Exercise 2: Stereo Vision

Due: 12.11.2020

Please answer the pen-and-paper questions of the exercise on this answer sheet. In case of insufficient space, use your own paper. The code templates provided for the programming parts are compatible for both Python 2.7 and Python 3.7. **In case of difficulties, don't hesitate to ask for help from the assistants!**

Note: In case you are working on your own PCs, instructions on installing necessary packages can be found at `https://docs.google.com/presentation/d/1HkQl8Jcvi3Bz4dNx8oQ1WcnbAkiOtCOmNhfvmeAbbWo/edit?usp=sharing`.

1 Overview

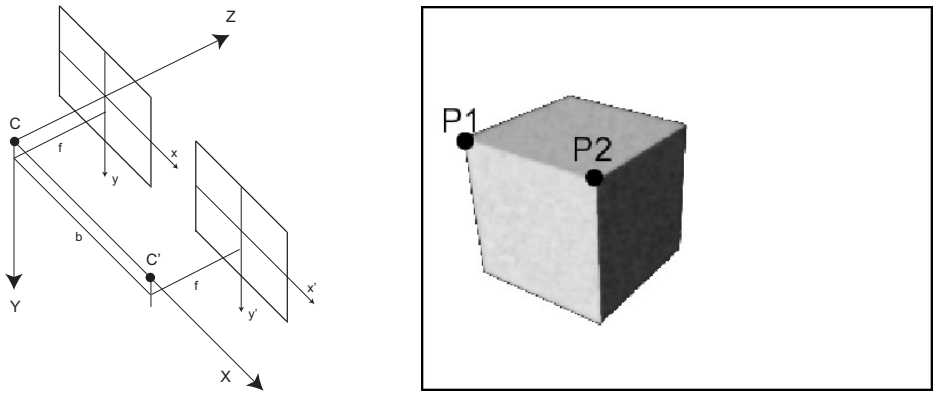
In the preceding exercises, several image processing methods have been introduced. In this exercise, we will now look into the problem of stereo vision. Given two images showing a scene from different points of view, the goal is to obtain 3d world coordinates for each point in the scene. In the theoretical part, we will first determine the external camera parameters. The practical part will then deal with looking for correspondences between the two images and triangulation.

2 Theoretical Exercises

For stereo reconstruction, both the internal and external parameters of each camera must be known. The internal parameters are given as follows: The focal length is 35 mm (1.378 in). The width of the film back equals to 1.417 in and its height is 0.945 in. The external parameters are provided only partly. The missing information has to be derived.

2.1 External Camera Parameters

The camera setup is shown in Fig. 1(a). Determine the position C' of the right camera. In the general case, this task requires the coordinates of three points in space and the image coordinates of these points with respect to the right camera. However, since there is no rotation involved two points are sufficient. In the world coordinate system, point P_1 is located at $(-0.023, -0.261, 2.376)$ and point P_2 at $(0.659, -0.071, 2.082)$. In the image of the right camera (640x480), P_1 has coordinates $(52, 163)$ and P_2 is positioned at $(218, 216)$ - see Fig. 1(b).



(a) (b)

Figure 1: (a) The world coordinate system is defined to be the same as the coordinate system of the left camera (C lies in the origin). Both cameras look in the direction of the positive Z-axis. Neither is rotated. (b) The cube observed by the right camera C'.

As a starting point, you can use the following equations showing the relation between the world coordinates and the image coordinates of a point.

$$\tau \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \begin{bmatrix} X - C_x \\ Y - C_y \\ Z - C_z \end{bmatrix} \tag{1}$$

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} k_{\mathrm{x}} & 0 & x_{\mathrm{center}} \\ 0 & k_{\mathrm{y}} & y_{\mathrm{center}} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} \tag{2}$$

Note that we have assumed $s = 0$. You can obtain k_{x} and k_{y} by using the image resolution and the size of the film back. Do not forget that the center of each image $(x_{\mathrm{center}}, y_{\mathrm{center}})$ at this scale is at $(320, 240)$. Round the calculated camera position with a precision of half an inch. You cannot assume that C' is 0 along the Y and Z directions

Solution

Starting from equation 1 and using the fact that no rotation is involved, one can derive $u = f * \frac{X-C_x}{Z-C_z}$ and $v = f * \frac{Y-C_y}{Z-C_z}$. Solving the first equation with respect to C_x delivers

$$C_x = X - \frac{u(Z - C_z)}{f}. \tag{3}$$

Using two points P_1 and P_2 we get

$$X_1 - \frac{u_1(Z_1 - C_z)}{f} = X_2 - \frac{u_2(Z_2 - C_z)}{f}. \tag{4}$$

Thus,

$$C_z = \frac{(X_1 - X_2)f - u_1 * Z_1 + u_2 * Z_2}{u_2 - u_1}. \tag{5}$$

u_1 and u_2 refer to image coordinates in inch and for their computation, k_x and k_y are needed. Note that k_x is the amount of pixels per inch in horizontal direction. Thus, we get $k_x = 640$ pixel /1.417 in. Similarly, $k_y = 480$ pixel /0.945 in. Next, using equation 2,

$$u_1 = \frac{x_1 - x_{\text{center}}}{k_x} = (52 - 320) * \frac{1}{451.658} = -0.5934. \tag{6}$$

$$u_2 = \frac{x_2 - x_{\text{center}}}{k_x} = (218 - 320) * \frac{1}{451.658} = -0.2258. \tag{7}$$

The result of Equation 5 using the computed u_1 and u_2 is $C_z = -0.0004 \approx 0$. Then, using P_1, equation 3 gives

$$C_x = X_1 - \frac{u_1(Z_1 - C_z)}{f} = -0.023 + 1.0233 = 1.0003 \approx 1 \tag{8}$$

Similarly,

$$C_y = Y_1 - \frac{v_1(Z_1 - C_z)}{f} = -0.261 + 0.2614 = +0.0004 \approx 0, \tag{9}$$

with

$$v_1 = \frac{y_1 - y_{\text{center}}}{k_y} = -77 * \frac{1}{507.937} = -0.152. \tag{10}$$

Result: the position of right camera is $(1.0, 0.0, 0.0)$

3 Practical Exercises

The goal of the practical exercise is to compute a 3d reconstruction for a given image pair. In this exercise we will be dealing with 2 example scenes, the *tsukuba_left.pgm*, *tsukuba_right.pgm* image pair, and the *cube_left.pgm*, *cube_right.pgm* from the previous exercises. Both the image pairs are included with the zip folder. To complete this exercise, you may use either the Python script template `2_stereo_template.py` or the Jupyter notebook template `2_stereo_template.ipynb`, which accompany this handout. **Before you start coding for a particular part (e.g. triangulation), please read carefully the complete description provided for that part! Further, note that some exercises also have related questions that you have to answer in writing.**

3d reconstruction from two views can be split into two sub task, (1) Correspondence: finding corresponding points in the two images and (2) Triangulation: recovering the 3d world coordinates (of the "physical" point) given such corresponding points. We will first implement a function to perform triangulation in exercise 3.1. Next, we will deal with finding correspondences in exercise 3.2 and 3.3. Finally, we will use the above functions perform 3d reconstruction of a scene in exercise 3.4 and 3.5.

3.1 Triangulation

Triangulation is the task of recovering the 3d world coordinates of a "physical" point, given it's coordinates in images from two views. Triangulation requires fully calibrated cameras, i.e. the internal and external camera parameters must be known. Calibration of cameras has been addressed in the theoretical exercise (see Fig. 1(a)). Assume that the two optical camera centers are 1.0 inch apart from each other. The optical axes of the cameras are parallel and their image planes are coplanar, with coincident x-axes.

Implement the function `def triangulate(x_left, x_right, y, m_width, m_height, camera_parameters)` to perform triangulation. The first two parameters correspond to the x-coordinates of projected 3d-points in the left and the right gray scale images. `y` is the common y-coordinate. `m_width` and `m_height` are the width and height of the image. `camera_parameters` is a python dictionary containing the camera parameters, namely `baseline, focal_length, aperture_x` and `aperture_y`. The function must return the world coordinates in inch.

Check the code with the test case provided in the code.

Solution: The code prints whether the test case was passed or not.
The expected solution:

$$\begin{bmatrix} 5.00e-01 & 0.00e+00 & 9.72e-01 \\ 5.00e-01 & 0.00+00 & 3.11e+02 \\ -2.29e-02 & -2.61e-01 & 2.37e+00 \end{bmatrix}$$

3.2 Correspondence

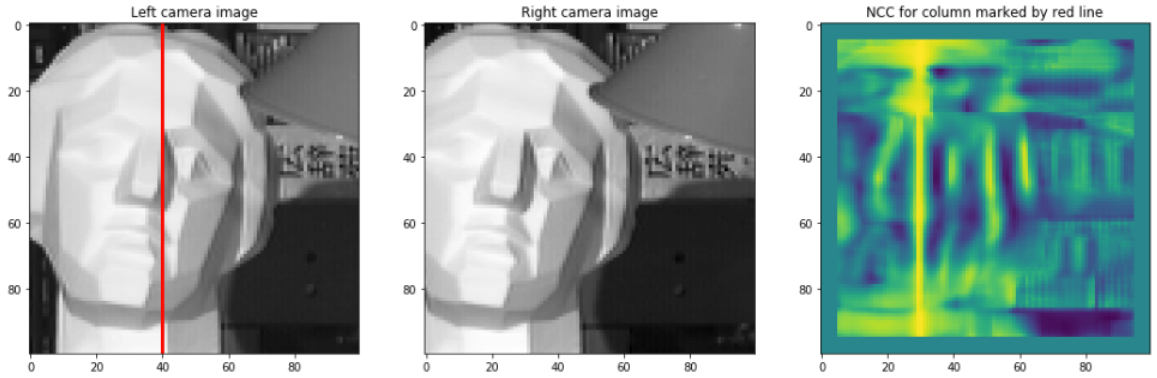
Triangulation requires coordinates of a "physical" point in the images obtained from both the left and the right cameras. Thus, for each point (pixel) in the left image, we want to find the corresponding point in the right image. Given our special camera setup, we know that corresponding points in the two images must have the same y coordinate, which simplifies finding point correspondences. Hence, we can process each horizontal line of the image (ie, each *scan line*) separately. In order to identify corresponding points we need a measure of similarity. In this exercise, we will use the normalised cross-correlation measure. The normalised cross-correlation of two points p_R and p_L (in the two views), is defined as

$$NCC(p_L, p_R) = \frac{1}{|\Delta|\sigma_L\sigma_R}\sum_{\Delta}(I(p_L+\Delta)-\mu_L)\cdot(I(p_R+\Delta)-\mu_R), \tag{11}$$

where Δ defines the (square) neighbourhood of a pixel, $\mu_R = \frac{1}{|\Delta|}\sum_\Delta I(p_R + \Delta)$ and $\sigma_R = \sqrt{\frac{1}{|\Delta|}\sum_\Delta (I(p_R + \Delta) - \mu_R)^2}$.

Complete the function `def compute_ncc(gray_left, gray_right, mask_halfwidth)`. Here, `mask_halfwidth` denotes half the size of the square neighbourhood used for computing NCC. For each patch in the left image, the method should compute its similarity with each a patch in the right image lying in the same scan line. **A detailed description of the function inputs and expected outputs is provided in the function documentation in the code.**

Check the code with the test case provided in the code. Also run it on the *tsukuba_left.pgm* and *tsukuba_right.pgm* image pairs and visualize the computed NCC. The computed NCC should look like figure 2.



(a)

Figure 2: Visualization of NCC for column 40.

Solution: The code prints whether the test case was passed or not.
The expected solution:

$$\begin{bmatrix} 1.0 & -0.5 \\ -0.5 & 1.0 \end{bmatrix}$$

3.3 Fast Correspondence

As you may have noticed, the naive NCC computation approach implemented in exercise 3.2 is quite slow. This is due to the use of "for loop" over NumPy arrays, which is extremely inefficient since Python is an interpreted language. Significant speedups can instead be obtained by using standard NumPy operations over arrays, which are mostly executed in C. For example, consider two approaches given below to compute the mean over the last dimension of a 3-dimensional NumPy array.

```
# Import NumPy
import numpy as np

# Initialize a random NumPy array
a = np.random.rand(1000, 1000, 5)

# Approach 1: Naive for loop
t0 = time.time()
a_mean_naive = np.zeros((1000, 1000))
for i in range(a.shape[0]):
    for j in range(a.shape[1]):
        a_mean_naive[i, j] = a[i, j, :].mean()
```

```
t1 = time.time()
print('Computation took {:.2f} seconds using for loops'.format(t1 - t0))

# Approach 2: Numpy operations
t0 = time.time()
a_mean_numpy = a.mean(axis=2)
t1 = time.time()
print('Computation took {:.2f} seconds using numpy operation'.format(t1 - t0))
```

Even though both approaches are performing the same computation, the second approach is over 200 times faster. Thus, when working with NumPy arrays, it's highly recommended to use standard NumPy operations when possible. Using this knowledge, complete the function `def compute_ncc_fast(gray_left, gray_right, mask_halfwidth)`, a faster version of `compute_ncc` which avoids looping over all pixels in the image. **Refer to the provided skeleton code for hints.**

Question: What speedup do you obtain using `compute_ncc_fast`, as compared to `compute_ncc`?

Solution: A speed-up of > 100 is expected.

3.4 Stereo Reconstruction

Now that we have a way to find correspondences and triangulate 3d world coordinates, the final step is to combine these functions to perform the actual 3d reconstruction of a scene. Implement the function `def define_points_3d(gray_left, gray_right, mask_halfwidth, camera_parameters)` which does the 3d reconstruction. In each scan line, find one corresponding pixel in the right image for each pixel in the left image. Use the normalized correlation coefficient as a similarity measure for the correspondence search. Note that the x-coordinate of the pixel in the right image always has to be smaller than the x-coordinate of the pixel in the left image.

Question: Why is this the case?.

Solution: This follows from perspective geometry and the stereo pair set-up.

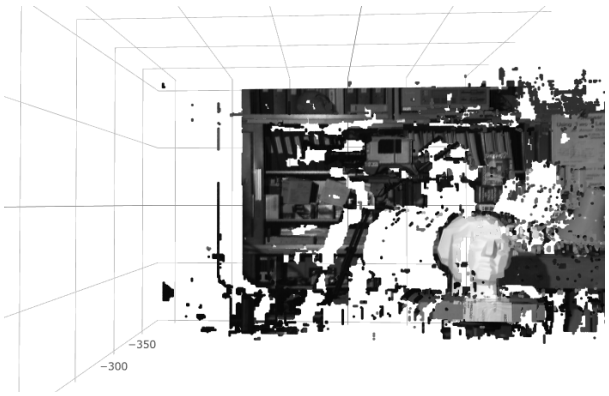
Next, compute a 3d-point by triangulation for every pixel pair. The output should be a NumPy array containing the reconstructed 3d point for each pair of corresponding points. Test your code with the `tsukuba` image pairs. Your results should look like figure 3 for a correlation mask of 11×11 (i.e. `mask_halfwidth` $= 5$).

3.5 Stereo Reconstruction - Cubes

Now, test your 3d reconstruction code on the *cube* image pairs. The results should look like figure 4.

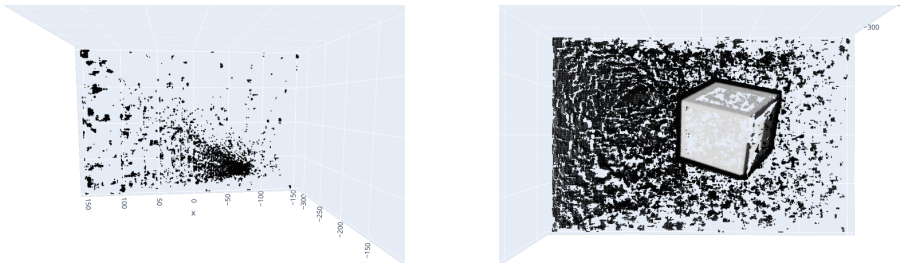
If you zoom in correctly, a wireframe corresponding to the cube should be visible (see figure 4(b)). However, we can see that the 3d reconstruction also contains a lot of noise

Question: Why is the reconstruction noisy?.



(a)

Figure 3: 3d reconstruction of tsukuba.



(a) (b)

Figure 4: 3d reconstruction of cube. (a) The default view plotted by the function. (b) Zoomed in view focusing on the cube.

Solution: Most of the image is plain background, without any structure. Thus, the correspondences obtained for these regions are random, leading to a noisy 3d reconstruction.

Question: Can you suggest any modification of this pipeline, that may lead to better results? You do not need to implement it!

Solution:
Some possible solutions.
1. One could ignore the plain untextured regions during correspondence estimation. i.e. if the variance of a patch is lower than some threshold, ignore those pixels.

2. Look at how 'certain' the estimated correspondence is for a pixel and ignore pixels with uncertain correspondences. i.e. for a point in the left image, if a high NCC score is obtained with multiple points in the right image, then ignore that point.

3. Add some regularization term to ensure smooth variation in the depth.