Tutorial - python, pyana and matplotlib

A quick walk-through of the tools that exist for analysis of xtc files with python.
The main focus is on pyana, and the examples are from and for XPP primarily,
but may be useful examples to other experiments too.

Outline of contents:

The Basics

Python

http://docs.python.org/tutorial/

Pyana

Analysis Workbook. Python-based Analysis

Setting up a work directory (a.k.a. offline release directory)

Prior to this, you may need to set up your account for offline analysis:
Analysis Workbook. Account Setup

The general version of this is in Analysis Workbook. Quick Tour

Open a terminal at pslogin or psana, and type:

```
newrel ana-current xpptutorial
cd xpptutorial
ls -la
less .sit_release
sit_setup
```

Exploring an xtc file

pyxtcreader

```
 pyxtcreader -h
usage: pyxtcreader [options] xtc-files ...

options:
  -h, --help            show this help message and exit
  -v, --verbose
  -l L1_OFFSET, --l1-offset=L1_OFFSET
```

Loops through the xtc datagrams and dumps info to screen. I recommend piping it to 'less'.

Try:

```
pyxtcreader /reg/d/psdm/xpp/xppi0310/xtc/e81-r0098-s0* | less
```

Try the same with different verbosity levels:

```
pyxtcreader -v /reg/d/psdm/xpp/xppi0310/xtc/e81-r0098-s0* | less
pyxtcreader -vv /reg/d/psdm/xpp/xppi0310/xtc/e81-r0098-s0* | less
```

xtcscanner

```
xtcscanner -h
usage: xtcscanner [options] xtc-files ...

options:
  -h, --help            show this help message and exit
  -n NDATAGRAMS, --ndatagrams=NDATAGRAMS
  -v, --verbose
  -l L1_OFFSET, --l1-offset=L1_OFFSET
  -e, --epics
```

Similar to pyxtcreader in that it loops throug xtc datagrams, but doesn't print to screen. Internally counts the datatypes it finds, and at the end dumps a summary only. Optinally prints out epics information (default no).

Try:

```
xtcscanner /reg/d/psdm/xpp/xppi0310/xtc/e81-r0098-s0*
```

You should see output similar to this:

```
Scanning....
Start parsing files:
['/reg/d/psdm/xpp/xppi0310/xtc/e81-r0098-s00-c00.xtc', '/reg/d/psdm/xpp/xppi0310/xtc/e81-r0098-s01-c00.xtc']
  14826 datagrams read in 4.120000 s .   .   .   .   .   .   .
------------------------------------------------------------
XtcScanner information:
  - 61 calibration cycles.
  - Events per calib cycle:
   [240, 240, 240, 240, 240, 240, 240, 240, 240, 240, 240, 240, 240, 240, 240, 240, 240, 240, 240, 240,
240, 240, 240, 240, 240, 240, 240, 240, 240, 240, 240, 240, 240, 240, 240, 240, 240, 240, 240, 240, 240,
240, 240, 240, 240, 240, 240, 240, 240, 240, 240, 240, 240, 240, 240, 240, 240, 240, 240, 240, 240, 240]

Information from  1  control channels found:
fs2:ramp_angsft_target
Information from  11  devices found
                     BldInfo:EBeam:            EBeamBld_V1 (14641)
            BldInfo:FEEGasDetEnergy:           FEEGasDetEnergy (14563)   Any (78)
             BldInfo:NH2-SB1-IPM-01:           SharedIpimb (14641)
                BldInfo:PhaseCavity:           PhaseCavity (14641)
    DetInfo:EpicsArch-0|NoDevice-0:            Epics_V1 (107580)
        DetInfo:NoDetector-0|Evr-0:            EvrConfig_V4 (62)   EvrData_V3 (14640)
       DetInfo:XppSb2Ipm-1|Ipimb-0:            IpimbConfig_V1 (1)   IpmFexConfig_V1 (1)   IpimbData_V1
(14640)   IpmFex_V1 (14640)
       DetInfo:XppSb3Ipm-1|Ipimb-0:            IpimbConfig_V1 (1)   IpmFexConfig_V1 (1)   IpimbData_V1
(14640)   IpmFex_V1 (14640)
       DetInfo:XppSb3Pim-1|Ipimb-0:            IpimbConfig_V1 (1)   IpmFexConfig_V1 (1)   IpimbData_V1
(14640)   IpmFex_V1 (14640)
       DetInfo:XppSb4Pim-1|Ipimb-0:            IpimbConfig_V1 (1)   IpmFexConfig_V1 (1)   IpimbData_V1
(14640)   IpmFex_V1 (14640)
                         ProcInfo::            RunControlConfig_V1 (62)
XtcScanner is done!
------------------------------------------------------------
```

xtcexplorer

XTC Explorer - Old - GUI interface that builds pyana modules for you.

Try:

```
xtcexplorer /reg/d/psdm/xpp/xppi0310/xtc/e81-r0098-s0*
```

- Then hit the "Scan File(s)" button (can you find it?!?)
  - What do you see? Compare the GUI that pops up, with the output in the terminal window.
- Checkmark the IPM3 checkbox ('XppSb3Ipm=1|Ipimb-0')
  - hit the 'Write configuration to file' button,
  - hit the 'Run pyana' button
  - hit 'OK' and wait till a plot pops up... Close the window and wait again...
  - hit the 'Quit pyana' button
  - go to the 'General Settings' tab and switch Display mode to 'SlideShow'
  - go back to 'General Settings' again and change the number of events to accumulate to 240
  - hit the 'Write configuration to file'
  - hit the 'Edit configuration file' button. Edit the line with 'quantities = ': remove 'fex:channels' and add 'fex:ch1' and 'fex:ch0' instead
  - hit 'Run pyana' button again (as well as 'OK' when that pops up). Stare at the plot for a while...

Try something else:

```
kinit
addpkg XtcExplorer
scons
xtcexplorer /reg/d/psdm/xpp/xppi0310/xtc/e81-r0098-s0*
```

('`kinit`' will ask you for your password and give you a Kerberos ticket valid for 24 h that you need to access our afs software repository)

Now you have a local version of the XtcExplorer package in your directory. That allows you to edit the source code and customize the analysis modules in the `XtcExplorer/src` directory.

Exercise for later:
Edit `XtcExplorer/src/pyana_ipimb.py` to make a loglog plot of channel1 vs channel0.

The xtcexplorer has several shortcomings. It tries to be very generic, and thus is sometimes slower than it would have to be. It is also currently only capable of plotting from a single device in each plot, so many correlation plots will need to be added by hand. However, it is a simple tool to just look at the data contents, and provides many examples through its pyana modules.

For some more useful analysis examples, in the following we'll stick to writing customized pyana modules and running pyana from the command line. But before getting to the pyana modules, I'll briefly touch on a few items general to python that may be useful: saving files, matplotlib for plotting, and IPython for interactive work.

NumPy, SciPy and MatPlotLib

These are packages that you may want to look into. Pretty much all our examples here are using them:

- http://numpy.scipy.org/ - Numerical package for python (arrays etc)
- http://www.scipy.org/ - Scientific tools
- http://matplotlib.sourceforge.net/ - plotting package

Other useful links:

- http://www.scipy.org/NumPy_for_Matlab_Users
- http://www.scipy.org/
- http://www.sagemath.org/
- http://code.google.com/p/spyderlib/

Saving data arrays

Here are a few examples of how you can save data arrays in python.

saving numpy arrays to numpy file

```
import numpy as np

myarray = np.arange(0,100)
np.save( "output_file_name.npy", myarray)
np.savetxt( "output_file_name.txt", myarray)
```

```
import scipy.io

N_array = np.arange(0,100)
x_array = np.random(100)
y_array = np.random(100)
scipy.io.savemat( "output_file_name.mat", mdict={'N': N_array, 'x' : x_array, 'y' : y_array } )
```

```
import h5py

ofile = h5py.File("output_file_name.h5",'w')
group = ofile.create_group("MyGroup")
group.create_dataset('delaytime',data=np.array(self.h_delaytime))
group.create_dataset('rawsignal',data=np.array(self.h_ipm_rsig))
group.create_dataset('normsignal',data=np.array(self.h_ipm_nsig))
ofile.close()
```

For more examples, see How to access HDF5 data from Python and http://code.google.com/p/h5py/

Plotting with MatPlotLib

One of the most commonly used tools for plotting in python: matplotlib. Other alternatives exist too.

Matplotlib:

- http://matplotlib.sourceforge.net/users/pyplot_tutorial.html
- The plotting can be done directly in the pyana module, but be aware that you need to disable plotting for the module to run successfully in a batch job.

```
import matplotlib.pyplot as plt

plt.plot(array)
plt.show()
```

- Or you can load arrays from a file and interactively plot them in iPython. The same ('recommended') syntax as above can be used, or if you use 'import *' you don't need to prepend the commands with the package name, which is handy when plotting interactively:

```
from matplotlib.pyplot import *

ion()
plot(array)
draw()
```

Interactive analysis with IPython

The LCLS offline analysis group does have plans for a real interactive pyana, but currently this is not available.
2011-11-04 iPsana Interactive Analysis Framework.pdf

The version available in our offline release system is
IPython 0.9.1 – An enhanced Interactive Python.
so this is the one I've been using in these examples.
Not a whole lot more than a python shell.

However, the latest IPython has loads of new and interesting features...

http://ipython.org/

Loading your arrays into (I)Python and plotting interactively:

This example reads in a file produced by the "point detector delay scan" example below.

```
[ofte@psana0106 xpptutorial]$ ipython --pylab
Python 2.4.3 (#1, Nov  3 2010, 12:52:40)
Type "copyright", "credits" or "license" for more information.

IPython 0.9.1 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object'. ?object also works, ?? prints more.

In [1]: ipm3 = load('point_scan_delay.npy')

In [2]: ipm3.shape
Out[2]: (200, 3)

In [3]: ion()

In [4]: delay = ipm3[:,0]

In [5]: ipmraw = ipm3[:,1]

In [6]: ipmnorm = ipm3[:,2]

In [7]: plot(delay,ipmnorm,'ro')
Out[7]: [<matplotlib.lines.Line2D object at 0x59c4c10>]

In [9]: draw()

In [10]:
```

Sometimes you need to issue the draw() command twice, for some reason. After drawing you can keep working on the arrays and plot more...

Extracting the data with pyana, some examples

Outline of a pyana module

Like the other frameworks, pyana is an executable that loops through the XTC file and calls all requested user modules at certain transitions. All the analysts need to do is to fill in the relevant functions in their user analysis module:

>>

code (pyana outline)

```
# useful imports
import numpy as np
import matplotlib.pyplot as plt
from pypdsdata.xtc import TypeId

class mymodule (object) :
    """Class whose instance will be used as a user analysis module. """

    def __init__ ( self,
                   source = ""
                   threshold = "" ):
        """Class constructor.
        The parameters to the constructor are passed from pyana configuration file.
        If parameters do not have default values  here then the must be defined in
        pyana.cfg. All parameters are passed as strings, convert to correct type before use.
```

```
        @param source         name of device, format 'Det-ID|Dev-ID'
        @param threshold      threshold value (remember to convert from string)
        """
        self.source = source
        self.threshold = float(threshold)

    def beginjob( self, evt, env ) :
        """This method is called once at the beginning of the job. It should
        do a one-time initialization possible extracting values from event
        data (which is a Configure object) or environment.

        @param evt    event data object
        @param env    environment object
        """
        pass

    def beginrun( self, evt, env ) :
        """This optional method is called if present at the beginning of the new run.

        @param evt    event data object
        @param env    environment object
        """
        pass

    def begincalibcycle( self, evt, env ) :
        """This optional method is called if present at the beginning
        of the new calibration cycle.

        @param evt    event data object
        @param env    environment object
        """
        pass

    def event( self, evt, env ) :
        """This method is called for every L1Accept transition.

        @param evt    event data object
        @param env    environment object
        """
        pass

    def endcalibcycle( self, env ) :
        """This optional method is called if present at the end of the
        calibration cycle.

        @param env    environment object
        """
        pass

    def endrun( self, env ) :
        """This optional method is called if present at the end of the run.

        @param env    environment object
        """
        pass

    def endjob( self, env ) :
        """This method is called at the end of the job. It should do
        final cleanup, e.g. close all open files.

        @param env    environment object
        """
        pass
```

For the following two examples, check out the latest version of the `pyana_examples` package:

```
addpkg pyana_examples HEAD
scons
```

(Note, if you don't already have a Kerberos ticket, you need to issue a '`kinit`' command before '`addpkg`'. You will be prompted for your unix password.)

Datatypes, and how to find data from your detector/device in the xtc file.

Pyana and psana has follows this naming scheme for labeling the datatypes from various devices. You can find the names by investigating the xtc file with the above-mentioned tools (pyxtcreader, xtcscanner, xtcexplorer).
To see some examples of how to fetch the various data types in pyana (or psana), look at Devices and Datatypes.

Point detector delay scan

The python code for this pyana module resides in `pyana_examples/src/xppt_delayscan.py`. In this example, we do a point detector delay scan, where we get the time as scan points via a control PV, and where time rebinning based on phase cavity measurement is used to improve the time resolution. One IPIMB device (a.k.a. IPM3) is used for normalization (i0, I Zero) (parameter name ipimb_norm) and another IPIMB device (a.k.a. PIM3) channel 1 is used as the signal (parameter name ipimb_sig).

Open an editor and save the following in a file named pyana.cfg:

```
[pyana]

modules = pyana_examples.xppt_delayscan

[pyana_examples.xppt_delayscan]
controlpv = fs2:ramp_angsft_target
ipimb_norm = XppSb3Ipm-1|Ipimb-0
ipimb_sig = XppSb3Pim-1|Ipimb-0
threshold = 0.1
outputfile = point_scan_delay.npy
```

If you look at the code (pyana_examples/src/xppt_delayscan.py) you'll notice there are no detector names in there. The names of the detectors in the XTC file are passed as parameters from the configuration file above. The `ipimb_norm` parameter represents the IPIMB diode used for normalization, and the configuration files set its value to "XppSb3Ipm-1|Ipimb-0" (a.k.a. IPM3). Similarly, the IPIMB diode used for signal is represented by the `pipmb_sig` and its value set to "XppSb3Pim-1|Ipimb-0" (a.k.a. PIM3). By changing these parameter values, the `pyana_examples/src/xppt_delayscan.py` module can easily be used for other experiments or instruments.

Run pyana (start with 200 events):

```
pyana -n 200 /reg/d/psdm/XPP/xppi0310/xtc/e81-r0098-s0*
```

>>

Highlighting of some code snippets from `xppt_delayscan.py`:

- **Fetching the ControlPV information:**
  ControlPV is available from the `env` object, and since it only changes at the beginning of each calibration cycle, the `begincalibcycle` function is the appropriate place to get it:

  ```
  def begincalibcycle( self, evt, env ) :
  ```

  The ControlConfig object may contain several pvControl and pvMonitor objects. In this case there's only one, but make sure the name matches anyway:

```
        ctrl_config = env.getConfig(TypeId.Type.Id_ControlConfig)

        for ic in range (0, ctrl_config.npvControls() ):
            cpv = ctrl_config.pvControl(ic)
            if cpv.name()=="fs2:ramp_angsft_target":

                # store the value in a class variable (visible in every class method)
                self.current_pv_value = cpv.value() )
```

- **Fetching the IPIMB and PhaseCavity information:**
  All the other information that we need, is available through the `evt` object, and `event` member function is the place to get it:

```
    def event( self, evt, env ) :
```

Use "XppSb3Ipm-1|Ipimb-0" (a.k.a. IPM3) sum of all channels for normalization and filtering

```
        ipmN_raw = evt.get(TypeId.Type.Id_IpimbData, "XppSb3Ipm-1|Ipimb-0")
        ipmN_fex = evt.get(TypeId.Type.Id_IpmFex, "XppSb3Ipm-1|Ipimb-0")

        ipmN_norm = ipmN_fex.sum
```

Use "XppSb3Pim-1|Ipimb-0" (a.k.a. PIM3) channel 1 as signal

```
        ipmS_raw = evt.get(TypeId.Type.Id_IpimbData, "XppSb3Pim-1|Ipimb-0" )
        ipmS_fex = evt.get(TypeId.Type.Id_IpmFex, "XppSb3Pim-1|Ipimb-0" )

        ipm_sig = ipmS_fex.channel[1]
```

Get the phase cavity:

```
        pc = evt.getPhaseCavity()
        phasecav1 = pc.fFitTime1
        phasecav2 = pc.fFitTime2
        charge1 = pc.fCharge1
        charge2 = pc.fCharge2
```

Compute delay time and fill histograms

```
        delaytime = self.current_pv_value + phasecav1*1e3

        # The "histograms" are nothing but python lists. Append to them, and turn them into arrays at
the end.
        self.h_ipm_rsig.append( ipm_sig )
        self.h_ipm_nsig.append( ipm_sig/ipm_norm )
        self.h_delaytime.append( delaytime )
```

Image peak finding

Here are a collection of useful algorithms for image analysis: http://docs.scipy.org/doc/scipy/reference/ndimage.html

The python code for this pyana module example resides in `pyana_examples/src/xppt_image_analysis.py`.
This particular example is done with a CSPad image, but only a single section is available. For more typical CSPad module, see next section.

Edit `pyana.cfg` to include configuration for xppt_image_analysis, and comment out the delay_scan module:

```
[pyana]

modules = pyana_examples.xppt_image_analysis
#modules = pyana_examples.xppt_delayscan

[pyana_examples.xppt_image_analysis]
source = XppGon-0|Cspad-0
region = 127.3, 188.4, 95.1, 126.9

[pyana_examples.xppt_delayscan]
controlpv = fs2:ramp_angsft_target
ipimb_norm = XppSb3Ipm-1|Ipimb-0
ipimb_sig = XppSb3Pim-1|Ipimb-0
threshold = 0.1
outputfile = point_scan_delay.npy
```

Then run the xppt_image_analysis pyana module on xppi0112 run 55:

```
pyana -n 10 /reg/d/psdm/XPP/xppi0112/xtc/e162-r0055-s00-c00.xtc
```

*Edit `pyana.cfg` again and comment out the region parameter (add a semicolon ";" to the beginning of the line).
Run again a single event and try to select a region by mouse clicks instead:*

```
pyana -n 1 /reg/d/psdm/XPP/xppi0112/xtc/e162-r0055-s00-c00.xtc
```

- Hit the "Zoom to rectangle" button in the matplotlib toolbar.
- Zoom in on a rectangle around the bright spot in the "Region of interest" plot to the right.
- You should now see the region marked out in the left window.
- Hit the "Zoom" button once more, to go back to normal mode.
- Click on the red rectangle in the left plot to print the region parameters and new Center of mass to screen.

Here are some code snippet highlights from the xppt_image_analysis.py module:

- For each event, fetch the CsPad information, and get the image array:

```
   def event( self, evt, env ) :

        elements = evt.getCsPadQuads(self.source, env)
        image = elements[0].data().reshape(185, 388)
```

- Select a region of interest. If none is given (optional module parameter), set RoI to be the whole image.

```
        # Region of Interest (RoI)
        if self.roi is None:
            self.roi = [ 0, image.shape[1], 0, image.shape[0] ] # [x1,x2,y1,y2]

        print "ROI   [x1, x2, y1, y2] = ", self.roi
```

- Using only the RoI subset of the image, compute center-of-mass using one of the SciPy.ndimamge algorithms. Add to it the position of the RoI to get the CMS in global pixel coordinates:

```
        roi_array = image[self.roi[2]:self.roi[3],self.roi[0]:self.roi[1]]
        cms = scipy.ndimage.measurements.center_of_mass(roi_array)
        print "Center-of-mass of the ROI: (x, y) = (%.2f, %.2f)" %(self.roi[0]+cms[1],self.roi[2]+cms[0])
```

- Here's an example how you can make an interactive plot and select the Region of Interest with the mouse. Here we plot the image in two axes (subpads on the canvas). The first will always show the full image. In the second axes, you can select a rectangular region in "Zoom" mode (click on the Toolbar's Zoom button). The selected region will be drawn on top of the full image to the left, while the right plot will zoom into the selected region:

```
       fig = plt.figure(1,figsize=(16,5))
        axes1 = fig.add_subplot(121)
        axes2 = fig.add_subplot(122)

        axim1 = axes1.imshow(image)
        axes1.set_title("Full image")

        axim2 = axes2.imshow(roi_array, extent=(self.roi[0],self.roi[1],self.roi[3],self.roi[2]))
        axes2.set_title("Region of Interest")

        # rectangular ROI selector
        rect = UpdatingRect([0, 0], 0, 0, facecolor='None', edgecolor='red', picker=10)
        rect.set_bounds(*axes2.viewLim.bounds)
        axes1.add_patch(rect)

        # Connect for changing the view limits
        axes2.callbacks.connect('xlim_changed', rect)
        axes2.callbacks.connect('ylim_changed', rect)
```

- To compute the center-of-mass of the selected region, revert back to non-zoom mode (hit the 'zoom' button again) and click on the rectangle. The rectangle is connected to the 'onpick' function which updates `self.roi` and computes the center-of-mass:

```
        def onpick(event):
            xrange = axes2.get_xbound()
            yrange = axes2.get_ybound()
            self.roi = [ xrange[0], xrange[1], yrange[0], yrange[1]]

            roi_array = image[self.roi[2]:self.roi[3],self.roi[0]:self.roi[1]]
            cms = scipy.ndimage.measurements.center_of_mass(roi_array)

            print "Center-of-mass of the ROI: (x, y) = (%.2f, %.2f)" % (self.roi[0]+cms[1],self.roi[2]
+cms[0])

        fig.canvas.mpl_connect('pick_event', onpick)

        plt.draw()
```

CSPad images and tile arangements

The python code for this pyana module example resides in `XtcExplorer/src/pyana_image.py`.

Try some plotting of CSPad data using xtcexplorer. Launch the explorer and load xpp48712 run 66 (a dark run):

```
xtcexplorer /reg/d/psdm/XPP/xpp48712/xtc/e153-r0066-s00-c00.xtc
```

- *Look through a couple of events, then "Quit Pyana" and edit the configuration file. Add an output file name, and switch to "NoDisplay" and run 100 events to collect an average of dark images.
- With darks collected, load another file from the same experiment: run 141. Edit the pyana configuration file to use the file you just generated to subtract darks. Run the explorer in "SlideShow" mode again.
- Change the color scale of the plot by left and right clicking on the colorbar.

CSPad data structure

CSPad data in xtc is a list of elements. In pyana get the list from the evt (event) object (notice the need for the env (environment) object too!):

```
elements = evt.getCsPadQuads(self.source, env)
```

`elements` here is a python list of ElementV1 or ElementV2 (or later versions) objects, each representing one quadrant. The list is not ordered, so to know which quadrant you have, you have to check with `element.quad()`. To store a local array of the whole CSPad detector, you can do the following.

- In beginjob, find out from the configuration object what part of the CSPad was in use (sometimes sections are missing):

```
def beginjob ( self, evt, env ) :

       config = env.getConfig(xtc.TypeId.Type.Id_CspadConfig, self.source)
        if not config:
            print '*** cspad config object is missing ***'
            return

        quads = range(4)

        # memorize this list of sections for later
        self.sections = map(config.sections, quads)
```

- In each event, get the current CSPad data:

```
def event(self, evt, env):
    elements = evt.getCsPadQuads(self.source,env)

    pixel_array = np.zeros((4,8,185,388), dtype="uint16")

    for element in elements:
        data = element.data() # the 3-dimensional data array (list of 2d sections)
        quad = element.quad() # current quadrant number (integer value)

        # if any sections are missing, insert zeros
        if len( data ) < 8 :
            zsec = np.zeros( (185,388), dtype=data.dtype)
            for i in range (8) :
                if i not in self.sections[quad] :
                    data = np.insert( data, i, zsec, axis=0 )

        pixel_array[quad] = data
```

What we have so far gives you a 4d numpy array of all pixels. And if you want to store it in e.g. a numpy array, you can reshape it down to 2 dimensions (this is the format of the official pedestal files made by the translator):

```
pixels = pixel_array.reshape(1480,388)
np.save("pixel_pedestal_file.npy", pixels )
```

CSPad tile arrangement

To get a rough picture of the full detector, here's an example of how XtcExplorer/src/cspad.py does it:

- For each Quadrant (cspad_layout.txt):

```
   def get_quad_image( self, data3d, qn) :
       """get_quad_image
       Get an image for this quad (qn)

       @param data3d           3d data array (row vs. col vs. section)
       @param qn               quad number
       """
       pairs = []
       for i in range (8) :

           # 1) insert gap between asics in the 2x1
           asics = np.hsplit( data3d[i], 2)
           gap = np.zeros( (185,3), dtype=data3d.dtype )
           #
           # gap should be 3 pixels wide
           pair = np.hstack( (asics[0], gap, asics[1]) )

           # all sections are originally 185 (rows) x 388 (columns)
           # Re-orient each section in the quad

           if i==0 or i==1 :
               pair = pair[:,::-1].T   # reverse columns, switch columns to rows.
           if i==4 or i==5 :
               pair = pair[::-1,:].T   # reverse rows, switch rows to columns
           pairs.append( pair )

           if self.small_angle_tilt :
               pair = scipy.ndimage.interpolation.rotate(pair,self.tilt_array[qn][i])

       # make the array for this quadrant
       quadrant = np.zeros( (850, 850), dtype=data3d.dtype )

       # insert the 2x1 sections according to
       for sec in range (8):
           nrows, ncols = pairs[sec].shape

           # colp,rowp are where the top-left corner of a section should be placed
           rowp = 850 - self.sec_offset[0] - (self.section_centers[0][qn][sec] + nrows/2)
           colp = 850 - self.sec_offset[1] - (self.section_centers[1][qn][sec] + ncols/2)

           quadrant[rowp:rowp+nrows, colp:colp+ncols] = pairs[sec][0:nrows,0:ncols]

       return quadrant
```

Then combine all four quadrant images into the full detector image:

```
       self.image = np.zeros((2*850+100, 2*850+100 ), dtype="float64")
       for quad in xrange (4):

           quad_image = self.get_quad_image( self.pixels[quad], quad )
           self.qimages[quad] = quad_image

           if quad>0:
               # reorient the quad_image as needed
               quad_image = np.rot90( quad_image, 4-quad)

           qoff_x = self.quad_offset[0,quad]
           qoff_y = self.quad_offset[1,quad]
           self.image[qoff_x:qoff_x+850, qoff_y:qoff_y+850]=quad_image

       return self.image
```

Fine tuning

Notice that the code snippets above make use of some predefined quantities which it reads in from "calibration files". The files contains calibrated numerical values for individual sections' and quads' rotations and shifts. All of these files are located in the experiment's 'calib' folder, but is not generated automatically. The XtcExplorer currently has a local version which is **not correct** but which is close enough to display a reasonable image. For how the files have been read in, you can take a look at `XtcExplorer/src/cspad.py`'s `read_alignment` function.

For how to find the correct constants for each experiment, look at the CSPAD Alignment page.

Non-interactive batch analysis

Pyana jobs are designed to do batch analysis, but matplotlib plotting does not go well with this. If you want your job to produce graphics, make sure to use a matplotlib backend that writes the graphics directly to file, e.g. png files.

Multiprocessing

Pyana can make use of multiple core processing. On the command line, add the option '-p N' where N is the number of cores to use.

Extra care needs to be taken when plotting. Also, output files need to be made with the pyana mkfile command. The output will be merged at the end of the job, but may not be in order. So if you need events to be written to a file in chronological order, you're better off using single core processing.