Making a centre of mass grain map using the fable python module: ImageD11/makemap.py

Jon Wright, For ImageD11 recent versions (roughly 1.5.0)

This is a step-by-step guide on how to run the different programs in ImageD11 and fable which allow you to make a centre of mass grain map. The overall purpose of the code is to analyse “spotty” diffraction data from a 2D detector so that you can index the spots and assign them as belonging to specific grains inside the sample. At the end of the process you get an orientation and a centre of mass position of the grain inside the sample and for each spot you get the corresponding h,k,l indices. You also get unit cell parameters, which can give you strain tensors if the grains are sufficiently strained (eg, more than a few 100 microstrain).

The document is quite long. It could be more concise if the software was more automatic and easier to use. Fortunately the code is all open source, perhaps you could help to improve it?

Before you start

If you are reading this then hopefully you already have some spotty diffraction data that you want to process, or perhaps you are about to collect some. For the method to work, you need to record a series of 2D diffraction images while the sample is rotated. It works much better if the sample stays in the X-ray beam during the rotation. You will need some information from your experimental logbook in order to get started:

- What was the wavelength you used?
- Roughly what was the distance from the sample to the detector?
- Was the rotation right handed or left handed and which direction did the rotation axis point?
- What was the pixel size of your detector?
- Are the images free from spatial distortion or do you need to use a calibration (grid image or spline file) ?
- What was the orientation of the detector, or which way is up?

Software installation

If you are at ESRF then log into the rnice6 cluster and source the bash script from:

```
$ . /sware/exp/fable/bin/fable.bash
```

Elsewhere you need to install a working python system (version 2.6 or 2.7, not version 3) together with a scientific python “stack”, which means the useful modules like numpy, scipy, pyopengl, PIL etc. On linux these are usually provided by the system package manager. On windows there is a pre-configured package called winpython (http://winpython.sourceforge.net/) with most of this included. There are also semi-commercial offerings from enthought or continuum analytics, which have apparently worked well on mac computers. The ImageD11 code is generally developed either on debian6 (ESRF installation) or on windows using the python interpreter from python.org and the windows packages from Christoph Gohlke (http://www.lfd.uci.edu/~gohlke/pythonlibs/).

To install the fable python modules you can download the code from fable.sourceforge.net, where there are some installation instructions. The best is to get the most up-to-date modules from the source repository and build and install them. On unix machines this means doing an svn checkout to get the code:

```
        svn co https://svn.code.sf.net/p/fable/code/ImageD11/trunk ImageD11
```

Then build and install it:

```
        cd ImageD11
        python setup.py build install
```

You might need the root password for a system wide install. Apart from ImageD11 you should do the same with the other modules needed. These are:

xfab, fabio, fabian

While you are there, it is also useful to get PolyXSim and FitAllB.

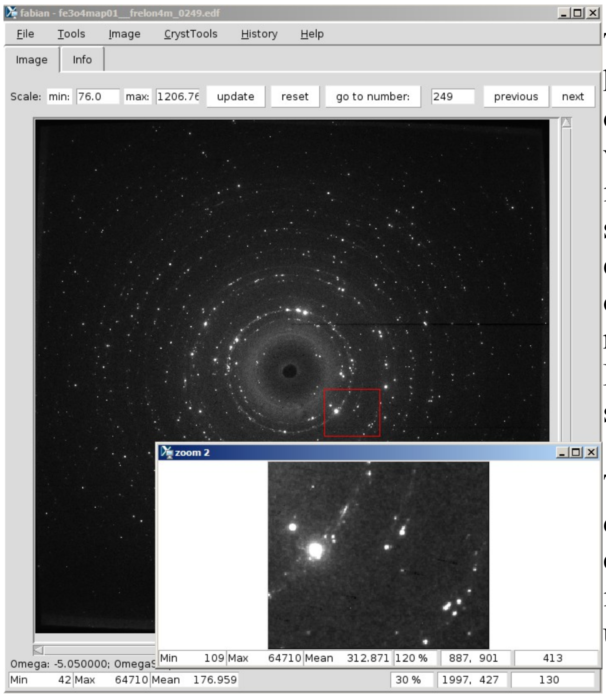
It is also worthwhile to install the java/eclipse based fable graphical user interface. This contains lots of documentation and gives access to the various other packages. In this specific document we will avoid using graphical interfaces as far as possible, the intention is to learn how to use the python codes so you can run scripts which make batch processing of many scans.

If you get stuck with installation please try pasting the error message you get into google and see if it can help you. If that fails then send an email to the fable-talk list (fable-talk@lists.sourceforge.net) and someone might be able to help you.

Your spotty images

For the purpose of this document we will run through the procedure using real data which was collected at beamline ID11 some time ago. These are located in the folder:

```
/data/id11/inhouse/jon/mar10/fe3o4map01_/
```



The sample was mounted in a kapton capillary which holds some grains of magnetite. The figure shows one of the images opened using the fabian image viewer (fabian.py). As you can see, the image does indeed contain relatively well isolated diffraction spots. This is not the most beautiful data which has ever been collected, the there is some saturation, spot overlap and a significant background coming from a near field detector which was used at the same time. Fortunately we have the tools to try to overcome such problems!

The first thing to note from the images is which detector has been used (Frelon4M in this case, you can see from the filename). In the headers of the images the fields "Omega" and "OmegaStep" allow us to figure out how the experiment was done. In this case the scan starts at -30 degrees and has steps of 0.1 degree up to 120 degrees. Ideally a larger scan range should be used, but we cannot go back in time. There are two wedges 90 degrees apart.

Determining the background

Expert users will have their own preferred method to remove the background, but in this case we will just use a simple script which comes with ImageD11 called bgmaker.py. It takes as input the part of the filename which does not change, the first and last image numbers and an output filename. As with most of these python scripts you can figure out how it works by running it with the –help argument:

```
rnice6-0103:/data/id11/inhouse/jon/mar10/fe3o4map01_ % bgmaker.py --help
Usage: bgmaker.py [options]

Options:
  -h, --help            show this help message and exit
  -n STEM, --namestem=STEM
                        Name of the files up the digits part, eg mydata in
                        mydata0000.edf
  -f FIRST, --first=FIRST
                        Number of first file to process, default=0
  -l LAST, --last=LAST  Number of last file to process
  -o OUTFILE, --outfile=OUTFILE
                        Output filename, default=bkg.edf
  -F FORMAT, --Format=FORMAT
                        File format [edf|bruker]
  -s STEP, --step=STEP  step - every nth image
  --ndigits=NDIGITS     Number of digits in file numbering [4]
  -k KALMAN_ERROR, --kalman-error=KALMAN_ERROR
                        Error value to use Kalman style filter (read noise)
```

In this case we will first make a new directory for our analysis and then run the script. We give the step as 120 images and it will run through taking the minimum of the series using every 120'th image. We repeat this 5 times with different start numbers and then make a median of the resulting images.

Since the images are one directory up and named “fe3o4map01__frelon4m0000.edf”, “fe3o4map01__frelon4m0001.edf”, …., “fe3o4map01__frelon4m1200.edf” the arguments for the script are:
-n ../fe3o4map01__frelon4m : unchanging part of the filename, one directory up.
-f 0 : number of the first image
-l 1200 : number of the last image
--ndigits is left as the default to get 4 (eg: 0000)
-F is left as the default (edf). Using “bruker” would give the name as stem.0000. Otherwise the format is appended to the name (eg: mccd or .edf.gz etc). Any file format which fabio can read can be used, there are quite a few.

```
cd guide
rnice6-0103:/data/id11/inhouse/jon/mar10/fe3o4map01_/guide % bgmaker.py -n
../fe3o4map01__frelon4m_ -f 0 -l 1200 -s 120  -o bkg0000.edf
../fe3o4map01__frelon4m_0000.edf
Using minimum image algorithm
../fe3o4map01__frelon4m_0120.edf
../fe3o4map01__frelon4m_0240.edf
```

```
../fe3o4map01__frelon4m_0360.edf
../fe3o4map01__frelon4m_0480.edf
../fe3o4map01__frelon4m_0600.edf
../fe3o4map01__frelon4m_0720.edf
../fe3o4map01__frelon4m_0840.edf
../fe3o4map01__frelon4m_0960.edf
../fe3o4map01__frelon4m_1080.edf
../fe3o4map01__frelon4m_1200.edf
writing bkg0000.edf in edf format
Total time = 1.333152 /s


…
bgmaker.py -n ../fe3o4map01__frelon4m_ -f 20 -l 1200 -s 120  -o bkg0001.edf
bgmaker.py -n ../fe3o4map01__frelon4m_ -f 40 -l 1200 -s 120  -o bkg0002.edf
bgmaker.py -n ../fe3o4map01__frelon4m_ -f 60 -l 1200 -s 120  -o bkg0003.edf
bgmaker.py -n ../fe3o4map01__frelon4m_ -f 80 -l 1200 -s 120  -o bkg0004.edf
…
```

We now make the median of those 5 background images using the median.py script which comes with fabian:
rnice6-0103:fe3o4map01_/guide % median.py -i bkg0000.edf -f 0 -l 5 -d -o bkg

You can view the background image itself using fabian and also the raw data with the background subtracted. In the image menu select correction → subtract background and then select your background image.

Using the tools → line profile option in fabian.py we can then see that the noise level in the images is about 200 ADU. It will be difficult to isolate peaks in the peaksearch which have a lower intensity than this value (could be done with smoothing or a smarter background algorithm). The method used here is fairly quick, if you spend more time you might get better results.

Peaksearching the images

The peaksearch.py script in ImageD11 finds connected regions of pixels which are above user supplied threshold values. In this case something like 200 ADU is about as low as you want to go, but with such a low value the strong peaks will merge together and become overlapped. To overcome that problem you use several threshold values. By setting the color range min and max values in fabian.py you can see what the peaksearch.py algorithm will see as connected objects. Or otherwise use the line profile tool to see what the pixel values are.

As you can see, peaksearch has lots more options than bgmaker.py, but some are the same.

```
rnice6-0103:/data/id11/inhouse/jon/mar10/fe3o4map01_/guide % peaksearch.py
--help
Usage: peaksearch.py [options]

Options:
  -h, --help            show this help message and exit
  -n STEM, --namestem=STEM
                        Name of the files up the digits part  eg mydata in
                        mydata0000.edf
  -F FORMAT, --format=FORMAT
                        Image File format, eg edf or bruker
  -f FIRST, --first=FIRST
                        Number of first file to process, default=0
  -l LAST, --last=LAST  Number of last file to process
  -o OUTFILE, --outfile=OUTFILE
                        Output filename, default=peaks.spt
  -d DARK, --darkfile=DARK
                        Dark current filename, to be subtracted, default=None
  -D DARKOFFSET, --darkfileoffset=DARKOFFSET
                        Constant to subtract from dark to avoid overflows,
                        default=0
  -s SPLINE, --splinefile=SPLINE
                        Spline file for spatial distortion,
                        default=/data/opid11/inhouse/Frelon2K/spatial2k.spline
  -p PERFECT, --perfect_images=PERFECT
                        Ignore spline Y|N, default=N
  -O FLOOD, --flood=FLOOD
                        Flood file, default=None
  -t THRESHOLDS, --threshold=THRESHOLDS
                        Threshold level, you can have several
  --OmegaFromHeader     Read Omega values from headers [default]
  --OmegaOverRide       Override Omega values from headers
  --singleThread        Do single threaded processing
  --profile=PROFILE_FILE
                        Write profiling information (you will want
                        singleThread too)
  -S OMEGASTEP, --step=OMEGASTEP
                        Step size in Omega when you have no header info
  -T OMEGA, --start=OMEGA
                        Start position in Omega when you have no header info
  -k KILLFILE, --killfile=KILLFILE
                        Name of file to create stop the peaksearcher running
  --ndigits=NDIGITS     Number of digits in file numbering [4]
  -P PADDING, --padding=PADDING
                        Is the image number to padded Y|N, e.g. should 1 be
                        0001 or just 1 in image name, default=Y
  -m, --median1D        Computes the 1D median, writes it to file .bkm and
                        subtracts it from image. For liquid background on
                        radially transformed images
  --monitorcol=MONITORCOL
                        Header value for incident beam intensity
  --monitorval=MONITORVAL
                        Incident beam intensity value to normalise to
  --omega_motor=OMEGAMOTOR
                        Header value to use for rotation motor position
                        [Omega]
  --omega_motor_step=OMEGAMOTORSTEP
                        Header value to use for rotation width [OmegaStep]
  --interlaced          Interlaced DCT scan
```

The ones we need here are:
-n stem : as before for bgmaker this is ../fe3o4map01__frelon4m_
-f 0 : first image
-l 1200 : last image
-o demo.spt : this is the output file specifier. It will create several files based on this name.
-d bkg.edf : We use our background image as the dark current, as it will be subtracted
-D is not used, it was a historical feature related to unsigned integers and avoiding negative numbers.
-s /data/id11/inhouse/Frelon4M/frelon4m.spline : This is the fit2d spline file for this detector
-p is not used. If your images are already corrected for distortion put -p Y.

-O /data/id11/inhouse/Frelon4M/F4M_Sm_July08.edf : This is the flood correction (pixel efficiency)
-t 200 -t 2000 -t 20000 : We use three threshold values for weak, medium and strong peaks. You can use more.

We don't use the rest of the arguments. The main ones you are likely to need are -S and -T if you don't have the angles in the image headers. They should be supplied in degrees. (If you did a DCT experiment and have an interlaced scan with names like stem_0_0000.edf, stem_1_0000.edf, stem_0_0001.edf, …, etc then you need "`--interlaced --OmegaOverRide -S 0.1 -T -90`").

The full command is:

```
 peaksearch.py -n ../fe3o4map01__frelon4m_ -f 0 -l 1200 -o demo.spt -d bkg.edf \
      -s /data/id11/inhouse/Frelon4M/frelon4m.spline \
       -O /data/id11/inhouse/Frelon4M/F4M_Sm_July08.edf \
      -t 200 -t 2000 -t 20000
```

While it runs you might have time to get a cup of tea, depending how fast the computer and disks are. Anything over a second or so per image suggests you have huge images or something is wrong.

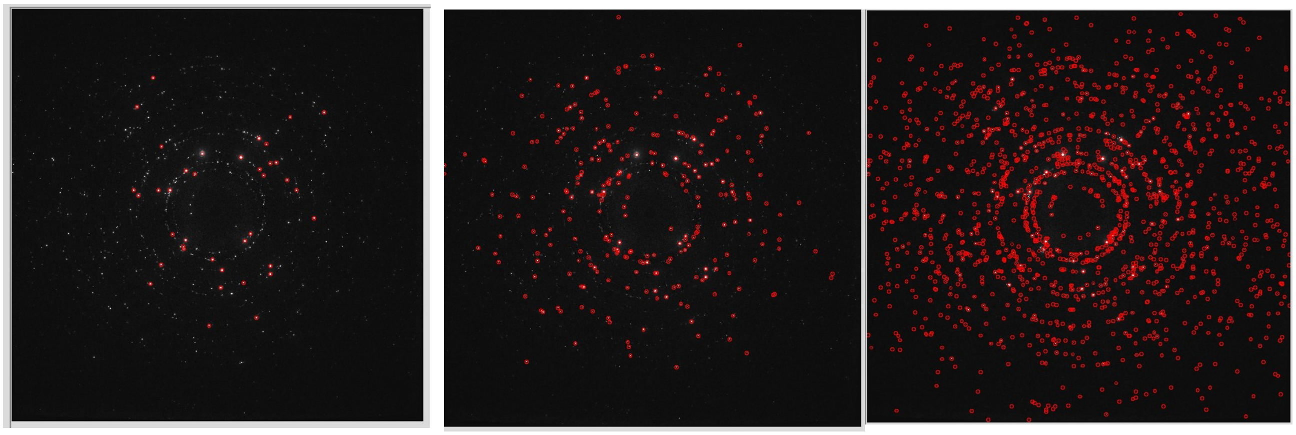
The program will spew masses of information to the screen about what is is doing and create some output files. These will be named:

```
demo_t200.spt
demo_t2000.spt
demo_t20000.spt
demo_t200.flt
demo_t2000.flt
demo_t20000.flt
```

If you did not give the "-o demo.spt" argument to the program then it writes "peaks_..." instead, but that gets confusing when you have lots of different scans. You might prefer something related to the stem and image numbers. Inside the .spt files you have a dump of the image header and then the list of spots found on that image, for each image in the series. This corresponds to a series of 2D peaksearches. In the flt files you have peaks which have been merged in 3D across the series of frames, so any spots which have pixels in common from one frame to the next are combined to make a single spot. We will come to the details of what is in these files later. All of these files are in ascii text format, so you can open them in any good text editor.

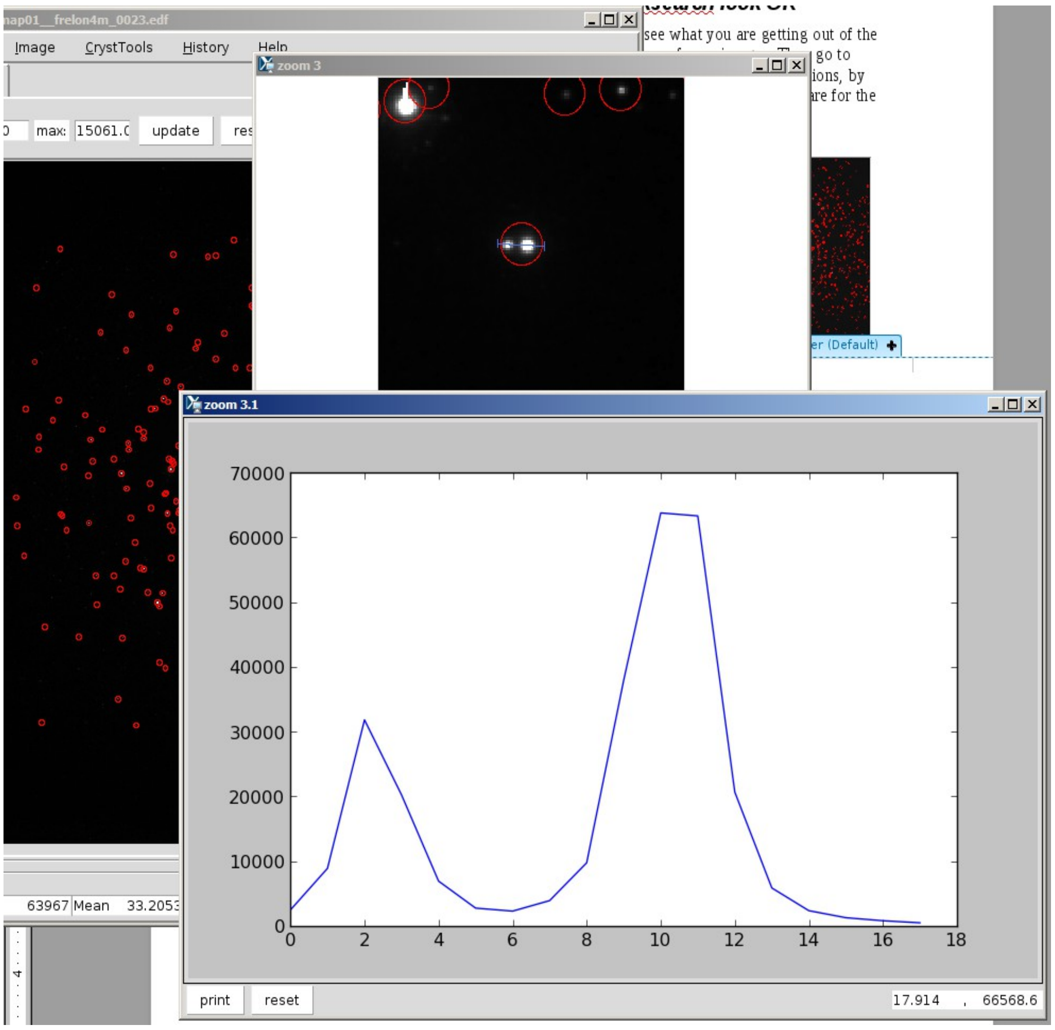
Checking the diffraction spots found in peaksearch look OK

Before you go any further it makes sense to have a look and see what you are getting out of the images. Turn again to the fabian image viewer and open up one of your images. Then go to CrystTools → peaks → read peaks and read in one of the spt files. Also check the options, by default it only shows peaks with more than 4 pixels inside. The three pictures below are for the 20000, 2000 and 200 thresholds:



In this specific example, a large number of the peaks at the 200 threshold level correspond to "satellite" reflections around strong peaks that come from an attenuator on the front of the detector. We will need to throw these out somehow. For the purposes of writing this document in a short amount of time, we will just ignore them in the first pass.

The next figure illustrates the reason for using multiple thresholds:

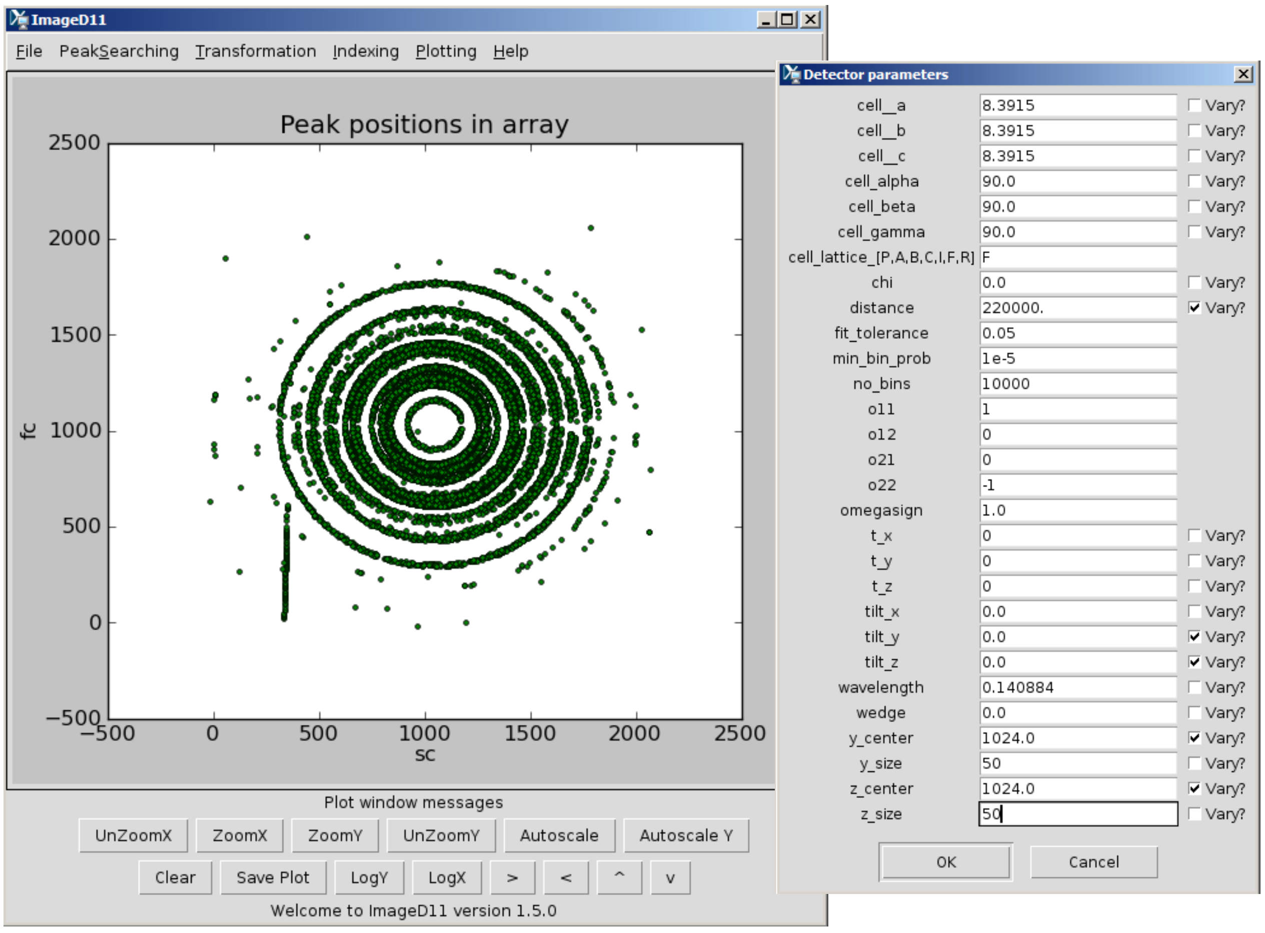


There are two spots which are close together and at the 2000 threshold level they are merged to make a single reflection. That is not good. At the 20000 threshold level they are separated into two spots. Therefore we would like to eventually combine these outputs. We will come to that with the merge_flt.py script later.

Calibration of the detector geometry

To process the diffraction data we need to go from 2D image coordinates to 3D diffraction scattering vectors. This means we need to know where the detector was compared to the sample and what the wavelength was, which way up it was, etc.

We start by opening the .flt file in the transformation menu of the ImageD11_gui.py program. Nothing will happen in the gui but you might get a message back in the console where you started. Next you can select “plot y/z” from the transformation menu to see where the spot were found on the detector. You notice the usual diffraction rings and also a horrible line which comes from a detector glitch on a single frame. We will see how to remove those problematic peaks later. For now we want to do the calibration, so we look back at the notebook to fill in some meta-data. Choose the “edit parameters” menu item.



Welcome to ImageD11 version 1.5.0

cell__a	8.3915	☐ Vary?
cell__b	8.3915	☐ Vary?
cell__c	8.3915	☐ Vary?
cell_alpha	90.0	☐ Vary?
cell_beta	90.0	☐ Vary?
cell_gamma	90.0	☐ Vary?
cell_lattice_[P,A,B,C,I,F,R]	F	
chi	0.0	☐ Vary?
distance	220000.	✓ Vary?
fit_tolerance	0.05	
min_bin_prob	1e-5	
no_bins	10000	
o11	1	
o12	0	
o21	0	
o22	-1	
omegasign	1.0	
t_x	0	☐ Vary?
t_y	0	☐ Vary?
t_z	0	☐ Vary?
tilt_x	0.0	☐ Vary?
tilt_y	0.0	✓ Vary?
tilt_z	0.0	✓ Vary?
wavelength	0.140884	☐ Vary?
wedge	0.0	☐ Vary?
y_center	1024.0	✓ Vary?
y_size	50	☐ Vary?
z_center	1024.0	✓ Vary?
z_size	50	☐ Vary?

I filled out the unit cell parameters, and “F” lattice as magnetite is face centred cubic at room temperature. Angstrom were chosen as units, so the same must be used for wavelength. If you don't know this information for your sample then the best thing to use a another material for the calibration and come back to your unknown after you have got the beam centre and distance sorted out. Going through the rest of the parameters that were changed:

chi = 0 : this is the chi rotation of the rotation axis around the beam. Should be zero for a vertical axis with the axis itself defining the "z" laboratory direction.

distance = 220000. It was about 220 mm. All the distance units for the detector geometry are the same, but since I want to get a grain map out with positions in microns directly, this is the unit to use.

o11,o12,o21,o22: These are the detector flips. You can explore them in fabian and read the definitions here: http://sourceforge.net/apps/trac/fable/wiki , there should be "geometry" and "fableimageorientdoc" files linked.

Omegasign: 1.0 is a right handed rotation about z. Put your right fist over the axis. Your fingers point in the positive sense when your thumb points along the axis. Use -1 for a left handed rotation.

The t_x, t_y and t_z are the translations of a single grain with respect to the rotation axis and beam centre. We will leave them as zero for now. If you calibrate using a single crystal and a large rotation range you can try to fit them. If your samples is large compared to the pixel size (true here) you can systematically step them around to bring different grains to the good calibration position. Units are the same as distance and pixel size (so microns here).

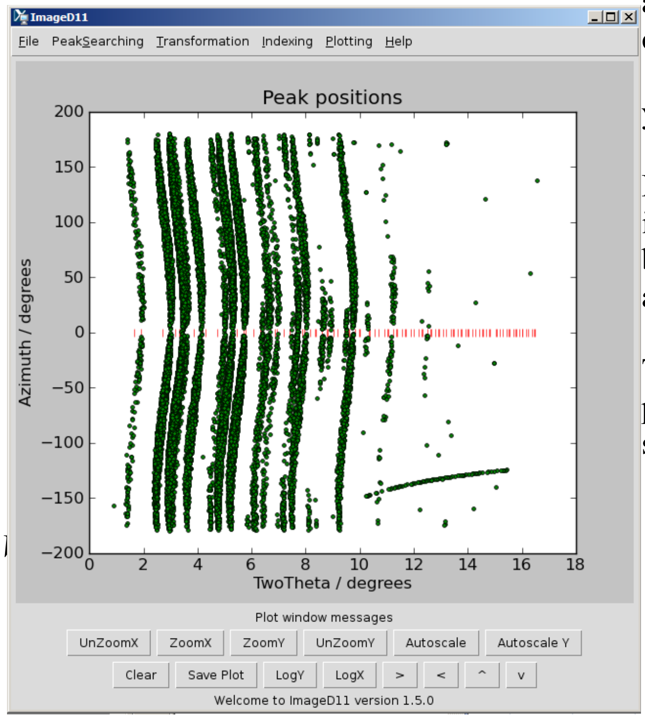
The "x,y,z" labels follow the convention of x along the beam, z perpendicular to the beam in the same plane as the rotation axis and y making a right handed set.

tilt_x, tilt_y and tilt_z are the tilts of the detector surface compared to the beam direction. Units are radians. Do not attempt to fit the tilt_x rotation (around the beam) in this gui, it makes no sense here, but you fit it later when you have indexed something.

Wavelength: of the radiation, units are the same as those used for the unit cell parameters. Angstrom here, but you can use inches if you prefer.

Wedge: This is the angle between the X-ray beam and the rotation axis. At zero they are exactly 90 degrees apart. Units are degrees. Sign should be verified.

y_center is the beam position in the slow pixel direction, in units of pixels.
z_center is the beam position in the fast pixel direction, in units of pixels.
Note that the beam centre does not depend on the image flips. The program first subtracts the beam centre and then applies the flips afterwards. This means you can try out a series of flips when you are not sure without needing to change the beam center. It also means the labels are misleading.


ImageD11
File PeakSearching Transformation Indexing Plotting Help
Peak positions
Azimuth / degrees
TwoTheta / degrees
200 150 100 50 0 −50 −100 −150 −200
0 2 4 6 8 10 12 14 16 18
Plot window messages
UnZoomX ZoomX ZoomY UnZoomY Autoscale Autoscale Y
Clear Save Plot LogY LogX > < ^ v

Welcome to ImageD11 version 1.5.0

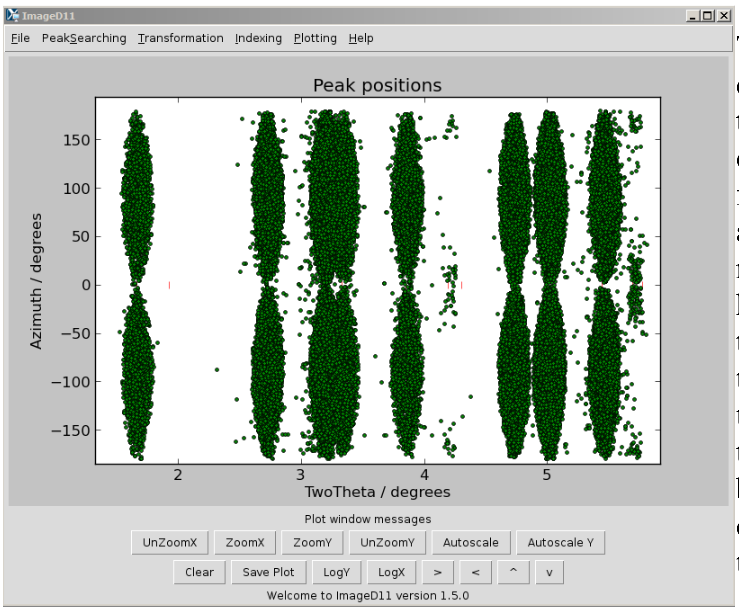
y_size, z_size: pixel sizes in microns.

Having gotten all that filled in, you click "OK" in the dialog, then clear the plot (bottom left button or plotting menu. Then click "plot tth/eta" and next "add unit cell peaks".

This is the radial transformation of the peak positions. The horizontal axis is the two theta scattering angle, between the direct beam and the diffracted ray. The vertical axis (azimuth) is the angle on the cone, so between the ray and the vertical.

When a sample is strain free we expect all rings with the same h,k,l indices to fall onto rings which have the same twotheta values. The different orientations of the grains spreads them around the azimuth. When the calibration is good, we should have nice vertical lines, but here we do not. Also we still have those detector glitches over on the right of the picture. To improve things first we zoom in on the low angle peaks. The gui only uses the peaks shown on the screen for fitting, so in this way we make the fitting problem easier.

To improve the fit the best is to select an single isolated ring (click and drag to zoom) and then fit the distance and centre only, then fit the tilts from the full range. Onec you find the good parameters you should see something like this for the low angle region:


ImageD11
File PeakSearching Transformation Indexing Plotting Help
Peak positions
Azimuth / degrees
150
100
50
0
−50
−100
−150
2
3
4
5
TwoTheta / degrees
Plot window messages
UnZoomX ZoomX ZoomY UnZoomY Autoscale Autoscale Y
Clear Save Plot LogY LogX > < ^ v
Welcome to ImageD11 version 1.5.0

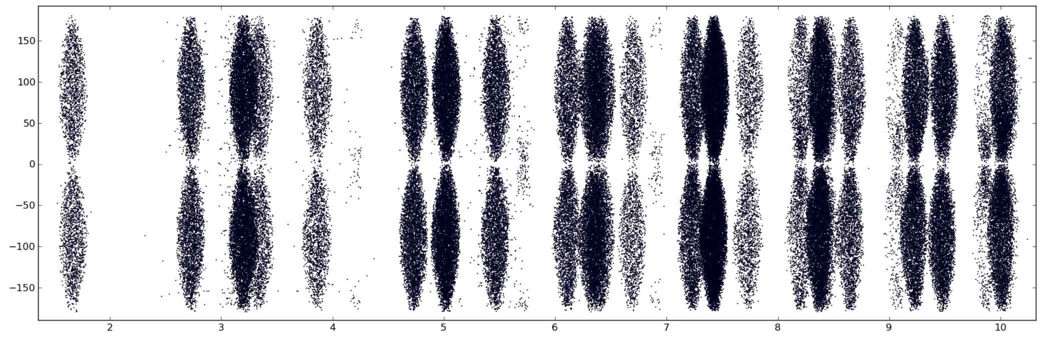
This is from the 200- threshold. From a quick inspection of the plot you can see that each hkl ring is forming a pair of elliptically shaped distributions as a function of azimuth. For a small sample and beam size we would expect all reflections with the same hkl indices to have exactly the same two theta angles and therefore we would get vertical lines on this plot. The scatter we observe is due to the peak shifts which are observed due to the centre of mass positions of the grains being shifted. This is good, it means we can expect to get something meaningful in the final grain map.

Once you are happy with the calibration save the parameter file and also save a new colfile file which will have the two theta and azimuth numbers inside it.

Cleaning up the peaks

Since there are some problematic peaks in this dataset from a detector glitch a procedure to clean them up is described here. You can do this much more easily using the fable gui interface, but the python based method is shown as it illustrates how you can work with the peaksearch output for other purposes. We will select just a few low angle rings to use for indexing as this will be much faster
First, we start up an ipython interpreter with the pylab module loaded (for plotting):

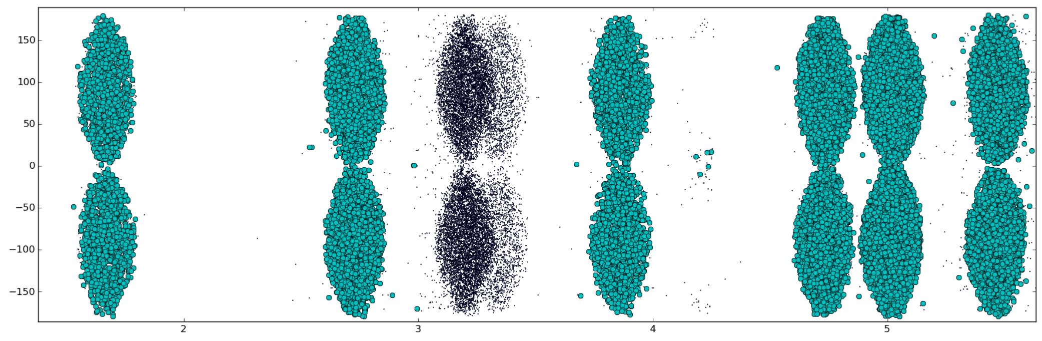
```
% ipython -pylab
…
In [1]: from ImageD11.columnfile import columnfile
In [2]: c = columnfile("demo_t2000c.flt")
In [3]: plot(c.tth, c.eta, ",")
```



The first line loaded the columnfile module from ImageD11 which we use to read and write the ascii files which are used in the program to describe the peaks. The second line reads in the file we saved from the gui (with tth and eta columns added now) and the third line plots the tth and eta columns. If you type "c." and then hit tab within ipython you get a list of the attributes the object contains, in this case the list of spot properties. These can also be found in c.titles.

To filter out peaks we do not wish to use we use a series of logical operations and apply these using the filter method of the columnfile object. Since we want to only use peaks with reasonably accurate centre of mass positions we first remove anything which have less than 4 pixels:

```
In [6]: c.filter( c.Number_of_pixels > 4)
In [7]: c.filter( logical_or(c.tth < 3, c.tth > 3.6) )
In [8]: c.filter( c.tth < 5.62)
In [9]: plot(c.tth, c.eta, "o")
In [10]: c.writefile("cleanpeaks.flt")
```



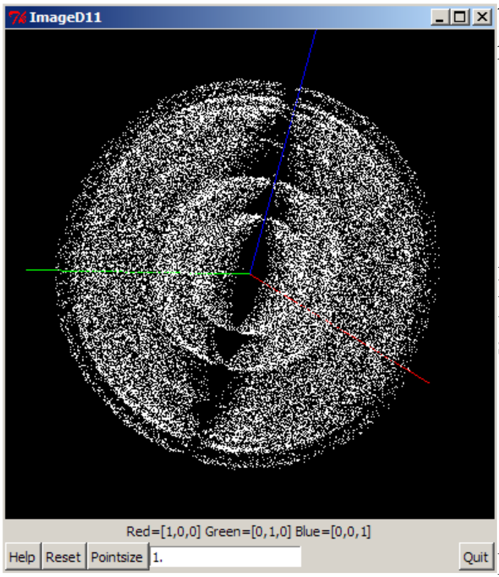
Only the blue spots are kept at this stage, and we see they correspond to a series of rings which are not quite overlapped. This is not actually necessary for the indexing in ImageD11, but it can make life easier. We have saved the file into cleanpeaks.flt.

We now go back to the ImageD11_gui.py program, transformation menu and do:
load filtered peaks (cleanpeaks.flt)
load parameters
plot tth/eta
add unit cell peaks
compute g-vectors
save g-vectors

Now we are ready to try to index some grains.

Indexing

Select the indexing menu in ImageD11_gui.py and read in the g-vector file you have just created. At this point we will assume you have the image flips and omega step size and sign correct. If it is the first time you are running the program then you need to run through the whole procedure with a single crystal first.



Once you have loaded g-vectors select "plot x/y/z" from the indexing menu. With a single crystal you will see a nice lattice but with these data you see the spheres in reciprocal space (powder rings with a rotation of the sample)

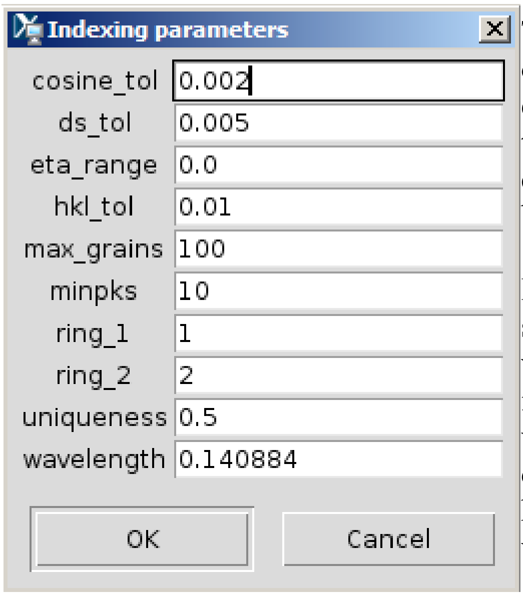
First select "assign peaks to powder rings"and the open up the indexing parameters menu. On the console you should have output something like this:

```
DEBUG    : Running: myindexer.assigntorings( )

Maximum d-spacing considered 0.69358
Ring assignment array shape (33267,)
Ring     (  h,  k,  l) Mult  total indexed to_index
Ring 10  ( -4, -4,  0)   12   3191      0   3191
Ring 9   ( -5, -1, -1)   32   6315      0   6315
Ring 8   ( -4, -2, -2)   24   2789      0   2789
Ring 7   ( -4, -2,  0)   24      0      0      0
Ring 6   ( -3, -3, -1)   24      4      0      4
Ring 5   ( -4,  0,  0)    6   1536      0   1536
Ring 4   ( -2, -2, -2)    8      0      0      0
Ring 3   ( -3, -1, -1)   24      0      0      0
Ring 2   ( -2, -2,  0)   12   2730      0   2730
Ring 1   ( -2,  0,  0)    6      0      0      0
Ring 0   ( -1, -1, -1)    8   1138      0   1138
Using only those peaks which are assigned to rings for scoring trial matrices
Shape of scoring matrix (17703, 3)
```

To understand what these mean it helps to have an idea what the program is trying to do. As a first step each of the observed spots is assigned to a hkl ring. Since you provided the unit cell in the transformation parameters and has a list of g-vectors it will try to assign each to the nearest ring. It decides whether or not to use peaks, and whether computed hkls belong to the same ring according to a tolerance in d-star, or 1/d-spacing. This is the ds_tol parameter and the default is 0.005. If you want this as two theta you would need to compute using:

wavelength x d-star = 2 sin(theta)

We will use double the default values as we can see from the ring assignment that it is using about half of the spots, eg, those closest to the rotation axis in the centre of the sample. By increasing this to 0.01 we pick up most of the spots without overlapping the computed rings.


Indexing parameters

cosine_tol	0.002
ds_tol	0.005
eta_range	0.0
hkl_tol	0.01
max_grains	100
minpks	10
ring_1	1
ring_2	2
uniqueness	0.5
wavelength	0.140884

The ring_1 and ring_2 parameters are used to select a pair of rings to use for generating orientation matrices. For each peak assigned in ring_1 the program will compute the angle between this ring and the peak in ring_2 and compare this to the angle which are theoretically possible based on the unit cell.

If the cosine_tol parameter is position then best pairs of spots which have the cosine of the angle between them within cosine_tol of the ideal value are retained for indexing. If the parameter is negative then the absolute value is used and all pairs are retained. For reference, a 1 degree error gives a cosine_tol of about sin(1) = 0.017. We had a step of 0.1 degrees so 0.0017 gives -0.002 as a good value.

Having selected rings 2 and 10 as the ones we want to try (why not) we run the generate trial orientations option from the indexing menu with cosine_tol of -0.002 and get the following output:

```
hkls of rings being used for indexing
Ring 1: [(-4, -4, 0), (-4, 0, -4), (-4, 0, 4), (-4, 4, 0), (0, -4, -4), (0, -4,
4), (0, 4, -4), (0, 4, 4), (4, -4, 0), (4, 0, -4), (4, 0, 4), (4, 4, 0)]
Ring 2: [(-2, -2, 0), (-2, 0, -2), (-2, 0, 2), (-2, 2, 0), (0, -2, -2), (0, -2,
2), (0, 2, -2), (0, 2, 2), (2, -2, 0), (2, 0, -2), (2, 0, 2), (2, 2, 0)]
Possible angles and cosines between peaks in rings:
60.0 0.5
90.0 0.0
120.0 -0.5
Number of peaks in ring 1: 5226
Number of peaks in ring 2: 4475
Minimum number of peaks to identify a grain 40
Percent done 99.981%   ... potential hits 142908
Number of trial orientations generated 142908
Time taken 8.6922981739
```

This is a lot of possible orientations! To decide which ones are any good we need to decide on criteria for accepting an orientation matrix as being true. There are two parameters which control this decision, the hkl_tol and number of peaks. The idea is that a true orientation will generate lots of peaks which show up in the right positions, but a false one will not. In this case we have measured about half of the possible data (two 90 degree wedges) and we are using a subset of the rings. If we add up the multiplicities of the rings used and take account of having half of the data we expect to get about 47 peaks for a good grain, so anything with more than 40 is likely to be OK. The hkl_tol is the difference between the integer hkl values which we should get when the peak position is perfect and the actual ones which come out due to experimental error and that the centre of mass of the grain has not yet been accounted for. Values less than 0.05 are often a good start.

Having set minpks to 40 and hkl_tol to 0.05 we rin "score trial orientations" and the program goes through and tests these out, printing the following output:

```
DEBUG    : Running: myindexer.scorethem( )

Scoring 142908 potential orientations
Tested    64277    Found       99    Rejected        8 as not being unique
Number of orientations with more than 40 peaks is 100
Time taken 32.7021529675
UBI for best fitting
[[ 4.55046822 -0.33566335  7.03863623]
 [ 0.53714342  8.36978834  0.03201506]
 [-7.02516621  0.43130667  4.57145652]]
Unit cell
(8.388195924157035, 8.3870677153324724, 8.3926873198502907, 90.014017345038042,
89.947709023251932, 90.113883092167725)
Indexes 47 peaks, with <drlv2>= 0.0343937443714
That was the best thing I found so far
Number of peaks assigned to rings but not indexed =  23778
```

Note that the program stopped when it got to 100 grains due to the max_grains parameter. Change that and run it again to get some more grains. After it runs out of trial orientations you can repeat the assignment to rings to see how things are looking:

```
DEBUG    : Running: myindexer.assigntorings( )

Maximum d-spacing considered 0.69358
Ring assignment array shape (33267,)
Ring     (  h,  k,  l) Mult  total indexed to_index
Ring 10  ( -4, -4,  0)   12   5226    820   4406
Ring 9   ( -5, -1, -1)   32  10334   2087   8247
Ring 8   ( -4, -2, -2)   24   4483   1120   3363
Ring 7   ( -4, -2,  0)   24      0      0      0
Ring 6   ( -3, -3, -1)   24      5      1      4
Ring 5   ( -4,  0,  0)    6   2549    381   2168
Ring 4   ( -2, -2, -2)    8      0      0      0
Ring 3   ( -3, -1, -1)   24      0      0      0
Ring 2   ( -2, -2,  0)   12   4475    891   3584
Ring 1   ( -2,  0,  0)    6      0      0      0
Ring 0   ( -1, -1, -1)    8   1828    512   1316
Using only those peaks which are assigned to rings for scoring trial matrices
Shape of scoring matrix (28900, 3)
```

It is clear that we have picked up a lot of spots and grains, but that there are a lot left. This is probably because the grains are quite far from the center of rotation, and we have been strict in trying to only accept things which are meaningful. A better check as to whether your grains are real is given by the histogram fit quality in the indexing menu. For each grain found, you get a plot of the number of observed peaks in the dataset versus the error in peak position. True grains have a lot of peaks with a small error. Random orientations find more peaks as the allowed error increases.

From the picture you can see a lot of our grains are likely to be good, but perhaps we have picked up some “noise” orientations. We can save the orientations into a ubi file and try to fit their centre of mass positions to decide if they are any good or not (indexing menu, save UBI matrices).

Fitting grain positions, makemap.py

The makemap.py script reads in the parameters found from the transformation menu, the ubi matrices from the indexing step and the filtered peaks from the peaksearch (and cleanup) steps. It creates as output another ubi file which also contains grain positions and a .new file which has the filtered peaks with grain assignments and extra columns added:

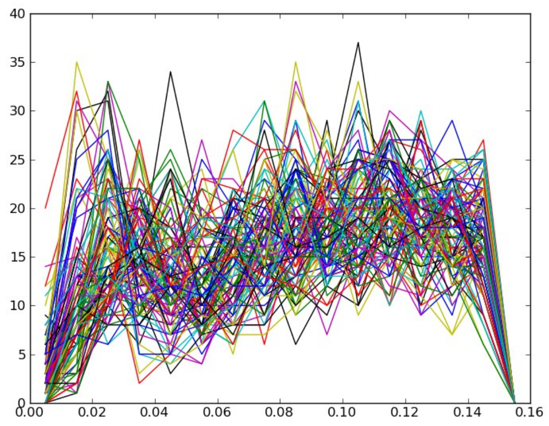
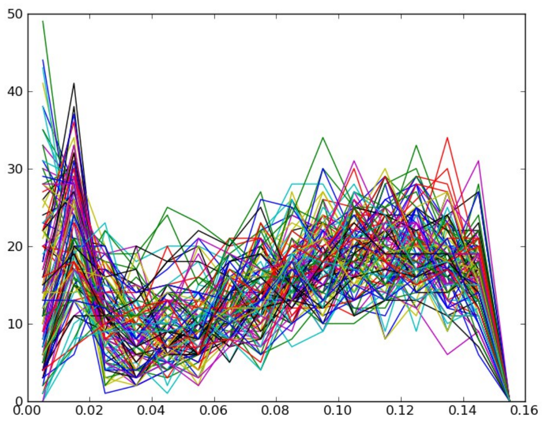
```
rnice6-0203:/data/id11/inhouse/jon/mar10/fe3o4map01_/guide % makemap.py --help
/sware/exp/fable/standalone/debian6/lib/python2.6/site-
packages/ImageD11/refinegrains.pyc
Usage: makemap.py [options]

Options:
  -h, --help            show this help message and exit
  -p PARFILE, --parfile=PARFILE
                        Name of parameter file
  -u UBIFILE, --ubifile=UBIFILE
                        Name of ubi file
  -U NEWUBIFILE, --newubifile=NEWUBIFILE
                        Name of new ubi file to output
  -f FLTFILE, --fltfile=FLTFILE
                        Name of flt file
  -F NEWFLTFILE, --newfltfile=NEWFLTFILE
                        Name of flt file containing unindexed peaks
  -t TOL, --tol=TOL     Tolerance to use in peak assignment
  --no_sort             Sort grains by number of peaks indexed
  -s SYMMETRY, --sym=SYMMETRY
                        Lattice symmetry for choosing orientation
  --tthrange=TTHRANGE   Two theta range for getting median intensity
  --omega_no_float      Use exact observed omega values
  --omega_slop=OMEGA_SLOP
                        Omega slop (step) size
```

```
% makemap.py -u first.ubi -p demo_t20000.par -f cleanpeaks.flt -U first.map -s
cubic -t 0.05 --omega_slop=0.05
…
lots of output
…
```

This program has converted first.ubi into first.map, which we hope is better due to having good grain positions inside.

We can compare the before and after fitting using the plotgrainhist.py script:

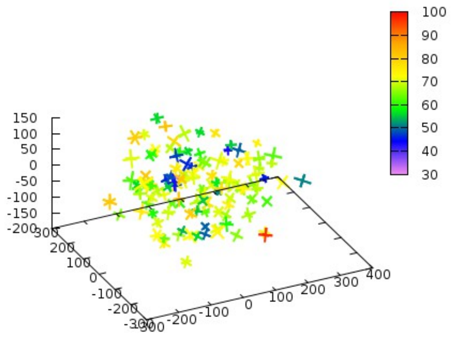
```
 plotgrainhist.py cleanpeaks.flt demo_t20000.par first.ubi 0.15 15
 plotgrainhist.py cleanpeaks.flt demo_t20000.par first.map 0.15 15
```

As you can see, it becomes a lot better. We iterate the process using first.map as input to makemap.py and changing the tolerance to 0.03, then 0.02. It looks vaguely promising at this stage.

We can take an initial look at our grains using the plotImageD11map.py script:

```
 plotImageD11map.py first.map junk 0.1
```



The “junk” command line argument is the name of a text file which is used by gnuplot to make the plot. The colors represent the number of spots indexed by each grain. The little crosses are the a,b and c real space axes of the crystals and the size is proportional to the grain size (from the intensities and using the 0.1 scale factor from the command line).

Now we are ready to try to fit the experimental geometry using the indexed grains and to go back and index the rest of the grains in the sample.

Fitting the geometry using indexed grains

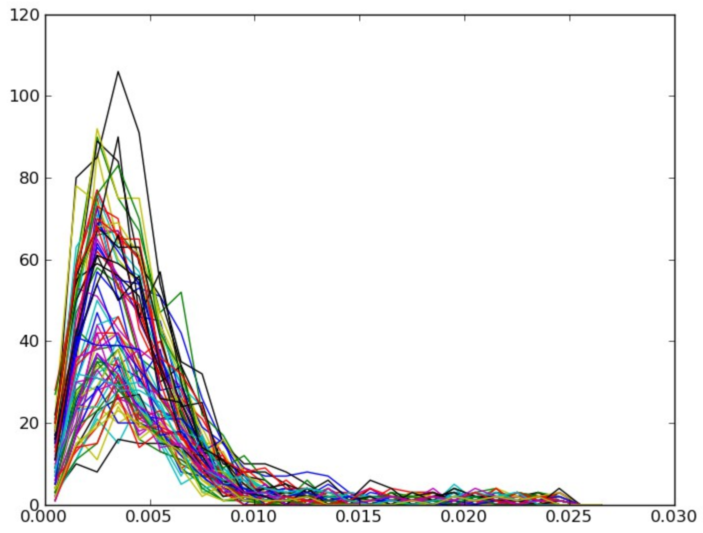
First repeat the makemap.py step adding the “--no_sort” command line argument. This keeps the order of the output grains the same as the input grains so the following scripts know which is which more easily. Also, use the full range of data and just remove the peaks which are saturated or with very few pixels, eg:

```
In [1]: from ImageD11.columnfile import *

In [2]: c = columnfile("demo_t2000.flt")

In [3]: c.filter(c.Number_of_pixels>4)

In [4]: c.filter(c.IMax_int < 55000)

In [5]: c.writefile("demo_t2000c.flt")

...

makemap.py  -u first.map -p demo_t20000.par -f demo_t2000c.flt -U first.map -s
cubic -t 0.02 --omega_slop=0.05  --no_sort
```

```
refine_em.py cleanpeaks.flt.new first.map demo_t20000.par
```

We then use the refine_em.py script to fit each grain one by one. This selects the peaks assigned to each grain by makemap.py and runs fitgrain.py for each one. Once that is finished you can use avg_par.py to average the parameters refined from each grain and make a new parameter file. The main difference to the original is that the tilt_x and wedge numbers have been fitted.

To get convergence, it can be worthwhile to go back to makemap.py with the new parameter file and then repeat refine_em.py. It all depends how many grains you have as to whether the original peak assignments are any good. The process will also work better if the entire dataset is used and not just the rings selected for indexing. If you only use the low angle peaks then the refinement of sample to detector distance will be very unstable. You can fix this parameter by making a copy of refine_em.py and editing in



The output parameters produced here are usually a good starting point to get going with fitallb if you use that to refine strain tensors. Coming back to plotgrainhist.py we now have a convincing plot indicating are grains are substantially correct (note the altered x-axis scale):

Finding the rest of the grains, grid_index.py

Now you have good parameters it is worthwhile to step around the t_x, t_y and t_z parameters in the transformation menu to try indexing grains which are not on the rotation axis. You can do this by manually going through the ImageD11_gui.py and then looking in help->history to see the recording of your actions, and then edit this script to loop over translation values. Alternatively dig out the grid_index.py script from ImageD11/sandbox and edit it according to the parameters you have discovered above for indexing and the sensible tolerance and numbers of peaks per grain.

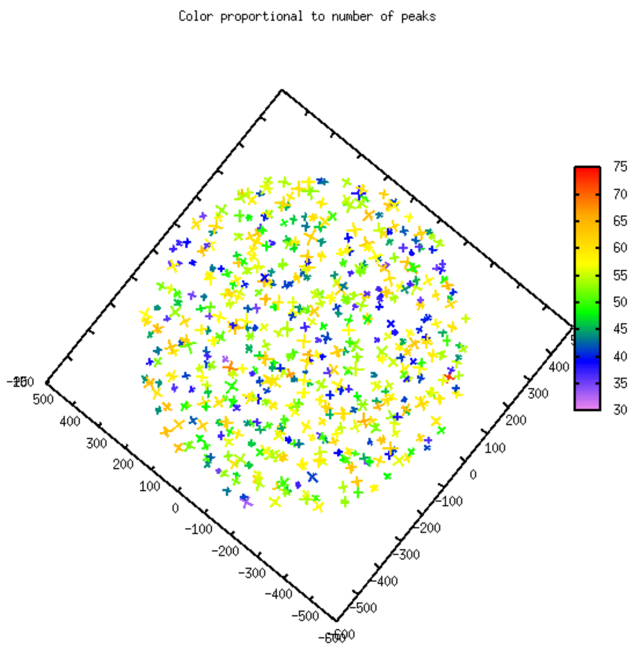
The entire script is reproduced here, together with some additional comments on what it is doing:

```
from ImageD11 import indexing, transformer
from ImageD11 import grain
import sys, os, numpy as np

mytransformer = transformer.transformer()
#
# Your work starts here:
#
mytransformer.loadfiltered( sys.argv[1] )
mytransformer.loadfileparameters(  sys.argv[2] )
tmp = sys.argv[3]
# arguments are peaksfile, parameters file and name for output
```

```
def doindex( gve, x, y, z):
    """ Function to find orientations """
    myindexer = indexing.indexer()
    myindexer.readgvfile( gve )
    # ring1 and ring2 depend on the sample
    for ring1 in [5,10,2]:
        for ring2 in [5,10,2,0]:
            # these are the indexing parameters to use:
            myindexer.parameterobj.set_parameters(  {
                'ds_tol': 0.004,
                'minpks': 35,
                'cosine_tol': 0.002,
                'max_grains': 1000,
                'hkl_tol': 0.03,
                'ring_1': ring1,
                'ring_2': ring2
                } )
            myindexer.loadpars( )
            myindexer.assigntorings( )
            myindexer.find( )
            myindexer.scorethem( )
    # saves ubis into a file with the x,y,z position in the grid
    grains = [grain.grain(ubi, [x,y,z]) for ubi in myindexer.ubis]
    grain.write_grain_file("%s.ubi"%(tmp),grains)
    return len(grains)

import random
random.seed(42)
# Generate a grid and step over it in randomised order
translations = [(t_x, t_y, t_z)
        for t_x in range(-600, 651, 50)
        for t_y in range(-600, 651, 50)
        for t_z in range(-300, 301, 50) ]
random.shuffle(translations)
import time
start = time.time()

# Make a copy of the peaks file
os.system("cp %s %s.flt"%(sys.argv[1],tmp))

first = True
# loop over translation positions
for t_x, t_y, t_z in translations:
    # create gvector file using this centre of mass position...
    mytransformer.updateparameters( )
    mytransformer.parameterobj.set_parameters(  {
                  't_x':t_x, 't_y':t_y, 't_z':t_z
                  } )
    mytransformer.compute_tth_eta( )
    if first:
        mytransformer.addcellpeaks( )
    mytransformer.computegv( )
    mytransformer.savegv( tmp+".gve" )
    if first:
        first=False
    # … and index grains here
    ng = doindex( tmp+".gve", t_x,t_y,t_z)
    print t_x, t_y, t_z, ng,time.time()-start
    if ng > 0:
        # run makemap with tolerance 0.03
        ret = os.system(
      "makemap.py -u %s.ubi -U %s.map -s cubic --omega_slop=0.05 "%(tmp,tmp) +
                                "-f %s.flt  -t 0.03 -p %s  "%(
                                   tmp,sys.argv[2]))
        if ret !=0 :
            print "bad 1"
            raise
        # repeat with 0.01
        ret = os.system(
      "makemap.py -u %s.map -U %s.map -s cubic --omega_slop=0.05 "%(tmp,tmp) +
                                "-f %s.flt  -t 0.01 -p %s  "%(
                                   tmp,sys.argv[2]))
        if ret !=0 :
            print "bad 1"
            raise
        # remove bad grains (less than 35 peaks)
        os.system("cutgrains.py %s.map %s.map 35"%(tmp,tmp))
        # final run of makemap
        ret = os.system(
       "makemap.py -u %s.map -U %s.map -s cubic --omega_slop=0.05  "%(tmp,tmp) +
                                "-f %s.flt -F %s.flt -t 0.01 -p %s "%(
                                    tmp,tmp,sys.argv[2]))
        if ret !=0 :
            print "bad 3"
            raise
        # finally save grains and update peaks with unindexed peaks
        open("all%s.map"%(tmp),"a").write(open("%s.map"%(tmp)).read())
        mytransformer.loadfiltered("%s.flt"%(tmp))
```

Once the script has run (it takes a while) then we get a fairly promising grain map out containing 517 grains. We run makemap.py a final time using all of the grains and all of the peaks as during the grid_index.py process.

```
makemap.py -u allgrid.map -p first.par -f demo_t2000c.flt -s cubic -t 0.01
--omega_slop=0.05 -no_sort -U allgridref.map
```



Looking at the results

After the final makemap.py run we have an output grain map and an output peak assignment in the .flt.new file. The plotImageD11map.py script gives the picture showing the centre of mass grain positions. We can get the unit cell parameters via ubi2cellpars.py. We can also look at the intensities of the different peaks which have been assigned to the different grains and also how many peaks are in the dataset which are not indexed.