DATA.ML.300 Computer Vision Exercise 4

Miska Romppainen
H274426

February 2021

1 Task: Hough transform and parameter spaces

Questions of task 1 were the following:
a) We have two pixels of an image in the (x, y)-plane: $(x0, y0) = (2, 1)$ and $(x1, y1) = (2, 5)$. Plot the lines that these points create in Cartesian parameter space. What is the point of intersection (m', b') ?
b) Plot the sinusoids that these points create in Polar coordinate parameter space. What is the point of intersection (θ, ρ) when $-\pi/2 \leq \theta \leq \theta/2$ (which I interpreted as a typo of $\pi/2$)?
c) What advantages does the Polar coordinate form have over Cartesian coordinate form?

For the task one I decided to use Matlab in order to solve it instead of writing everything by hand. Below you can see my implementation of the Matlab code. The code is in the exercise zip-file with a name of "Task1.mlx".

Unfortunately importing Matlab code to LaTeX does not print the values nor the outputs. That's why the output values for the task are presented here:
a) The points of intersection are:
$m\_dot = -4/3$
$b\_dot = 5/3$
b) The points of intersection values presented in polar coordinates are:
$rho = \sqrt{\frac{\pi^2}{16} + \frac{9}{2}} = 2.2620$
$theta = \pi + \text{atan}\left(\frac{6\sqrt{2}}{\pi}\right) = 4.3578$

c) What advantages does the Polar coordinate form have over Cartesian coordinate form?
There are multiple different advantages that Polar coordinate form has over the Cartesian coordinate form. One of these advantages is for example, easy transformations. Easy transformation means the rotations and translations are easy to perform in polar coordinates as they only affect the angle component. In Cartesian coordinates, these transformations affect both the x and y components. Other advantage is the circular data presentation by default. Polar

coordinates are good for representing any circular data, as the radial component (ρ) provides a natural representation of distance from the origin.

Matlab Code ($task1.mlx$)

```
% (x1,y1) => (m1, b1)
% (x2, y2) => (m2, b2)
% y = m*x + b => b = -m*x + y

%a)
x1 = - 2
y1 = 1

x2 = 2
y2 = 5

% b1 = -m1*x1 + y1
% b1 = 0
% 0 = 2*m1 + 1 => m1 = -1/2
% m1 = 0
% b1 = 0 + 1 => b1 = 1
m1 = -1/2
b1 = 1

% b2 = -m2*x2 + y2
% => b2 = 0
% 0 = -2*m2 + 5 => m1 = 5/2
% m2 = 0
% b2 = 0 + 5 => b2 = 5
m2 = 5/2
b2 = 5
syms x
m_dot = solve(m2*x+b2 == m1*x+b1, x)
b_dot = m2*m_dot+b2

%b)
syms theta

rho1 = x1*cos(theta) + y1*sin(theta)
rho2 = x2*cos(theta) + y2*sin(theta)

x = solve(rho1==rho2, theta)
y = -2*cos(x) + 1*sin(x)
double(y)
```

```
% Convert values of rho and theta to Polar coordinates:
rho = sqrt(x^2+y^2)
% Numerical value of rho
double(rho)
theta = atan(y/x)+pi
% Numerical value of theta
double(theta)
```

2 Task: Robust line fitting using RANSAC.

Here is my solution to task 3. Robust line fitting using RANSAC. Idea of the task was to implement a code that is part of the RANSAC loop. The goal of the code was to determine best model that has biggest number of inlier points and draw a line for it.

Basic idea of the RANSAC algorithm is as follows:
1. Randomly choose s samples.
2. Fit the model to the randomly chosen samples
3. Count the number *inlier_count* of data points (inliers) that fit the model within a measure of error e (in our code t is the inlier distance threshold)
4. Repeat steps 1-3, N times
5. Choose the model that has the largest number *inlier_count* of inliers

Following this RANSAC algorithm and instructions given in the exercise code the code was implemented. The implemented line can be seen in the figure 1. The snippet of the implemented code can also be seen below.

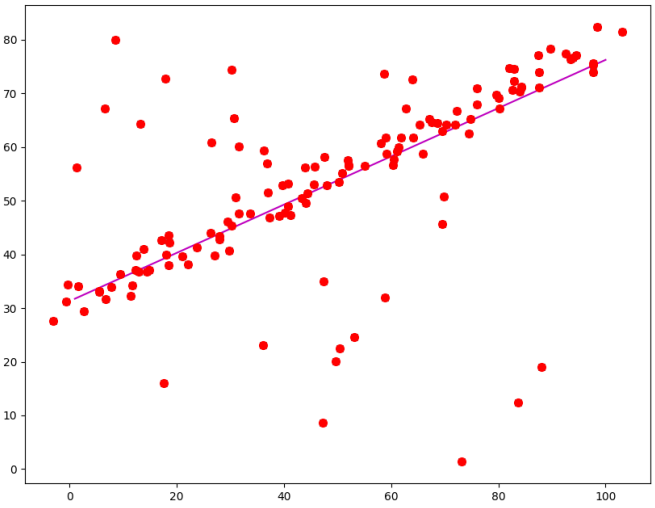

Figure 1: Fitted line to the points using RANSAC

CODE SNIPPET FROM: *RobustLineFitting.py*

```
################ RANSAC loop #########################

# First initialize some variables
N = np.inf
sample_count = 0
max_inliers = 0
best_line = np.zeros((3, 1))

# Data points in homogeneous coordinates
points_h = np.vstack((id1, id2, np.ones((int(m)))))

num_points = len(id1)

while N > sample_count:
    # Pick two random samples
    samples = np.random.choice(np.arange(num_points), 2, replace=False)
    id1 = samples[0]  # sample id 1
    id2 = samples[1]  # sample id 2

    # Determine the line crossing the points with the cross product of the point
    # Also normalize the line by dividing each element by sqrt(a^2+b^2), where a

    ## your code starts here ##
    point1 = points_h[:, id1]
    point2 = points_h[:, id2]
    line = np.cross(point1, point2)
    a, b, c = line
    norm = np.sqrt(a**2 + b**2)
    l = np.array([a/norm, b/norm, c/norm])
    ## your code ends here ##

    # Determine inliers by finding the indices for the line and data point dot
    # products (absolute value) that are less than inlier distance threshold.
    # Hint: point to line distance.

    ## your code starts here ##
    distances = np.abs(np.dot(l, points_h)) / norm
    inliers = np.where(distances < t)[0]
    ## your code ends here ##

    # Store the line in best_line and update max_inliers if the number of
    # inliers is the best so far
    inlier_count = np.size(inliers)
```

```
        if inlier_count > max_inliers:
            best_line = l
            max_inliers = inlier_count

        # Update the estimate of the outlier ratio
        e = 1 - inlier_count / m
        # Update also the estimate for the required number of samples
        N = np.log(1 - p) / np.log(1 - (1 - e) ** s)


        sample_count += 1

    # Least squares fitting to the inliers of the best hypothesis, i.e
    # find the inliers similarly as above but this time for the best line.

    ## your code starts here ##
    id1, id2 = data[0, :], data[1, :]
    x_inliers = id1[inliers]
    y_inliers = id2[inliers]
    ## your code ends here ##

    # Fit a line to the given points (non-homogeneous)
    l = linefitlsq(x_inliers, y_inliers)
    print(l)

    # Plot the resulting line and the inliers
    k = -l[0] / l[1]
    b = -l[2] / l[1]
    plt.plot(np.arange(1, 101), k * np.arange(1, 101) + b, 'm-')
    plt.plot(id1[inliers], id2[inliers], 'ro', markersize=7)
    plt.show()
```

3 Task: Matching Harris corner points.

Here is my solution to task 3. Matching Harris corner points. Idea of the task was to implement a point stitching for two images (finding same points from an image taken from a different position/angle etc.) technique called normalised cross-correlation (NCC).
In the figure 2 we can see the image stitching of SSD measure and in the figure 3 we can see the image stitching of NCC measure. We can clearly see that the NCC has lot less of outliers and false positive lines running through the photos. NCC however also has some outliers, but doesn't have nearly as many as the SSD. (b) The NCC performs a lot better since its more precise (less sensitive to proportional changes of intensity) but computationally more expensive.

Below we can also see the code snippet from *HarrisMatching.py*.

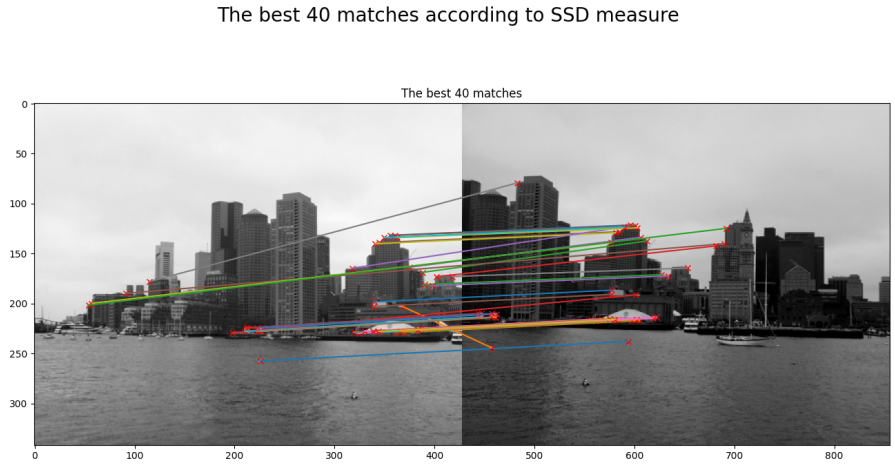

Figure 2: The best 40 matches according to SSD measure

Figure 3: The best 40 matches according to NCC measure

The *distmat* can be calculated using equation:

$$NCC(i,j) = \frac{\sum_{m=0}^{M-1}\sum_{n=0}^{N-1} I(i+m,j+n)*T(m,n)}{(\sqrt{\sum_{m=0}^{M-1}\sum_{n=0}^{N-1} I(i+m,j+n)^2})*((\sqrt{\sum_{m=0}^{M-1}\sum_{n=0}^{N-1} T(m,n)^2})}$$

In the code $I(i+m, j+n)$ can be interpreted as $patches1[:,:,i]$ and $T(m,n)$ as $patches2[:,:,i]$. Saving I and T as the corresponding patches and pointing out that the denominator has the norm $(\sqrt{\sum_{i,j} a_{ij}^2})$ and a function called *linalg.norm* can be used to simplify the equation and the algorithm can be presented as:

$$distmat[i1, i2] = np.sum((I * T)/(np.linalg.norm(I) * np.linalg.norm(T)))$$

Afterwhich implementing the NCC was pretty straight forward and lot similar to implementation of SSD until sorting mutual nearest neighbors based on the SSD where the lists need to be reversed so matches are sorted in descending order when the best matches are found first when iterating the neighbors.

CODE SNIPPET FROM: *HarrisMatching.py*

```
######################### NCC MEASURE ########################
# Now, your task is to do matching in similar manner but using normalised
# cross-correlation (NCC) instead of SSD. You should also report the
# number of correct correspondences for NCC as shown above for SSD.
#
# HINT: Compared to the previous SDD-based implementation, all you need
# to do is to modify the lines performing the 'distmat' calculation
# from SSD to NCC.
# Thereafter, you can proceed as above but notice the following details:
# You need to determine the mutually nearest neighbors by
# finding pairs for which NCC is maximized (i.e. not minimized like SSD).
# Also, you need to sort the matches in descending order in terms of NCC
# in order to find the best matches (i.e. not ascending order as with SSD).

##-your-code-starts-here-##

distmat = np.zeros((npts1, npts2))
for i1 in range(npts1):
    for i2 in range(npts2):
        # Similarity measure for template matching
        # SSD => NCC can be done by:
        # I(i+m, j+n) = patches1[:,:,i1] = I
        # T(m,n) = patches2[:,:,i2] = T
        #          I*T                          I*T
        # ------------------------   =   ------------------
        # (sqrt(I^2))*(sqrt(T^2))          norm(I)*norm(T)
        I = patches1[:,:,i1]
        T = patches2[:,:,i2]
        distmat[i1, i2] = np.sum((I*T)/(np.linalg.norm(I)*np.linalg.norm(T)))

# Compute pairs of patches that are mutually nearest neighbors
# according to the NCC measure
nn1 = np.amax(distmat, axis=1)
ids1 = np.argmax(distmat, axis=1)
nn2 = np.amax(distmat, axis=0)
ids2 = np.argmax(distmat, axis=0)

pairs = []
for k in range(npts1):
    if k == ids2[ids1[k]]:
        pairs.append(np.array([k, ids1[k], nn1[k]]))
pairs = np.array(pairs)
```

```
# Sort the mutually nearest neighbors based on the NCC
# Sort the list in reverse order => matches are sorted in
# descending order so that the best matches are found first
sorted_ncc = np.sort(pairs[:,2], axis=0)[::-1]
id_ncc = np.argsort(pairs[:,2], axis=0)[::-1]

##-your-code-ends-here-##


# Next we visualize the 40 best matches which are mutual nearest neighbors
# and have the smallest SSD values
Nvis = 40
montage = np.concatenate((I1, I2), axis=1)

plt.figure(figsize=(16, 8))
plt.suptitle("The best 40 matches according to NCC measure", fontsize=20)
plt.imshow(montage, cmap='gray')
plt.title('The best 40 matches')
for k in range(np.minimum(len(id_ncc), Nvis)):
    l = id_ncc[k]
    plt.plot(x1[int(pairs[l, 0])], y1[int(pairs[l, 0])], 'rx')
    plt.plot(x2[int(pairs[l, 1])] + I1.shape[1], y2[int(pairs[l, 1])], 'rx')

    plt.plot([x1[int(pairs[l, 0])], x2[int(pairs[l, 1])]+I1.shape[1]],
         [y1[int(pairs[l, 0])], y2[int(pairs[l, 1])]])
plt.show()
```