Exercise 5: Nonlinear optimization, camera calibration

02504 Computer vision

Morten R. Hannemose, mohan@dtu.dk, DTU Compute

March 3, 2022

These exercises will take you through:

checkerboard calibration, in real life with OpenCV.

non-linear optimization, where you will implement a function non-linear triangulation

Nonlinear optimization

This exercise will take you through doing nonlinear optimization for triangulation of a single point. The same principles can be applied to more complex situations such as camera calibration, or situations where we lack a linear algoritm.

Construct two cameras with $\boldsymbol{R}_1 = \boldsymbol{R}_2 = \boldsymbol{I}$, $\boldsymbol{t}_1 = \begin{bmatrix}0 & 0 & 1\end{bmatrix}^\mathrm{T}$, $\boldsymbol{t}_2 = \begin{bmatrix}0 & 0 & 20\end{bmatrix}^\mathrm{T}$, and

$$\boldsymbol{K}_1 = \boldsymbol{K}_2 = \begin{bmatrix}700 & 0 & 600\\ 0 & 700 & 400\\ 0 & 0 & 1\end{bmatrix}.$$

The cameras both observe the same 3D point $\boldsymbol{Q} = \begin{bmatrix}1 & 1 & 0\end{bmatrix}^\mathrm{T}$.

Exercise 5.1

What are the projection matrices $\boldsymbol{P}_1$ and $\boldsymbol{P}_2$?.

What is the projection of $\boldsymbol{Q}$ in cameras one and two ($\boldsymbol{q}_1$ and $\boldsymbol{q}_2$)?

Exercise 5.2

To simulate noise in the detection of points, we add errors to our projections.
$\tilde{\boldsymbol{q}}_1 = \boldsymbol{q}_1 + \begin{bmatrix}1 & -1\end{bmatrix}^{\mathrm{T}}$, $\tilde{\boldsymbol{q}}_2 = \boldsymbol{q}_2 + \begin{bmatrix}1 & -1\end{bmatrix}^{\mathrm{T}}$.

Use your function `triangulate` from week 3 to triangulate $\boldsymbol{Q}$ from $[\tilde{\boldsymbol{q}}_1, \tilde{\boldsymbol{q}}_2]$ and $[\boldsymbol{P}_1, \boldsymbol{P}_2]$.

Take the newly triangulated point $\tilde{\boldsymbol{Q}}$ and re-project it to the cameras. How far is it from our observations of the point ($\tilde{\boldsymbol{q}}_1$, $\tilde{\boldsymbol{q}}_2$)? In other words, what is the reprojection error for each camera?

Is this as you expected when recalling the lecture from week 3?

How far is $\tilde{\boldsymbol{Q}}$ from $\boldsymbol{Q}$?

Exercise 5.3

We are going to make a new function `triangulate_nonlin` that does triangulation using nonlinear optimization. It should take the same inputs as `triangulate`, i.e. a list of n pixel coordinates (`q1`, `q2`, . . . , `qn`), and a list of n projection matrices (`P1`, `P2`, . . . , `Pn`).

Start by defining a helper-function inside `triangulate_nonlin`.
This function, called `compute_residuals`, should take the parameters we want to optimize (in this case $\boldsymbol{Q}$) as input, and should returns a vector of residuals (i.e. the numbers that we want to minimize the sum of squares of). In this case the residuals are the differences in projection, i.e.

$$\begin{bmatrix}\Pi(\boldsymbol{P}_1 Q) - \tilde{\boldsymbol{q}}_1 \\ \vdots\end{bmatrix}$$

Call `triangulate` inside your function to get an initial guess `x0` and use
`scipy.optimize.least_squares(compute_residuals, x0)`
to do least squares optimization, starting from the initial guess of your linear algorithm.

Exercise 5.4

Use `triangulate_nonlin` with $[\tilde{\boldsymbol{q}}_1, \tilde{\boldsymbol{q}}_2]$ and $[\boldsymbol{P}_1, \boldsymbol{P}_2]$.
Let us call the nonlinearly estimated point $\hat{\boldsymbol{Q}}$.

What is the reprojection error of $\hat{\boldsymbol{Q}}$ to camera 1 and 2?

How far is $\hat{\boldsymbol{Q}}$ from $\boldsymbol{Q}$?

Is this an improvement over the result in Exercise 5.2?

Congratulations! You now have a useful function that does not currently exist in OpenCV!

Camera calibration with OpenCV

In the following exercises you will be calibrating your own camera. For this we suggest using a camera in your phone or similar. Disable HDR on your phone.

If you have a phone with a wide angle camera, consider using this camera for the exercise (as more lens distortion is more fun and challenging). Remember to disable lens correction in your camera app before taking the pictures.

If you get stuck with the OpenCV functions, start by looking it up in the OpenCV documentation.

Exercise 5.5

The calibration target (pdf)

Take one of the provided calibration targets or print your own. If you do not have access to a printer, showing the target on a laptop or tablet display is also an option, albeit less ideal due to the glass on top of the display, which can cause reflection and refraction.

Using your calibration target, take pictures of it from many different angles. Make sure to have an image of it straight on and well lit, and try more extreme angles as well. Try to get every part of the frame covered. You should have around twenty images.

Exercise 5.6

Transfer the images to your computer and load them into Python.

Take the image where the checkerboard is seen straight on and use `cv2.resize` to resize the image to 400×600.

Use the function `cv2.findChessboardCorners` to detect checkerboards in your images. Be aware that the function needs the number of internal corners on the checkerboard as input *not* the total size of the checkerboard. Use the small version of the image to figure out which combination of arguments wants as input (as the function for detecting corners takes a long time if it can't find corners in a high resolution image).

Is the method able to detect checkerboards in all images? If you have a too extreme angle on some images, or there is too much of the checkerboard outside the frame, the detection function may fail. Use the images it was able to successfully detect checkerboards in, and continue to the next exercise.

Exercise 5.7

Now it's time to calibrate the camera!

Use `checkerboard_points(n, m)` from last week to construct the points on the checkerboard in 3D, and either use your own function `calibratecamera(qs, Q)` or the one from OpenCV `cv2.calibrateCamera` to calibrate the camera.

If you use the OpenCV function, make sure to set the `flags` argument to not have any lens distortion initially (feel free to add it later).

```
flags = cv2.CALIB_FIX_K1+cv2.CALIB_FIX_K2+cv2.CALIB_FIX_K3+cv2.CALIB_FIX_K4+
        cv2.CALIB_FIX_K5+cv2.CALIB_FIX_K6+cv2.CALIB_ZERO_TANGENT_DIST
```

Inspect the $\boldsymbol{K}$ matrix. Is the principal point approximately in the center of your images?

Exercise 5.8

Reproject the checkerboard corners to the images.
Tip: `cv2.calibrateCamera` returns `rvecs`, which are the $\boldsymbol{R}$ matrices stored in axis-angle representation. You can convert them to rotation matrices with `cv2.Rodrigues`.

Compute the reprojection error for each frame. Find the frame with the highest reprojection error and show both the detected and reprojected corner points. Hopefully you have an RMSE of a few pixels.

Exercise 5.9

Try changing your distortion model to include k_1 and k_2.

Do you get a lower reprojection error?

If your camera has visible lens distortion, try using the function from week 2 to undistort one of your images.

Solutions

Answer of exercise 5.1

$\boldsymbol{q}_1 = \begin{bmatrix}1300 & 1100\end{bmatrix}^{\mathrm{T}}$

$\boldsymbol{q}_2 = \begin{bmatrix}635 & 435\end{bmatrix}^{\mathrm{T}}$

Answer of exercise 5.2

$\tilde{\boldsymbol{Q}} = \begin{bmatrix}1.015 & 9.853 & 2.9 \cdot 10^{-4}\end{bmatrix}^{\mathrm{T}}$

The reprojection error in camera 1 is 13.4 pixels and it is 0.67 pixels in camera 2.

We expect the linear algorithm to place a larger weight on the error of camera 2 than camera 1, as it has a larger s. Therefore camera 2 having the smallest reprojection error is as we expected.

$\left\|\boldsymbol{Q} - \tilde{\boldsymbol{Q}}\right\|_2 = 0.021$

Answer of exercise 5.4

$\hat{\boldsymbol{Q}} = [1.00153897e+00, 9.98546320e-01, 4.27473316e-05]$ The reprojection error in camera 1 is 0.067 pixels and 1.34 pixels in camera 2.

$\left\|\boldsymbol{Q} - \tilde{\boldsymbol{Q}}\right\|_2 = 0.0021$. That's approximately 10 times closer to the true position.