Exercise 1: Rigid and perspective transformations in homogeneous coordinates

02504 Computer vision

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

February 10, 2023

Homogeneous coordinates

Exercise 1.1

Consider the following four points in 2D

$$\boldsymbol{p}_1 = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix}, \ \boldsymbol{p}_2 = \begin{bmatrix} 4 \\ 2 \\ 2 \end{bmatrix}, \ \boldsymbol{p}_3 = \begin{bmatrix} 6 \\ 4 \\ -1 \end{bmatrix}, \text{ and } \ \boldsymbol{p}_4 = \begin{bmatrix} 5 \\ 3 \\ 0.5 \end{bmatrix}, \tag{1}$$

written in their *homogeneous* form. What are their corresponding inhomogeneous coordinates $\boldsymbol{q}_i$?

Exercise 1.2

Consider now the following four points in 3D

$$\boldsymbol{P}_1 = \begin{bmatrix} 1 \\ 10 \\ -3 \\ 1 \end{bmatrix}, \ \boldsymbol{P}_2 = \begin{bmatrix} 2 \\ -4 \\ 1.1 \\ 2 \end{bmatrix}, \ \boldsymbol{P}_3 = \begin{bmatrix} 0 \\ 0 \\ -1 \\ 10 \end{bmatrix}, \text{ and } \ \boldsymbol{P}_4 = \begin{bmatrix} -15 \\ 3 \\ 6 \\ 3 \end{bmatrix}. \tag{3}$$

What are their corresponding inhomogeneous coordinates $\boldsymbol{Q}_i$?

Exercise 1.3

A 2D line is given as

$$x + 2y = 3\,. \tag{5}$$

Write this line in homogeneous form i.e. $\boldsymbol{l}^{\mathrm{T}}\boldsymbol{p} = 0$. What is $\boldsymbol{l}$?

Exercise 1.4

Using $\boldsymbol{l}$ from Exercise 1.3, which of the following 2D points are on this line?

$$\boldsymbol{p}_1 = \begin{bmatrix} 3 \\ 0 \\ 1 \end{bmatrix}, \ \boldsymbol{p}_2 = \begin{bmatrix} 6 \\ 0 \\ 2 \end{bmatrix}, \ \boldsymbol{p}_3 = \begin{bmatrix} 1 \\ 1 \\ 2 \end{bmatrix}, \ \boldsymbol{p}_4 = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}, \ \boldsymbol{p}_5 = \begin{bmatrix} 110 \\ -40 \\ 10 \end{bmatrix}, \text{ and } \boldsymbol{p}_6 = \begin{bmatrix} 11 \\ 4 \\ 1 \end{bmatrix}. \tag{7}$$

Exercise 1.5

Given the two lines

$$\boldsymbol{l}_0 = \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix} \text{ and } \boldsymbol{l}_1 = \begin{bmatrix} -1 \\ 1 \\ -3 \end{bmatrix}, \tag{9}$$

what is their point of intersection $\boldsymbol{q}_\text{I}$?

Exercise 1.6

Consider the following matrix

$$\boldsymbol{A} = \begin{bmatrix} 10 & 0 & 2 \\ 0 & 10 & -3 \\ 0 & 0 & 1 \end{bmatrix}. \tag{11}$$

What is the result of $\boldsymbol{Ap} = \boldsymbol{q}$, where $\boldsymbol{p}$ and $\boldsymbol{q}$ are 2D points in homogeneous coordinates? Explain what each of the (non-zero) coefficients in $\boldsymbol{A}$ does to the coordinates of $\boldsymbol{q}$.

Exercise 1.7

A new line is given

$$\boldsymbol{l} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \\ -1 \end{bmatrix}. \tag{12}$$

What is the (shortest) distance between the line $\boldsymbol{l}$ and the points

$$\boldsymbol{p}_1 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, \ \boldsymbol{p}_2 = \begin{bmatrix} \sqrt{2} \\ \sqrt{2} \\ 1 \end{bmatrix}, \text{ and } \boldsymbol{p}_3 = \begin{bmatrix} \sqrt{2} \\ \sqrt{2} \\ 4 \end{bmatrix}? \tag{13}$$

Exercise 1.8

Repeat Exercise 1.7 with

$$\boldsymbol{l} = \begin{bmatrix} 2 \\ 2 \\ -1 \end{bmatrix}. \tag{16}$$

Programming exercises

The programming exercises in this course will assume that you are programming in Python and using OpenCV. It will be possible to follow the course using a different language such as MATLAB, but you will be more on your own. We suggest that you use a tool where you can *interactively* run Python code, such as Jupyter notebook, Spyder or VS code.

Setting up

First you should do a few introductory exercises that will prepare you for the later exercises in the course.

Exercise 1.9

Please install OpenCV 4.4.0 or later in your Python environment.

Tip: You can check which version you have installed by running

```
import cv2
print(cv2.__version__)
```

Exercise 1.10

In later weeks you will work on images your capture yourself. To prepare for this

- Capture an image with a camera
- Transfer it to your computer
- Load it with OpenCV
- Display it with Matplotlib

Tip: You can import Matplotlib as follows

```
import matplotlib.pyplot as plt
plt.imshow(im)
```

Tip: Do the colors of the image look weird? This is because OpenCV stores an image as (blue, green, red) while Matplotlib uses the more common (red, green blue). Flip the channels of the image (`im[:,:,::-1]`), to make the colors display correctly.

Pinhole camera

This time you will get familiar with manipulating points in programs. We will build a couple of helper functions, which in the end will let you project several 3D points into an image plane.

Exercise 1.11

First lets make some more 3D points to "photograph". We will use a function that generates a number of 3D points in a recognizable shape like Figure 1.

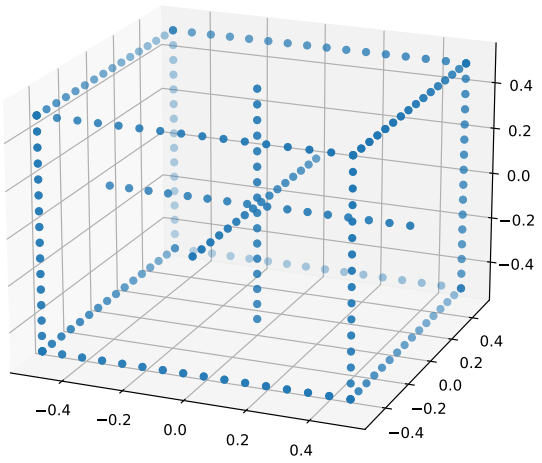

Figure 1: A 3D plot of a set of points generated by the `box3d` function.

Let us define a function `box3d`, that generates a list of coordinates (a $3 \times n$ array) in a box shaped like Figure 1. The box is made of points along the 12 edges and in addition, we insert a cross through the middle. Each line has 16 points between -0.5 and 0.5.

Consider using the following implementation of the function:

```
import itertools as it
def box3d(n=16):
    points = []
    N = tuple(np.linspace(-1, 1, n))
    for i, j in [(-1, -1), (-1, 1), (1, 1), (0, 0)]:
        points.extend(set(it.permutations([(i, )*n, (j, )*n, N])))
    return np.hstack(points)/2
```

Exercise 1.12

Implement a two helper-functions:

- `Pi` that converts from homogeneous to inhomogeneous coordinates, and

- `PiInv` that converts from inhomogeneous to homogeneous coordinates.

The functions should take Numpy arrays with size (dimension of point $\times$ number of points) as their input.

Tip: To add homogeneous coordinate to a matrix you can use `np.vstack` with `np.ones`.
Tip: You can divide by and remove the last coordinate by doing `p = ph[:-1]/ph[-1]`

Exercise 1.13

Now lets us make our "camera". Create a function `projectpoints`, that takes as inputs:

- the camera matrix $\boldsymbol{K}$
- the pose of the camera ($\boldsymbol{R}$, $\boldsymbol{t}$)
- a $3 \times n$ matrix ($\boldsymbol{Q}$), representing n points in 3D to be projected into the camera.

The function should return the projected 2D points as a $2 \times n$ matrix.

Test your function $\boldsymbol{Q} =$ `box3d`,

$$\boldsymbol{K} = \boldsymbol{R} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \boldsymbol{t} = \begin{bmatrix} 0 \\ 0 \\ 4 \end{bmatrix} \tag{18}$$

Tip: You can do matrix multiplication in Numpy using @, for example `A@b`.

Exercise 1.14

Try instead with $\boldsymbol{R} = \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{bmatrix}$

where $\theta = 30°$.

What is the effect?

Exercise 1.15

Play around with changing $\boldsymbol{R}$ and $\boldsymbol{t}$.

What is the relationship between the position of the camera and $\boldsymbol{t}$?

Solutions

Answer of exercise 1.1

The four 2D points are given in standard coordinates

$$\boldsymbol{q}_1 = \begin{bmatrix} 1 \\ 2 \end{bmatrix}, \ \boldsymbol{q}_2 = \begin{bmatrix} 2 \\ 1 \end{bmatrix}, \ \boldsymbol{q}_3 = \begin{bmatrix} -6 \\ -4 \end{bmatrix}, \text{ and } \ \boldsymbol{q}_4 = \begin{bmatrix} 10 \\ 6 \end{bmatrix}. \tag{2}$$

Answer of exercise 1.2

The four 2D points are given in standard coordinates

$$\boldsymbol{Q}_1 = \begin{bmatrix} 1 \\ 10 \\ -3 \end{bmatrix}, \ \boldsymbol{Q}_2 = \begin{bmatrix} 1 \\ -2 \\ 0.55 \end{bmatrix}, \ \boldsymbol{Q}_3 = \begin{bmatrix} 0 \\ 0 \\ -0.1 \end{bmatrix}, \text{ and } \ \boldsymbol{Q}_4 = \begin{bmatrix} -5 \\ 1 \\ 2 \end{bmatrix}. \tag{4}$$

Answer of exercise 1.3

The 2D line $\boldsymbol{l}$ where $\boldsymbol{l}^{\mathrm{T}}\boldsymbol{p} = 0$ is

$$\boldsymbol{l} = \begin{bmatrix} 1 \\ 2 \\ -3 \end{bmatrix} \tag{6}$$

Answer of exercise 1.4

Any point on the line must obey $\boldsymbol{l}^{\mathrm{T}}\boldsymbol{p} = 0$. Using the same line $\boldsymbol{l}$ as defined in answer to Exercise 1.3 we find

$$\boldsymbol{l}^{\mathrm{T}}\boldsymbol{p}_1 = 0\,, \ \ \boldsymbol{l}^{\mathrm{T}}\boldsymbol{p}_2 = 0\,, \ \ \boldsymbol{l}^{\mathrm{T}}\boldsymbol{p}_3 = -3\,, \ \ \boldsymbol{l}^{\mathrm{T}}\boldsymbol{p}_4 = 0\,, \ \ \boldsymbol{l}^{\mathrm{T}}\boldsymbol{p}_5 = 0\,, \text{ and } \ \ \boldsymbol{l}^{\mathrm{T}}\boldsymbol{p}_6 = 16\,. \tag{8}$$

In other words, points $\boldsymbol{p}_1$, $\boldsymbol{p}_2$, $\boldsymbol{p}_4$, and $\boldsymbol{p}_5$ are all on the line $\boldsymbol{l}$; the points $\boldsymbol{p}_3$ and $\boldsymbol{p}_6$ are not.

Answer of exercise 1.5

The intersection of two lines is the cross product of the homogeneous coordinate definitions.

$$\boldsymbol{l}_0 \times \boldsymbol{l}_1 = \begin{bmatrix} -2 \\ 4 \\ 2 \end{bmatrix}. \tag{10}$$

Answer of exercise 1.6

The direct solution is $\boldsymbol{Ap} = [10x + 2, 10y - 3, 1]$. The two 10's scale the x- and y-coordinates. If the last diagonal factor was 10 and not 1, this would have no effect on the inhomogeneous coordinates. The two remaining factors are 2D translations. Finally, also notice that the translations happened after scaling.

Answer of exercise 1.7

The shortest distance d between a line $\boldsymbol{l} = [l_x, l_y, l_w]^{\mathrm{T}}$ and a point $\boldsymbol{p}$ is given

$$d = \frac{\left|\boldsymbol{l}^{\mathrm{T}}\boldsymbol{p}\right|}{|p_w|\sqrt{l_x^2 + l_y^2}}\,. \tag{14}$$

The solutions are

$$d_1 = 1\,,\ d_2 = 1\,,\ \text{and}\ \ d_3 = 1/2\,. \tag{15}$$

Answer of exercise 1.8

The definition of the shortest distance d is given in answer to Exercise 1.7, and the new solutions are

$$d_0 = \frac{1}{\sqrt{8}} = 0.3536\,,\ d_1 = \frac{4\sqrt{2} - 1}{\sqrt{8}} = 1.6464\,,\ \text{and}\ \ d_2 = \frac{\sqrt{2} - 1}{\sqrt{8}} = 0.1464\,. \tag{17}$$

Exercise 2: Rigid and perspective transformations in homogeneous coordinates

02504 Computer vision

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

February 9, 2024

Mathematical exercises: Pinhole camera

In these exercises we will assume a *modern* camera with completely square pixels. What are the skew parameters then?

Exercise 2.1

Reuse the `box3d` function from last week. Assume that $f = 600$, $\alpha = 1$, $\beta = 0$, and $\delta x = \delta y = 400$. Given a traditional camera, what is the resolution in pixels?

Also assume $\boldsymbol{R} = \boldsymbol{I}$, and $\boldsymbol{t} = [0, .2, 1.5]^{\mathrm{T}}$. Use `projectpoints` from last week, to project the points.

Are all the points are captured by the image sensor?

Where does the corner $\boldsymbol{P}_1 = [-0.5, -0.5, -0.5]$ project to?

Exercise 2.2

Create a new or change your function `projectpoints` to a version that also takes `distCoeffs` as an input. The list `distCoeffs` should contain the distortion coefficients $[k_3, k_5, k_7, \dots]$. Make the function work for at least 3 coefficients.

Test your function with the same setup as in Exercise 2.1 and but assume that the distortion is

$$\Delta r(r) = -0.2r^2 \,. \tag{2}$$

What are the distortion coefficients in this case?

Where does the corner $\boldsymbol{P}_1$ project to?

Are all the points captured by the image sensor?

Plot the results and try changing the distortion coefficients. Do they behave as they should?

Exercise 2.3

Download the following image:
https://people.compute.dtu.dk/mohan/02504/gopro_robot.jpg

The image has been captured using a GoPro. Assume that the focal length is 0.455732 times the image width, and a reasonable guess of principal point, α, and β. The distortion coefficients are

$$k_3 = -0.245031, \quad k_5 = 0.071524, \quad k_7 = -0.00994978$$

What is $\boldsymbol{K}$?

Exercise 2.4

Implement a function `undistortImage` that takes an image, a camera matrix, and distortion coefficients and returns an undistorted version of the same image. Use the following as an outline of your function

```
x, y = np.meshgrid(np.arange(im.shape[1]), np.arange(im.shape[0]))
p = np.stack((x, y, np.ones(x.shape))).reshape(3, -1)

q = ...
q_d = ...
p_d = ...

x_d = p_d[0].reshape(x.shape).astype(np.float32)
y_d = p_d[1].reshape(y.shape).astype(np.float32)
assert (p_d[2]==1).all(), 'You did a mistake somewhere'
im_undistorted = cv2.remap(im, x_d, y_d, cv2.INTER_LINEAR)
```

Test the function by undistorting the image from the previous exercise. Are the lines straight now?

Tip: Reshape `r` back to an image and show it to check that it looks like expected.

Homographies

Exercise 2.5

Consider the following points on a plane

$$\boldsymbol{p}_{2a} = \begin{bmatrix}1\\1\end{bmatrix}, \quad \boldsymbol{p}_{2b} = \begin{bmatrix}0\\3\end{bmatrix}, \quad \boldsymbol{p}_{2c} = \begin{bmatrix}2\\3\end{bmatrix}, \quad \boldsymbol{p}_{2d} = \begin{bmatrix}2\\4\end{bmatrix}.$$

Using the homography

$$\boldsymbol{H} = \begin{bmatrix}-2 & 0 & 1\\ 1 & -2 & 0\\ 0 & 0 & 3\end{bmatrix},$$

map the points using the homography ($\boldsymbol{H}\boldsymbol{q}_2$).

Exercise 2.6

Create a function `hest` that takes two sets of points in 2D, `q1` and `q2`, and returns the estimated homography matrix using the linear algorithm. You can extract the singular vector corresponding to the smallest singular value with SVD. However, Numpy returns $\boldsymbol{V}^{\mathrm{T}}$ instead of $\boldsymbol{V}$, so you can do

```
U, S, VT = np.linalg.svd(B)
VT[-1, :]
```

Test your function by using the points from the exercise above. Do you get the exact same numbers in your homography, or are they scaled? Explain why this is fine.

Exercise 2.7

Create a helper function `normalize2d`. This function finds the transformation $\boldsymbol{T}$ such that $\boldsymbol{q}_{ih} = \boldsymbol{T}\boldsymbol{p}_{ih}$, has mean $[0, 0]$ and standard deviation $[1, 1]$ for all $\boldsymbol{q}_i$.

Test it out with some 2D points and make sure the mean and standard deviation are as expected.

Exercise 2.8

Improve your `hest` function by adding an option (`normalize=true/false`) to normalize the points with `normalize2d` before estimating the homography. Apply the $\boldsymbol{T}$ matrices to the estimated $\boldsymbol{H}$ so the estimated homography still operates on non-normalized points.

Exercise 2.9

Generate 100 random 2D points, and a random homography. Map the points using the homography, and use `hest` to estimate the homography from the points. You can use the following code as a starting point:

```
q2 = np.random.randn(2, 100)
q2h = np.vstack((q2, np.ones(1, 100)))
H_true = np.random.randn(3,3)
q1h = H_true@q2h
q1 = Pi(q1h)
```

Exercise 2.10

Take a piece of paper and draw at least four $\times$-marks on it in random locations. Make sure to number them.

Put a small object on top of the paper, and use your phone to take pictures of your paper from two different viewpoints (A and B) Get the x, y coordinates of your $\times$-marks in image coordinates. You can get the coordinates of clicked points with `plt.ginput`.

Use your annotated points to estimate the homography from image A to image B. Can you click on any point in one image and show where it is in the other image? What happens when you click on the object?

Exercise 2.11

Re-create image A, using only pixel intensities from image B. Generate an overlay of the two images. Does the object on top align?

To accomplish this, use the following function:

```
def warpImage(im, H):
    imWarp = cv2.warpPerspective(im, H, (im.shape[1], im.shape[0]))
    return imWarp
```

It takes an image and a homography and returns the image warped with the homography.

Solutions

Answer of exercise 2.1

In a traditional camera, the principal point is exactly in the middle of the sensor. So, for this camera the sensor has $2 \times 400 = 800$ pixels along each dimension i.e. a resolution of 800×800 pixels.

The projection matrix reads

$$\boldsymbol{\mathcal{P}} = \begin{bmatrix} 600 & 0 & 400 & 600 \\ 0 & 600 & 400 & 720 \\ 0 & 0 & 1 & 1.5 \end{bmatrix}, \tag{1}$$

Some points have an y value greater than 800, and are not visible in the image, as they are outside the image sensor.

$\boldsymbol{P}_1$ projects to $[100, 220]^{\mathrm{T}}$

Answer of exercise 2.2

The first coefficient is negative, so this is barrel distortion.

The distortion coefficients are $k_3 = -0.2$, $k_5 = 0$, $k_7 = 0, \ldots$

$\boldsymbol{P}_1$ projects to $[120.4, 232.24]^{\mathrm{T}}$

The projection now reads:

$$\boldsymbol{q} = \boldsymbol{K}\ \Pi^{-1}\left[\mathrm{dist}\left[\Pi\left(\begin{bmatrix}\boldsymbol{R} & \boldsymbol{t}\end{bmatrix}\boldsymbol{Q}\right)\right]\right],\ \text{where} \tag{3}$$

$$\mathrm{dist}\left(\boldsymbol{p}\right) = \boldsymbol{p}(1 - 0.2\left\|\boldsymbol{p}\right\|_2^2)\,. \tag{4}$$

Notice in particular the transformations between inhomogeneous and homogeneous coordinates (Π).

All the points are now projecting inside the image, and will thus be visible.

Answer of exercise 2.3

The principal point is in the center of the image. $\boldsymbol{K} = \begin{bmatrix} 875 & 0 & 960 \\ 0 & 875 & 540 \\ 0 & 0 & 1 \end{bmatrix}$.

Answer of exercise 2.5

We convert the $\boldsymbol{p}_i$s to homogeneous coordinates, and compute

$$\boldsymbol{q}_{ih} = \boldsymbol{H}\boldsymbol{p}_{ih}.$$

This gives us:

$$\boldsymbol{q}_1 = \begin{bmatrix} -\frac{1}{3} \\ -\frac{1}{3} \end{bmatrix}, \quad \boldsymbol{q}_2 = \begin{bmatrix} \frac{1}{3} \\ -2 \end{bmatrix}, \quad \boldsymbol{q}_3 = \begin{bmatrix} -1 \\ -\frac{4}{3} \end{bmatrix}, \quad \boldsymbol{q}_4 = \begin{bmatrix} -1 \\ -2 \end{bmatrix}$$

Answer of exercise 2.6

You should obtain the same homography, but multiplied with a scalar such that $\|\boldsymbol{H}\|_F = 1$.

Exercise 3: Stereo vision, and triangulation

02504 Computer vision

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

February 16, 2024

Epipolar geometry

Set up two cameras, both with the internal parameters

$$\boldsymbol{K} = \begin{bmatrix} 1000 & 0 & 300 \\ 0 & 1000 & 200 \\ 0 & 0 & 1 \end{bmatrix}. \tag{1}$$

Now, for the first camera — let us call that `Cam1` — set the rotation to identity $\boldsymbol{R}_1 = \boldsymbol{I}$ and set the translation to zero $\boldsymbol{t}_1 = \boldsymbol{0}$. For the second camera `Cam2` use the rotation given by the $\boldsymbol{\mathcal{R}}$ function

$$\boldsymbol{R}_2 = \boldsymbol{\mathcal{R}}(0.7, -0.5, 0.8) \tag{2}$$

$$\boldsymbol{\mathcal{R}}(\theta_x, \theta_y, \theta_z) = \begin{bmatrix} \cos(\theta_z) & -\sin(\theta_z) & 0 \\ \sin(\theta_z) & \cos(\theta_z) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos(\theta_y) & 0 & \sin(\theta_y) \\ 0 & 1 & 0 \\ -\sin(\theta_y) & 0 & \cos(\theta_y) \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta_x) & -\sin(\theta_x) \\ 0 & \sin(\theta_x) & \cos(\theta_x) \end{bmatrix}, \tag{3}$$

and the translation

$$\boldsymbol{t}_2 = \begin{bmatrix} 0.2 \\ 2 \\ 1 \end{bmatrix}. \tag{4}$$

The rotation can be constructed in Python using `Rotation` module from `scipy` as follows:

```
from scipy.spatial.transform import Rotation
R2 = Rotation.from_euler('xyz', [0.7, -0.5, 0.8]).as_matrix()
```

Exercise 3.1

Consider the 3D point

$$\boldsymbol{Q} = \begin{bmatrix} 1 \\ 0.5 \\ 4 \\ 1 \end{bmatrix} \tag{5}$$

and find the projections in `Cam1` and `Cam2`, respectively, points $\boldsymbol{q}_1$ and $\boldsymbol{q}_2$.

Exercise 3.2

Implement a function `CrossOp` that takes a vector in 3D and returns the 3×3 matrix corresponding to taking the cross product with that vector. In the case that $\boldsymbol{p} = \begin{bmatrix} x & y & z \end{bmatrix}^{\mathrm{T}}$ you should have

$$\texttt{CrossOp}(\boldsymbol{p}) = [\boldsymbol{p}]_{\times} = \begin{bmatrix} 0, & -z, & y \\ z, & 0, & -x \\ -y, & x, & 0 \end{bmatrix}. \tag{8}$$

As always, verify that your function works. In this case, you can testing it on random vectors and ensure that

$$[\boldsymbol{p}_1]_{\times}\boldsymbol{p}_2 = \boldsymbol{p}_1 \times \boldsymbol{p}_2. \tag{9}$$

Exercise 3.3

Compute the fundamental matrix $\boldsymbol{F}$ of the two cameras.

Exercise 3.4

What is the epipolar line $\boldsymbol{l}$ of $\boldsymbol{q}_1$ in camera two?

Exercise 3.5

Is $\boldsymbol{q}_2$ located on the epipolar line from Exercise 3.4? Do the computations, but also explain why this *must* be so.

Exercise 3.6

Now assume that both camera one and two have local coordinate systems that are different from the coordinate system of the world.

Let $\boldsymbol{Q}$ and $\tilde{\boldsymbol{Q}}$ denote *the same* 3D point in world space and in the frame of camera one. In other words we have relation

$$\tilde{\boldsymbol{Q}} = \begin{bmatrix} \boldsymbol{R}_1 & \boldsymbol{t}_1 \\ \boldsymbol{0} & 1 \end{bmatrix} \boldsymbol{Q}. \tag{13}$$

Make sure you understand why this is true.

Show analytically that

$$\boldsymbol{Q} = \begin{bmatrix} \boldsymbol{R}_1^{\mathrm{T}} & -\boldsymbol{R}_1^{\mathrm{T}}\boldsymbol{t}_1 \\ \boldsymbol{0} & 1 \end{bmatrix} \tilde{\boldsymbol{Q}}. \tag{14}$$

Exercise 3.7

Show that the projection can work only in the coordinate system of camera one, by showing that we can project points with

$$\boldsymbol{q}_1 = \boldsymbol{K}\begin{bmatrix}\boldsymbol{I} & \boldsymbol{0}\end{bmatrix}\tilde{\boldsymbol{Q}}, \text{ and } \boldsymbol{q}_2 = \boldsymbol{K}\begin{bmatrix}\tilde{\boldsymbol{R}}_2 & \tilde{\boldsymbol{t}}_2\end{bmatrix}\tilde{\boldsymbol{Q}}, \tag{18}$$

where

$$\tilde{\boldsymbol{R}}_2 = \boldsymbol{R}_2\boldsymbol{R}_1^{\mathrm{T}}, \text{ and } \tilde{\boldsymbol{t}}_2 = \boldsymbol{t}_2 - \boldsymbol{R}_2\boldsymbol{R}_1^{\mathrm{T}}\boldsymbol{t}_1. \tag{19}$$

Applied epipolar geometry

Exercise 3.8

Load the file TwoImageDataCar.npy, and compute the fundamental matrix between the images.
Tip: You can load the file with:
`np.load('TwoImageDataCar.npy', allow_pickle=True).item()`

Exercise 3.9

Write code that can show both images at the same time. Now write code such that you can click on a point in image one, and display the corresponding epipolar line in image two. Experiment with your code, verifying that the point you click on is on the epipolar line in the other image.

Tip: To click on a point and get the pixel coordinates you can use `plt.ginput(1)`
This needs the figure to open in a new window, so you might need to run `%matplotlib qt` first

Tip: To draw a line given in homogeneous coordinates, you can use the `DrawLine` function below. It takes as input a line in homogeneous coordinates `l` and the size of the image it will be drawn on, as returned by `im.shape`.

```
def DrawLine(l, shape):
    #Checks where the line intersects the four sides of the image
    # and finds the two intersections that are within the frame
    def in_frame(l_im):
        q = np.cross(l.flatten(), l_im)
        q = q[:2]/q[2]
        if all(q>=0) and all(q+1<=shape[1::-1]):
            return q
    lines = [[1, 0, 0], [0, 1, 0], [1, 0, 1-shape[1]], [0, 1, 1-shape[0]]]
    P = [in_frame(l_im) for l_im in lines if in_frame(l_im) is not None]
    if (len(P)==0):
        print("Line is completely outside image")
    plt.plot(*np.array(P).T)
```

Exercise 3.10

Do the same thing as the last exercise, but where you can click in image two and get the epipolar line displayed in image one. You do not compute a new fundamental matrix to do this.

Programming exercise: Triangulation

Exercise 3.11

Create a function `triangulate`. It should be able to triangulate a single 3D point that has been seen by n different cameras. The function should take as input: a list of n pixel coordinates (`q1`, `q2`, ..., `qn`), and a list of n projection matrices (`P1`, `P2`, ..., `Pn`), and should return the triangulation of the point in 3D using the linear algorithm.

Test your function by defining a 3D point, project this point to the image planes of the two cameras, and then triangulate it using the projection. Try reprojecting your estimated 3D point to the cameras. Do you find the same 2D pixels?

Solutions

Answer of exercise 3.1

The projections are

$$\boldsymbol{p}_1 = \begin{bmatrix} 550 \\ 325 \end{bmatrix}, \text{ and} \tag{6}$$

$$\boldsymbol{p}_2 = \begin{bmatrix} 582.473 \\ 185.990 \end{bmatrix}. \tag{7}$$

Answer of exercise 3.3

The fundamental matrix is

$$\boldsymbol{F} = \begin{bmatrix} 3.293 \cdot 10^{-7} & 8.194 \cdot 10^{-7} & 1.792 \cdot 10^{-3} \\ 5.155 \cdot 10^{-7} & -8.769 \cdot 10^{-7} & 9.314 \cdot 10^{-5} \\ -1.299 \cdot 10^{-3} & 1.520 \cdot 10^{-3} & -1.101 \end{bmatrix}. \tag{10}$$

Answer of exercise 3.4

The epipolar line is found by $\boldsymbol{F}\boldsymbol{p}_1$

$$\boldsymbol{l} = \begin{bmatrix} 8.956 \cdot 10^{-3} \\ 3.668 \cdot 10^{-4} \\ -5.285 \end{bmatrix}. \tag{11}$$

Remember that you can get different numbers, if you have a different scale of $\boldsymbol{p}_1$, but the line should be the same up to scale.

Answer of exercise 3.5

To see if a point $\boldsymbol{q}$ is on a line we can use that $\boldsymbol{q}^{\mathrm{T}}\boldsymbol{l} = 0$:

$$\boldsymbol{q}_2^{\mathrm{T}}\boldsymbol{l} = -1.943 \times 10^{-15} \approx 0. \tag{12}$$

Taking numerical precision into account, the point is on the line.

This must be true, since both the point $\boldsymbol{q}_2$ and the line $\boldsymbol{l}$ are derived from the same 3D point $\boldsymbol{Q}$. This 3D point yields a single epipolar plane, and the plane yields a single line in each camera. The projections of the 3D point must lie on the epipolar lines.

Answer of exercise 3.6

We insert Equation 14 into Equation 13:

$$\tilde{\boldsymbol{Q}} = \begin{bmatrix} \boldsymbol{R}_1 & \boldsymbol{t}_1 \\ \boldsymbol{0} & 1 \end{bmatrix} \begin{bmatrix} \boldsymbol{R}_1^{\mathrm{T}} & -\boldsymbol{R}_1^{\mathrm{T}}\boldsymbol{t}_1 \\ \boldsymbol{0} & 1 \end{bmatrix} \tilde{\boldsymbol{Q}}, \tag{15}$$

$$\tilde{\boldsymbol{Q}} = \begin{bmatrix} \boldsymbol{R}_1\boldsymbol{R}_1^{\mathrm{T}} & -\boldsymbol{R}_1\boldsymbol{R}_1^{\mathrm{T}}\boldsymbol{t}_1 + \boldsymbol{t}_1 \\ \boldsymbol{0} & 1 \end{bmatrix} \tilde{\boldsymbol{Q}} \tag{16}$$

$$\tilde{\boldsymbol{Q}} = \begin{bmatrix} \boldsymbol{I} & \boldsymbol{0} \\ \boldsymbol{0} & 1 \end{bmatrix} \tilde{\boldsymbol{Q}}. \tag{17}$$

And we find that it is valid! This is true as the matrices are inverses of each other.

Answer of exercise 3.7

For the first projection in camera one we reduce the projection equation:

$$\boldsymbol{q}_1 = \boldsymbol{K}[\boldsymbol{R}_1|\boldsymbol{t}_1]\boldsymbol{Q}, \tag{20}$$

$$= \boldsymbol{K} \begin{bmatrix} \boldsymbol{I} & 0 \end{bmatrix} \begin{bmatrix} \boldsymbol{R}_1 & \boldsymbol{t}_1 \\ \boldsymbol{0} & 1 \end{bmatrix} \boldsymbol{Q}, \tag{21}$$

$$= \boldsymbol{K} \begin{bmatrix} \boldsymbol{I} & 0 \end{bmatrix} \tilde{\boldsymbol{Q}}, \tag{22}$$

$$\tag{23}$$

For the second projection into camera two we insert Equation 14

$$\boldsymbol{q}_2 = \boldsymbol{K}[\boldsymbol{R}_2|\boldsymbol{t}_2]\boldsymbol{Q}, \tag{24}$$

$$\boldsymbol{q}_2 = \boldsymbol{K}[\boldsymbol{R}_2|\boldsymbol{t}_2] \begin{bmatrix} \boldsymbol{R}_1^{\mathrm{T}} & -\boldsymbol{R}_1^{\mathrm{T}}\boldsymbol{t}_1 \\ \boldsymbol{0} & 1 \end{bmatrix} \tilde{\boldsymbol{Q}}, \tag{25}$$

$$\boldsymbol{q}_2 = \boldsymbol{K} \begin{bmatrix} \boldsymbol{R}_2\boldsymbol{R}_1^{\mathrm{T}} & \boldsymbol{t}_2 - \tilde{\boldsymbol{R}}_2\boldsymbol{t}_1 \end{bmatrix} \tilde{\boldsymbol{Q}}, \tag{26}$$

$$\boldsymbol{q}_2 = \boldsymbol{K} \begin{bmatrix} \tilde{\boldsymbol{R}}_2 & \tilde{\boldsymbol{t}}_2 \end{bmatrix} \tilde{\boldsymbol{Q}}. \tag{27}$$

Answer of exercise 3.9

You should get the following fundamental matrix

$$\boldsymbol{F} = \begin{bmatrix} -1.502 \cdot 10^{-8} & -3.460 \cdot 10^{-7} & -3.476 \cdot 10^{-5} \\ -2.068 \cdot 10^{-7} & 3.963 \cdot 10^{-8} & -9.296 \cdot 10^{-4} \\ 2.616 \cdot 10^{-5} & 1.122 \cdot 10^{-3} & 1.174 \cdot 10^{-2} \end{bmatrix} \tag{28}$$

Exercise 4: Camera calibration

02504 Computer vision

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

February 23, 2024

These exercises will take you through:

Direct linear transform (DLT), linear algorithm for camera calibration and

checkerboard calibration, and bundle adjustment from Zhang (2000).

You should be able to perform camera calibration using both methods.

Mathematical exercises: Direct linear transform (DLT)

In this section consider the 3D points

$$\boldsymbol{Q}_{ijk} = \begin{bmatrix} i \\ j \\ k \end{bmatrix}, \tag{1}$$

where $i \in \{0, 1\}$, $j \in \{0, 1\}$, and $k \in \{0, 1\}$. Consider also a camera with $f = 1000$ and a resolution of 1920×1080. Furthermore, the camera is transformed such that

$$\boldsymbol{R} = \begin{bmatrix} \sqrt{1/2} & -\sqrt{1/2} & 0 \\ \sqrt{1/2} & \sqrt{1/2} & 0 \\ 0 & 0 & 1 \end{bmatrix} \text{ and } \boldsymbol{t} = \begin{bmatrix} 0 \\ 0 \\ 10 \end{bmatrix}. \tag{2}$$

Exercise 4.1

Find the projection matrix $\boldsymbol{\mathcal{P}}$ and the projections $\boldsymbol{q}$.

Exercise 4.2

Write a function `pest` that uses $\boldsymbol{Q}$ and $\boldsymbol{q}$ to estimate $\boldsymbol{\mathcal{P}}$ with the DLT. Do not normalize your points.

Use the estimated projection matrix $\boldsymbol{P}_{est}$ to project the points $\boldsymbol{Q}$, giving you the reprojected points $\boldsymbol{q}_{est}$.
What is the overall reprojection error $\sqrt{\sum \frac{1}{n} ||\boldsymbol{q}_{\text{est}} - \boldsymbol{q}||_2^2}$ (RMSE)?
Does normalizing the 2D points before estimating $\boldsymbol{P}$ (like we did for the homography) improve the reprojection error?

Programming exercises: Checkerboard calibration

Here we will perform camera calibration with checkerboards. We do not yet have the ability to detect checkerboards, so for now we will define the points ourselves.

Exercise 4.3

Define a function `checkerboard_points(n, m)` that returns the 3D points

$$\boldsymbol{Q}_{ij} = \begin{bmatrix} i - \frac{n-1}{2} \\ j - \frac{m-1}{2} \\ 0 \end{bmatrix}, \tag{7}$$

where $i \in \{0, \ldots, n-1\}$ and $j \in \{0, \ldots, m-1\}$. The points should be returned as a $3 \times (n \cdot m)$ matrix and their order does not matter. These points lie in the $z = 0$ plane by definition.

Exercise 4.4

Let $\boldsymbol{Q}_\Omega$ define a set of corners on a checkerboard. Then define three sets of checkerboard points $\boldsymbol{Q}_a$, $\boldsymbol{Q}_b$, and $\boldsymbol{Q}_c$, where

$$\boldsymbol{Q}_a = \boldsymbol{\mathcal{R}}\left(\tfrac{\pi}{10}, 0, 0\right) \boldsymbol{Q}_\Omega, \tag{8}$$
$$\boldsymbol{Q}_b = \boldsymbol{\mathcal{R}}(0, 0, 0) \boldsymbol{Q}_\Omega, \tag{9}$$
$$\boldsymbol{Q}_c = \boldsymbol{\mathcal{R}}\left(-\tfrac{\pi}{10}, 0, 0\right) \boldsymbol{Q}_\Omega, \tag{10}$$
$$\tag{11}$$

where

$$\boldsymbol{\mathcal{R}}(\theta_x, \theta_y, \theta_z) = \begin{bmatrix} \cos(\theta_z) & -\sin(\theta_z) & 0 \\ \sin(\theta_z) & \cos(\theta_z) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos(\theta_y) & 0 & \sin(\theta_y) \\ 0 & 1 & 0 \\ -\sin(\theta_y) & 0 & \cos(\theta_y) \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta_x) & -\sin(\theta_x) \\ 0 & \sin(\theta_x) & \cos(\theta_x) \end{bmatrix}. \tag{12}$$

Recall that you can compute $\boldsymbol{\mathcal{R}}$ with `scipy` as follows:

```
from scipy.spatial.transform import Rotation
R = Rotation.from_euler('xyz', [θx, θy, θz]).as_matrix()
```

The points should look like Figure 1.

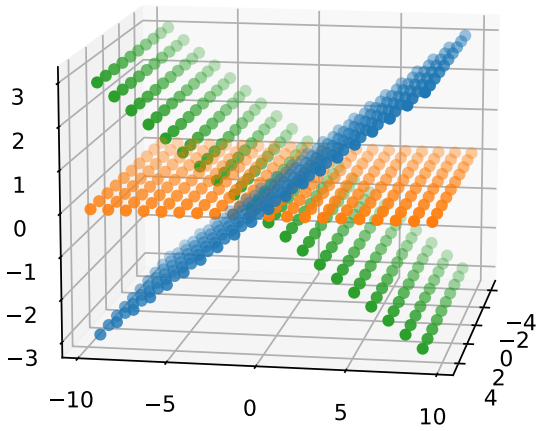

Figure 1: A 3D plot of $\boldsymbol{Q}_a$, $\boldsymbol{Q}_b$, and $\boldsymbol{Q}_c$. In this case $n = 10$, $m = 20$.

Using the projection matrix from Exercise 4.1, project all the checkerboard points to the image plane, obtaining: $\boldsymbol{q}_a$, $\boldsymbol{q}_b$, and $\boldsymbol{q}_c$.

We will now go through the method outlined in Zhang's method[1] step by step.

Exercise 4.5

Define a function `estimateHomographies(Q_omega, qs)` which takes the following input arguments:

- `Q_omega`: an array original un-transformed checkerboard points in 3D, for example $\boldsymbol{Q}_\Omega$.
- `qs`: a list of arrays, each element in the list containing $\boldsymbol{Q}_\Omega$ projected to the image plane from different views, for example `qs` could be $[\boldsymbol{q}_a, \boldsymbol{q}_b, \boldsymbol{q}_c]$.

The function should return the homographies that map from `Q_omega` to each of the entries in `qs`. The homographies should work as follows:

$$\boldsymbol{q} = \boldsymbol{H}\tilde{\boldsymbol{Q}}_\Omega, \tag{13}$$

where $\tilde{\boldsymbol{Q}}_\Omega$ is $\boldsymbol{Q}_\Omega$ without the z-coordinate, in homogeneous coordinates. Remember that we need multiple orientations of checkerboards e.g. rotated and translated.

Use your function `hest` from week 2 to estimate the individual homographies. Your should return a list of homographies; one homography for each checkerboard orientation.

Test your function using $\boldsymbol{Q}_\Omega$, $\boldsymbol{q}_a$, $\boldsymbol{q}_b$, and $\boldsymbol{q}_c$. Check that the estimated homographies are correct with Equation 13.

Exercise 4.6

[1] Zhang, Zhengyou. "A flexible new technique for camera calibration." IEEE Transactions on pattern analysis and machine intelligence 22.11 (2000): 1330-1334.

Now, define a function `estimate_b(Hs)` that takes a list of homographies `Hs` and returns the vector $\boldsymbol{b}$. Form the matrix $\boldsymbol{V}$. This is the coefficient matrix used to estimate $\boldsymbol{b}$ using SVD.

Test your function with the homographies from previous exercise. See if you get the same result as by constructing $\boldsymbol{B}_{\text{true}} = \boldsymbol{K}^{-\text{T}}\boldsymbol{K}^{-1}$, and converting this into $\boldsymbol{b}_{\text{true}}$.

Is $\boldsymbol{b}$ a scaled version of $\boldsymbol{b}_{\text{true}}$?

Suggestions for debugging:

- Check that $\boldsymbol{v}_{11} \cdot \boldsymbol{b}_{\text{true}} = \boldsymbol{h}_1^{\text{T}}\boldsymbol{B}_{\text{true}}\boldsymbol{h}_1$
- Be aware that $\boldsymbol{v}_{\alpha\beta}$ use 1-indexing, while your code might not.

Exercise 4.7

Next, define a function `estimateIntrisics(Hs)` that takes a list of homographies `Hs` and returns a camera matrix $\boldsymbol{K}$. Use your `estimate_b` from the previous exercise. From $\boldsymbol{b}$, estimate the camera matrix $\boldsymbol{K}$ (they use $\boldsymbol{A}$ in the paper). Find the solution in Appendix B from the paper.

Test your function with the homographies from Exercise 4.5. Do you get the original camera matrix?

Exercise 4.8

Now, define a function `Rs, ts = estimateExtrinsics(K, Hs)` that takes the camera matrix $\boldsymbol{K}$ and the homographies `Hs` and returns the rotations `Rs` and translations `ts` of each checkerboard. Use the formulas given in the paper but you do not need to bother with Appendix C — we can live with the error.

Remember that the rotations should not be identical to $\boldsymbol{R}_a$, $\boldsymbol{R}_b$ and $\boldsymbol{R}_c$ as the camera is also rotated. What do you expect the rotations to be? What kind of rotations do you get, and are they valid?

Join the functions to make a larger function `K, Rs, ts = calibratecamera(qs, Q)` that finds the camera intrinsics and extrinsics from the checkerboard correspondences `q` and `Q`.

Exercise 4.9

Use your function to estimate $\boldsymbol{K}$ and $\boldsymbol{R}$ and $\boldsymbol{t}$ for each view. Use these to project $\boldsymbol{Q}_\omega$ thus re-obtaining $\boldsymbol{q}_a$, $\boldsymbol{q}_b$ and $\boldsymbol{q}_c$, and verify that you get the same points by computing the reprojection error.

Exercise 4.10

Finally, we want to check if the function you've created can also work in real life, where we cannot assume that we have perfect 2D points. Therefore, add noise to your ground truth 2D points from a normal distribution with mean $\mathbf{0}$ and standard deviation of $\mathbf{1}$, to obtain $\tilde{\boldsymbol{q}}_a$, $\tilde{\boldsymbol{q}}_b$ and $\tilde{\boldsymbol{q}}_c$. Use these noisy versions with your `calibratecamera` function, and verify that you are still able to estimate a camera matrix that is almost correct.

Solutions

Answer of exercise 4.1

The camera matrix is

$$\begin{bmatrix} 1000 & 0 & 960 \\ 0 & 1000 & 540 \\ 0 & 0 & 1 \end{bmatrix}. \tag{3}$$

The projection matrix is

$$\boldsymbol{\mathcal{P}} = \begin{bmatrix} 707.11 & -707.11 & 960 & 9600 \\ 707.11 & 707.11 & 540 & 5400. \\ 0 & 0 & 1 & 10 \end{bmatrix}, \tag{4}$$

and the projections are

$$\boldsymbol{q}_{000} = \begin{bmatrix} 960 \\ 540 \end{bmatrix}, \quad \boldsymbol{q}_{001} = \begin{bmatrix} 960 \\ 540 \end{bmatrix}, \quad \boldsymbol{q}_{010} = \begin{bmatrix} 889.29 \\ 610.71 \end{bmatrix}, \quad \boldsymbol{q}_{011} = \begin{bmatrix} 895.72 \\ 604.28 \end{bmatrix}, \tag{5}$$

$$\boldsymbol{q}_{100} = \begin{bmatrix} 1030.71 \\ 610.71 \end{bmatrix}, \quad \boldsymbol{q}_{101} = \begin{bmatrix} 1024.28 \\ 604.28 \end{bmatrix}, \quad \boldsymbol{q}_{110} = \begin{bmatrix} 960 \\ 681.42 \end{bmatrix}, \quad \text{and } \boldsymbol{q}_{111} = \begin{bmatrix} 960. \\ 668.56 \end{bmatrix}. \tag{6}$$

Answer of exercise 4.2

The projection matrix and the reprojections are identical to the original within machine precision. The reprojection error on my system is approximately 10^{-11} without normalization and 10^{-14} with normalization.

Answer of exercise 4.10

In one instance of random noise, the difference between the estimated and ground truth camera matrix was:

$$\boldsymbol{K} - \boldsymbol{K}_{true} = \begin{bmatrix} 4.59 & -0.53 & -0.16 \\ 0 & 4.67 & 0.39 \\ 0 & 0 & 0 \end{bmatrix}$$

Exercise 5: Nonlinear optimization, camera calibration

02504 Computer vision

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

March 1, 2024

These exercises will take you through:

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

checkerboard calibration, in real life with OpenCV.

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.

$$\begin{bmatrix} \Pi(\boldsymbol{P}_1\boldsymbol{Q}_h) - \tilde{\boldsymbol{q}}_1 \\ \Pi(\boldsymbol{P}_2\boldsymbol{Q}_h) - \tilde{\boldsymbol{q}}_2 \\ \vdots \\ \Pi(\boldsymbol{P}_n\boldsymbol{Q}_h) - \tilde{\boldsymbol{q}}_n \end{bmatrix}$$

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.

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.

Be aware that most phones rotate the image to make it appear correctly on a computer (portrait or landscape). Therefore, try to hold the phone so it rotates all images in the same way during capture.

Exercise 5.6

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

Check that all images have the same dimensions, to see if it has rotated some of them inadvertently, and discard these images.

Why is it a problem for the camera calibration if two images are rotated(in software) differently by the camera (for example portrait and landscape)?

Exercise 5.7

Now, let's detect checkerboards in your images. Use the function `cv2.findChessboardCorners`.

Start with the image where the checkerboard is seen straight on and resize the image to a smaller resolution

```
im_small = cv2.resize(im, None, fx=0.25, fy=0.25)
```

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).

Run `cv2.findChessboardCorners` on all of your images.

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

Exercise 5.8

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).

`objpoints` should be a list of $(n \cdot m) \times 3$ arrays in `np.float32`. Each list element contains the 3D points of the checkerboard for each camera (from `checkerboard_points`).
`imgpoints` should be a list of $(n \cdot m) \times 1 \times 2$ arrays in `np.float32` from `cv2.findChessboardCorners`.

```
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?

Tip: Make sure that the order of points from `checkerboard_points(n, m)` matches with the order that they are returned by `cv2.findChessboardCorners`.

Exercise 5.9

Reproject the checkerboard corners to the images. You can use your `projectpoints` function from week 2.
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 on top of the original image. Your RMSE should not be more than a few pixels.

Exercise 5.10

Using the `box3d` function from week 1, create a new set of points like so

```
Q = 2*box3d() + 1
```

For one of your pictures, use the estimated $\boldsymbol{R}$ and $\boldsymbol{t}$ to project these points to the image and visualize the result. It should look similar to Figure 1.

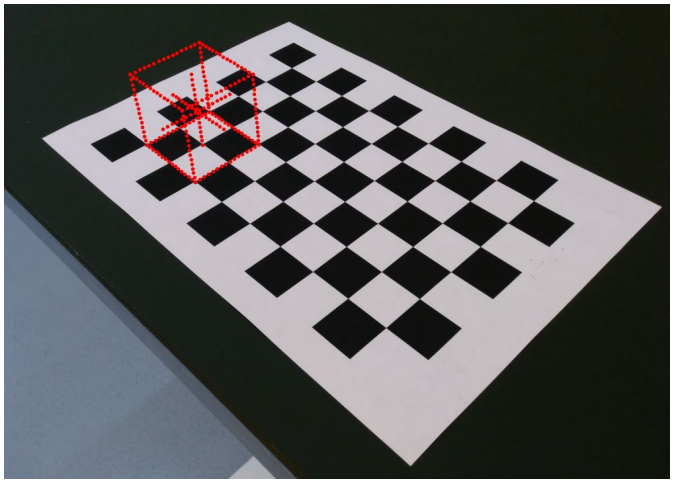

Figure 1: The points from `box_3d` projected to the camera.

Exercise 5.11

Do the camera calibration again, this time allowing the first order distortion coefficient k_1.

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

$$\tilde{\boldsymbol{Q}} = \begin{bmatrix} 1.015 \\ 0.9853 \\ 2.9 \cdot 10^{-4} \end{bmatrix}$$

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

$$\hat{\boldsymbol{Q}} = \begin{bmatrix} 1.00153897e+00 \\ 9.98546320e-01 \\ 4.27473316e-05 \end{bmatrix}$$

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.

Answer of exercise 5.6

A digital camera rotating images automatically is a problem because when you rotate an image 90° the camera matrix changes.

Exercise 6: Simple Features

02504 Computer vision

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

March 8, 2024

Learning objectives

These exercises will introduce you to feature extraction. In this exercise you will write the code for the Harris corner detector as well as try the Canny edge detector.

Programming exercise: Harris corner detector

You will implement the Harris corner detector and apply it to the image `TestIm1.png` shown in Figure 1 to test that your implementation is correct. Furthermore you should test the Harris corner detector on some of the other images given in the data folder for this exercise to see

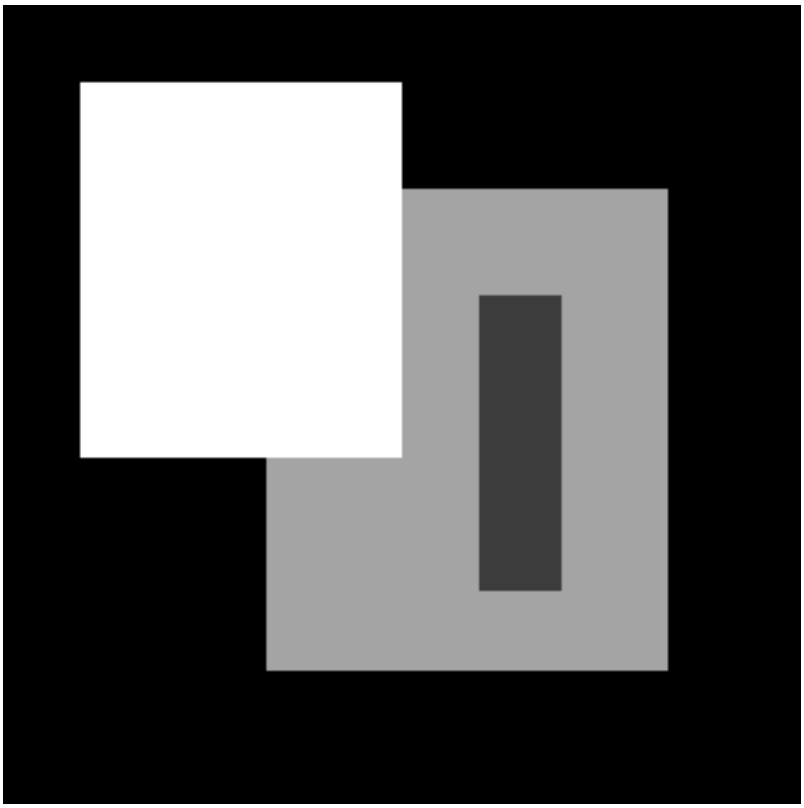

Figure 1: An image for testing your Harris corner detector.

Exercise 6.1

To start with, create the function `g, gd = gaussian1DKernel(sigma)`, where `g` is the 1D Gaussian kernel, `gd` is the derivative of `g`, and `sigma` is the Gaussian width.

In this function, you have a choice: what length should the Gaussian kernel have? What is the error in the truncation when setting the length to `sigma`, $2 \cdot$ `sigma`, or $6 \cdot$ `sigma`?

No matter which length you end up choosing you should normalize `g` such that it sums to 1.

Exercise 6.2

Now create the function `I, Ix, Iy = gaussianSmoothing(im, sigma)`, where `I` is the Gaussian smoothed image of `im`, and `Ix` and `Iy` are the smoothed derivatives of the image `im`. The `im` is the original image and `sigma` is the width of the Gaussian.

Remember to convert the image `im` to a single channel (greyscale) and floating point, so you are able to do the convolutions.

Using the `g, gd = gaussian1DKernel(sigma)` function, how would you do 2D smoothing?
Tip: Using a 1D kernel in one direction e.g. x is independent of kernels in the other directions.

What happens if `sigma` $= 0$? What should the function return if it supported that?

Use the smoothing function on your test image. Do the resulting images look correct?

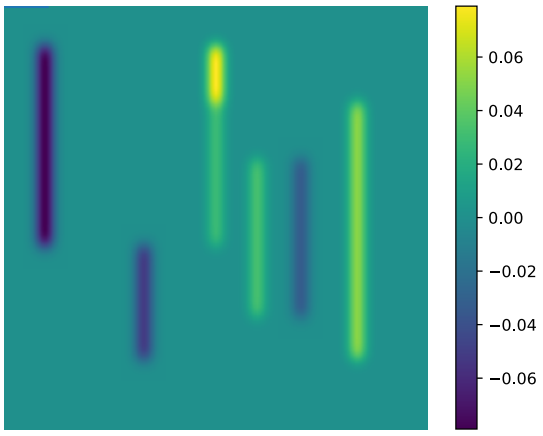

Figure 2: An example of `Ix` for `sigma`=5, when `I` is normalized to the [0,1] range.

Exercise 6.3

Now create the function `C = structureTensor(im, sigma, epsilon)` where

$$\mathtt{C}(x,y) = \begin{bmatrix} g_\epsilon * \mathtt{Ix}^2(x,y) & g_\epsilon * \mathtt{Ix}(x,y)\mathtt{Iy}(x,y) \\ g_\epsilon * \mathtt{Ix}(x,y)\mathtt{Iy}(x,y) & g_\epsilon * \mathtt{Iy}^2(x,y) \end{bmatrix} \tag{1}$$

and $g_\epsilon * \ldots$ is the convolution of a new Gaussian kernel with width `epsilon`.

Use the structure tensor function on your test image. Do the resulting images still look correct?

We use two Gaussian widths in this function: `sigma` and `epsilon`. The first one `sigma` is used to calculate the derivatives and the second one to calculate the structure tensor. Do we need to use both? If you don't know the answer, start on the next exercise and return to this question afterwards.

Exercise 6.4

Create the function `r = harrisMeasure(im, sigma, epsilon, k)` where

$$\mathtt{r}(x,y) = a \cdot b - c^2 - \mathtt{k}\,(a+b)^2\,, \quad \text{where} \tag{2}$$

$$\mathtt{C}(x,y) = \begin{bmatrix} a & c \\ c & b \end{bmatrix} \tag{3}$$

Now return to the question from last exercise.
Tip: What happens to `r` if you set `epsilon` $= 0$? Take a look at Equations 1 to 3. Why is it essential that `epsilon` $\neq 0$?

Use the `harrisMeasure` function on your test image. Recall that $k = 0.06$ is a typical choice of k. Are there large values near the corners?

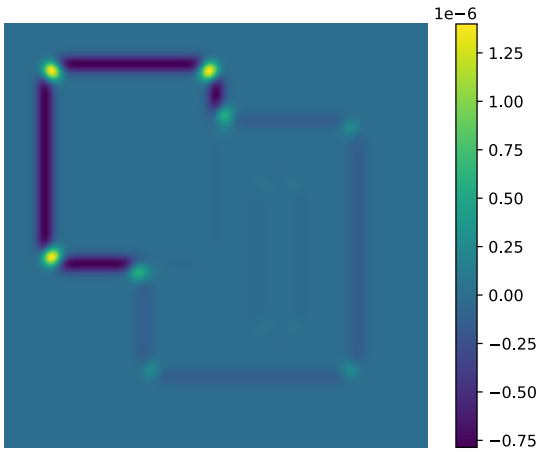

Figure 3: An example of `r`.

Exercise 6.5

Finally, create the function `c = cornerDetector(im, sigma, epsilon, k, tau)` where `c` is a list of points where `r` is the local maximum and larger than some relative threshold i.e.

$$\mathtt{r}(x,y) > \mathtt{tau}.$$

To get local maxima, you should implement non-maximum suppression, see the slides or Sec. 4.3.1 in the LN. Non-maximum suppression ensures that $\texttt{r}(x, y) > \texttt{r}(x \pm 1, y)$ and $\texttt{r}(x, y) > \texttt{r}(x, y \pm 1)$. Once you have performed non-maximum suppression you can find the coordinates of the points using `np.where`.

Use the corner detector on your test image. Does it find all the corners, or too many corners?

Programming exercises: Canny edge detection

Just like many other imaging operations the Canny edge detector is available in both Matlab and OpenCV. Instead of implementing it ourselves, let us start using someone else's implementation.

Exercise 6.6

Figure out how to run the Canny edge detector in your language and apply it to the two images `TestIm1.png` and `TestIm2.png`

Exercise 6.7

What is the effect of the threshold parameters in the Canny edge detector. Try them out specifically on `TestIm2.png`?

Exercise 7: Robust model fitting

02504 Computer vision

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

March 15, 2024

Learning objectives

These exercises will introduce you to robust model fitting. You will find straight lines with the Hough transform, and then implement RANSAC to do the same.

Hough Transform

Here you should extract lines from the image `Box3.bmp` from last week, via the Hough transform. The image should be found together with this exercise. Perform the following steps:

Exercise 7.1

Load the image and detect edges in it. Here the function `cv2.Canny` can be used. Visualize the edges you have detected.

Exercise 7.2

Compute the Hough space from the detected edges. Use the function
`hspace, angles, dists = skimage.transform.hough_line(edges)`
What do the returned values `hspace`, `angles`, `dists` mean?

Exercise 7.3

Visualize the Hough space. To get the correct units on the axes you can use

```
extent = [angles[0], angles[-1], dists[-1], dists[0]]
plt.imshow(hspace, extent=extent, aspect='auto')
```

Exercise 7.4

Find peaks in your Hough space, using `skimage.transform.hough_line_peaks`.
`extH, extAngles, extDists = hough_line_peaks(hspace, angles, dists, num_peaks=n)`
Display your identified peaks on top of the Hough space.

Exercise 7.5

Draw the lines that correspond to the identified peaks on top of your original image. For this you can use the `DrawLine` function from week 3 (repeated below)

```
def DrawLine(l, shape):
    #Checks where the line intersects the four sides of the image
    # and finds the two intersections that are within the frame
    def in_frame(l_im):
        q = np.cross(l.flatten(), l_im)
        q = q[:2]/q[2]
        if all(q>=0) and all(q+1<=shape[1::-1]):
            return q
    lines = [[1, 0, 0], [0, 1, 0], [1, 0, 1-shape[1]], [0, 1, 1-shape[0]]]
    P = [in_frame(l_im) for l_im in lines if in_frame(l_im) is not None]
    if (len(P)==0):
        print("Line is completely outside image")
    plt.plot(*np.array(P).T)
```

RANSAC

Here you should fit a line to a data set consisting of inliers with noise and outliers. Such data is generated by the following function:

```
def test_points(n_in, n_out):
    a = (np.random.rand(n_in)-.5)*10
    b = np.vstack((a, a*.5+np.random.randn(n_in)*.25))
    points = np.hstack((b, 2*np.random.randn(2, n_out)))
    return np.random.permutation(points.T).T
```

It is recommended that you do so in the following steps.

Exercise 7.6

Make a function that fits a line, in homogeneous coordinates, given two points.

Exercise 7.7

Make a function that determines which of a set of 2D points are an inliers or outliers with respect to a given line. The threshold should be supplied as parameter to this function, such that it can easily be tuned later

Exercise 7.8

Make a function that calculates the consensus, i.e. the number of inliers, for a line with respect to a set of points

Exercise 7.9

Make a function that randomly samples two of n 2D points (without replacement).

Exercise 7.10

Assemble the functions made above to a working RANSAC algorithm for estimating lines. Set the number of iterations and the threshold manually.

Exercise 7.11

Experiment with the algorithm, what is a good threshold for distinguishing between inliers and outliers?

Exercise 7.12

Add the final step to your implementation, where you fit a new line to all inliers of the best line. The total least squares fit of a straight line to a set of points is given by the first principal component of them. Consider using the code below to get a homogeneous line along the first principal component.

```
def pca_line(x): # assumes x is a (2 x n) array of points
    d = np.cov(x)[:, 0]
    d /= np.linalg.norm(d)
    l = [d[1], -d[0]]
    l.append(-(l@x.mean(1)))
    return l
```

Exercise 7.13

Implement the stopping criteria for RANSAC as described on the slides. Use $p = 0.99$.

Exercise 8: BLOBs and SIFT features

02504 Computer vision

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

March 22, 2024

Learning objectives

These exercises will introduce you to BLOBs and SIFT features. In this exercise you will write the code for the blob detector as well as use the SIFT feature detector and matcher.

BLOB detector

You will implement a BLOB detector using the Difference-of-Gaussians (DoG) method and apply it to the following image of sunflowers.

Start by loading the image into Python, and converting it to black and white and floating point.
`im = im.astype(float).mean(2)/255`

Exercise 8.1

Create the function `im_scales = scaleSpaced(im, sigma, n)`, where `im_scales` is a list containing the scale space pyramid of the original image `im`. The width and height of all images in the pyramid `im_scales` are exactly the same as the original image `im`. I.e. here we do the naïve implementation with increasing widths of Gaussians and no image downsampling, to make the exercise easier. In other words `im_scales` is not a pyramid in image sizes; only in scale space.

This function should apply a Gaussian kernel with standard deviation $\texttt{sigma} \cdot 2^i$, where $i = 0, 1, \ldots, \texttt{n} - 1$.

Exercise 8.2

Now, create the function `DoG = differenceOfGaussians(im, sigma, n)`, where `DoG` is a list of scale space DoGs of the original image `im`. Like the `scaleSpaced` function, the returned images are all the same size as the original.

Exercise 8.3

Finally, create the function `blobs = detectBlobs(im, sigma, n, tau)`, where `blobs` are the BLOBs (pixels) of the original image `im` with a DoG magnitude larger than a threshold `tau`. The function should use non-maximum suppression to only give a single response for each BLOB. Reuse or extend your code from week 6 to do the non-maximum suppression for the 8 neighbours in the same scale. Instead of checking against all nine neighbours in the scale above and below, you can apply a 3×3 max filter to all scales above and below, and only compare against the center pixel in the max filtered image above and below.

```
MaxDoG[i] = cv2.dilate(abs(DoG[i]), np.ones((3,3)))
```

Try the detector on the image of sunflowers. Visualize your result by drawing a circle for each BLOB, with the radius proportional to the scale of the BLOB. You can use `cv2.circle` for this.

Figure 1: Detected blobs with `sigma`=2, `n`=7, `tau`=0.1. You may have slight deviations in the detected blobs depending on how you choose

Using SIFT features

The SIFT feature detector and descriptor are quite difficult to implement, so we will use existing implementations. However, first we need a good test case scenario.

Exercise 8.4

Create the function `r_im = transformIm(im, theta, s)`, where `r_im` is a scaled and rotated version of the original image `im`. In this case, `theta` is a rotation angle and `s` is a scale factor.

Use this function to produce a transformed version of the original image.

Exercise 8.5

Use the SIFT detector to detect features in both the original and the transformed image. Plot the features on top of the images. There are quite a few parameters to play with. Try changing them and see the results.

Now match the features to each other. For this you can use `cv2.BFMatcher()`.

Plot the matches; do they look qualitatively correct?

Filter your matches with the ratio test. Does this remove incorrect matches?

Exercise 8.6

Take two photos of the same scene from different angles using e.g. your smartphone and find matching SIFT features.

Exercise 8.7

This is an optional exercise. Try downloading R2D2 and use it to match features in your images.

Exercise 9: Geometry Constrained Feature Matching

02504 Computer vision

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

April 5, 2024

The Eight Point Algorithm

Here you will write a function that uses the eight point algorithm for fundamental matrix estimation. This should be a slight modification of your algorithm for homography estimation. As for the homography estimation the data points should also be normalized.

Find the fundamental matrix that you computed for the images in `TwoImageData.npy` in week 3 and denote it `Ftrue`.

Exercise 9.1

Implement a function `Fest_8point` that estimates a fundamental matrix from eight or more point correspondences, using the linear algorithm.

Test your function using the points in `Fest_test.npy`. Check that your estimated fundamental matrix is identical to `Ftrue` up to scale and numerical error.

You can load the file using `np.load('Fest_test.npy', allow_pickle=True).item()`.

Feature Matching

Exercise 9.2

Download `TwoImageData.npy` (not the car version). As you did last week, find features in both images and match them. However, this time do not filter matches by the ratio test, only use cross checking as done by `cv2.BFMatcher_create(crossCheck=True)`.

Visualize the result and confirm that it looks reasonable compared to your expectations.

Fundamental matrix estimation via RANSAC

Make a copy of your RANSAC algorithm that fits straight lines, and modify it to fit fundamental matrices instead.

Exercise 9.3

Sample eight random matches. This can be done with the following code
`np.random.choice(matches, 8, replace=False)`.

Use your function `Fest_8point` to estimate the fundamental matrix from these eight matches.

Write a function `SampsonsDistance(F, p1, p2)` that computes Sampson's distance.

Set points to inliers if their Sampson's distance is less than $3.84 \cdot 3\hat{}2$. Explain where this value comes from.

Repeat the above steps for the a fixed number of iterations, such as 200.

Finally, use `Fest_8point` to estimate the final fundamental matrix using all inliers of the best model.

Run your algorithm on the the images from `TwoImageData.npy`. Compare your estimated `F` to `Ftrue` using the following code:
`(F*Ftrue).sum() / (np.linalg.norm(F)*np.linalg.norm(Ftrue))`.
Explain what this code does.

Exercise 9.4

Find the images you captured last week or capture new ones. Match SIFT features between these images using cross checking.

Estimate the fundamental matrix between these images using your RANSAC algorithm.

Comment on how well the fundamental matrix acts as a regularizer on which matches are used.

Exercise 10: Image stitching

02504 Computer vision

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

April 11, 2024

Today's exercise is different from the previous weeks in that it has only a few exercises followed by multiple options for how to improve your algorithm. You are going to implement an algorithm that can stitch at least two images together, by using RANSAC to estimate a homography.

Start out with using `im1` and `im2`.

Exercise 10.1

Find SIFT keypoints (`kp1`, `kp2`) in both images and compute their descriptors (`des1`, `des2`).

Match the SIFT features from both images to each other. Make sure to use cross checking.

Exercise 10.2

Implement a RANSAC algorithm for finding the homography between `im1` and `im2`.

What is the minimum number of matches you need to estimate a homography? (*Tip*: it's four) Explain why this is the case.

Use Equation (2.45) from the lecture notes to compute the distance of a match to a homography.

Assume $\sigma = 3$ and use the formulas from the lecture last week to determine the threshold for when a match should be considered an inlier.

While best practice would be to determine the number of iterations while running the algorithm, you can fix it to i.e. 200.

To verify that the inliers of the best model are reasonable, visualize them. Consider using the following code:

```
plt.imshow(cv2.drawMatches(im1, kp1, im2, kp2, np.array(matches)[bestInliers], None))
```

On the provided images you should find in the ballpark of a thousand inliers.

Exercise 10.3

Wrap your code from the previous exercise in a function:
`H = estHomographyRANSAC(kp1, des1, kp2, des2)`.

The function should take SIFT keypoints and descriptors computed on two images and use RANSAC to estimate the best homography between the images.

Make sure that the function ends with fitting a homography to the largest amount of inliers.

Exercise 10.4

Run the function on `im1` and `im2`. You can use the following function for warping the images using your homography.

```
def warpImage(im, H, xRange, yRange):
    T = np.eye(3)
    T[:2, 2] = [-xRange[0], -yRange[0]]
    H = T@H
    outSize = (xRange[1]-xRange[0], yRange[1]-yRange[0])
    mask = np.ones(im.shape[:2], dtype=np.uint8)*255
    imWarp = cv2.warpPerspective(im, H, outSize)
    maskWarp = cv2.warpPerspective(mask, H, outSize)
    return imWarp, maskWarp
```

It takes an image and a homography and returns the image warped with the homography, where `xRange` and `yRange` specifies for which range of x and y values the image should be sampled. The function returns the transformed version of the image, and a mask that is 1 where the image is valid.

Start out by setting

```
xRange = [0, im1.shape[1]]
yRange = [0, im1.shape[0]]
```

Warp one of your images using the estimated homography. Which image you should warp, depends on if you have found the homography going from image one to two or vice versa. This should warp this image to the other, thus cutting off a lot of the content of that image.

Use the warping function on the other image but set the homography to the identity. Change `xRange` and `yRange` so the images are no longer getting cropped by the warp.

Exercise 10.5

Use the mask returned by the warping function, to generate a single image that contains both images. Where the images overlap you can use the intensities from either image.

Optional algorithm improvements

Congratulations! You have now created a panorama! But you can improve your algorithm. Here are some suggestions.

Exercise 10.6

Devise a way to set `xRange` and `yRange` automatically so image content is not lost.

Tip: You can use the homography to warp points, but which ones?

Exercise 10.7

Expand your algorithm so it's able to handle the situation where three or more images are taken in a line.

Exercise 10.8

Expand your algorithm so it's able to handle arbitrarily overlapping images, such as four images taken in a rectangle (i.e. north-east, north-west, south-west, south east). It's fine to manually designate one of the images as the reference image.

Implement non-linear optimization on top, so the homographies fit well to all overlaps.

Exercise 11: Visual Odometry

02504 Computer vision

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

April 18, 2024

This is a longer exercise, and you have both week 11 and 12 to work on it. The second part of the exercise is again more free where you have options for improving your algorithm.

Download Glyp.zip which contains the images and camera matrix you will need for the exercise.

Load the camera matrix with
`K = np.loadtxt('K.txt')`

Load the first three images (`000001.png`, `000002.png`, `000003.png`) into Python as `im0`, `im1` and `im2`.

Exercise 11.1

Find SIFT keypoints (`kp0`, `kp1`, `kp2`) in all three images and compute their corresponding descriptors (`des0`, `des1`, `des2`). For speed reasons, you can limit the number of SIFT features to 2000. Convert the features to numpy arrays of 2D points
`kp = np.array([k.pt for k in kp])`

Match the SIFT features between `im0` and `im1` (`matches01`), and between `im1` and `im2` (`matches12`). Convert the matches to numpy arrays of the indices
`matches = np.array([(m.queryIdx, m.trainIdx) for m in matches])`.

Exercise 11.2

Estimate the essential matrix between `im0` and `im1` with RANSAC. You can use the OpenCV function `cv2.findEssentialMat` to do this. The `mask` returned by this function indicates which of the matches are inliers.

Decompose the essential matrix and find the correct relative pose (`R1`, `t1`). For this we can again use an OpenCV function namely `cv2.recoverPose`.

The `mask` returned by `cv2.recoverPose` indicates which matches, that lie in front of both cameras. Combine this mask with the mask from `cv2.findEssentialMat`, to get the matches that are both

inliers and lie in front of both cameras. Remove the matches that are not inliers from `matches01`, so that only contains the inliers.

Exercise 11.3

Use `matches01` and `matches12` and find the subset of matches such that we can match features all the way from image 0 to image 2. In other words, create three lists such that `points0[i]`, `points1[i]`, and `points2[i]` are the 2D locations of the same point in the corresponding images. For this you can use

```
_, idx01, idx12 = np.intersect1d(matches01[:,1], matches12[:,0], return_indices=True)
```

Exercise 11.4

For the points that have been tracked through all three images, use the 2D positions in image 0 and 1 to triangulate the points in 3D (`Q`). Using the 2D positions in image 2, estimate the pose of image 2 with RANSAC. Use `cv2.solvePnPRansac` to do this. As the lens distortion is already corrected, you can set `distCoeffs = np.zeros(5)`.

Visualize the 3D points that are also inliers for `solvePnPRansac`.

```
fig = plt.figure()
ax = fig.add_subplot(projection='3d')
ax.scatter(*Q[inliers.flatten()])
```

Also plot the position of the cameras. Recall that the position of the camera is not the translation. How do you find the position?

Expand your algorithm

Now you will expand your algorithm so it can work for more than three images.

For this I suggest that you create lists of store the relevant objects for each frame (`Rs`, `ts`, `kps`, etc.)

Exercise 11.5

Iterate through all images in the folder, repeating the steps in Exercises 11.3 and 11.4 for the previous three images each time.

Visualize all the 3D points and camera positions. Does it look correct?

Optional exercises

Exercise 11.6

Instead of just triangulating the 3D points from the previous two frames, use all frames that the feature has been seen in to triangulate it.

Exercise 11.7

Use the calibration you did of your phone and capture your own sequence of images to try out.

Exercise 11.8

Implement bundle adjustment after your algorithm has run, where you optimize over the camera poses and 3D points in all frames at the same time. Does this improve the result?

Exercise 11.9

For a more challenging evaluation, you can try your code on a sequence from the KITTI dataset. Download KITTI09.zip that contains the images from left camera in sequence 09 in the KITTI dataset, and the corresponding ground truth camera positions and orientations.

Load the ground truth by running

```
gt = np.loadtxt('09.txt').reshape(-1, 3, 4)
```

Now, `gt` contains the ground truth rotation and position (NOT the pose) of each camera in the sequence. The translations are true to scale, so it's allowed to scale your positions to match as good as possible.

Exercise 13: Structured light

02504 Computer vision

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

May 3, 2023

Learning objectives

These exercises will introduce you to a 3D scanning pipeline using the phase shifting encoding. In this exercise you will use code for phase shift decoding, phase matching, and triangulation. Also, this time you will have some real images to play with.

3D scanning

Please download casper.zip which contains a structured light scanning of a baby T-Rex. The cameras are already calibrated and the camera calibration is in `calib.npy`. Load it by calling `c = np.load('calib.npy', allow_pickle=True).item()`

The images are in the folder `sequence`. The first number indicates which camera it is from (0 or 1) and the second number describes the image, where

0	is an image fully illuminated by the projector (projector showing a white image)
1	is an image with the projector fully off
2-17	are the 16 images in the primary pattern shifting. This pattern has 40 periods.
18-25	are the 8 images of the secondary pattern shifting. This pattern has 41 periods.

For example, `frames1_0.png` is the image from camera 1 that is fully illuminated by the projector.

Exercise 13.1

If the cameras were not calibrated, you would need to calibrate them first.

How would you find their intrinsics?

What about the extrinsics?

Exercise 13.2

First we need to rectify the images. First run the following code to initialize the maps for rectification

```
im0 = cv2.imread("sequence/frames0_0.png")
size = (im0.shape[1], im0.shape[0])
stereo = cv2.stereoRectify(c['K0'], c['d0'], c['K1'],
                           c['d1'], size, c['R'], c['t'], flags=0)
R0, R1, P0, P1 = stereo[:4]
maps0 = cv2.initUndistortRectifyMap(c['K0'], c['d0'], R0, P0, size, cv2.CV_32FC2)
maps1 = cv2.initUndistortRectifyMap(c['K1'], c['d1'], R1, P1, size, cv2.CV_32FC2)
```

You can now rectify images by doing `cv2.remap(im, *maps, cv2.INTER_LINEAR)`, where `maps` is either `maps0` or `maps1` depending on which camera the image is coming from. This also handles the undistortion of the images. `P0` and `P1` are the projection matrices for the rectified images, which will come in handy when we need to triangulate the points in 3D.

Now for each image,

- load it,
- convert it floating point and gray-scale, and
- rectify it.

Store the resulting images in two lists (`ims0` and `ims1`), one for each camera.

Show `ims0[0]` and `ims1[0]` side-by-side and verify visually that the images have been rectified.

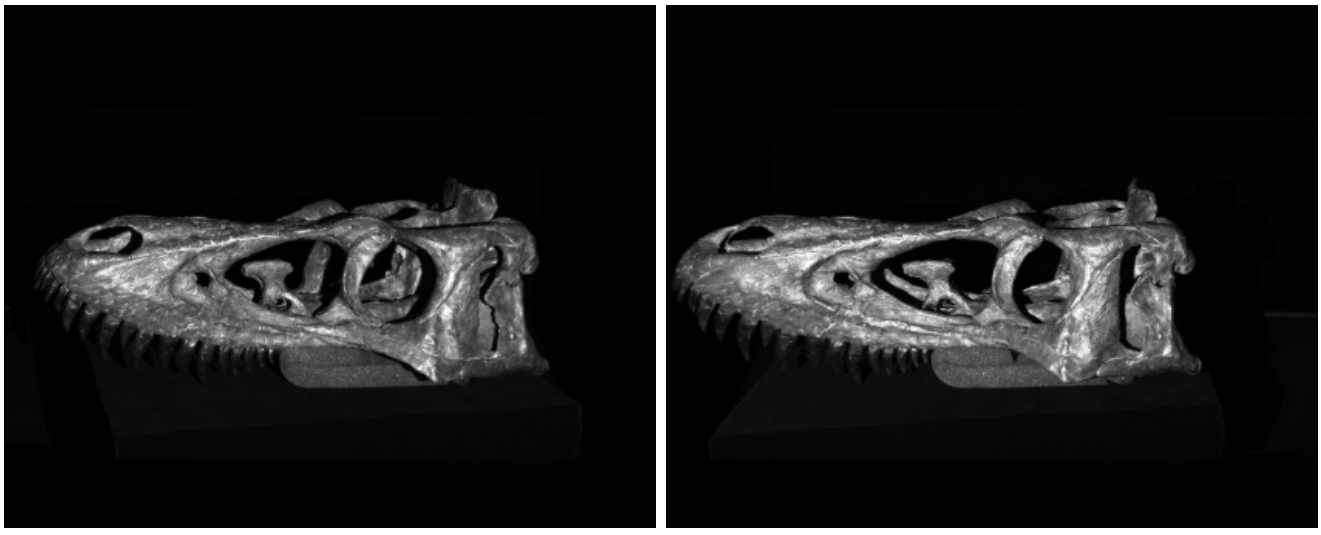

Figure 1: The rectified images with full projector light. Note how features in one image lie on a the same row in the other image, i.e. they are rectified.

Exercise 13.3

At this point, we are ready to make a function that can compute the phases for each camera `theta = unwrap(ims)`. I suggest to write the code for camera 0 and to put it into a function once it's working.

Use indexing to get a list of the primary images out, and make sure it has length 16. Put this list into the Fast Fourier Transform (`np.fft.rfft`) to find the Fourier spectrums of the primary images (`fft_primary`). We use `rfft` as the input is only real numbers. The function can operate on a list of arrays, which is ideal for our situation. Make sure to specify that the FFT should operate along the first dimension of the array (`axis=0`).

The Fourier component corresponding to the pattern is in the second component (`fft_primary[1]`). Get the phase of this using `np.angle` and call it `theta_primary`.

Repeat the same steps for the secondary phase to obtain `theta_secondary`. Compute the *phase cue* (`theta_c`) using the heterodyne principle.

Find the *order* (`o_primary`) of the primary phase.

Use the order of the primary phase to obtain the unwrapped phase (`theta`).

Wrap all of the above into a function `theta = unwrap(ims)` and use it to obtain the phase for both cameras (`theta0` and `theta1`).

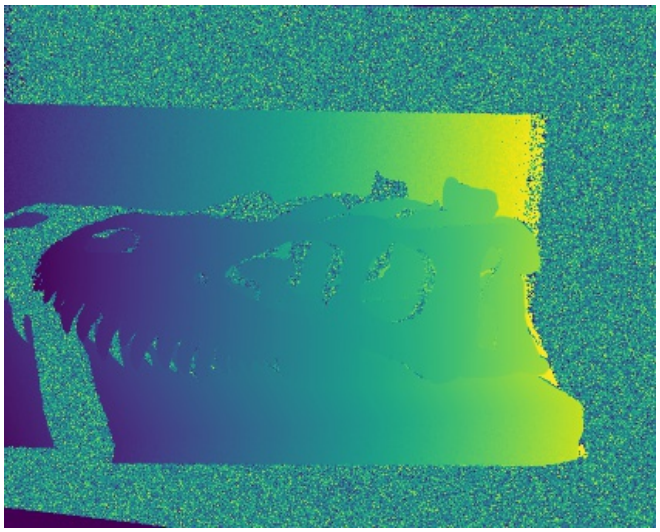

Figure 2: `theta0`. You can also get a solid background instead of noise if you convert to float after converting to black and white. This is also fine.

Exercise 13.4

When inspecting the phase images `theta0` and `theta1` it is clear that not all pixels contain a valid measurement of the phase. This is because some pixels do not reflect enough light from the projector to give a meaningful measurement. To fix this we introduce a binary mask that contains

the areas that are sufficiently illuminated by the projector. Subtract the fully on and fully off projector image from each other (the first two elements of `ims`), to obtain a measurement of how much projector light is in each pixel.

Apply a threshold to this difference image to obtain a mask for each camera (`mask0` and `mask1`). I suggest using a threshold of 15.

Exercise 13.5

Now we need to find matches between the two cameras. As the images are rectified, we can constrain ourselves to search for a match on the corresponding row in the other image. That is we need to create two lists (`q0s` and `q1s`) that contain the pixel coordinates of matches between camera 0 and 1.

Use a double for-loop to iterate over all pixels in camera 0. For each valid pixel (`mask0[i0,j0] = True` which has the phase `theta0[i0,j0]`), we need to find the pixel in the other image that matches the best. As the images are rectified the epipolar line is a the row `i0` in camera 1. Thus, we find the matching pixel in camera 1, selecting the pixel from row `i0` which is valid `mask1[i0,j1] = True` and which has the closest phase match: `theta0[i0,j0]` $\sim$ `theta1[i0,j1]`.

Be aware that the points have to be of the form (x, y) i.e. `[j, i]`, in order to work with our projection matrices.

To verify your matches you can compute the disparity image. This is the image such that `disparity[i0,j0] = j0-j1` for all valid pixels. Initialize it with 0 everywhere, and fill it where you have matches.

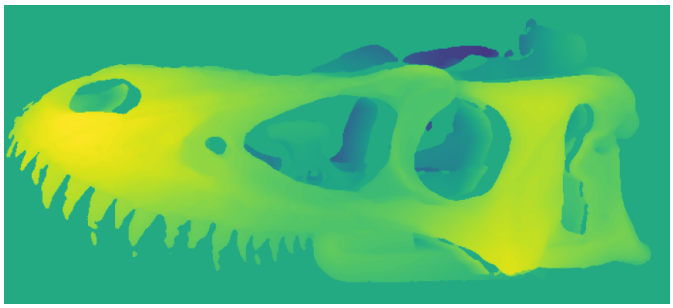

Figure 3: The disparity map visualized. Median filtering has been applied to suppress small errors throwing the colormap off, by doing `cv2.medianBlur(disparity.astype(np.float32), 5)`.

Exercise 13.6

Finally, use the matches from the previous exercise and triangulate the 3D points, using the projection matrices `P0` and `P1` from Exercise 13.2. To triangulate you can use your own function or `cv2.triangulatePoints`. For the OpenCV function you need to convert your lists to arrays, transpose them, and convert them to floating point. The output of `cv2.triangulatePoints` is in homogeneous coordinates.

Remove any points that lie behind the cameras (negative third coordinate of `Q`).

To visualize the points you can use Open3D.
Install it with
`pip install open3d`

Visualize your points in 3D by doing (press [ESC] to close the window again)

```
import open3d as o3d
pcd = o3d.geometry.PointCloud()
pcd.points = o3d.utility.Vector3dVector(Q.T)
o3d.visualization.draw_geometries([pcd])
```

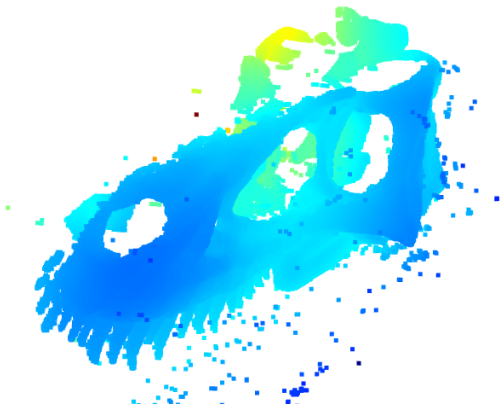

Figure 4: A visualization of the resulting 3D point cloud.

Optional exercises

Exercise 13.7

To make the point cloud more visually pleasing, we want to add color information to it.

As we know which points correspond to which matches, we can sample the color of the points directly in the image. Add this color information to the point cloud visualization.

Exercise 13.8

In Exercise 13.5, you found the camera 1 pixels which were closest to the camera 0 pixels. This is a discrete result i.e. only integer pixels will return. However, we can perform linear interpolation of phases between pixels in camera 1. In this case, sub-pixel precision is obtained by finding the sub-pixel position between two pixels, where the phases match exactly.

Try to implement sub-pixel precision in the your matcher function, and see if you get a less noisy result.