# Contents

1. Getting Hummingbird ........................................... 3
2. Getting started .................................................. 5
3. Getting help ...................................................... 7
4. Supported facilities ............................................. 9
5. Contribute to Hummingbird ................................. 11
6. How to cite ....................................................... 13
Python Module Index ............................................. 41
Hummingbird is a python-based software tool for monitoring and analysing Flash X-ray Imaging (FXI) experiments in real time.
The easiest way to get Hummingbird is to clone it from the

\$ git clone https://github.com/FXIhub/hummingbird.git

More instructions can be found in the Installation guide.
CHAPTER 2

Getting started

Hummingbird is very simple to use. Configuration is done using a single python configuration file. For the beginning, checkout our collection of basic examples.
CHAPTER 3

Getting help

More advanced examples and a full API documentation are available here at .
Hummingbird is intended to be used across different user facilities. It has been extensively tested at the LCLS facility, see LCLS examples. Future releases of Hummingbird will include the European XFEL facility and more XFEL facilities.
CHAPTER 5

Contribute to Hummingbird

Hummingbird is meant to be an open project, developed by users of Flash X-ray Imaging (FXI) using modern X-ray sources. You are welcome to contribute.
6.1 Installation

6.1.1 Supported Operating Systems

• Linux
• MacOS

6.1.2 Requirements

• Python 2.7 or 3.4
• PyQt4, PyQt5 or PySide
• PyQtGraph >= 0.10
• Numpy
• Scipy
• PyZMQ
• pexpect
• mpi4py
• h5py
• h5writer
• pint
6.1.3 Requirements for testing

- subprocess32 (only for python 2.7)
- pytest

Some of the more advanced examples require an installation of libspimage and/or condor.

6.1.4 Downloading the Source Code

To be able to use Hummingbird you first need to obtain the code. To do that just clone the git repository:

```
$ git clone https://github.com/FXIhub/hummingbird.git
```

There is no need for any other installation steps.

6.2 Getting Started

6.2.1 Running Hummingbird

You can run `./hummingbird.py -h` to get some information about the available options of Hummingbird:

```
$ ./hummingbird.py -h

Hummingbird - the Online Analysis Framework.

optional arguments:
  -h, --help            show this help message and exit
  -i, --interface       start the control and display interface
  -b [conf.py], --backend [conf.py]
                         start the backend with given configuration file
  -r, --reload          reloads the backend
  -v, --verbose         increase output verbosity
  -d, --debug           output debug messages
  -p, --profile         generate and output profiling information
  --no-restore          no restoring of Qsettings
```

You can run Hummingbird in either interface (`-i`) or backend (`-b`) mode.

Backend

When running in backend mode the program will read and analyse data according to the provided configuration file. When no configuration file is given, the file `examples/basic/dummy.py`, the default configuration file, will be used.

Frontend

When running in interface mode the GUI program will be launched:
After adding a backend (upper left button) the program is waiting for connections from backends and can display any data that the backends send to it:

6.3 Configuration

The backend of Hummingbird uses a configuration file written in Python. This gives the user immense flexibility and power, with the responsibility that carries.

6.3.1 Learning by Example

Here is an example of a configuration file used to gather some basic statistics about a run stored in an XTC file:
import analysis.event
import analysis.beamline
import analysis.pixel_detector

state = {
    'Facility': 'LCLS',
    'LCLS/DataSource': '/data/rawdata/LCLS/cxi/cxic9714/xtc/e419-r0203-s01-c00.xtc'
}

def onEvent(evt):
    analysis.beamline.printPulseEnergy(evt['pulseEnergies'])
    analysis.beamline.printPhotonEnergy(evt['photonEnergies'])
    print "EPICS photon energy = \$g\ eV" %evt['parameters']['SIOC:SYS0:ML00:A051'].data
    analysis.pixel_detector.printStatistics(evt['photonPixelDetectors'])
    analysis.pixel_detector.printStatistics(evt['ionTOFs'])
    analysis.event.printID(evt['eventID'])
    analysis.event.printProcessingRate()

One can divide in three sections. In the first one the necessary analysis modules are imported:

import analysis.event
import analysis.beamline
import analysis.pixel_detector

In this case three modules are imported. You can find what modules are available by peeking into the src/analysis and src/plotting directory or by browsing the API documentation.

In the second section of the configuration file the global options for the program are set:

state = {
    'Facility': 'LCLS',
    'LCLS/DataSource': '/data/rawdata/LCLS/cxi/cxic9714/xtc/e419-r0203-s01-c00.xtc'
}

The global options must always called state. In this particular example first the Facility is set to LCLS, the only supported option at the moment. The following line defines where data is read from. It accepts any format that psana accepts (e.g. an XTC filename, an exp/run pair like exp=XCS/xcstut13:run=15, or a shared memory string like shmem=0_21_psana_AMO.0:stop=no).

In the third and final section of the configuration file the algorithms that are run on each of the events are defined:

def onEvent(evt):
    analysis.beamline.printPulseEnergy(evt['pulseEnergies'])
    analysis.beamline.printPhotonEnergy(evt['photonEnergies'])
    print "EPICS photon energy = \$g\ eV" %evt['parameters']['SIOC:SYS0:ML00:A051'].data
    analysis.pixel_detector.printStatistics(evt['photonPixelDetectors'])
    analysis.pixel_detector.printStatistics(evt['ionTOFs'])
    analysis.event.printID(evt['eventID'])
    analysis.event.printProcessingRate()

The list of algorithms to run must always be defined inside the onEvent function, which must take exactly one argument, named evt in this case.

This is a function that is called once for every event. The argument is a dictionary containing the measurements in each event. For example in this case evt['pulseEnergies'] is a dictionary with 4 entries, corresponding to the 4 gas monitor detector at LCLS. Each of the entries is a Record class containing at least name, data and often
Tip: To see a list of the available keys inside the `evt` simply run `evt.keys()`.

Using the `evt` variable one can pass data to multiple analysis algorithms. These algorithms will do the required analysis, communicate the results with the interface, and store any eventual output back onto the `evt` dictionary, using a new key. That way future analysis can use the output of previous ones.

For a list of available analysis algorithms please check the relevant API documentation.

This example is `examples/psana/xtc/conf.py`. You can find more example configurations inside the directories in `examples` and explained in More examples.

### 6.4 Basic examples

#### 6.4.1 dummy.py

This is the most basic and simple example which generates randomized 256x256 images as fake detector events showing up as `evt['photonPixelDetectors']['CCD']` for virtual events created at a repetition rate of 10 Hz:

```python
# Import analysis/plotting modules
import analysis.event
import plotting.image
import numpy as np

# Set new random seed
np.random.seed()

# Specify the facility
state = {}
state['Facility'] = 'Dummy'

# Create a dummy facility
state['Dummy'] = {
    # The event repetition rate of the dummy facility [Hz]
    'Repetition Rate': 10,
    # Dictionary of data sources
    'Data Sources': {
        # The name of the data source.
        'CCD': {
            # A function that will generate the data for every event
            'data': lambda: np.random.rand(256, 256),
            # The units to be used
            'unit': 'ADU',
            # The name of the category for this data source.
            # All data sources are aggregated by type, which is the key
            # used when asking for them in the analysis code.
            'type': 'photonPixelDetectors'
        }
    }
}

# This function is called for every single event
```
# following the given recipe of analysis

def onEvent(evt):
    
    # Processin rate [Hz]
    analysis.event.printProcessingRate()

    # Visualize detector image
    plotting.image.plotImage(evt['photonPixelDetectors']['CCD'], send_rate=10)

Notice that facility and data source is defined using the `state` variable. For every event, the current processing rate is printed and an image with the current virtual detector image is sent to the interface. Running this example in the backend (`hummingbird.py -b examples/basic/dummy.py`) we can start the frontend (`hummingbird.py -i`) in a separate shell (or even on a separate machine) and connect it to the backend by clicking on the left-most button:

Once we are connecting, the virtual CCD shows up as a data source. After opening an image window (4th button from the left), we can subscribe to the CCD (menu `Data sources`) and the image that we were sending from the backend is displayed at a refreshing rate of 1 second:
For the next example, we replace the random detector images with CCD images that simulate diffraction from an object produced at a given hit rate (here 50%):

```python
# Import analysis/plotting/simulation modules
import analysis.event
import plotting.image
import simulation.base

# Simulate diffraction data
sim = simulation.base.Simulation()
sim.hitrate = 0.5
sim.sigma = 1
```

(continues on next page)
# Specify the facility
state = {}
state['Facility'] = 'Dummy'

# Create a dummy facility
state['Dummy'] = {
    # The event repetition rate of the dummy facility [Hz]
    'Repetition Rate': 10,
    # Specify simulation
    'Simulation': sim,
    # Dictionary of data sources
    'Data Sources': {
        # Data from a virtual diffraction detector
        'CCD': {
            # Fetch diffraction data from the simulation
            'data': sim.get_pattern,
            'unit': 'ADU',
            'type': 'photonPixelDetectors'
        }
    }
}

# This function is called for every single event
# following the given recipe of analysis
def onEvent(evt):
    # Processing rate [Hz]
    analysis.event.printProcessingRate()
    # Visualize detector image
    plotting.image.plotImage(evt['photonPixelDetectors']['CCD'], vmin=-10, vmax=40)

Following the same procedure as for dummy.py we can follow the hits (left) and misses (right) show up in the interface:
6.4.3 detector.py

In order to add more analysis of the detector we print some statistics, count the number of photons and send a history of the photon counts and per-event detector histograms along with the CCD image:

```python
# Import analysis/plotting/simulation modules
import analysis.event
import analysis.pixel_detector
import plotting.line
import plotting.image
import simulation.base

# Simulate diffraction data
sim = simulation.base.Simulation()
sim.hitrate = 0.5
sim.sigma = 1

# Specify the facility
state = {}
state['Facility'] = 'Dummy'

# Create a dummy facility
state['Dummy'] = {
    # The event repetition rate of the dummy facility [Hz]
    'Repetition Rate': 10,
    # Specify simulation
    'Simulation': sim,
    # Dictionary of data sources
    'Data Sources': {
        # Data from a virtual diffraction detector
        'CCD': {
            # Fetch diffraction data from the simulation
            'data': sim.get_pattern,
            'unit': 'ADU',
            'type': 'photonPixelDetectors'
        }
    }
}
```

(continues on next page)
# Configuration for histogram plot

```python
histogramCCD = {
    'hmin': -10,
    'hmax': 100,
    'bins': 200,
    'label': "Nr of photons",
    'history': 200
}
```

# This function is called for every single event
# following the given recipe of analysis
```python
def onEvent(evt):

    # Processing rate [Hz]
    analysis.event.printProcessingRate()

    # Detector statistics
    analysis.pixel_detector.printStatistics(evt["photonPixelDetectors"])

    # Count Nr. of Photons
    analysis.pixel_detector.totalNrPhotons(evt, evt["photonPixelDetectors"]['CCD'],
                                        outkey="nrPhotons")
    plotting.line.plotHistory(evt["analysis"]['nrPhotons'], label='Nr of photons /frame', history=50)

    # Detector histogram
    plotting.line.plotHistogram(evt["photonPixelDetectors"]['CCD'], **histogramCCD)

    # Detector images
    plotting.image.plotImage(evt["photonPixelDetectors"]['CCD'])
```

In the interface, we can now open a new line plot (3rd button from the left) and display the history of the photon counts by subscribing to the `History(analysis/nrPhotons - CCD)` data source:

The depth of the history is defined by the length of the buffer, which can be resized in the main window. To the per-event histogram of the CCD we can subscribe both from an image plot (left panel) and from a line plot (right panel):
While the line plot shows the current histogram of the CCD, the image plot shows the history of the most recent detector histograms.

### 6.4.4 hitfinding.py

In the next example, we add hitfinding to our analysis pipeline. We use a simply lit pixel counter given thresholds for the definition of a photon (aduThreshold=10) and for the definition of a hit (hitscoreThreshold=100):

```python
# Import analysis/plotting/simulation modules
import analysis.event
import analysis.hitfinding
import plotting.image
import plotting.line
import simulation.base

# Simulate diffraction data
sim = simulation.base.Simulation()
sim.hitrate = 0.5
sim.sigma = 1

# Specify the facility
state = {}
state['Facility'] = 'Dummy'

# Create a dummy facility
state['Dummy'] = {
    # The event repetition rate of the dummy facility [Hz]
    'Repetition Rate': 100,
    # Specify simulation
    'Simulation': sim,
    # Dictionary of data sources
    'Data Sources':{
        # Data from a virtual diffraction detector
        'CCD':{
            # Fetch diffraction data from the simulation
            'data': sim.get_pattern,
            'unit': 'ADU',
            'type': 'photonPixelDetectors'
        }
    }

(continues on next page)
# This function is called for every single event following the given recipe of analysis

def onEvent(evt):
    
    # Processing rate [Hz]
    analysis.event.printProcessingRate()

    # Simple hit finding (counting the number of lit pixels)
    analysis.hitfinding.countLitPixels(evt, evt["photonPixelDetectors"]['CCD'],
                                         aduThreshold=10, hitscoreThreshold=100)

    # Extract boolean (hit or miss)
    hit = evt['analysis']['litpixel: isHit'].data

    # Compute the hitrate
    analysis.hitfinding.hitrate(evt, hit, history=5000)

    # Plot the hitscore
    plotting.line.plotHistory(evt['analysis']['litpixel: hitscore'], label='Nr. of lit pixels', hline=100, group="A")

    # Plot the hitrate
    plotting.line.plotHistory(evt['analysis']['hitrate'], label='Hit rate [%]', group="B")

    # Visualize detector image of hits
    if hit:
        plotting.image.plotImage(evt['photonPixelDetectors']['CCD'], vmin=-10, vmax=40, group="Detectors")

As compared to previous examples, we are plotting the CCD image only for hits. We are also sending history plots of hitscore and hitrate. The former can be very useful for finding the correct thresholds. When changing the threshold in the configuration file, there is no need to restart the backend. We can simply reload the configuration using the reload button (right-most button). Having all plots connected, the frontend looks like this:
In the last example, we show how it is possible to correlate and compare different parameters. Therefore, we first add more virtual data to our simulation: randomized pulse energies and (x,y) injector positions. Along with plotting the history of pulse energies and plotting the correlation of pulse energy vs. hitscore as a scatter plot, we populate a map of averaged hitrates as a function the (x,y) injector position tuple:

```python
# Import analysis/plotting/simulation modules
import analysis.event
import analysis.hitfinding
import analysis.beamline
import plotting.line
import plotting.image
import plotting.correlation
import simulation.base

# Simulate diffraction data
sim = simulation.base.Simulation()
sim.hitrates = 0.5
sim.sigma = 1

# Specify the facility
state = {}
state['Facility'] = 'Dummy'
```

(continues on next page)
# Create a dummy facility
state['Dummy'] = {
    # The event repetition rate of the dummy facility [Hz]
    'Repetition Rate': 10,
    # Specify simulation
    'Simulation': sim,
    # Dictionary of data sources
    'Data Sources': {
        # Data from a virtual diffraction detector
        'CCD': {
            # Fetch diffraction data from the simulation
            'data': sim.get_pattern,
            'unit': 'ADU',
            'type': 'photonPixelDetectors'
        },
        # Data from a virtual pulse energy detector
        'pulseEnergy': {
            # Fetch pulse energy values from the simulation
            'data': sim.get_pulse_energy,
            'unit': 'J',
            'type': 'pulseEnergies'
        },
        # Data from a virtual injector motor
        'injectorX': {
            # Fetch injector motor values (x) from the simulation
            'data': sim.get_injector_x,
            'unit': 'm',
            'type': 'parameters'
        },
        # Data from a virtual injector motor
        'injectorY': {
            # Fetch injector motor values (y) from the simulation
            'data': sim.get_injector_y,
            'unit': 'm',
            'type': 'parameters'
        }
    }
}

# Configuration for hitrate meanmap plot
hitmapParams = {
    'xmin': 0,
    'xmax': 1e-6,
    'ymin': 0,
    'ymax': 1e-6,
    'xbins': 10,
    'ybins': 10
}

# This function is called for every single event following the given recipe of analysis
def onEvent(evt):
    # Processing rate [Hz]
    analysis.event.printProcessingRate()
# Simple hit finding (counting the number of lit pixels)

```python
analysis.hitfinding.countLitPixels(evt, evt["photonPixelDetectors"]["CCD"],
    aduThreshold=10, hitscoreThreshold=100)
```

```
#analysis.beamline.averagePulseEnergy(evt, evt["pulseEnergies"])
```

# Extract boolean (hit or miss)

```python
hit = evt["analysis"]["litpixel: isHit"].data
```

# Compute the hitrate

```python
analysis.hitfinding.hitrate(evt, hit, history=1000)
```

# Plot history of pulse energy

```python
plotting.line.plotHistory(evt['pulseEnergies']['pulseEnergy'])
```

# Plot scatter of pulse energy vs. hitscore

```python
plotting.correlation.plotScatter(evt['pulseEnergies']['pulseEnergy'],
    evt["analysis"]['litpixel: hitscore'],
    xlabel='Pulse energy [J]', ylabel='Hitscore')
```

# Plot heat map of hitrate as function of injector position

```python
plotting.correlation.plotMeanMap(evt['parameters']['injectorX'], evt['parameters']['injectorY'],
    evt["analysis"]['hitrate'].data, name=
    'hitrateMeanMap', **hitmapParams)
```

In the interface, these plots look like this:
6.5 LCLS examples

To be able to run from LCLS data you need to have a functioning PSANA environment. There is lots of documentation about how to set up things on the page.

**Tip:** If you get strange errors running Hummingbird like syntax errors or ImportError: No module named psana make sure that you have run the setup step:

```
./reg/g/psdm/etc/ana_env.sh
```

6.5.1 Hit finding of mimivirus

This example is based on diffraction data from mimivirus (cite) published in the (entry 30). In order to run this example, it is necessary to download raw data files (XTC format) for a dark run (73) and a diffraction run (92) and put it inside a directory for XTC files.

6.5.2 mimi_dark.py

First, we need to get the average dark image running the backend with the following configuration:
import analysis.event
import plotting.image
import h5py

state = {
    'Facility': 'LCLS',
    'LCLS/DataSource': 'exp=amo15010:dir=/reg/d/psdm/AMO/amo15010/xtc:run=73'
}
dark = 0.
event_number = 0

def onEvent(evt):
    global dark, event_number
    dark += evt['photonPixelDetectors']['pnccdBackfullFrame'].data
    event_number += 1

def end_of_run():
    print "Saving average dark image to dark_run73.h5"
    with h5py.File('/tmp/amo15010_dark_run73.h5', 'w') as f:
        f['mean'] = dark / event_number

This will save the average dark in an HDF5 file named dark_run73.h5.

### 6.5.3 mimi_hits.py

With the average dark image ready, we can run with the following configuration:

import analysis.event
import analysis.hitfinding
import analysis.pixel_detector
import plotting.line
import plotting.image
import utils.reader

state = {
    'Facility': 'LCLS',
    'LCLS/DataSource': 'exp=amo15010:dir=/reg/d/psdm/AMO/amo15010/xtc:run=92',
    'indexing': True,
    'index_offset': 2250
}

dark = utils.reader.H5Reader('/tmp/amo15010_dark_run73.h5', 'mean').dataset

# Parameters
adu_photon = 12
threshold = 90000
hist_min = 5
hist_max = 35
hist_bins = 40

def onEvent(evt):
    # Processing rate
    analysis.event.printProcessingRate()

    try:
evt['photonPixelDetectors']['pnccdBackfullFrame']

except KeyError:
  return

# Dark calibration
analysis.pixel_detector.subtractImage(evt, 'photonPixelDetectors',
  'pnccdBackfullFrame',
  dark, outkey='pnccdBackSubtracted')

# Common mode correction
analysis.pixel_detector.commonModePNCCD(evt, 'analysis', 'pnccdBackSubtracted',
  outkey='pnccdBackCorrected')

# Plot back detector histogram (to figure out ADU/photon -> aduThreshold)
plotting.line.plotHistogram(evt['analysis']['pnccdBackCorrected'],
  hmin=hist_min, hmax=hist_max, bins=hist_bins,
  vline=adu_photon)

# Hitfinding
analysis.hitfinding.countLitPixels(evt, evt['analysis']['pnccdBackCorrected'],
  aduThreshold=adu_photon,
  hitscoreThreshold=threshold)

# Plot hitscore (to monitor hitfinder -> hitscoreThreshold)
plotting.line.plotHistory(evt['analysis']['litpixel: hitscore'], hline=threshold)

# Plot back detector image for hits only
if bool(evt['analysis']['litpixel: isHit'].data):
  plotting.image.plotImage(evt['analysis']['pnccdBackCorrected'],
    log=True, name='pnccdBack - only hits')

This performs detector correction (subtraction of average dark, common-mode), does hit finding based on a simple lit pixel counter and sends off detector images of hits as well as diagnostic plots for tuning the hit finder to the frontend.

Connecting to the backend interface and subscribing to the available source, Hummingbird shows hit images and other diagnostic information (hit-score, detector histogram, . . .):
6.5. LCLS examples
6.6 Reference API

6.6.1 analysis

analysis.beamline
analysis.event
analysis.hitfinding
analysis.sizing
analysis.pixel_detector
analysis.template

6.6.2 backend

backend.lcls

6.6.3 interface

Displays the results of the analysis to the user, using images and plots.

class interface.GUI(restore)
    Bases: interface.Qt.QMainWindow, object
    Main Window Class.
    Contains only menus and toolbars. The plots will be in their own windows.
    add_backend(data_source)
        Add backend to menu if it’s not there yet and append to _data_sources
    clear()
        Closes all DataWindows and remove all DataSources
    closeEvent(event)
        Save settings and exit nicely
    data_sources
        Provide access to the GUI data sources
    data_windows
        Provide access to the GUI data widows
    instance = None
    resizeEvent(event)
    restore_settings(do_restore, filename=None)
    save_data_windows()
        Save data windows state and data sources to the settings file
    save_settings(filename=None)
class interface.DataSource(parent, hostname, port, ssh_tunnel=None, conf={})
    Bases: interface.Qt.QObject
    Manages a connection with one backend

    add_item_to_group_structure(title, group)
    group_structure

    hostname
    Give access to the data source hostname

    name()
    Return a string representation of the data source

    plotdata
    Returns the data source dictionary of plotdata

    plotdata_added = None

    port
    Give access to the data source port

    query_configuration()
    Ask to the backend for the configuration

    query_reloading()
    Ask the backend to reload its configuration

    restore_state(state)
    Restores any plotdata that are saved in the state

    ssh_tunnel
    Give access to the data source ssh_tunnel

    subscribe(title, plot)
    Subscribe to the broadcast named title, and associate it with the given plot

    subscribe_for_recording(title)
    Subscribe to the broadcast named title, and associate it with recorder

    subscribed = None

    subscribed_titles
    Returns the currently subscribed titles

    unsubscribe(title, plot)
    Dissociate the given plot with the broadcast named title. If no one else is associated with it unsubscribe

    unsubscribe_for_recording(title)
    Dissociate the recorder with the broadcast named title. If no one else is associated with it unsubscribe

    unsubscribed = None

class interface.PlotData(parent, title, maxlen=1000, group=None)
    Bases: object
    Stores the data associated with a given broadcast

    append(y, x, l)
    Append the new data to the ringbuffers

    clear()
    Clear the buffers
group
   Returns the plot group
l
   Gives access to the l buffer
maxlen
   Gives access to maximum size of the buffers
nbytes
   Returns the number of bytes taken by the three buffers
resize(new maxlen)
   Change the capacity of the buffers
restore_state(state, parent)
   Restore a previous stored state
save_state(save_data=False)
   Return a serialized representation of the PlotData for saving to disk
sum_over(y, x, l)
title
   Returns the plot data title
x
   Gives access to the x buffer
y
   Gives access to the y buffer
class interface.H5Recorder(outpath, maxFileSizeMB=1)
   Recording event variables to an HDF5 file.

   Note: When reading from the recorder file, it might be necessary to sort them using the timestamp before comparing different datasets.

append(title, data, data_x)
   Append a tuple of time and event variable to dataset with the name of the variable.
closefile()
   Close existing file.
公开赛()
   Open new file using a unique filename.
class interface.RingBuffer(maxlen, data=None, index=0, length=0)
   Bases: object
   Provides a ring buffer for scalar and numpy data. It's always possible to retrieve the buffer data as a numpy array in O(1). This is achieved by always inserting two copies of any appended data, so it's a bit slower to add data, and it takes twice as much memory as a regular buffer.
append(x)
   Append a value to the end of the buffer
clear()
   Empty the buffer
max
   Returns the maximum value in the buffer, like a numpy array
min
Returns the minimum value in the buffer, like a numpy array

 nbytes
Returns the number of bytes taken by the buffer

 number_of_added_elements

 resize (new_maxlen)
Change the capacity of the buffers

 static restore_state (state)

 save_state ()
Return a serialized representation of the RingBuffer for saving to disk

 shape
Returns the shape of the buffer, like a numpy array

 class interface.ZmqContext (iothreads)
 Bases: object
Provides a singleton wrapper for a ZeroMQ context

 static instance (iothreads=4)
Returns the singleton instance of the ZeroMQ context

 self_ = None

 socket (socket_type)
Creates and returns a socket of the given type

 class interface.ZmqSocket (_type, parent=None, **kwargs)
 Bases: interface.Qt.QObject
Wrapper around a zmq socket. Provides Qt signal handling

 activity ()
Callback run when there’s activity on the socket

 bind (addr)
Bind socket to address

 connect_socket (addr, tunnel=None)
Connect socket to endpoint, possible using an ssh tunnel The tunnel argument specifies the hostname of the ssh server to tunnel through. Note that this still succeeds even if there’s no ZMQ server on the other end as the connection is asynchronous. For more details check the zmq_connect(3) documentation.

 identity ()
Return the zmq socket identity

 ready_read = None

 ready_write = None

 recv (flags=0)
Receive a message on the socket

 recv_array (flags=0, copy=True, track=False)
Receive a numpy array

 recv_json (flags=0)
Receive and json decode a message on the socket
recv_multipart()  
Receive a multipart message on the socket

send(_msg)  
Send a message on the socket

send_multipart(_msg)  
Send a list of messages as a multipart message on the socket

set_identity(name)  
Set zmq socket identity

subscribe(title)  
Subscribe to a broadcast with the given title

unsubscribe(title)  
Unsubscribe to a broadcast with the given title

interface.start_interface(restore)  
Initialize and show the Interface

ui  
Contains all the Designer created widgets, one per module

class interface.ui.DataWindow(parent=None)  
Bases: interface.Qt.QMainWindow  
Base class for all the data display windows (e.g. PlotWindow, ImageWindow)

blink_alert()  

closeEvent(event)  
Unsubscribe to any remaining broadcasts before closing

eventFilter(obj, event)  

get_state(settings)  
Returns settings that can be used to restore the widget to the current state

get_time(index=None)  
Returns the time of the given index, or the time of the last data point

on_menu_show()  
Show what data sources are available

on_save_to_png()  
Save a screenshot of the window

resizeEvent(event)  

restore_from_state(settings, data_sources)  
Restores the widget to the same state as when the settings were generated

set_sounds_and_volume()  

set_source_title(source, title, enable=True)  
Enable/disable a given broadcast

source_and_titles()  
Iterate through all available broadcasts

toggle_sounds()
toggle_volume()

class interface.ui.PlotWindow(parent=None)
    Bases: interface.ui.data_window.DataWindow, object
    Window to display 2D plots

    get_state(_settings=None)
        Returns settings that can be used to restore the widget to the current state

    get_time(index=None)
        Returns the time of the given index, or the time of the last data point

    keyPressEvent(event)
        Handle key presses

    on_view_legend_box()
        Show/hide legend box

    on_view_x_axis()
        Show/hide X axis

    on_view_y_axis()
        Show/hide Y axis

    replot()
        Replot data

    restore_from_state(settings, data_sources)
        Restores the widget to the same state as when the settings were generated

    updateFonts()

class interface.ui.ImageWindow(parent=None)
    Bases: interface.ui.data_window.DataWindow, object
    Window to display images

    get_state(_settings=None)
        Returns settings that can be used to restore the widget to the current state

    get_time(index=None)
        Returns the time of the given index, or the time of the last data point

    get_time_and_msg(index=None)
        Returns the time/msg of the given index, or the time/msg of the last data point

    init_running_hist(source, title)

    on_reset_cache()

    replot()
        Replot data

    restore_from_state(settings, data_sources)
        Restores the widget to the same state as when the settings were generated

    set_colormap_full_range()
        Ensures that the colormap covers the full range of values in the data

    set_colormap_range()
        Set the minimum and maximum values for the colormap

    toggle_axis(visible)
toggle_settings (visible)
    Show/hide settings widget

updateFonts ()

6.6.4 ipc

broadcast

mpi

zmqserver

6.6.5 plotting

plotting.line

plotting.image

plotting.correlation

6.6.6 utils

reader

class utils.reader.GeometryReader (filename, pixel_size=1.0)
class utils.reader.H5Reader (filename, key=None)

load_dataset (key)
class utils.reader.MaskReader (filename, key='data/data')

array

class utils.array.RunningHistogram (length=100, window=20, bins=100, hmin=0, hmax=100)

clear ()

next (new_value, length=None, window=None, bins=None, hmin=None, hmax=None)

utils.array.assembleImage (x, y, img=None, nx=None, ny=None, dtype=None, return_indices=False)

utils.array.cheetahToSlacH5 (cheetahArr)

utils.array.get2D (data)

utils.array.runningHistogram (new_data, name, length=100, window=20, bins=100, hmin=0, hmax=100)

utils.array.runningMean (x, N)
    http://stackoverflow.com/questions/13728392/moving-average-or-running-mean

utils.array.runningTrend (array, window, trend)
utils.array.slacH5ToCheetah(slacArr)

io

utils.io.load_condor()
   Loading the `condor` module if available
utils.io.load_spimage()
   Loading the `libspimage` module if available

## 6.7 How to contribute?

If you would like to contribute just go the issues page and create a new issue asking to be added to the team.

### 6.7.1 Getting Started

To be able to contribute first you need a copy of the repository. If you have not done so already submit your rsa public key to github under Manage/SSH keys and then clone the repository:

```
$ git clone https://github.com/FXIhub/hummingbird.git
```

For help on using git please check the official git documentation and the Github tutorials.

### 6.7.2 Contributing algorithms

You can find all the analysis algorithms inside the `src/analysis` directory and all plotting functions inside the `src/plotting` directory.

To add your own analysis algorithm to Hummingbird first check if there is already an algorithm that already does, or could be easily extended to do what you want. If there, then just edit the existing one and submit.

In many cases you will have to start from scratch though. Start by choosing to what module you should add your algorithm (e.g. `event.py`), or create a new module if you think it does not fit in any existing one.

Most algorithms take as argument a `Record` (e.g. `evt['pixel_detectors']['CCD']`) or a dictionary of `Records` (e.g. `evt['pulseEnergies']`), although many also take the entire event variable. You are free to choose, as long as you document it.

If you want to keep some result from your algorithm to be used by subsequent algorithms just return as a `Record` object and assign it to a new key of the event variable in the conf file (e.g. `examples/dummy/conf.py`)

There is a template for new analysis modules in `src/analysis/template.py`.

**Note:** Algorithms that want to keep results for further processing by other algorithms must be able to access the `evt` variable, so they must receive it as an argument.

### 6.7.3 Contributing to documentation

You can find documentation about the project at [spidocs.readthedocs.org](http://spidocs.readthedocs.org).
**Editing Documentation**

The documentation is written in [reStructuredText](https://www.sphinx-doc.org/en/master/), which is a simple to use and read markup language. The documentation is automatically built and published on the website after every commit to the [Hummingbird](https://github.com) repository.

There are two ways to edit documentation, online using the Github built-in editor, or offline using your favourite text editor.

### Online Editing

Simply click on the [Edit on Github](https://github.com) button at the top of the desired page in spidocs.readthedocs.org. This will take you to the Github page corresponding to the source of the page. Click on the `Edit` button, due the changes you want, and finally commit.

### Offline Editing

For editing the documentation on your computer you will need:

- A copy of the Hummingbird git repository
- Your favourite text editor
- `sphinx` installed: `pip install sphinx` or `sudo pip install sphinx`
- `sphinx_rtd_theme` installed: `pip install sphinx_rtd_theme` or `sudo pip install sphinx_rtd_theme`

Now you can simply edit existing `.rst` files, or add new ones, in the `docs` directory inside the root of the hummingbird git repository:

```
$ cd docs
$ emacs index.rst
```

After you finish editing you can look at the result by doing:

```
$ make html
```

This will create the html files inside `.build/html`, which you can open in your browser.

If you’re happy with the result you can now simply commit the changes and push. Your changes should be automatically pushed to [http://fxihub.github.io/hummingbird/docs](http://fxihub.github.io/hummingbird/docs) by Github.
Index

N
name() (interface.DataSource method), 33
nbytes (interface.PlotData attribute), 34
nbytes (interface.RingBuffer attribute), 35
next() (utils.array.RunningHistogram method), 38
number_of_added_elements (interface.RingBuffer attribute), 35

O
on_menu_show() (interface.ui.DataWindow method), 36
on_reset_cache() (interface.ui.ImageWindow method), 37
on_save_to_png() (interface.ui.DataWindow method), 36
on_view_legend_box() (interface.ui.PlotWindow method), 37
on_view_x_axis() (interface.ui.PlotWindow method), 37
on_view_y_axis() (interface.ui.PlotWindow method), 37
openfile() (interface.H5Recorder method), 34

P
PlotData (class in interface), 33
plotdata (interface.DataSource attribute), 33
PlotWindow (class in interface.ui), 37
port (interface.DataSource attribute), 33

Q
query_configuration() (interface.DataSource method), 33
query_reloading() (interface.DataSource method), 33

R
ready_read (interface.ZmqSocket attribute), 35
ready_write (interface.ZmqSocket attribute), 35
recv() (interface.ZmqSocket method), 35
recv_array() (interface.ZmqSocket method), 35
recv_json() (interface.ZmqSocket method), 35
recv_multipart() (interface.ZmqSocket method), 35
replot() (interface.ui.ImageWindow method), 37
replot() (interface.ui.PlotWindow method), 37
resize() (interface.PlotData method), 34
resize() (interface.RingBuffer method), 35
resizeEvent() (interface.GUI method), 32
resizeEvent() (interface.ui.DataWindow method), 36
restore_from_state() (interface.ui.DataWindow method), 36
restore_states() (interface.GUI method), 32
restore_state() (interface.DataSource method), 33
restore_state() (interface.PlotData method), 34
restore_state() (interface.RingBuffer static method), 35
RingBuffer (class in interface), 34
RunningHistogram (class in utils.array), 38
runningHistogram() (in module utils.array), 38
runningMean() (in module utils.array), 38
runningTrend() (in module utils.array), 38

S
save_data_windows() (interface.GUI method), 32
save_settings() (interface.GUI method), 32
save_state() (interface.PlotData method), 34
save_state() (interface.RingBuffer method), 35
self_ (interface.ZmqContext attribute), 35
send() (interface.ZmqSocket method), 36
send_multipart() (interface.ZmqSocket method), 36
set_colormap_full_range() (interface.ui.ImageWindow method), 37
set_colormap_range() (interface.ui.ImageWindow method), 37
set_identity() (interface.ZmqSocket method), 36
set_source_title() (interface.ui.DataWindow method), 36
set_sounds_and_volume() (interface.ui.DataWindow method), 36
subscribe() (interface.ZmqSocket method), 36
subscribe() (interface.ZmqSocket method), 36
subscribe_for_recording() (interface.DataSource method), 33
subscribed (interface.DataSource attribute), 33
subscribed_titles (interface.DataSource attribute), 33
sum_over() (interface.GradientData attribute), 33

T
title (interface.PlotData attribute), 34
toggle_axis() (interface.ui.ImageWindow method), 37
toggle_settings() (interface.ui.ImageWindow method), 37
toggle_sounds() (interface.ui.DataWindow method), 36
toggle_volume() (interface.ui.DataWindow method), 36

U
unsubscribe() (interface.DataSource method), 33
unsubscribe() (interface.ZmqSocket method), 36
unsubscribe_for_recording() (interface.DataSource method), 33
unsubscribed (interface.DataSource attribute), 33
updateFonts() (interface.ui.ImageWindow method), 38
updateFonts() (interface.ui.PlotWindow method), 37
utils (module), 38
utils.array (module), 38
utils.io (module), 39
utils.reader (module), 38

X
x (interface.PlotData attribute), 34

Y
y (interface.PlotData attribute), 34

Z
ZmqContext (class in interface), 35
ZmqSocket (class in interface), 35