Vision Aided Navigation - Exercise 1

Prefix:

Over the semester we will gradually develop a large-scale project that employs a suite of algorithms to provide local visual odometry and global trajectory estimation. Simply put, we will find the vehicle locations using only its cameras.

The system will run on the KITTI stereo dataset and will tackle these challenges using image processing, image understanding and 3D optimization techniques to estimate the vehicle path and the geometric structure of the world.

In this exercise we will develop the feature-tracking system, which is a key-component in any vision-based navigation system. Basically, the feature-tracking system receives as input a series of images (or, in our case, stereo pairs) and outputs trajectories of points in the image, where each track represents the different pixel locations of a certain object in the scene.

Part 0

Please open a new folder named “VAN_ex”. All future paths will be relative to it.
Open the following subfolders:

VAN_ex/data
VAN_ex/code
VAN_ex/docs

We will use data from the KITTI benchmark http://www.cvlibs.net/datasets/kitti/. We use the visual odometry sequence #5.
Download it from:
https://drive.google.com/file/d/12XYG2BHRTSSWjZcmI3ypZh9Gs71gDwCi/view?usp=sharing
Into the path: *VAN_ex/data/*
And unzip it there.

First, we’ll examine the first stereo pair:

Left_0:	VAN_ex\data\dataset05\sequences\05\image_0\000000.png
Right_0:	VAN_ex\data\dataset05\sequences\05\image_1\000000.png

Part 1

1.1 Detect at least 1000 key-points on each image of the first stereo-pair. You can use any popular feature detection method. Note that most feature descriptor methods needed in section 1.2 have a coupled detection method.
Present the key-points pixel locations on both images.

```
Useful code:
cv2.ORB_create / cv2.AKAZE_create / cv2.SIFT_create , and many more…
detectAndCompute
```

```
DATA_PATH = r'...\VAN_ex\data\dataset05\sequences\05\\'
def read_images(idx):
  img_name = '{:06d}.png'.format(idx)
  img1 = cv2.imread(DATA_PATH+'image_0\\'+img_name, 0)
  img2 = cv2.imread(DATA_PATH+'image_1\\'+img_name, 0)
  return img1, img2
```

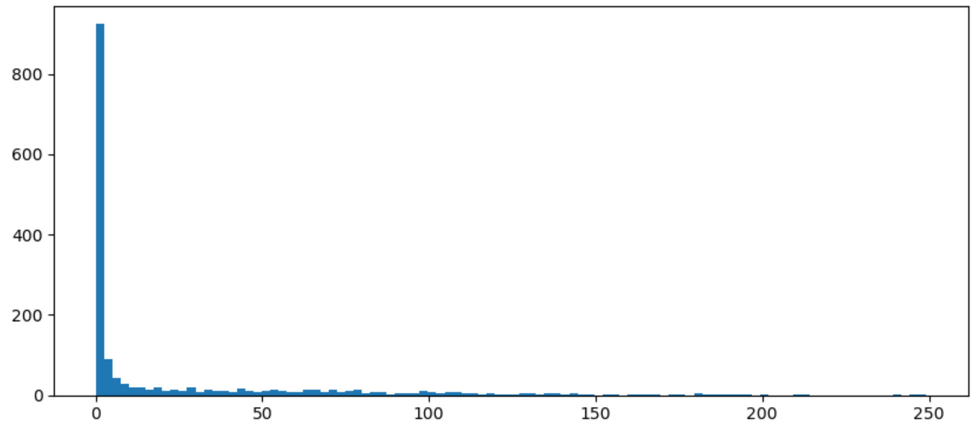
1.2 Calculate feature-descriptors for each key-point in both key-points lists. You can use any popular feature-descriptor method. Print the description of the two first features.

1.3 Match the two descriptors list: find, for each descriptor in the left image, the closest feature in the right image. The distance function should not consider the pixel location, but only the feature descriptor. You can use any popular distance function. Present the first 20 matches as lines connecting the key-point pixel location on the images pair. There may be some mismatches, it is okay for now.

```
Useful code: cv2.BFMatcher, cv2.drawMatches, pyplot.imshow
```

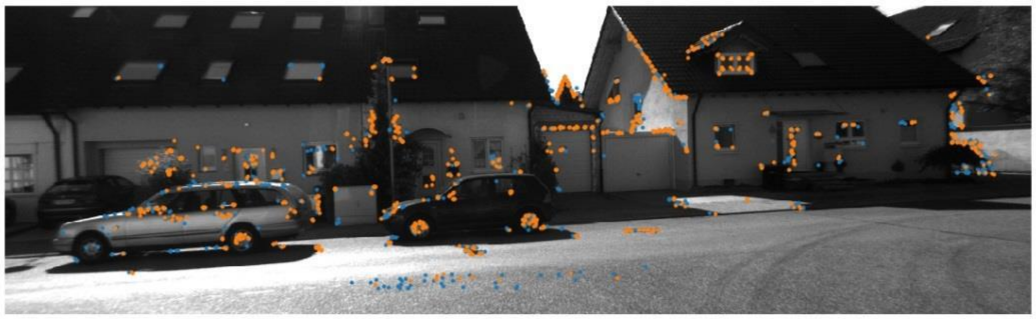
1.4 We are working with a pair of rectified stereo images. Explain the special pattern of correct matches on such images. What is the cause of this pattern?
Create a histogram of the deviations from this pattern for all the matches. Print the percentage of matches that deviate by more than 2 pixels.

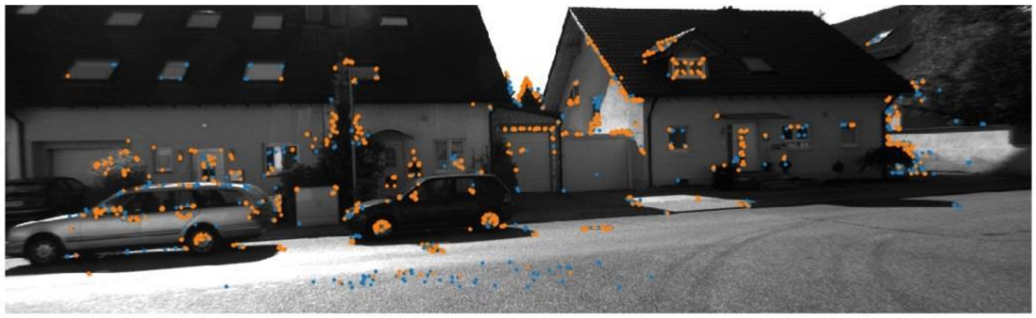
```
Useful code: matplotlib.pyplot.hist
```



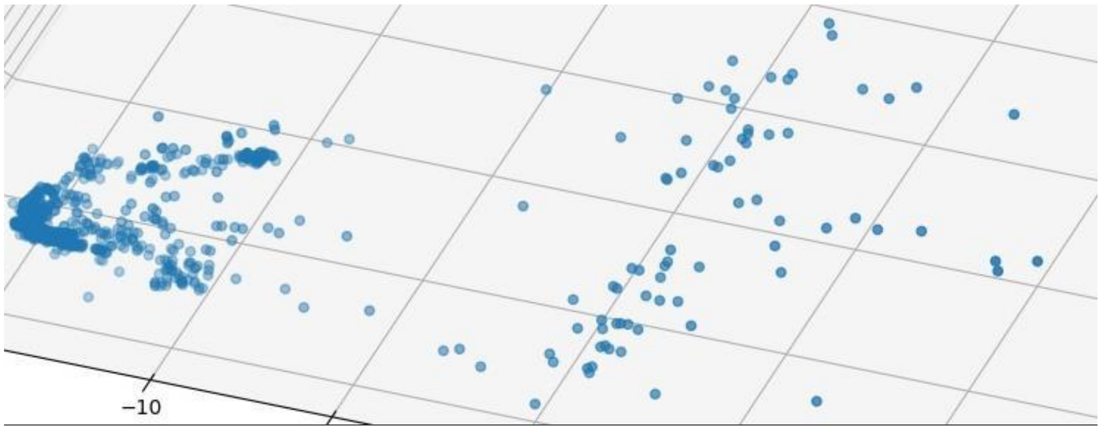
1.5 Use significance test to reject some matches, generate the histogram and output (as in section 1.4) with the resulting matches.

1.6 Use the rectified stereo pattern to reject matches.
Present all the resulting matches as points on the image pair. Accepted matches (inliers) in orange and rejected matches (outliers) in cyan.
Assuming the Y-coordinate of erroneous matches is distributed uniformly across the image, what ratio of matches would you expect to be wrong with this rejection policy?





1.7 Read the relative camera matrices of the stereo cameras from 'calib.txt'.
Use the matches and the camera matrices to define and solve a linear least squares triangulation problem. Do **not** use the opencv triangulation function.
Present a 3D plot of the calculated 3D points.
Repeat the triangulation using 'cv2.triangulatePoints' and compare the results.



```
Useful code:
mpl_toolkits.mplot3d.Axes3D.scatter, cv2.triangulatePoints
```

```
def read_cameras():
  with open(DATA_PATH + 'calib.txt') as f:
    l1 = f.readline().split()[1:]      # skip first token
    l2 = f.readline().split()[1:]      # skip first token
  l1 = [float(i) for i in l1]
  m1 = np.array(l1).reshape(3, 4)
  l2 = [float(i) for i in l2]
  m2 = np.array(l2).reshape(3, 4)
  k = m1[:, :3]
  m1 = np.linalg.inv(k) @ m1
  m2 = np.linalg.inv(k) @ m2
  return k, m1, m2
```

1.8 Look at the matches and 3D points, are there any 3D points you believe to have erroneous location? What in your opinion is the reason for the error?
Can you think of other criteria for outlier removal?