# Py4ET - Python for Eyetracking

***Release 2013.1***

**Sol Simpson and Jon Peirce**

August 06, 2013

# CONTENTS

Welcome to the Python for Eyetracking Workshop. We haven't given a workshop quite like this before and some of the technologies we'll be showing you are very much at the cutting edge of what has been developed. So bear with us!

CHAPTER

ONE

# PRESENTERS

- **Jon Peirce** created PsychoPy. He's never used an eye-tracker!
- **Sol Simpson** has been contributing to PsychoPy for about a year and has about 12 years experience in eye tracking. He's a PsychoPy *Coder*, and has never used PsychoPy's Experiment Builder!
- **Michael MacAskill** is a long-standing PsychoPy user and contributor, and studies eye-movements and neurological disorders. He's never used Sol's ioHub!

CHAPTER

TWO

# SCHEDULE

## 2.1 9:00 - 9:20am: Workshop facilities

These machines each has an SMI iView-RED eyetracker. Machines 9, 17, 23 don't work!

They are designed only to communicate with this teacher computer. You do not have internet access, and please don't plug a USB stick into these machines.

**Things to check before we carry on:**

- The keyboards are set to Swedish, which you might find confusing so go to the bottom-right of the Windows screen and change the SV to EN
- We need to check that the Eyetracker software is initiated. Open up iView-RED-M on the desktop and go to Advanced so we can check the camera is operating

**If you go to the shared folder on your C drive (see favourites) you should see a *PsychoPyWorkshop* folder in which you'll find**

- Py4ET/index.html *these teaching materials*
- Py4ET.pdf *a pdf replica of these materials*
- demos - PsychoPy files that we'll be using

One other thing - if you want to leave the room you have to press the green key button. If you set the alarm off, then step back, shut the door and press the green key button!

## 2.2 9:50 - 11:20am Introducing PsychoPy

General issues:

- PsychoPy handles a variety of colour spaces and stimulus units but these are a common source of confusion and PsychoPy doesn't check if you've done something sensible!
- PsychoPy expects to control on each frame (each refresh of the screen) what gets drawn. So, especially when writing code, you need to think about controlling things at that level.
- PsychoPy expects a decent graphics card (seriously, avoid intel integrated graphics) and aims to synchronise to the screen. If it can't then stimulus timing cannot be controlled.
- Even when it's a good graphics card you should know the limits of what your computer can manage and be careful, especially when loading images from disk.

### 2.2.1 Coder experiments

PsychoPy has very many stimulus types:

- *ImageStim* for bitmaps (pretty much any format). These can be used either as the image or as an alpha mask

- *GratingStim* is similar to ImageStim but it will use the image as a cyclic texture and you can specify the spatial frequency of that texture
- *RadialStim* for radial patterns (e.g. retinotopy patches)
- *TextStim*, including choice of fonts and Unicode characters (but no real 'formatting')
- *DotStim* for random dot kinematograms with a variety of signal/noise methods
- *ElementArray* allows dynamic (hardware-optimised) presentation of similar-textured elements. e.g. create an array of gabors or a point-light walker
- *Shapes* (basic or custom vertices, filled or unfilled)
- *RatingScale* OK this isn't a stimulus exactly but it's pretty handy
- *Sounds* (either generated or stored in wav files)

Many inputs options:

- mouse, keyboard, joystick
- eyetrackers
- serial/parallel ports
- ...anything that can communicate with your computer!
- microphone, including speech recognition by google!!

Many data output formats:

- 'long-wide' trial-by-trial data in a csv file
- 'summarised' data (one row per condition) in csv or Excel
- Python binary files for scripting analyses
- log files to check timing and for when you forget to store the right stuff!

Now, the best place to work out what's possible is to look through some of the demo code

### 2.2.2 Builder Experiments

#### Stroop

The key concepts you'll need for the Builder is the idea that you have *Components* that are put together in *Routines* and these *Routines* are combined in a *Flow* diagram.

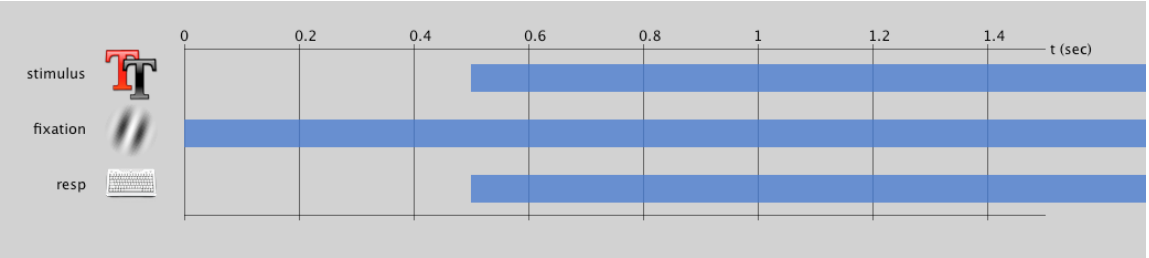
Let's just take a look in the Builder view to see how those work together...

Now the next issue is how to create repetition of the trials in such a way that the condition changes on each repeat. To do that we need to:

1. Change the settings of the experiment

   - click on the Experiment Settings icon (see right)
   - we want to use *pixels* as the units in this experiment, whereas the default is *normalised* (in the user preferences)
2. Set up the basic trial components:
   - create a text Component called *stimulus* in our trial:
     - text height = 100 (we're using pixels)

  - start = 0.5 secs
  - end = <leave blank> (meaning infinite)
  - text = red (for now!)
  - color = blue (for now!)
- create a Grating Component called *fixation*:
  - start = 0
  - end =
  - texture = None
  - mask = circle
  - opacity = 0.7
  - size = 16 (pixels)
- create a Keyboard Component called *resp*:
  - start = 0.5 sec
  - end =
  - *force end routine* = *True*
- save your experiment and run it. Did your one trial look right?

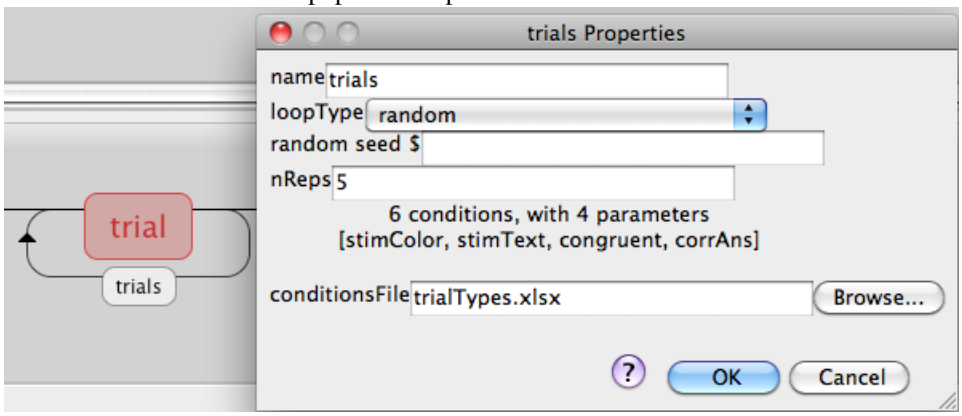



3. Create a conditions file in excel (an xlsx or csv file)
   - columns with headings *stimText*, *stimColor*, *cong*, *corrAns*. These parameters must have no spaces or punctuation. Case matters! They will become variable names in a script.
   - stimText = red, green and blue
   - stimColor = red, green and blue
   - cong = 1 or 0
   - corrAns = left, down, right (for red, green, blue ink colors)

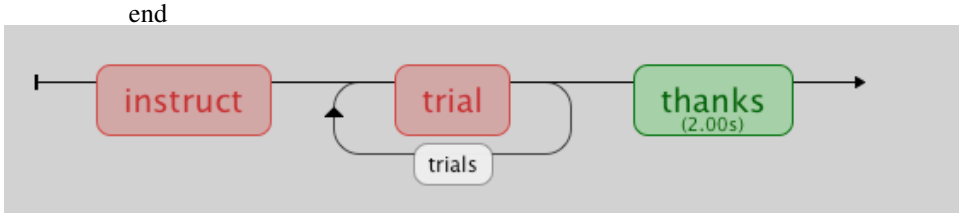
| | A | B | C | D |
|---|---|---|---|---|
| 1 | stimText | stimColor | corrAns | congruent |
| 2 | red | red | left | 1 |
| 3 | red | green | down | 0 |
| 4 | green | green | down | 1 |
| 5 | green | blue | right | 0 |
| 6 | blue | blue | right | 1 |
| 7 | blue | red | left | 0 |
| 8 | | | | |

4. Create a loop in the *Flow* and assign conditions file
   - click on *Insert Loop* (once)

- click (once) where you want the loop to start. On this occasion the other end of the loop is already determined because there's only one other possible place.
- the loop dialog will now appear, in which you can set how the trials (that's a good name for hits loop!) will run
- for the conditions file fetch the file you just created. This should then tell you how many trial types and parameters there are
- set nReps to 1 for now (can add further reps later, when you aren't debugging!)
- now, on every pass around that loop a new random trial (row in the excel file) will be fetched and populate the parameter names as variables




5. Go back and insert those variable names into your *word* stimulus:
   - set the color to be *$stimColor*
   - set the text to be *$stimText*
   - notice when you inserted *$* the font changed. The *$* indicates to PsychoPy that this should not be treated literally - it's a variable or even pure python code.
   - for both these properties **set the value to 'update every repeat'** !!
6. If you like you could create additional Routines for giving instructions and saying thanks at the end




There's a 15min youtube video *building the Stroop <http://www.youtube.com/watch?v=VV6qhuQgsiI>* task from scratch, that will take you through all these steps again if you forget something. The slight difference is that we're going to use pixels as the units in the version we created today.

The finished product is available in the demos folder of the workshop materials.

At the top of the Builder view there is a demos menu. If you unpack them and go back to this menu you can take a look at other example experiments.

### Common errors

My stimulus didn't appear:

- possibly it was smaller than a pixel (0.5 is a sensible size in normalised units but not in pixels)
- possibly it was off the screen; [10, 10] is sensible in pixels but not in normalised units

I got an error that 'xxxxxxxx is not defined':

- did you remember to set that variable to 'set every repeat'. PsychoPy is confused by things that look like variables but it's told are constant?
- did you spell it correctly in the conditions file (case sensitive)?
- have you referred to it in a Routine before the Loop where the variable gets defined?

### 2.2.3 Half-way houses

You can compile your experiment (.psyexp) file into a script (.py) and see what it looks like. But note that this is a one-way street. If you change that file you cannot then go back and make changes in the Builder and expect them to be updated. Once you switch and experiment to being in a script that is how it must stay.

BUT you can also insert code into your Builder Experiment using a Code Component (under the *custom* section of the Components panel. Using this in your Routine you can set chunks of code that will executed at various chosen points in your experiment. (In fact that's all that the other components are - little chunks of code being executed in the same 5 places).

We'll use this concept later with the eye trackers.

## 2.3 11.20 - 12.00 Introducing the ioHub

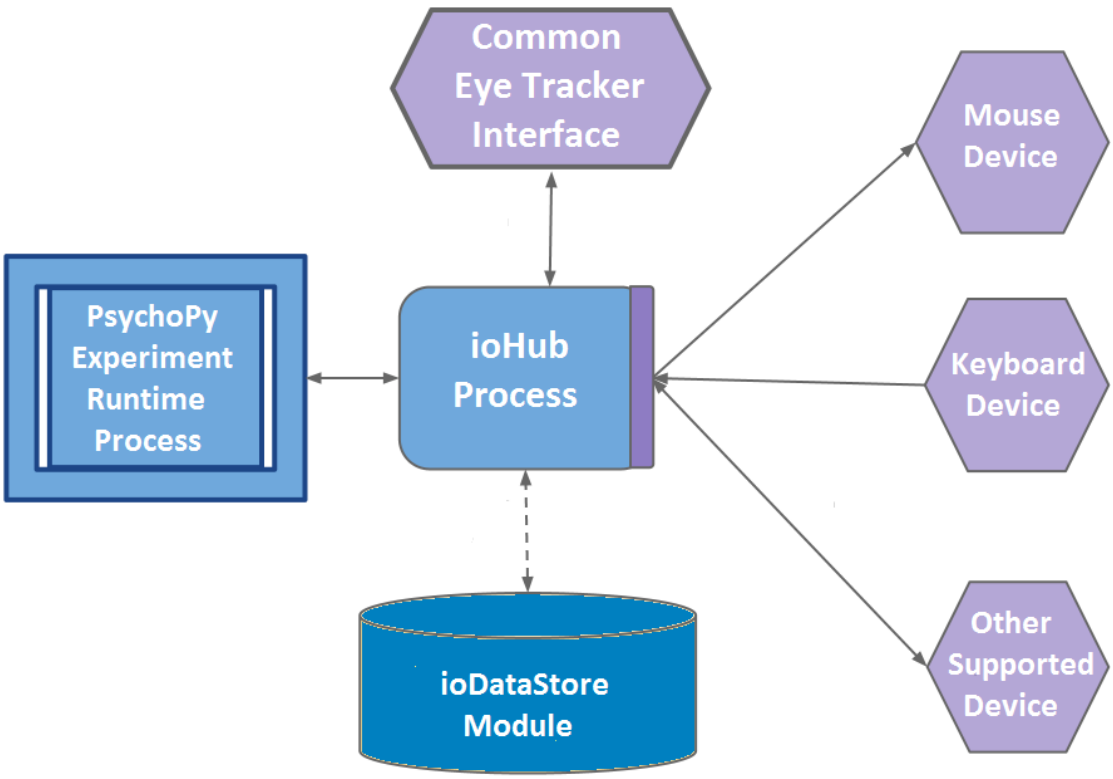
### 2.3.1 Overview

- PsychoPy.ioHub is a Python package providing a cross-platform device event monitoring and storage framework.
- ioHub is free to use and is GPL version 3 licensed.
- **Support for the following operating Systems:**
  - Windows XP SP3, 7, 8
  - Apple OS X 10.6+
  - Linux 2.6+
- **Monitoring of events from computer _devices_ such as:**
  - Keyboard
  - Mouse
  - Analog to Digital Converter
  - XInput compatible gamepads
  - Remote ioHub Server instances
  - **Eye Trackers, via the ioHub Common Eye Tracking Interface**

**The ioHub is designed to solve several issues in experiments involving eyetracking and other high-throughput data collectio**

- Asynchronous from the stimulus presentation process so that all hardware polling or callback handling occurs at a high rate. Events are collected and time stamped if necessary regardless of what the experiment is doing (loading images or videos, waiting for the next retrace start, etc.
- Not only asynchronous but running on a separate core (assuming you have more than one). That means intensive data collection doesn't result in sloppy timing in stimulus presentation
- Can also save high-throughput data to disk using without impacting either stimulus presentation or data collection, and quickly query and access very large data sets.

### PsychoPy.ioHub MultiProcess Design




### Useful PsychoPy.ioHub Links

#### Installation

- ioHub is installed as part of PsychoPy

#### Documentation

*Docs are yet to be merged with the core PsychoPy documentation*

- PsychoPy Docs
- ioHub Docs

#### Support

PsychoPy User Group

#### Want to Contribute?

PsychoPy Developer Group

### High Level ioHub API Review

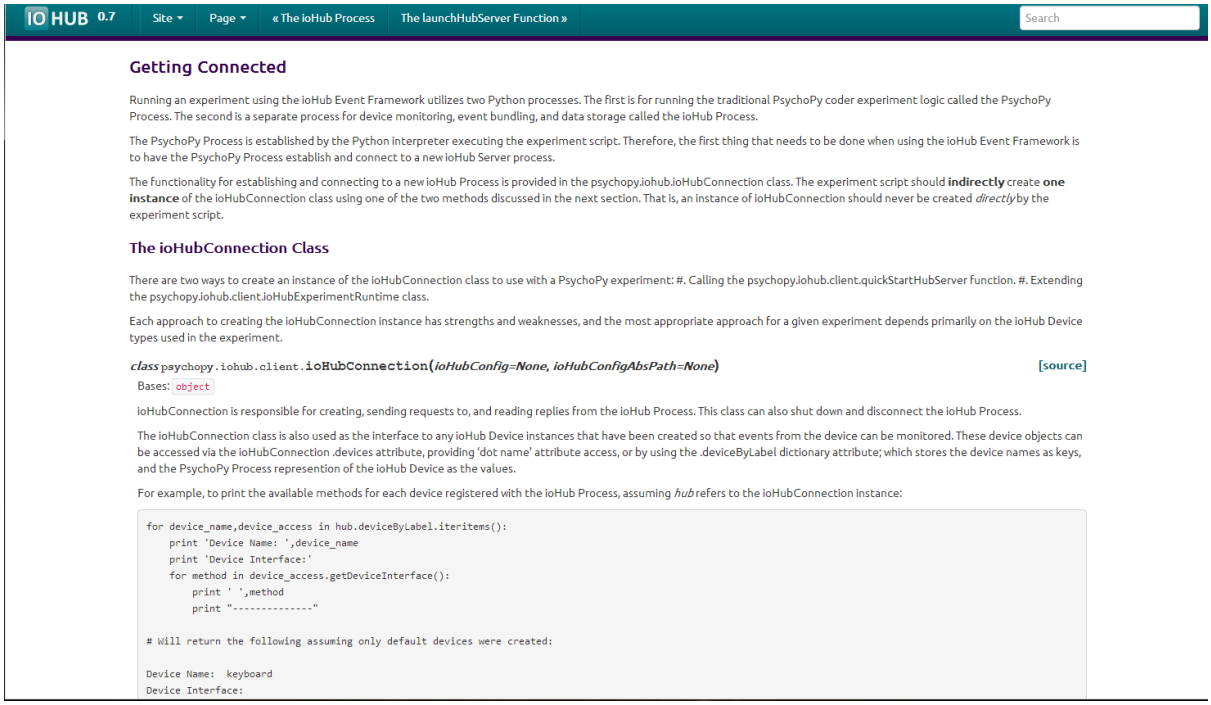
#### Starting the ioHub Server Process

There are three ways to create a PsychoPy experiment which uses the iohub process. All approaches ultimately give you access to an instance of the ioHubConnection Class for communication and control of the iohub server:

1. Directly create a ioHubConnection Class instance.
2. Use the psychopy.iohub.launchHubServer() function.
3. Use the psychopy.iohub.client.ioHubExperimentRuntime, implementing the class's run() method for your experiment's starting python code.

Each approach has pros and cons depending on the devices being used, whether Coder or Buider is being used for experiment creation, etc.

See the ioHub Getting Connected documentation section for details on each approach.




An example python script is available at [workshop_materials_root]python_source/launchHubServer.py, which illustrates how to use the launchHubServer() method. Several other examples included in the workshop materials illustrate how to use the different connection approaches.

## 2.3.2 The ioHub Common Eye Tracker Interface

The API details for the ioHub EyeTracker device can be found here

The Common Eye Tracking Interface provides the same user level Python API for all supported eye tracking hardware, meaning:

- The same experiment script can be run with any supported eye tracker hardware.
- The eye event types supported by an eye tracker implementation are saved using the same format regardless of eye tracker hardware.

### Currently Supported Eye Tracking Systems

- **LC Technologies** EyeGaze and EyeFollower models.

- **SensoMotoric Instruments** iViewX models.
- **SR Research** EyeLink models.
- **Tobii Technologies** Tobii models.

Visit the ioHub documentation for Eye Tracking Hardware Implementations for details on each implementation.

#### Areas of Functionality

1. **Initializing the Eye Tracker / Setting the Device State.**
2. **Performing Calibration / System Setup.**
3. **Starting and Stopping of Data Recording.**
4. Sending Messages or Codes to an Eye Tracker.
5. **Accessing Eye Tracker Data During Recording.**
6. **Accessing the Eye Tracker native time base.**
7. **Synchronizing the ioHub time base with the Eye Tracker time base.**

**Note:** Area's of Functionality in **bold** are considered core areas, and must be implemented for every eye tracker interface.

See the Common Eye Tracker Interface API specification for API details.

### 2.3.3 ioDataStore - Saving Event Data

For relatively small amounts of data you can fetch information from the ioHub while back to the stimulus presentation thread and use PsychoPy's standard data storage facilities. (You would still benefit from the fact that the events had been timestamped by ioHub on collection so it doesn't matter that you only retrieve the data from ioHub once per screen refresh). At other times you might be saving large streams of eye-movement data and you can use ioHub to save the data directly to disk using the HDF5 standardised data format.

#### What can be Stored

- All events from all monitored devices during an experiment runtime.
- Experiment meta-data (experiment name, code, description, version,...)
- Session meta-data (session code, any user defined session level variables)
- Experiment DV and IV information (generally saved on a trial by trial basis, but up to you)

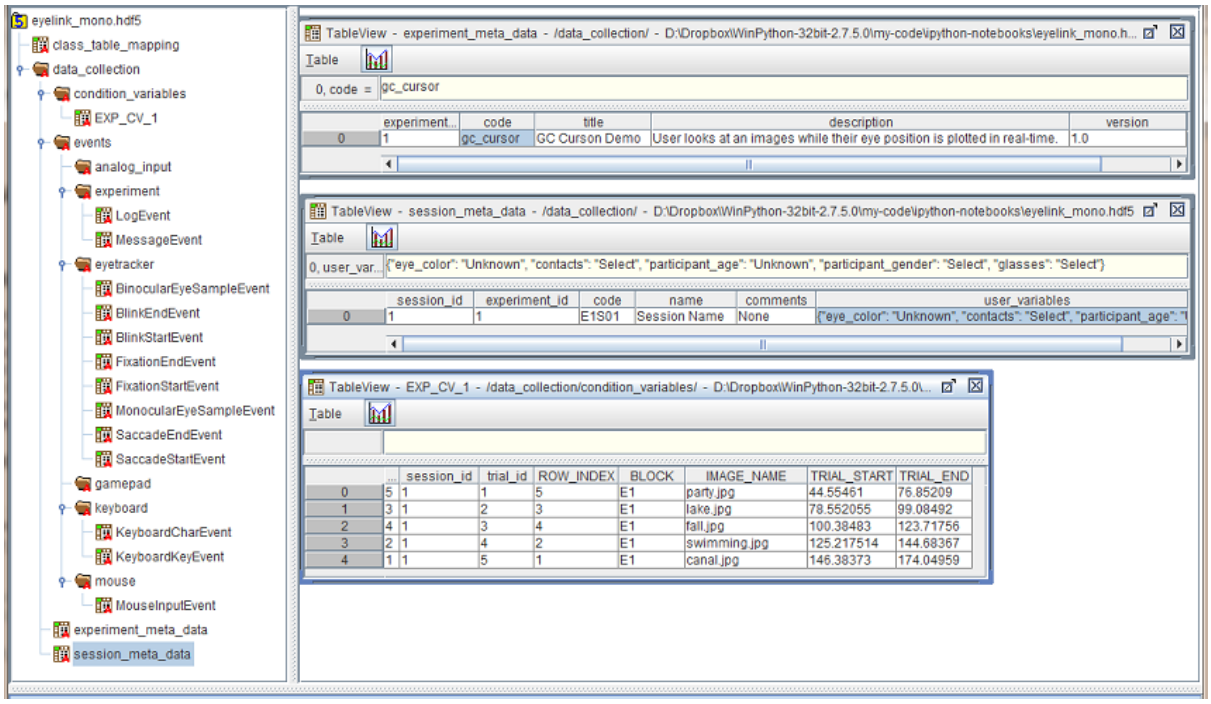
#### Notable Features

- You control what devices save which event types to the data store.
- Data is saved in the standard HDF5 file format
- Pytables is the Python package used to provide HDF5 read / write access.
- Each device event is saved in a table like structure; different event types use different event tables.
- Events are retrieved as numpy ndarrays, and can therefore easily and directly be used by packages such as numpy, scipy, and matplotlib.
- Data Files can be opened and viewed using free HDF5 Viewing tools such as *HDFView* <http://www.hdfgroup.org/hdf-java-html/hdfview/>‘_ and ViTables, both of which are open-source, free, and cross-platform.

## 2.3.4 ioDataStore File Structure

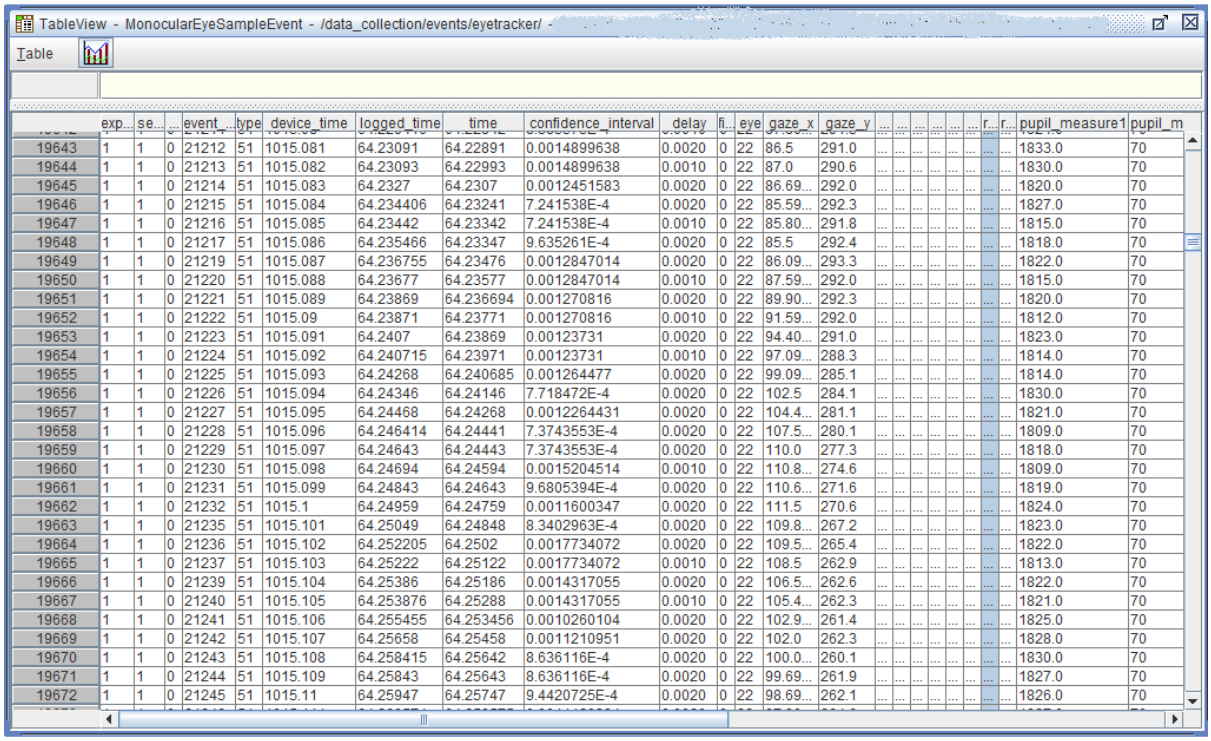
### Hierchial File Structure and Meta-Data Tables

ioDataStore HDF5 File Viewed using the HDFView Application




### Example Event Table: Monocular Eye Sample Event

Example Event Table Viewed using the HDFView Application



### 2.3.5 Reading Saved Data - the ExperimentDataAccessUtility Class

- Contains ioHub device event reading functionality
- Simple event access API
- Access events using the same type constants and event attributes as are used during on-line event access.
- Supports on-disk querying of event tables based on event attribute values and meta-data info.; *fast* retieval of only the events which meet the query constraints.
- **When combined with use of the ExperimentVariableProvider class, events access can be filtered by:**
  - Dependent and Independent Conditions
  - Session and Trial IDs
  - Other Variables Calculated at Runtime, e.g. Trial Start and End Times, Stimulus Onset and Offset Times, etc
  - Any Event Attribute Value

#### Examples of using the ioDataStore Access API

##### Open ExperimentDataAccessUtility and explore ioDataStore file structure

**Source file:** python_source/datastore_examples/printing_datastore_file_structure.py

```
from psychopy.iohub.datastore.util import ExperimentDataAccessUtility

# Create an instance of the ExperimentDataAccessUtility class
# for the selected DataStore file. This allows us to access data
# in the file based on Device Event names and attributes.
#
experiment_data=ExperimentDataAccessUtility('.\hdf5_files' , 'events.hdf5')

# Print the HDF5 Structure for the given ioDataStore file.
#
experiment_data.printHubFileStructure()

# Close the HDF5 File
#
experiment_data.close()
```

##### Accessing Experiment and Session Meta Data

**Source file:** python_source/datastore_examples/access_exp_metadata.py

```
from psychopy.iohub.datastore.util import ExperimentDataAccessUtility

def printExperimentMetaDataDemo():
    # Create an instance of the ExperimentDataAccessUtility class
    # for the selected DataStore file. This allows us to access data
    # in the file based on Device Event names and attributes.
    #
    experiment_data=ExperimentDataAccessUtility('..\hdf5_files' , 'events.hdf5')

    # Access the Experiment Meta Data for the first Experiment found in the file.
    # Note that currently only one experiment's data can be saved in each hdf5 file
    # created. However multiple sessions / runs of the same experiment are all
    # saved in one file.
    #
    exp_md=experiment_data.getExperimentMetaData()[0]
```

```
    printExperimentMetaData(exp_md)

    # Close the HDF5 File
    #
    experiment_data.close()

def printExperimentMetaData(exp_md):
    """
    Function to 'pretty print' the data. Should be added to the
    ExperimentDataAccessUtility class itself.
    """
    exp_md_dict=exp_md._asdict()
    print 'ExperimentMetaData:'
    for fname,fvalue in exp_md_dict.iteritems():
        if fname !='sessions':
            print "{0} : {1}".format(fname,fvalue)
    printSessionMetaData(exp_md.sessions)

def printSessionMetaData(sess_md):
    """
    Function to 'pretty print' the data. Should be added to the
    ExperimentDataAccessUtility class itself.
    """
    for session_info in sess_md:
        sess_md_dict=session_info._asdict()
        print '\tSessionMetaData:'
        for fname,fvalue in sess_md_dict.iteritems():
            print "\t\t{0} : {1}".format(fname,fvalue)

# Run the main demo function
#
printExperimentMetaDataDemo()
```

#### Read Any Saved Trial Condition Variables

**Source file:** python_source/datastore_examples/read_condition_data.py

```
from psychopy.iohub.datastore.util import ExperimentDataAccessUtility
from pprint import pprint

def printExperimentConditionVariableDemo():
    # Create an instance of the ExperimentDataAccessUtility class
    # for the selected DataStore file. This allows us to access data
    # in the file based on Device Event names and attributes.
    #
    experiment_data=ExperimentDataAccessUtility('..\hdf5_files' , 'events.hdf5')

    # Here we are accessing the condition values saved.
    # A list is returned, with each element being the condition variable data
    # for a trial of the experiment, in the order the trials
    # were run for the given session.
    #
    condition_variables=experiment_data.getConditionVariables()

    print "Experiment Condition Variable values:"
    print

    for variable_set in condition_variables:
        pprint(dict(variable_set._asdict()))
        print
    # Close the HDF5 File
```

```
    #
    experiment_data.close()

# Run the main demo function
#
printExperimentConditionVariableDemo()
```

#### List Device Event Types Where the Event Count > 0

**Source file:** python_source/datastore_examples/access_events_with_data.py

```
from psychopy.iohub.datastore.util import ExperimentDataAccessUtility
from psychopy.iohub import EventConstants

def printEventTypesWithDataDemo():
    # Create an instance of the ExperimentDataAccessUtility class
    # for the selected DataStore file. This allows us to access data
    # in the file based on Device Event names and attributes.
    #
    experiment_data=ExperimentDataAccessUtility('..\hdf5_files' , 'events.hdf5')

    # Get any event tables that have >=1 event saved in them
    #
    events_by_type=experiment_data.getEventsByType()

    # print out info on each table
    #
    for event_id, event_gen in events_by_type.iteritems():
        event_constant=EventConstants.getName(event_id)
        print "{0} ({1}): {2}".format(event_constant,event_gen.table.nrows,event_gen)

    # Close the HDF5 File
    #
    experiment_data.close()

# Run the main demo function
#
printEventTypesWithDataDemo()
```

#### Retrieving Specific Event Fields Grouped by Trial using a Trial Condition Query Selection

**Source file:** python_source/datastore_examples/access_single_event_table.py

```
from psychopy.iohub.datastore.util import ExperimentDataAccessUtility
from psychopy.iohub import EventConstants

def printQueriedEventsDemo():
    # Create an instance of the ExperimentDataAccessUtility class
    # for the selected DataStore file. This allows us to access data
    # in the file based on Device Event names and attributes.
    #
    experiment_data=ExperimentDataAccessUtility('..\hdf5_files' , 'events.hdf5')

    # Retrieve the 'time','device_time','event_id','delay','category','text'
    # attributes from the Message Event table, where the event time is between
    # the associated trials condition variables TRIAL_START and TRIAL_END
    # value.
    # i.e. only get message events sent during each trial of the eperiment, not any
    #      sent between trials.
    #
```

```
    event_results=experiment_data.getEventAttributeValues(EventConstants.MESSAGE,
                        ['time','device_time','event_id','delay','category','text'],
                        conditionVariablesFilter=None,
                        startConditions={'time':('>=','@TRIAL_START@')},
                        endConditions={'time':('<=','@TRIAL_END@')})


    for trial_events in event_results:
        print '==== TRIAL DATA START ======='
        print "Trial Condition Values:"
        for ck,cv in trial_events.condition_set._asdict().iteritems():
            print "\t{ck} : {cv}".format(ck=ck,cv=cv)
        print

        trial_events.query_string
        print "Trial Query String:\t"
        print trial_events.query_string
        print

        event_value_arrays=[(cv_name,cv_value) for cv_name,cv_value in trial_events._asdict().ite
        print "Trial Event Field Data:"
        for field_name,field_data in event_value_arrays:
            print "\t"+field_name+': '+str(field_data)
            print
        print '===== TRIAL DATA END ========'

    experiment_data.close()

# Run the main demo function
#
printQueriedEventsDemo()
```

12:00-13:00: Lunch

## 2.4 1:00 - 1:40pm: Incorporating Eye tracking into PsychoPy Experiments

To add eyetracking into your study you will generally:

1. Configure ioHub for the eye tracker to be used ( configuration settings are hardware-dependent )
2. Run the eye tracker setup routine, which will hopefully result in the successful calibration of the ET hardware
3. Start event reporting for the ET device.
4. Monitor eye tracker events or status as needed
5. Inform the eye tracker to stop reporting events.
6. Close the connection to the ET device.

This can be done by writing Python script and using PsychoPy in the Coder mode, or by adding custom python code segments to the PsychoPy Builder. Support for graphically adding eye tracking support and data access from within Builder is expected to occur by end of this year.

### 2.4.1 Using an Eye Tracker from Coder

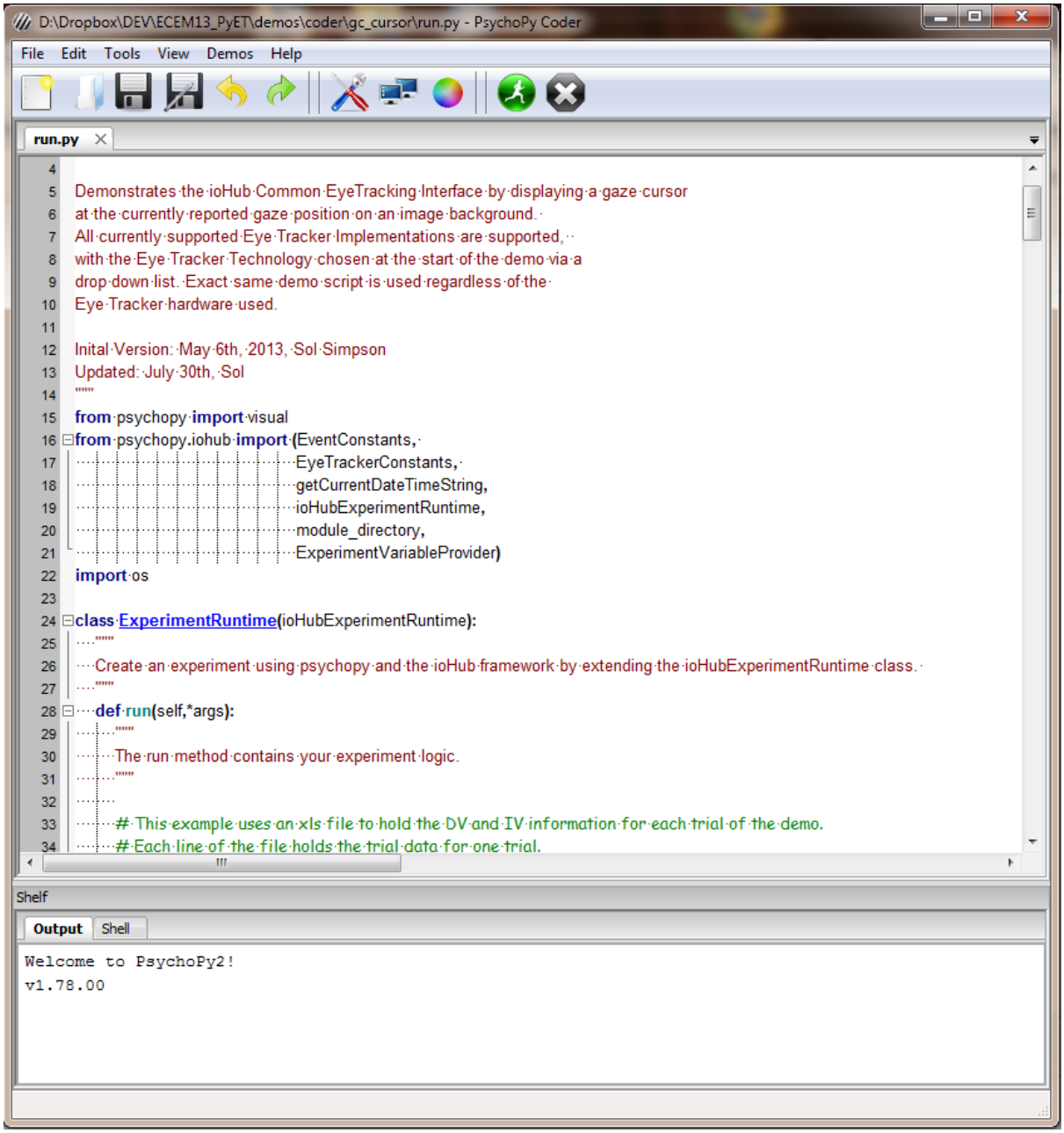
For this Section of the Workshop we will use the PsychoPy Coder.

1. Open the PsychoPy Coder IDE

(a) Start->Programs->PsychoPy2->PsychoPy

2. Ensure the IDE is in *Coder* Mode
   (a) If title of IDE has *Coder* in it, you are in the Coder View.
   (b) Otherwise, select menu View->Open Coder View.
   (c) Close the Builder View.
3. **Open the the getting_started.py demo script:**
   - Select Menu File->Open
   - Python file is found in [Worshop Materials Root]demoscodergetting_startedgetting_started.py

So now you should have the PsychoPy Coder IDE open and it should look soemthing like this:



#### Basic Eye Tracking Coder Example

### 2.4.2 Using an Eye Tracker in Builder

There isn't currently an Eyetracker Component in Builder (I'm sure there will be very soon!) but you can effectively create one yourself using a code component. Remember, these have 5 sections for *Beginning the Experiment*, *Beginning the Routine* (e.g. trial), *Each Frame* of the Routine, *End of the Routine* and *End of the Experiment*.

The way we've set up the demos is that they check first whether you've asked for an eye tracker to be used - in *Experiment Settings* we added an entry to the experiment info dialog box called 'Eye tracker'. In the code below, if that is set to be a string that represents a valid yaml config file then we'll have an eyetracker installed and if not we'll revert to using the mouse as before (handy while creating the experiment in your office!).

#### Stroop:

We'll look at these steps for a new version of the Stroop task where we simply check whether fixation was maintained during the trial and flag trials where it was broken (at any point).

##### Begin the Experiment

Here we need to import and launch the ioHub server and set up some default values for the rest of the experiment (like how large a window we think is reasonable for fixation to be maintained):

```
maintain_fix_pix_boundary=66.0 # pixels
eyetracker =False #will change if we get one!

if expInfo['Eye Tracker']:
    from psychopy.iohub import EventConstants,ioHubConnection,load,Loader
    from psychopy.data import getDateStr

    # Load the specified iohub configuration file converting it to a python dict.
    io_config=load(file(expInfo['Eye Tracker'],'r'), Loader=Loader)

    # Add / Update the session code to be unique. Here we use the psychopy getDateStr() function
    session_info=io_config.get('data_store').get('session_info')
    session_info.update(code="S_%s"%(getDateStr()))

    # Create an ioHubConnection instance, which starts the ioHubProcess, and informs it of the re
    io=ioHubConnection(io_config)

    iokeyboard=io.devices.keyboard
    mouse=io.devices.mouse
    if io.getDevice('tracker'):
        eyetracker=io.getDevice('tracker')

        # Run the eye tracker setup routine.

        win.winHandle.minimize()
        eyetracker.runSetupProcedure()
        win.winHandle.activate()
        win.winHandle.maximize()

    display_gaze=False
    x,y=0,0
```

**Notes:**

- We only import ioHub and set it up if it will be needed!
- We perform the eye tracker setup procedure.

- We should create initial values here for things that will be updated during the script (like the current x,y so that other parts of the script won't throw an error if they use them before the first time the true values are determined)

#### Begin the Routine

Simple code that runs if the eyetracker exists (remember, that started as False but was then assigned an eyetracker object if one was successfully created):

```
if eyetracker:
    heldFixation = True #unless otherwise
    io.clearEvents('all')
    eyetracker.setRecordingState(True)
```

**Notes:**

- at the beginning of the trial we create a variable *heldFixation* and set it to be True. We'll check on each frame if it stays true but this is our default.
- clearing events means we don't worry what happened before the trial started
- we start collecting eye data

#### Each Frame

Now we need to check whether gaze has strayed outside the valid fixation window. But we'll also check whether the user pressed 'g' and if so we'll toggle the *display_gaze* variable.:

```
if eyetracker:
    # check for 'g' key press to toggle gaze cursor visibility
    iokeys=iokeyboard.getEvents(EventConstants.KEYBOARD_PRESS)
    for iok in iokeys:
        if iok.key==u'g':
            display_gaze=not display_gaze
    # get /eye tracker gaze/ position
    gpos=eyetracker.getPosition()
    if type(gpos) in [list,tuple]:
        x,y=gpos
        d=np.sqrt(x**2+y**2)
        if d>maintain_fix_pix_boundary:
            heldFixation = False #unless otherwise
```

**Notes:**

- On each frame we check if the 'g' key has been pressed. If so, we toggle the visibility of a gaze contingent dot.
- We get the latest eye tracker gaze position from the iohub. If it is a tuple or list, we know we have valid eye position info.
- If we have a valid eye position, we calculate the distance between the center of the screen and the eye position reported.
- If the eye to screan center distance is above threshold, we flag that fixation was not maintained for the trial.

#### End of Routine

This is some simple code at the end of the trial that uses the standard data outputs of PsychoPy - a column will appear in the excel/csv file showing whether fixation was held on each trial. We also stop recording eye data at the end of each trial:

```
if eyetracker:
    eyetracker.setRecordingState(False)
    #add eye-track data to data file
    trials.addData("heldFixation", heldFixation)
```

#### End Experiment

At the end of the experiment we close the connection to the eye tracker. Since ioHub runs in a separate process; it's good practice to shut that down just in case it fails to do so itself!:

```
if eyetracker:
    eyetracker.setConnectionState(False)
    io.quit()
```

#### Gaze cursor

**How do we make use of that *display_gaze* variable to show where the gaze is currently located? Simple! We add a *Grating C***

- a small size
- an opacity of *$display_gaze*. That means it uses the display_gaze variable (which is True=1, False=0). Make sure you *set every frame*
- a location of *$[x,y]*. Make sure you *set every frame*

## 2.5 1:40 - 2:40pm Eye Data Visualization

There are seemingly endless ways to visualize and plot data collected from eye tracking devices. Which types of visualization are most relevant depends directly on the application / experimental paradigm the eye tracker is being used in, and the type of data analysis that will be performed on the resulting data collected.

We will review example implementations for the following data visualization / plotting methods:

All the example code for data visualization is available in the python_source/data_visualization folder of the workshop materials.

### 2.5.1 Plotting Eye Position Traces

The basic steps involved in plotting eye position traces collected from an eye tracker via the ioHub Common Eye Tracker Interface are:

1. Read Data From ioDataStore
2. Identify Missing Sample Data Periods (i.e. from Blinks, Eye Occlusion, and Other Causes of Eye Tracking Loss)
3. Plot the Data (we will use Matplotlib)

The following code example outlines these steps. It is assumed that you are running matplotlib using the windowed graph viewer.

```
# This Python Source File Available in python_source/data_visualization/sample_trace_plot.py

from psychopy.iohub.datastore.util import ExperimentDataAccessUtility
from psychopy.iohub import EventConstants

import matplotlib.pyplot as plt
import matplotlib.transforms as mtransforms
```

```
from matplotlib.font_manager import FontProperties

from common_workshop_functions import processSampleEventGaps

import numpy as np

# Load an ioDataStore file containing 120 Hz sample data from a
# remote eye tracker that was recording both eyes. In the plotting example
dataAccessUtil=ExperimentDataAccessUtility('../hdf5_files','remote_data.hdf5', experimentCode=None

##### STEP A. #####
# Retrieve a subset of the BINOCULAR_EYE_SAMPLE event attributes, for events that occurred
# between each time period defined by the TRIAL_START and TRIAL_END trial variables of each entry
# in the trial_conditions data table.
#
event_type=EventConstants.BINOCULAR_EYE_SAMPLE
retrieve_attributes=('time','left_gaze_x','left_gaze_y','left_pupil_measure1',
            'right_gaze_x','right_gaze_y','right_pupil_measure1','status')
trial_event_data=dataAccessUtil.getEventAttributeValues(event_type,
            retrieve_attributes,
            conditionVariablesFilter=None,
            startConditions={'time':('>=','@TRIAL_START@')},
            endConditions={'time':('<=','@TRIAL_END@')},
            )

# No need to keep the hdf5 file open anymore...
#
dataAccessUtil.close()

# Process and plot the sample data for each trial in the data file.
#
for trial_index,trial_samples in enumerate(trial_event_data):
    ##### STEP B. #####
    # Find all samples that have missing eye position data and filter the eye position
    # and pupil size streams so that the eye track plot is more useful. In this case that
    # means setting position fields to NaN and pupil size to 0.
    #
    # left eye manufacturer specific missing data indicator
    left_eye_invalid_data_masks=trial_samples.status//10>=2
    # Right eye manufacturer specific missing data indicator
    right_eye_invalid_data_masks=trial_samples.status%10>=2
    # Get the needed left eye sample arrays
    #
    left_gaze_x=trial_samples.left_gaze_x
    left_gaze_y=trial_samples.left_gaze_y
    left_pupil_size=trial_samples.left_pupil_measure1
    # Process the left eye fields using the processSampleEventGaps function defined
    # in the common_workshop_functions.py file. The last argument of 'clear'
    # tells the function to set any x or y position missing data samples to NaN
    # and to set the pupil size field to 0. The operations are preformed in-place
    # on the numpy arrays passed to the function.
    # The returned valid_data_periods is a list of each group of temporally adjacent
    # samples that are valid, but providing a list where each element is the (start, stop)
    # index for a given period of valid data.
    #
    left_valid_data_periods=processSampleEventGaps(left_gaze_x,left_gaze_y,
        left_pupil_size,
        left_eye_invalid_data_masks,
        'clear')

    # Get the needed right eye sample field arrays
    #
    right_gaze_x=trial_samples.right_gaze_x
```

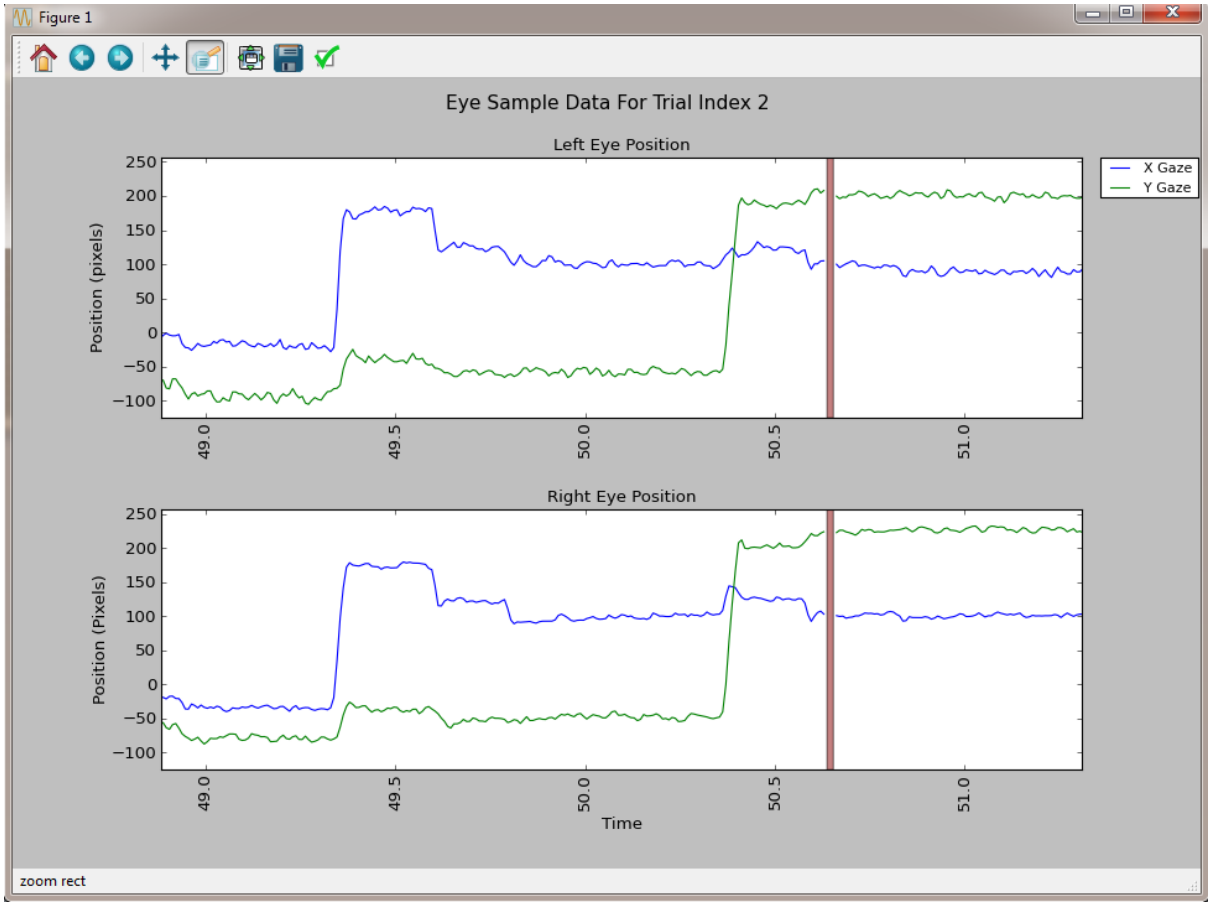
```
right_gaze_y=trial_samples.right_gaze_y
right_pupil_size=trial_samples.right_pupil_measure1

# Process the right eye fields
#
right_valid_data_periods=processSampleEventGaps(right_gaze_x,right_gaze_y,
    right_pupil_size,
    right_eye_invalid_data_masks,
    'clear')

# get the array of sample times for the current trial
time=trial_samples.time
##### STEP C. #####
# Plot the sample traces for x and y gaze positions separately for each eye
# using two sub plots.
#
# Get the range to use for the x axis
tmin=time.min()//1
tmax=time.max()//1+1

#Create a 12x8 inch figure
fig = plt.figure(figsize=(12,8))
# Create a subplot for the left eye, 2,1,1 means the subplot
# grid will be 2 rows and 1 column, and we are about to create
# the subplot for row 1 of 2
#"Left Eye Position"
left_axis = fig.add_subplot(2,1,1)
left_axis.plot(time,left_gaze_x,label="X Gaze")
left_axis.plot(time,left_gaze_y,label="Y Gaze")
plt.xticks(np.arange(tmin,tmax,0.5),rotation='vertical')
# Fill in missing eye data areas of the plot with a vertical bar the full
# height of the sub plot.
trans = mtransforms.blended_transform_factory(left_axis.transData, left_axis.transAxes)
left_axis.fill_between(time, 0, 1, where=left_pupil_size==0,
        facecolor='DarkRed',
        alpha=0.5, transform=trans)
#text(0.5, 0.95, 'test', transform=fig.transFigure, horizontalalignment='center')
left_axis.set_ylabel('Position (pixels)')
# Left Eye Sample Sub Plot
left_axis.set_title("Left Eye Position", fontsize=12)

# Resize the plot x axis by 85% so that the legend , which is outside
# the plot, will still fit in the matplotlib window.
#
box = left_axis.get_position()
left_axis.set_position([box.x0, box.y0, box.width * 0.8, box.height])
fontP = FontProperties()
fontP.set_size('small')
plt.legend(loc='upper left', bbox_to_anchor=(1.02, 1), borderaxespad=0,prop = fontP)
# Right Eye Sample Sub Plot
# Basically the same as the left eye, but we are adding to the row 2 sub plt now.
#
right_axis = fig.add_subplot(2,1,2,sharex=left_axis,sharey = left_axis)
right_axis.plot(time,right_gaze_x,label="X Gaze")
right_axis.plot(time,right_gaze_y,label="Y Gaze")
plt.xticks(np.arange(tmin,tmax,0.5),rotation='vertical')
trans = mtransforms.blended_transform_factory(right_axis.transData, right_axis.transAxes)
right_axis.fill_between(time, 0, 1, where=right_pupil_size==0,
        facecolor='DarkRed',
        alpha=0.5, transform=trans)
right_axis.set_xlabel('Time')
right_axis.set_ylabel('Position (Pixels)')
right_axis.set_title("Right Eye Position", fontsize=12)
```

```
    plt.subplots_adjust(hspace=0.35, bottom=0.125)
    fig.suptitle("Eye Sample Data For Trial Index %d"%(trial_index+1),fontsize=14)

    # Show each trial's eye sample trace. The program will block until you close
    # the trial plot, and will then open the next trial plt.
    #
    plt.show()
```

### Example Plots

Left and Right Eye Position Traces using MatPlotLib


Same Plot as above, zoomed into a Two Second Interval of Eye Position Data




### 2.5.2 Scan Path Visualization of Eye Sample Data

**Scan path overlay plots** are generally created using either:

- the eye tracker sample stream to generate the line segements for the scan path ( **Sample Based Scan Paths** )
- the fixation (and possibly saccade) event stream generated by an event parser ( **Fixation / Event Based Scan Paths** )

When a scan path overlay is sample based, areas of increased dwell time can usually be seen as the clustering of sample line segments within areas of the scene.

When an event based scan path is used, it is common for each fixation event to be plotted as a circle shape, often with each fixation circle diameter being scaled by the fixation event duration. Saccade events can be represented as lines joining the relevent surrounding fixation events, or fixations can simply be joined by lines anchored to each fixations center point, thereby not attempting to use any available saccadic information (angle, duration, etc.).

The scan path overlay code to follow uses an eye sample based approach.

To aid in the visualization of the temportal order of the sample scan trace, a color map is used, with the color ramp normalized to the start and end time of the sample data being plotted (the trial time). This is often a more effective way of relaying temporal information on the 2D scan path plot compared to cluttering the scan path with numbers giving the trial time of eveny Nth scan path segment, or the sample index, etc.

The basic steps involved in scan path visualization using eye data from the ioDataStore include:

1. Read eye sample (or if available eye fixation and sacade event) data from the ioDataStore.
2. Clean areas of data that are determined to have missing eye position fields.
3. Draw the screen background for the trial.

4. Draw eye position data over the screen background. (What you draw is up to you and will depend on the type of eye information being used in the plot).
5. Applying a color map to the scan path data using the time of the scan path data points as the reference into the color map.

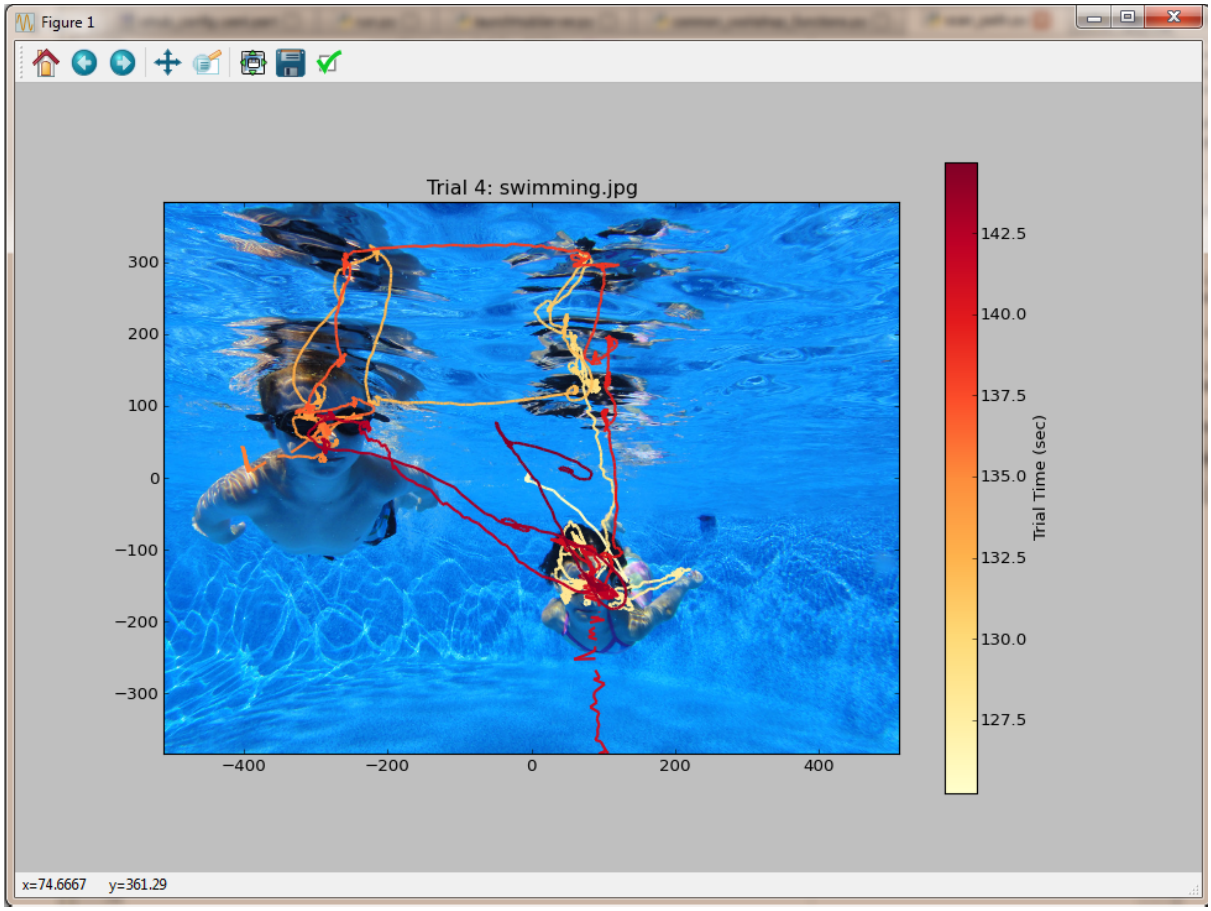
```
# This Python Source File Available in python_source/data_visualization/scan_path.py

from psychopy.iohub.datastore.util import ExperimentDataAccessUtility
from psychopy.iohub import EventConstants

import matplotlib.pyplot as plt
import matplotlib.image as mpimg
from matplotlib.collections import LineCollection

from common_workshop_functions import processSampleEventGaps

import numpy as np

# Load an ioDataStore file containing 1000 Hz sample data from a
# head supported eye tracker that was recording the right eye.
#
dataAccessUtil=ExperimentDataAccessUtility('..\hdf5_files','head_supported_data.hdf5',
                                           experimentCode=None,sessionCodes=[])

##### STEP A. #####
# Retrieve a subset of the MONOCULAR_EYE_SAMPLE event attributes, for events that occurred
# between each time period defined by the TRIAL_START and TRIAL_END trial variables of each entry
# in the trial_conditions data table.
#
event_type=EventConstants.MONOCULAR_EYE_SAMPLE
retrieve_attributes=('time','gaze_x','gaze_y','pupil_measure1','status')
trial_event_data=dataAccessUtil.getEventAttributeValues(event_type,
                            retrieve_attributes,
                            conditionVariablesFilter=None,
                            startConditions={'time':('>=','@TRIAL_START@')},
                            endConditions={'time':('<=','@TRIAL_END@')})

# No need to keep the hdf5 file open anymore...
#
dataAccessUtil.close()

# Process and plot the sample data for each trial in the data file.
#
for trial_index,trial_data in enumerate(trial_event_data):
    plt.close()
    ##### STEP B. #####
    # Find all samples that have missing eye position data and filter the eye position
    # and pupil size streams so that the eye track plot is more useful. In this case that
    # means setting position fields to NaN and pupil size to 0.
    #

    # Eye manufacturer specific missing data indicator
    #
    invalid_data_mask=trial_data.pupil_measure1==0


    # Get the needed left eye sample arrays
    #
    gaze_x=trial_data.gaze_x
    gaze_y=trial_data.gaze_y
    pupil_size=trial_data.pupil_measure1

    # Process the eye fields using the processSampleEventGaps function defined
```

```
# in the common_workshop_functions.py file. The last arguement of 'clear'
# tells the function to set any x or y position missing data samples to NaN
# and to set the pupil size field to 0. The operations are preformed in-place
# on the numpy arrays passed to the function.
# The returned valid_data_periods is a list of each group of temporally adjacent
# samples that are valid, but providing a list where each element is the (start, stop)
# index for a given period of valid data.
#
valid_data_periods=processSampleEventGaps(gaze_x,gaze_y,
                                          pupil_size,
                                          invalid_data_mask,
                                          'clear')

# get the array of sample times for the current trial
#
time=trial_data.time

# Start plotting for the trial
plt.figure(figsize=(12,8))

##### STEP C. #####
# Create Image background for each trial scanpath
# Get the condition variable set used for the current trial
#
condition_set=trial_data.condition_set
# Get the image name and trial_id from the condition data for
# the trial.
#
image_name=condition_set.IMAGE_NAME
trial_id=condition_set.trial_id
plt.title("Trial {0}: {1}".format(trial_id,image_name))
# Load the image
#
trial_image_array=np.flipud(mpimg.imread("./images/"+image_name))
# Get background image size
#
image_size=(trial_image_array.shape[1],trial_image_array.shape[0])
ihw,ihh=image_size[0]/2,image_size[1]/2
# Draw the image to the plot
#
bip=plt.imshow(trial_image_array,origin='lower',extent=(-ihw, ihw,-ihh, ihh))

##### STEP D. #####
# To create the scan path data for the plot, convert the two 1D numpy
# arrays into one 2D array of shape (num_samples,2), where the second
# dimention is the x,y position for every sample in num_samples.
#
sample_points = np.array([gaze_x, gaze_y]).T.reshape(-1, 1, 2)

# To create the csan path graphics, we will create one line for every sample
# position point by specifying the start and end point of each line as
# x1,y1,x2,y2 ... xn-1,yn-1,xn,yn, where n == the number of sample points
# in the trial.
#
sample_segments = np.concatenate([sample_points[:-1], sample_points[1:]], axis=1)

# Create the actual matplotlib line graphics group, and use the built in
# color map 'YlOrRd', meaning Yellow->Orange->Red
#
scan_path_line_collection = LineCollection(sample_segments,
                                           cmap=plt.get_cmap('YlOrRd'),
                                           norm=plt.Normalize(time.min(), time.max()))
scan_path_line_collection.set_array(time)
```

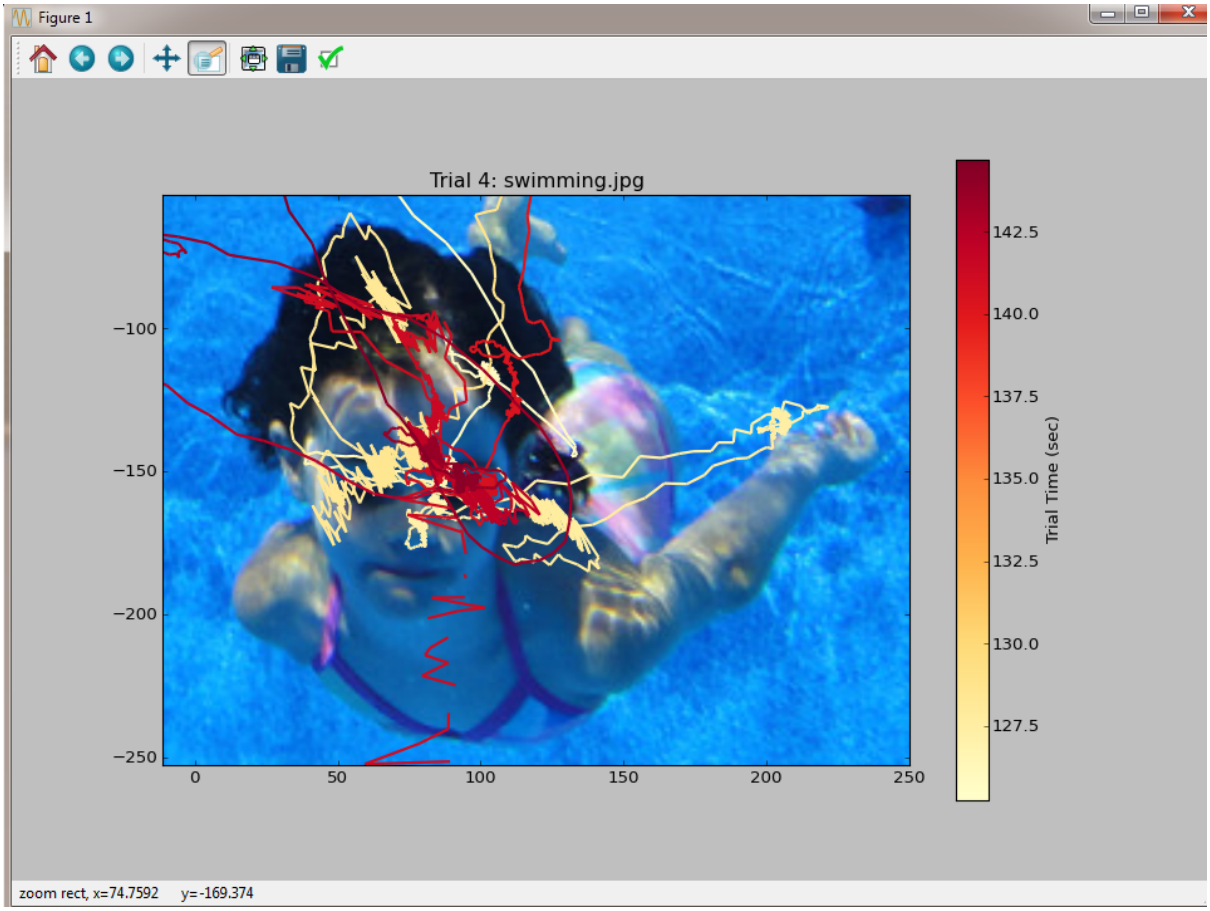
```
scan_path_line_collection.set_linewidth(2)
plt.gca().add_collection(scan_path_line_collection)

##### STEP E. #####
# Display the color bar and different sample times associated with the
# different colors in the color range.
#
cb=plt.colorbar(scan_path_line_collection)

# Give the Color Bar a title
#
cb.set_label("Trial Time (sec)")

plt.show()
```

### Example Plots

Right Eye Scan Path Using Eye Sample Data. Created with MatPlotLib




Same Plot as above, Zoomed into a Small Area of the Image Viewed.




### 2.5.3 Fixation Distribution Using Heat Map Visualization

The basic steps involved in Fixation Distribution Using Heat Map Visualization using eye data from the ioDataStore:

X. Load the eye data from the ioDataStore (in this example we will just create simulated fixation data to spare time). A. Define settings to control how fixation data is used to great the heat map, and how the heat map should look. B. Create 2D Gaussian Mask templateto use as the fixation denisity map for each fixation being used. C. Load the background image displayed during eye data collection. D. In our example, create some random fixation data. E. Create the fixation density map based on the variable values specified in A, the gaussian created in B., and the Fixation data either loaded in X., or in our case, created in D. F. Turn the fixation density map into a heat map by applying a color range to the fixation map data. G. (Optional) Create a figure plotting

> # Gausian mask being used. # Background image the heatmap will be applied to. # The fixation data points used in creating the fixation density map. # The heat map created which will be overlayed on the background image.

8. Create the final fixation denisity based heat map overlayed on the background image.

```
# This Python Source File Available in python_source/data_visualization/heat_map.py

import matplotlib.pyplot as plt
import matplotlib.pylab as plb
import matplotlib.image as mpimg
import matplotlib.cm as cm

from scipy.stats import scoreatpercentile
import numpy as np

plt.close()
```

```
##### STEP A. #####
# Define the value for settings when creating the heat map.
#
# We will use a sigma 33 pixels for the gaussian distribution applied to
# the fixation density map for each fixation position, which
# = ~ 2 visual degrees on a 1024x768 monitor when viewed at 60 cm.
#
sigma_x = sigma_y = 33.0
# If fixation duration is used to weight each fixation when added to the
# fixation density array, these two variables specify the min and max fixation
# duration that will be applied.
#
min_fix_duration=0
max_fix_duration=500
# use_dwell_time_weighting:
# True : Each fixations impact on the fixation map is linearly proprotional
#        to the fixation dwell time within the fixation duration range
#        min_fix_duration to max_fix_duration
# False: Fixations are still filtered by min_fix_duration, max_fix_duration;
#        however each fixation provides equal weight to the fixtion map,
#        regardless of duration.
#
use_dwell_time_weighting=True
# Percentile range of fixation map distribution to include in heat map
# calculation.
#
fix_perc_range=[.05,.95]
# Percentile floor of fixation map distribution for heat map visualization
#
min_fix_dist_perc=10
# We will be creating simulated fixation data, this specifies the number of
# fixation points to create.
#
sim_fix_count=500

##### STEP B. #####
# Create 2D Gaussian Mask template as a 2D numpy array
#
# Create x and y pixel ranges for Gauss Mask.
#
x = np.arange(-sigma_x*2.5,sigma_x*2.5, 1)
y = np.arange(-sigma_y*2.5, sigma_y*2.5, 1)
# Create X and Y pixel position values for each element of Gauss. Mask.
#
X, Y = np.meshgrid(x, y)
# Create 2D Gauss Mask as numpy array using X and Y mesh grid data
# and sigma's, with Gauss centered in 2D array (0,0)
#
gauss=plb.bivariate_normal(X, Y, sigma_x, sigma_y, 0,0)
# Normalize the Gausian, such that the max value in the is 1.0.
#
gauss*=1.0/gauss.flatten().max()
ghw,ghh=gauss.shape[0]//2,gauss.shape[1]//2

##### STEP C. #####
# Load Background Image Displayed During Eye Data Collection
# Flip vertically
#
image_array=np.flipud(mpimg.imread("./images/canal.jpg"))
# Get background image size
#
image_size=image_array.shape#(image_array.shape[0],image_array.shape[1])
```

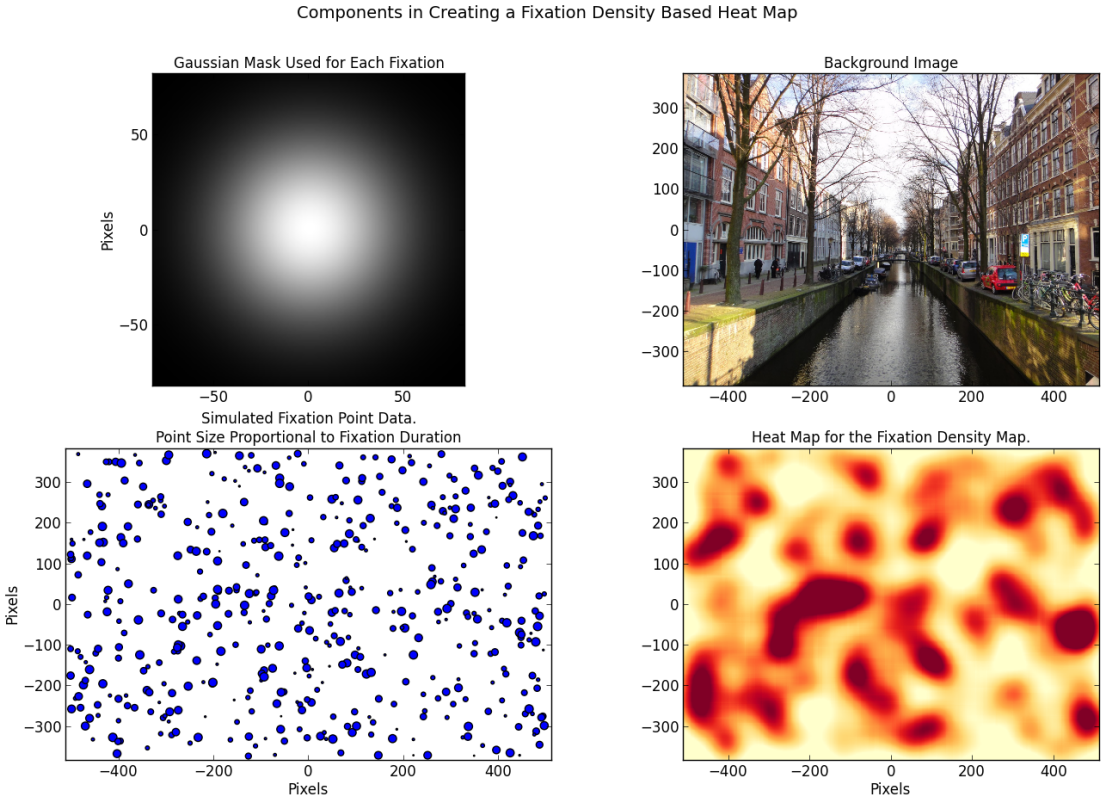
```
ihw,ihh=image_size[0]/2,image_size[1]/2

##### STEP D. #####
# Create some Random Fixation Data
#
# Here, the fixation event data is being simulated as sim_fix_count fixations
# of random position within center 50% of fixation density map (since it
# is created with 2*width, 2*height of the image that the fixation density map
# will be applied to).Random fixation durations between 150 and 1500 msec
# are used.
#
border=10
fix_duration_range=min_fix_duration,max_fix_duration
fixation_x_range=-ihw+border, ihw-border
fixation_y_range=-ihh+border, ihh-border

# Create the dummy Fixation Data as a 3x500 numpy array
#
fix_pos=np.column_stack( (np.random.randint(*fixation_y_range, size=sim_fix_count),
                          np.random.randint(*fixation_x_range, size=sim_fix_count),
                          np.random.randint(*fix_duration_range, size=sim_fix_count)))

##### STEP E. #####
# Create the Fixation Density Map Layer based on
# the Gauss Mask Template and the Fixation Data
# Start with empty 2D numpy array 2x size of background image to be used
# (this makes applying Gauss mask for each fixation easier as array clipping
# is not a consern.). The density map will be trimmed back to the center
# 50% later.
#
fixation_map=np.zeros((image_size[0]*2,image_size[1]*2))

# Apply Gaussian Mask for each fixation position to the density array
# based on the created fixation event data.
#

if use_dwell_time_weighting:
    for fx,fy,fix_dur in fix_pos:
        fx+=ihh*2
        fy+=ihw*2
        fixation_map[fy-ghh:fy+ghh+1,fx-ghw:fx+ghw+1]+=(gauss*fix_dur)
else:
    for fx,fy,fix_dur in fix_pos:
        fx+=ihh*2
        fy+=ihw*2
        fixation_map[fy-ghh:fy+ghh+1,fx-ghw:fx+ghw+1]+=gauss

fixation_map=fixation_map[ihw:image_size[0]+ihw,ihh:image_size[1]+ihh]
# Apply fixation duration and distribution pertentile heuristics to heat map:
#
fixation_map_min=fixation_map.min()
fixation_map_max=fixation_map.max()
fix_range=fixation_map_max-fixation_map_min
fix_range=fixation_map_min+fix_perc_range[0]*fix_range,fixation_map_min+fix_perc_range[1]*fix_ran
min_fix_map_value=scoreatpercentile(fixation_map, min_fix_dist_perc, limit=fix_range)
fix_floor_value=scoreatpercentile(fixation_map, fix_perc_range[0]*100.0)
fix_ceil_value=scoreatpercentile(fixation_map, fix_perc_range[1]*100.0)

fixation_map[fixation_map<fix_floor_value]=fix_floor_value
fixation_map[fixation_map>fix_ceil_value]=fix_ceil_value
fixation_map[fixation_map<min_fix_map_value]=min_fix_map_value
```

```
##### STEP F. #####
# Plot the Fixation Gaussian, the Simulated Fixation Points,
# the resulting Fixation Density Map, and the background image
# to be used for illustrative purposes.
#

#Create a 12x8 inch figure
#
fig = plt.figure(figsize=(14,10))
fig.suptitle("Components in Creating a Fixation Density Based Heat Map",fontsize=14)
# Figure will have 2 x 2 subplots
#
gauss_axis = fig.add_subplot(2,2,1)
#left_axis.set_ylabel('Position (pixels)')
plt.imshow(gauss,cmap=cm.gray,origin='lower',extent=(-ghh, ghh,-ghw, ghw))
gauss_axis.set_title("Gaussian Mask Used for Each Fixation", fontsize=12)
gauss_axis.set_ylabel('Pixels')

# Display the background image.
#
image_axis = fig.add_subplot(2,2,2)
plt.imshow(image_array,origin='lower',extent=(-ihh, ihh,-ihw, ihw))
image_axis.set_title("Background Image", fontsize=12)


# Plot the simulated fixation data.
#
fix_point_axis = fig.add_subplot(2,2,3)
plt.scatter(fix_pos[:,0],fix_pos[:,1],s=fix_pos[:,2]/10)
fix_point_axis.set_title("Simulated Fixation Point Data.\nPoint Size Proportional to Fixation Dura
fix_point_axis.set_ylabel('Pixels')
fix_point_axis.set_xlabel('Pixels')
plt.xlim((-ihh, ihh))
plt.ylim((-ihw, ihw))
# Plot fixation density mask using a Yellow->Orange->Red Color Map,
# clipped to center 50% of
#.
heat_map_axis = fig.add_subplot(2,2,4)
clmap=plt.get_cmap('YlOrRd')
im = plt.imshow(fixation_map, interpolation='nearest',
                                origin='lower',
                                extent=(-ihh, ihh,-ihw, ihw),
                                cmap=clmap)
heat_map_axis.set_title("Heat Map for the Fixation Density Map.", fontsize=12)
heat_map_axis.set_xlabel('Pixels')


plt.tight_layout()
plt.subplots_adjust(left = 0.1, bottom=0.1, top=0.9,hspace=0.2, wspace=.2)

plt.show()

###### STEP G. #####
## Putting it all Together: Heat Map Representation of Fixation Position
## and Dwell Time Density During Image Viewing
##
fig = plt.figure(figsize=(14,10))
fig.suptitle("Fixation Density Heat Map",fontsize=14)

## Draw the background Image
#
image_array=np.flipud(mpimg.imread("./images/canal.jpg"))
plt.imshow(image_array,origin='lower',extent=(-ihh, ihh,-ihw, ihw))
```

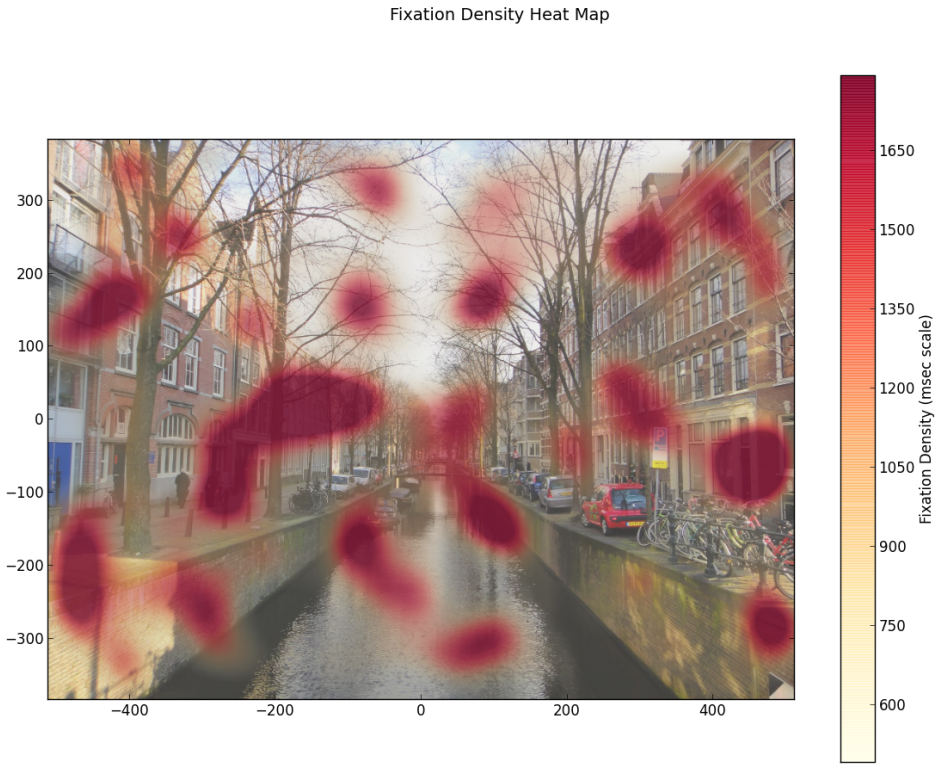
```
# Create RGBA values for the color map created above.
# Set the Color Map Transparency to Increase as a Function of Fixation Dwell Time.
#
clmap._init()
alphas = np.linspace(.3, 0.9, clmap.N+3)
clmap._lut[:,-1] = alphas
# Draw the Fixation Map Mask over the Background Image using the Color Map:
#
plt.imshow(fixation_map, origin='lower',extent=(-ihh, ihh,-ihw, ihw),cmap=clmap)
# Display the Heat Map Scale:
#
cb=plt.colorbar()
if use_dwell_time_weighting:
    cb.set_label("Fixation Density (msec scale)")
else:
    cb.set_label("Fixation Density (count scale)")

plt.show()
```

### Example Plots

Components used in creating a fixation density based heat map.




The final heat map result.




### 2.5.4 Animated Gaze Overlay

Here we illustrate a cute, but perhaps not so scientifically useful, animated gaze position overlay cursor.

The basic steps involved in creating an Animated Gaze Overlay using eye data from the ioDataStore:

1. Load the eye sample data from the ioDataStore.
2. Clean position and pupil data for samples tagged as having missing eye data.
3. Load the background image used for the trial selected for the gaze overlay playback.
4. Create the matplotlib animated figure, including the gaze overlay graphic.
5. Start the animation.

```
# This Python Source File Available in python_source/data_visualization/gaze_overlay_animation.py

# This is a slightly more advanced demo to show you fancy things like animations
#
# It's based on a how-to about python plotting animations at
# http://nickcharlton.net/posts/drawing-animating-shapes-matplotlib.html
# It is also possible to save the animation as a video file

from psychopy.iohub.datastore.util import ExperimentDataAccessUtility
from psychopy.iohub import EventConstants

#import some maths/plotting libs
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
from matplotlib import animation

#import our own helper funcs (from the python_source folder)
```

```
from common_workshop_functions import processSampleEventGaps

##### STEP A. #####
# Load an ioDataStore file containing 1000 Hz sample data from a
# head supported eye tracker that was recording the right eye.
dataAccessUtil=ExperimentDataAccessUtility('../hdf5_files',
                                           'head_supported_data.hdf5',
                                           experimentCode=None,
                                           sessionCodes=[])

TRIAL_ID=1 #we'll just play back a single trial here
et_sampling_rate=1000.0 #eye tracker sampling rate
desired_playback_rate=20 #what rate (in Hz) will we update our figure (not every eye frame!)

# Retrieve a subset of the MONOCULAR_EYE_SAMPLE event attributes, for events that occurred
# between each time period defined by the TRIAL_START and TRIAL_END trial variables of each entry
# in the trial_conditions data table.
event_type=EventConstants.MONOCULAR_EYE_SAMPLE
retrieve_attributes=('time','gaze_x','gaze_y','pupil_measure1','status')
trial_event_data=dataAccessUtil.getEventAttributeValues(event_type,
                            retrieve_attributes,
                            conditionVariablesFilter=None,
                            startConditions={'time':('>=','@TRIAL_START@')},
                            endConditions={'time':('<=','@TRIAL_END@')})

# No need to keep the hdf5 file open anymore...
dataAccessUtil.close()

# Get the data for the one trial we will playback
trial_data=trial_event_data[TRIAL_ID]

##### STEP B. #####
# Get the needed left eye sample arrays
gaze_x=trial_data.gaze_x
gaze_y=trial_data.gaze_y
pupil_size=trial_data.pupil_measure1
# get the array of sample times for the current trial
time=trial_data.time
#clear absent data
invalid_data_mask = (trial_data.pupil_measure1==0) #vendor specific codes
valid_data_periods=processSampleEventGaps(gaze_x,gaze_y,
                                          pupil_size,
                                          invalid_data_mask,
                                          'clear')

##### STEP C. #####
# Load the image used in the current trial.
# get the trial condition values used for each trial in example experiment.
condition_set=trial_data.condition_set
# Get the image name used for display during the trial
image_name=condition_set.IMAGE_NAME
trial_id=condition_set.trial_id

# load the image as a numpy array
trial_image_array=np.flipud(mpimg.imread("./images/"+image_name))

# Reduce size for easier viewing
w, h = trial_image_array.shape[1]/2, trial_image_array.shape[0]/2

##### STEP D. #####
# Create the Animated Figure
dpi = 100
margin = 0.05 # (add 5% of the width/height of the figure...)
```

```
# Make a figure big enough to accomodate an axis of xpixels by ypixels
# as well as the ticklabels, etc...
figsize =  w / dpi,  h / dpi
fig = plt.figure(figsize=figsize, dpi=dpi)
plt.title("Trial %i: %s" %(trial_id,image_name))
# get the current axes
ax = fig.gca()

# Draw the background image array
ax.imshow(trial_image_array,origin='lower',extent=(-w/2, w/2,-h/2, h/2))

# Create a circle graphic to use as the gaze overlay cursor.
circle = plt.Circle((1000, 1000), radius=9, facecolor='r',edgecolor='y',
                    linewidth=2, alpha=0.7)

# Create a semi-transparent text box to display the current trial time.
time_text = ax.text(0.02, 0.95, '', color='black', fontsize=12,
                    bbox={'facecolor':'red', 'alpha':0.5, 'pad':10},
                    transform=ax.transAxes)

# Calculate the eye trackers sampling rate in msec.
ifi=1000.0/et_sampling_rate
# Calculate how many samples occur within the requested playback rate.
sample_frame_interval=desired_playback_rate//ifi+1 #note that // means integer divide
actual_playback_rate=int(sample_frame_interval*ifi) #true rate after rounding
sample_frame_count=int(len(time)/sample_frame_interval) #true n frames after rounding

# Create the matplotlib Animation object
def init():
    """
    This gets called each time the animation first starts.
    You must return any of the plot graphics that change
    Each frame of the animation.
    """
    ax.add_patch(circle)
    time_text.set_text('time = %.1f sec' % time[0])
    return circle,time_text

def animate(i):
    """
    This gets called each frame of the animation.
    This is where the animated graphics can be updated for the next frame.
    You must return any of the plot graphics that change
    Each frame of the animateion.
    """
    s=int(i*sample_frame_interval)
    circle.center = (gaze_x[s]/2., gaze_y[s]/2.)
    time_text.set_text('time = %.1f sec' % time[s])
    return circle,time_text

# Start the animation, but only play it for 1 frame
# (This gets around a current bug in the matplotlib animation code when
# you want to use blit=True during the real playback; which you do as it is
# 10x faster than when blit=False)
anim = animation.FuncAnimation(fig, animate,
                               init_func=init,
                               frames=1,
                               interval=actual_playback_rate,
                               blit=False)

# Start the animation for real this time. Based on the args provided,
# the animation will play from start to finish and then loop to the
```

```
# start and play over again. This repeats until you close the matplotlib window.
anim = animation.FuncAnimation(fig, animate,
                               init_func=init,
                               frames=sample_frame_count,
                               interval=actual_playback_rate,
                               blit=True)

plt.show()
```

## 2.6 2:40-3:30 Processing Recorded Eye Data

This section of the workshop will cover some common areas of eye data processing, including:

1. *Pixel to Visual Angle Conversion*
2. *Velocity and Acceleration Calculation*
3. *Filtering Out Noise*
4. *Parsing Eye Sample Data into Eye Events*

### 2.6.1 Pixel to Visual Angle Conversion

When using PsychoPy and the ioHub the experiment creator can specify that position information should be represented in one of several coordinate spaces, including pixels and visual degrees. When visual degrees are specified, stimuli are drawn using visual degree coordinates and sizes, and position data returned by devices capable of doing so (like the mouse or an Eye Tracker), will also report position in visual degrees.

In some cases you may wish to use pixels for the coordinate space in your experiment, but convert the eye position data to visual degrees after the data has been collected. Examples of this include wanting to use a different pixel to visual degree calculation or having incorrect or unknown eye to calibration plane distance during recording. A fanother use case for pixel2degree conversion is when using a remote eye tracker with the nead free and moving about. Ssome of the remote eye tracking systems available now report the current eye to screen distance for each sample collected, and this can be used to calculate visual degrees without using a fixed eye distance for a whole trial.

In situations like the above, using the example code provide here will allow the conversion of pixel data to angle data; either using a fixed eye to calibration plain distance, or a varying distance based on data in an array that is the same length as the pixel position data being processed. The relevent section of the example sript is:

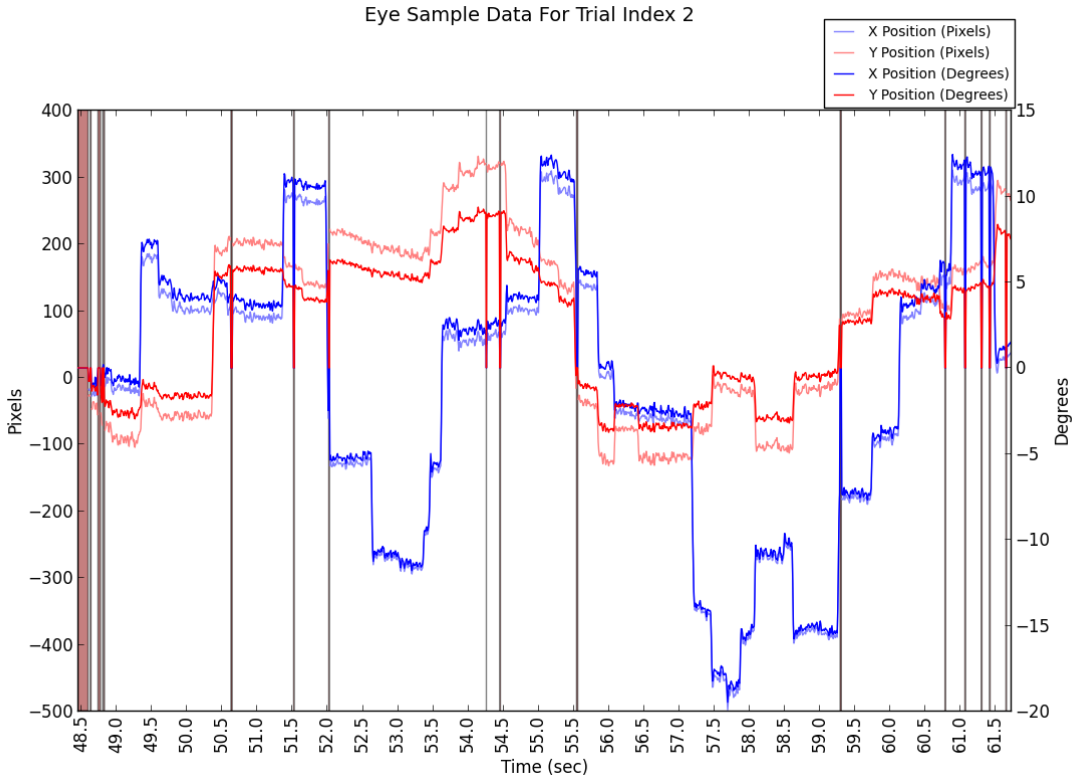
```
# Load the ioDataStore data file
# ....example code in many of the scripts provided ....
#

# Provide the necessary display geometry information.
# Note that in the future this default information could be read from
# the loaded data file so it does not need to be manually entered here.
#
calibration_area_info=dict(display_size_mm=(500,280.0),
                           display_res_pix=(1280.0,1024.0),
                           eye_distance_mm=550.0)

# Use the VisualAngleCalc class defined in the common_workshop_functions to
# generate an object that can convert data from pixel coordinates
# to visual angles based on the supplied calibration / display surface geometry
# and eye distance.
#
vac=VisualAngleCalc(**calibration_area_info)

# Calculate the visual degree position in x and y for the given pixel position arrays.
```

```
# If an array of head position data is provided to the pix2deg method,
# the degrees data will be calculated for each eye sample useing the head
# distance reported for that sample.
#
degree_x,degree_y=vac.pix2deg(pix_x,pix_y)

# Do as you may with the angle data.
# .....
```

A full python script which loads eye data from a iodataStore file, converts it to degrees, and plots the pixel and degree eye position sample traces can be found at python_source/data_processing/pixels2angle.py

#### Example Plot

Eye Position Traces in Pixel and Visual Degree Coordinates




### 2.6.2 Velocity and Acceleration Calculation

Velocity and Acceleration are often calculated using eye sample position data for use in eye event parsing algorithms (well, parsers based on velocity thresholds at least ;) ). Data is converted from pixel to visual degree coordinate space before being passed to the velocity and acceleration algorithms.

#### Velocity Calculation

The following function can be used to calculate the instantaneous velocity from eye position sample data:

```
def calculateVelocity(time,degrees_x,degrees_y=None):
    """
    Calculate the instantaneous velocity (degrees / second) for data points in
    degrees_x and (optionally) degrees_y, using the time numpy array for
    time delta information.
```

```
    Numpy arrays time, degrees_x, and degrees_y must all be 1D arrays of the same
    length.

    If both degrees_x and degrees_y are provided, then the euclidian distance
    between each set of points is calculated and used in the velocity calculation.

    time must be in seconds.msec units, while degrees_x and degrees_y are expected
    to be in visual degrees. If the position traces are in pixel coordinate space,
    use the VisualAngleCalc class to convert the data into degrees.
    """
    if degrees_y is None:
        data=degrees_x
    else:
        data=np.sqrt(degrees_x*degrees_x+degrees_y*degrees_y)

    velocity_between = (data[1:]-data[:-1])/(time[1:]-time[:-1])
    velocity = (velocity_between[1:]+velocity_between[:-1])/2.0
    return velocity
```

A full example python script which loaded eye data from an ioDataStore file, calculates the velocity for one trial of the data, and plots the result can be found in the workshop source materials: python_source/data_processing/velocity_accelleration.py

### Accelleration Calculation

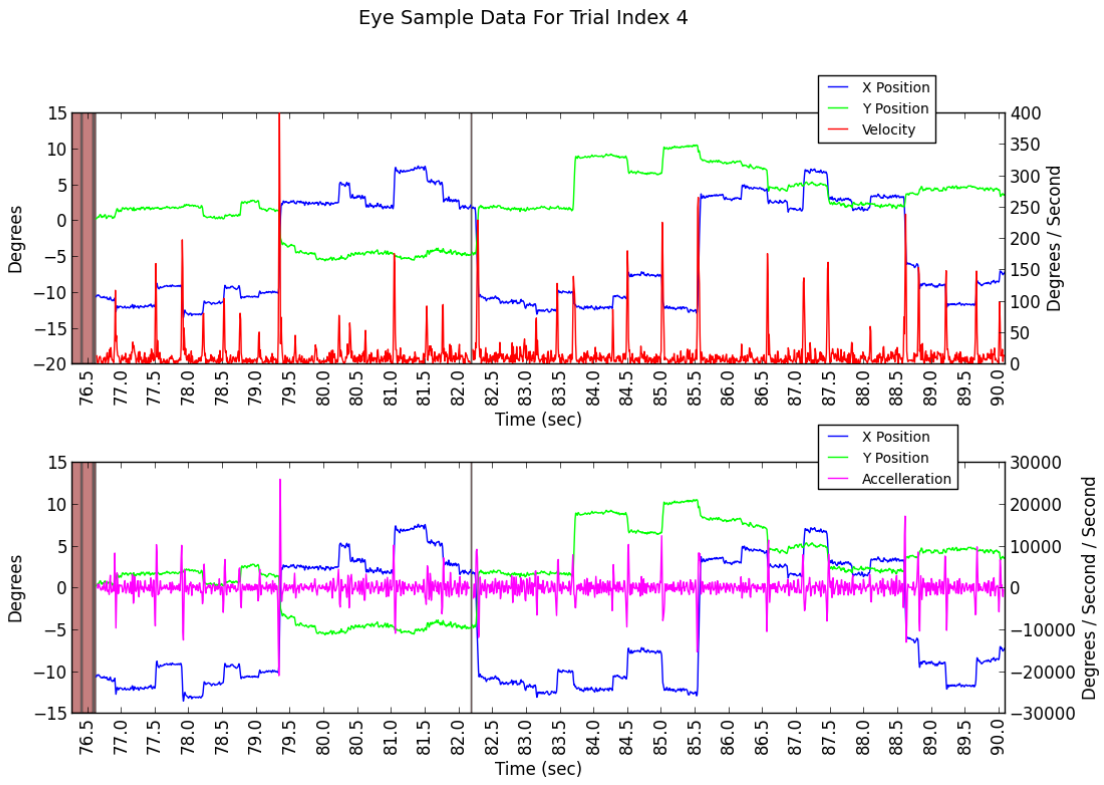
Accelleration, or the rate of velocity change over time, can be calculated at follows:

```
def calculateAccelleration(time,data_x,data_y=None):
    """
    Calculate the accelleration (degrees / second / second) for data points in
    degrees_x and (optionally) degrees_y, using the time numpy array for
    time delta information.
    """
    velocity=calculateVelocity(time,data_x,data_y)
    accel = calculateVelocity(time[1:-1],velocity)
    return accel
```

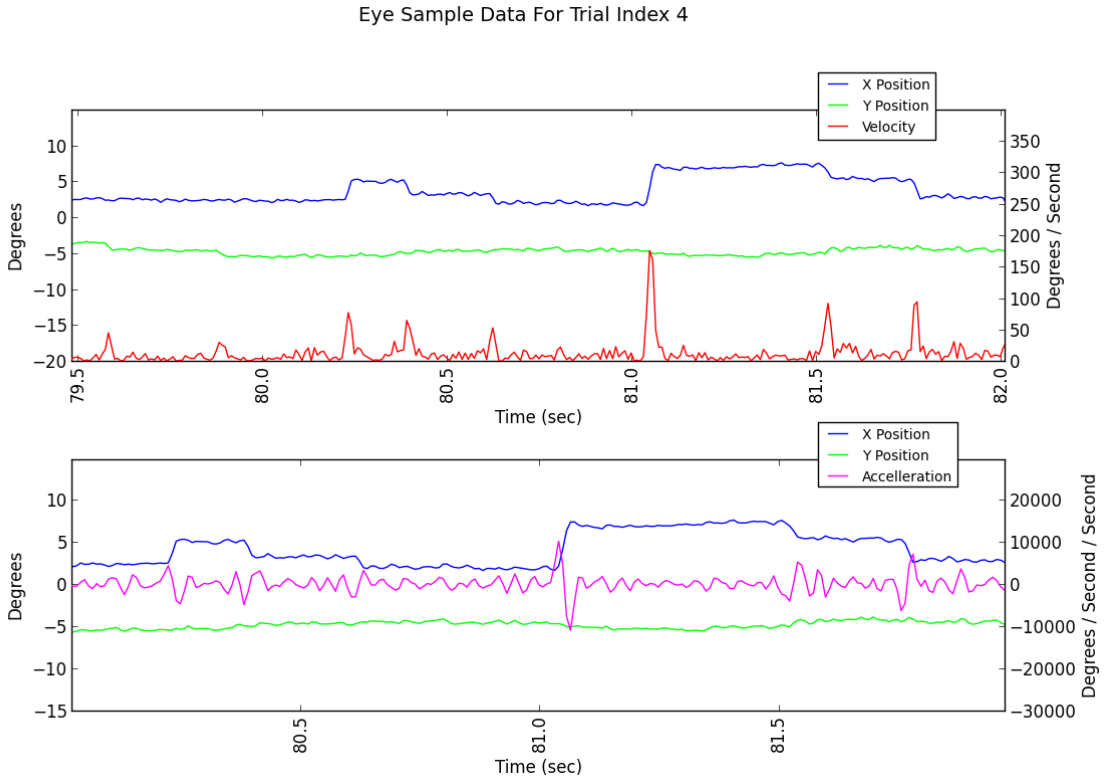
A full example python script which loaded eye data from an ioDataStore file, calculates the velocity for one trial of the data, and plots the result can be found in the workshop source materials: python_source/data_processing/velocity_accelleration.py

#### Example Plots

Eye Angle Traces with associated XY Velocity and Accelleration Trace




Magnified Eye Angle Traces with associated XY Velocity and Acceleration Trace




### 2.6.3 Filtering Out Noise

With any eye tracking system it is often beneficial to filter the sample data data recorded device to:

1. Reduce high-frequency noise from eye position data, possibly increasing precision measures.
2. Decrease velocity and acceleration noise in the eye signal, possibly improving eye event detection (Saccades, Fixations, etc).

When using a filtering algorithm on your eye sample data, it is important to consider:

A. What effect does the filter have on the reported characteristics of the occulomotor behaviour:

1. Are small saccades being removed?
2. Is the duration, amplitude, peak velocity, or other such properties of saccades being significantly effected?
3. Are over-shoots to target locations being exagerated?

2. What is the impact of the filter on the overall delay of on-line access to the eye data being reported?

3. What parameters are adjustable in the filtering algorithm, and what are the *best* settings to use?

There is often no one right answer to the above questions and considerations. The experimental paradigm being run and the way in which the resulting eye data collected will be analyzed can significantly influence what may be considered as *correct*.

With this in mind, it can be fruitful (and fun) to try different filtering algorithms and see how each changes the data being reported; both for better or worse.

### Some Example Filters

All the example filters demonstrated in the workshop can be used from the same python file. Simply uncomment the filter you wish to test and comment out the previously active filter.

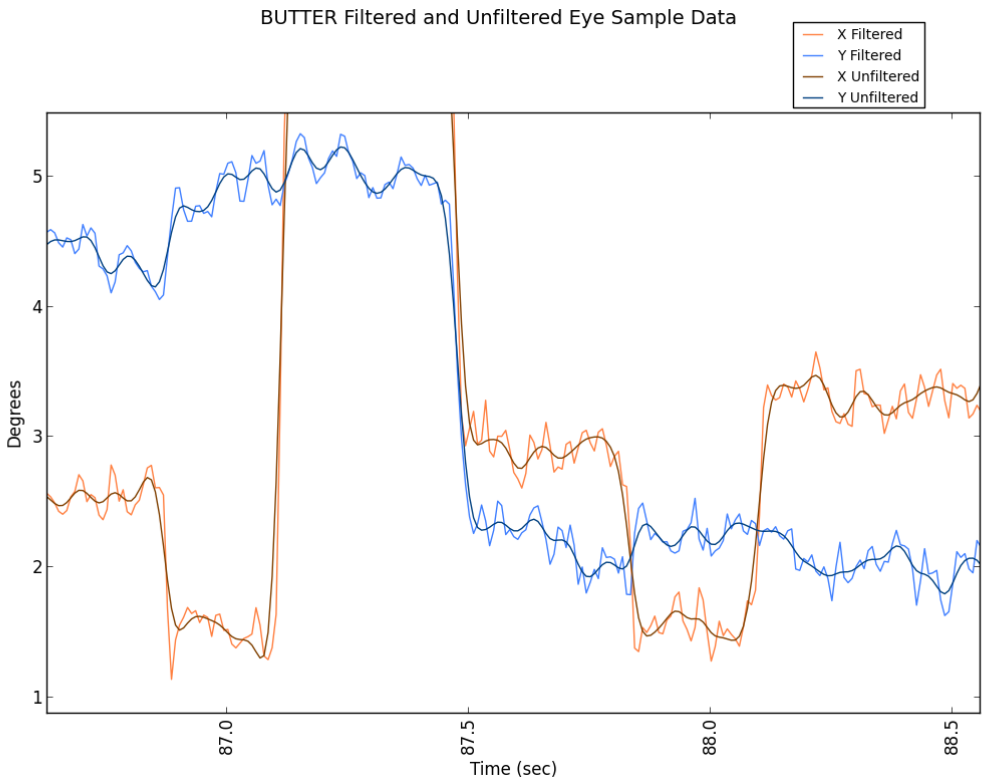
```
# -*- coding: utf-8 -*-

# This source file is available in python_source/data_processing/filters.py

from psychopy.iohub.datastore.util import ExperimentDataAccessUtility
from psychopy.iohub import EventConstants
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.transforms as mtransforms
from matplotlib.font_manager import FontProperties

from scipy.ndimage.filters import gaussian_filter1d
from scipy.signal import butter,filtfilt,medfilt

from common_workshop_functions import processSampleEventGaps,VisualAngleCalc,savitzky_golay

calibration_area_info=dict(display_size_mm=(500,280.0),
                           display_res_pix=(1280.0,1024.0),
                           eye_distance_mm=550.0)

def filterEyeSamples(filter_type,xpix,ypix,pupil,invalid_data_mask,**kwargs):
    processSampleEventGaps(xpix,ypix,pupil,invalid_data_mask,'linear')
    vac=VisualAngleCalc(**calibration_area_info)
    xdeg,ydeg=vac.pix2deg(xpix,ypix)

    if filter_type=='butter':
        wn=kwargs.get('wn',0.2)
        order=kwargs.get('order',2)
        b, a = butter(order, wn, 'low')
        x_filtered = filtfilt(b, a, xdeg)
        y_filtered = filtfilt(b, a, ydeg)

    elif filter_type=='gauss':
        sigma=kwargs.get('sigma',2)
```

```
        x_filtered = gaussian_filter1d(xdeg,sigma)
        y_filtered = gaussian_filter1d(ydeg,sigma)

    elif filter_type=='median':
        size=kwargs.get('size',5)
        x_filtered=medfilt(xdeg,size)
        y_filtered=medfilt(ydeg,size)

    elif filter_type=='sg':
        size=kwargs.get('size',7)
        order=kwargs.get('order',2)
        x_filtered=savitzky_golay(xdeg,window_size=size, order=order)
        y_filtered=savitzky_golay(ydeg,window_size=size, order=order)

    elif filter_type=='average':
        weights=np.asarray(kwargs.get('weights',[1.,2.,3.,2.,1.]))
        weights=weights/np.sum(weights)
        x_filtered=np.convolve(xdeg, weights,'same')
        y_filtered=np.convolve(ydeg, weights,'same')

    else:
        raise ValueError('Unknown Filter Type: %s. Must be one of %s'%(filter_type,str(['sg','butt

    xdeg[invalid_data_mask]=np.NaN
    ydeg[invalid_data_mask]=np.NaN
    x_filtered[invalid_data_mask]=np.NaN
    y_filtered[invalid_data_mask]=np.NaN

    return (xdeg,ydeg),(x_filtered,y_filtered)

# Enter data for use in this example
#
# We will do the pixel to degree calculation and plotting for one trial in
# the sample data file, select which to use (0 - 4):
#
TRIAL_INDEX=3
# Enter the eye tracker setup used for the data collection.
#
calibration_area_info=dict(display_size_mm=(500,280.0),
                           display_res_pix=(1280.0,1024.0),
                           eye_distance_mm=550.0)

dataAccessUtil=ExperimentDataAccessUtility('../hdf5_files','remote_data.hdf5',
                                           experimentCode=None,sessionCodes=[])
# Get the filtered event data.
#
event_type=EventConstants.BINOCULAR_EYE_SAMPLE
retrieve_attributes=('time','left_gaze_x','left_gaze_y','left_pupil_measure1',
            'right_gaze_x','right_gaze_y','right_pupil_measure1','status')
trial_event_data=dataAccessUtil.getEventAttributeValues(event_type,
            retrieve_attributes,
            conditionVariablesFilter=None,
            startConditions={'time':('>=','@TRIAL_START@')},
            endConditions={'time':('<=','@TRIAL_END@')},
            )

trial_data=trial_event_data[TRIAL_INDEX]

time=trial_data.time
status=trial_data.status
pix_x=trial_data.right_gaze_x
pix_y=trial_data.right_gaze_y
pupil=trial_data.right_pupil_measure1
```

```
invalid_data_mask=trial_data.status%10>=2

# No need to keep the hdf5 file open anymore...
#
dataAccessUtil.close()

# Get the range to use for the x axis
#
tmin=time.min()//1
tmax=time.max()//1+1

#filter_label='butter'
#unfiltered,filtered=filterEyeSamples(filter_label,pix_x,pix_y,pupil,invalid_data_mask,wn=0.2,ord

#filter_label='gauss'
#unfiltered,filtered=filterEyeSamples(filter_label,pix_x,pix_y,pupil,invalid_data_mask,sigma=1.5)

#filter_label='median'
#unfiltered,filtered=filterEyeSamples(filter_label,pix_x,pix_y,pupil,invalid_data_mask,size=5)

#filter_label='sg'
#unfiltered,filtered=filterEyeSamples(filter_label,pix_x,pix_y,pupil,invalid_data_mask,size=7,ord

filter_label='average'
unfiltered,filtered=filterEyeSamples(filter_label,pix_x,pix_y,pupil,invalid_data_mask,weights=[.5,
unfiltered_x,unfiltered_y=unfiltered
filtered_x,filtered_y=filtered

# Create a plot of filtered and unfiltered eye position for the full trial.
#
fig = plt.figure(figsize=(12,8))
fig.suptitle("%s Filtered and Unfiltered Eye Sample Data"%(filter_label.upper()),fontsize=14)
gax=plt.gca()
gax.plot(time, unfiltered_x,label='X Filtered',color=(1,.5,.25))
gax.plot(time, unfiltered_y,label='Y Filtered',color=(.25,.5,1))
gax.plot(time, filtered_x,label='X Unfiltered',color=(.5,.25,0))
gax.plot(time, filtered_y,label='Y Unfiltered',color=(0,.25,.5))
plt.xticks(np.arange(tmin,tmax,0.5),rotation='vertical')

trans = mtransforms.blended_transform_factory(gax.transData, gax.transAxes)
gax.fill_between(time, 0, 1, where=invalid_data_mask, facecolor='DarkRed',
                     alpha=0.5, transform=trans)
gax.set_ylabel('Degrees')
gax.set_xlabel('Time (sec)')

box = gax.get_position()
gax.set_position([box.x0, box.y0, box.width, box.height-.05])
fontP = FontProperties()
fontP.set_size('small')
plt.legend(loc='upper left', bbox_to_anchor=(.8, 1.15), borderaxespad=0,prop = fontP)

plt.show()
```

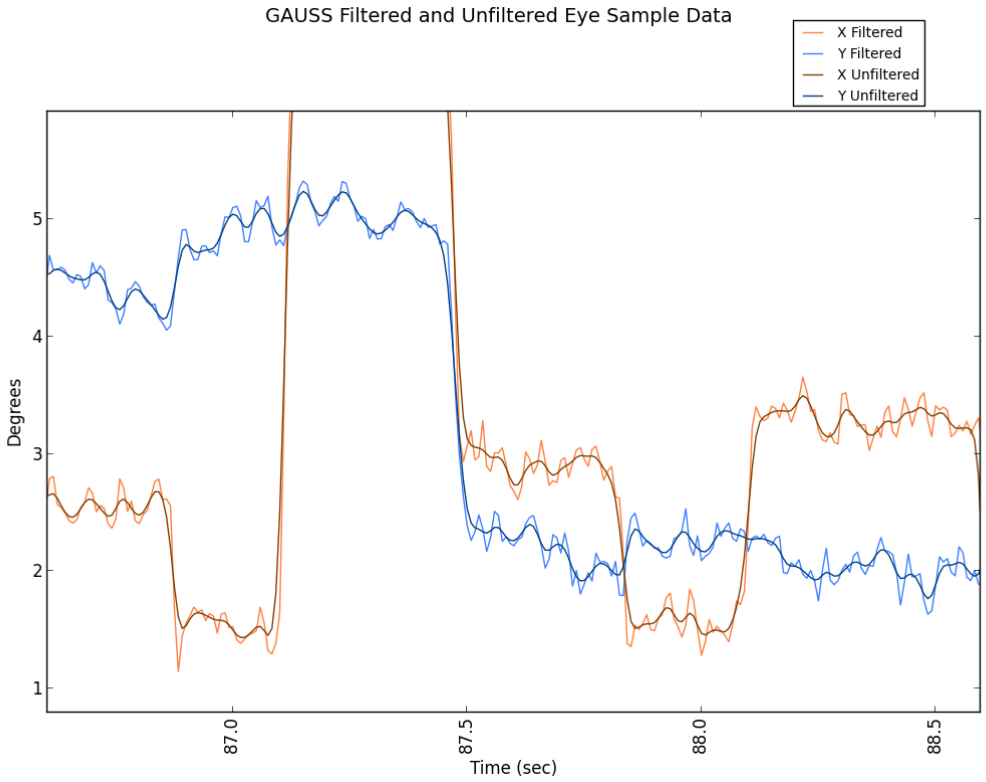
### Butterworth Filter

Unfiltered vs. Butterworth Filtered Eye Position Data



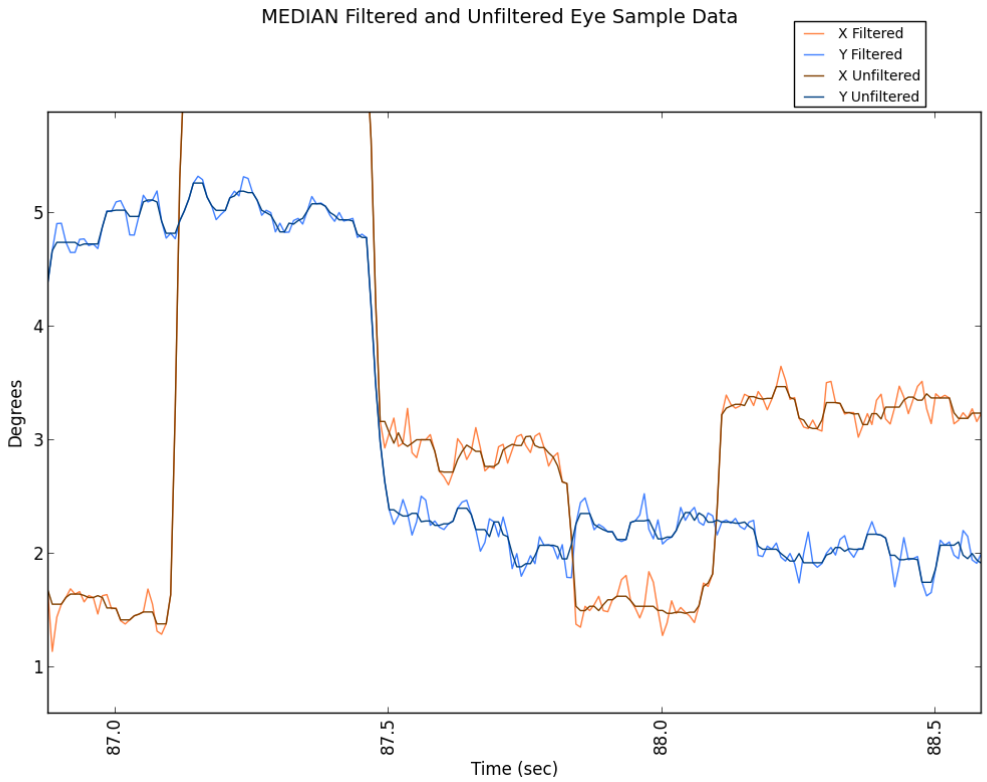

### Gaussian Filter

Unfiltered vs. Gaussian Filtered Eye Position Data



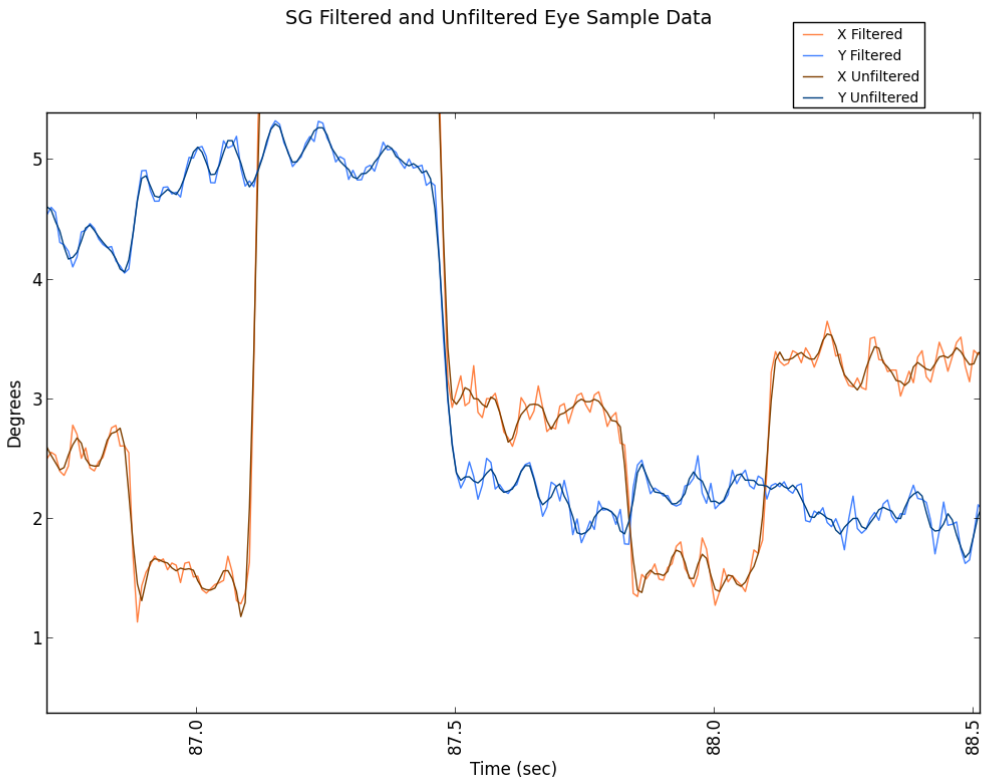
### Median Filter

Unfiltered vs. Median Filtered Eye Position Data



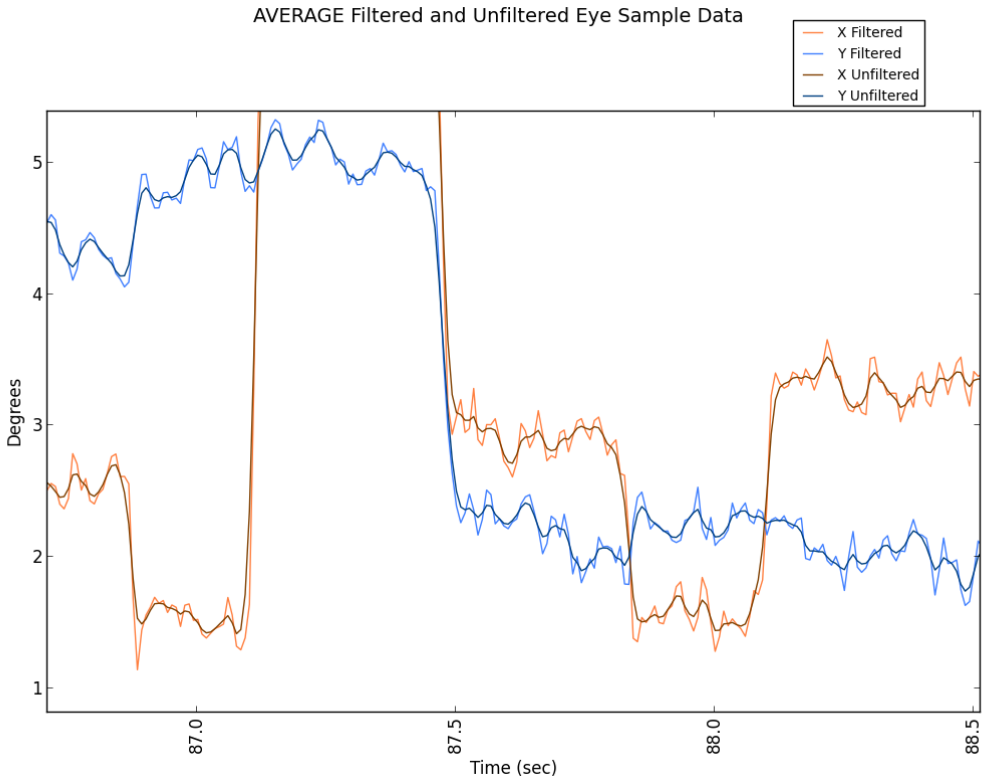

### Savitzky Golay Filter

Unfiltered vs. Savitzky Golay Filtered Eye Position Data




### Weighted Moving Average Filter

Unfiltered vs. Weighted Moving Average Filtered Eye Position Data



### 2.6.4 Parsing Eye Sample Data into Eye Events

Content to Be Moved from notebook.

## 2.7 Online Python Resources and Q&A

### 2.7.1 Going forwards

PsychoPy and ioHub are still improving a great deal all the time. You'll soon see;

- tighter integration of the two (e.g. less overlap in functionality)
- less need for code to perform the eye-tracking (e.g. a Builder Component for it)
- functions being built in for data processing to allow for greater offline (and potentially online) processing of the eye tracking data.
- within PsychoPy, the major areas of development are better 'sanity checking' of the experiment
- get on the user list and see how things develop; and email if you find things that need fixing

### 2.7.2 Online resources

Below are some links to web resources that are valuable sources of information on Python, useful Python Tools, Python in Neuroscience or Psychology, and Eye Tracking with Python. It is not complete so love google!

#### Python

NB **stick to Python 2.7.x for now.** Python 3 is a substantial rewrite and many packages don't yet support it.

- Python 2.7.5 Documentation
- Python's built-in modules

**Learning Python**

- The official Python tutorial
- http://www.Codecademy.com/
- Dive into Python

#### Python Tools

- Numpy is for fast numeric computations on arrays
- SciPy extends Numpy with mathematical routines (like optimization, filtering etc.)
- Matplotlib is the main plotting library for Python, using similar syntax to Matlab
- IPython provides a nice interactive shell for using Python and also a 'notebook' interface for storing notes and documentation along with your analysis code.

#### Python and Psychology or Neuroscience

PsychoPy :

- PsychoPy Docs
- ioHub Docs
- PsychoPy downloads (includes ioHub)

- PsychoPy User Group

Also:

- NiPy for neuroimaging
- The Frontiers journal special issue on Python in Neuroscience tools (many about modelling packages)

There's also *Open Sesame* (graphical interface for experiments with some eye-tracking support) and *PyEPL* (scripted, no eye-tracking) but neither is as flexible as PsychoPy!

### Python and Eye Tracking

Some eyetracker manufacturers provide general Python interfaces. Many an difficult to use and incomplete ctypes wrappers of their C library.

If your eye tracker isn't supported by ioHub (yet) and you would like to help make adding support for it happen, contact us! Why learn a different API for each eye tracker manufacturer you have systems from, when you could only need to be using one, psychopy.iohub!