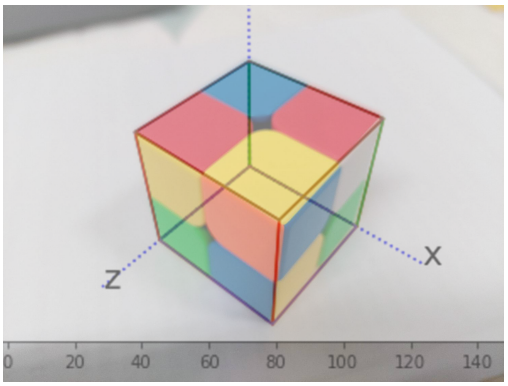
Linear Algebra Assignment 2

Anastasia Riana 邱慧莉 109006234

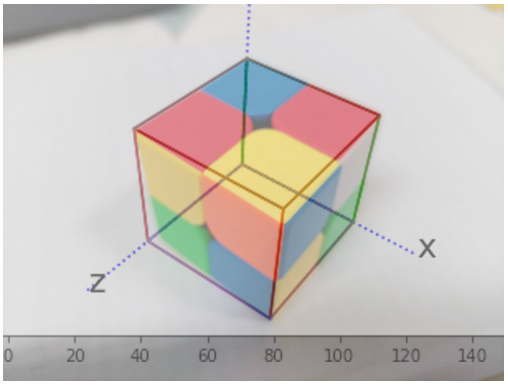
1.

- Using the 2nd to 12th equations from the 14x11 matrix



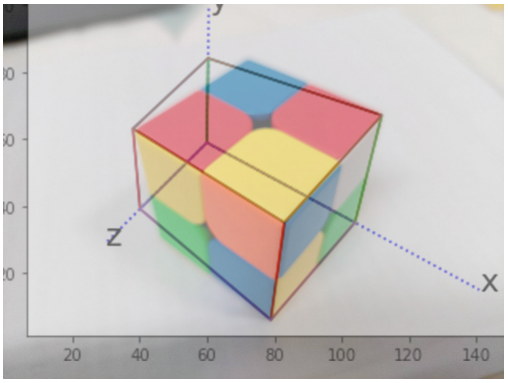
Using these selection of equations, it yields a quite accurate model of the cube, without distortion. As we can see, the points are all aligned and matched except for the point on the top left of the right plane. It should be placed slightly outwards to the right about 1-2 pixels compared to the 1st scenario. The top right point of the right plane also should be placed inwards for about 1-2 pixels.

- Using the 3rd to 13th equations from the 14x11 matrix



Using these selection of equations, it yields a slightly deformed model of the cube. The points are all aligned and matched except for the left visible plane. The top right point of the left plane should be a bit downwards and the bottom left point should extend outwards a bit. The top left corner of the right plane should also extend outwards.

- Using the 4th to 14th equations from the 14x11 matrix



Using these selection of equations, it yields a very deformed model of the cube. We only have a few points that are aligned and matched. The model yields more of a cuboid model instead of a cube model.

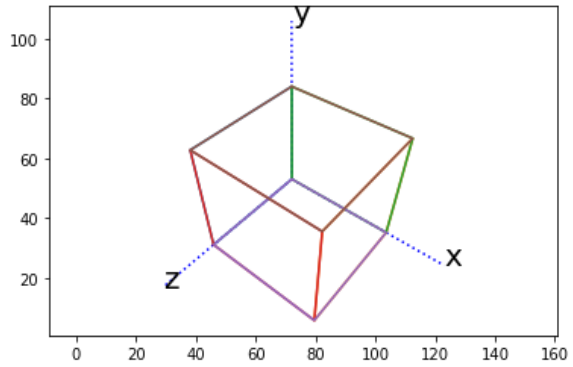
Conclusion: The selection of the 2nd to 13th equations yields the most accurate results.

2.

In this case, we have an overdetermined system that is a 14x11 matrix. If we use ***numpy.linalg.solve***, we have to take into consideration which 11 equations that we are going to use to solve the linear system. The system will yield a different result. These results can be a really accurate or distorted model of the cube.

Using ***numpy.linalg.lstsq,*** the algorithm will discreetly do row elimination. Then, after a lot of row elimination, the matrix will supposedly have 11 leading variables, which we can use to model the cube. The function will return the results in x array, residuals of matrix A, rank of matrix A and singular value of A. Using ***numpy.linalg.lstsq*** to solve this linear system will be more efficient (in terms of lines of code) because it can at least reduce two lines of code that we use to hand-pick 11 equations of matrix **A** and **B** that we are going to use.

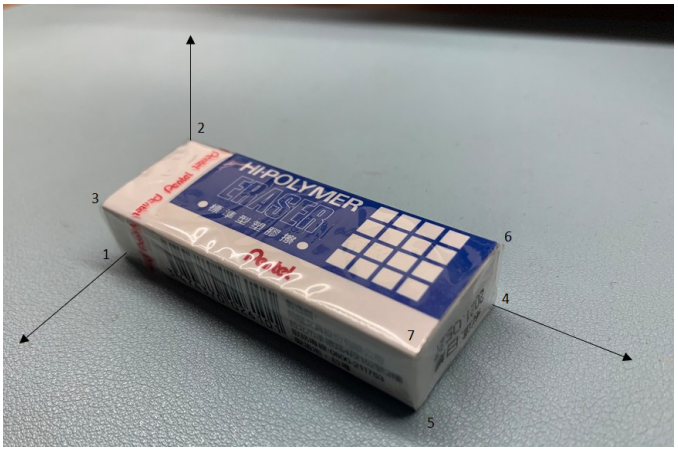
Result:



Compared to Q1 shown by the plot, the result of ***numpy.linalg.lstsq*** gives a pretty accurate model of the cube just like the first scenario of Q1.

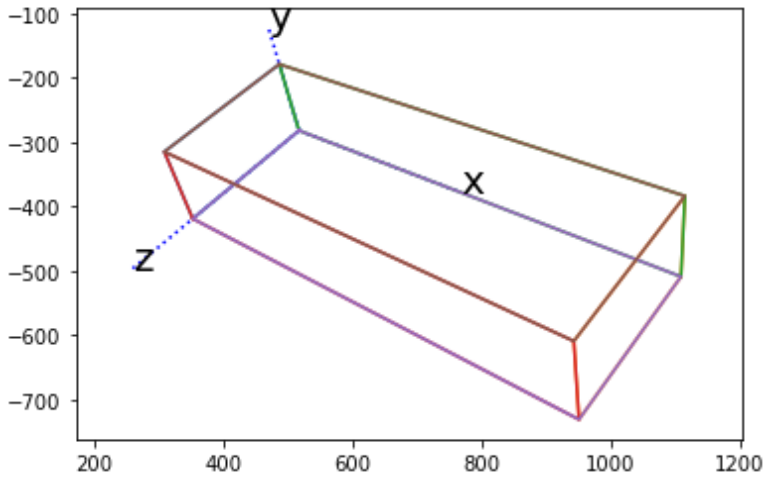
3.

Image and Coordinates:



Number	3D Point	2D Point
1	(0,0,1)	(353,519)
2	(0,1,0)	(487,279)
3	(0,1,1)	(309,415)
4	(3,0,0)	(1109,609)
5	(3,0,1)	(951,831)
6	(3,1,0)	(1115,491)
7	(3,1,1)	(947,715)

Result:



Explanation:

The object is an eraser and we set the height of the eraser to be 1 unit, width to be 1 unit and the length to be 3 units. To find the 2D coordinates of the points of the eraser, we use Microsoft Paint. As you can see from the result, the algorithm is correct and we don't really see that much distortion in the plot. We can conclude that the projection matrix of the linear system is accurate.

4.

From this article, we have to first create a matrix P from 3D given points. Just like the description of the homework, for each point in the model, will generate two equations. The general equation that we will have is: P = C[R|T].

The matrix P is also called the camera matrix, where it's going to map the 3D points to a 2D plane. The way the camera calibration works is that we need to convert the 3D points given (3D), to a camera coordinates (3D) through extrinsic parameter. Then, the camera coordinates will be transformed again to pixel coordinates (2D) through intrinsic parameters.

For the extrinsic parameter, we use the QR matrix decomposition to find R (rotation matrix) and T(translation matrix). From this, we can find the origin of the camera's coordinate system at its optical center and its coordinates will define the image plane. For the intrinsic parameter, we need to find matrix C shown below, where $\mathbf{s = f_x \tan \alpha}$, $\mathbf{f_x = F/p_x}$, $\mathbf{f_y = F/p_y}$

$$\begin{bmatrix} f_x & s & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

Note: f_x , f_y = focal length in pixels ; c_x , c_y = optical center in pixels ; **F** = focal length in world units

After we compose P, we use it to get, $[X,Y,1]^T = P\ [X,Y,Z,1]^T$. There we will get the exact location of the camera.

5.

Source: https://www.geeksforgeeks.org/displaying-the-coordinates-of-the-points-clicked-on-the-image-using-python-opencv/

In this problem, we utilize the OpenCV library, which is mostly used for image processing and performing computer vision tasks. First thing that we have to do is to install the OpenCV package for Python. To do that, we need to type ***pip install opencv-python*** and launch it. In this problem, we can find the 2D coordinates from an image from a mouse click. Hence, it is required to have an image that is going to be loaded to the program in the same folder as the .ipynb file (Jupyter Notebook file).

Firstly, we need to create a function that we are going to call when there's mouse activity called **click** function. In that function, if we detect either a left or right mouse click, it will print the 2D coordinates of the points in the terminal (See Figure 1). The location where we click a specific point will be indicated by a green point. The model is going to open in another window and when we click a point, on that image too, will show the coordinates (See Figure 2).

```
(x,y): (726,166)
(x,y): (586,254)
(x,y): (897,237)
(x,y): (1109,326)
```

Figure 1

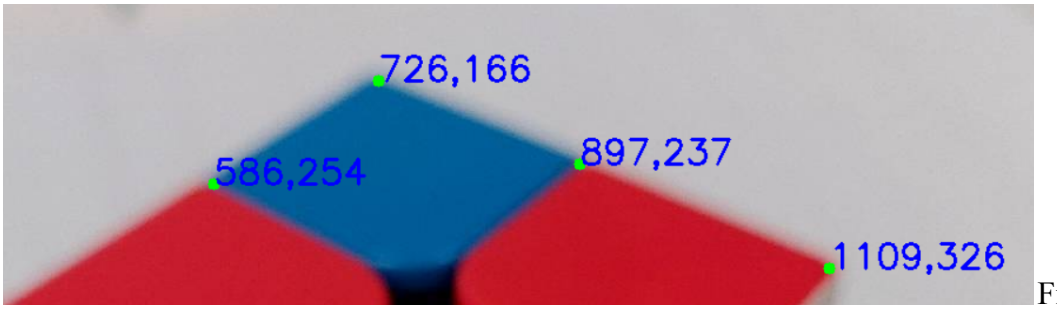


Figure 2

In the driver function, we need to load the model which is the cube image from Q1 and Q2, with ***cv2.imread('cube.jpg')*** and we display the image using ***cv2.imshow***. After that we can initialize the mouse event using ***cv2.setMouseCallback*** where in its parameter, it will call the **click** function we mentioned before. To terminate, we can just press any key, indicated by ***cv2.waitKey(0)*** and close the window using ***cv2.destroyAllWindows().*** We avoid closing the window using the X button on the top right corner, because it sometimes will interrupt the kernel.

Comparing the result of OpenCV to other software like Microsoft Paint, in Figure 2, we see that the points are 892 and 237 for x-axis and y-axis respectively. Using Microsoft Paint, we also get the same results, though it is quite challenging to get the exact position of the pixels using the mouse.