

Optical Imaging to Improve Crystal Hit Rates with Serial Crystallography on I24

Laurence Cullen, Supervised by Danny Axford
20th of September 2016

Abstract

Over this 12 weeks project, two offline imaging techniques were tested in an attempt to identify crystal locations on a serial crystallography sample holder. A data collection setup was built in addition to a software pipeline to process optical images and extract an intensity value for each cell in the sample chip. These values were compared to X-Ray diffraction data of the sample chips to see if correlations between optical images and crystal locations could be observed. A similar experiment was run using polarised light to see if the way that crystals altered linearly polarised light could be used as a good guide to their presence. Whilst the fitting and read out tools themselves were effective it was not possible to strongly correlate output from optical imaging and diffraction data both with and without polarising filters.

1 Introduction

This project is looking to improve pre-experiment steps for serial crystallography in which X-Ray diffraction data is collected from thousands of individual crystals. These crystals are at random orientations with the idea being that good coverage of the full rotation space can be obtained. This compares to the classic rotation experiment where a single crystal is exposed to X-Ray beam and is slowly turned on a goniometer, diffraction data being collected at a large number of angles with fine angle resolution.

The classic rotation experiment requires a crystal to be exposed for several seconds meaning any reaction taking place on or below this time scale is out of reach. With serial crystallography and the new FEL, free electron lasers, time resolutions of 10^{-15} seconds are possible. Snapshots can be taken of reactions part way through to greatly increase understanding. The idea is that the reaction is triggered at exactly the same time before each crystal takes beam so that all the X-Ray diffraction data is of the same intermediate species.

Another advantage of serial crystallography with femtosecond pulses is that the problem of radiation damage is removed as the pulse has such a short duration that the crystal has no time in which to get destroyed before the entire pulse has passed through. This is known as diffraction before destruction, the term and technique coined in [1].

Due to the advantages of FEL, techniques to optimise sample delivery for them are currently receiving more attention. The first attempts involved a high-speed jet of fluid with small crystals mixed in. The FEL was repeatedly triggered with the hope of hit-

ting the crystals suspended in the fluid jet. This was not an optimal technique as there was a high degree of waste, lots of crystals were used and there were large numbers of FEL exposures which did not connect with a crystal.

A new technique is being tested on I24 at Diamond Light Source by filling a sample "chip" which has 11,664 holes or "cells" and dimensions of 2cm by 2cm with as many individual crystals as possible. This is then placed into the X-Ray beam and diffraction data is taken for each cell which is later combined in an attempt to solve the molecules structure. The location of every potential crystal is known and the bigger problem this project addressed is that some of the cells are filled and others not. If this can be determined in advance then time will be saved at all stages of the data collection processing steps.

2 Theory

Due to the way that the sample chips are loaded with crystals, the number of cells with good crystal candidates is often well below 100%. To reduce overheads in terms of data processing and collection time it would be helpful if prior to X-Ray exposure the cells on the sample chip which contained good crystal candidates could be distinguished from cells with bad candidates. This is complementary to previous serial crystallography pre-screening work [3] using UV spectrometry to identify crystal locations on the same type of sample chips used in this experiment.

This project tested two additional ideas for detecting the presence of crystals on sample chips. Firstly the idea that light from a background source passing

through the sample chip would be more strongly attenuated when crystals were present and that examining the emission per cell before and after a chip was loaded with crystals could give clues as to whether good crystal samples were present in that cell.

The second idea used a linearly polarised light source with a polarising filter in front of the camera aligned so that all the emission from the background light was blocked. When a sample chip was placed in front of the background light the birefringence of the crystals present in the sample chip would distort the polarisation of the background light so that it would no longer be cancelled out by the filter in front of the camera. This would increase the quantity of light detected by the camera for cells with crystals present. To test the efficacy of these methods predictions of crystal locations were compared to X-Ray diffraction data for each cell to see if significant correlations existed.

The motivation for investigating these techniques is that they are very quick and images can be taken in a minute or two compared to 10+ minutes for other scanning techniques.

3 Method

3.1 Data Acquisition Setup

Figure 1 shows the setup used for collecting images of sample chips, a magnetic sample chip holder was chosen to ensure that captured photos are all closely aligned. The light source has a linearly polarising filter in front of it with another polarising filter before the Mako G Camera, this filter can be rotated so that the effect of crystals on the purely polarised background light can be observed. For most of the project, images were taken without polarising filters and just the effect on the overall light level was observed.

Unfortunately due to beam time scheduling it was only possible to collect data during the first and second last week of the project so there were limited opportunities to iterate on the data collection techniques.

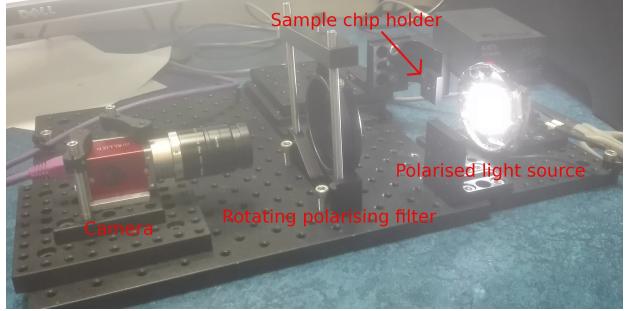


Figure 1: Image collection setup, sample chip holder is magnetic and clips onto chip holder

3.2 Fitting Problem Setup

The problem of extracting how much light was coming through each cell in the sample chips and how this varied depending on the presence of crystals was complicated by several factors. Firstly due to the manufacturing process the holes in the sample chip are of different sizes so that even with no crystals present there is a large difference in the amount of background light coming through. To defeat this effect images were captured before and after the chips were loaded with crystals and the differences were divided out so that the effects of the size of each cell could be removed.

Another problem was the misalignment of the chips, random translations in x and y of a certain amount were observed in photos depending on how the chips were mounted in the sample holder attached the magnetic clamp. Some variation in the rotation was also present, usually between 0 and 10 degrees but in some cases, the chips were flipped by 90 or 180 degrees complicating how the cells were indexed. Many of the chips also had significant gaps due to the fragility of their design and general wear and tear.

To account for the randomised nature of the chips within the captured photos a theoretical layout of the chip detailing the exact x,y position of each cell (figure 2) was taken and modified and masked over the photo. The parameters defining how the theoretical mask was applied included the pixel to physical size scaling, x and y translation and the angle of rotation. When all these parameters can be determined

to small errors the mask has been well fit and the summed pixels in a small radius around each theoretical cell location are returned in an indexed list giving the brightness value for each cell.

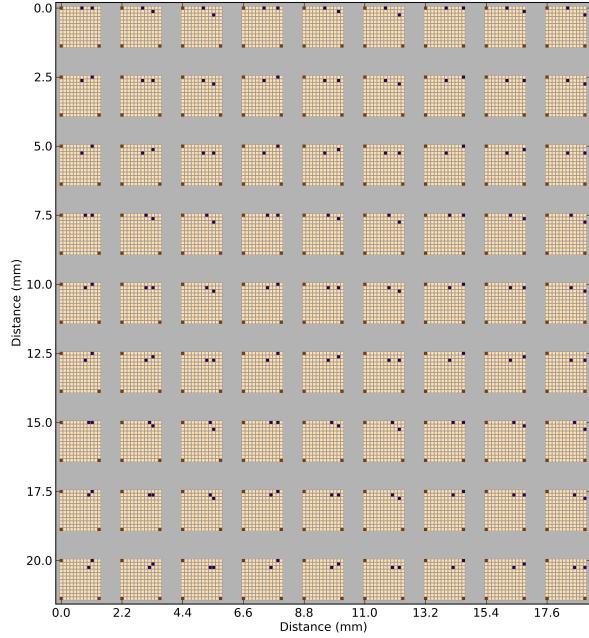


Figure 2: A schematic of the sample chip showing corner gaps in red and fiducial cells in blue, during manufacturing cells are not created here to allow identification of city blocks.

3.3 Rotation and Pixel Distance Ratio Extraction

Unfortunately finding a good mask fit is challenging as simultaneously fitting 4 parameters is computationally taxing and filled with local optimums and all attempts at pure brute forcing a solution ended up far too slow and ineffective to be useful.

Instead, a method was developed to extract the angle of rotation and pixel to real size scaling from features of the image. To do this the locations of the "city block" collections of cells as shown in the figure –x– were identified in a multi-stage computer vision algorithm. Firstly the image was converted to black

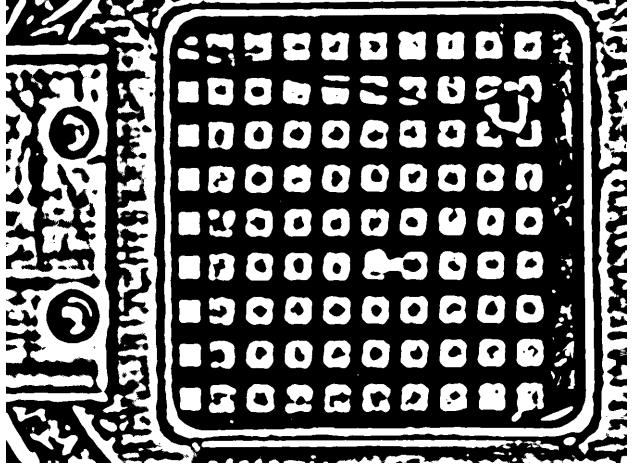


Figure 3: This image shows what the computer "sees" after the adaptive threshold has been taken, rectangles are fitted to the shapes derived in this step.

and white then a Gauss blur was applied on the scale of the cells making up the city blocks so that the interior of the city blocks were close to homogeneous. An adaptive threshold binarization (using equation 1) was applied to the image which converted each pixel into a 1 if it was a certain fraction of the local mean of pixel values and to a 0 if below, the result of this process is figure 3.

$$dst(x, y) = \begin{cases} 1 & \text{if } src(x, y) > T(x, y) \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

Where dst is the destination image, src is the source image and $T(x, y)$ is the threshold value which is the mean of the pixels in the $blocksize \times blocksize$ neighbourhood of (x, y) minus C , a modifier to the adaptive threshold value which can be tuned.

From this binarized image contours were calculated for each distinct object and as the city block are square, rectangles were fitted around the list of contours calculated. As the rough size of the city blocks in pixels are known, rectangles with areas too small and too large are discarded as they are unlikely to represent a clean city block identification. Also, rectangles with too great a ratio between their side lengths (strongly squashed or extended) were dis-

carded. This leaves a set of rectangles bounding all good rectangle determinations on the photo as shown in figure 4.

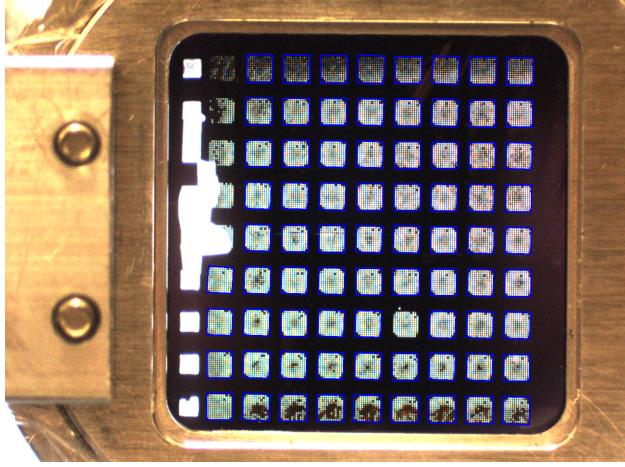


Figure 4: A sample chip example image with rectangles drawn around city blocks which have been confirmed by the feature extraction code.

From this collection of solved city block rectangles, central coordinates for each rectangle were determined. Each central coordinate point was compared to every other central point and an array was created which contained the 4 closest neighbours to each pair of coordinates, discarding values outside of a reasonable range of possible values for the close neighbours (horizontal) and far neighbours (vertical). This corresponds to the 4 closest city blocks, as their physical distance separation is known from the theoretical chip layout. To derive the best value of the pixel to distance ratio statistics of the neighbour population in the vertical and horizontal were taken. These yielded a mean and standard deviation which along with the population size gives the standard error, to get the best values possible the thresholding and feature recognition code was run in a loop with varying threshold parameters which caused different populations of city blocks to be identified.

Some thresholding passes dug deeper into the noise in the image and were able to identify more city blocks definitively but these were also more prone to

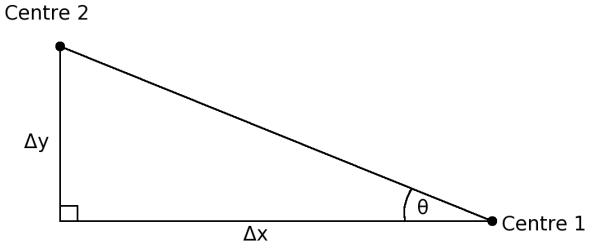


Figure 5: Diagram showing two adjacent proven rectangle centres next to each other and how the overall orientation θ is obtained.

giving slightly distorted values for the central position, this compared to the more conservative threshold passes which identified fewer city blocks but with a lower degree of error. The optimum thresholding values varied on each image depending on the light levels and the amount of excess protein floating around between city blocks which could make it harder to separate or "watershed" them. After each pass statistics for the pixel to distance ratio was recorded and the threshold pass which yielded the lowest standard error was taken.

A similar process was taken when calculating the rotation of the image, the vectors connecting city block centres to their neighbours had their angles calculated with simple trigonometry as shown in figure 5. The thresholding was evaluated to take into account both the rotation and pixel to distance ratio and simultaneously minimising the standard error in both.

After this information had been extracted from a photo the theoretical mask was overlaid over the photo and a range of x and y offset values were tested to find a tight match. The attempted fits were evaluated by trying to find the mask position that when convolved with the original image yielded the highest value.

3.4 Spurious Emission Removal

A problem encountered when cycling over x and y offset values was that the edges of the chip holder and gaps in the chip were illuminated more brightly

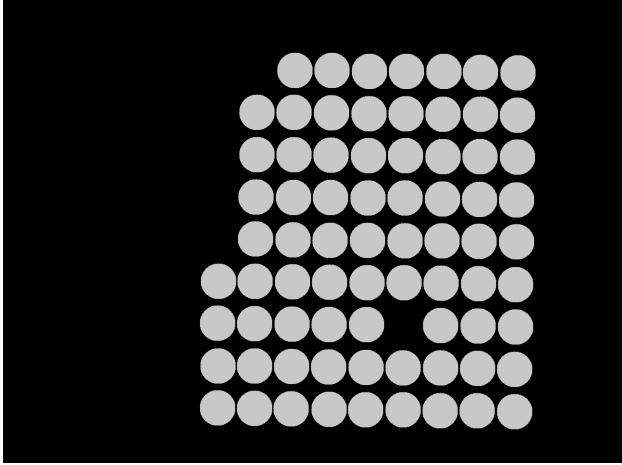


Figure 6: White areas are from around confirmed rectangles and will not have saturated pixels removed within them.

than the city blocks of cells that the fitting program was attempting a match with. Several approaches were used to tackle this problem.

Firstly all maxed out pixels (value of 255) were set to 0 as they were almost universally from gaps in the sample chip where light was shining directly through from the backlight. However, some of the cells also showed maxed out pixel values which we did not want to erode. To counteract this a mask was created which was positive in the area around each proven city block as shown in figure 6, if a pixel was equal to 255 whilst being in the vicinity of a proven city block its value would not be set to 0 whereas anything off a city block would be removed as before.

Another problem was reflected light from the sample holder where the data was collected which was not so easy to remove as it often did not saturate pixel values in the camera. To deal with this a new image was derived which was the local standard deviation of the original image. The rationale behind this process was that the emission from the sample chip holder had very small amounts of local variation in brightness, thus the local standard deviation of this area would be very low compared to on city blocks where the variation varies very strongly locally moving on

and off cells in close proximity to each other.

The local standard deviation map was calculated by evaluating a pixel in the new image as the value of the standard deviation of pixels in the local area (5x5 box of pixels). This standard deviation map was binarized such that every pixel with a value equal to or above the 78th percentile of the overall pixel value population was set to 1 and everything else 0. The 78th percentile was optimised by extensive testing, it was found to yield the best balance of signal and noise.

The standard deviation map was effective at picking out city blocks compared to reflected light from flat areas of the sample holder, however, it also picked up significant amounts of noise from free floating pieces of loaded sample on the sample chip and showed large values at the transition zones between the sample chip and the sample holder. These features were fairly constant so they were possible to remove largely, contours were taken off the binarized standard deviation map and any contour containing a very small area or very large perimeter was removed. This was effective at removing the border with the sample holder (very large perimeter and low area contour) and the small flecks of excess sample (contours with a very low area). An example of how effective this contour cleanup was can be seen in figures 7 and 8.

3.5 Sweep Generations

The strategy when searching in the x,y translation space was to fit chip mask so that each cell in the model mask is aligned to the cells in the image in addition to having each city block matched. Thus sweeps were set up to sweep at a small fraction of each of these space scales. First, the cells were aligned, then the city blocks and finally the mask was moved by the distance between the city blocks. During this stage instead of masking the original image, the mask was convolved with the binarized standard deviation maps as the fitting routine had a tendency to move the mask towards bright parts of the sample holder. However when iterating over an entire city block some misalignment's tended to be introduced so that further fine fitting was required after the city block by

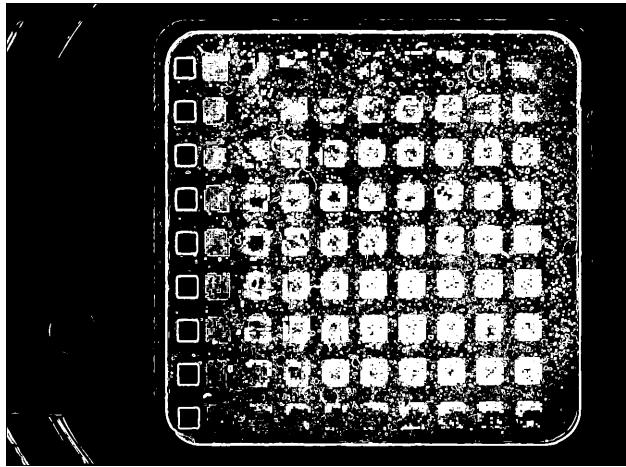


Figure 7: Standard deviation map derived from original image.

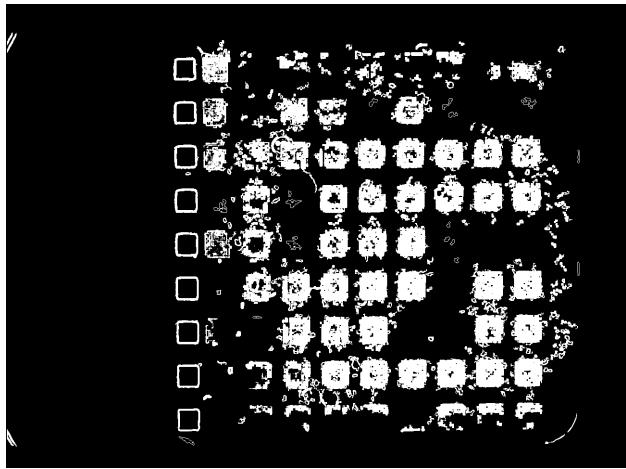


Figure 8: The same standard deviation map as figure 6 after spurious objects have been cleaned.

city block fitting on subcell dimensions.

3.6 Read Out

When the final mask fit parameters had been determined a routine went through cell by cell summing the values of the pixels in a small radius around each defined cell location from the chip mask. This yields a list of values in order of cell id with the integrated pixel values.

4 Results

4.1 Fitting

The final fitting algorithm was highly effective at matching sample chips even for badly lit samples and those with large amounts of excess sample. Time limitations meant that only chips within around 40 degrees rotation of the chip mask as my fitting program would have required significant reconstruction to be able to handle these situations. As some chips were loaded at 90 and 180-degree rotations some data could not be processed.

(fit_plot.py) Code linking pipeline elements together when comparing chips before and after loading to see relative changes in light transmission per cell. This is setup for imaging without polarising filters

(polar_fit_plot.py) Code linking pipeline for chips imaged with the polarising filter setup, fitting is performed on chips where the background light is allowed through so the mask can be correctly located. The mask fit parameters are saved and applied to the frame where the background light has been largely removed by the polarising filter for readout purposes.

An example of a well-fitted mask after read out is shown in figure 9 and how it compares to the original image in figure 10. Note that the mask has managed to determine effectively the rotation of the sample chip in the original image in addition to identifying and removing spurious emission from light coming through damaged parts of the chip.

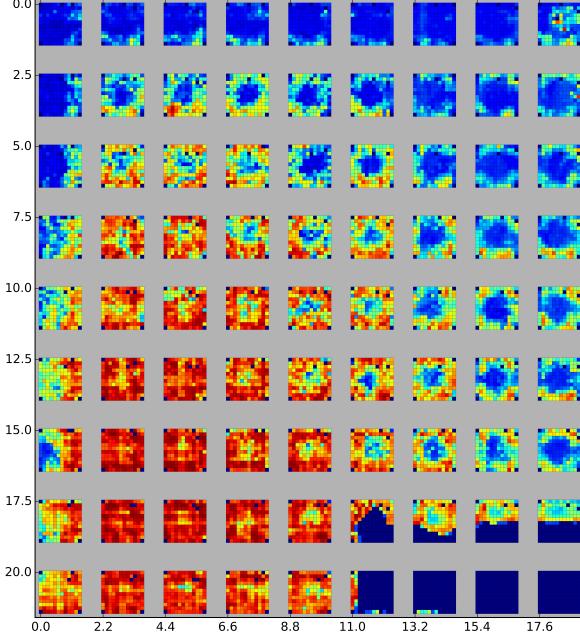


Figure 9: The read out of summed local pixel values around each fitted cell position, chip called Zurich.

4.2 Comparison with Diffraction data

4.2.1 Non Polarised Imaging

Shown in figures 11 and 12 are X-Ray diffraction data showing good crystal candidates and the extracted per cell brightness computed using my fitting program respectively.

4.2.2 Polarised Imaging

Figures 13 and 14 show examples of images taken with a polarising filter and the good crystal candidates from diffraction data on the same sample chip.

5 Conclusions

This project has demonstrated the feasibility of fitting and reading out brightness information from photographs of sample chips being tested for use in serial crystallography on the I24 beamline at Diamond Light Source and the XFEL facility at

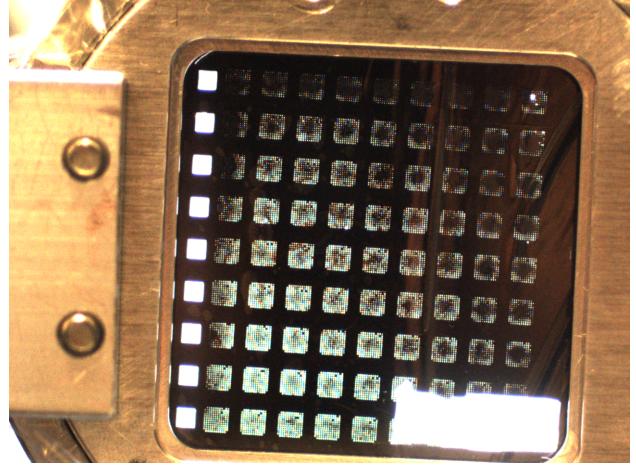


Figure 10: Original sample chip image mask was fitted to, Zurich.

SACLA in Japan. Correlations between brightness and proven crystal locations from X-Ray diffraction data were not clearly observed for images taken with a Mako G camera and a simple imaging setup but images taken with optical microscopes showed much greater promise. Time constraints meant that microscope imaging could not be well explored as it required a new imaging setup and a process to stitch multiple images together as the field of view of the microscope was too small to image the entire sample chip at once. This would be worth further attention in the future as it offered a far higher level of image quality and spatial resolution.

In the future trying to find another type of transparent film that had less of an effect on the polarisation of the background light than Mylar whilst still holding in the protein sample on the sample chip would be useful. The effect of Mylar added significant noise to the images taken with polarising filters.

Whilst my fitting tool was effective at angles of orientation between 30 and -30 degrees for sample chips it was not built to handle orientations outside of this. To make the system more flexible in the future a feature could be added that would attempt to apply the fitting mask at 90, 180 and 270-degree rotations to attempt to correctly handle extreme rotations. This

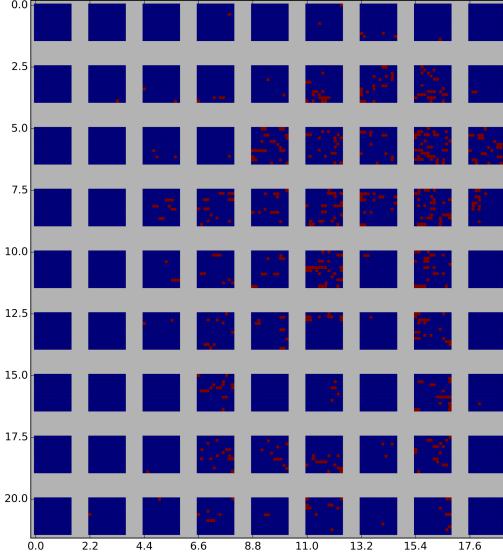


Figure 11: Good crystal candidates determined from X-Ray diffraction data taken for each sample chip cell, sample chip name Yamoto.

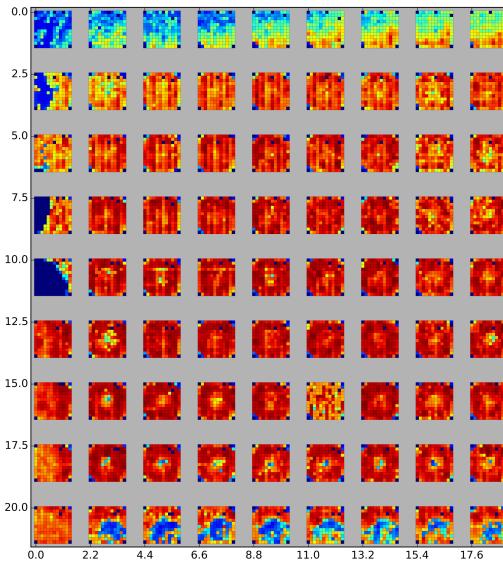


Figure 12: The plotted readout values from Yamoto.

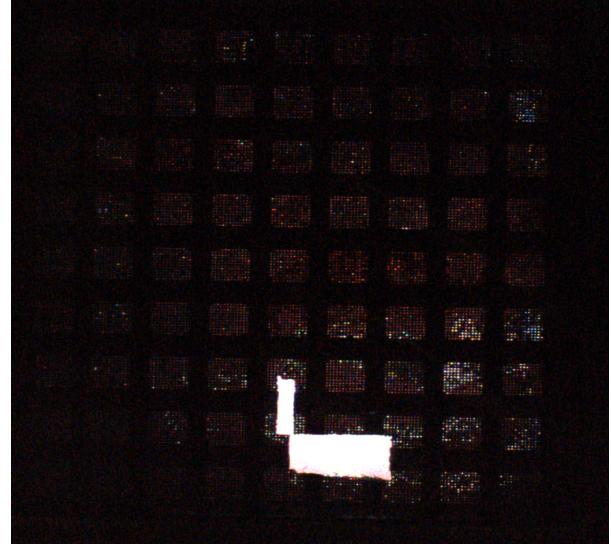


Figure 13: An optical image taken with a polarising filter in front of the camera and with purely polarised back lighting of the sample chip Dallas.

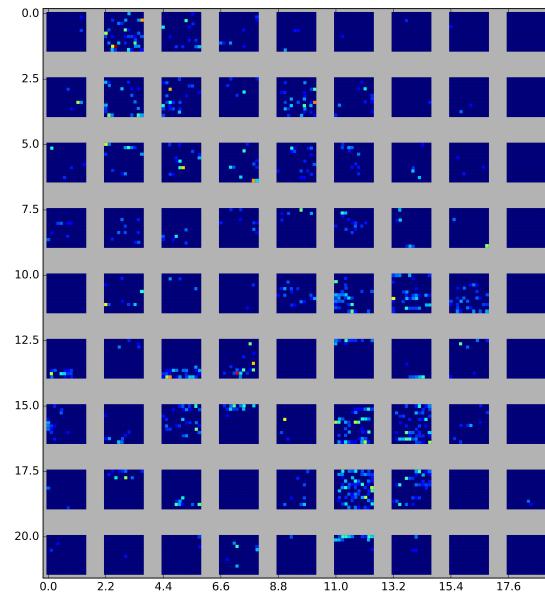


Figure 14: Good crystal candidates on the sample chip Dallas from diffraction data.

was not initially a development priority as most of the data initially collected only had modest rotations from 0 degrees.

References

- [1] Henry N. Chapman, Carl Caleman, and Nicusor Timneanu, Diffraction before destruction. *Philosophical Transaction of the Royal society B* (2014)
- [2] Mark S. Hunter, Brent Segelke, Marc Messer-schmidt, Garth J. Williams, Nadia A. Zatsepin, Anton Barty, W. Henry Benner, David B. Carlson, Matthew Coleman, Alexander Graf, Stefan P. Hau-Riege, Tommaso Pardini, M. Marvin Seibert, James Evans, Sébastien Boutet , and Matthias Frank, Fixed-target protein serial microcrystallography with an x-ray free electron laser. *Nature Scientific Reports* (2014)
- [3] Oghbaey S1, Sarracini A1, Ginn HM2, Pare-Labrosse O1, Kuo A3, Marx A4, Epp SW4, Sherrell DA5, Eger BT3, Zhong Y4, Loch R4, Mariani V6, Alonso-Mori R7, Nelson S7, Lemke HT7, Owen RL5, Pearson AR8, Stuart DI2, Ernst OP3, Mueller-Werkmeister HM1, Miller RJ1, Fixed target combined with spectral mapping: approaching 100% hit rates for serial crystallography. *Acta Crystallographica Section D* (2016)

6 Appendix

6.1 Code Guide

Here follows a brief run down of the purpose of the different scripts written as part of this project. Most are in Python however for some tasks Bash was more appropriate.

6.1.1 `reduc`

Main analysis code which does the mask creation and fitting operations, includes a deprecated top section (commented out) which employs an alternative method to determine the orientation of the sample

tray using 2d Fourier transforms. The top level function is `meta_sweep()` which pulls together the other lower level functions to execute the multiple sweep generations and call the scripts which create all the image derivatives necessary to get a good fit.

6.1.2 `watershed`

Named after the watershed problem where a series of objects are separated from each other, this script identifies the city blocks and their central locations. From this, it derived the pixel to mm ratio of the image and determines the orientation from how the city block centres are positioned compared to their neighbours.

6.1.3 `variance_map`

Computes the local standard deviation map of the of an image highlighting features which change on a small scale and wiping out those constant on large scales. Used to counter the reflection of light from the metal edges of the sample holder which could often outshine the light coming through the sample chip from the backlight. The output of this code was used in one of the sweep generations to ensure spurious emission was not being fitted for.

6.1.4 `visual_map`

This code (Credit Darren Sherrell) marks the locations of each of the cells on the schematic mask in physical space, this was taken by `reduc` in order to generate the mask which was applied to the captured photos.

6.1.5 `spot_map`

Called from `./grid_plot.sh` this took the diffraction data information for a chip and plotted as shown in figure 11.

6.1.6 `polar_plot`

Used to plot fittings of images taken with a polarising filter (warning incomplete).

6.1.7 polar_fit_plot

Brings together all the listed core analysis scripts to fit a well lit polarising filter image and then uses that fitting information for the badly illuminated frame to maximise the chance of an effective fit.

6.1.8 intensity_plot

Plots the summed local pixel intensities per cell calculated from reduc as seen in figure 12 for sample chips before and after loading to create a difference map.

6.1.9 fit_plot

Pulls together the other lower level analysis routines to generate a difference map directly from two input photos assuming that it is the same sample chip before and after loading with crystals.

```
from PIL import Image
import numpy as np
import numpy.linalg as la
import math
import visual_map
import watershed
import variance_map
import os
from numba import jit

#####
# creates mask that blocks the centre and extremities of fourier transform
# leaving only city block scale signal
def fourier_mask(shape):

    # read dimensions of image to create mask for
    x_pix = int(shape[0])
    y_pix = int(shape[1])

    print('input image has dimensions of x = %d and y = %d') % (x_pix, y_pix)

    # set size of mask (circular shape)
    mask_inner_radius = x_pix / 13
    mask_outer_radius = x_pix / 4
    x_cent = x_pix / 2
    y_cent = y_pix / 2

    # initialising zeroed array to hold mask
    f_mask = np.zeros((x_pix, y_pix))

    # cycling over all pixels in the zeroed array
    for x in xrange(0, x_pix):
        for y in xrange(0, y_pix):
            r = math.sqrt((x - x_cent) ** 2 + (y - y_cent) ** 2)

            # setting circular region larger than mask_inner_radius and less than
            # mask_outer_radius to 1, all other regions of array left at 0
            if ((r > mask_inner_radius) & (r < mask_outer_radius)):
                f_mask[x][y] = 1

    # mask saved as image for debugging purposes, maybe remove?
    img = Image.fromarray(np.uint8(f_mask))
    img.save('./images/mask.png')
    print('mask saved to disk')

    return f_mask


def threshold_tweak(ftrans, max_peak, peaks):

    thresh_step = 0.0001

    # setting threshold values to iterate over
    thresh_iter = np.arange(0.001, 0.2, thresh_step)

    for thresh in thresh_iter:
        ftrans_temp = ftrans
        ftrans_temp[ftrans_temp < (max_peak * thresh)] = 0

        # uncomment to make function verbose
        #print('%d non zero pixels detected at threshold of %f %% of peak value') \
        % (np.count_nonzero(ftrans_temp), thresh * 100)

        if(np.count_nonzero(ftrans_temp) == peaks):
            print('%d peaks found when the threshhold = %f %% of the max peak \
                  intensity') % (peaks, thresh * 100)
            return ftrans_temp

    if(np.count_nonzero(ftrans_temp) < peaks):
        print('threshold iteration has skipped over %d peak values, try again \
              with a finer threshold step') % peaks

    return 0

    print('no good threshold found, sorry...')
```

```
return 0

# extracts the angle of inclination of the chip from a filtered 2dfft
def find_angle(clean_fft, peaks):
    args = np.zeros((peaks, 2))

    # fills array args up with the indicies of non zero pixels
    for peak in xrange(0, peaks):
        args[peak] = np.unravel_index(np.argmax(clean_fft), np.shape(clean_fft))

        print('peak %d has intensity of %f') % (peak + 1, np.amax(clean_fft))
        print('and position of [%d, %d]') % (args[peak][0], args[peak][1])

    clean_fft[int(args[peak][0])][int(args[peak][1])] = 0

    # calculate vector between two identified pixels
    vector = np.zeros(2)
    vector[0] = args[0][0] - args[1][0]
    vector[1] = args[0][1] - args[1][1]

    #finding magnitude of vector
    vec_mag = math.sqrt(vector[0] ** 2 + vector[1] ** 2)

    print('vec_mag = %f') % vec_mag

    # setting reference vertical vector
    vert_vect = np.array([0, 1])

    #computing angle between calculated and reference vector
    angle = vector_angle(vector, vert_vect)

    deg_angle = angle * 360 / (2 * math.pi)
    print('angle = %f degrees before quadrant correction') % deg_angle

    # finding quadrant in which the calculated angle is closest to reference values of
    # 0, pi/2, pi and (3 * pi) / 4
    quadrant_angles = np.array([0.0, math.pi / 2.0, math.pi, -math.pi / 2.0])
    quadrant_delta = np.array([0.0, 0.0, 0.0, 0.0])

    for quadrant in xrange(0,4):
        quadrant_delta[quadrant] = angle - quadrant_angles[quadrant]

    quadrant = int(np.argmin(np.fabs(quadrant_delta)))

    angle = quadrant_delta[quadrant]

    deg_angle = angle * 360 / (2 * math.pi)
    print('chip is rotated %f degrees counter clockwise') % deg_angle

    return angle, vec_mag

# returns the angle in radians between vectors v1 and v2
def vector_angle(v1, v2):
    cosang = np.dot(v1, v2)
    sinang = la.norm(np.cross(v1, v2))

    return np.arctan2(sinang, cosang)

# attempts to determine the orientation of a chip image
# (clockwise rotation in radians)
def orient(filename):

    img_array = np.asarray(Image.open(filename).convert('L'))
    print('image loaded')

    shape = np.shape(img_array)

    # performing fourier transform
    ftrans = np.fft.fft2(img_array)

    # gets mask for fourier transform
    f_mask = fourier_mask(shape)
```

```

# sets peak intensity to be at the centre of the image
ftrans = np.fft.fftshift(ftrans)

# determine peak of fourier transform
max_peak = np.max(np.abs(ftrans))

# convolve mask with fourier data
masked_ftrans = ftrans * f_mask

# image of mask loaded into image and saved
img = Image.fromarray(np.uint8(masked_ftrans))
img.save('./images/masked_ftrans.png')
print('masked ftrans saved')

# number of peaks that the threshold will be tweaked to find (2 by default),
# different angle determination method required
# with more than 2 peaks
peaks = 2

masked_ftrans = threshold_tweak(masked_ftrans, max_peak, peaks)

# log scale data
abs_data = 1 + np.abs(masked_ftrans)
c = 255.0 / np.log(1 + max_peak)
log_data = c * np.log(abs_data)

# array loaded into image and saved
img = Image.fromarray(np.uint8(log_data))
img.save('./images/orient.png')
print('image saved to disk')

theta, vec_mag = find_angle(log_data, peaks)

return theta, vec_mag
"""

@jit(nopython=True)
def rotate(x, y, rot_matrix):
    col_vec = np.zeros((2))
    col_vec[0] = x
    col_vec[1] = y
    col_vec = np.dot(rot_matrix, col_vec)

    return col_vec[0], col_vec[1]

# generates a mask from the theoretical layout of a chip
def chip_mask_gen(x_pix_max, y_pix_max, cell_real_size,
                   x_real_offset, y_real_offset, pix_scale, theta, X, Y, Z):

    chip_mask = chip_mask_crunch(x_pix_max, y_pix_max, cell_real_size,
                                 x_real_offset, y_real_offset, pix_scale, theta, X, Y, Z)

    chip_mask = np.fliplr(chip_mask)
    chip_mask = np.rot90(chip_mask, 3)

    return chip_mask

@jit(nopython=True)
def chip_mask_crunch(x_pix_max, y_pix_max, cell_real_size,
                      x_real_offset, y_real_offset, pix_scale, theta, X, Y, Z):

    # final cell index
    i_max = 11664

    i_half = 5832

    # initializing mask array
    mask = np.zeros((x_pix_max, y_pix_max))

    cell_pix_size_float = cell_real_size * pix_scale # width of cell in pixels
    cell_pix_size_int = int(math.ceil(cell_pix_size_float))

    x_pix = np.zeros(i_max)

```

```

Y_pix = np.zeros(i_max)

# location of cells in terms of image pixels
for i in xrange(0,11664):

    X_pix[i] = round((X[i] + x_real_offset) * pix_scale)
    Y_pix[i] = round((Y[i] + y_real_offset) * pix_scale)

x_mid = X_pix[i_half]
y_mid = Y_pix[i_half]

rot_matrix = np.zeros((2, 2))
rot_matrix[0][0] = math.cos(theta)
rot_matrix[1][1] = math.cos(theta)
rot_matrix[1][0] = - math.sin(theta)
rot_matrix[0][1] = math.sin(theta)

# accounting for x, y shift of the middle of mask so rotation only takes
# place around centre of chip
x, y = rotate(x_mid, y_mid, rot_matrix)
delt_x = x - x_mid
delt_y = y - y_mid

for i in xrange(0, i_max):
    for r_x in xrange(-cell_pix_size_int, cell_pix_size_int):
        for r_y in xrange(-cell_pix_size_int, cell_pix_size_int):

            if (math.sqrt(r_x ** 2 + r_y ** 2) <= cell_pix_size_float):

                x = int(round(X_pix[i] + r_x)) # temporary x
                y = int(round(Y_pix[i] + r_y)) # temporary y

                x, y = rotate(x, y, rot_matrix)

# correcting for non centred rotation
                x = int(round(x - delt_x))
                y = int(round(y - delt_y))

            if (x >= 0 and x < x_pix_max and y >= 0 and y < y_pix_max):
                mask[int(x)][int(y)] = Z[int(i)]

return mask

@jit(nopython=True)
def rect_cent_mask_gen (x_pix_max, y_pix_max, pix_scale, real_circ_rad, good_rect_log, cent_cords):

    pix_circ_rad = int(round(real_circ_rad * pix_scale))

    mask = np.zeros((x_pix_max, y_pix_max))

    length = good_rect_log.size

    for count in xrange(1, length):
        for x in xrange(-pix_circ_rad, pix_circ_rad):
            for y in xrange(-pix_circ_rad, pix_circ_rad):

# only masking for rectangles with adjacent neighbours (horizontal and vertical)
                if (good_rect_log[count] == 1):
                    if (math.sqrt(x ** 2 + y ** 2) <= pix_circ_rad):

                        if ((cent_cords[count][0] + x >= 0) and (cent_cords[count][0] + x < x_pix_max) \
                            and (cent_cords[count][1] + y >= 0) and (cent_cords[count][1] + y < y_pix_max)):

                            mask[cent_cords[count][0] + x][cent_cords[count][1] + y] = 1.0

    return mask

# returns the index of an iterable given its value
@jit(nopython=True)
def index (value, min, step):
    I = int(round((value - min) / step) - 1)

    return I

```

```

# returns the value of an iterable for a given index
@jit(nopython=True)
def de_index (I, min, step):
    value = ((I + 1) * step) + min

    return value

@jit(nopython=True)
def smart_img_clean (rect_mask, img_array, x_pix_max, y_pix_max):

    for x in xrange(0, y_pix_max):
        for y in xrange(0, x_pix_max):
            if ((rect_mask[x][y] <= 0.5) & (img_array[x][y] >= 255)):
                img_array[x][y] = 0

    return img_array

# Search mask generation parameter space to find optimal fitting
@jit
def sweep(filename, x_r_off_min, x_r_off_max, y_r_off_min, y_r_off_max, \
real_trans_step_x, real_trans_step_y, cell_real_size, \
X, Y, Z, cent_cords, good_rect_log, img_array, rect_mask,
pix_scale, theta, std_dev_map, sweep_type=0):

    print ('in sweep' )

# stops searching of space outside of this zone
x_hard_min = 8.0
x_hard_max = 16
y_hard_min = 0
y_hard_max = 5.2

# size of image
#x_pix_max = 1292
#y_pix_max = 964

print ('before shape' )

img_shape = np.shape(img_array)

x_pix_max = img_shape[0]
y_pix_max = img_shape[1]

print ('after shape' )

real_circ_rad = 1.05 # radius of circle around identified rectangles to be masked (mm)

# values to iterate x_real_offset and y_real_offset over
x_r_off_iter = np.arange(x_r_off_min, x_r_off_max, real_trans_step_x)
y_r_off_iter = np.arange(y_r_off_min, y_r_off_max, real_trans_step_y)

print ('iterables created' )

# calculating number of solutions
ind_x_max = 1 + index(x_r_off_max, x_r_off_min, real_trans_step_x)
ind_y_max = 1 + index(y_r_off_max, y_r_off_min, real_trans_step_y)

print ('calculating array dimension sizes' )

sums = np.zeros((ind_x_max, ind_y_max))
sums_omni = np.zeros((ind_x_max, ind_y_max))
conv_std_dev_img_array = np.zeros((x_pix_max, y_pix_max))

print ('about to start loop' )

# iterating over x and y offset values for chip mask
for x_real_offset in x_r_off_iter:
    for y_real_offset in y_r_off_iter:

        print ('x offset = %f' % x_real_offset

```

```

print ('y offset = %f'      ) % y_real_offset

sum_current = 0
sum_current_rect = 0
non_zero_pixels = 0
sum_current_std_dev = 0

if ((x_real_offset > x_hard_min) & (x_real_offset < x_hard_max) & \
    (y_real_offset > y_hard_min) & (y_real_offset < y_hard_max)):

    # creating mask of chip
    chip_mask = chip_mask_gen(y_pix_max, x_pix_max, cell_real_size, x_real_offset, \
        y_real_offset, pix_scale, theta, X, Y, Z)

    # convolving chip mask with image
    chip_conv_img_array = img_array * chip_mask
    sum_current = np.sum(chip_conv_img_array)

    # sweep using a larger cell size and against the good rect mask to
    # ensure a good city block match
    if (sweep_type == 1):

        # convolving chip mask convolved image with known city block mask
        conv_rect_img_array = chip_conv_img_array * rect_mask

        non_zero_pixels = np.count_nonzero(conv_rect_img_array)
        print ('%d non zero pixels'      ) % non_zero_pixels

        # loading array into image and saving
        #img = Image.fromarray(np.uint8(conv_rect_img_array))
        #imname = ('./images/%f_%f.png') % (round(x_real_offset, 4), round(y_real_offset, 4))
        #img.save(imname)

        sum_current_rect = np.sum(conv_rect_img_array)

        # convolving std dev map with chip mask
        conv_std_dev_img_array = std_dev_map * chip_mask

        # convert to non zero pix count!
        sum_current_std_dev = np.count_nonzero(conv_std_dev_img_array)

        print ('%d non zero std dev pixels'      ) % sum_current_std_dev

    ind_x = index(x_real_offset, x_r_off_min, real_trans_step_x)
    ind_y = index(y_real_offset, y_r_off_min, real_trans_step_y)

    sums[ind_x, ind_y] = sum_current

    # edit for different weighting!
    #sums_omni[ind_x, ind_y] = non_zero_pixels + (1.0 / 15000.0) * sum_current + (1.0 / 200) * sum_cu
    current_std_dev
    sums_omni[ind_x, ind_y] = non_zero_pixels + sum_current_std_dev

    print ('sum current = %f'      ) % sums[ind_x, ind_y]
    print ('sum_omni = %f'      ) % sums_omni[ind_x, ind_y]

if (sweep_type == 1):

    # returns the indices of the sums element with the highest value
    i, j = np.unravel_index(sums_omni.argmax(), sums_omni.shape)
    print (i, j)
    x_real_offset = de_index(i, x_r_off_min, real_trans_step_x)
    y_real_offset = de_index(j, y_r_off_min, real_trans_step_y)

    print ('%f pixels per mm, x offset of %f mm and y offset of %f mm'      ) % \
        (pix_scale, x_real_offset, y_real_offset)

    # calculating and displaying the convolution of the best fit mask with the input image
    chip_mask = chip_mask_gen(y_pix_max, x_pix_max, cell_real_size, x_real_offset, \
        y_real_offset, pix_scale, theta, X, Y, Z)

    img_array = img_array * chip_mask * rect_mask

    # loading array into image and saving
    img = Image.fromarray(np.uint8(img_array))
    img.save('./images/type1_fit.png')

```

```

elif (sweep_type == 0):
    print ('sweep_type = %d' % sweep_type)
    print ('determining best fit from sweep type 0')

    # returns the indices of the sums element with the highest value
    i, j = np.unravel_index(sums.argmax(), sums.shape)

    x_real_offset = de_index(i, x_r_off_min, real_trans_step_x)
    y_real_offset = de_index(j, y_r_off_min, real_trans_step_y)

    print ('%f pixels per mm, x offset of %f mm and y offset of %f mm'
           ) % \
        (pix_scale, x_real_offset, y_real_offset)

    # calculating and displaying the convolution of the best fit mask with the input image

    chip_mask = chip_mask_gen(y_pix_max, x_pix_max, cell_real_size, x_real_offset, \
                               y_real_offset, pix_scale, theta, X, Y, Z)

    img_array = img_array * chip_mask

    img = Image.fromarray(np.uint8(img_array))
    img.save('./images/type0_fit.png')
    print ('image saved')

# returns best fit mask generation parameters and image data
return img_array, pix_scale, x_real_offset, y_real_offset, theta

#
def meta_sweep(filename, filename_out):
    # opening image and converting to greyscale
    img_array = np.asarray(Image.open(filename).convert('L'))
    img_array.flags.writeable = True # making array readable

    X, Y, Z = visual_map.main() # get real space position of cells on chip
    Z[Z == 2] = 0
    Z[Z == 7] = 0
    Z[Z == 4] = 1

    # get fitting information from feature recognition code
    # format of angle_mean, angle_std_err, pix_scale_mean, pix_scale_std_err, count
    # getting fit which gives lowest pixel scale and angle error
    data, cent_cords_null, good_rect_log_null = watershed.error_minimize(filename, 0)

    # getting fit which yields highest number of rectangles
    data_null, cent_cords, good_rect_log = watershed.error_minimize(filename, 1)

    theta = - math.radians(data[0])
    pix_scale = data[2]

    print ('theta = %f rads' % theta)
    print ('pix scale = %f pixels per mm' % pix_scale)

    std_dev_box_size = 2
    std_dev_map = variance_map.main(filename, std_dev_box_size)

    img_shape = np.shape(img_array)

    x_pix_max = img_shape[1]
    y_pix_max = img_shape[0]

    real_circ_rad = 1.05 # radius of circle around identified rectangles to be masked (mm)
    rect_mask = rect_cent_mask_gen(x_pix_max, y_pix_max, pix_scale, real_circ_rad, good_rect_log, cent_cords)

    # correcting for orientation
    rect_mask = np.flipud(rect_mask)
    rect_mask = np.rot90(rect_mask, 3)

    # loading array into image and saving
    rect_mask_img = Image.fromarray(np.uint8(rect_mask * 200))
    rect_mask_img.save('./images/rect_mask.png')

```

```

img_array_perm = smart_img_clean(rect_mask, img_array, x_pix_max, y_pix_max)

# loading array into image and saving
smart_clean_img = Image.fromarray(np.uint8(img_array_perm))
smart_clean_img.save('./images/smart_clean.png')

# default cell radius (mm)
cell_real_size = 0.03

cell_real_size_city_block = 0.08

sweep_num_max = 8

# setting x and y offset iteration parameters (mm)
x_r_off_min = 10.5
x_r_off_max = 13

y_r_off_min = 1.5
y_r_off_max = 4

# loading sweep pattern
sweep_data = sweep_pattern(sweep_num_max)

for sweep_num in xrange(1, sweep_num_max):
    sweep_type = 0
    print ('sweep number = %d' % sweep_num)
    cell_size = cell_real_size

    if (sweep_num == 5):
        print ('sweep type of 1')
        sweep_type = 1
        cell_size = cell_real_size_city_block

    print (x_r_off_min, x_r_off_max, y_r_off_min, y_r_off_max, \
           sweep_data[sweep_num - 1][1], sweep_data[sweep_num - 1][2])

    img_array = img_array_perm

    img_array, pix_scale, x_real_offset, y_real_offset, theta = \
        sweep(filename, x_r_off_min, x_r_off_max, y_r_off_min, y_r_off_max, \
              sweep_data[sweep_num - 1][1], sweep_data[sweep_num - 1][2], cell_real_size, \
              X, Y, Z, cent_cords, good_rect_log, img_array, rect_mask,
              pix_scale, theta, std_dev_map, sweep_type)

    print ('after sweep of sweep_num %d of sweep_type %d' % (sweep_num, sweep_type))

    if (sweep_num <= sweep_num_max - 2):
        x_r_off_min = x_real_offset - sweep_data[sweep_num][0] * sweep_data[sweep_num][1]
        x_r_off_max = x_real_offset + sweep_data[sweep_num][0] * sweep_data[sweep_num][1]

        y_r_off_min = y_real_offset - sweep_data[sweep_num][0] * sweep_data[sweep_num][2]
        y_r_off_max = y_real_offset + sweep_data[sweep_num][0] * sweep_data[sweep_num][2]

fit_params = (img_array_perm, cell_real_size, x_real_offset, y_real_offset, \
              pix_scale, theta, X, Y, Z)

i_list = read_out(img_array_perm, cell_real_size, \
                  x_real_offset, y_real_offset, pix_scale, theta, X, Y, Z)

np.savetxt(filename_out, i_list)

print ('i_list.txt saved')

return img_array, pix_scale, x_real_offset, y_real_offset, theta

@jit
def sweep_pattern (sweep_num_max):

# translational offset iteration step (mm), 0.125 is spacing between cells
real_trans_step_1 = 0.25

# size of search area in second sweep

```

```

steps_2 = 5
real_trans_step_2 = 0.0125

# size of search area in third sweep
steps_3 = 6
real_trans_step_3 = 0.125

# size of search area in fourth sweep
steps_4 = 6
real_trans_step_4 = 0.125

# size of search area in fifth sweep by city block
steps_5 = 3
real_trans_step_x_5 = 2.2
real_trans_step_y_5 = 2.5

# size of search area in sixth sweep
steps_6 = 5
real_trans_step_x_6 = 0.125
real_trans_step_y_6 = 0.125

# size of search area in seventh sweep
steps_7 = 5
real_trans_step_x_7 = 0.0125
real_trans_step_y_7 = 0.0125

# stores the steps, real_trans_step_x, real_trans_step_y
sweep_data = np.zeros((sweep_num_max - 1, 3))

sweep_data = [[0, real_trans_step_1, real_trans_step_1], [steps_2, \
    real_trans_step_2, real_trans_step_2], [steps_3, real_trans_step_3, \
    real_trans_step_3], [steps_4, real_trans_step_4, \
    real_trans_step_4], [steps_5, real_trans_step_x_5, real_trans_step_y_5], \
    [steps_6, real_trans_step_x_6, real_trans_step_y_6], \
    [steps_7, real_trans_step_x_7, real_trans_step_y_7]]


return sweep_data

!!!!!! sometimes segfaults!
# reads out summed values for for each cell
def read_out (img_array, cell_real_size, \
    x_real_offset, y_real_offset, pix_scale, theta, X, Y, Z):

# putting image array into correct frame
img_array = np.rot90(img_array, 1) # maybe
img_array = np.flipud(img_array)

i_list = read_out_crunch(img_array, cell_real_size, \
    x_real_offset, y_real_offset, pix_scale, theta, X, Y, Z)

return i_list

@@jit(nopython=True)
def read_out_crunch (img_array, cell_real_size, \
    x_real_offset, y_real_offset, pix_scale, theta, X, Y, Z):

# final cell index
i_max = 11664
i_half = 5832

x_pix_max = 1292
y_pix_max = 964

# [cell number][total intensity, number of pixels for cell]
i_list = np.zeros((i_max, 2))

cell_pix_size_float = cell_real_size * pix_scale # width of cell in pixels
cell_pix_size_int = int(math.ceil(cell_pix_size_float))

X_pix = np.zeros(i_max)
Y_pix = np.zeros(i_max)

# location of cells in terms of image pixels
for i in xrange(0,11664):

```

```
x_pix[i] = round((X[i] + x_real_offset) * pix_scale)
Y_pix[i] = round((Y[i] + y_real_offset) * pix_scale)

x_mid = X_pix[i_half]
y_mid = Y_pix[i_half]

rot_matrix = np.zeros((2, 2))
rot_matrix[0][0] = math.cos(theta)
rot_matrix[1][1] = math.cos(theta)
rot_matrix[1][0] = -math.sin(theta)
rot_matrix[0][1] = math.sin(theta)

# accounting for x, y shift of the middle of mask so rotation only takes
# place around centre of chip
x, y = rotate(x_mid, y_mid, rot_matrix)
delt_x = x - x_mid
delt_y = y - y_mid

for i in xrange(0, i_max):
    for r_x in xrange(-cell_pix_size_int, cell_pix_size_int):
        for r_y in xrange(-cell_pix_size_int, cell_pix_size_int):

            if (math.sqrt(r_x ** 2 + r_y ** 2) <= cell_pix_size_float):

                x = int(round(X_pix[i] + r_x)) # temporary x
                y = int(round(Y_pix[i] + r_y)) # temporary y

                x, y = rotate(x, y, rot_matrix)

                # correcting for non centred rotation
                x = int(round(x - delt_x))
                y = int(round(y - delt_y))

                if (x >= 0 and x < x_pix_max and y >= 0 and y < y_pix_max):
                    i_list[int(i)][0] = i_list[int(i)][0] + img_array[int(x)][int(y)]
                    i_list[int(i)][1] += 1

            else :
                b = 2 # placeholder

return i_list
```

```

import numpy as np
import cv2
from PIL import Image
import math
import matplotlib.mlab as mlab
import matplotlib.pyplot as plt
import random
from numba import jit

@jit(#(nopython=True)
def find_neighbours (cent_cords, count):

    # calculating the distance from every good rect centre to every other
    # rect centre in x, y and vector magnitude
    p_to_p_dist = np.zeros((count, count, 3))

    # recording the closest to fourth closest neighbours delt_x and delt_y
    # and vector mag values
    closest_neighbours = np.zeros((count, 4, 3))

    closest_neighbours = closest_neighbours + 3000

    # looping over start points
    for s_p in xrange(0, count):
        # looping over end points
        for e_p in xrange(0, count):

            if (s_p != e_p):
                p_to_p_dist[s_p][e_p][0] = cent_cords[s_p][0] - cent_cords[e_p][0]
                p_to_p_dist[s_p][e_p][1] = cent_cords[s_p][1] - cent_cords[e_p][1]
                p_to_p_dist[s_p][e_p][2] = math.sqrt((p_to_p_dist[s_p][e_p][0] ** 2) \
                    + (p_to_p_dist[s_p][e_p][1] ** 2))

            if (closest_neighbours[s_p][3][2] > p_to_p_dist[s_p][e_p][2]):

                if (closest_neighbours[s_p][0][2] > p_to_p_dist[s_p][e_p][2]):
                    closest_neighbours[s_p][0][0] = p_to_p_dist[s_p][e_p][0]
                    closest_neighbours[s_p][0][1] = p_to_p_dist[s_p][e_p][1]
                    closest_neighbours[s_p][0][2] = p_to_p_dist[s_p][e_p][2]

                elif (closest_neighbours[s_p][1][2] > p_to_p_dist[s_p][e_p][2]):
                    closest_neighbours[s_p][1][0] = p_to_p_dist[s_p][e_p][0]
                    closest_neighbours[s_p][1][1] = p_to_p_dist[s_p][e_p][1]
                    closest_neighbours[s_p][1][2] = p_to_p_dist[s_p][e_p][2]

                elif (closest_neighbours[s_p][2][2] > p_to_p_dist[s_p][e_p][2]):
                    closest_neighbours[s_p][2][0] = p_to_p_dist[s_p][e_p][0]
                    closest_neighbours[s_p][2][1] = p_to_p_dist[s_p][e_p][1]
                    closest_neighbours[s_p][2][2] = p_to_p_dist[s_p][e_p][2]

                else :
                    closest_neighbours[s_p][3][0] = p_to_p_dist[s_p][e_p][0]
                    closest_neighbours[s_p][3][1] = p_to_p_dist[s_p][e_p][1]
                    closest_neighbours[s_p][3][2] = p_to_p_dist[s_p][e_p][2]

            return closest_neighbours

def output (pix_scale_omni, angle_omni, hor_neigh_count, \
    ver_neigh_count):
    pix_scale_std_err = np.std(pix_scale_omni) / math.sqrt(pix_scale_omni.size)

    pix_scale_mean = np.mean(pix_scale_omni)

    angle_std_err = np.std(angle_omni) / math.sqrt(angle_omni.size)

    angle_mean = np.mean(angle_omni)

    return angle_mean, angle_std_err, pix_scale_mean, pix_scale_std_err

def error_minimize (filename, mode=0):
    # iterates over values of C to find the best angle and pix_scale disagreement
    max_C = 15
    length = 2 * max_C
    min_good_rects = 10

    # angle_mean, angle_std_err, pix_scale_mean, pix_scale_std_err, count, cent_cords
    # good_rect_log

```

```

data_out = np.zeros((5, length))

for C in xrange(-max_C, max_C):
    data_out[0][C + max_C], data_out[1][C + max_C], data_out[2][C + max_C], \
        data_out[3][C + max_C], data_out[4][C + max_C], cent_cords, good_rect_log = main(filename, C)

    if (np.count_nonzero(good_rect_log) < min_good_rects):
        data_out[1][C + max_C] = float('nan')
        data_out[3][C + max_C] = float('nan')

angle_std_err = data_out[1,:]
pix_scale_std_err = data_out[3,:]
rects = data_out[4,:]

ang_weight = 1 / np.nanmean(angle_std_err)
pix_weight = 1 / np.nanmean(pix_scale_std_err)

avrg_err = (angle_std_err * ang_weight + pix_scale_std_err * pix_weight) / \
            (ang_weight + pix_weight)

min_angle_err_C = np.nanargmin(angle_std_err) - max_C
min_pix_err_C = np.nanargmin(pix_scale_std_err) - max_C
min_avrg_err_c = np.nanargmin(avrg_err) - max_C
max_rects_C = np.nanargmax(rects) - max_C

if (mode == 0):
    C = min_avrg_err_c
elif (mode == 1):
    C = max_rects_C
else :
    print ('error, incorrect mode, must be either 0 or 1')
    return 0

data = data_out[:,C + max_C]

data_out[0][C + max_C], data_out[1][C + max_C], data_out[2][C + max_C], \
    data_out[3][C + max_C], data_out[4][C + max_C], cent_cords, good_rect_log = main(filename, C)

print ('best C = ')
print (min_avrg_err_c)

return data, cent_cords, good_rect_log

# when flag == '-p' image threshold and fitted rectangles will be displayed
def main(filename, C, flag=-n) :

    # thresholding parameters, iterate over C perhaps, strong effect on angle disagreement
    #C = -10 # background subtraction when thresholding, 2 recommended value
    kern_dims = 21 # size of gaussian blurring kernel, default 21
    adapt_box_size = 40 # size of adaptive threshold box, good value of 40

    # rectangle sifting parameters
    min_area = 2500.0 # 2500 - 2700 default
    max_area = 3700.0 # 3700 default
    max_side_ratio = 1.1 # max accepted value of longer side divided by shorter side

    # spacing of city blocks in mm
    horizontal_spacing = 2.2
    vertical_spacing = 2.5

    # creating random colours for circles representing rectangle points
    R = random.random()
    G = random.random()
    B = random.random()

    #print('R = %f, G = %f, B = %f') % (R, G, B)

    input_filename = filename

    pil_image = Image.open(input_filename).convert('RGB')

    open_cv_image = np.array(pil_image)

    gray = cv2.cvtColor(open_cv_image, cv2.COLOR_BGR2GRAY)

```

```
# gaussian blur input image
gray = cv2.GaussianBlur(gray, (kern_dims, kern_dims), 0)

if ((adapt_box_size % 2) == 0):
    adapt_box_size += 1

# performs adaptive thresholding of image, C = 3 by default
thresh = cv2.adaptiveThreshold(gray.astype(np.uint8), 255, \
                               cv2.ADAPTIVE_THRESH_MEAN_C, cv2.THRESH_BINARY, adapt_box_size, c)

# finds OTSU binarisation of image
#ret, thresh = cv2.threshold(gray,0,255, cv2.THRESH_BINARY_INV+cv2.THRESH_OTSU)

#thresh = np.invert(thresh)

# saving hard copy of thresh to avoid damage from findcontours
thresh_perm = np.copy(thresh)

# find array of contours
(cnts, _) = cv2.findContours(thresh, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)

shape = np.shape(cnts)

length = int(shape[0])

areas = np.zeros(length)
side_lengths = np.zeros((length, 4))

cent_cords = [0, 0]

#cent_cords = np.zeros((length, 2))
side_ratio = np.zeros(length)

# reduced array containing only coordinates of
count = 0

# cycle over contours
for i in xrange(0, length):

    rect = cv2.minAreaRect(cnts[i])
    box = cv2.cv.BoxPoints(rect)
    box = np.int0(box)

    for side in xrange(0,4):
        if (side == 3):
            side_plus = 0
        else:
            side_plus = side + 1

        side_lengths[i][side] = math.sqrt((box[side][0] - box[side_plus][0]) \
                                         ** 2 + (box[side][1] - box[side_plus][1]) ** 2)

    areas[i] = side_lengths[i][0] * side_lengths[i][1]

    # ensure no zero area rectangles are further processed
    if ((side_lengths[i][0] != 0) & (side_lengths[i][1] != 0)):

        side_ratio[i] = side_lengths[i][0] / side_lengths[i][1]

        if (side_ratio[i] < 1):
            side_ratio[i] = 1.0 / side_ratio[i]

        if (areas[i] > 5000):
            areas[i] = 0

        if ((areas[i] > min_area) & (areas[i] < max_area) & (side_ratio[i] \
          < max_side_ratio)):

            count += 1

            delt_x_1 = box[0][0] - box[1][0]
            delt_y_1 = box[0][1] - box[1][1]

            delt_x_2 = box[1][0] - box[2][0]
            delt_y_2 = box[1][1] - box[2][1]
```

```

        delt_x = (delt_x_1 / 2) + (delt_x_2 / 2)
        delt_y = (delt_y_1 / 2) + (delt_y_2 / 2)

        cent_cords_x = box[0][0] - delt_x
        cent_cords_y = box[0][1] - delt_y

        cent_cords_vec = [cent_cords_x, cent_cords_y]

        cent_cords = np.vstack((cent_cords, cent_cords_vec))

        cv2.circle(open_cv_image, (int(cent_cords[count][0]), \
                                int(cent_cords[count][1])), 40, (B * 255, G * 255, R * 255), 2)

#cv2.drawContours(open_cv_image,[box],0,(0,0,255),2)

closest_neighbours = find_neighbours(cent_cords, count)

# number of horizontal neighbours
hor_neigh_count = 0

# number of vertical neighbours
ver_neigh_count = 0

# attributes of horizontal neighbours
hor_neigh = []

# attributes of vertical neighbours
ver_neigh = []

# recording the pix scale implied by the horizontal and vertical spacing of city blocks
pix_scale_hor = []
pix_scale_ver = []

# recording the angles of connections between neighbouring points
angle_hor = []
angle_ver = []

good_rect_log = np.zeros(count)

# iterating over starting point
for s_p in xrange(1, count):
    # iterating over both first to fourth closest neighbour
    for neighbour in xrange(0, 4):
        if ((closest_neighbours[s_p,neighbour,2] > 75) & \
            (closest_neighbours[s_p,neighbour,2] < 84)):

            good_rect_log[s_p] = 1

            pix_scale_hor.append(closest_neighbours[s_p,neighbour,2] \
                                  / horizontal_spacing)

            angle = math.atan(closest_neighbours[s_p,neighbour,1] \
                              / closest_neighbours[s_p,neighbour,0])

            angle = math.degrees(angle)
            angle_hor.append(angle)

            hor_neigh_count += 1

        if ((closest_neighbours[s_p,neighbour,2] > 88) & \
            (closest_neighbours[s_p,neighbour,2] < 94)):

            good_rect_log[s_p] = 1

            pix_scale_ver.append(closest_neighbours[s_p,neighbour,2] / \
                                  vertical_spacing)

            angle = math.atan(closest_neighbours[s_p,neighbour,0] / \
                              closest_neighbours[s_p,neighbour,1])

            angle = math.degrees(angle)
            angle_ver.append(-angle)

            ver_neigh_count += 1

```

```
pix_scale_omni = np.asarray(pix_scale_hor + pix_scale_ver)
angle_omni = np.asarray(angle_hor + angle_ver)

angle_mean, angle_std_err, pix_scale_mean, pix_scale_std_err = \
    output(pix_scale_omni, angle_omni, \
            hor_neigh_count, ver_neigh_count)

# if -p is given as a flag output is printed
if (flag == '-p'):

    print (cent_cords)

    print ('%d rectangles drawn' % count

    filename = ('./vision/adaptive_test_kern_%d.png') % kern_dims

#image = Image.fromarray(open_cv_image)
#image.save(filename)
#print('image saved to disk')

cv2.imshow('city block centres', thresh_perm)
#cv2.imshow('city block centres', open_cv_image)

# saving threshold binarisation
image_out = Image.fromarray(np.uint8(thresh_perm))
image_out.save ('./images/adapt_thresh.png')

# saving fitted rectangles
image_out = Image.fromarray(np.uint8(open_cv_image))
image_out.save ('./images/rect_fit.png')

if cv2.waitKey(0) & 0xff == 27:
    cv2.destroyAllWindows()

# angle_mean, angle_std_err, pix_scale_mean, pix_scale_std_err, count, cent_cords
# good_rect_log
#data_out = [angle_mean, angle_std_err, pix_scale_mean, pix_scale_std_err, count, cent_cords, good_re
ct_log]

return angle_mean, angle_std_err, pix_scale_mean, pix_scale_std_err, count, cent_cords, good_rect_log
```

```
import cv2
import numpy as np
from PIL import Image
from numba import jit
import cv2

#####
This script takes the local standard deviation of an image and thresholds it,
removing spurious emission from the edge of sample holder and small flecks of
dirt on the sample chip
#####

def main(filename, box_size):
    img = cv2.imread(filename, 0)

    shape = np.shape(img)

    x_pix_max = shape[0]
    y_pix_max = shape[1]

    #box_size = 3 # default value of 3
    box_length = (2 * box_size) + 1

    std_dev_map = process(img, box_length, box_size, x_pix_max, y_pix_max)
    std_dev_map = np.asarray(std_dev_map)
    std_dev_map = np.uint8(std_dev_map)

    thresh = np.zeros((x_pix_max, y_pix_max))
    thresh_value = np.percentile(std_dev_map, 78)

    std_dev_map[std_dev_map >= thresh_value] = 255
    std_dev_map[std_dev_map < thresh_value] = 0

    std_dev_map = np.uint8(std_dev_map / 255)
    std_dev_img = Image.fromarray(np.uint8(std_dev_map * 255))

    filename_out = ('./images/std_dev_map_pre.png'      )
    std_dev_img.save(filename_out)

    mask = np.ones(std_dev_map.shape, dtype='uint8'   )

    # saving hard copy of thresh to avoid damage from findcontours
    thresh_perm = np.copy(std_dev_map)

    # find array of contours
    (cnts, hierarchy) = cv2.findContours(std_dev_map, cv2.RETR_CCOMP,\n                                         cv2.CHAIN_APPROX_SIMPLE)

    shape = np.shape(cnts)
    cnts_number = shape[0]

    for i in xrange(0, cnts_number):
        #if contour is bad draw to mask
        if (is_contour_bad(cnts, i, hierarchy) != 0):
            cv2.drawContours(mask, [cnts[i]], -1, (0, 0, 0), -1)

    for i in xrange(0, cnts_number):
        #if contour is good revert mask back to original state over shape area
        if (is_contour_bad(cnts, i, hierarchy) == 0):
            cv2.drawContours(mask, [cnts[i]], -1, (1, 1, 1), -1)

    std_dev_map = thresh_perm * mask

    std_dev_img = Image.fromarray(np.uint8(std_dev_map * 255))
    filename_out = ('./images/std_dev_map_post.png'      )
    std_dev_img.save(filename_out)
    print ('standard deviation map saved'          )

    return std_dev_map

@jit(nopython=True)
def process (img, box_length, box_size, x_pix_max, y_pix_max):

    std_dev_map = np.zeros((x_pix_max, y_pix_max))

    for x in xrange(0, x_pix_max):
```

```
for y in xrange(0, y_pix_max):

    box = np.zeros((box_length, box_length))

    if ((x >= box_size) and (x < x_pix_max - box_size) and (y >= box_size) \
        and (y < y_pix_max - box_size)):

        for x_box in xrange(-box_size, box_size):
            for y_box in xrange(-box_size, box_size):
                box[x_box + box_size][y_box + box_size] = img[x + x_box][y + y_box]

    if (np.mean(box) != 0):
        std_dev_map[x][y] = (150 * np.std(box)) / np.mean(box)
    else :
        std_dev_map[x][y] = 0

return std_dev_map

# returns 1 for bad contour and 0 for good contour
def is_contour_bad (cnts, i, hierarchy):
    area = cv2.contourArea(cnts[i])
    length = cv2.arcLength(cnts[i], True)
    extension = 0

    shape = np.shape(cnts)
    cnts_number = shape[0]

    area = cv2.contourArea(cnts[i])

    # if(area >= 10000):
    #     print('inloop')
    #     for i_h in xrange(0, cnts_number):
    #         if((i == hierarchy[0][i_h][3]) & (i != i_h)):
    #             area = area - cv2.contourArea(cnts[i_h])

    if (area != 0):
        extension = ((length / 4) ** 2) / area

    if ((length >= 700) & (area >= 8000)):
        return 1
    elif (extension >= 10):
        return 1
    elif (area <= 50):
        return 1
    else :
        return 0
```

```

import os, re, sys
import numpy as np
import time, math, string
import matplotlib
from matplotlib import pyplot as plt

def index11664_fiducials():
    road_list = ['Adams', 'Bush', 'Clinton', 'Dwight', 'Eisenhwr', 'Ford', 'Grant', 'Hoover', 'India']
    cross_list = ['1st', '2nd', '3rd', '4th', '5th', '6th', '7th', '8th', '9th']
    block_row_list = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l']
    block_col_list = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l']
    corners_list = []
    for road in road_list:
        for cross in cross_list:
            for r2 in block_row_list:
                for c2 in block_col_list:
                    addr = road[0] + cross[-2:] + '_' + r2 + c2
                    if r2+c2 in ['aa', 'la', 'll']:
                        corners_list.append(addr)
    fid_list = [
        'A1_ag', 'A2_ag', 'A3_ag', 'A4_ag', 'A5_ag', 'A6_ag', 'A7_ag', 'A8_ag', 'A9_ag',
        'A1_aj', 'A2_bj', 'A3_cj', 'A4_ak', 'A5_bk', 'A6_ck', 'A7_al', 'A8_bl', 'A9_cl',
        'B1_bg', 'B2_bg', 'B3_bg', 'B4_bg', 'B5_bg', 'B6_bg', 'B7_bg', 'B8_bg', 'B9_bg',
        'B1_aj', 'B2_bj', 'B3_cj', 'B4_ak', 'B5_bk', 'B6_ck', 'B7_al', 'B8_bl', 'B9_cl',
        'C1_cg', 'C2_cg', 'C3_cg', 'C4_cg', 'C5_cg', 'C6_cg', 'C7_cg', 'C8_cg', 'C9_cg',
        'C1_aj', 'C2_bj', 'C3_cj', 'C4_ak', 'C5_bk', 'C6_ck', 'C7_al', 'C8_bl', 'C9_cl',
        'D1_ah', 'D2_ah', 'D3_ah', 'D4_ah', 'D5_ah', 'D6_ah', 'D7_ah', 'D8_ah', 'D9_ah',
        'D1_aj', 'D2_bj', 'D3_cj', 'D4_aj', 'D5_bk', 'D6_ck', 'D7_al', 'D8_bl', 'D9_cl',
        'E1_bh', 'E2_bh', 'E3_bh', 'E4_bh', 'E5_bh', 'E6_bh', 'E7_bh', 'E8_bh', 'E9_bh',
        'E1_aj', 'E2_bj', 'E3_cj', 'E4_aj', 'E5_bk', 'E6_ck', 'E7_al', 'E8_bl', 'E9_cl',
        'F1_ch', 'F2_ch', 'F3_ch', 'F4_ch', 'F5_ch', 'F6_ch', 'F7_ch', 'F8_ch', 'F9_ch',
        'F1_aj', 'F2_bj', 'F3_cj', 'F4_aj', 'F5_bk', 'F6_ck', 'F7_al', 'F8_bl', 'F9_cl',
        'G1_aj', 'G2_aj', 'G3_aj', 'G4_aj', 'G5_aj', 'G6_aj', 'G7_aj', 'G8_aj', 'G9_aj',
        'G1_aj', 'G2_bj', 'G3_cj', 'G4_aj', 'G5_bk', 'G6_ck', 'G7_al', 'G8_bl', 'G9_cl',
        'H1_bj', 'H2_bj', 'H3_bj', 'H4_bj', 'H5_bj', 'H6_bj', 'H7_bj', 'H8_bj', 'H9_bj',
        'H1_aj', 'H2_bj', 'H3_cj', 'H4_aj', 'H5_bk', 'H6_ck', 'H7_al', 'H8_bj', 'H9_cl',
        'I1_ci', 'I2_ci', 'I3_ci', 'I4_ci', 'I5_ci', 'I6_ci', 'I7_ci', 'I8_ci', 'I9_ci',
        'I1_aj', 'I2_bj', 'I3_cj', 'I4_aj', 'I5_bk', 'I6_ck', 'I7_al', 'I8_bj', 'I9_cl'
    ]
    fid_list = sorted(fid_list)
    corners_list = sorted(corners_list)
    return fid_list, corners_list

def hits_scrape(fid, diamond_dict):
    hits_dict = {}
    for i in range(11664):
        hits_dict[diamond_dict[i]] = 0
    f = open(fid)
    for line in f.readlines()[1:]:
        entry = line.split()
        i = int(entry[0])
        yesno = 1
        #yesno = int(entry[1])
        hits_dict[diamond_dict[i]] = yesno
    return hits_dict

# valid for data collection in June 2016
def collect_dicts():
    road_list = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I']
    daor_list = ['I', 'H', 'G', 'F', 'E', 'D', 'C', 'B', 'A']
    cros_list = ['1', '2', '3', '4', '5', '6', '7', '8', '9']
    sorc_list = ['9', '8', '7', '6', '5', '4', '3', '2', '1']
    wind_list = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l']
    dniw_list = ['l', 'k', 'j', 'i', 'h', 'g', 'f', 'e', 'd', 'c', 'b', 'a']
    ordr_list = []
    addr_dict = {}
    ordr_dict = {}

    i = 0
    for c in range(9):
        for r in range(9):
            for wc in range(12):
                #print
                for wr in range(12):
                    if (c % 2 == 0):
                        if (wc % 2 == 0):
                            #addr = daor_list[r] + sorc_list[c] + '_' + dniw_list[wc] + dniw_list[wr]
                            addr = daor_list[r] + sorc_list[c] + '_' + dniw_list[wc] + dniw_list[wr]
                            if (addr not in addr_dict):
                                addr_dict[addr] = 1
                            else:
                                addr_dict[addr] += 1
                            ordr_list.append(addr)
                            ordr_dict[addr] = i
                            i += 1

```

```

        ordr_list.append(addr)
        #print addr,'1',
    else :
        #addr = daor_list[r] + sorc_list[c] + '_' + dniw_list[wc] + wind_list[wr]
        addr = daor_list[r] + sorc_list[c] + '_' + dniw_list[wc] + wind_list[wr]
        ordr_list.append(addr)
        #print addr,'2',
    else :
        if (wc % 2 == 0) :
            #addr = daor_list[r] + cros_list[c] + '_' + wind_list[wc] + dniw_list[wr]
            addr = road_list[r] + sorc_list[c] + '_' + wind_list[wc] + dniw_list[wr]
            ordr_list.append(addr)
            #print addr,'3',
        else :
            #addr = daor_list[r] + cros_list[c] + '_' + wind_list[wc] + wind_list[wr]
            addr = road_list[r] + sorc_list[c] + '_' + wind_list[wc] + wind_list[wr]
            ordr_list.append(addr)
            #print addr,'4',
        addr_dict[addr] = i
        ordr_dict[i] = addr
        #print i,
        i += 1
    return addr_dict, ordr_dict

def normal_dicts () :
    road_list = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I']
    daor_list = ['I', 'H', 'G', 'F', 'E', 'D', 'C', 'B', 'A']
    cros_list = ['1', '2', '3', '4', '5', '6', '7', '8', '9']
    sorc_list = ['9', '8', '7', '6', '5', '4', '3', '2', '1']
    wind_list = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l']
    dniw_list = ['l', 'k', 'j', 'i', 'h', 'g', 'f', 'e', 'd', 'c', 'b', 'a']
    ordr_list = []
    ordr_dict = {}
    addr_dict = {}
    i = 0
    for c in range(9):
        #print
        for r in range(9):
            #print
            for wc in range(12):
                #print
                for wr in range(12):
                    if (r % 2 == 0):
                        if (wr % 2 == 0):
                            addr = road_list[r] + cros_list[c] + '_' + wind_list[wc] + wind_list[wr]
                            ordr_list.append(addr)
                            #print addr,
                        else :
                            addr = road_list[r] + cros_list[c] + '_' + wind_list[wc] + wind_list[wr]
                            ordr_list.append(addr)
                            #print addr,
                    else :
                        if (wr % 2 == 0):
                            addr = road_list[r] + cros_list[c] + '_' + wind_list[wc] + wind_list[wr]
                            ordr_list.append(addr)
                            #print addr,
                        else :
                            addr = road_list[r] + cros_list[c] + '_' + wind_list[wc] + wind_list[wr]
                            ordr_list.append(addr)
                            #print addr,
                    ordr_dict[i] = addr
                    addr_dict[addr] = i
                    #print i,
                    i += 1
    return addr_dict, ordr_dict

def get_xy (xtal_name):
    w2w = 0.125
    b2b_horz = 0.825
    b2b_vert = 1.125
    #b2b_horz = 0
    #b2b_vert = 0
    cell_format = [9, 9, 12, 12]
    entry = xtal_name.split('_')[-2:]
    R, C = entry[0][0], entry[0][1]
    r2, c2 = entry[1][0], entry[1][1]
    blockR = int(string.uppercase.index(R))

```

```
blockC = int(C) - 1
windowR = string.lowercase.index(r2)
windowC = string.lowercase.index(c2)
x = (blockC * b2b_horz) + (blockC * (11) * w2w) + (windowC * w2w)
y = (blockR * b2b_vert) + (blockR * (11) * w2w) + (windowR * w2w)
return x, y

def main():
    x_list, y_list, z_list = [], [], []
    #[addr] = i, [i] = addr
    normal_addr_dict, normal_ordr_dict = normal_dicts()
    fid_list, corners_list = index11664_fiducials()
    for i in sorted(normal_ordr_dict.keys()):
        addr = normal_ordr_dict[i]
        x, y = get_xy(addr)
        if addr in corners_list:
            z = 2
        elif addr in fid_list:
            z = 7
        else:
            z = 4
        x_list.append(float(x))
        y_list.append(float(y))
        z_list.append(float(z))

    X = np.array(x_list)
    Y = np.array(y_list)
    Z = np.array(z_list)
    xr = X.ravel()
    yr = Y.ravel()
    zr = Z.ravel()

    fig = plt.figure(num=None, figsize=(9,9), facecolor='0.6' , edgecolor='k' )
    fig.subplots_adjust(left=0.03,bottom=0.03,right=0.97,top=0.97,wspace=0,hspace=0)
    ax1 = fig.add_subplot(111, aspect='equal' , axisbg='0.7' )
    ax1.scatter(xr, yr, c=zr, s=14, alpha=1, marker='s' , linewidth=0.1, cmap='PuOr' )
    ax1.set_xticks([2.2*x for x in range(11)])
    ax1.set_yticks([2.5*x for x in range(11)])
    ax1.set_xlim(xr.min()-0.2, xr.max()+0.2)
    ax1.set_ylim(yr.min()-0.2, yr.max()+0.2)
    ax1.invert_yaxis()
    plt.savefig('chip_image.png' , dpi=600, bbox_inches='tight' , pad_inches=0.05)

    return X, Y, Z

if __name__ == '__main__':
    main()
    plt.show()
    plt.close()
```

```
import numpy as np
import visual_map
import os, re, sys
import time, math, string
from matplotlib import pyplot as plt

print ('in python script')

f = open('filter_list.tmp')

num_lines = sum(1 for line in open('filter_list.tmp'))

info_log = np.zeros(11664)

# reading input file line by line to get ID of cell and spot number
for line in xrange(0, num_lines):

    line_content = f.readline()

    line_content = line_content.split()

    filename = line_content[0]

    i = int(filename[-9:-4])

    if (sys.argv[1] == '-s' ):
        info_log[i] = int(line_content[1])

    if (sys.argv[1] == '-i' ):
        info_log[i] = line_content[1]

#for i in xrange(0,num_lines):
#    print(info_log[i])

x_list, y_list, z_list = [], [], []
#[addr]=i, [i]=addr
collect_addr_dict, collect_ordr_dict = visual_map.collect_dicts()
#normal_addr_dict, normal_ordr_dict = visual_map.normal_dicts()
fid_list, corners_list = visual_map.index11664_fiducials()

for j in range(0, 11664):
    addr = collect_ordr_dict[j]
    #addr = normal_ordr_dict[j]
    x, y = visual_map.get_xy(addr)

    if (sys.argv[1] == '-s' ):
        z = info_log[j]
        *****
        if(info_log[j] > 50):
            z = 100
        else:
            z = 0
        *****
    if (sys.argv[1] == '-i' ):
        if (info_log[j] > 0.1):
            z = 100
        else :
            z = 0

    x_list.append(float(x))
    y_list.append(float(y))
    z_list.append(float(z))

X = np.array(x_list)
Y = np.array(y_list)
Z = np.array(z_list)
xr = X.ravel()
yr = Y.ravel()
zr = Z.ravel()

print ('before plot')

fig = plt.figure(num=None, figsize=(9,9), facecolor='0.6', edgecolor='k')
fig.subplots_adjust(left=0.03,bottom=0.03,right=0.97,top=0.97,wspace=0,hspace=0)
ax1 = fig.add_subplot(111, aspect='equal', axisbg='0.7')
```

```
ax1.scatter(xr, yr, c=zr, s=16, alpha=1, marker='s' , linewidth=0.1) #,cmap='PuOr')
ax1.set_xticks([2.2*x for x in range(11)])
ax1.set_yticks([2.5*x for x in range(11)])
ax1.set_xlim(xr.min()-0.2, xr.max()+0.2)
ax1.set_ylim(yr.min()-0.2, yr.max()+0.2)
ax1.invert_yaxis()

if (sys.argv[1] == '-s' ):
    plt.savefig('spot_plot.png' , dpi=600, bbox_inches='tight' , pad_inches=0.05)

if (sys.argv[1] == '-i' ):
    plt.savefig('index_plot.png' , dpi=600, bbox_inches='tight' , pad_inches=0.05)

print ('after plot')

np.save('info_log.npy' , info_log)
```

```
import numpy as np
import visual_map
import os, re, sys
import time, math, string
from matplotlib import pyplot as plt

def main(off_zero_frame_file, on_zero_frame_file, plot_save_path):

    off_zero_frame_load = np.loadtxt(off_zero_frame_file)
    on_zero_frame_load = np.loadtxt(on_zero_frame_file)

    off_zero_frame_load = negative_frame_load / np.mean(off_zero_frame_load)
    on_zero_frame_load = zero_frame_load / np.mean(on_zero_frame_load)

    j_max = 11664

    off_zero_length = np.size(off_zero_frame_load)
    on_zero_length = np.size(on_zero_frame_load)

    if (off_zero_length != j_max * 2):
        print ('off zero frame file is incorrect length, should contain %d elements' ) % j_max
        return 0
    if (on_zero_length != j_max * 2):
        print ('on zero frame file is incorrect length, should contain %d elements' ) % j_max
        return 0

    crystal_strength_log = on_zero_frame_file

    x_list, y_list, z_list = [], [], []
    # [addr] - i, [j] - addr
    #collect_addr_dict, collect_ordr_dict = visual_map.collect_dicts()
    normal_addr_dict, normal_ordr_dict = visual_map.normal_dicts()
    fid_list, corners_list = visual_map.index11664_fiducials()

    for j in range(0, j_max):
        #addr = collect_ordr_dict[j]
        addr = normal_ordr_dict[j]
        x, y = visual_map.get_xy(addr)

        if (crystal_strength_log[j][0] >= 10):
            z = 0
        else :
            z = relative_intensity_log[j][0]

        x_list.append(float(x))
        y_list.append(float(y))
        z_list.append(float(z))

    X = np.array(x_list)
    Y = np.array(y_list)
    Z = np.array(z_list)
    xr = X.ravel()
    yr = Y.ravel()
    zr = Z.ravel()

    print ('before plot' )

    fig = plt.figure(num=None, figsize=(9,9), facecolor='0.6' , edgecolor='k' )
    fig.subplots_adjust(left=0.03,bottom=0.03,right=0.97,top=0.97,wspace=0,hspace=0)
    ax1 = fig.add_subplot(111, aspect='equal' , axisbg='0.7' )
    ax1.scatter(xr, yr, c=zr, s=14, alpha=1, marker='s' , linewidth=0.1) #,cmap='PuOr'
    ax1.set_xticks([2.2*x for x in range(11)])
    ax1.set_yticks([2.5*x for x in range(11)])
    ax1.set_xlim(xr.min()-0.2, xr.max()+0.2)
    ax1.set_ylim(yr.min()-0.2, yr.max()+0.2)
    ax1.invert_yaxis()

    plt.show

    plt.savefig(plot_save_path, dpi=600, bbox_inches='tight' , pad_inches=0.05)

return 1
```

polar_fit_plot.py

Tue Sep 06 15:17:00 2016

1

```
import reduc
import intensity_plot
import os

def main(off_zero_filename_in, on_zero_filename_in, plot_save_path):

    off_zero_filename_out = './off_zero_i_list.tmp'
    on_zero_filename_out = './on_zero_i_list.tmp'

    img_array, pix_scale, x_real_offset, y_real_offset, theta = \
        reduc.meta_sweep(off_zero_filename_in, off_zero_filename_out)

    img_array, pix_scale, x_real_offset, y_real_offset, theta = \
        reduc.meta_sweep(on_zero_filename_in, on_zero_filename_out, on_zero=1)

    intensity_plot.main(off_zero_filename_out, on_zero_frame_filename_out, plot_save_path)

    os.remove(off_zero_filename_out)
    os.remove(on_zero_filename_out)

    return 0
```

```
import numpy as np
import visual_map
import os, re, sys
import time, math, string
from matplotlib import pyplot as plt

def main(pre_load_log_filename, post_load_log_filename, plot_save_path):

    intensity_log_pre_load = np.loadtxt(pre_load_log_filename)
    intensity_log_post_load = np.loadtxt(post_load_log_filename)

    intensity_log_pre_load = intensity_log_pre_load / np.mean(intensity_log_pre_load)
    intensity_log_post_load = intensity_log_post_load / np.mean(intensity_log_post_load)

    j_max = 11664

    pre_length = np.size(intensity_log_pre_load)
    post_length = np.size(intensity_log_post_load)

    if (pre_length != j_max * 2):
        print ('pre load file is incorrect length, should contain %d elements' ) % j_max
        return 0
    if (post_length != j_max * 2):
        print ('post load file is incorrect length, should contain %d elements' ) % j_max
        return 0

    relative_intensity_log = intensity_log_post_load / intensity_log_pre_load

    x_list, y_list, z_list = [], [], []
    #[addr] = i, [i] = addr
    #collect_addr_dict, collect_ordr_dict = visual_map.collect_dicts()
    normal_addr_dict, normal_ordr_dict = visual_map.normal_dicts()
    fid_list, corners_list = visual_map.index11664_fiducials()

    for j in range(0, j_max):
        #addr = collect_ordr_dict[j]
        addr = normal_ordr_dict[j]
        x, y = visual_map.get_xy(addr)

        if (relative_intensity_log[j][0] >= 3):
            z = 0
        else :
            z = relative_intensity_log[j][0]

        x_list.append(float(x))
        y_list.append(float(y))
        z_list.append(float(z))

    X = np.array(x_list)
    Y = np.array(y_list)
    Z = np.array(z_list)
    xr = X.ravel()
    yr = Y.ravel()
    zr = Z.ravel()

    print ('before plot' )

    fig = plt.figure(num=None, figsize=(9,9), facecolor='0.6' , edgecolor='k' )
    fig.subplots_adjust(left=0.03,bottom=0.03,right=0.97,top=0.97,wspace=0,hspace=0)
    ax1 = fig.add_subplot(111, aspect='equal' , axisbg='0.7' )
    ax1.scatter(xr, yr, c=zr, s=14, alpha=1, marker='s' , linewidth=0.1) #,cmap='PuOr'
    ax1.set_xticks([2.2*x for x in range(11)])
    ax1.set_yticks([2.5*x for x in range(11)])
    ax1.set_xlim(xr.min()-0.2, xr.max()+0.2)
    ax1.set_ylim(yr.min()-0.2, yr.max()+0.2)
    ax1.invert_yaxis()

    plt.show

    plt.savefig(plot_save_path, dpi=600, bbox_inches='tight' , pad_inches=0.05)

return 1
```

```
fit_plot.py           Fri Aug 26 13:49:24 2016           1
import  reduc
import  intensity_plot
import  os

def  main(filename_in_pre, filename_in_post, plot_save_path):

    filename_out_pre = './pre_load_i_list.tmp'
    filename_out_post = './post_load_i_list.tmp'

    img_array, pix_scale, x_real_offset, y_real_offset, theta = reduc.meta_sweep(filename_in_pre, filename_out_pre)
    img_array, pix_scale, x_real_offset, y_real_offset, theta = reduc.meta_sweep(filename_in_post, filename_out_post)

    intensity_plot.main(filename_out_pre, filename_out_post, plot_save_path)

    os.remove(filename_out_pre)
    os.remove(filename_out_post)

return  0
```