HW 2

陳翰雯 - B08902092

Q1-1. PnP algorithm

The algorithm chosen was similar to OpenCV's solvePnPRansac. It is like a python implementation of the original `p3p.cpp` by OpenCV. However, the achieved results is still not as good as the original OpenCV's.

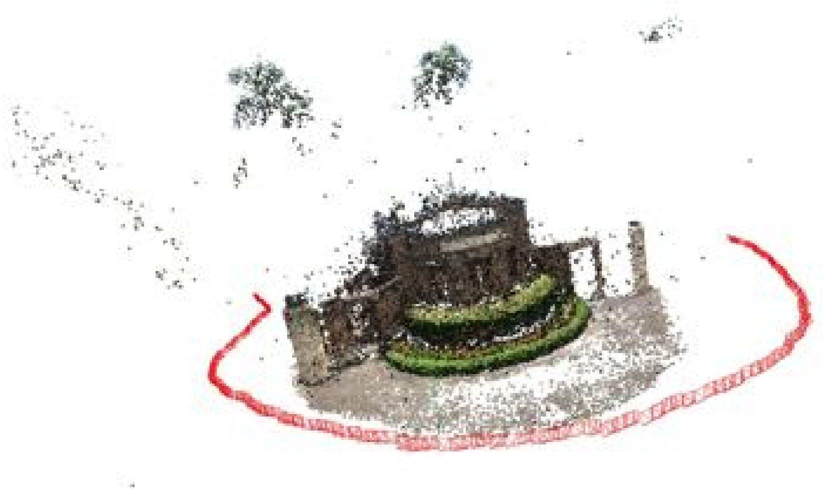

Fig 1. OpenCV's cv2.solvePnPRansac algorithm

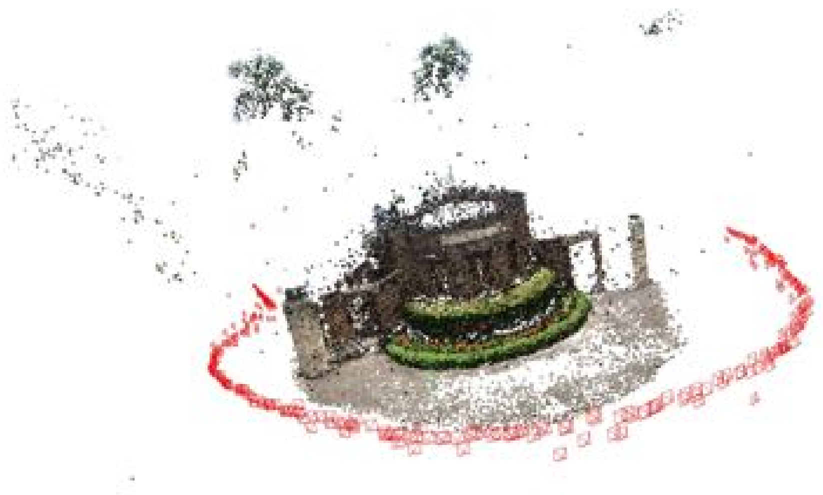

Fig 2. my P3P + RANSAC algorithm

From both of the images above, we could see that OpenCV's has a more neat placement of the cameras. In the case of my own algorithm, even though it has succeeded in gaining the circular pattern, we could see that some of the camera's pose are too front or too far back.

The main algorithm used here is based on the paper "Complete Solution Classification for the Perspective-Three-Point Problem.

Instead of only taking 3 3D and 2D points, this algorithm takes 4 3D and 2D points. Hence the algorithm could also be called as P4P algorithm. The algorithm is pretty much the same with the conventional P3P algorithm. In P4P, after solving for the real roots of the 3 points and obtaining the different rotation and translation matrices, we use the 4th points to calculate the projection errors. The rotation and translation matrix that gives the lowest projection error will be the final answer.

```
# The obtained lengths from obtained real roots
for length in lengths:
    M_orig = np.tile(length,3).reshape(3,3).T
    M_orig = M_orig * undistorted_points

    # calculating the rotation and translation matrix
    R, T = self.align(M_orig, X0, Y0, Z0, X1, Y1, Z1, X2, Y2, Z2)

    # calculating reprojection error from the 4th point
    XYZ3p = np.dot(R, np.array([X3,Y3,Z3])) + T
    XYZ3p = XYZ3p / XYZ3p[2]
    mu3p, mv3p = XYZ3p[:2]

    reproj_error = (mu3p - mu3)**2 + (mv3p - mv3)**2
    reproj_errors.append(reproj_error)
    Rs.append(R)
    Ts.append(T)

reproj_errors = np.array(reproj_errors)
Rs = np.array(Rs)
Ts = np.array(Ts)

# sorting the matrices (from lowest to highest reprojection error)
sorted_idx = np.argsort(reproj_errors)
sorted_Rs = Rs[sorted_idx]
sorted_Ts = Ts[sorted_idx]

# takin the matrix with lowest reprojection error
best_Rs = sorted_Rs[0]
best_Ts = sorted_Ts[0]
```

For the RANSAC algorithm, we use projection error with L2 norm as our main grading metric system. By default, we do a total of 100 iterations. On each iterations 4 random 3D and 2D points are chosen. These chosen points then undergo the P3P algorithm as described above. We then obtain the rotation and translation vectors. For every obtained rotation and translation vectors, we compute the re-projection error as follows:

```
# to get the projected points from the computed rvec and tvec
project_points,_ = cv2.projectPoints(points3D, rvec, tvec, cameraMatrix, distCoeffs)

for i, project_point in enumerate(project_points):
    pixel_point = points2D[i]

    # computing the error using L2 norm
    total_error += np.linalg.norm(pixel_point - project_point)
```

At last, the 4 random points that yields lowest total error will then be chosen.

Few other algorithms are also tried beforehand, such as DLT and conventional P3P method, but the results was not the best.

For DLT, we calculate its homography from corresponding 3D and 2D points through the use of SVD, similar to what we did on homework1. In the case of conventional P3P, we again try to find the 4th order quartic polynomial by the means of finding its distances and cosine equations. This polynomial equations then give us up to 4 real solutions. These solutions are then used to calculate the rotation and translation matrix.

The results of these 2 algorithms is as follows:

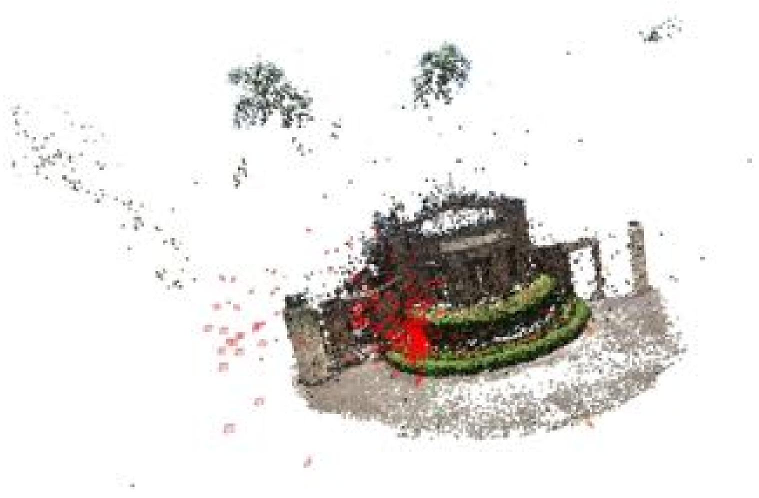
Fig 3. DLT algorithm

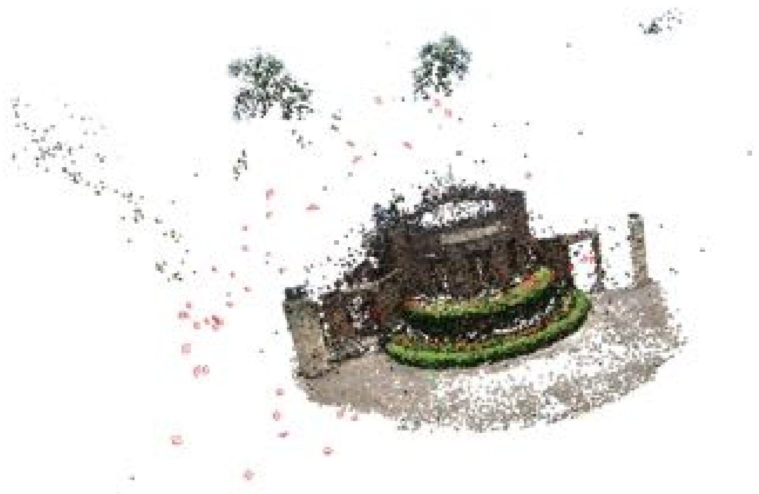
Fig 4. conventional P3P algorithm

In both of the images, we could see that the camera are all over the places. In the result of DLT algorithm there are even an area where the cameras are clustered together. This somehow tells us that the DLT and P3P algorithm used here is not right or it is wrongly implemented.

Q1-2. Median Pose Error

The median pose error of translation and rotation is implemented as follows:

```
# rotq_gt and tvec_gt are the ground truth camera pose
def differences(rotq, tvec, rotq_gt, tvec_gt):
  nor_rotq = rotq / np.linalg.norm(rotq)
  nor_rotq_gt = rotq_gt / np.linalg.norm(rotq)
  dif_r = np.clip(np.sum(nor_rotq * nor_rotq_gt), 0, 1)

  d_r = np.degrees(np.arccos(2 * dif_r * dif_r - 1))
  d_t = np.linalg.norm(tvec-tvec_gt, 2)

  differences_r.append(d_r)
  differences_t.append(d_t)

# median pose error
err_r = np.median(differences_r)
err_t = np.median(differences_t)
```

From the 3 algorithms that is mentioned above, below are the obtained median pose error:

1. **P3P + RANSAC:**

   pose error: 0.02988279282462082, rotation error: 0.7785275549845175

2. **DLT:**

   pose error: 4.499411404932163, rotation error: 103.9864471744359

3. **Conventional P3P:**

   pose error: 4.50948783830821, rotation error: 109.21544155577249

Q1-3. Plotting camera poses

We first find the camera poses of each image and save it in `camera_position.pkl`. This saves the rotation and translation matrix of each camera in each image.

We draw the camera as quadrangular pyramid with the use of `o3d.geometry.LineSet()`, `o3d.utility.Vector3dVector`, `o3d.utility.Vector2iVector`. Then, we use the following to find the camera's position:

```
model.rotate(r)
model.translate(p)
# where r is the rotation matrix and p is the translation matrix
```

Q2-1. Virtual cube

To draw the virtual cube, I use the OpenCV's `cv2.solvePnPRansac` to obtain the camera's rotation and translation matrix. This is because OpenCV's algorithm produces more accurate matrices rather than own's algorithm.

To do this, we first obtain the cube's vertices from the file `cube_vertices.npy`. Once the vertices are obtained, we then start to find each point's relative position. This is implemented as follows:

```
# amt is the number of dots for each face of the cube.
# amt = 10 means that for each face there will be a total of 10x10 dots
amt = 10
for x in range(amt):
        for y in range(amt):
            point = origin + x/amt * width + y/amt * height
```

The points are then drawn to the image with the following code:

```
pos = transform_matrix@np.array(point)
pos /= pos[2]
x,y = int(pos[0]), int(pos[1])
cv2.circle(rimg, (x,y), 5, colors[i], -1)
```

Below are few of the results:

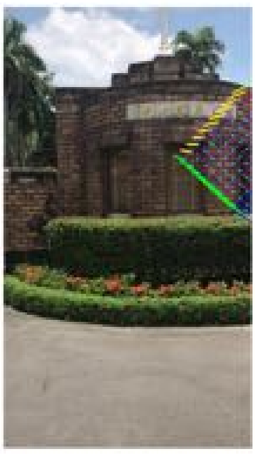
Fig 5. 194.jpg

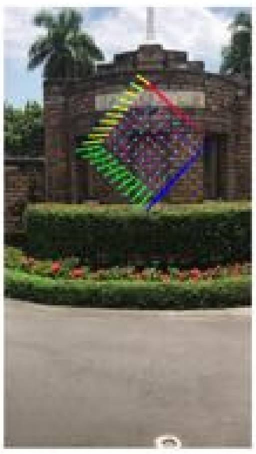
Fig 6. 49.jpg

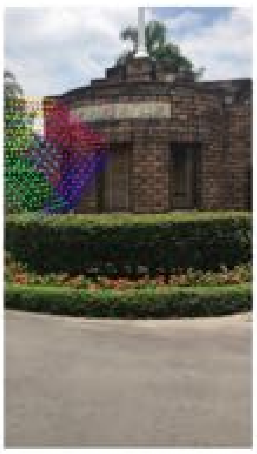
Fig 7. 23.jpg

How to execute:

Problem 1:

`python 2d3dmathcing.py`

This will produce “camera_position.pkl”

`python transform_cube.py`

This will draw the camera poses.

Problem 2:

`python draw_3dcube.py`

This will generate the cube in each images from 1.jpg to 293.jpg. The resulted image will be saved in the folder “cube_images.jpg”.

(Make sure to make the folder first before running the code)

Environment

Python = 3.9.9