# REAL TIME 3D SCAN CALIBRATION

Patrick Green

January 4, 2021

### 1 Introduction

The aim of this coursework was to estimate the 3D reconstruction accuracy of data obtained from a stereo depth sensor. The data in question consists of 4 greyscale image sets tracking a glass sheet covered with textured paper. Each image set corresponds to a movement of the plane relative to the camera: three of the sets show movement in the X, Y, and Z axes, while the fourth one shows movement of the depth sensor in the Z axis.

Since the glass/paper plane is assumed to be flat, ideal reconstruction would also produce a flat 2D plane in its region. However, the texture on the paper sheet causes the sensor to interpret it at different depths, thereby causing reconstruction errors. The accuracy is then measured by calculating the best fit 2D plane equation. This allows to calculate the residual errors at each point, and the overall root-mean-square error. The latter two are then presented as results.

Note that this assignment was written in *Python* and thus references various *Python* modules.

## 2 Method

The original data obtained from each individual image contained a set of 1200x1200 pixels, each of which stored two sets of values:

- The RGB color values all of these were equal due to the image being grayscale
- The XYZ positional values detected by the stereo sensor

The analysis process consisted of three main stages, each of which are further described in detail:

- Data Selection identifying the set of values from the dataset relevant to the analysis
- Analysis Algorithm invoking a plane fitting algorithm and calculating the residuals
- Tracking Algorithm maintaining the selected dataset between image changes

#### 2.1 Data Selection

Knowing that the two cameras do not see the same portion of the scene, some areas of the image will contain erroneous data. Furthermore, the image sets visualise the movement of the glass sheet, which means that the sheet is not always guaranteed to fill the entire field of view of both cameras. These two properties meant that accurate analysis could only be carried out on a specific region of the image. This region is further referred to as the **patch**.

Selection of the patch was based on examining the image set, and identifying a region of the textured sheet that is present on the screen for the majority of the images. Since each image set described different motions of the sheet, the aforementioned region of one set would not necessarily be applicable to another set. This suggested specifying a patch for each of the four image sets.

In addition, it was known that the errors resulting from differences between the cameras were concentrated at the bottom of the image, and hence care was taken for the patch not to extend too far down the full image. Finally, to avoid image boundary problems, the patch was never placed closer than 50 pixels to any edge.

Hence, the chosen patches per dataset (referenced to the first image of the set) were as follows:

- zstatic 500x500 pixels at (300, 200).
- *xmove* 100x900 pixels at (50, 200). This allows the patch to be on screen for all images in the set, and never exceed the 50 pixel edge boundary.
- ymove 900x100 pixels at (100, 50). The patch is on the screen only for the first 6 images.
- *zmove* 500*x*500 pixels at (300, 200).

#### 2.2 Plane Fitting

To calculate the RMS for each patch, it was required to fit a plane. First, the patch was cropped about the x and y coordinates of its top left and bottom right corners. Next, a method of least squares was used to solve for the coefficients of the plane equation shown below.

$$z = ax + by + c \tag{1}$$

The numpy.linalg.lstsq method was used to calculate the parameters. The method uses the matrix A holding every point of the mesh grid as a row, and the value for x, y, z and 1 in each column. A scatter matrix S is then computed from A after subtracting the center of mass of the data and multiplying itself by its transpose  $S = D^T D$ . The normal vector of the best-fit plane is then found by taking the eigenvector of S with the smallest eigenvalue. The method returns the coefficients, and the normal vector and plane equation take the following form.

$$z = ax + by + c$$
  $n = \{-a/c, -b/c, 1/c\}$  (2)

The residuals  $e_{i,j}$  are then computable by taking any point given by the plane equation r, subtracting this from the original data point p, and taking the dot product with the normal vector of the plane to find the perpendicular distance:

$$e_{i,j} = (r-p)\dot{n} \qquad RMS = \sqrt{\sum e_{i,j}^2} \quad (3)$$

The RMS was then calculated by taking the square root of the sum of the square of the residuals  $e_{i,j}$ , as shown in the equation above.

The *Python* code of this implementation is as follows:

```
def fitplane_m(patch):
    surface = np.copy(patch)
    X_p = surface[:, :, 0]
    Y_p = surface[:, :, 1]
    Z_p = surface[:, :, 2]
```

```
patch = patch.reshape(-1,3)
size = patch.shape[0]
A = np.c_[patch[:,0], patch[:,1], np.ones(size)]
C, r, rank, s = np.linalg.lstsq(A,patch[:,2])
Z = C[0] * X_p + C[1] * Y_p + C[2]
Xr = X_p - X_p
Yr = Y_p - Y_p
Zr = Z_p - Z
# n.(p-r)
residuals = (-C[0] / C[2]) * Xr + (-C[1] / C[2]) * Yr + (1 / C[2]) * Zr
rms = np.sqrt(np.average(np.square(r)))
return C, rms, residuals
```

As such, the algorithm enabled straightforward calculation of the residual array for any patch.

### 2.3 Tracking Algorithm

To ensure the RMS plots show the correct correlation, the exact same patch is used in adjacent frames. To do this, the translation of the patch was tracked through the set of images using a method of cross correlation, implemented with the phaseCorrelate method of Python's cv2 package.

Cross correlation works by taking single strips from each image that lie in the same position, and working out the difference in phase between them. In the images below, comparison between the columns in the left and the right images is of interest.



Let A be a column in the left image and B be a column in the right. The phase distribution can be calculated by taking the Fourier transform of the multiplication of both arrays' inverse Fourier transform, where one array is reversed. The phase difference is then calculated by finding the argument of the maximum in this distribution. The average of each column is the overall phase shift in the image.

$$\mathcal{F}{A} = a \qquad \qquad \mathcal{F}{B[::-1]} = b^* \qquad \qquad phase^y = argmax(\mathcal{F}{a \times phase^y}) = b^*$$

The same operation is done with each row, giving a translation vector  $[phase^x, phase^y]$ . In order to rescale the image between adjacent frames, a similar approach was taken, but instead the phase difference between the two images was examined in polar coordinates.

 $b^*$ ) (4)



Taking the center of the patch as the origin for the conversion to polar coordinates, the new x and y images are computed using the equations below.

 $Patch\_center = (c_x, c_y) \quad r = \sqrt{(x - c_x)^2 + (y - c_y)^2} \quad theta = \arctan\left((y - c_y)/(x - c_x)\right) (5)$ 

The *Python* code to implement the above algorithm is as follows:

```
def reproject_image_into_polar(data, origin):
    ny, nx = data.shape[:2]
    if origin is None:
        origin = (nx//2, ny//2)
    # Determine that the min and max r and theta coords
    x, y = index_coords(data, origin=origin)
    r, theta = cart2polar(x, y)
    # Make a regular (in polar space) grid based on the min and max r \vartheta theta
    r_i = np linspace(r min(), r max(), nx)
    theta_i = np.linspace(theta.min(), theta.max(), ny)
    theta_grid, r_grid = np.meshgrid(theta_i, r_i)
    # Project the r and theta grid back into pixel coordinates
    xi, yi = polar2cart(r_grid, theta_grid)
    xi += origin[0] # We need to shift the origin back to
    yi += origin[1] # back to the lower-left corner...
    xi, yi = xi.flatten(), yi.flatten()
    coords = np.vstack((xi, yi)) # (map_coordinates requires a 2xn array)
```

```
# Reproject each band individually and the restack
bands = []
for band in data.T:
    zi = map_coordinates(band, coords, order=1)
    bands.append(zi.reshape((nx, ny)))
output = np.dstack(bands)
return output, r_i, theta_i
```

By taking the cross correlation, the phase difference in the radial dimension dr can be computed. dr can then be utilized to rescale an arbitrary sized rectangle about its center, using the following equations.

$$(w,h) = (width, height)$$
  $w = w + dr/\sqrt{1 + (h/w)}$   $h = h + dr/\sqrt{1 + (w/h)}$  (6)

In addition, to make the cross correlation easier to detect, instead of using the original gray scale image, the image was filtered with a **threshold** equal to a multiple of its average brightness. This set areas of the brightness below this threshold to zero, allowing for a more distinguishable difference in phase. An example of a thresholded image used for the cross correlation is shown below, for Cartesian and polar coordinates.



## 3 Results

### 3.1 Overall Performance

The tracking worked well in the cases of the *xmove* and the *ymove* plane data, where the thresholding coefficient was 1.5. The cross correlation was able to recognize patterns of texture over multiple images, as the lighting stayed consistently distributed across the image. This allowed for a similar threshold to be used before the cross correlation, in order to enhance specific patterns in the image. Below is a good example of when the tracking worked, shown using part of the y plane data set. Over the time step from left to right, the patch has moved downwards with the sheet, staying in the same relative position.



However, the tracking method struggled with both the *zmove* and *zstatic* plane data sets, as the lighting changed considerably, which gave rise to variations in the phase between adjacent images. This is evident in the images below, taken from the static plane data set. Over the time step from left to right, it is clear that the plate has moved away from the camera, yet the patch has failed to rescale, and remains the same size. This suggested reducing the thresholding coefficient to 1.0 for the *zstatic* set. The coefficient was left at 1.5 for the *zmove* set.



### 3.2 RMS Against Movement Across Axis

The calculated RMS values allowed to easily visualize the reconstruction accuracy across each data set. For this, a reference value was assigned to every image of each data set:

- For the *zstatic* and *zmove* datasets, this was the average Z value.
- For the *xmove* and *ymove* sets, this was the center X or Y value of the patch. Note that the *ymove* dataset only contains 6 points, as explained in 2.1.

This allowed to plot a single graph for each dataset, plotting the reference value against RMS. These graphs are as follows:



(a) The RMS against average Z value of the (b) The RMS against average Z value of the *zstatic* dataset *zmove* dataset



(c) The RMS against central X value of the (d) The RMS against central Y value of the *xmove* dataset *ymove* dataset

It can be seen that the trends are not consistent for different datasets. The *zstatic* dataset appears to gradually increase in RMS as the source gets closer to the image, also increasing sharply once the panel has moved too far from the sensor. The *zmove* dataset appears to generally show an upwards trend as well, with the RMS increasing as the plane is moved further away from the sensor. The x axis on each of these graphs plots the diagonal length of the patch, which is inversely proportional to the sheet's distance from the camera. It is also seen that the low-Z RMS values of *zstatic* are considerably larger than any RMS value for z data set, further leading to suspicions that said detections may have been flawed. An assumption can thus be made that the sensor does not work at all after a specific distance, and that it is also more error prone the closer the sheet is held.

For the *xplane* plot, it is obvious that the error reduces the further to the right the patch is moved. With the *yplane* plot it is discovered that the error is largest when the patch is at the top and the bottom of the camera, and reduces the closer to the center it gets. However, neither error plot shows particularly smooth curves, which could be due the the patch not being tracked perfectly. From these results it can be assumed that the errors are high at the extremities of the sensor. Thus it can be inferred that the sensor sits level with the camera in the vertical dimension, but sits further to the right of the camera in the horizontal.

#### 3.3 Core Data Table

In order to present data obtained from each individual image, four tables were created, one per image set. For each image in its set, they indicate the size of the patch, the obtained RMS, and the number of outlier pixels - data points whose residuals are more than 5 times the value of the RMS. The tables are as follows:

Image Name	Patch Size	RMS	Outliers
plane_1	500x500	0.000838595853324	0
$plane_2$	498x497	0.000731488965102	66
$plane_3$	498x497	0.000574810665347	0
$plane_4$	425x424	0.000245128257334	79
$plane_5$	373x372	0.000183003742057	40
$plane_6$	369x368	0.00043299739139	20
$plane_7$	369x368	0.000712188629862	0
$plane_8$	369x368	0.00105651878929	0
$plane_9$	369x368	0.00161889378992	0
$\mathrm{plane}\_10$	369x368	0.00184708297143	0
${\rm plane}\_11$	369x368	0.00322334027933	0

Table 1: Data obtained for *zstatic* 

In the above table it can be noted that most images have no outliers, though the few that do, appear to be located around the point where RMS begins to increase, as the sheet is too far from the sensor to function. This implies that the increased RMS is due to both poor tracking (as assumed before) and the sensor's functionality being impacted as the patch moves out of range.

Table 2: Data obtained for xmove

Image Name	Patch Size	RMS	Outliers
planex_0004	100x900	0.000554471342961	0
planex_0008	100x900	0.000507144083826	0
planex_0012	100x900	0.0004375587023	0
planex_0016	100x900	0.00040250978544	0
planex_0020	100x900	0.000392016674409	0
planex_0024	100x900	0.000374092926782	24
planex_0028	100x900	0.000343853071251	0
planex_0032	100x900	0.000359262931683	28
planex_0036	100x900	0.0003326686517	30
planex_0040	100x900	0.000346005319161	55
planex_0044	100x900	0.000342467606148	0

For the *xmove* the outliers exist where the patch is furthest to the right. This seems weird as the RMS is never as much as a multiple of 5 larger or smaller.

Image Name	Patch Size	RMS	Outliers
planey_06	900x100	0.000606973636011	0
planey_11	900 x 100	0.000461616146083	0
planey_16	900x100	0.000420548611793	0
planey_21	900x100	0.000471577836028	0
planey_26	900x100	0.000482248269004	0
planey_31	900x100	0.000570198258386	0

Table 3: Data obtained for ymove

The ymove table has no outliers so we can assume that the sensors functionality varies as the RMS insinuates.

Image Name	Patch Size	RMS	Outliers
$planez_{01}$	500x500	0.000987374627712	0
$\rm planez\_04$	500 x 500	0.000868365445868	0
$\rm planez\_07$	488x482	0.000686700738521	0
$\rm planez\_10$	466x460	0.000576914850785	30
$planez_{13}$	431x425	0.000449638364089	0
$\rm planez\_16$	390x384	0.000326042991249	56
$planez_{19}$	369x363	0.000238867955045	110
$\rm planez\_22$	350x344	0.000246150549064	16
$\rm planez\_25$	350x344	0.000345236000602	54
$planez_{28}$	350x344	0.000483857967057	0
$\rm planez\_31$	350x344	0.000566167547685	0

Table 4: Data obtained for *zmove* 

For the *zmove* dataset the largest amount of outliers lye in the center of the varying z positions. As the error at the boundaries is roughly 5 times the error in the center it is likely the outliers are due to the sensor but are only visible when the center and plates relative positions are optimal.

#### 3.4 Residual Color Maps

To look at how the error varied throughout a patch, the residual array of each patch was obtained. Each point  $e_i$  of this array was then colored according to its value relative to the RMS value  $\sigma$ :

- Red if  $e_i < -\sigma$
- Yellow if  $-\sigma < e_i < 0$
- Green if  $0 < e_i < \sigma$
- Blue if  $e_i > \sigma$

What follows is a set of side-by-side comparisons: the middle image of each dataset was chosen, and a 500x500 patch starting at (400, 200) was selected. Figures were then plotted for both the selected patch, and its colored residual array.



Figure 2: The patch and residual array for the zstatic dataset



Figure 3: The patch and residual array for the xmove dataset



Figure 4: The patch and residual array for the ymove dataset



Figure 5: The patch and residual array for the *zmove* dataset

It is seen that mapping the colored array to the patch is generally not obvious. Certain texture shapes can rarely be identified on the colored array, for example, the **xmove** residual color map contains a vague outline of a rectangle, which is also present on the patch image.

One observation that can be made from this data is that the majority of positive errors (blue and green points) are situated near the edges of the patch. This is the case for all datasets except zstatic, which instead contains a mixture of blue and red points (positive and negative extremes) in the middle. This may be explained by the least squares plane fitting method being biased to the center of the patch.

However, correlations between the patch pixel color and residual point color are not clear. Nonetheless, these images verify that the 3D image reconstruction always creates plenty of residual offset values across the entire patch.

### 4 Overall Observations and Conclusions

The most important observation for the obtained results is that not all datasets perform equally. It is especially obvious with the *zstatic* and *zmove* sets: the residual array of *zstatic* differed the most from the rest, and the performance of both of these sets was hampered by detection problems. These could include image blurring and a significant decrease in lightness for the later images of both sets. Image blurring can be solved by readjusting the camera focus, either manually or through autofocus, whereas equalizing lighting may be possible by having a uniform light source, placed alongside the movement axes.

The sensor itself is assumed to lie to the right of the image and is level with the camera, assuming the error is largest at the boundaries of its view. Its performance also reduces the closer the sheet sits to the camera, as the sensor itself is picking up more texture than it should. The error also increases substantially once the diagonal of the patch reduces to a size of 440. Furthermore, it was inferred that for the *zstatic* set, the increased RMS is mainly due to the sensor, whereas for *zplane*, RMS is most affected by poor tracking. From this it can be assumed that the *zstatic* data set gives a better interpretation of the sensor's performance over the Z dimension.

It can be seen that while the generated tables already contain the information of the RMS graphs, the latter do not specify the time order of these points. This turned out to be crucial to identifying RMS trends, which could not be seen from the graphs, but were present in the tables. This suggests that further performance measurements are best shown in an ordered list, rather than a scatter plot, since this also allows to link RMS to the number of outliers.

Overall, it can be concluded that while reconstruction was generally not a problem for a welldefined patch, tracking performance is nonetheless much better across the X and Y axes.

## 5 Appendix - Code Listings

```
5.1 main.py
```

```
from fetch import load_mesh
import cv2
from flask import request, Response, Flask
import numpy as np
#import serial # COMMENT OUT ON DICE
from lib.match import phase1, reproject_image_into_polar, fitplane_m
import matplotlib.pyplot as plt
from matplotlib import colors
plots = {"plane": (-1,11, lambda src: src < 1.0 * np.sum(src) / (src.shape[0]</pre>
           * src.shape[1]), True, 2, np.array([300, 200]),np.array([500, 500])),
         "planex": (-1,11, lambda src: src < 1.5 * np.sum(src) / (src.shape[0]
           * src.shape[1]), False, 0,np.array([50, 200]),np.array([100, 900])),
         "planey": (-1,11, lambda src: src < 1.5 * np.sum(src) / (src.shape[0]
           * src.shape[1]), False, 1,np.array([100, 50]),np.array([900, 100])),
         "planez": (-1,11, lambda src: src < 1.5 * np.sum(src) / (src.shape[0]
           * src.shape[1]), True, 2,np.array([300, 200]),np.array([500, 500]))}
application = Flask(__name__, static_path="/static")
def show(plotname):
    startafter, end, thresh, resize, xaxis, orgs, width = plots[plotname]
    start = True
    s = orgs
    w = width
    results = []
    for i, data in enumerate(load_mesh(plotname)):
        if i > startafter and i < end:
            # grey scale image anyway so just call first indices
            frame = data[:, :, 0] * 255.0
            if start:
                last = frame
            # Translation
            dp, conv1 = phase1(last, frame, thresh)
            s = s + dp
            conv2 = np.copy(conv1)
            if resize:
                framep, r_i, theta_i = reproject_image_into_polar(data[:, :, 0:3] * 255.0,
                  (s + w / 2.0).astype("int"))
                framep = framep[:, :, 0]
                if start:
                    lastp = framep
```

```
# Rescale size
    dp, conv2 = phase1(lastp, framep, thresh)
    if dp[1] < 0:
        w[0] += int(dp[1] / np.sqrt(1+w[1]/w[0]))
        w[1]+=int(dp[1] / np.sqrt(1+w[0]/w[1]))
        s[0] = int(dp[1] / 2*np.sqrt(1+w[1]/w[0]))
        s[1]-=int(dp[1] / 2*np.sqrt(1+w[1]/w[0]))
    lastp = framep
start = False
# Check if still visable
if np.all([s + w < np.asarray(frame.shape), 0 < s]):</pre>
    plt.close("all")
    e = s + w
    patch = data[s[1]:e[1], s[0]:e[0], 3:]
    # Save stats
    C, rms, residuals = fitplane_m(patch)
    # Create residual array
    heatmat = np.copy(residuals)
    cmap = colors.ListedColormap(['red', 'yellow', 'green', 'blue'])
    bounds=[np.min(residuals), -rms, 0, rms, np.max(residuals)]
    norm = colors.BoundaryNorm(bounds, cmap.N)
    # plot residual color map
    error = plt.figure(num="Errors")
    img = plt.imshow(residuals, interpolation='nearest', origin='upper',
      cmap=cmap, norm=norm)
    plt.colorbar(img, cmap=cmap, norm=norm, boundaries=bounds,
      ticks=[np.min(residuals), -rms, 0, rms, np.max(residuals)])
    error.savefig("demo_results/res_"+plotname+str(i)+".png")
    # Save datapoint
    c = s + (w/2.0)
    datapoint = np.array([c[0], c[1], np.sqrt((w[0]/2.0)**2+(w[1]/2.0)**2), rms])
    results.append(datapoint)
    print "{}_{} / Size {} / RMS {} / {} outliers".format(plotname, i, width,
      rms, np.sum(residuals[:]>5*rms))
    # Draw box
    cv2.rectangle(frame, tuple(s), tuple(s + w), (255,0,0), 2)
    cv2.circle(frame, tuple(s+(w/2)), 3, (255,0,0), -1)
last = frame
img = np.c_[frame,conv1,conv2]
ret, jpeg = cv2.imencode('.jpg', frame)
```

```
#frame = serial.to_bytes(jpeg) # COMMENT OUT ON DICE
frame = jpeg.tobytes() # COMMENT OUT ON LOCAL MACHINE
yield (b'--frame\r\n'b'Content-Type: image/jpeg\r\n\r\n' + frame + b'\r\n\r\n')
cv2.imwrite("demo_results/"+plotname+str(i)+".jpg",last)
data = np.stack(results)
np.save("results/results_"+plotname+".npy",data)
@application.route('/<plotname>')
def run(plotname):
return Response(show(plotname), mimetype='multipart/x-mixed-replace; boundary=frame')
if __name__ == "__main__":
application.run(debug=True, host='0.0.0.0', port=3000)
```

#### 5.2 fetch.py

```
import scipy.io
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import glob
import re
def load_mesh(name):
    for frame in sorted(glob.glob("data/*.mat")):
        if re.sub("^[^\/]+\/|_[0-9]*\.mat", "", frame) == name:
            mat = scipy.io.loadmat(frame)
            XYZ = np.asarray(mat['XYZ'])
            Img = np.asarray(mat['Img'])
            m = XYZ.shape[0]
            n = XYZ.shape[1]
            RGBXYZ = np.zeros((m,n,6))
            X = XYZ[:,:,0]
            Y = XYZ[:,:,1]
            Z = XYZ[:,:,2]
            K = np.isnan(X) | np.isinf(X) | np.isnan(Y) | np.isinf(Y) | np.isnan(Z)
              | np.isinf(Z)
            X[K] = 0
            Y[K] = 0
            Z[K] = 0
            RGBXYZ[:, :, 0] = Img[:,:]
            RGBXYZ[:, :, 1] = Img[:,:]
            RGBXYZ[:, :, 2] = Img[:,:]
            RGBXYZ[:, :, 3] = X
            RGBXYZ[:, :, 4] = Y
            RGBXYZ[:, :, 5] = Z
```

#### yield RGBXYZ

```
5.3 match.py
```

```
import cv2
import numpy as np
from scipy.ndimage import map_coordinates
def phase1(src1,src2, thresh):
    cv2.imwrite("results/1.jpg",src1)
    cv2.imwrite("results/2.jpg",src2)
    src1 = np.abs(cv2.imread("results/1.jpg", 0)-255)
    src2 = np.abs(cv2.imread("results/2.jpg", 0)-255)
    src1[thresh(src1)] = 0
    src2[thresh(src2)] = 0
    src1 = np.float32(src1)
    src2 = np.float32(src2)
    #p = cv2.phaseCorrelate(src1,src2) # COMMENT OUT ON DICE
    p, error = cv2.phaseCorrelate(src1,src2) # COMMENT OUT ON LOCAL MACHINE
    r = np.array([p[0],p[1]]).astype("int")
    return r, src2
def fitplane_m(patch):
    surface = np.copy(patch)
    X_p = surface[:, :, 0]
    Y_p = surface[:, :, 1]
    Z_p = surface[:, :, 2]
    patch = patch.reshape(-1,3)
    size = patch.shape[0]
    A = np.c_[patch[:,0], patch[:,1], np.ones(size)]
    C, r, rank, s = np.linalg.lstsq(A,patch[:,2])
    Z = C[0] * X_p + C[1] * Y_p + C[2]
    Xr = X_p - X_p
    Yr = Y_p - Y_p
    Zr = Z_p - Z
    residuals = (-C[0] / C[2]) * Xr + (-C[1] / C[2]) * Yr + (1 / C[2]) * Zr
    rms = np.sqrt(np.average(np.square(residuals)))
    return C, rms, residuals
def index_coords(data, origin=None):
    """Creates x \ \theta y coords for the indicies in a numpy array "data".
    "origin" defaults to the center of the image. Specify origin=(0,0)
    to set the origin to the lower left corner of the image."""
```

```
ny, nx = data.shape[:2]
    if origin is None:
        origin_x, origin_y = nx // 2, ny // 2
    else:
       origin_x, origin_y = origin
    x, y = np.meshgrid(np.arange(nx), np.arange(ny))
    x -= origin_x
    y _= origin_y
    return x, y
def cart2polar(x, y):
    r = np.sqrt(x**2 + y**2)
    theta = np.arctan2(y, x)
    return r, theta
def polar2cart(r, theta):
   x = r * np.cos(theta)
    y = r * np.sin(theta)
    return x, y
def reproject_image_into_polar(data, origin):
   ny, nx = data.shape[:2]
    if origin is None:
        origin = (nx//2, ny//2)
    \# Determine that the min and max r and theta coords
    x, y = index_coords(data, origin=origin)
    r, theta = cart2polar(x, y)
    # Make a regular (in polar space) grid based on the min and max r arepsilon theta
    r_i = np.linspace(r.min(), r.max(), nx)
    theta_i = np.linspace(theta.min(), theta.max(), ny)
    theta_grid, r_grid = np.meshgrid(theta_i, r_i)
    # Project the r and theta grid back into pixel coordinates
    xi, yi = polar2cart(r_grid, theta_grid)
    xi += origin[0] # We need to shift the origin back to
    yi += origin[1] # back to the lower-left corner...
    xi, yi = xi flatten(), yi flatten()
    coords = np.vstack((xi, yi)) # (map_coordinates requires a 2xn array)
    # Reproject each band individually and the restack
    bands = []
    for band in data.T:
        zi = map_coordinates(band, coords, order=1)
        bands.append(zi.reshape((nx, ny)))
    output = np.dstack(bands)
    return output, r_i, theta_i
```

#### 5.4 plot.py

```
import matplotlib.pyplot as plt
import numpy as np
datax = np.load("results/results_planex.npy")
imgx = plt.figure(num="RMS vs X scene position")
plt.scatter(datax[:,0], datax[:,-1])
plt.ylim([0, np.max(datax[:,-1])])
imgx.show()
datay = np.load("results/results_planey.npy")
imgy = plt.figure(num="RMS vs Y scene position")
plt.scatter(datay[:,1], datay[:,-1])
plt.ylim([0, np.max(datay[:,-1])])
imgy.show()
dataz = np.load("results/results_planez.npy")
imgz = plt.figure(num="RMS vs average Z")
plt.scatter(dataz[:,0], dataz[:,-1])
plt.ylim([0, np.max(dataz[:,-1])])
imgz.show()
data = np.load("results/results_plane.npy")
img = plt.figure(num="RMS vs average Z - static")
plt.scatter(data[:,0], data[:,-1])
plt.ylim([0, np.max(data[:,-1])])
img.show()
plt.show(block=True)
5.5
     draw single.py
import scipy.io
import cv2
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from lib.match import fitplane_m
from matplotlib import colors
plt.close("all")
s = np.array([400, 200]) \# top left of patch
w = np.array([500, 500]) \# col and row count of patch
e = s + w
                         # bottom right of patch
mat = scipy.io.loadmat('data/planez_16.mat')
# create data array
XYZ = np.asarray(mat['XYZ'])
```

```
Img = np.asarray(mat['Img'])
m = XYZ.shape[0]
n = XYZ.shape[1]
data = np.zeros((m,n,6))
X = XYZ[:,:,0]
Y = XYZ[:,:,1]
Z = XYZ[:,:,2]
K = np.isnan(X) | np.isinf(X) | np.isnan(Y) | np.isinf(Y) | np.isnan(Z) | np.isinf(Z)
X[K] = 0
Y[K] = 0
Z[K] = 0
data[:, :, 0] = Img[:,:]
data[:, :, 1] = Img[:,:]
data[:, :, 2] = Img[:,:]
data[:, :, 3] = X
data[:, :, 4] = Y
data[:, :, 5] = Z
# data done
# grey scale image anyway so just call first indices
frame = data[:, :, 0] * 255.0
# add patch box
cv2.rectangle(frame, tuple(s), tuple(e), (0, 0, 255), 5)
# plot the base image + patch box
img = plt.figure(num="Base image")
plt.imshow(frame, cmap = 'gray')
plt.xlabel('X')
plt.ylabel('Y')
img.show()
# get X and Y coords for mesh drawing
X_m = np.arange(m)
Y_m = np.arange(n)
X_m, Y_m = np.meshgrid(X_m, Y_m)
...
# plot 3D mesh
mesh = plt.figure(num="Entire 3D plot")
ax = mesh.gca(projection='3d')
ax.plot_surface(X_m, Y_m, Z, linewidth=0.2, antialiased=True)
plt.xlabel('X')
plt.ylabel('Y')
ax.set_zlabel('Z')
mesh.show()
...
# extract patch points
patch = data[s[1]:e[1],s[0]:e[0],3:]
X_p = patch[:,:,0]
```

```
Y_p = patch[:,:,1]
Z_p = patch[:,:,2]
# plot X component
imgx = plt.figure(num="Patch X component")
plt.imshow(X_p, cmap = 'gray')
plt.xlabel('X')
plt.ylabel('Y')
plt.colorbar()
imgx.show()
# plot Y component
imgy = plt.figure(num="Patch Y component")
plt.imshow(Y_p, cmap = 'gray')
plt.xlabel('X')
plt.ylabel('Y')
plt.colorbar()
imgy.show()
# plot Z component
imgz = plt.figure(num="Patch Z component")
plt.imshow(Z_p, cmap = 'gray')
plt.xlabel('X')
plt.ylabel('Y')
plt.colorbar()
imgz.show()
# get X and Y coords for patch mesh drawing
X_m = np.arange(s[0],e[0])
Y_m = np.arange(s[1],e[1])
X_m, Y_m = np.meshgrid(X_m, Y_m)
...
# plot 3D patch
patch_mesh = plt.figure(num="Patch 3D plot")
ax = patch_mesh.gca(projection='3d')
ax.plot_surface(X_m, Y_m, Z_p, linewidth=0.2, antialiased=True)
plt.xlabel('X')
plt.ylabel('Y')
ax.set_zlabel('Z')
patch_mesh.show()
111
# calculate plane equation, rms, residuals
C, rms, residuals = fitplane_m(patch)
Z = C[0] * X_p + C[1] * Y_p + C[2]
print "RMS is {}".format(rms)
...
# plot fitted plane
plane = plt.figure(num="Fitted plane")
```

```
ax = plane.gca(projection='3d')
ax.plot_surface(X_m, Y_m, Z, linewidth=0.2, antialiased=True)
plt.xlabel('X')
plt.ylabel('Y')
ax.set_zlabel('Z')
plane.show()
'''
```

```
# build the color map for residuals
heatmat = np.copy(residuals)
cmap = colors.ListedColormap(['red','yellow','green','blue'])
bounds=[np.min(residuals),-rms,0,rms,np.max(residuals)]
norm = colors.BoundaryNorm(bounds, cmap.N)
```

plt.show(block=True)