Computer Vision Assignment - Camera Calibration and Structure From Motion

Sophie Selgrad

November 19, 2021

1 Calibration

It is important to keep in mind that non-linear effects, such as radial distortion, are not being calibrated in this implementation.

1.1 Data Normalization

Here, we avoid numerical instabilities by normalising the data. The methods NormalizePoints2D and NormalizePoints3D have already been implemented, and we simply need to call them. Sometimes, the values within our input data can be vastly different and the scales of these numbers can be very different. Without normalisation, these differences can cause issues when one tries to combine the values as features during modelling. In normalisation, we alter the values so they fit within a common scale, without losing any vital information, changing the ratios in the source data or the changing the differences in the ranges of the values. It is important to normalise data to improve numerical stability.

1.2 DLT

Using the following from the exercise slides, we build our constraint matrix:

$$\begin{bmatrix} \mathbf{0}^T & -\boldsymbol{X}^T & x_2\boldsymbol{X}^T \\ \boldsymbol{X}^T & \mathbf{0}^T & -x_1\boldsymbol{X}^T \end{bmatrix}$$

In the end, our constraint matrix will have 12 columns, as the 0 and X^T vectors have four items each. Below, we can see how we build the constraint matrix:

```
def BuildProjectionConstraintMatrix(points2D, points3D):

  # TODO
  # For each correspondence, build the two rows of the constraint matrix and stack them

  num_corrs = points2D.shape[0]
  constraint_matrix = np.zeros((num_corrs * 2, 12))

  for i in range(num_corrs):
    constraint_matrix[2 * i, :] = np.array([0,0,0,0,
      -points3D[i,0], -points3D[i,1], -points3D[i,2], -1,
      points2D[i,1]*points3D[i,0], points2D[i,1]*points3D[i,1],points2D[i,1]*points3D[i,2], points2D[i,1]])

    constraint_matrix[2 * i + 1, :] = np.array([points3D[i,0], points3D[i,1], points3D[i,2], 1,
      0, 0, 0, 0,
      -points2D[i,0]*points3D[i,0], -points2D[i,0]*points3D[i,1], -points2D[i,0]*points3D[i,2], -points2D[i,0]])

  return constraint_matrix
```

As we can see, constraint_matrix[2 * i, :] corresponds to the first row in the previous figure, whereas constraint_matrix[2 * i + i, :] corresponds to the second row. Each correspondence results in the addition of two rows to the constraint matrix. In total, 2 independent constraints can be derived from a single 2D-3D correspondence.

1.3 Optimizing reprojection errors

Here, we simply compute P_hat_opt = OptimizeProjectionMatrix(P_hat, normalized_points2D, normalized_points3D). The methods OptimizeProjectionMatrix has already been implemented. The optimization error decreases after reprojection.
The algebraic error is $|Avec(P)|_2$. The geometric error is $sum_i|x_i - PX_i|_2$. This is the sum of distances between the ground truth image points and approximated projected image points. Both x_i and PX_i have been normalized.
The problem with the algebraic error is that the quantity being minimised is not geometrically or statistically meaningful. Also, the the solutions that minimize algebraic distance are not always intuitive.

1.4 Denormalizing the projection matrix

Using the previously saved data normalization matrices, and the following formula, we denormalize the projection matrix:

$$P = T^{-1}\hat{P}U$$

In our program, this formula takes the form below:

```
# Denormalize P
inv_T = np.linalg.inv(T2D)
P_hat_U = P_hat_opt @ T3D
P = inv_T @ P_hat_U
```

Here, T^{-1} is the inverse of T2D, and U in our formula is T3D.

1.5 Decomposing the projection matrix

Below are the results for K, R, and t. We can also observe the re-projection errors before and after nonlinear optimization.

```
(venv) Sophies-MacBook-Pro:code sophiese$ python calibration.py
Reprojection error before optimization: 0.0006316426059796243
Optimization terminated successfully    (Exit mode 0)
            Current function value: 0.0006253538899291337
            Iterations: 15
            Function evaluations: 209
            Gradient evaluations: 15
Reprojection error after optimization: 0.0006253538899291337
K=
[[2.713e+03 3.313e+00 1.481e+03]
 [0.000e+00 2.710e+03 9.654e+02]
 [0.000e+00 0.000e+00 1.000e+00]]
R =
[[-0.774  0.633 -0.007]
 [ 0.309  0.369 -0.877]
 [-0.552 -0.681 -0.481]]
t = [0.047 0.054 3.441]
```

After reprojection, the optimization error decreases by 0.00000628871.
Below is the formula for K:

$$\mathrm{K} = \begin{bmatrix} \alpha_x & s & x_0 \\ & \alpha_y & y_0 \\ & & 1 \end{bmatrix}$$

If we compare the results of K with the formula, we can see that the diagonal values areas are as expected. α_x and α_y are equal, as expected. The values in the bottom-left corner are all 0, as expected. Therefore, we can deduce that our results are reasonable.

2 Structure from Motion

2.1 Essential matrix estimation

Here, we compute the essential matrix. We have to define a method called EstimateEssentialMatrix. This takes K, the image objects im1 and im2, and the matches. Matches allow us to find corresponding points in two images. First, we have have to make the image coordinates of the key points in im1 and im2 homogeneous by computing MakeHomogeneous(im1.kps,1) and MakeHomogeneous(im2.kps,1). Then, we have compute the matrix multiplication of the inverse of K and the transposes of homo_im1 and homo_im2. Finally, we transpose these results again.

```python
def EstimateEssentialMatrix(K, im1, im2, matches):
  # TODO
  # Normalize coordinates (to points on the normalized image plane)

  # These are the keypoints on the normalized image plane (not to be confused
  # with the normalization in the calibration exercise)
  homo_im1 = MakeHomogeneous(im1.kps,1)
  homo_im2 = MakeHomogeneous(im2.kps,1)
  normalized_kps1 = (np.linalg.inv(K) @ homo_im1.transpose()).transpose()
  normalized_kps2 = (np.linalg.inv(K) @ homo_im2.transpose()).transpose()
```

Now, we assemble the constraint matrix as seen below:

$$\begin{bmatrix} x'_1 x_1 & x'_1 y_1 & x'_1 & y'_1 x_1 & y'_1 y_1 & y'_1 & x_1 & y_1 & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x'_n x_n & x'_n y_n & x'_n & y'_n x_n & y'_n y_n & y'_n & x_n & y_n & 1 \end{bmatrix}$$

x' and y' represent the x and y values for the first image, whilst x and y are the x and y values for the second image. After that, we solve for the nullspace of the constraint matrix, and we reshape the vectorized matrix to a 3×3 matrix. We then need to fulfil the following internal constraints of E, as seen below:

$$U, S, V^T = svd(\hat{E})$$

$$E = U \begin{bmatrix} 1 & & \\ & 1 & \\ & & 0 \end{bmatrix} V^T$$

In our program, these constraints are represented in the following form:

```python
# TODO
# We need to fulfill the internal constraints of E
# The first two singular values need to be equal, the third one zero.
# Since E is up to scale, we can choose the two equal singluar values arbitrarily
U, S, trans_V = np.linalg.svd(E_hat)
S[1] = S[0]
S[-1] = 0
inter = U @ np.array([[1,0,0],[0,1,0],[0,0,0]])
E = inter @ trans_V
```

2.2 Point triangulation

Below is the code for the filtering step in the point triangulation:

```python
# Filter points behind the cameras by transforming them into each camera space and checking the depth (Z)
# Make sure to also remove the corresponding rows in `im1_corrs` and `im2_corrs`

homo_points = MakeHomogeneous(points3D,1)
projected_1 = (P1 @homo_points.transpose()).transpose()
projected_2 = (P2 @homo_points.transpose()).transpose()

negative_indices = []
for i in range(projected_1.shape[0]):
  if projected_1[i,2] < 0 or projected_2[i,2] < 0:
    negative_indices.append(i)

points3D = np.delete(points3D, negative_indices, 0)
im1_corrs = np.delete(im1_corrs, negative_indices, 0)
im2_corrs = np.delete(im2_corrs, negative_indices, 0)

return points3D, im1_corrs, im2_corrs
```

We first have to make Points3D homogeneous. Using the result, we now project the points into each camera space. For a point to be behind the camera, the distance has to be less than 0. The indices of the points for which this condition is true are stored, and then the items in points3D, im1_corrs and im2_corrs at these indices are removed.

2.3 Finding the correct decomposition

Using the fully implemented TriangulatePoints, we now need to find the correct decomposition. First, we set the pose for one of the images. Here, we choose the second images. We then need to calculate the result of TriangulatePoints and store the results in the correct order. Then, we compute the size of rel_points3D and append this to a list of sizes called rel_points_list. From this, we choose the pose

at the index of the largest item in rel_points_list, and set the image pose for the first image.

```
# For each possible relative pose, try to triangulate points.
# We can assume that the correct solution is the one that gives the most points in front of both cameras
# Be careful not to set the transformation in the wrong direction

e_im2.SetPose(np.eye(3), np.array([0,0,0]))
best_pose_idx = 0
rel_points_list = []

for i in possible_relative_poses:
  e_im1.SetPose(i[0],i[1])
  rel_points3D, rel_im1_corrs, rel_im2_corrs = TriangulatePoints(K,e_im1,e_im2,e_matches)
  rel_points_list.append(rel_points3D.shape[0])

best_pose_idx = rel_points_list.index(max(rel_points_list))
best_pose = possible_relative_poses[best_pose_idx]
```

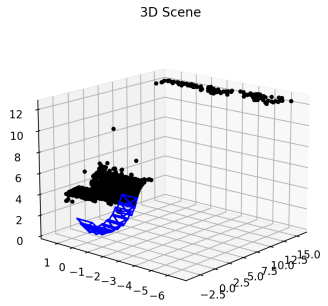
```
# Set the image poses in the images (image.SetPose(...))
e_im1.SetPose(best_pose[0], best_pose[1])

# TODO Triangulate initial points
points3D, im1_corrs, im2_corrs = TriangulatePoints(K, e_im1, e_im2, e_matches)
```

2.4 Absolute pose estimation

We do not need to implement anything additional here.

2.5 Map extension

In TriangulateImage, we loop over all registered images and triangulate new points with the new image. We save the key point correspondence indices in the corrs dictionary. Also, we store the new 3d points in an array which we later append to the global points3D array.
In UpdateReconstructionState, we add the new 2d-3d correspondences that we gained from TriangulateImage. From here, we adjust the indices to turn the local indices from the triangulation function into global indices using proper offsets by counting the number of elements that were previously stored in the points3D array, and then using the number of points that were added.
Below, the results of our implementation can be seen below:



Here, we can observe the shape of the fountain (the wavy part) as well as a pole of points in the upper-right hand corner. If we observe some of the input images, we can deduce that the pole of points is from another building to the right of the fountain, as seen below: