3DCV Hw2

R11944014 戴靖婷

tags: `3DCV` `Python` `Report`

Problem 1: Camera Relocalization

- Settings:
  - Execute codes:
    `python .\2d3dmathcing.py`
  - Package:
    - scipy.spatial.transform.Rotation, pandas, numpy, cv2, open3d

Q1-1: P3P + RANSAC

- Code Explanation:
  - First, I made use of the sample code for averaging the descriptors to describe the same points, and obtaining the good matches of 2D and 3D correspondences.
  - Then I passed the points to the function **ransac**.
    - Randomly choose three corresponding points of 2D and 3D and solve P3P by the function P3P.
    - Iterate each rotation and translation matrices generated from P3P, fit all the points to this model, and find the best model until convergence.

```python
# Randomly choose 3 points and count inliers for num_iter times
def ransac(points3D, points2D, cameraMatrix, distCoeffs, smallest_points: int, num_iter: int, threshold: float):
    """...
    max_inlier = 0
    best_rot, best_t = np.array([]), np.array([])

    # At least iterate num_iter times
    for _ in range(num_iter):
        rand_idx = np.random.choice(len(points2D), smallest_points, replace=False)
        sample_points3D, sample_points2D = points3D[rand_idx], points2D[rand_idx]

        # Find the rotation and translation matrices by the three sampled points through solveP3P
        rotms, tvecs = P3P(sample_points3D, sample_points2D, cameraMatrix, distCoeffs)

        # Iterate each rotation and translation matrix
        for rotm, tvec in zip(rotms, tvecs):
            tvec = tvec.reshape(3, 1)
            # Count the inliers of this model for all points
            projection_mat = np.matmul(cameraMatrix, np.hstack((rotm, tvec)))  # size: 3 x 4 from 3x3 * 3x4
            ones = np.ones(len(points3D))
            homo_points3D = np.vstack((points3D.T, ones))                       # size: 4 x n_points
            est_homo_points2D = np.matmul(projection_mat, homo_points3D)        # size: 3 x n_points

            est_points2D = est_homo_points2D / est_homo_points2D[-1, :]         # size: 3 x n_points
            est_points2D = est_points2D[:-1, :].T                               # size: n_points x 2

            errors = np.linalg.norm((points2D - est_points2D), axis=1)
            num_inlier = np.count_nonzero(errors < threshold)

            # Save the best model of rotation and translation matrices
            if num_inlier > max_inlier:
                max_inlier = num_inlier
                best_rot = rotm
                best_t = tvec.reshape(3)

    best_rotq = R.from_matrix(best_rot).as_quat()

    return best_rotq, best_t
```

- RANSAC from the slide: 9_RANSAC

Algorithm 15.4: RANSAC: fitting lines using random sample consensus

Determine:
- **s** — the smallest number of points required
- **N** — the number of iterations required
- **d** — the threshold used to identify a point that fits well
- **T** — the number of nearby points required to assert a model fits well

Until **N** iterations have occurred
  Draw a sample of **s** points from the data uniformly and at random
  Fit to that set of **s** points
  For each data point outside the sample
    Test the distance from the point to the line against **d** if the distance from the point to the line is less than **d** the point is close
  end
  If there are **T** or more points close to the line then there is a good fit. Refit the line using all these points.
end
Use the best fit from this collection, using the fitting error as a criterion

- In the function **P3P**, it solves and return possible extrinsic matrix.

- First of all, undistort the points by camera matrix and distort coefficients.
- Solving the camera projection matrix:
  - $\lambda_i u_i = KR[I| -T]X_i$.
- $v_i = K^{-1}u_i$: non-homogeneous 2D points
- R_{ij}: distance from i to j
- C_{ij}: angle cosine of (i, j)
- $K_1 = (R_{bc}/R_{ac})^2$, $K_2 = (R_{bc}/R_{ab})^2$
- G4,G3,G2,G1,G0: from slide 8_Camera_Pose_Estimation_P3P

  After regrouping terms, we obtain that this is a quartic polynomial in x,

  $$0 = G_4x^4 + G_3x^3 + G_2x^2 + G_1x + G_0 \tag{12}$$

  where
  $$\begin{aligned}
  G_4 &= (K_1K_2 - K_1 - K_2)^2 \\
  &\quad - 4K_1K_2C_{bc}^2 \\
  G_3 &= 4(K_1K_2 - K_1 - K_2)K_2(1 - K_1)C_{ab} \\
  &\quad + 4K_1C_{bc}[(K_1K_2 - K_1 + K_2)C_{ac} + 2K_2C_{ab}C_{bc}] \\
  G_2 &= [2K_2(1 - K_1)C_{ab}]^2 \\
  &\quad + 2(K_1K_2 - K_1 - K_2)(K_1K_2 + K_1 - K_2) \\
  &\quad + 4K_1[(K_1 - K_2)C_{bc}^2 + K_1(1 - K_2)C_{ac}^2 - 2(1 + K_1)K_2C_{ab}C_{ac}C_{bc}] \\
  G_1 &= 4(K_1K_2 + K_1 - K_2)K_2(1 - K_1)C_{ab} \\
  &\quad + 4K_1[(K_1K_2 - K_1 + K_2)C_{ac}C_{bc} + 2K_1K_2C_{ab}C_{ac}^2] \\
  G_0 &= (K_1K_2 + K_1 - K_2)^2 \\
  &\quad - 4K_1^2K_2C_{ac}^2
  \end{aligned}$$

  - Solve the equation by companion matrix.
- a,b,c: the distances between the apex and x_1, x_2, x_3
  - $a = +(R_{2ab}/(1 + x^2 - 2 * x * C_{ab}))^{1/2}$
  - $b = x * a$
  - $c = y * a$: from slide 8_Camera_Pose_Estimation_P3P

    $$\begin{aligned}
    0 &= (1 - K_1)y^2 + 2(K_1C_{ac} - xC_{bc})y + (x^2 - K_1) \\
    0 &= y^2 + 2(-xC_{bc})y + [x^2(1 - K_2) + 2xK_2C_{ab} - K_2]
    \end{aligned}$$
- Construct the 3D points in CCS.
  - $CCS_1 = (a * v_1)/norm(v_1)$, $CCS_2 = (b * v_2)/norm(v_2)$, $CCS_3 = (a * v_3)/norm(v_3)$

- Then, use CCS and WCS 3D points as input for ICP.

```python
# Find the three points in the camera coordinate system, and calculate the rotation and translation matrices by ICP
def P3P(points3D: np.ndarray, points2D: np.ndarray, cameraMatrix: np.ndarray, distCoeffs: np.array):
    """...

    # Undistort the points by camera matrix and distort coefficients
    points2D = cv2.undistortPoints(points2D, cameraMatrix, distCoeffs, None, cameraMatrix).reshape(-1, 2)

    # xi: non-homogeneous 3D points (x y z)
    x1, x2, x3 = points3D[0], points3D[1], points3D[2]

    # vi: non-homogeneous 2D points (x y), ui: homogeneous 2D points (x y 1)
    # vi = inv(K) * ui
    K_inv = np.linalg.inv(cameraMatrix)
    ones = np.ones((3, 1))
    U = np.append(points2D, ones, axis=1)
    v1, v2, v3 = np.matmul(K_inv, U[0]), np.matmul(K_inv, U[1]), np.matmul(K_inv, U[2])

    # Cij: angle cosine of (i, j)
    # Rij: distance from i to j
    Rab, Rac, Rbc = np.linalg.norm(x1 - x2), np.linalg.norm(x1 - x3), np.linalg.norm(x2 - x3)
    Cab = np.dot(v1, v2) / (np.linalg.norm(v1) * np.linalg.norm(v2))
    Cac = np.dot(v1, v3) / (np.linalg.norm(v1) * np.linalg.norm(v3))
    Cbc = np.dot(v2, v3) / (np.linalg.norm(v2) * np.linalg.norm(v3))

    K1, K2 = (Rbc / Rac) ** 2, (Rbc / Rab) ** 2

    # Quartic polynomial: 0 = G4 * x**4 + G3 * x**3 + G2 * x**2 + G1 * x + G0
    G4 = (K1 * K2 - K1 - K2) ** 2 - 4 * K1 * K2 * Cbc**2
    G3 = 4 * (K1 * K2 - K1 - K2) * K2 * (1 - K1) * Cab \
        + 4 * K1 * Cbc * ((K1 * K2 - K1 + K2) * Cac + 2 * K2 * Cab * Cbc)
    G2 = (2 * K2 * (1 - K1) * Cab) ** 2 \
        + 2 * (K1 * K2 - K1 - K2) * (K1 * K2 + K1 - K2) \
        + 4 * K1 * ((K1 - K2) * Cbc**2 + K1 * (1 - K2) * Cac**2 - 2 * (1 + K1) * K2 * Cab * Cac * Cbc)
    G1 = 4 * (K1 * K2 + K1 - K2) * K2 * (1 - K1) * Cab \
        + 4 * K1 * ((K1 * K2 - K1 + K2) * Cac * Cbc + 2 * K1 * K2 * Cab * Cac**2)
    G0 = (K1 * K2 + K1 - K2) ** 2 - 4 * K1**2 * K2 * Cac**2

    # Solving roots by companion matrix
    if G4 != 0:
        first_row = np.zeros((1, 3))
        identity_mat = np.identity(3)
        last_col = np.array([-G0 / G4, -G1 / G4, -G2 / G4, -G3 / G4]).reshape(-1, 1)
    elif G3 != 0:
        first_row = np.zeros((1, 2))
        identity_mat = np.identity(2)
        last_col = np.array([-G0 / G3, -G1 / G3, -G2 / G3]).reshape(-1, 1)
    else:
        return [], []
    companion_mat = np.vstack((first_row, identity_mat))
    companion_mat = np.hstack((companion_mat, last_col))
    roots, _ = np.linalg.eig(companion_mat)

    # By the real part of root x, compute the corresponding a, y, b, c
    x_list, rotms, tvecs = roots[np.isreal(roots)].real, [], []
    for x in x_list:
        a = np.sqrt( (Rab**2) / (1 + x**2 - 2 * x * Cab))

        m = 1 - K1
        p = 2 * (K1 * Cac - x * Cbc)
        q = x**2 - K1
        p_ = -2 * x * Cbc
        q_ = x**2 * (1 - K2) + 2 * x * K2 * Cab - K2

        y = -(q - m * q_) / (p - p_ * m)

        b = x * a
        c = y * a

        # a, b, c are distances, which is impossible to be negative numbers
        if a < 0 or b < 0 or c < 0:
            continue

        # 3 points in the camera coordinate system
        CCS_pnt1 = a * v1 / np.linalg.norm(v1)
        CCS_pnt2 = b * v2 / np.linalg.norm(v2)
        CCS_pnt3 = c * v3 / np.linalg.norm(v3)

        # Calculate and save the possible rotation and translation matrices
        rotm, tvec = ICP(np.array([CCS_pnt1, CCS_pnt2, CCS_pnt3]), points3D)

        rotms.append(rotm)
        tvecs.append(tvec)

    return rotms, tvecs
```

- To find the rigid transform between 3D points of CCS and WCS, use **ICP**.
  - Find the mean of the two sets.

- Translate the sets by each mean.
- Solve SVD of multiplication of WCS and CCS.
  - Let $W = \sum_{i=1}^{N_p} x_i' p_i'^T$

    denote the singular value decomposition (SVD) of W by:

    $$W = U \begin{bmatrix} \sigma_1 & 0 & 0 \\ 0 & \sigma_2 & 0 \\ 0 & 0 & \sigma_3 \end{bmatrix} V^T$$
  - from the slide: 10_Iterative_Solutions_for PnP & Intro_to_ICP
- Rotation matrix: $R = UV^T$.
- Translation vector: $t = mu_x - Rmu_p$.

```python
# Calculate the rotaion and translation matrices, whose projection is from world to the camera
def ICP(CCS_points: np.ndarray, WCS_points: np.ndarray):
    """...

    CCS_mean = np.mean(CCS_points, axis=0)
    WCS_mean = np.mean(WCS_points, axis=0)

    CCS_points_tran = CCS_points - CCS_mean
    WCS_points_tran = WCS_points - WCS_mean

    W = np.matmul(WCS_points_tran.T, CCS_points_tran)
    U, S, V_T = np.linalg.svd(W)

    rotm = np.matmul(V_T.T, U.T)
    t = CCS_mean - np.matmul(rotm, WCS_mean)

    return rotm, t
```

- The final output is the best rotation vector represented in quaternion and the best translation vector.

- Discussion
  - I saved the rotation and translation matrices of each validation image in 'estimation.pkl' in order to accelerate Q2.
  - For the function RANSAC,
    - I used 35 iterations because I want to get 50 proportion of outliers.
    - Also, I have tried different values for RANSAC threshold. Increasing the threshold gets more inliers but it is more possibly to overfit. So, I chose 1.5 to have trade-off between accuracy and overfitting.

Q1-2: Median Pose Error

- Median of rotation error: 0.0021679184919041992
  - Relative rotation angle between estimation and ground-truth.
- Median of translation error: 0.006588267977888796
  - Euclidean distance of all absolute pose differences.
  - $t_e = \|\mathbf{t} - \hat{\mathbf{t}}\|_2$
- Discussion

  - For calculating rotation error, first calculate relative rotation, then use axis angle representation instead of matrix or quaternion.
  - Relative rotation:
    - $R_e=R_{gt}*R_{-1est}$
  - Axis angle representation:
    - $\theta=\cos^{-1}((trace(R)-1)/2)$

Q1-3: Camera trajectory and camera poses

- The camera trajectory and camera poses (image plane) as a quadrangular pyramid.
  - According to the estimated rotation and translation matrices of each validation image, it is possible to find the camera position and its image plane in the world coordinate system.
    - The estimated rotation and translation matrices are the projection from the world to camera.
      - For plotting on the 3D point cloud, we need to transform apex and image plane from CCS to WCS. Therefore, I made use of inverse of rotation and negated translation.
      - $apex=[R^{-1}|t]$
    - Besides, the normal of each image plane is the orientation of the camera, which is along the +z axis (real points is in front of the camera).

- Discussion
  - For plotting the image planes,
    - I built four corners of each image plane by setting combination of +1, -1 values for x, y, and +1 for z.
    - And I scaled a little bit to make the quadrangular pyramids easier to distinguish.

Problem 2: Augmented Reality

- Settings:
  - Execute codes:
    ```
    python .\transform_cube.py
    ```
  - Package:
    - scipy.spatial.transform.Rotation, pandas, numpy, cv2, open3d
- The video:
  - https://drive.google.com/file/d/19h6Wmr2XaVmvekz16yoU0blCgEx_zmrx/view?usp=sharing (https://drive.google.com/file/d/19h6Wmr2XaVmvekz16yoU0blCgEx_zmrx/view?usp=sharing)
  - With the following adjusted cube:
    - `[R | t] = [[ 0.29954323 -0.01331663 0.00982445 0.88 ] [ 0.0156984 0.28501061 -0.09231744 0.12 ] [-0.00523572 0.09269098 0.2852735 -0.32 ]]`
    - `8 vertices = [[ 0.88 0.12 -0.32 ] [ 1.17954323 0.1356984 -0.32523572] [ 0.88982445 0.02768256 -0.0347265 ] [ 1.18936768 0.04338095 -0.03996222] [ 0.86668337 0.40501061 -0.22730902] [ 1.16622661 0.42070901 -0.23254474] 0.87650783 0.31269317 0.05796448] [ 1.17605106 0.32839157 0.05272876]]`
- Code Explanation:
  - First of all, I made use of the sample code for adjusting the position of the cube.
  - Then, build another cube points with unit length in the 3D point cloud set and assign different color to each surface.
  - Besides, I loaded the estimation.pkl file, which included the information of rotation and translation matrices of validation images from the question one.
  - For every image, project the cube points in 3D to 2D by multiplying the camera intrinsic matrix and the extrinsic matrix.
    - $homogeneous2D = intrinsicMat * extrinsicMat * homogeneous3D$
  - In order to deal with occlusion, apply painter's algorithm, sort points by depth from the furthest to the nearest (the third value of the homogeneous 2D points).
  - Check whether the points are inside the image size and only plot the ones inside the image.
  - Last, convert all images with plotting points to a video.
- Discussion
  - It is much faster to load the estimated extrinsic matrices from the pkl file I created.
  - At first, I created a list of points with some repetitions, which made the point colors complicated. In order not to repeatly assign color to the points, I found that I should only take the unique values for each point.