!V FIXPOINT 0 0 0 255 255 255 312 384 80 75') #
non-target
    tk.sendMessage('!V FIXPOINT 255 0 0 255 0 0 712 384 80 0') # target

    pylink.pumpDelay(2000) # record for 2-sec
    tk.stopRecording() # stop recording

    # send the TRIAL_RESULT message to mark the end of a trial
    tk.sendMessage('TRIAL_RESULT 0')

# retrieve data file
tk.closeDataFile()
tk.receiveDataFile('seg.edf', 'seg.edf')

# close the link
tk.close()
```

The lines of code needed to reconstruct the screen shown in Figure 5-5 are listed below. Here we use the FIXPOINT command to draw a black dot as the central fixation, an empty circle as the non-target, and filled red circle as the saccade target. Note that we also specified the DISPLAY_SCREEN message, so Data Viewer knows how large the screen should be and where to put the FIXPOINTs. The screen reconstructed by Data Viewer from the messages may not precisely match that presented during testing, but it is good enough for data visualization purposes (see Figure 5-6).

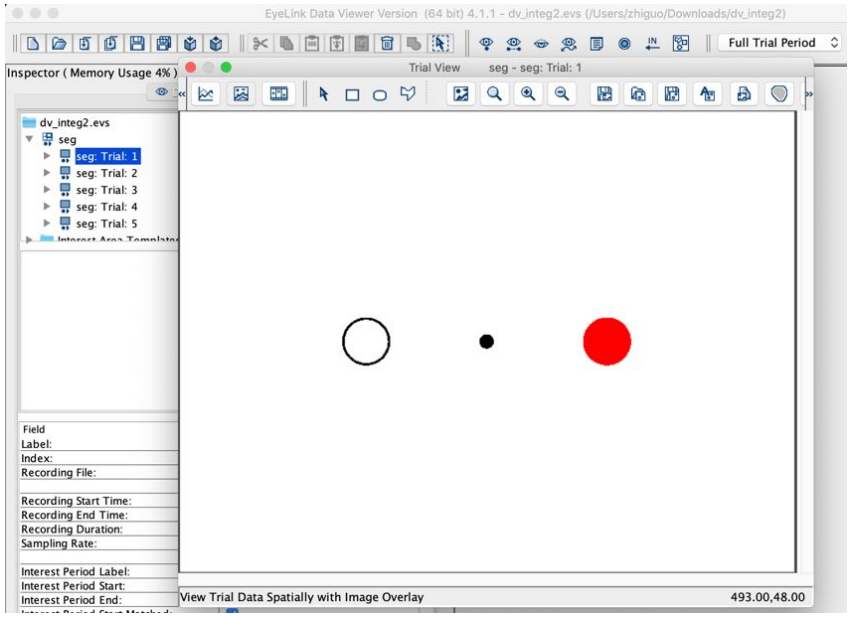

Figure 5-6 *Data Viewer will reconstruct the background graphics from drawing commands recorded in the EDF data files.*

Draw list file

If lots of image loading and drawing messages are needed to construct the stimulus screen in Data Viewer, a single message pointing to a plain text file containing all the commands can be used instead. These files are known as draw list files, and they have the extension of ".dlf" (short for "draw list file"). The draw list file can contain simple drawing commands, as well as commands for loading background images. As with interest area set files, the commands in the draw list files do not require the "!V" prefix. In the example .dlf file below, the first command clears the screen and fills it with white color; the second command draws the image "color.bmp" to the screen pixel position (33, 33); the third command will draw a fixation point at (512, 384), on top of the background image.

```
CLEAR 255 255 255
IMGLOAD TOP_LEFT color.bmp 33 33 772 644
FIXPOINT 0 0 0 255 255 255 512 384 18 4
```

The message referencing a draw list file should be in the following format. As with interest area set files, the <draw list file> parameter should contain the full path to the draw list file, relative to where the EDF data files are stored.

!V DRAW_LIST <draw list file>

Target position

For experiments involving moving targets (i.e., a smooth pursuit task), the position of the target on each screen refresh can be recorded, by logging TARGET_POS messages in the EDF data files. Data Viewer uses these messages to plot target position in the temporal graph view, and to calculate target position and velocity for output in Sample Reports.

```
!V TARGET_POS <Target1 Key> <(target1 x, target1 y)> <target 1 visibility>
<target 1 step>; <Target2 Key> <(target2 x, target2 y)> <target 2
visibility> <target 2 step>
```

Note that a semicolon is required if there are two moving targets. Users can use any keyword to identify the targets. The position of the targets must be in a pair of x, y coordinates in parentheses, e.g., (411, 322); the visibility parameter can be 1 (visible) or 0 (hidden).

The TARGET_POS message is usually not updated until the next screen refresh, i.e., when the screen can change the position of the target. For the time interval between two TARGET_POS messages, the “step” parameter can be set to 0, in which case Data Viewer will interpolate the location of the target across samples in between screen refreshes. If the “step” parameter is set to 1, the target position will be in steps. This parameter affects how the target position data looks like in the sample report generated with Data Viewer. The target position data is useful for deriving dependent measures like gain and phase lag in smooth pursuit tasks. We will present an example script for a smooth pursuit task in the following section to illustrate the use of the TARGET_POS message.

Examples in PsychoPy

In this section, I will present three short examples to illustrate the Data Viewer integration messages discussed in this chapter.

The Stroop task

The first example script implements the classic "Stroop task". As noted by Colin MacLeod in 1992, by that time, "it would be impossible to find anyone in cognitive psychology who does not have at least a passing acquaintance with the Stroop effect." MacLeod's claim still holds nearly three decades later. The Stroop task requires the participant to name the color of the ink in which a color word is printed. For example, the participant would say "blue" in response to the word "red" printed in blue ink. The semantic interference in these "incongruent" trials significantly delays the response, compared to "congruent" trials in which, for example, the word green is printed in green ink.

Here I will present an example script in PsychoPy to illustrate the Stroop task and the Data Viewer integration messages, most notably, the trial-variable messages. The script is heavily commented, but I will further explain some of the critical lines of code.

```
# Filename: Stroop_task.py

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

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

# STEP 2: Open an EDF data file on the Host
tk.openDataFile('stroop.edf')
# add personalized data file header (preamble text)
tk.sendCommand("add_file_preamble_text 'Psychopy Stroop demo'")

# STEP 3: # open a window for graphics and calibration
scnWidth, scnHeight = (1280, 800)
```

```
#always create a monitor object before you run the script
customMon = monitors.Monitor('demoMon', width=35, distance=65)
customMon.setSizePix((scnWidth, scnHeight))

# open a window
win = visual.Window((scnWidth, scnHeight), fullscr=True, monitor=customMon,
                    units='pix', allowStencil=True)

# require Pylink to use custom calibration
graphics--"EyeLinkCoreGraphicsPsychopy.py"
graphics = EyeLinkCoreGraphicsPsychoPy(tk, win)
pylink.openGraphicsEx(graphics)

# STEP 4: Setup Host parameters
# put the tracker in idle mode before we change its parameters
tk.setOfflineMode()
pylink.pumpDelay(100)

# sampling rate, 250, 500, 1000, or 2000; this command does not support
EyeLInk II/I
tk.sendCommand('sample_rate 500')

# send the resolution of the monitor to the tracker
tk.sendCommand("screen_pixel_coords = 0 0 %d %d" % (scnWidth-1,
scnHeight-1))
# save monitor resolution in EDF data file for Data Viewer to correctly load
background graphics
tk.sendMessage("DISPLAY_COORDS = 0 0 %d %d" % (scnWidth-1, scnHeight-1))

# choose a calibration type, H3, HV3, HV5, HV13 (HV = horizontal/vertical),
tk.sendCommand("calibration_type = HV9")

# STEP 5: calibrate the tracker, and run through all the trials
instructions = 'Press LEFT to RED\n\nPress RIGHT to BLEU\n\nPress ENTER to
calibrate tracker'
calInstruct = visual.TextStim(win, text=instructions, color='white', )
calInstruct.draw()
win.flip()
event.waitKeys()

# calibrate the tracker
tk.doTrackerSetup()

# SETP 6: Run through all the trials
```

```
# specify all possible experimental trials
# the columns are 'text', 'textColor', 'correctAnswer' and "congruency"
myTrials = [['red',   'red',  'left',  'cong'],
            ['red',   'blue', 'right', 'incg'],
            ['blue',  'blue', 'right', 'cong'],
            ['blue',  'red',  'left',  'incg']]

# For convenience, here we define a runTrial function to group the lines
# of code executed in each trial
def runTrial(params):
    """ Run a single trial

    params: a list containing tiral parameters, e.g.,
            ['red',   'red',   'left',  'cong']"""

    # unpacking the parameters
    text, textColor, correctAnswer, congruency = params

    # prepare the stimuli
    word = visual.TextStim(win=win, text= text, font='Arial', height=100.0,
color=textColor)

    # take the tracker offline
    tk.setOfflineMode()
    pylink.msecDelay(50)

    # send the standard "TRIALID" message to mark the start of a trial
    tk.sendMessage("TRIALID %s %s %s" % (text, textColor, congruency))

    # record_status_message : show some info on the Host PC
    tk.sendCommand("record_status_message 'word: %s, color: %s'" % (text,
textColor))

    # drift check/correction, params, x, y, draw_target, allow_setup
    try:
        tk.doDriftCorrect(int(scnWidth/2), int(scnHeight/2), 1, 1)
    except:
        tk.doTrackerSetup()

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

```
    # draw the target word in the back video buffer
    word.draw()
    # save a screenshot to use as background graphics in Data Viewer
    if not os.path.exists('screenshots'):
        os.mkdir('screenshots')
    screenshot = 'screenshots/cond_%s_%s.jpg' % (text, textColor)
    win.getMovieFrame('back')
    win.saveMovieFrames(screenshot)

    # flip the window to show the word, then send messages to mark stimulus
onset
    win.flip()
    # record the onset time of the stimuli
    tOnset = core.getTime()
    # message to mark the onset of visual stimuli
    tk.sendMessage("stim_onset")
    # send a Data Viewer integration message here, so DV knows which
screenshot to load
    tk.sendMessage('!V IMGLOAD FILL %s' % ('..' + os.sep + screenshot))

    # Clear bufferred events (in PsychoPy), then wait for key presses
    event.clearEvents(eventType='keyboard')
    gotKey  = False
    keyPressed, RT, ACC = ['None', 'None', 'None']
    while not gotKey:
        keyp = event.getKeys(['left', 'right'])
        if len(keyp) > 0:
            keyPressed = keyp[0] # which key was pressed
            RT = core.getTime() - tOnset # response time
            ACC = int(keyPressed == correctAnswer) # correct=1, incorrect=0

            # send a message mark the key response
            tk.sendMessage("Key_resp %s" % keyPressed)
            gotKey = True

    # clear the window at the end of a trials2Test
    win.color = (0,0,0)
    win.flip()

    # stop recording
    tk.stopRecording()

    # send trial variables to record in the EDF data file
```

```
    tk.sendMessage("!V TRIAL_VAR word %s" % (text))
    tk.sendMessage("!V TRIAL_VAR color %s" % (textColor))
    tk.sendMessage("!V TRIAL_VAR congruency %s" % (congruency))
    tk.sendMessage("!V TRIAL_VAR keyPressed %s" % (keyPressed))
    tk.sendMessage("!V TRIAL_VAR RT %d" % (RT))
    tk.sendMessage("!V TRIAL_VAR ACC %d" % (ACC))

    # send over the standard 'TRIAL_RESULT' message to mark the end of trial
    tk.sendMessage("TRIAL_RESULT %d" % ACC)

# run a block of of 8 trials, in random order
trials2Test = myTrials[:]*2
random.shuffle(trials2Test)
for trial in trials2Test:
    runTrial(trial)

# Step 7: close the EDF data file and put the tracker in idle mode
tk.closeDataFile()
tk.setOfflineMode()
pylink.pumpDelay(100)

# Step 8: copy EDF file to Display PC and put it in local folder ('edfData')
edfTransfer = visual.TextStim(win, text='EDF data is transfering from
EyeLink Host PC ...', color='white')
edfTransfer.draw()
win.flip()

if not os.path.exists('edfData'):
    os.mkdir('edfData')
tk.receiveDataFile('stroop.edf', 'edfData/stroop.edf')

# Step 9: close the connection to tracker, close graphics
tk.close()
core.quit()window and quit PsychoPy
core.quit()
```

Custom graphics for calibration

At the beginning of the script, in addition to the various standard libraries (e.g., *pylink* and *random*) and the PsychoPy modules (*visual*, etc.), we also import a module named "EyeLinkCoreGraphicsPsychoPy." We will discuss this module in detail in Chapter 7. For now, it is sufficient to understand that this module contains a set of functions for drawing the calibration target and the camera image, etc., on the Display PC during tracker calibration.

```
from EyeLinkCoreGraphicsPsychoPy import EyeLinkCoreGraphicsPsychoPy
```

Passing a tracker instance (*tk*) and a PsychoPy window (*win*) to *EyeLinkCoreGraphicsPsychoPy* will create a "graphics environment" (*graphics*). We then pass this graphics environment to *pylink.openGraphicsEx()* to request Pylink to use the custom graphics functions defined in the *EyeLinkCoreGraphicsPsychoPy* module.

```
graphics = EyeLinkCoreGraphicsPsychoPy(tk, win)
pylink.openGraphicsEx(graphics)
```

The DISPLAY_COORDS message

As noted, it is necessary to log a "DISPLAY_COORDS" message in the EDF data file to store the resolution of the screen we use in the lab. Data Viewer needs this information to draw the background graphics and to overlay the gaze correctly.

```
tk.sendMessage("DISPLAY_COORDS = 0 0 %d %d" % (scnWidth-1, scnHeight-1))
```

The TRIALID and TRIAL_RESULT messages

We log the TRIALID and TRIAL_RESULT messages to mark the start and end of a single trial. These messages can contain only the keyword "TRIALID" or "TRIAL_RESULT," but can also include additional information. In the example script above, we put the trial parameters (e.g., text, textColor, and congruency) in the TRIALID message. For the TRIAL_RESULT message, it includes the response accuracy variable (ACC). This additional information is beneficial if you plan to analyze the data with tools other than Data Viewer.

```
tk.sendMessage("TRIALID %s %s %s" % (text, textColor, congruency))
tk.sendMessage("TRIAL_RESULT %d" % ACC)
```

Record status message

As noted in the previous chapter, it can be helpful to show some information about the present trial on the Host PC, so the experimenter can easily monitor the progress of the task and make sure that the participant is actively engaged in the task. In the example script above, the status message shows the word

presented and its color, for instance, “word: blue, color: red,” in the bottom-right corner of the Host PC screen.

```
tk.sendCommand("record_status_message 'word: %s, color: %s'" % (text,
textColor))
```

Background graphics: the IMGLOAD message

The example script above first takes a screenshot of the back buffer in the video memory and saves the screenshot in a folder; then, immediately, the back buffer is “flipped” and displayed on the screen, the following IMGLOAD message is sent to the tracker.

```
tk.sendMessage('!V IMGLOAD FILL %s' % ('..' + os.sep + screenshot))
```

Trial variables messages

For this Stroop task, we store the parameters of each trial in a list. There are two target words (‘red’ and ‘blue’), with each in red or blue, so we have a total of four possible combinations. The four items specified in the list correspond to the word presented, the color of the word, and the correct key response (left or right arrow key). An additional variable (congruency) is included in the list to tell if the current experimental trial is a congruent or incongruent one.

```
# the columns are 'text', 'textColor', 'correctAnswer' and "congruency"
myTrials = [['red',   'red',  'left',  'cong'],
            ['red',   'blue', 'right', 'incg'],
            ['blue',  'blue', 'right', 'cong'],
            ['blue',  'red',  'left',  'incg']]
```

The parameters of each trial, along with the response data (e.g., reaction time and accuracy), should be saved in the EDF data files to facilitate offline data analysis. The example script above includes the following trial variable messages. These variables will show up in the Trial Variable Manager when the EDF data file is loaded into Data Viewer.

```
tk.sendMessage("!V TRIAL_VAR word %s" % (text))
tk.sendMessage("!V TRIAL_VAR color %s" % (textColor))
tk.sendMessage("!V TRIAL_VAR congruency %s" % (congruency))
tk.sendMessage("!V TRIAL_VAR keyPressed %s" % (keyPressed))
```

```
tk.sendMessage("!V TRIAL_VAR RT %d" % (RT))
tk.sendMessage("!V TRIAL_VAR ACC %d" % (ACC))
```

Messages marking critical trial events

To properly analyze and visualize eye movement data in Data Viewer, in addition to the above noted Data Viewer integration messages, it is also important to have messages that mark the occurrence of critical trial events. Without these messages, it would be impossible to tell what events occurred at what time during testing.

```
tk.sendMessage("stim_onset")
```

In the example script above, we send the above message to the tracker when the visual stimuli appear on the screen. When the participant issues a keyboard response, we log this event with the following message.

```
tk.sendMessage("Key_resp %s" % keyPressed)
```

Video playback

Support for video playback PsychoPy 3 has greatly improved. To properly analyze eye movement recorded during video viewing, it is important to be able to play the video in Data Viewer so dynamic interest areas can be defined if needed. The video playback example that accompanies this book shows how to embed Data Viewer integration messages in a PsychoPy script.

```
# Filename: video_playback.py
import pylink, os, random
from psychopy import visual, core, event, monitors
from EyeLinkCoreGraphicsPsychoPy import EyeLinkCoreGraphicsPsychoPy
from psychopy.constants import STOPPED, PLAYING

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

# STEP 2: Open an EDF data file on the Host
tk.openDataFile('video.edf')
# add personalized data file header (preamble text)
tk.sendCommand("add_file_preamble_text 'Psychopy video demo'")
```

```
# STEP 3: # open a window for graphics and calibration
scnWidth, scnHeight = (1280, 800)

#always create a monitor object before you run the script
customMon = monitors.Monitor('demoMon', width=35, distance=65)
customMon.setSizePix((scnWidth, scnHeight))

# open a window
win = visual.Window((scnWidth, scnHeight), fullscr=True, monitor=customMon,
units='pix', allowStencil=True)

# require Pylink to use custom calibration
graphics--"EyeLinkCoreGraphicsPsychopy.py"
graphics = EyeLinkCoreGraphicsPsychoPy(tk, win)
pylink.openGraphicsEx(graphics)

# STEP 4: Setup Host parameters
# put the tracker in idle mode before we change its parameters
tk.setOfflineMode()
pylink.pumpDelay(50)

# sampling rate, 250, 500, 1000, or 2000; this command does not support
EyeLInk II/I
tk.sendCommand('sample_rate 500')

# send the resolution of the monitor to the tracker
tk.sendCommand("screen_pixel_coords = 0 0 %d %d" % (scnWidth-1,
scnHeight-1))
# save monitor resolution in EDF data file for Data Viewer to correctly load
background graphics
tk.sendMessage("DISPLAY_COORDS = 0 0 %d %d" % (scnWidth-1, scnHeight-1))

# choose a calibration type, H3, HV3, HV5, HV13 (HV = horiztonal/vertical),
tk.sendCommand("calibration_type = HV9")

# Step 5: show some instructions and calibrate the tracker.
msg = visual.TextStim(win, text = 'Press Enter twice to calibrate the
tracker', color = 'white', units = 'pix')
msg.draw()
win.flip()
event.waitKeys()

# calibrate the tracker
```

```
tk.doTrackerSetup()

# SETP 6: Run through a couple of trials
# put the videos we would like to play in a list
trials = [['t1', 'Seoul.mp4'],
          ['t2', 'Seoul.mp4']]

# here we define a helper function to group the code executed on each trial
def runTrial(pars):
    """ pars corresponds to a row in the trial list"""

    # retrieve paramters from the trial list
    trial_num, movieFile = pars

    # load the video to display
    mov = visual.MovieStim3(win, filename=movieFile, size=(960, 540))

    # take the tracker offline
    tk.setOfflineMode()
    pylink.msecDelay(50)

    # send the standard "TRIALID" message to mark the start of a trial
    tk.sendMessage("TRIALID %s %s" % (trial_num, movieFile))

    # record_status_message : show some info on the Host PC
    tk.sendCommand("record_status_message 'Trial: %s, movie File: %s'" %
(trial_num, movieFile))

    # drift check/correction, params, x, y, draw_target, allow_setup
    try:
        tk.doDriftCorrect(int(scnWidth/2), int(scnHeight/2), 1, 1)
    except:
        tk.doTrackerSetup()

    # start recording;
    # params: sample_in_file, event_in_file, sampe_over_link,
event_over_link (1-yes, 0-no)
    tk.startRecording(1, 1, 1, 1)
    # wait for 50 ms to cache some samples
    pylink.msecDelay(50)

    # the size of the video
    movWidth, movHeight = mov.size
    # position the movie at the center of the screen
```

```
    movX = int(scnWidth/2-movWidth/2)
    movY = int(scnHeight/2-movHeight/2)

    # play the video till the end
    frameNum = 0
    previousFrameTimeStamp = mov.getCurrentFrameTime()
    while mov.status is not STOPPED:
        # draw a movie frame and flip the video buffer
        mov.draw()
        win.flip()

        # if a new frame is drawn, check frame timestamp and send a VFRAME
message
        currentFrameTimeStamp = mov.getCurrentFrameTime()
        if currentFrameTimeStamp != previousFrameTimeStamp:
            frameNum += 1
            # store a message in the EDF to mark the onset of each video
frame
            tk.sendMessage('Video_Frame: %d' % frameNum)
            # VFRAME message: "!V VFRAME frame_num movie_pos_x, movie_pos_y,
path_to_file"
            tk.sendMessage("!V VFRAME %d %d %d %s" % (frameNum, movX, movY,
"../" + movieFile))
            previousFrameTimeStamp = currentFrameTimeStamp

    tk.sendMessage("Video_terminates")
    # clear the subject display
    win.color=[0,0,0]
    win.flip()

    # stop recording
    tk.stopRecording()

    # send over the standard 'TRIAL_RESULT' message to mark the end of trial
    tk.sendMessage('TRIAL_RESULT')
    pylink.pumpDelay(50)

# run a block of 2 trials, in random order
testList = trials[:]
random.shuffle(testList)
for trial in testList:
    runTrial(trial)

# Step 7: close the EDF data file and put the tracker in idle mode
```

```
tk.closeDataFile()
tk.setOfflineMode()
pylink.pumpDelay(100)

# Step 8: copy EDF file to Display PC and put it in local folder ('edfData')
edfTransfer = visual.TextStim(win, text='EDF data is transferring from
EyeLink Host PC ...', color='white')
edfTransfer.draw()
win.flip()

if not os.path.exists('edfData'):
    os.mkdir('edfData')
tk.receiveDataFile('video.edf', 'edfData/video.edf')

# Step 9: close the connection to tracker, close graphics
tk.close()
core.quit()
```

This script is very similar to the Stroop example. The only lines of code that are worth mentioning are those related to video playback. Once a video is loaded with the *visual.Movie3* module, the *draw()* function is called, and then the video buffers are flipped in a while-loop to show the video frames one by one on the screen.

```
mov = visual.MovieStim3(win, filename=movieFile, size=(960, 540))
while mov.status is not STOPPED:
    mov.draw()
    win.flip()
```

For Data Viewer to correctly playback a video as background graphics, appropriate messages need to be written to the EDF file to let Data Viewer know when to load which video frame. The format of the message is as follows. The message starts with the '!V VFRAME' command, followed by the current frame number, the x and y pixel coordinates of the top-left corner of the video, and the relative path to the video file.

```
!V VFRAME frame_num movie_pos_x, movie_pos_y, path_to_file
```

To send this message in PsychoPy, we first retrieve the *size* of the movie to figure out the correct position of the video.

```
# the size of the video
movWidth, movHeight = mov.size
# position the movie at the center of the screen
movX = int(scnWidth/2-movWidth/2)
movY = int(scnHeight/2-movHeight/2)
```

We can use the *getCurrentFrameTime()* function to retrieve the current frame time relative to the start of a video file. Before the start of video playback, the *getCurrentFrameTime()* function will return a negative timestamp. The script stores this negative timestamp in a variable named *"previousFrameTimeStamp*." In the while loop, we check if the current frame time is different from the *previousFrameTimeStamp* variable; if so, we know a new frame is on the screen, and we send a standard message to mark the onset of a new frame. Then, we send a separate VFRAME message so Data Viewer can use it to draw the video frames during playback. At the end of the while loop, of course, we need to assign the current frame time to the *previousFrameTimeStamp* variable so that we can check if a new frame is on the screen in the next iteration of the loop.

```
# play the video till the end
frameNum = 0
previousFrameTimeStamp = mov.getCurrentFrameTime()
while mov.status is not STOPPED:
    # draw a movie frame and flip the video buffer
    mov.draw()
    win.flip()

    # if a new frame is drawn, check frame timestamp and send a VFRAME
message
    currentFrameTimeStamp = mov.getCurrentFrameTime()
    if currentFrameTimeStamp != previousFrameTimeStamp:
        frameNum += 1
        # store a message in the EDF to mark the onset of each video frame
        tk.sendMessage('Video_Frame: %d' % frameNum)
        # VFRAME message: "!V VFRAME frame_num movie_pos_x, movie_pos_y,
path_to_file"
        tk.sendMessage("!V VFRAME %d %d %d %s" % (frameNum, movX, movY,
"../" + movieFile))
        previousFrameTimeStamp = currentFrameTimeStamp
```

Pursuit task

The smooth pursuit task is frequently used in clinical studies. In the task, the participant is required to follow a visual target that follows a movement pattern, e.g., a sinusoidal movement with their eyes. In the example script shown in this section, the visual target follows a pre-defined circular movement pattern. The code relevant to eye-tracking is mostly the same as in the previous example scripts. The lines of code that are specific to this pursuit task will be discussed in detail below.

```
# Filename: pursuit.py

import pylink, os, random
from psychopy import visual, core, event, monitors
from EyeLinkCoreGraphicsPsychoPy import EyeLinkCoreGraphicsPsychoPy
from math import sin, pi

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

# STEP 2: Open an EDF data file on the Host
tk.openDataFile('pursuit.edf')
# add personalized data file header (preamble text)
tk.sendCommand("add_file_preamble_text 'Psychopy pursuit demo'")

# STEP 3: # open a window for graphics and calibration
scnWidth, scnHeight = (1280, 800)

#always create a monitor object before you run the script
customMon = monitors.Monitor('demoMon', width=35, distance=65)
customMon.setSizePix((scnWidth, scnHeight))

# open a window
win = visual.Window((scnWidth, scnHeight), fullscr=True, monitor=customMon,
units='pix', allowStencil=True)

# require Pylink to use custom calibration
graphics--"EyeLinkCoreGraphicsPsychopy.py"
graphics = EyeLinkCoreGraphicsPsychoPy(tk, win)
pylink.openGraphicsEx(graphics)

# STEP 4: Setup Host parameters
```

```
# put the tracker in idle mode before we change its parameters
tk.setOfflineMode()
pylink.pumpDelay(50)

# sampling rate, 250, 500, 1000, or 2000; this command does not support
EyeLInk II/I
tk.sendCommand('sample_rate 500')

# send the resolution of the monitor to the tracker
tk.sendCommand("screen_pixel_coords = 0 0 %d %d" % (scnWidth-1,
scnHeight-1))
# save monitor resolution in EDF data file for Data Viewer to correctly load
background graphics
tk.sendMessage("DISPLAY_COORDS = 0 0 %d %d" % (scnWidth-1, scnHeight-1))

# choose a calibration type, H3, HV3, HV5, HV13 (HV = horizontal/vertical)
tk.sendCommand("calibration_type = HV9")

# STEP 5: prepare the pursuit target, the clock and the movement parameters
target = visual.GratingStim(win, tex=None, mask='circle', size=25)
pursuitClock = core.Clock()

# parameters for the Sinusoidal movement pattern
# [amp_x, amp_y, phase_x, phase_y, freq_x, freq_y]
mov_pars = [[300, 300, pi*3/2, pi*2, 1.0, 1.0],
            [300, 300, pi*3/2, pi, 1.0, 1.0]]

# Step 6: show some instructions and calibrate the tracker
msg = visual.TextStim(win, text = 'Press Enter twice to calibrate the
tracker', color = 'white', units = 'pix')
msg.draw()
win.flip()
event.waitKeys()

# calibrate the tracker
tk.doTrackerSetup()

# STEP 7: Run through a couple of trials
# here we define a function to group the code that will executed on each
trial
def runTrial(trial_duration, movement_pars):
    """ trial_duration: the duration of the pursuit movement
        movement_pars: [ amp_x, amp_y, phase_x, phase_y, freq_x, freq_y]
        The Sinusoidal movement pattern is determined by the following
```

```
equation
        y(t) = amplitude * sin(frequency * t + phase)
        for a circular or elliptical movements, the phase in x and y
directions
        should be pi/2 (direction matters) """

    # parse the movement pattern parameters
    amp_x, amp_y, phase_x, phase_y, freq_x, freq_y = movement_pars

    # take the tracker offline
    tk.setOfflineMode()
    pylink.msecDelay(50)

    # send the standard "TRIALID" message to mark the start of a trial
    tk.sendMessage("TRIALID")

    # record_status_message : show some info on the Host PC
    tk.sendCommand("record_status_message 'Pursuit demo'")

    # drift check/correction, params, x, y, draw_target, allow_setup
    try:
        tk.doDriftCorrect(int(scnWidth/2-amp_x), int(scnHeight/2), 1, 1)
    except:
        tk.doTrackerSetup()

    # start recording
    # params: sample_in_file, event_in_file, sampe_over_link,
event_over_link (1-yes, 0-no)
    tk.startRecording(1, 1, 1, 1)
    # wait for 50 ms to cache some samples
    pylink.msecDelay(50)

    # movement starts here
    win.flip()
    pursuitClock.reset()

    # send a message to mark movement onset
    tk.sendMessage('Movement_onset')
    while True:
        time_elapsed = pursuitClock.getTime()
        if time_elapsed >= trial_duration:
            break
        else:
            tar_x = amp_x*sin(freq_x * time_elapsed + phase_x)
```

```
            tar_y = amp_y*sin(freq_y * time_elapsed + phase_y)
            target.pos = (tar_x, tar_y)
            target.draw()
            win.flip()
            tk.sendMessage('!V TARGET_POS target %d, %d 1 0' % (tar_x +
int(scnWidth/2), int(scnHeight/2)-tar_y))

    # send a message to mark movement offset
    tk.sendMessage('Movement_offset')
    # clear the subject display
    win.color=[0,0,0]
    win.flip()

    # stop recording
    tk.stopRecording()

    # send over the standard 'TRIAL_RESULT' message to mark the end of trial
    tk.sendMessage('TRIAL_RESULT')
    pylink.pumpDelay(50)

# run a block of 2 trials, in random order
testList = mov_pars[:]
random.shuffle(testList)
for trial in testList:
    runTrial(10.0, trial)

# Step 8: close the EDF data file and put the tracker in idle mode
tk.closeDataFile()
tk.setOfflineMode()
pylink.pumpDelay(100)

# Step 9: copy EDF file to Display PC and put it in local folder ('edfData')
edfTransfer = visual.TextStim(win, text='EDF data is transferring from
EyeLink Host PC ...', color='white')
edfTransfer.draw()
win.flip()

if not os.path.exists('edfData'):
    os.mkdir('edfData')
tk.receiveDataFile('pursuit.edf', 'edfData/pursuit.edf')

# Step 10: close the connection to tracker, close graphics
tk.close()
core.quit()
```

Researchers frequently use the sinusoidal movement pattern for clinical diagnostic purposes. The x, y position of the visual target is determined with the following equations.

```
pos_x = amplitude_x * (frequency_x * t + phase_x)
pos_y = amplitude_y * (frequency_y * t + phase_y)
```

In the equations, the *amplitude_** parameters determine the amplitude of the sine wave. For instance, with an amplitude of 300 pixels in the horizontal direction, the target will move in the -300 to the 300-pixel range, assuming 0 is the center of the screen. The *frequency_** parameters specify how many oscillations occur in a unit time interval (e.g., a second). The *t* parameter specifies how much time has elapsed since movement onset. The *phase_** parameters determine the start phase of the movement, i.e., where to start the movement. For instance, in the horizontal direction, the movement starts (at *t* = 0) from the center if the start phase is 0 [sin(0) =0], from the right if the start phase is pi/2 [sin(pi/2) = 1], and from the left if the start phase is pi*3/2 [sin(pi*3/2) = -1].

By manipulating these parameters, we can flexibly configure the target movement pattern, velocity, and amplitude. For vertical movement, set the amplitude in the horizontal direction to 0 (*amplitude_x = 0*); for horizontal movement, set the amplitude in the vertical direction to 0 (*amplitude_y = 0*). For circular or elliptical movement, the start phase in the horizontal and vertical directions should be pi/2 apart. The movement will be clockwise if the start phase in the vertical direction is pi/2 greater than that in the horizontal direction; the movement will be counterclockwise if the start phase is pi/2 greater in the vertical direction. Other start phase differences between the horizontal and vertical directions will put the target moving on a complex Lissajous curve.

We need the following message to log the target position in the EDF data file. Note that the origin (0,0) of the screen coordinates in PsychoPy is the center of the screen, so we need to convert the target position to refer to the top-left corner of the screen as the origin, which is the location assumed by Data Viewer.

```
tk.sendMessage('!V TARGET_POS target %d, %d 1 0' % (tar_x + int(scnWidth/2),
int(scnHeight/2)-tar_y))
```

The TARGET_POS message is sent to the tracker every time the screen redraws. The last parameter in the TARGET_POS tells Data Viewer what to do with the screen refresh intervals, during which the target position does not change. Here we set this parameter to 0 to ask Data Viewer to interpolate the target position during the screen refresh intervals. In the temporal graph view in Data Viewer, we will see smooth sinusoidal movement in both the horizontal and vertical directions. The movement pattern will show up in Data Viewer if you switch to the temporal graph view (see Figure 5-6).

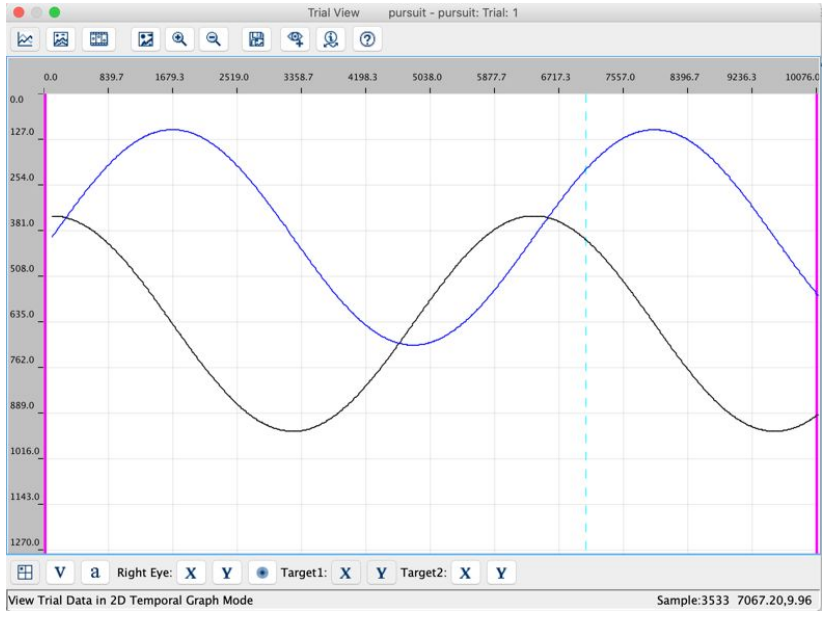

Figure 5-6 *Data Viewer reconstructed the target movement pattern from the TARGET_POS messages recorded in the EDF data files.*

To briefly summarize, this chapter illustrated the various messages that one needs to log into the EDF data files to make them ready for analysis and visualization in the Data Viewer software. We will discuss alternative ways in analyzing and visualizing eye movement data in Chapter 8, but as noted, Data Viewer is likely the handiest tool available. I would encourage you to log the messages discussed in this chapter, even if you have no plan to use Data Viewer

for analysis and visualization. These messages are part of a mature data logging protocol, which can be extremely useful when analyzing your data in Matlab, Python, or any other programming languages or tools.