EX round 4

February 6, 2022

[ ]:
```
Daniel Kusnetsoff
#Task 2
```

[ ]:
```python
from linefitlsq import linefitlsq
import numpy as np
import matplotlib.pyplot as plt

# Load and plot points
data = np.load('points.npy')
x, y = data[0, :], data[1, :]
plt.figure(1, (10, 10))
plt.plot(x, y, 'kx')
plt.axis('scaled')

# RANSAC parameters
# m is the number of data points
m = np.size(x) * 1.0
# s is the size of the random sample
s = 2
# t is the inlier distance threshold
t = np.sqrt(3.84) * 2
# e is the expected outlier ratio
e = 0.8
# at least one random sample should be free
# from outliers with probability p
p = 0.999
# required number of samples
N_estimated = np.log(1 - p) / np.log(1 - (1 - e) ** s)

############### RANSAC loop ######################

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

```python
# Data points in homogeneous coordinates
points_h = np.vstack((x, y, np.ones((int(m)))))

while N > sample_count:
    # Pick two random samples
    samples = np.random.choice(np.arange(len(x)), 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␣
↪points (in homogeneous coordinates).
    # Also normalize the line by dividing each element by sqrt(a^2+b^2), where␣
↪a and b are the line coefficients

    ##-your-code-starts-here-##
    l = np.cross(points_h[:,id1],points_h[:,id2])

    ##-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.

    ##-your-code-starts-here-##
    inliers =[]

    for i in range(int(m)):
        distance = np.abs(np.dot(l, points_h[:,i]))
        if (distance <= t):
            inliers.append(i)


    ##-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-##
for i in range(int(m)):
    distance = np.abs(np.dot(best_line, points_h[:,i]))
    if (distance <= t):
        inliers.append(i)

x_inliers = x[inliers]
y_inliers = y[inliers]


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

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

# 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(x[inliers], y[inliers], 'ro', markersize=7)
plt.show()
```

[ ]: Task 3

[ ]:
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.ndimage import maximum_filter
from scipy.ndimage.interpolation import map_coordinates
from scipy.ndimage.filters import convolve as conv2
from skimage.io import imread
from utils import gaussian2, maxinterp


# Familiarize yourself with the harris function
def harris(im, sigma=1.0, rel_thresh=0.0001, k=0.04):
    im = im.astype(np.float)  # Make sure im is float

    # Get smoothing and derivative filters
    g, _, _, _, _, _, = gaussian2(sigma)
    _, gx, gy, _, _, _, = gaussian2(np.sqrt(0.5))

    # Partial derivatives
    Ix = conv2(im, -gx, mode='constant')
    Iy = conv2(im, -gy, mode='constant')
```

```python
    # Components of the second moment matrix
    Ix2Sm = conv2(Ix**2, g, mode='constant')
    Iy2Sm = conv2(Iy**2, g, mode='constant')
    IxIySm = conv2(Ix*Iy, g, mode='constant')

    # Determinant and trace for calculating the corner response
    detC = (Ix2Sm*IxIySm)-(Iy2Sm**2)
    traceC = Ix2Sm+IxIySm

    # Corner response function R
    # "Corner": R > 0
    # "Edge": R < 0
    # "Flat": |R| = small
    R = detC-k*traceC**2
    maxCornerValue = np.amax(R)

    # Take only the local maxima of the corner response function
    fp = np.ones((3,3))
    fp[1,1] = 0
    maxImg = maximum_filter(R, footprint=fp, mode='constant')

    # Test if cornerness is larger than neighborhood
    cornerImg = R>maxImg

    # Threshold for low value maxima
    y, x = np.nonzero((R > rel_thresh * maxCornerValue) * cornerImg)

    # Convert to float
    x = x.astype(np.float)
    y = y.astype(np.float)

    # Remove responses from image borders to reduce false corner detections
    r, c = R.shape
    idx = np.nonzero((x<2)+(x>c-3)+(y<2)+(y>r-3))[0]
    x = np.delete(x,idx)
    y = np.delete(y,idx)

    # Parabolic interpolation
    for i in range(len(x)):
        _,dx=maxinterp((R[int(y[i]), int(x[i])-1], R[int(y[i]), int(x[i])], R[int(y[i]), int(x[i])+1]))
        _,dy=maxinterp((R[int(y[i])-1, int(x[i])], R[int(y[i]), int(x[i])], R[int(y[i])+1, int(x[i])]))
        x[i]=x[i]+dx
        y[i]=y[i]+dy
```

```python
    return x, y, cornerImg


# Let's try to do Harris corner extraction and matching using our own
# implementation in a less black-box manner.

# Load images
I1 = imread('Boston1.png')/255.
I2 = imread('Boston2m.png')/255.

# Harris corner extraction, take a look at the source code above
x1, y1, cimg1 = harris(I1)
x2, y2, cimg2 = harris(I2)

# Pre-allocate the memory for the 15*15 image patches extracted
# around each corner point from both images
patch_size = 15
npts1 = x1.shape[0]
npts2 = x2.shape[0]
patches1 = np.zeros((patch_size, patch_size, npts1))
patches2 = np.zeros((patch_size, patch_size, npts2))

# The following part extracts the patches using bilinear interpolation
k = (patch_size-1)/2.
xv, yv = np.meshgrid(np.arange(-k, k+1), np.arange(-k, k+1))
for i in range(npts1):
    patch = map_coordinates(I1, (yv + y1[i], xv + x1[i]))
    patches1[:, :, i] = patch
for i in range(npts2):
    patch = map_coordinates(I2, (yv + y2[i], xv + x2[i]))
    patches2[:, :, i] = patch


############################ SSD MEASURE ######################################
# Compute the sum of squared differences (SSD) of pixels' intensities
# for all pairs of patches extracted from the two images
distmat = np.zeros((npts1, npts2))
for i1 in range(npts1):
    for i2 in range(npts2):
        distmat[i1, i2] = np.sum((patches1[:,:,i1]-patches2[:,:,i2])**2)

# Next, compute pairs of patches that are mutually nearest neighbors
# according to the SSD measure
ss1 = np.amin(distmat, axis=1)
ids1 = np.argmin(distmat, axis=1)
ss2 = np.amin(distmat, axis=0)
ids2 = np.argmin(distmat, axis=0)
```

```
pairs = []
for k in range(npts1):
    if k == ids2[ids1[k]]:
        pairs.append(np.array([k, ids1[k], ss1[k]]))
pairs = np.array(pairs)

# We sort the mutually nearest neighbors based on the SSD
sorted_ssd = np.sort(pairs[:,2], axis=0)
id_ssd = np.argsort(pairs[:,2], axis=0)

# 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 SSD measure", fontsize=20)
plt.imshow(montage, cmap='gray')
plt.title('The best 40 matches')
for k in range(np.minimum(len(id_ssd), Nvis)):
    l = id_ssd[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])]])


############################ 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))

g_a = np.mean(patches1)
```

```
f_a = np.mean(patches2)
for i1 in range(npts1):
    for i2 in range(npts2):
        number = np.sum((patches1[:,:,i1] - g_a) * (patches2[:,:,i2] - f_a))
        den1 = np.sqrt(np.sum((patches1[:,:,i1] - g_a)**2) * np.sum((patches2[:
↪,:,i2] - f_a)**2))
        distmat[i1, i2] = number / den1

ss1 = np.amax(distmat, axis=1)
ids1 = np.argmax(distmat, axis=1)
ss2 = 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], ss1[k]]))
pairs = np.array(pairs)

sorted_ncc = np.sort(pairs[:,2], axis=0)
sorted_ncc = np.flip(sorted_ncc) # Flip to desc. order
id_ncc = np.argsort(pairs[:,2], axis=0)
id_ncc = np.flip(id_ncc) # Flip to desc. order

##-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()


# b.
```

```
# The NCC measure outperforms the ssd as it takes the local average intensity␣
↪into account.
```

[ ]:
```
task 4
```

[ ]:
```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.ndimage import maximum_filter
from scipy.ndimage.interpolation import map_coordinates
from scipy.ndimage.filters import convolve as conv2
from skimage.io import imread
from utils import gaussian2, maxinterp


# Familiarize yourself with the harris function
def harris(im, sigma=1.0, rel_thresh=0.0001, k=0.04):
    im = im.astype(np.float)  # Make sure im is float

    # Get smoothing and derivative filters
    g, _, _, _, _, _, = gaussian2(sigma)
    _, gx, gy, _, _, _, = gaussian2(np.sqrt(0.5))

    # Partial derivatives
    Ix = conv2(im, -gx, mode='constant')
    Iy = conv2(im, -gy, mode='constant')

    # Components of the second moment matrix
    Ix2Sm = conv2(Ix**2, g, mode='constant')
    Iy2Sm = conv2(Iy**2, g, mode='constant')
    IxIySm = conv2(Ix*Iy, g, mode='constant')

    # Determinant and trace for calculating the corner response
    detC = (Ix2Sm*IxIySm)-(Iy2Sm**2)
    traceC = Ix2Sm+IxIySm

    # Corner response function R
    # "Corner": R > 0
    # "Edge": R < 0
    # "Flat": |R| = small
    R = detC-k*traceC**2
    maxCornerValue = np.amax(R)

    # Take only the local maxima of the corner response function
    fp = np.ones((3,3))
    fp[1,1] = 0
    maxImg = maximum_filter(R, footprint=fp, mode='constant')
```

```
    # Test if cornerness is larger than neighborhood
    cornerImg = R>maxImg

    # Threshold for low value maxima
    y, x = np.nonzero((R > rel_thresh * maxCornerValue) * cornerImg)

    # Convert to float
    x = x.astype(np.float)
    y = y.astype(np.float)

    # Remove responses from image borders to reduce false corner detections
    r, c = R.shape
    idx = np.nonzero((x<2)+(x>c-3)+(y<2)+(y>r-3))[0]
    x = np.delete(x,idx)
    y = np.delete(y,idx)

    # Parabolic interpolation
    for i in range(len(x)):
        _,dx=maxinterp((R[int(y[i]), int(x[i])-1], R[int(y[i]), int(x[i])],
 ↪R[int(y[i]), int(x[i])+1]))
        _,dy=maxinterp((R[int(y[i])-1, int(x[i])], R[int(y[i]), int(x[i])],
 ↪R[int(y[i])+1, int(x[i])]))
        x[i]=x[i]+dx
        y[i]=y[i]+dy

    return x, y, cornerImg


# Let's try to do Harris corner extraction and matching using our own
# implementation in a less black-box manner.

# Load images
I1 = imread('Boston1.png')/255.
I2 = imread('Boston2m.png')/255.

# Harris corner extraction, take a look at the source code above
x1, y1, cimg1 = harris(I1)
x2, y2, cimg2 = harris(I2)

# Pre-allocate the memory for the 15*15 image patches extracted
# around each corner point from both images
patch_size = 15
npts1 = x1.shape[0]
npts2 = x2.shape[0]
patches1 = np.zeros((patch_size, patch_size, npts1))
patches2 = np.zeros((patch_size, patch_size, npts2))
```

```python
# The following part extracts the patches using bilinear interpolation
k = (patch_size-1)/2.
xv, yv = np.meshgrid(np.arange(-k, k+1), np.arange(-k, k+1))
for i in range(npts1):
    patch = map_coordinates(I1, (yv + y1[i], xv + x1[i]))
    patches1[:, :, i] = patch
for i in range(npts2):
    patch = map_coordinates(I2, (yv + y2[i], xv + x2[i]))
    patches2[:, :, i] = patch


############################ SSD MEASURE ######################################
# Compute the sum of squared differences (SSD) of pixels' intensities
# for all pairs of patches extracted from the two images
distmat = np.zeros((npts1, npts2))
for i1 in range(npts1):
    for i2 in range(npts2):
        distmat[i1, i2] = np.sum((patches1[:,:,i1]-patches2[:,:,i2])**2)

# Next, compute pairs of patches that are mutually nearest neighbors
# according to the SSD measure
ss1 = np.amin(distmat, axis=1)
ids1 = np.argmin(distmat, axis=1)
ss2 = np.amin(distmat, axis=0)
ids2 = np.argmin(distmat, axis=0)

pairs = []
for k in range(npts1):
    if k == ids2[ids1[k]]:
        pairs.append(np.array([k, ids1[k], ss1[k]]))
pairs = np.array(pairs)

# We sort the mutually nearest neighbors based on the SSD
sorted_ssd = np.sort(pairs[:,2], axis=0)
id_ssd = np.argsort(pairs[:,2], axis=0)

# 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 SSD measure", fontsize=20)
plt.imshow(montage, cmap='gray')
plt.title('The best 40 matches')
for k in range(np.minimum(len(id_ssd), Nvis)):
    l = id_ssd[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])]])


############################ 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))

g_a = np.mean(patches1)
f_a = np.mean(patches2)
for i1 in range(npts1):
    for i2 in range(npts2):
        number = np.sum((patches1[:,:,i1] - g_a) * (patches2[:,:,i2] - f_a))
        den1 = np.sqrt(np.sum((patches1[:,:,i1] - g_a)**2) * np.sum((patches2[:
↪,:,i2] - f_a)**2))
        distmat[i1, i2] = number / den1

ss1 = np.amax(distmat, axis=1)
ids1 = np.argmax(distmat, axis=1)
ss2 = 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], ss1[k]]))
pairs = np.array(pairs)

sorted_ncc = np.sort(pairs[:,2], axis=0)
sorted_ncc = np.flip(sorted_ncc) # Flip to desc. order
id_ncc = np.argsort(pairs[:,2], axis=0)
```

```
id_ncc = np.flip(id_ncc) # Flip to desc. order

##-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()


# b.
# The NCC measure outperforms the ssd as it takes the local average intensity␣
 ↪into account.
```