Fundamentals of Image Processing

Alpen-Adria-Universität Klagenfurt

Philipp Freislich, Muamer Hrncic, Daniel Siebenhofer

February 2022

Contents

1 Question 1

1.1 Task

For question 1 the task was to align the reference image with the tampered image and then to subtract the tampered from the original image. The result should then be binarized, where the pixels that differ should be white and the pixels that are the same as in the reference painting should be black.

1.2 Solution

The first step was to align the reference and the tampered image. For this purpose we decided to use the **ORB** image matching technique. ORB is a fusion of the **FAST** key point detector and **BRIEF** descriptor with some modifications. Initially to determine the key points, it uses FAST. Then a Harris corner measure is applied to find top N points. FAST does not compute the orientation and is rotation variant. It computes the intensity weighted centroid of the patch with located corner at center. The direction of the vector from this corner point to centroid gives the orientation. Moments are computed to improve the rotation invariance. The descriptor BRIEF performs poorly, if there is an in-plane rotation. In ORB, a rotation matrix is computed using the orientation of patch and then the BRIEF descriptors are steered according to the orientation. We decided to go with ORB instead of **SIFT**, because it is faster than SIFT and shows similar performances, if the angle of rotation is proportional to 90 degrees. The 10 best matches were then used to determine the translation.

```
    # Initiate ORB detector
    orb = cv2.ORB_create()
    # find the keypoints and descriptors with ORB
    kp1, des1 = orb.detectAndCompute(original_img,None)
    kp2, des2 = orb.detectAndCompute(tampered_img,None)
    # create BFMatcher object
    bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)
    # Match descriptors.
    matches = bf.match(des1,des2)
    # Sort them in the order of their distance.
    matches = sorted(matches, key = lambda x:x.distance)
    # only take 10 best matches
    matches= matches[:10]
    # Draw first 10 matches.
    img3 = cv2.drawMatches(original_img,kp1,tampered_img,kp2,
    matches,None,flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)
    # show matches and lines between them
    plt.figure("Features_" + str(counter),figsize=(14,8)), plt.
    imshow(img3, "gray")
```

Listing 1: ORB feature matching

Using the median of those 10 translation values provides us with the best result. After aligning the 2 images we subtracted the shifted tampered image from the original and applied thresholding to reduce noise.

```
    # store coordinates of matches
    pts1 = np.float32([kp1[m.queryIdx].pt for m in matches])
    pts2 = np.float32([kp2[m.trainIdx].pt for m in matches])
    # get the difference in position of the matches of the
    keypoints
    diff = pts1 -pts2
    # get the median values to remove noise
    (delta_y,delta_x) = (np.median(diff[:,0]),np.median(diff[:,1]))
    # translation matrix that takes overlaps the images for
    comparison
    M = np.float32([[1,0,-delta_x],[0,1,-delta_y]])
    # perform the translation
    shifted_2=cv2.warpAffine(tampered_img, M, (tampered_img.shape
    [1],tampered_img.shape[0]))
    # substract the alligned tampered img from the original
    output = np.abs(np.float32(original_img) -np.float32(shifted_2)
    )
    # thresholding to reduce noise
    output[output>2] = 255
    output[output<=2] = 0
```

Listing 2: Alignment and tampering detection

The matched features of painting1 can be seen in figure 1 and the difference of the original and the tampered painting1 are shown in figure 2.

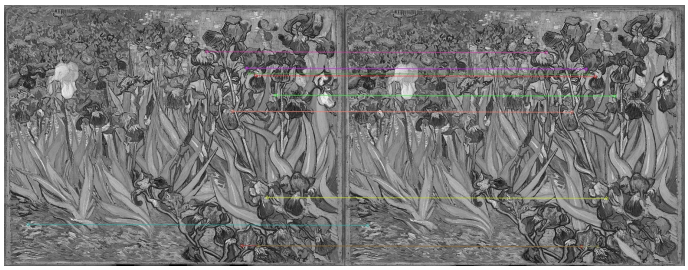


Figure 1: Matched features of painting1

Figure 2: Difference between original/tampered painting1

2 Question 2

2.1 Task

For question 2 the task was to apply contrast enhancing techniques to images captured in the night and to generate the histograms before and after using those techniques.

2.2 Solution

To show the histograms for section a of the task we used the **calcHist()** function implemented in openCV on the grayscale images.

```
# read the images as grayscale
img_1  = cv2.imread("hw1_dark_road_1.jpg",0)
img_2  = cv2.imread("hw1_dark_road_2.jpg",0)
img_3  = cv2.imread("hw1_dark_road_3.jpg",0)
#calc the histograms
hist_1 = cv2.calcHist([img_1],[0],None,[256],[0,256])
hist_2 = cv2.calcHist([img_2],[0],None,[256],[0,256])
hist_3 = cv2.calcHist([img_3],[0],None,[256],[0,256])

# show imgs+histograms
fig=plt.figure("Histograms")
plt.subplot(3,2,1), plt.imshow(img_1, 'gray')
plt.subplot(3,2,2), plt.plot(hist_1)
plt.subplot(3,2,3), plt.imshow(img_2, 'gray')
plt.subplot(3,2,4), plt.plot(hist_2)
plt.subplot(3,2,5), plt.imshow(img_3, 'gray')
plt.subplot(3,2,6), plt.plot(hist_3)
```

Listing 3: Histograms of grayscale images

The contrast of the provided images is very bad, as can be seen in figure 3 down below.Image 1 has very low contrast and this can be seen by most of the intensity values ranging from 0 to around 40.The same applies for image 2 and 3 although for image 3 there are less pixels with exacly the same intensity values and the range of the values is not as narrow as for the other two images.

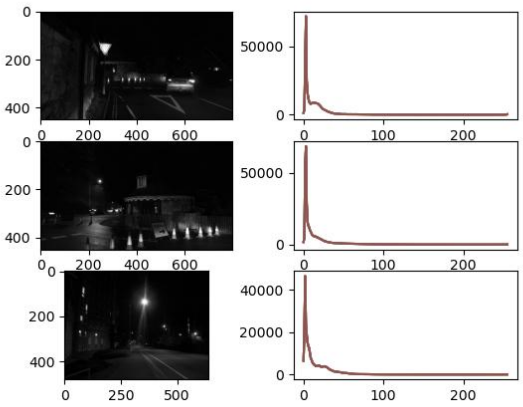


Figure 3: Histogram of input images

To apply the global histogram equalization to the images, the **equalizeHist()** function implemented in openCV was used.

```
# use openCV fct to globally eq. histograms
equ_1 = cv2.equalizeHist(img_1)
equ_2 = cv2.equalizeHist(img_2)
equ_3 = cv2.equalizeHist(img_3)
# calc histograms
hist_1 = cv2.calcHist([equ_1],[0],None,[256],[0,256])
hist_2 = cv2.calcHist([equ_2],[0],None,[256],[0,256])
hist_3 = cv2.calcHist([equ_3],[0],None,[256],[0,256])
```

Listing 4: Histogram equalization

For all three images this procedure produced quite nice results for most parts of the images with way more of the image beeing visible after the gloabl histogram equalization, except for the sky and the light sources. For those segments of the images we produced a lot of noise which is not desirable.

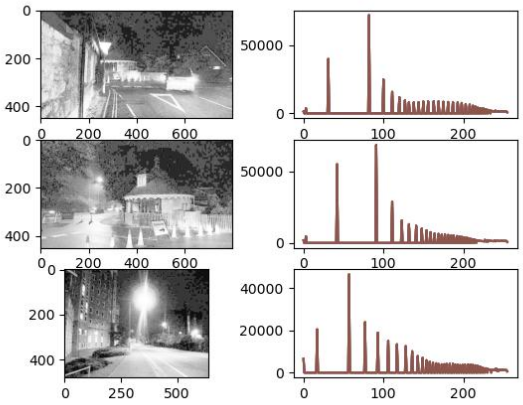


Figure 4: Global histogram equalization

To apply locally adaptive histogram equalization to the image, we used the **createCLAHE()** function of openCV and then applied the generated CLAHE object to our grayscaled input image.

```
1
2  clahe1 = cv2.createCLAHE(clipLimit=7.1, tileGridSize=(12,12))
3  # apply to all images
4  bl1 = clahe1.apply(img_1)
5  bl2 = clahe1.apply(img_2)
6  bl3 = clahe1.apply(img_3)
7  # calculate the histograms
8  hist_1 = cv2.calcHist([bl1],[0],None,[256],[0,256])
9  hist_2 = cv2.calcHist([bl2],[0],None,[256],[0,256])
10 hist_3 = cv2.calcHist([bl3],[0],None,[256],[0,256])
```

Listing 5: Locally adaptive histogram equalization

After trying many different settings for the CLAHE object we found that for the best results a clipLimit between 7 and 8 would be optimal. A value closer to 7 results in less noise, but a darker image in general. A value closer to 8 produces more noise, but increases the overall brightness of the image. We thought it is more important to keep the noise to a minimum, while inreasing the quality of the image, so we decided to go with a value of 7.1 for the clipLimit. A gridsize of 144 rectangular tiles worked pretty well, so we diced to not go with the default of 64 tiles, since the 12x12 version produced results with more details. For comparison purposes the resulting images after applying the 2 different methods for histogram equalization and the original images are shown in figure 5.

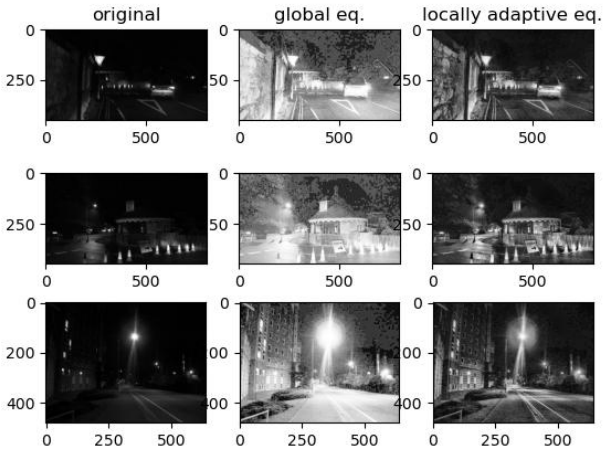


Figure 5: Comparison

The quality of the image produced with the adaptive histogram equalization is way better (subjectively) compared to the resuls achieved by using global histogram equalization. The noise is greatly reduced while still mainting a way better contrast than in the original image.

3 Question 3

3.1 Task

For question 3 the task was to use 2 different types of median filtering to reduce the salt-and-pepper noise in vintage photographs.

3.2 Solution

To apply median filters with a kernel size of 3x3 and 5x5 to the given images, the **medianBlur()** function provided by openCV was used.

```
1
2  # Load images
3  img_1  = cv2.imread("hw3_building.jpg",0)
4  img_2  = cv2.imread("hw3_train.jpg",0)
5
6  # Apply median filter with 3x3 and 5x5 kernel to img1
7  img_1_m3 = cv2.medianBlur(img_1,3)
8  img_1_m5 = cv2.medianBlur(img_1,5)
9  # Apply median filter with 3x3 and 5x5 kernel to img2
10 img_2_m3 = cv2.medianBlur(img_2,3)
11 img_2_m5 = cv2.medianBlur(img_2,5)
12
13 # show the images
14 plt.figure("Median Filter")
15 plt.subplot(2,3,1), plt.imshow(img_1, 'gray'), plt.title("Original"
       ), plt.xticks([]), plt.yticks([])
16 plt.subplot(2,3,2), plt.imshow(img_1_m3,'gray'), plt.title("3x3
       Median Filter"),plt.xticks([]), plt.yticks([])
17 plt.subplot(2,3,3), plt.imshow(img_1_m5, 'gray'),plt.title("5x5
       Median Filter"),plt.xticks([]), plt.yticks([])
18 plt.subplot(2,3,4), plt.imshow(img_2,'gray'),plt.title("Original"),
       plt.xticks([]), plt.yticks([])
19 plt.subplot(2,3,5), plt.imshow(img_2_m3, 'gray'),plt.title("3x3
       Median Filter"),plt.xticks([]), plt.yticks([])
20 plt.subplot(2,3,6), plt.imshow(img_2_m5,'gray'),plt.title("5x5
       Median Filter"),plt.xticks([]), plt.yticks([])
```

Listing 6: Median filtering (3x3 & 5x5)

For the 3x3 median filter the noise is still in the image although very reduced, while only slightly reducing the quality of the edges and features the image contains. For the 5x5 median filter however the noise is practically completely removed, but the quality of the edges and features of the images is also visibly reduced. This kind of tradeoff is common and it depends on what the resulting image is used for to decide which kind of filtering would be more adequate. The resulting image of using the median filter with a kernelsize of 3x3 and 5x5 for one of the two images provided is shown in figure 6 and figure 7.

Figure 6: 3x3 median filtering

Figure 7: 5x5 median filtering

To apply weighted median filtering to the provided images, we decided to write a function that convolves the given kernel with the selected image.

```
# kernel for weighted median filtering
kernel = np.array
    ([[0,1,1,1,0],[1,2,2,2,1],[1,2,4,2,1],[1,2,2,2,1],[0,1,1,1,0]])

def calcCustomMedian(img,kernel):
    # get amount of padding needed, assuming kernel = mxn matrix m=
    n, m odd
    offset = np.size(kernel,0)//2
    # add padding to the image for the filtering
    img_padded = cv2.copyMakeBorder(img, offset, offset, offset,
    offset,cv2.BORDER_REPLICATE)
    # get height and width
    height,width = img.shape
    # copy img for output
    new_image = img.copy()

    # y between the two paddings
    for y in range(offset,height+offset):
        # x between the padding and from right to left
        for x in range(width+offset-1,offset-1,-1):
            # array for values we calculate the median from
```

```
19              values = []
20              # go through every value of the kernel
21              for a in range(0,5):
22                  for b in range(0,5):
23                      # current weight
24                      w = kernel[a,b]
25                      # current value
26                      c_value = img_padded[y+a-offset,x+b-offset]
27                      # add w amount of c_values to the list
28                      for c in range(0,w):
29                          values.append(c_value)
30                          # get the median
31                          new_image[y-offset,x-offset] =statistics.
    median(values)
32      return new_image
```

Listing 7: Median filtering (3x3 & 5x5)

The first step of applying the weighted median filter is to add padding to the image, because the filter with a kernelsize of 5x5 needs a border of 2 pixels in every direction to function properly. We decided to calculate the amount of pixels needed in every direction needed for more portability of the function. Using the **copyMakeBorder()** function provided by openCV with the option **BORDER_REPLICATE** we got a border with the same intensity value as the pixel right next to it. A simple loop through the image starting at the right top corner and moving left implements the convolution of the image with the kernel. For every pixel we calculate its new value by appending it and its neighbors n times to a temporary list, where n depends on the kernel used. We then take the median of that list and this value is the new value for that exact pixel. The application of given weighted median filter to the provided images can be seen in figure 8.

Figure 8: Weighted median filtering

4 Question 4

4.1 Task

For question 4 the task was to use an iterative grayscale morphological image processing algorithm to enhance the sharpness of structures in a blurry image show in 9. The enhancement algorithm ist shown in 10.

Figure 9: Blurry image

```
Im := Input Image
For Iteration = 1:NumIterations
      Im_d = dilate(Im, W)     % Note that this is grayscale dilation
      Im_e = erode(Im, W)      % or erosion with structuring element W
      Im_h = 0.5(Im_d + Im_e)

      % Perform the following test for each pixel
      If Im > Im_h
            Im := Im_d
      Else
            Im := Im_e
      End
End
```

Figure 10: Algorithm pseudo code

4.2 Solution

First we load the blurry image and get the necessary kernels for morphological operations with the method **getKernel()** from the file **enhanceImage.py**. Since the processing procedure is at the start we use two different kernel sizes and the three default kernels with rectangular, elliptic and cross shape. We also used user defined kernels, like tilted cross, or combinations of the three default shapes, but tests showed that it is sufficient to use the default kernels from the **open-cv** library.

```
#import necessary libraries
import cv2 as cv
import matplotlib.pyplot as plt
from enhanceImage import getKernel, enhanceSharpness

#read image
```

```
img = cv.imread("hw3_road_sign_school_blurry.jpg", cv.
    IMREAD_GRAYSCALE)

#show original image
plt.imshow(img, cmap='gray')

#kernel sizes
s1 = 3
s2 = 5

#rectangular kernel, 3x3
ker_rect3 = getKernel('rect', ksize_x=s1, ksize_y=s1)
#rectangular kernel, 5x5
ker_rect5 = getKernel('rect', ksize_x=s2, ksize_y=s2)
#elliptic kernel, 3x3
ker_ell3 = getKernel('ellipse', ksize_x=s1, ksize_y=s1)
...
```

Listing 8: Load image, get kernels

After collecting the necessary data the enhancement procedure **enhanceSharpness()** ,also from **enhanceImage.py**, is applied for the different kernel shapes and kernel sizes.

```
#process the original image and get the enhanced image
#enhanced image, rectangular kernel, 3x3
enhanced_1 = enhanceSharpness(img, ker_rect3, nrIter=10)
#enhanced image, rectangular kernel, 5x5
enhanced_2 = enhanceSharpness(img, ker_rect5, nrIter=10)
#enhanced image, elliptical kernel, 3x3
enhanced_3 = enhanceSharpness(img, ker_ell3, nrIter=10)
#enhanced image, elliptical kernel, 5x5
enhanced_4 = enhanceSharpness(img, ker_ell5, nrIter=10)
#enhanced image, cross kernel, 3x3
enhanced_5 = enhanceSharpness(img, ker_cross3, nrIter=10)
#enhanced image, cross kernel, 5x5
enhanced_6 = enhanceSharpness(img, ker_cross5, nrIter=10)
...
#Conclusion:
#Depending on the used kernel and its size but also
#depending on the forms in the original picture (circles,
#horizontals, diagonals) the algorithm sharpens certain
#parts of the original picture and other parts get
#getting thinned out (compare rect. kernel or cross
#kernel 5x5, holes on heads).
```

Listing 9: Enhance images

The enhancement procedure is implemented as described and two additional parameter were added to get control of the number of morphological operation, which makes a difference as will be shown. The default number of iterations is ten and the default number of dilation and erosion steps is one. The enhancement procedure looks as follows:

```
def enhanceSharpness(img, kernel, nrIter=10, dilIter=1, erodIter=1)
    :
    ...
    #main algorithm
    for i in range(nrIter):
        #dilate the image once with the given kernel and save it
        Im_d = cv.dilate(img_enh, kernel, iterations=dilIter)
        #erode the image once with the given kernel and save it
```

```
        Im_e = cv.erode(img_enh, kernel, iterations=erodIter)
        #save the "mean" of the dilated and the eroded image
        Im_h = 0.5 * (Im_d + Im_e)
        for j in range(rows):
            for k in range(cols):
                #if the pixel value of the original is greater
                #than the mean value
                if(img_enh[j, k] > Im_h[j, k]):
                    #save pixel value of the dilated image
                    img_enh[j, k] = Im_d[j, k]
                else:
                    #save the pixel value of the eroded image
                    img_enh[j, k] = Im_e[j, k]
    return img_enh
```

Listing 10: Enhancement procedure

The results of applying the enhancement procedure with different shapes and sizes of 3×3 are shown in 11.

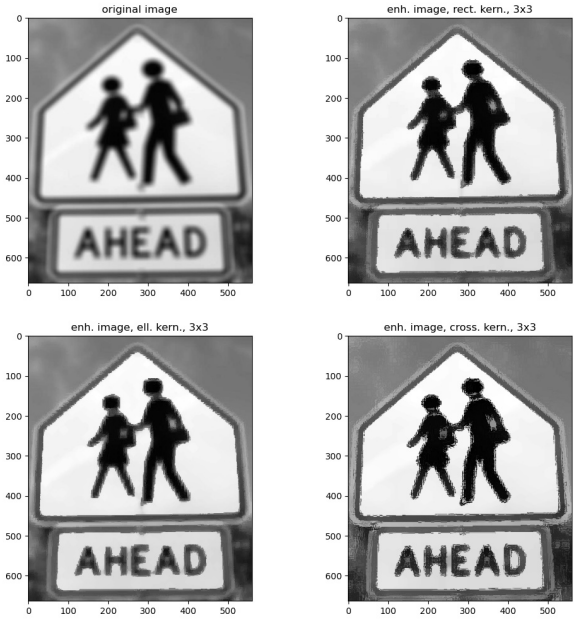


Figure 11: Enhanced images, different kernels, kernel size 3×3

After applying kernels of size 3×3 the resulting images are a bit sharper but there is still some noise in the enhanced image. Different kernels sharpen different areas with different precision as expected. The elliptic kernel performs better on shapes with curvature than on edges and the cross kernel makes a better job on long edges than the rectangular kernel, but fails for other shapes. Since there is still some noise we considered also kernels of size 5×5 and get the results shown in 12. In this case the elliptic kernel does the best overall

performance and one can conclude that the rectangular shape is not the best choice for this image and the chosen kernel size.

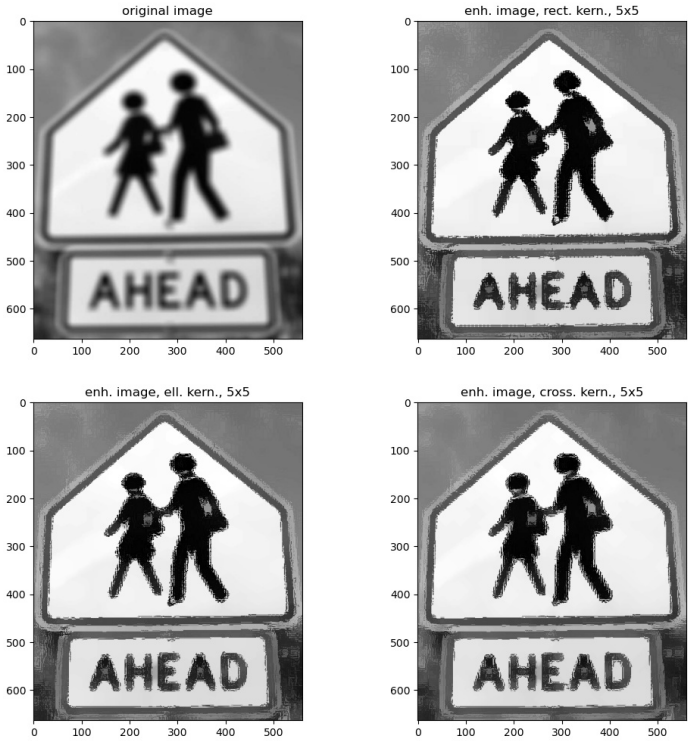


Figure 12: Enhanced images, different kernels, kernel size 5×5

One negative aspect of the process is that enhancement takes some time and the results could be better. So we extended the procedure as mentioned before by controlling the number of morphological operations. Now we use kernels of size 3×3 as in the beginning but we choose different values for the number of overall iterations and for the number of dilation and erosion operations.

```
...
############################################################
#Another try with different number of dilation and
#eroding iterations, but with fewer overall iterations!

#rectangular kernel
ker_rect = getKernel('rect', ksize_x=s1, ksize_y=s1)

#elliptic kernel
ker_ell = getKernel('ellipse', ksize_x=s1, ksize_y=s1)

```

```
#cross kernel
ker_cross = getKernel('cross', ksize_x=s1, ksize_y=s1)

#Process the original image and get the enhanced image
#enhanced image, rectangular kernel
enhanced_rect = enhanceSharpness(img, ker_rect, nrIter=3, dilIter
    =3, erodIter=2)
#enhanced image, rectangular kernel
enhanced_ell = enhanceSharpness(img, ker_ell, nrIter=3, dilIter=3,
    erodIter=2)
#enhanced image, elliptical kernel
enhanced_cross = enhanceSharpness(img, ker_cross, nrIter=3, dilIter
    =3, erodIter=2)
...
#Conclusion:
#The last result shows, the algorithm takes less time but performs
#better in sharpening the picture.
```

Listing 11: Enhance image, different parameters

The first positive result one can observe is that the overall run time is a lot shorter than with default parameters, which is interesting, since the number of morphological operations is increased. The next fact can be observed in the enhancement results shown in 13. Now the smaller kernels have an overall good performance and all shapes are sharpened equally well. The noise in the important areas is also reduced. The only side effect is that some shapes, like letters, are thicker than before, which results from the choice of parameters. Our opinion is that this side effect is nothing negative since the shapes can be recognized a lot better than in the original image or the enhanced images with default parameters, and as mentioned the run time is a lot shorter.

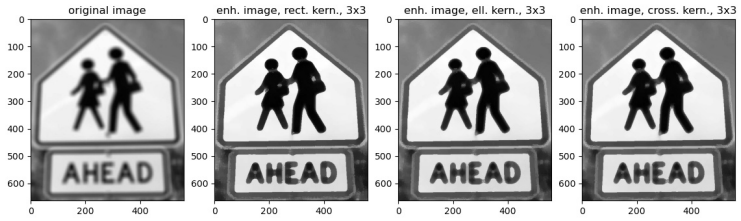


Figure 13: Enhanced images, different parameters

In the end we also tried another approach which also leads to interesting results and can be combined with the enhancement algorithm. This approach is simple but also delivers good results although nothing complex happens. The number of overall iterations is even smaller than in the second approach.

```
...
#Another approach
#Lets take a threshold image of the original image
_, threshold = cv.threshold(img, 69, 255, 0)

#Take the bitwise AND of the original image and the threshold
    picture
img_and_thres = cv.bitwise_and(img, threshold)
img_and_thres =cv.medianBlur(img_and_thres, 3)
...
#Process the original threshold image and get the enhanced image
#enhanced image, rectangular kernel
enhanced_rect = enhanceSharpness(img_and_thres, ker_rect, nrIter=3,
     dilIter=1, erodIter=1)
#enhanced image, rectangular kernel
enhanced_ell = enhanceSharpness(img_and_thres, ker_ell, nrIter=3,
    dilIter=1, erodIter=1)
#enhanced image, elliptical kernel
enhanced_cross = enhanceSharpness(img_and_thres, ker_cross, nrIter
    =3, dilIter=1, erodIter=1)
...
```

Listing 12: Threshold approach

We apply thresholding to the original image and combine the threshold image with the original with an logical AND-operation, which can be seen in 14.

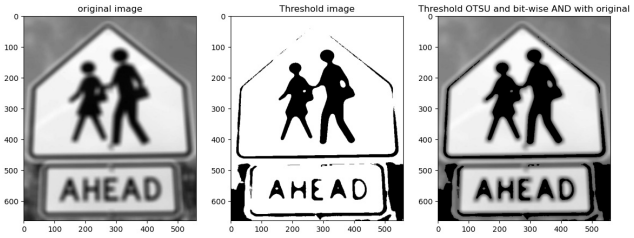


Figure 14: Threshold approach

After applying the enhancement procedure we get really nice results as can be seen in 15. There are still some thin shadows around the shapes but the shapes are sharpened enough for good recognition. The letters are also not that thick, which is also better than before. This concludes that starting with sufficiently preprocessed images delivers also good results but with less computational effort.

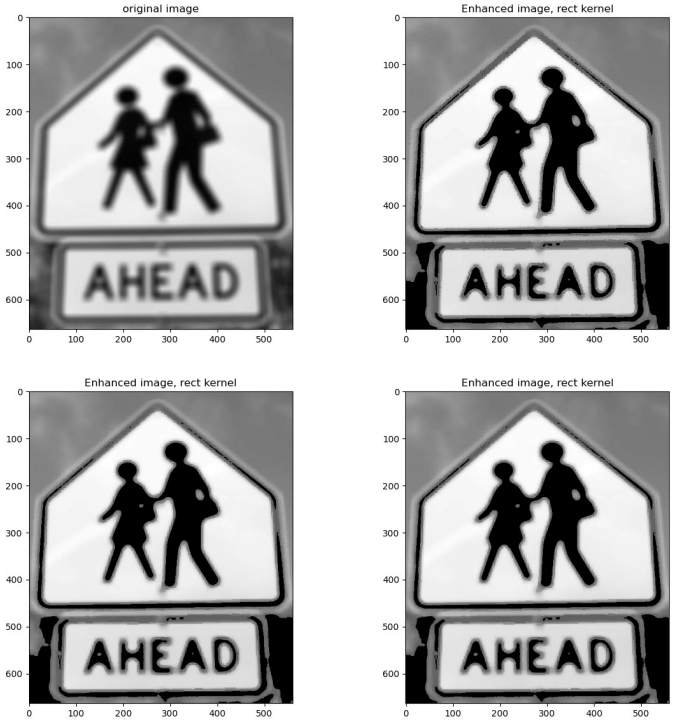
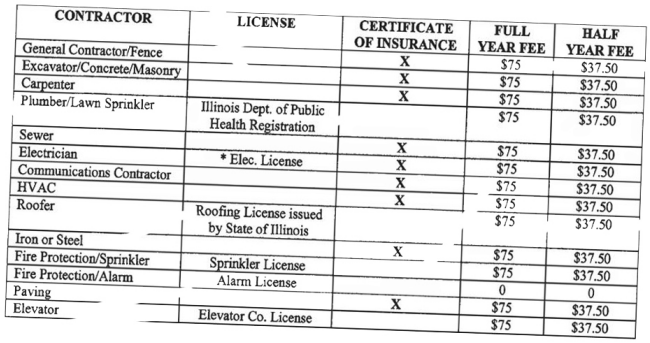


Figure 15: Threshold approach

5 Question 5

5.1 Task

For question 5 the task was to implement a procedure to automatically rotates the given image 16 to an upward rotation and to repair the broken lines in the table show in.



CONTRACTOR	LICENSE	CERTIFICATE OF INSURANCE	FULL YEAR FEE	HALF YEAR FEE
General Contractor/Fence		X	$75	$37.50
Excavator/Concrete/Masonry		X	$75	$37.50
Carpenter		X	$75	$37.50
Plumber/Lawn Sprinkler	Illinois Dept. of Public Health Registration		$75	$37.50
Sewer		X	$75	$37.50
Electrician	* Elec. License	X	$75	$37.50
Communications Contractor		X	$75	$37.50
HVAC		X	$75	$37.50
Roofer	Roofing License issued by State of Illinois		$75	$37.50
Iron or Steel		X	$75	$37.50
Fire Protection/Sprinkler	Sprinkler License		$75	$37.50
Fire Protection/Alarm	Alarm License		0	0
Paving		X	$75	$37.50
Elevator	Elevator Co. License		$75	$37.50

Figure 16: Distorted fax

5.2 Solution

We load the image and convert it to gray scale. The first step is to perform canny edge detection.The result of the edge detection is shown in 17.

```
...
# read original image
img = cv.imread("hw5_insurance_form.jpg")

# save image dimensions
(h, w) = img.shape[:2]

# convert to grayscale image for edge detection
gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)

# save grayscale image
cv.imwrite('out/1_grayscale.jpg', gray)
############################################################

# Perform canny edge detection to indicate existing edges
thr1 = 50
thr2 = 200

edges = cv.Canny(gray, thr1, thr2)

# save image of found edges
cv.imwrite('out/2_canny_first.jpg', edges)
```

Listing 13: Read image and detect edges

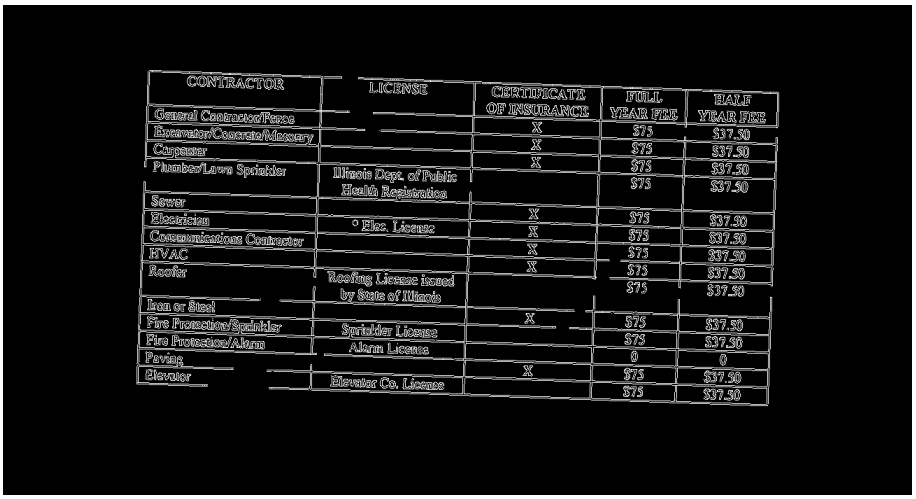


Figure 17: Canny edge detection

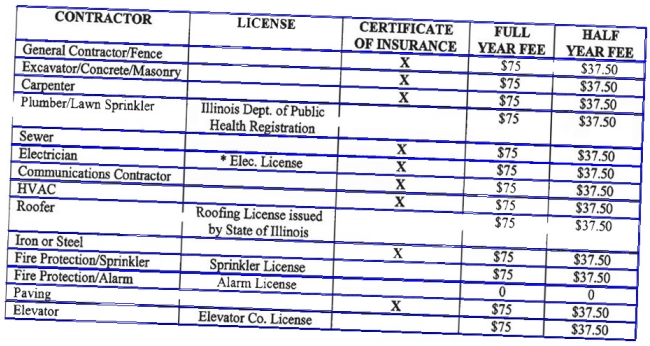
Afterwards Hough transformation is applied to the canny edge image in order to detect the vertical and/or horizontal lines which are highlighted in blue color in image 18.

```
############################################################
...
# find lines with hough transformation in the canny
#edge detection image
lines = getEdges(edges, rho=0.5, theta=1 * np.pi / 180.0,
                 threshold=180, minLineLength=20, maxGap=100)
imgWithLines = img.copy()

for line in lines:
    # extract end points of lines
    x1, y1, x2, y2 = line[0]
    # and draw them to original image
    cv.line(imgWithLines, (x1, y1), (x2, y2), (255, 0, 0), 1)

...
############################################################
...
```

Listing 14: Hough transformation for line detection



CONTRACTOR	LICENSE	CERTIFICATE OF INSURANCE	FULL YEAR FEE	HALF YEAR FEE
General Contractor/Fence		X	$75	$37.50
Excavator/Concrete/Masonry		X	$75	$37.50
Carpenter		X	$75	$37.50
Plumber/Lawn Sprinkler	Illinois Dept. of Public Health Registration		$75	$37.50
Sewer		X	$75	$37.50
Electrician	* Elec. License	X	$75	$37.50
Communications Contractor		X	$75	$37.50
HVAC		X	$75	$37.50
Roofer	Roofing License issued by State of Illinois		$75	$37.50
Iron or Steel		X	$75	$37.50
Fire Protection/Sprinkler	Sprinkler License		$75	$37.50
Fire Protection/Alarm	Alarm License		0	0
Paving		X	$75	$37.50
Elevator	Elevator Co. License		$75	$37.50

Figure 18: Hough lines

At this step we have the detected lines and we use them to find the rotation angle of the table which is bounded by such vertical and horizontal lines.

```
############################################################
...
# determine rotation of image with help of found lines
# save coordinates of first found line
(x1, y1, x2, y2) = lines[0, 0]

# get rotation angle in rad and convert to deg
rho = getRotation(x1, y1, x2, y2) * 180 / np.pi

# get rotation matrix with rotation angle rho
rotMatrix = cv.getRotationMatrix2D((int(x1), int(y1)), rho, 1.0)

# rotate the picture and assign generated blank space white color
rotated = cv.warpAffine(img, rotMatrix, (int(w), int(h)),
                        borderValue=(255, 255, 255))
...
############################################################
...
```

Listing 15: Get rotation

The rotation angle is determined with the help of the method **getRotation**(). The method first determines if the line is a vertical line or a horizontal line and how the line points are related to each other in order to get an anti-clockwise or clockwise rotation angle. The angle itself is calculated as $\rho = \arctan(\frac{y}{x})$.

```
def getRotation(x1, y1, x2, y2):
    ...
    xdif = np.abs(x1-x2)
    ydif = np.abs(y1-y2)

    # Determine if horizontal or vertical line (suitable for this
    example)
    if(xdif > ydif):
        isHorizontal = True
    else:
```

```
        isHorizontal = False

    # let(x1,y1) be the origin
    # we want to know if must rotate the image clockwise or
    anticlockwise
    if(isHorizontal):
        # assume rotation angle to be clockwise
        rho = np.arctan(ydif/xdif)
        # if y - coordinate of second endpoint greater than zero
    then
        # rotationangle anticlockwise
        if(y1 > y2):
            rho = -rho

    else:
        # assume rotation angle to be clockwise
        rho = np.arctan(xdif/ydif)
        # if x - coordinate of second endpoint smaller than zero
    then
        # rotationangle anticlockwise
        if(x2 > x1):
            rho = -rho

    return rho
```

Listing 16: Get rotation angle of the rotated table

Afterwards the picture is rotated back to an upward orientation around the upper left corner of the table which can be seen in image 19.

CONTRACTOR	LICENSE	CERTIFICATE OF INSURANCE	FULL YEAR FEE	HALF YEAR FEE
General Contractor/Fence		X	$75	$37.50
Excavator/Concrete/Masonry		X	$75	$37.50
Carpenter		X	$75	$37.50
Plumber/Lawn Sprinkler	Illinois Dept. of Public Health Registration		$75	$37.50
Sewer		X	$75	$37.50
Electrician	* Elec. License	X	$75	$37.50
Communications Contractor		X	$75	$37.50
HVAC		X	$75	$37.50
Roofer	Roofing License issued by State of Illinois		$75	$37.50
Iron or Steel		X	$75	$37.50
Fire Protection/Sprinkler	Sprinkler License		$75	$37.50
Fire Protection/Alarm	Alarm License		0	0
Paving		X	$75	$37.50
Elevator	Elevator Co. License		$75	$37.50

Figure 19: Original image rotated back

The next step is to repair the lines with gaps. This is done with two separate Hough line detection steps, one for the vertical lines and one for the horizontal lines. The two separate steps are necessary because different parameters are needed to detect all lines, which can be seen in 20 and 21.

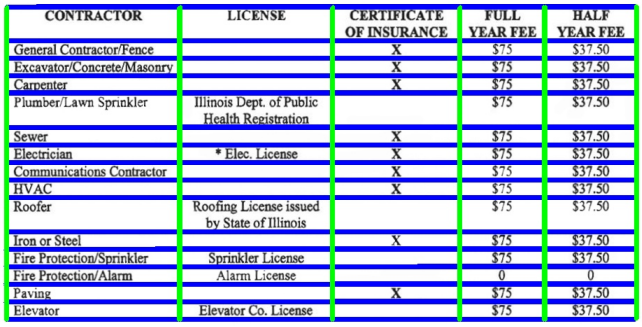
```
############################################################
# find lines with hough transformation in the canny edge
# detection image first the vertical lines and the most
# of the horizontal
```

```
lines_vert = getEdges(edges_rot, rho=1, theta=1 * np.pi /
                      180.0, threshold=300, minLineLength=100,
    maxGap=100)
# then the remaining horizontal lines
lines_hor = getEdges(edges_rot, rho=1, theta=17 * np.pi /
                     180.0, threshold=200, minLineLength=20, maxGap
    =150)
for line in lines_vert:
    # extract end points of lines
    x1, y1, x2, y2 = line[0]
    # and draw them to original image
    cv.line(imgWithLines_rot, (x1, y1), (x2, y2), (255, 0, 0), 4)
...
for line in lines_hor:
    # extract end points of lines
    x1, y1, x2, y2 = line[0]
    # and draw them to original image
    cv.line(imgWithLines_rot, (x1, y1), (x2, y2), (0, 255, 0), 4)
############################################################
...
```

Listing 17: Find vertical and horizontal lines

CONTRACTOR	LICENSE	CERTIFICATE OF INSURANCE	FULL YEAR FEE	HALF YEAR FEE
General Contractor/Fence		X	$75	$37.50
Excavator/Concrete/Masonry		X	$75	$37.50
Carpenter		X	$75	$37.50
Plumber/Lawn Sprinkler	Illinois Dept. of Public Health Registration		$75	$37.50
Sewer		X	$75	$37.50
Electrician	* Elec. License	X	$75	$37.50
Communications Contractor		X	$75	$37.50
HVAC		X	$75	$37.50
Roofer	Roofing License issued by State of Illinois		$75	$37.50
Iron or Steel		X	$75	$37.50
Fire Protection/Sprinkler	Sprinkler License		$75	$37.50
Fire Protection/Alarm	Alarm License		0	0
Paving		X	$75	$37.50
Elevator	Elevator Co. License		$75	$37.50

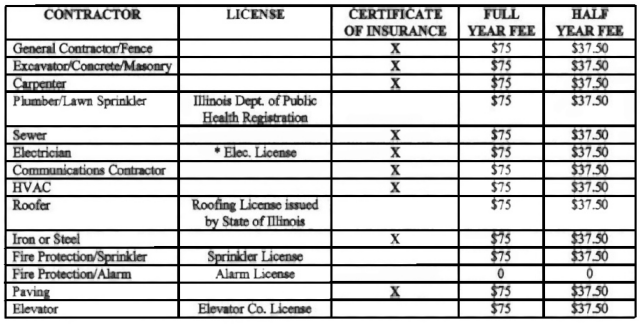
Figure 20: Detected vertical lines



CONTRACTOR	LICENSE	CERTIFICATE OF INSURANCE	FULL YEAR FEE	HALF YEAR FEE
General Contractor/Fence		X	$75	$37.50
Excavator/Concrete/Masonry		X	$75	$37.50
Carpenter		X	$75	$37.50
Plumber/Lawn Sprinkler	Illinois Dept. of Public Health Registration		$75	$37.50
Sewer		X	$75	$37.50
Electrician	* Elec. License	X	$75	$37.50
Communications Contractor		X	$75	$37.50
HVAC		X	$75	$37.50
Roofer	Roofing License issued by State of Illinois		$75	$37.50
Iron or Steel		X	$75	$37.50
Fire Protection/Sprinkler	Sprinkler License		$75	$37.50
Fire Protection/Alarm	Alarm License		0	0
Paving		X	$75	$37.50
Elevator	Elevator Co. License		$75	$37.50

Figure 21: Detected horizontal lines

After detecting all lines, the lines are drawn with color black and line width of $1pt$ into the image and the gaps are closed. To get a nice restored image the morphological operation of closing is applied which is shown in image 22.



CONTRACTOR	LICENSE	CERTIFICATE OF INSURANCE	FULL YEAR FEE	HALF YEAR FEE
General Contractor/Fence		X	$75	$37.50
Excavator/Concrete/Masonry		X	$75	$37.50
Carpenter		X	$75	$37.50
Plumber/Lawn Sprinkler	Illinois Dept. of Public Health Registration		$75	$37.50
Sewer		X	$75	$37.50
Electrician	* Elec. License	X	$75	$37.50
Communications Contractor		X	$75	$37.50
HVAC		X	$75	$37.50
Roofer	Roofing License issued by State of Illinois		$75	$37.50
Iron or Steel		X	$75	$37.50
Fire Protection/Sprinkler	Sprinkler License		$75	$37.50
Fire Protection/Alarm	Alarm License		0	0
Paving		X	$75	$37.50
Elevator	Elevator Co. License		$75	$37.50

Figure 22: Redraw lines and close the image

In the end the image is sharpened twice with a sharpening filter and image 23 shows the result of the restoration.

```
############################################################
# 1. Close the image
# define kernel for morphological operations
kernel = cv.getStructuringElement(cv.MORPH_RECT, (3, 3))

# dilate the image with the given kernel and save it
img_e = cv.erode(img_out, kernel, iterations=1)

```

```
# erode the image with the given kernel and save it
img_d = cv.dilate(img_e, kernel, iterations=1)
...
# 2. Sharpen the image
# define sharpening kernel
kernel = -np.ones((3, 3), np.float32)/9
kernel[1, 1] += 2

# sharpen the picture twice
img_out = cv.filter2D(img_d, -1, kernel)
img_out = cv.filter2D(img_d, -1, kernel)
##########################################################
...
```

Listing 18: Find vertical and horizontal lines

CONTRACTOR	LICENSE	CERTIFICATE OF INSURANCE	FULL YEAR FEE	HALF YEAR FEE
General Contractor/Fence		X	$75	$37.50
Excavator/Concrete/Masonry		X	$75	$37.50
Carpenter		X	$75	$37.50
Plumber/Lawn Sprinkler	Illinois Dept. of Public Health Registration		$75	$37.50
Sewer		X	$75	$37.50
Electrician	* Elec. License	X	$75	$37.50
Communications Contractor		X	$75	$37.50
HVAC		X	$75	$37.50
Roofer	Roofing License issued by State of Illinois		$75	$37.50
Iron or Steel		X	$75	$37.50
Fire Protection/Sprinkler	Sprinkler License		$75	$37.50
Fire Protection/Alarm	Alarm License		0	0
Paving		X	$75	$37.50
Elevator	Elevator Co. License		$75	$37.50

Figure 23: Restored image

6 Question 6

6.1 Task

For question 6 the task was to count coins in a given image.

6.2 Solution

As the assignment did not clearly specify if we should count the amount of coins or the monetary value of the coins in the image, we decided to count the monetary value, which includes the counting of coins given in the provided image.

```
# img loading and graying for algorithms
oimg  = cv2.cvtColor(cv2.imread("coins.jpg"),cv2.COLOR_BGR2RGB)
# copy for output
input_img = oimg.copy()
img = cv2.cvtColor(oimg,cv2.COLOR_RGB2GRAY)
# medianblur for noise reduction
```

```
img = cv2.medianBlur(img,7)
# HoughCircles with carefully chosen parameters to detect all the coins
circles = cv2.HoughCircles(img,cv2.HOUGH_GRADIENT,1,120,
                           param1=50,param2=30,minRadius=40,maxRadius=70)
circles = np.uint16(np.around(circles))
# variables for numbering and counting the amount
counter=0
sum_coins = 0
# make original image all white to display only the detected coins + values
oimg[:,:,:]=255
for i in circles[0,:]:
    counter+=1
    # draw the outer circle
    cv2.circle(oimg,(i[0],i[1]),i[2],(255,0,0),2)
    # check the detected coins
    sum_coins+=checkCoin(i[0], i[1], i[2], counter)
```

Listing 19: Circle detection

To detect the circles/coins in the given image, we decided to use a median filter with a kernelsize of 7x7 followed by the **HoughCircle()** function provided by openCV. Choosing a minimal distance between the detected circles of 120 pixels, 50 and 30 as the the two parameters for the **HOUGH_GRADIENT** and the min/max radius as 40 and 70 pixels gave us the desired result of detecting every coin in the image. We then applied our developed function **checkCoin()** to every circle detected to determine the monetary value of each coin.

```
    # get the avg color of the detected circle
    for x in range(x_offset-radius,x_offset+radius):
        for y in range(y_offset-radius,y_offset+radius):
            if((y-y_offset)*(y-y_offset)+(x-x_offset)*(x-x_offset)<=(radius*radius)):
                amount+=1
                r+=input_img[y,x,0]
                g+=input_img[y,x,1]
                b+=input_img[y,x,2]

    text = "["+str(int(r/amount))+","+str(int(g/amount))+ ","+str(int(b/amount))+"]"
    print("Circle Nr:" + str(number) + "\n" + text)
    # offset for display of text
    position = (x_offset-20,y_offset+4)
    value = checkValue(int(r/amount),radius)
    # console display of amount for testing
    print(checkValue(int(r/amount),radius))
    # write the values of the coin ontop of it
    cv2.putText(oimg, str(value),position,cv2.FONT_HERSHEY_SIMPLEX, 0.8,(255, 0, 0), 2)
    return value
```

Listing 20: Average color of coins

By looking at the average color of the coins we found that it would be the easiest way to categorize the coins by looking at the intensity values provided by the RED channel. We then used those values and the radius of the coins to determine their monetary values. Adding up the calculated values and using the **putText()** function of openCV we displayed the combined monetary value and the amount of coins in the given image.

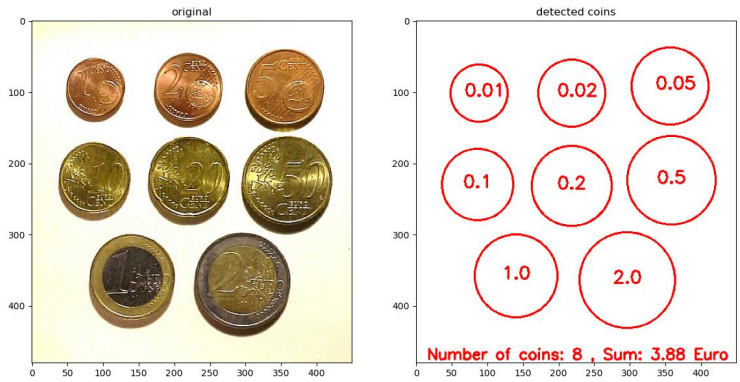


Figure 24: Amount and monetary value of coins

7 Question 7

7.1 Task

For question 7 the task was to use image processing techniques in order to solve a jigsaw puzzle. The reference image **Mona-Lisa**, shown in image 25, shall be assembled, using the pieces in image 26.

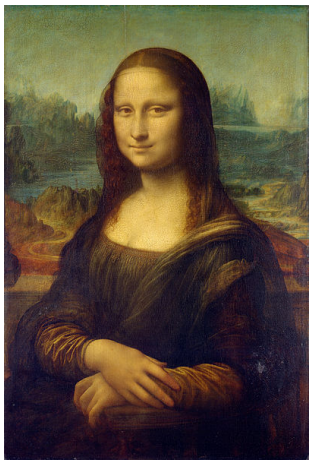


Figure 25: Leonardo da Vinci's Mona Lisa

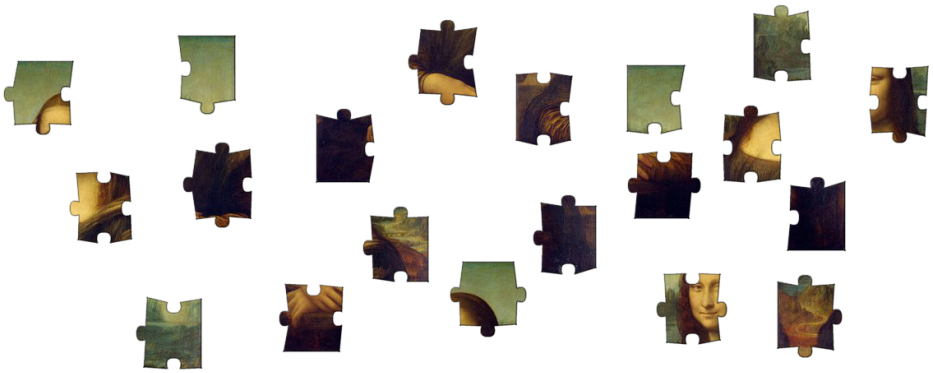


Figure 26: Puzzle pieces

7.2 Solution

7.2.1 General

The whole implementation for assembling the given puzzle is included in the file **puzzleAssembly.py** and the assembly process is called in the file **q7.py**. First we load both the reference image and the image with the single pieces. After this we use two different approaches for assembling the puzzle. Since

we use SIFT-descriptors and FLANN based feature matching, the first approach searches for keypoints in the whole reference picture and the second one searches in defined regions of interest, that is in parts of the reference picture.

```
import cv2 as cv
import matplotlib.pyplot as plt
from puzzleAssembly import assemblePuzzle, SiftParams
##################################################
#Read images
img_puzz = cv.cvtColor(cv.imread("hw5_puzzle_pieces.jpg"),cv.
    COLOR_BGR2RGB)
img_ref = cv.cvtColor(cv.imread("hw5_puzzle_reference.jpg"),cv.
    COLOR_BGR2RGB)

#First we perform feature matching on whole reference image
#Different SIFT parameters from bad to very good
...
#Empirically best parameters
params = SiftParams(nFeatPuzz = 0, nFeatRef = 0, nOctLayPuzz = 15,
nOctLayRef = 7, contrThrPuzz = 0.00014, contrThrRef = 0.0001,
edgeThrPuzz = 8900, edgeThrRef = 10165, sigPuzz = 0.95, sigRef =
    0.9)

#Get assembled puzzle and not matched pieces
notMatched, puzzle = assemblePuzzle(img_puzz, img_ref, params)

#Apply median filter to filter noise on the puzzle edges.
puzzle = cv.medianBlur(puzzle, ksize = 3)

#Conclusion:
# The results really depend on the choice of SIFT parameters.
# Several test showed that the most important parameters are
# the number of octave layers, the contrast threshold and the
# edge threshold. The pieces need a lot more octave layers
# then the reference picture and also the edge threshold has
# to be smaller.
##################################################
#now we try the region search

#Parameter for ROI feature matching
params_roi = SiftParams(nFeatPuzz = 0, nFeatRef = 0, nOctLayPuzz =
    16,
nOctLayRef = 12, contrThrPuzz = 0.0001, contrThrRef = 0.0001,
edgeThrPuzz = 7500, edgeThrRef = 10050, sigPuzz = 0.9, sigRef =
    0.9)

#Get assembled puzzle and not matched pieces
notMatched_roi, puzzle_roi = assemblePuzzle(img_puzz, img_ref,
params_roi, searchOnRegions = True, ybord=390, xbord = 195)

#Apply median filter to filter noise on the puzzle edges.
puzzle_roi = cv.medianBlur(puzzle_roi, ksize = 3)

#Conclusion:
#In this case results not only depend on the choice of SIFT
#parameters but also of ROI window. Sometimes smaller windows
#perform well for some pieces, but perform very bad for others.
#It can happen that, no homography can be found at all!
#In general bigger ROI windows performed better.
```

Listing 21: Assembling the puzzle

7.2.2 Assembling the puzzle: Extraction of pieces

The whole assembly process takes place in the method **assemblePuzzle**(). As a first step we extract the found contours of all the single puzzle pieces with the method **getcontours**(). Now it depends if the keypoint search is performed on the whole image or on regions of it. If the whole image is taken then we search and detect keypoints only once to reduce computations. Contour search has to be performed only once and the found contours are shown in image 27.

```
# Variable for final image
  puzzle = np.zeros_like(image)

  # Get contours of the puzzle pieces
  contours = getContours(img_puzz)

  # Variable for not matched pieces
  notMatchedPieces = []

  # Number of contours
  n = len(contours)

  # If keypoint and descriptor search is performed on the whole
  # reference image, we only have to do this once.
  if(not searchOnRegions):
      # Find keypoints and compute of the whole reference image.
      kp_ref, desc_ref = getSiftKeypAndDesc(
          image, nFeat=0, nOctLay=params.nOctLayRef, contrThr=
params.contrThrRef, edgeThr=params.edgeThrRef, sig=params.
sigRef)

  ...
```

Listing 22: Assembling the puzzle

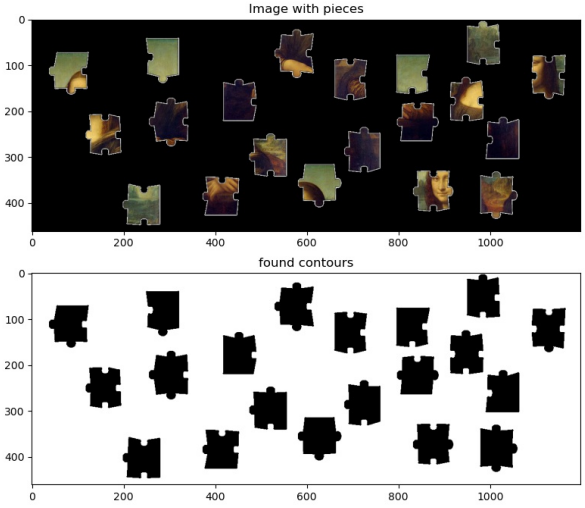


Figure 27: Puzzle piece contours

With this data we go into the main loop where we iterate over all found contours. For every contour we first extract the found piece from the pieces-images. For the i-th contour we calculate the coordinates and the size of the bounding box and extract the piece corresponding to the found contour. The first extracted piece is shown in image 28.

```
    # Main algorithm.
    for i in range(n):

        # Get a single puzzle piece
        piece = getPiece(img_puzz, contours, i)
        ...
    ...
```

Listing 23: Get piece i

After this step it depends wether we choose to search for keypoints in the whole reference image or in regions of it. We call either the method **putSinglePiece()** for the whole image or the method **putSinglePieceROI()** which will be explained in the next sections.

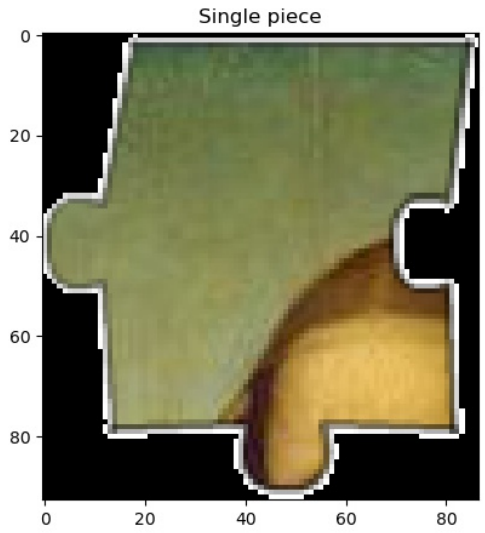


Figure 28: Puzzle piece

```
    for i in range(n):
        ...
        if(not searchOnRegions):
            # Try to find homography and to draw piece into
            # assembly
            status, fit = putSinglePiece(
                piece, image, i, params, kp_ref, desc_ref)
        else:
            status, fit = putSinglePieceROI(
                piece, image, i, params, ybord, xbord)
        ...
```

Listing 24: Put single piece

7.2.3 Putting single pieces

As mentioned if the whole image is taken as reference we actually search for keypoints only once in the reference image. With this we first detect SIFT-keypoints in the puzzle piece and try to match them with those of the reference image using a **FLANN based feature matching technique**. The matching technique and its combination with SIFT keypoints is briefly described in [1], [2], [3] and other literature. Matching of keypoints is shown in 29.

```
def putSinglePiece(piece, image, index, params: SiftParams, kp,
    desc):

    # Try to fit a single piece.
    # Search for keypoints and descriptos in the
    # whole reference image.

    # Status variable which indicates, if a homography could be
    found.
    status = 0

    # Find Keypoints and compute descriptors of the piece.
    kp_puzz, desc_puzz = getSiftKeypAndDesc(
        piece, nFeat=0, nOctLay=params.nOctLayPuzz, contrThr=params
    .contrThrPuzz, edgeThr=params.edgeThrPuzz, sig=params.sigPuzz)

    # Variable to hold the fitted piece.
    fit = np.zeros_like(image)

    # Execute FLANN based feature matching and save matches.
    matches = getMatches(desc_puzz, desc)
    ...
```

Listing 25: Feature matching

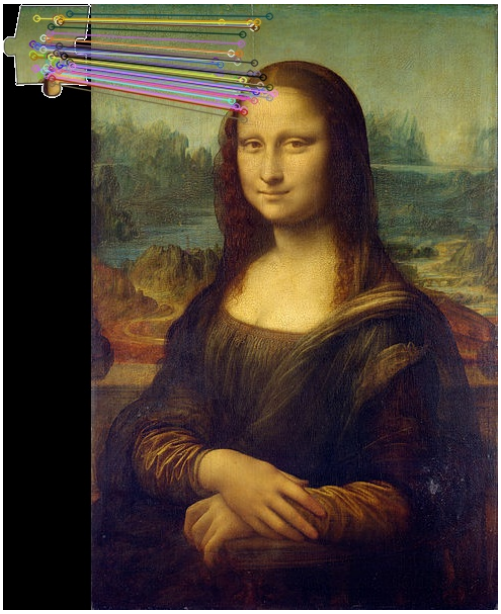


Figure 29: Matching keypoints

After this the next step is to find a homography using the **RANSAC**-algorithm [4] which can place the single piece into the right place with respect to the reference image. If a homography can be found we fit the piece and otherwise we mark the piece as not matched.

```
def putSinglePiece(piece, image, index, params: SiftParams, kp,
    desc):
    ...

    # Try to find Homography. Return status and the fitted piece.
    status, fit = findHomography(piece, image, kp_puzz, kp, matches
    )
    path = "out/whole_image/matches/matches_piece_" + str(index +
    1) + ".jpg"
    if(status == 1):
        # Homography could be found. Save Image.
        saveImageOfMatchedKP(piece, image, kp_puzz, kp, matches,
    path)

    return status, fit
```

Listing 26: Finding homography

If we search in rectangular regions of interest of a given size the algorithm behaves a bit different. We first detect keypoints of the puzzle piece and the keypoints in the region of interest and then we try to perform feature matching and homography search. This is done for all regions until a homography can be found. If no homography can be found we mark the pieces as not matched.

```
def putSinglePieceROI(piece, image, index, params: SiftParams,
    ybord=110, xbord=25):
    ...
    # Find Keypoints and compute descriptors of the piece.
    kp_puzz, desc_puzz = getSiftKeypAndDesc(
        piece, nFeat=0, nOctLay=params.nOctLayPuzz, contrThr=params
    .contrThrPuzz, edgeThr=params.edgeThrPuzz, sig=params.sigPuzz)
    ...
    # Try to fit the piece on the grid of regions.
    for i in range(qv + 1):
        for j in range(qh + 1):
            # Region with the dimensions of the piece + border
            if(i < qv and j < qh):
                y = i * (pieceHeight + ybord)
                x = j * (pieceWidth + xbord)

            # Rightmost region in horizontal strip of
            # height = pieceheight + border
            elif(i < qv and j >= qh):
                y = i * (pieceHeight + ybord)
                x = wref - (pieceWidth + xbord)

            # Bottom region in vertical strip of
            # width = pieceWidth + border
            elif(i >= qv and j < qh):
                y = href - (pieceHeight + ybord)
                x = j * (pieceWidth + xbord)

            # Region in the right lower corner
            elif(i >= qv and j >= qh):
                y = href - (pieceHeight + ybord)
                x = wref - (pieceWidth + xbord)

            # Get Region of Interest
            roi = getRegionOfInterest(image, y, x, h, w)

            # Find keypoints and compute in region of interest.
            kp_ref, desc_ref = getSiftKeypAndDesc(
                roi, nFeat=0, nOctLay=params.nOctLayRef,
                contrThr=params.contrThrRef,
                edgeThr=params.edgeThrRef,
                sig=params.sigRef)

            # Execute FLANN based feature matching and save
            # matches.
            matches = getMatches(desc_puzz, desc_ref)
            # Try to compute homography.
            status, fit = findHomography(
                piece, image, kp_puzz, kp_ref, matches)

            if(status == 1):
                # Homography could be found. Save Image.
                saveImageOfMatchedKP(piece, image,
                                     kp_puzz, kp_ref,
                                     matches, path)

                # Return status and the fitted piece.
                return status, fit

    return status, fit
```

Listing 27: Iterate over ROI until piece fits

7.2.4 Return to assemblePuzzle()

When we return to the main algorithm the last step for a single piece is to put the piece into the final assembly or to append it to the list of not matched pieces. The result of fitting one piece can be seen in 30.

```
    # Main algorithm.
    for i in range(n):
        ...
        if(not searchOnRegions):
            # Try to find homography and to draw piece into
            # assembly
            status, fit = putSinglePiece(
                piece, image, i, params, kp_ref, desc_ref)
        else:
            status, fit = putSinglePieceROI(
                piece, image, i, params, ybord, xbord)
        ...
        # If status equals 1, a homography was found
        if(status == 1):
            # Put the piece into the assembly
            puzzle = cv.add(puzzle, fit)

            # Write grayscale image of assembly
            cv.imwrite(str(path + "assembling/piece_" +
                       str(i + 1) + ".jpg"), puzzle)
        else:
            # mark piece i as not fitted
            notMatchedPieces.append(contours[i])

    ...
```

Listing 28: Fit the piece into the final image

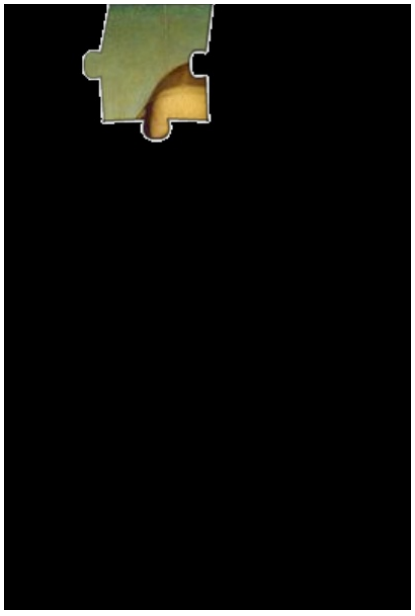


Figure 30: Fitting a piece

7.2.5 Final results and discussion

Through calling the method **assemblePuzzle()** we assemble the puzzle as whole or at least with the pieces that could be fitted. The final assembly when considering the whole image as reference is shown in 31.

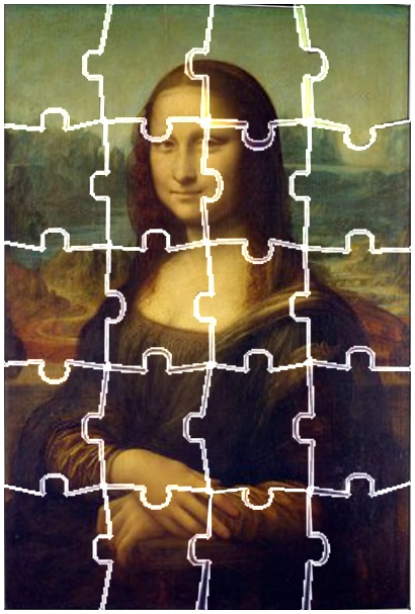


Figure 31: Final assembly, whole image reference

The method fitted all pieces quite well except for single pieces in the first and the last row. The difficulty of the whole process is to find parameters which work good for all pieces. Some default parameters work good for the most pieces but for others they have to be adjusted in order to find matching keypoints at all. Finding matches on the other hand does not say if a homography can be found or about the quality of the found homography. So the standard parameters of the SIFT-descriptor are useless and they had to be adjusted accordingly in many attempts. The border of the pieces produce some keypoints which lead to homographies that produce affine transformations resulting in tilted pieces when fitting. Nevertheless the quality of the final piece is not that bad and we were also able to assemble the whole puzzle. The final assembly in the ROI approach shown in image 32 is also quite good but it is computationally much more expensive.

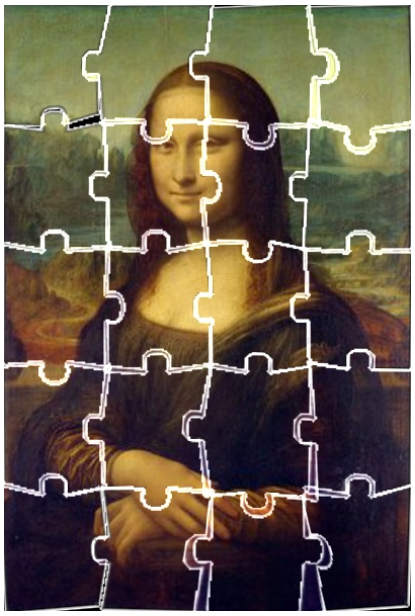


Figure 32: Final assembly, ROI reference

We first expected that sliding over regions with size equal to the piece size will find good keypoints but with this approach we were not able to fit all pieces and some of the fitted were really badly transformed into the final image. So the region of interest was successively enlarged to get better results. In the end the ROI window was nearly as big as half of the original reference image. So this approach does not pay of at all since parameter search is really difficult and the results are not better than those in the original approach. The original approach reflects the some intuitive human behavior when assembling puzzles in real life. That is take a piece and determine or estimate where it could fit, but keep distance to the whole image when searching for candidate regions. The second approach corresponds to taking a single piece and comparing it to every small region beginning in the left upper corner and maybe ending in the right lower corner, which is no fun at all.

8 Question 8

8.1 Task

In Scale Invariant Features Transform (SIFT), explain how key point localization works using Taylor expansion in details.

8.2 Solution

We assume that we have already created the scalespace for the image we want to process, as described in [5]. If our scale space is a continuous space with x, y coordinates and σ is the standard deviation of the convolution, we would now want to calculate the Laplace function. The extrema of the Laplacian

would then be candidates for the key points. We use the Difference of Gaussians technique, because we are only working with a discrete approximation of this continuous space. For each pair of horizontally adjacent pictures, the differences of the individual pixels are computed. A pixel is considered as extrema, if its gray value is larger/smaller than than that of all neighbors. The discrete extrema of these difference images will now be good approximations for the actual extrema. If the absolute value of an extrema is too small, it is treated as noise and is discarded. Now the coordinates can be refined, by using the quadratic Taylor expansion of the Difference of Gaussian. The Taylor expansion is given by:

$$D(x) = D + \frac{\partial D^T}{\partial x}x + \frac{1}{2}x^T\frac{\partial^2 D^T}{\partial x^2}x$$

where D and its derivatives are evaluated at the candidate keypoint and

$$x = (x, y, \sigma)^T$$

is the offset from this point.
This approach leads to too many keypoints being identified. To improve matching and stability, the interpolated location of the extremum is calculated. The location of the extremum x is determined by taking the derivative of this function with respect to x and setting it to zero. If the offset x is larger than 0.5 in any dimension, then that's an indication that the extremum lies closer to another candidate keypoint and the interpolation is then performed around that point instead. Otherwise the offset is added to its candidate keypoint to get the interpolated estimate for the location of the extremum.

References

[1] Marius Muja and David Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. volume 1, pages 331–340, 01 2009.

[2] Jerome H. Friedman, Jon Louis Bentley, and Raphael Ari Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Softw.*, 3(3):209–226, sep 1977.

[3] Chanop Silpa-Anan and Richard Hartley. Optimised kd-trees for fast image descriptor matching. In *2008 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–8, 2008.

[4] M. Fischler and R. Bolles. Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography. *Communications of the ACM*, 24(6):381–395, 1981.

[5] Dawid G. Lowe. Object recognition from local scale-invariant features. In *International Conference on Computer Vision, 20–25 September, 1999, Kerkyra, Corfu, Greece, Proceedings*, volume 2, pages 1150–1157, 1999.