HW 3

陳翰雯 - B08902092

Step 1: Camera Calibration

The camera intrinsic matrix used is:

$$\begin{bmatrix} 523.88889709 & 0 & 316.69443537 \\ 0 & 525.14246659 & 182.51162748 \\ 0 & 0 & 1 \end{bmatrix}$$

The distortion coefficients are:

$$\begin{bmatrix} 1.08010860^{-1} & -7.89853522^{-1} & -1.90262717^{-3} & 5.26271916^{-4} & 1.44662578 \end{bmatrix}$$

Step 2: Feature Matching

As recommended, ORB is used as a feature extractor. Along with it, we compute the Hamming distance with the help of `cv.NORM_HAMMING`

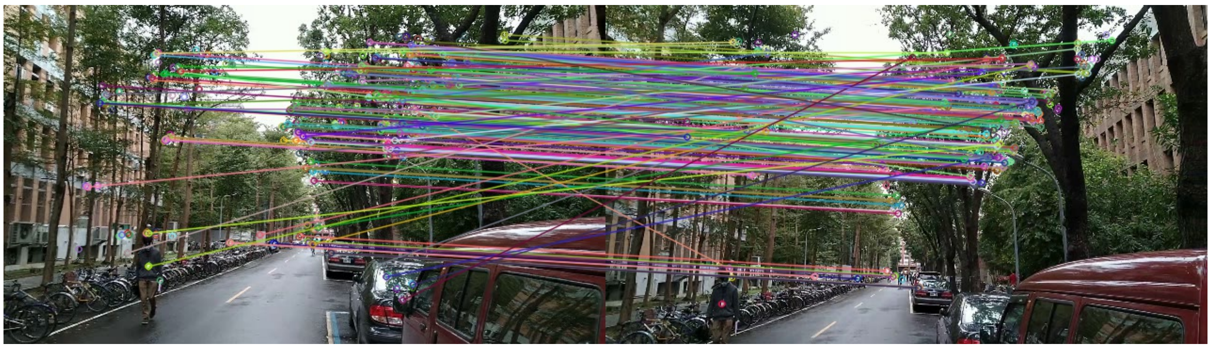
The feature matching is implemented as follows:

```
def extract_features(self, img1, img2):
        orb = cv.ORB_create()
        kp1, des1 = orb.detectAndCompute(img1, None)
        kp2, des2 = orb.detectAndCompute(img2, None)

        bf = cv.BFMatcher(cv.NORM_HAMMING, crossCheck=True)
        matches = bf.match(des1, des2)
        matches = sorted(matches, key = lambda x:x.distance)

        points1 = np.array([kp1[m.queryIdx].pt for m in matches])
        points2 = np.array([kp2[m.trainIdx].pt for m in matches])

        return points1, points2
```



Step 3: Pose from Epipolar Geometry

Following according to the steps in the slides, the code is implemented as follows:

`img1` and `img2` could be denoted as img_k and img_{k+1} respectively

```
for i, frame_path in enumerate(self.frame_paths[1:]):
  img1 = cv.imread(self.frame_paths[i])
  img2 = cv.imread(frame_path)

  # Extract features between I_k+1 and I_k
  points1, points2 = self.extract_features(img1,img2)

  # Estimate the essential matrix E
  E, mask = cv.findEssentialMat(points1, points2, self.K, method=cv.RANSAC, prob=0.999, threshold=1.0)

  # Decompose E to get relative pose
  _, R, t, mask, triangulated = cv.recoverPose(E, points1, points2, self.K, mask=mask, distanceThresh=500)
```

Once we got the relative pose (R and t), we then do the rescaling process

The rescaling process is done as follows:

Let's say we have 3 points from img_{k-1}, img_k and img_{k+1} respectively, we first calculate triangulations of img_{k-1} and img_k and then the triangulations of img_k and img_{k+1} which is denoted as `prev_triangulated` and `triangulated` in the code.

From the given 2 scene points, we can then now calculate the ratio with the following implementation:

In this case `np.linalg.norm` is used to calculate the distance

```
def calculate_scale(self, prev_triangulated, triangulated):
    prev_new = np.roll(prev_triangulated, shift=-5)
    curr_new = np.roll(triangulated, shift=-5)
    ratios = (np.linalg.norm(prev_3d - prev_new)) / (np.linalg.norm(curr_3d - curr_new))

    ratio = np.median(ratios)
    return ratio
```

Once we have the ratio we then calculate our final pose as follows:

```
current_pos += current_rot.dot(t) * ratio
current_rot = R.dot(current_rot)
```

Step 4: Results Visualization

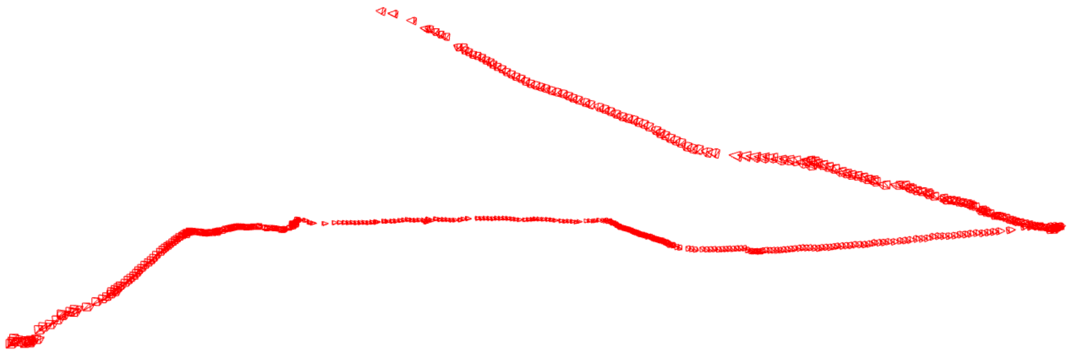
Here, for each camera pose, we visualize it as follows:

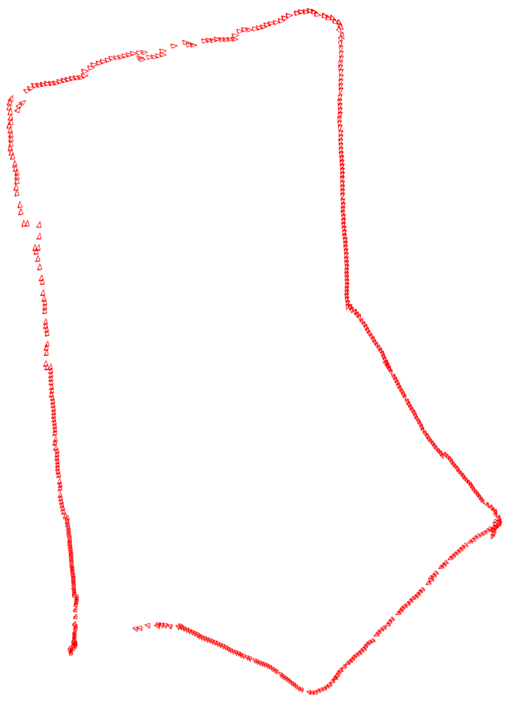
```
def load_camera(self, t, R):
    points = np.array([[0, 0, 1], [360, 0, 1], [360, 360, 1], [0, 360, 1]])
    points = np.linalg.pinv(self.K) @ points.T
    points3D = t + R @ (points)
    points3D = points3D.T
    points3D = np.concatenate((points3D, t.T), axis=0)

    model = o3d.geometry.LineSet()
    model.points = o3d.utility.Vector3dVector(points3D)
    model.lines = o3d.utility.Vector2iVector([[0, 1], [0, 2], [0, 3], [0, 4], [1, 2], [2, 3], [3, 4], [4, 1]])

    color = np.array([1, 0, 0])
    model.colors = o3d.utility.Vector3dVector(np.tile(color, (8, 1)))
```

The obtained camera trajectory is as follows:





The video demo of the whole process could be seen in the link below:

https://drive.google.com/file/d/1x9i1HkloyBivgT0cGasejdrTX7O4Xl4W/view?usp=sharing

How to execute:

To run the program, simply execute the following code:

```
python vo.py frames
```

Environment

Python = 3.8