MAMAN 11, intro to computer vision,22928

Ayala Shaubi-Mann
200244242

Question 1

The code used to execute entire sections of this question is q1.py

To execute code use “python q1.py –image1_path=<path to image> –image2_path=<path to image> --section=<one of A,B,C,D, E>”

If not mentioned, image1_path and image2_path are initialized to gan1.jpg and gan2.jpg in accordance, saved under input library.

Upon execution, outputs will be saved in outputs folder (will overwrite current files)

Main method code to be used

```
if __name__ == '__main__':
    # read input image file path from user and section to be executed
    arg_dict = command_line_args(argv=sys.argv)
    if "image1_path" in (arg_dict.keys()):
        image1_path = str(arg_dict.get('image1_path')[0])
    else:
        image1_path = "inputs//gan1.jpg"
    if "image2_path" in (arg_dict.keys()):
        image2_path = str(arg_dict.get('image2_path')[0])
    else:
        image2_path = "inputs//gan2.jpg"
    if "section" in (arg_dict.keys()):
        section = str(arg_dict.get('section')[0])
    else:
        section = "A"

    img1 = cv2.imread(image1_path)

    # A: canny edge detection
    if section == "A":
        print("A: canny edge detection")
        canny_edges = canny_edge_detector(img1, 50, 100)  # threshold1, threshold2

    # B: Harries corners detection
    # inputs are blockSize, ksize,and k
    if section == "B":
        print("B: Harries corners detection")
        rgb_img, dst = harries_conrners_detector(img1, 2, 3, 0.04)

    # C: SIFT keypoint and descriptors detection
    if section == "C":
        print("C: SIFT keypoint and descriptors detection")
        kp, des = sift(img1)

    # D: Matching interest points
    # 1. calc SIFT for each image
    # 2. Law ratio between the images (sift distance in both images and ratio
      between the most closest point to the second closes point)
    # take all matched points above decided threshold
    # inputs are image1 and image2 and law ratio threshold
    if section == "D":
        print("D: Matching interest points between two images")
        img2 = cv2.imread(image2_path)
        matching_points(img1, img2, 0.5)
```

```
    # E: Hough line in image
    # 1. get image with edges, using canny edges detector
    # 2. find lines using hough lines detector
    if section == "E":
        print("E: Hough line in image")
        hough_transform(img1)
```

A: Canny edge detection

The input image I used to perform canny edge detection given below, Figure 1.
Image file saved under inputs/gan1.jpg.
Execution command used is “python q1.py –image1_path=inputs//gan1.jpg --section=A”
After testing multiple parameters, presented output is with min value of 50 and max value of 100 in Canny’s edge detection.

Main method code to be used

```
def canny_edge_detector(input_img, threshold1, threshold2, draw=True, save=True):
    canny_img = cv2.cvtColor(np.copy(input_img), cv2.COLOR_BGR2GRAY)
    cv2.namedWindow('image', cv2.WINDOW_NORMAL)
    cv2.resizeWindow('image', 600, 600)
    cv2.imshow('image', canny_img)
    cv2.waitKey(1)

    edges = cv2.Canny(canny_img, threshold1, threshold2)
    if draw:
        cv2.namedWindow('CannyEdges', cv2.WINDOW_NORMAL)
        cv2.resizeWindow('CannyEdges', 600, 600)
        cv2.imshow('CannyEdges', edges)
        cv2.waitKey(1)
    if save:
        cv2.imwrite('outputs//cannyout_edges.jpg', edges)

    return edges
```

Output image presented in Figure 2 and saved under **outputs//cannyout_edges.jpg**

Figure 1- input image, gan1.jpg

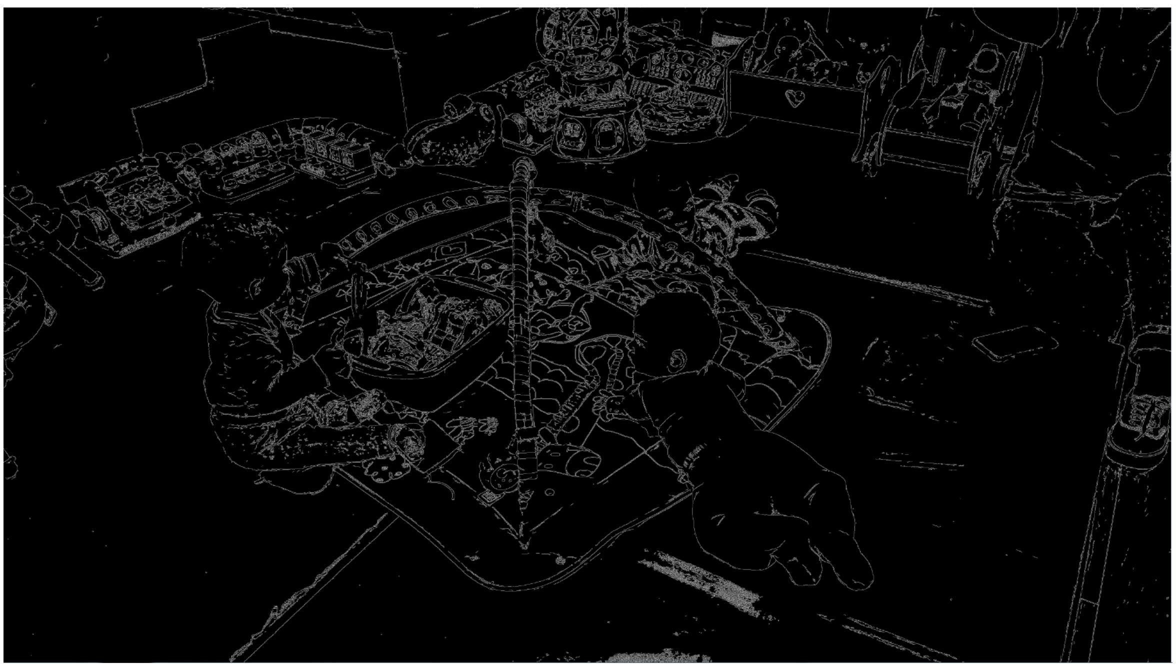


Figure 2- output image, Canny edge detection with minval=50, maxval=100

B: Harries corner detection

Execution command used is “python q1.py –image1_path=inputs//gan1.jpg --section=B”

After testing multiple parameters, presented output is when using block size of 2, k size of 3 and k value 0.04.

Main method code to be used

```python
def harries_conrners_detector(input_img, blockSize, ksize, k, draw=True, save=True):
    harriescrners_img = cv2.cvtColor(np.copy(input_img), cv2.COLOR_BGR2GRAY)
    gray = np.float32(harriescrners_img)
    dst = cv2.cornerHarris(gray, blockSize, ksize, k)

    dst = cv2.dilate(dst, None)

    # Mark corner index pixels on gray image
    b, g, r = cv2.split(input_img)  # get b,g,r
    rgb_img = cv2.merge([r, g, b])  # switch it to rgb
    rgb_img[dst > 0.01 * dst.max()] = [0, 0, 255]  # mark corner index pixels in red

    if draw:
        cv2.namedWindow('harries_corners', cv2.WINDOW_NORMAL)
        cv2.resizeWindow('harries_corners', 600, 600)
        cv2.imshow('harries_corners', rgb_img)
        cv2.waitKey(1)
    if save:
        cv2.imwrite('outputs//harriesout_corners.jpg', rgb_img)

    return rgb_img, dst
```

Output image presented in Figure 3, where corners marked as red spot, and saved under **outputs//harriesout_corners.jpg.**

Figure 3-output image, Harries corners detection with block size 2, ksize 3 and k value of 0.04

C: SIFT

Execution command used is "python q1.py –image1_path=inputs//gan1.jpg --section=C"
"sift" method under q1.py returning the list of interesting points and additional object containing it's descriptors (size 128 for each point).

Main method code to be used

```
def sift(input_img, draw=True, save=True):
    sift_img = np.copy(input_img)
    gray = cv2.cvtColor(sift_img, cv2.COLOR_BGR2GRAY)
    sift = cv2.xfeatures2d.SIFT_create()
    # sift.detect finds the keypoint in the images.
    # Each keypoint is a special structure which has many attributes like its (x,y)
    coordinates, size of the meaningful neighbourhood,
    # angle which specifies its orientation, response that specifies strength of
    keypoints etc.
    kp, des = sift.detectAndCompute(gray, None)

    # cv.drawKeyPoints() function which draws the small circles on the locations of
    keypoints
    if draw:
        img = cv2.drawKeypoints(gray, kp, sift_img,
                      flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
        cv2.namedWindow('sift_keypoints', cv2.WINDOW_NORMAL)
        cv2.resizeWindow('sift_keypoints', 600, 600)
        cv2.imshow('sift_keypoints', sift_img)
        cv2.waitKey(1)
    if save:
        cv2.imwrite('outputs//siftout_keypoints.jpg', sift_img)

    return kp, des
```

The visualization of above information presented in Figure 4.
Output image saved under **outputs// siftout_keypoints.jpg.**

Figure 4- output image, SIFT interesting points, with description visualization (scale and grandient)

D: Matching interesting points in 2 images

Execution command used is "python q1.py –image1_path=inputs//gan1.jpg –image2_path=inputs//gan2.jpg --section=D".

In this section execution include performing SIFT to each image, calc Law ratio between the images (sift distance in both images and Law ratio between the closest point to the second closest point).

Eventually we take all points below decided ratio threshold.

Main method code to be used

```
def matching_points(input_img1, input_img2, threshold=0.75, draw=True, save=True):
    # find the keypoints and descriptors with SIFT
    kp1, des1 = sift(input_img1, draw=False, save=False)
    kp2, des2 = sift(input_img2, draw=False, save=False)

    # BFMatcher with default params
    bf = cv2.BFMatcher()
    matches = bf.knnMatch(des1, des2, k=2)

    # Apply ratio test
    good = []
    for m, n in matches:
        if m.distance < threshold * n.distance:
            good.append([m])

    # cv2.drawMatchesKnn expects list of lists as matches.
    if draw:
        img3 = None
        img3 = cv2.drawMatchesKnn(img1=input_img1, keypoints1=kp1, img2=input_img2,
                                  keypoints2=kp2, outImg=img3,
                                  matches1to2=good,
                                  flags=2)
        cv2.namedWindow('matched_keypoints', cv2.WINDOW_NORMAL)
        cv2.resizeWindow('matched_keypoints', 600, 600)
        cv2.imshow('matched_keypoints', img3)
        cv2.waitKey(1)
    if save:
        cv2.imwrite('outputs//matched_keypoints.jpg', img3)

    return img3
```

Second input image used to perform the matching shown at Figure 5 and saved under inputs//gan2.jpg. Matched points presented at Figure 6. Original output image saved under **outputs//matched_keypoints.jpg**

Figure 5- input image 2, from same scene, used to perform matching

Figure 6- matched key points, with Law ratio threshold of 0.5

E: Hough transform to find lines

Execution command used is “python q1.py –image1_path=inputs//gan1.jpg --section=E”.
In this section execution, include performing Gaussian bluer to each image, following by edge detection (canny) to produce binary image.
I used minimum line length of 100, max line gap of 10, and voting threshold of 20.

Main method code to be used

```
def hough_transform(input_img, draw=True, save=True):
    gray = cv2.cvtColor(input_img, cv2.COLOR_BGR2GRAY)

    # find canny edges
    kernel_size = 3
    blur_gray = cv2.GaussianBlur(gray, (kernel_size, kernel_size), 0)
    edges = cv2.Canny(blur_gray, 50, 150)
    cv2.imwrite('outputs//canny_withblur.jpg', edges)

    # find hough lines using found edges
    lines = cv2.HoughLinesP(image=edges, rho=1, theta=np.pi / 180, threshold=20,
                            minLineLength=100, maxLineGap=10)
    # top_lines = lines[0:100]

    if draw:
        hough_lines = input_img
        for line in lines:
            x1, y1, x2, y2 = line[0]
            cv2.line(hough_lines, (x1, y1), (x2, y2), (0, 0, 255), 2)

        cv2.namedWindow('hough_lines', cv2.WINDOW_NORMAL)
        cv2.resizeWindow('hough_lines', 600, 600)
        cv2.imshow('hough_lines', hough_lines)
        cv2.waitKey(1)

        if save:
            cv2.imwrite('outputs//houghlines.jpg', hough_lines)
```

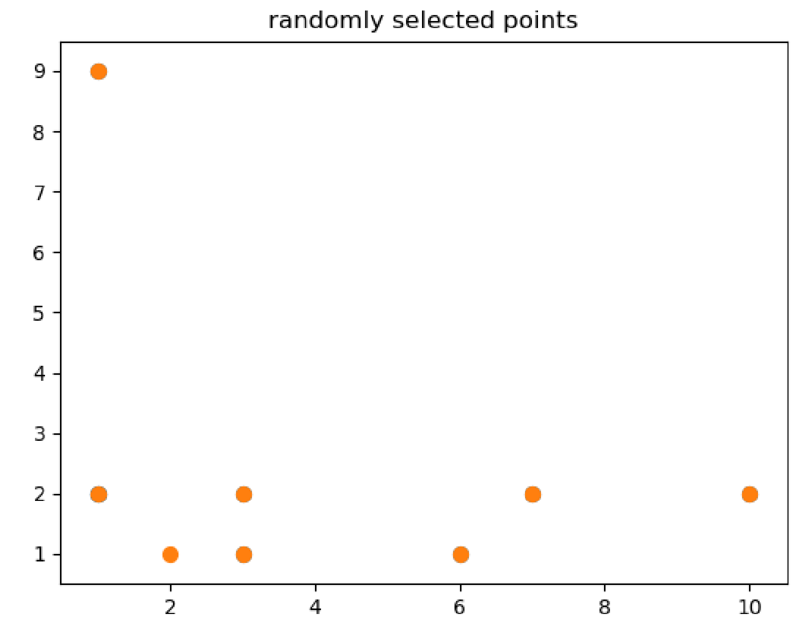
Detected lines presented in Figure 7, where Hough lines marked in Red. Original output image saved under **outputs//houghlines.jpg**

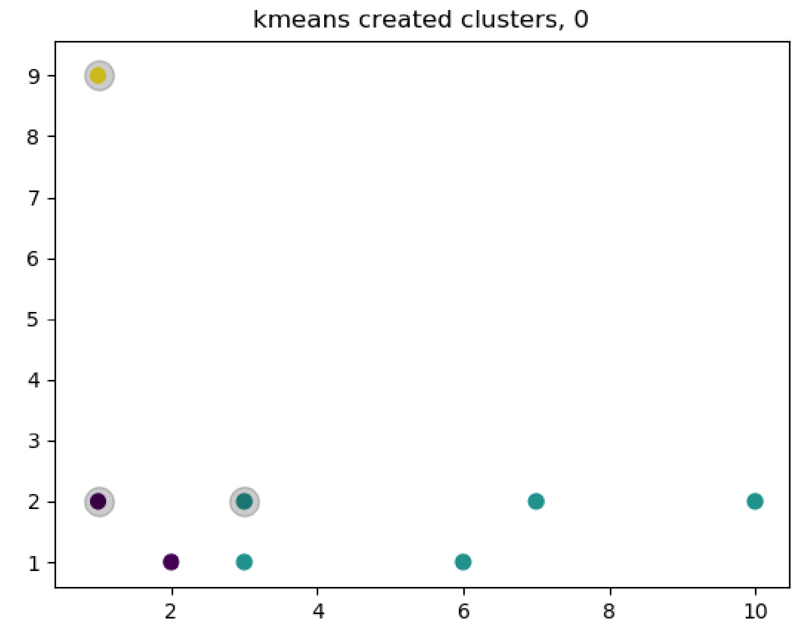
Figure 7- Hough transform to detect lines, min line length of 100, max gap 0f 10, min voting= 20

Question 2

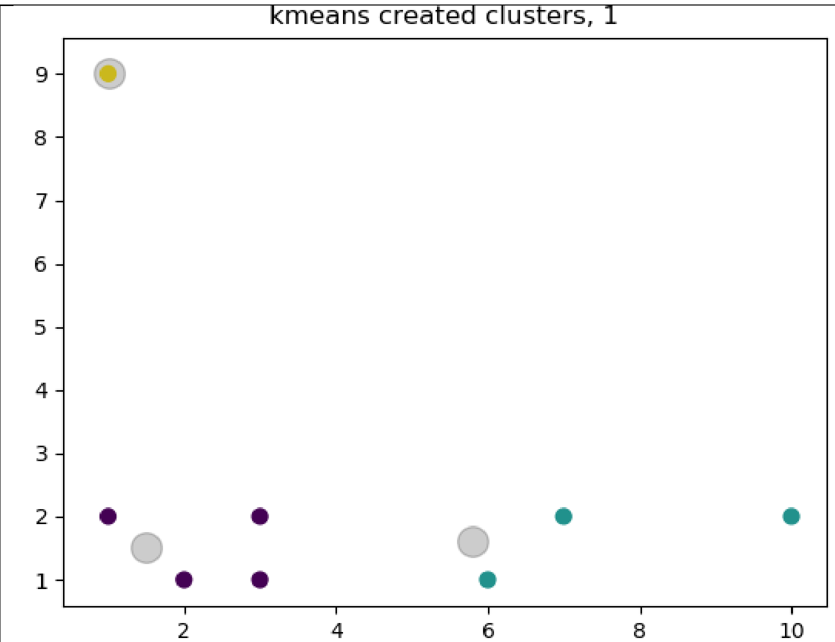
The code used to execute entire sections of this question is q2.py.
No configuration required.
To execute code use “python q2.py”.
Upon execution, 8 random points will be selected in 2d space [1,12]X[1,12].
On this points KMeans algorithm, with k=3, will be executed until convergence.

Example Run, including all steps specification:

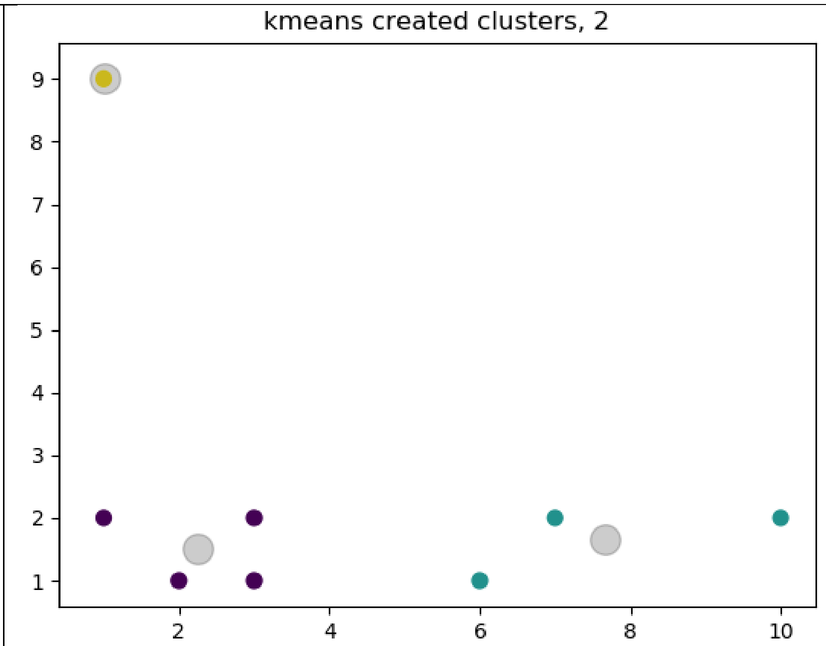
Step	**output**
Input points, random selection	 Points: (7, 2), (2,1), (10, 2), (3, 1), (3, 2), (1, 9), (1, 2), (6, 1)
Initialization- centers selection (k=3)	Selected centers for 3 clusters: Cluster 1 center(label=0)= (1, 2) Cluster 2 center(label=1)= (3, 2) Cluster 3 center(label=2)= (1,9)

<table>
<tr>
<td>Iteration 0.1- assign clusters to each point, based on pairwise distance (Euclidian distance) minimum argument, from each point to cluster center</td>
<td>


Points assignment to clusters:
(7, 2) – cluster 1
(2,1) – cluster 0
(10, 2) – cluster 1
(3, 1) – cluster 1
(3, 2) – cluster 1
(1, 9) – cluster 2
(1, 2) – cluster 0
(6, 1) – cluster 1</td>
</tr>
<tr>
<td>Iteration 0.2- update clusters centers based on:
Center(x, cluster)=average(xi) where point I assigned to cluster</td>
<td>Cluster 1 center(label=0)= (1.5,1.5)
Cluster 2 center(label=1)= (5.8,1.6)
Cluster 3 center(label=2)= (1,9)</td>
</tr>
<tr>
<td>Iteration 0.3 - Check convergence</td>
<td>Cluster centers changed => continue</td>
</tr>
</table>

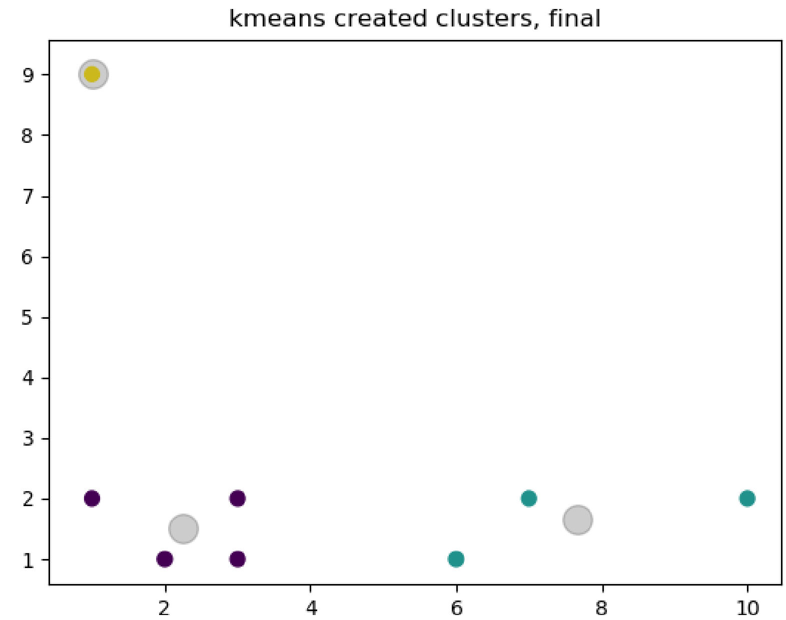
<table>
<tr>
<td>Iteration 1.1- assign clusters to each point, based on pairwise distance (Euclidian distance) minimum argument, from each point to cluster center</td>
<td>


Points assignment to clusters:
(7, 2) – cluster 1
(2,1) – cluster 0
(10, 2) – cluster 1
(3, 1) – cluster 0
(3, 2) – cluster 0
(1, 9) – cluster 2
(1, 2) – cluster 0
(6, 1) – cluster 1
</td>
</tr>
<tr>
<td>Iteration 1.2- update clusters centers based on:
Center(x, cluster)=average(xi) where point I assigned to cluster</td>
<td>Cluster 1 center(label=0)= (2.25,1.5)
Cluster 2 center(label=1)= (7.66666667,1.66666667)
Cluster 3 center(label=2)= (1,9)</td>
</tr>
<tr>
<td>Iteration 1.3 - Check convergence</td>
<td>Cluster centers changed => continue</td>
</tr>
</table>

<table>
<tr>
<td>Iteration 2.1- assign clusters to each point, based on pairwise distance (Euclidian distance) minimum argument, from each point to cluster center</td>
<td>

Points assignment to clusters:
(7, 2) – cluster 1
(2,1) – cluster 0
(10, 2) – cluster 1
(3, 1) – cluster 0
(3, 2) – cluster 0
(1, 9) – cluster 2
(1, 2) – cluster 0
(6, 1) – cluster 1</td>
</tr>
<tr>
<td>Iteration 2.2- update clusters centers based on:
Center(x, cluster)=average(xi) where point I assigned to cluster</td>
<td>Cluster 1 center(label=0)= (2.25,1.5)
Cluster 2 center(label=1)= (7.66666667,1.66666667)
Cluster 3 center(label=2)= (1,9)</td>
</tr>
<tr>
<td>Iteration 2.3- Check convergence</td>
<td>Cluster centers remain the same as iteration 1 => stop process</td>
</tr>
</table>

Final clusters



Points assignment to clusters:
(7, 2) – cluster 1
(2,1) – cluster 0
(10, 2) – cluster 1
(3, 1) – cluster 0
(3, 2) – cluster 0
(1, 9) – cluster 2
(1, 2) – cluster 0
(6, 1) – cluster 1

Centers:
Cluster 1 center(label=0)= (2.25,1.5)
Cluster 2 center(label=1)= (7.66666667,1.66666667)
Cluster 3 center(label=2)= (1,9)

Main method code to be used

```
def compute_knn_clusters(arrpoints, k, verbose=True, rseed=2):
    # 1. Randomly choose clusters
    arrpoints = np.asarray(arrpoints)

    if verbose:
        print("input points:")
        print(arrpoints)

    # Randomly select initial centers
    rng = np.random.RandomState(rseed)
    i = rng.permutation(arrpoints.shape[0])[:k]
    centers = arrpoints[i]

    itr = 0
    while True:
        # Assign labels based on closest center
        labels = pairwise_distances_argmin(arrpoints, centers)

        if verbose:
            print("iteration number: {itr}".format(itr=itr))
```

```
            print("cluster centers:")
            print(centers)
            print("labels for each point:")
            print(labels)
            plot_clusters(arrpoints,labels,centers,itr)

        # Find new centers from means of points
        new_centers = np.array([arrpoints[labels == i].mean(0)
                                for i in range(k)])

        # Check for convergence
        if np.all(centers == new_centers):
            break
        centers = new_centers
        itr = itr + 1

    if verbose:
        print("previous iteration centers equals to current iteration centers => process
                     finished at itr {itr}".format(itr=itr))
        print("final clusters centers:")
        print(centers)
        print("final label for each point:")
        print(labels)
        plot_clusters(arrpoints, labels, centers, iteration_idx="final")

    return centers, labels
```

Question 3

The code used to execute entire sections of this question is q3.py.
No configuration required.
To execute code with provided input points use "python q3.py".

Using least square method, the purpose is finding the best parabola to fit given input points (1, 3.96), (4, 27.96), (3, 15.15), (5, 45.8), (2, 7.07), (6, 69.4).
Parabola is defined by the equation $= ax^2 + bx + c$.
To find the best fit parameters a,b and c, according to least square method, we aim to solve the equation $B = (X^tX)^{-1}X^tY$ where, in our case X, B and Y will be defined as follows:

$$B = \begin{bmatrix} a \\ b \\ c \end{bmatrix}, X = \begin{bmatrix} x_1^2 & x_1 & 1 \\ \vdots & \vdots & \vdots \\ x_n^2 & x_n & 1 \end{bmatrix}, Y = \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix}$$

Placing the input points, we get the following matrix:

$$X = \begin{bmatrix} 1 & 1 & 1 \\ 16 & 4 & 1 \\ 9 & 3 & 1 \\ 25 & 5 & 1 \\ 4 & 2 & 1 \\ 36 & 6 & 1 \end{bmatrix}, Y = \begin{bmatrix} 3.96 \\ 27.96 \\ 15.15 \\ 45.8 \\ 7.07 \\ 69.4 \end{bmatrix}$$

Solving the equation $B = (X^tX)^{-1}X^tY$:

$$X^t = \begin{bmatrix} 1 & 1 & 1 \\ 16 & 4 & 1 \\ 9 & 3 & 1 \\ 25 & 5 & 1 \\ 4 & 2 & 1 \\ 36 & 6 & 1 \end{bmatrix}^t = \begin{bmatrix} 1 & 16 & 9 & 25 & 4 & 36 \\ 1 & 4 & 3 & 5 & 2 & 6 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

$$X^t X = \begin{bmatrix} 1 & 16 & 9 & 25 & 4 & 36 \\ 1 & 4 & 3 & 5 & 2 & 6 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 \\ 16 & 4 & 1 \\ 9 & 3 & 1 \\ 25 & 5 & 1 \\ 4 & 2 & 1 \\ 36 & 6 & 1 \end{bmatrix} = \begin{bmatrix} 2275 & 441 & 91 \\ 441 & 91 & 21 \\ 91 & 21 & 6 \end{bmatrix}$$

$$(X^t X)^{-1} = \begin{bmatrix} 2275 & 441 & 91 \\ 441 & 91 & 21 \\ 91 & 21 & 6 \end{bmatrix}^{-1} = \begin{bmatrix} 0.02678571\,2275 & -0.1875 & 0.25 \\ -0.1875 & 1.3696428691 & -1.95 \\ 0.2591 & -1.95 & 3.2 \end{bmatrix}$$

$$X^t Y = \begin{bmatrix} 1 & 16 & 9 & 25 & 4 & 36 \\ 1 & 4 & 3 & 5 & 2 & 6 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 3.96 \\ 27.96 \\ 15.15 \\ 45.8 \\ 7.07 \\ 69.4 \end{bmatrix} = \begin{bmatrix} 4259.35 \\ 820.79 \\ 169.34 \end{bmatrix}$$

$$B = (X^t X)^{-1} X^t Y = \begin{bmatrix} 0.02678571\,2275 & -0.1875 & 0.25 \\ -0.1875 & 1.3696428691 & -1.95 \\ 0.2591 & -1.95 & 3.2 \end{bmatrix} \begin{bmatrix} 4259.35 \\ 820.79 \\ 169.34 \end{bmatrix} = \begin{bmatrix} 2.52660714 \\ -4.65196429 \\ 6.185 \end{bmatrix}$$

We got the parabola equation $\boldsymbol{y = 2.52660714x^2 - 4.65196429x + 6.185}$ described in Figure 8.

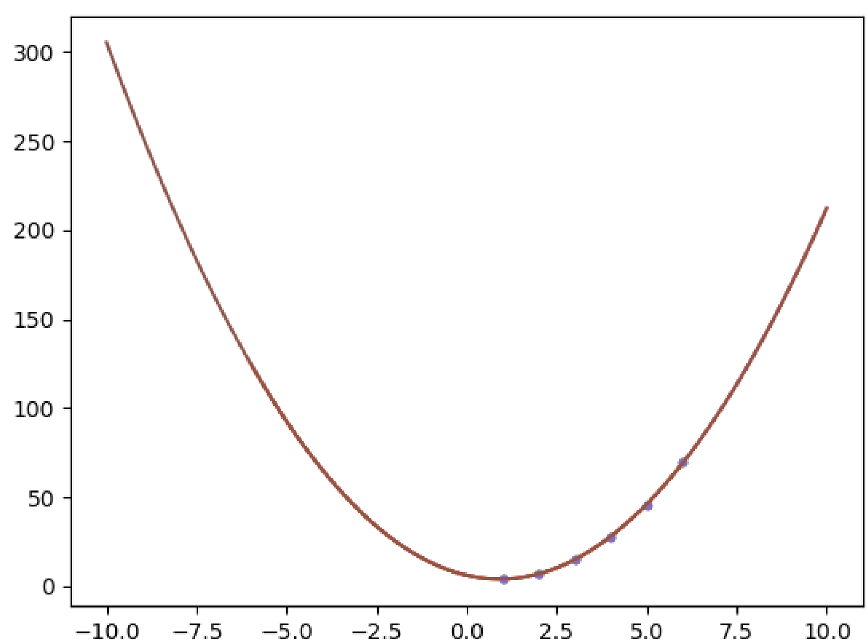


Figure 8- Parabola described by the equation $y = 2.52660714x^2 - 4.65196429x + 6.185$

Main method code to be used

```python
if __name__ == '__main__':
    input_points = np.asarray([[1, 3.96], [4, 27.96], [3, 15.15], [5, 45.8], [2, 7.07],
                               [6, 69.4]])
    X = input_points[:, 0]
    X_matx = np.transpose([np.square(X), X, np.zeros(X.shape) + 1])
    Y = input_points[:, 1]
    Y_matx = Y.reshape(-1, 1)

    # define Y,X,B
    print("Equestion params: ")
    print("Y=")
    print(Y_matx)
    print()
    print("X=")
    print(X_matx)
    print()
    print("B=")
    B_matx = np.asarray(["a", "b", "c"])
    B_matx = B_matx.reshape(-1, 1)
    print(B_matx)
    print()

    print("Solving equation XtXB=XtY ==> B=((XtX)^(-1))*XtY")

    # calc (XtX)
    xtx= np.matmul(np.transpose(X_matx), X_matx)
    print("XtX=")
    print(xtx)
    print()
    # calc (XtX)^(-1)
    xtx_inv = np.linalg.inv(xtx)
    print("XtX^(-1)=")
    print(xtx_inv)
    print()
    # calc (XtY)
    xty = np.matmul(np.transpose(X_matx), Y_matx)
    print("XtY=")
    print(xty)
    print()
    # calc ((XtX)^(-1))*(XtY)
    B_matx = np.matmul(xtx_inv, xty)
    print("((XtX)^(-1))*(XtY)=B=")
    print(B_matx)
    print()

    z = np.polyfit(X, Y, 2)
    p = np.poly1d(z)
    print("fitted parabula:")
    print("a={}".format(z[0]))
    print("b={}".format(z[1]))
    print("c={}".format(z[2]))

    xp = np.linspace(-6, 10, 100)
    _ = plt.plot(X, Y, '.', xp, p(xp), '-')

    plt.show()
```

Question 4

The code used to execute entire sections of this question is q4.py.
No configuration required.
To execute code with provided input points use “python q4.py”.

Using PCA method, the purpose is finding the imposition of points to a straight line, given input points (2.5, 2.9), (0.5, 1.2), (2.2, 3.4), (1.9, 2.7), (3.1, 3.5), (2.3, 3.2), (2, 2.1), (1, 1.6), (1.5, 2.1),(1.1, 1.4).

To gain 2D PCA we will perform the following steps:

1. normalize all points to be with (0,0) average by $(x_{norm_i}, y_{norm_i}) = (x_i - x_{avg}, y_i - y_{avg})$
   (0.69 0.49), (-1.31 -1.21), (0.39 0.99), (0.09 0.29), (1.29 1.09), (0.49 0.79), (0.19 -0.31), (-0.81 -0.81), (-0.31 -0.31), (-0.71 -1.01)
2. calculate covariance matrix $COV = \frac{X^t X}{n-1}$
   $$\boldsymbol{COV} = \begin{bmatrix} \mathbf{0.61655556} & \mathbf{0.61544444} \\ \mathbf{0.61544444} & \mathbf{0.71655556} \end{bmatrix}$$
3. calculate eigenvalues by solving the equation $0 = \det(COV - \lambda * I)$
   $$\det(COV - \lambda * I) = \det\left(\begin{bmatrix} \mathbf{0.61655556} - \lambda & \mathbf{0.61544444} \\ \mathbf{0.61544444} & \mathbf{0.71655556} - \lambda \end{bmatrix}\right)$$
   $$= (\mathbf{0.61655556} - \lambda)(\mathbf{0.71655556} - \lambda) - \mathbf{0.61544444}^2$$
   By solving the quadratic equation for λ, we will have two eigenvalues
   $\lambda_{\mathbf{1}} = \mathbf{1.2840277121727839}$, $\lambda_2 = \mathbf{0.0490833989383273}$
4. calculate eigenvectors by solving the equation $COV * \begin{bmatrix} v_{i1} \\ v_{i2} \end{bmatrix} = \lambda_1 \begin{bmatrix} v_{i1} \\ v_{i2} \end{bmatrix}, COV * \begin{bmatrix} v_{i1} \\ v_{i2} \end{bmatrix} = \lambda_2 \begin{bmatrix} v_{i1} \\ v_{i2} \end{bmatrix}$
   $$\boldsymbol{v_1} = \begin{bmatrix} -\mathbf{0.6778734} \\ -\mathbf{0.73517866} \end{bmatrix} \quad \boldsymbol{v_2} = \begin{bmatrix} -\mathbf{0.73517866} \\ \mathbf{0.6778734} \end{bmatrix}$$
5. as we want to get horizontal 1d line, we set $W = [v_1]$ (single PCA component)
   $$\boldsymbol{W} = \begin{bmatrix} -\mathbf{0.6778734} \\ -\mathbf{0.73517866} \end{bmatrix}, \boldsymbol{W^t} = [-\mathbf{0.6778734} \quad -\mathbf{0.73517866}]$$
6. get PCA points by $x_{pca} = W^t \begin{bmatrix} x_{norm} \\ y_{norm} \end{bmatrix}$
   PCA points:
   (-0.82797019),(1.77758033),(-0.99219749),(-0.27421042),(-1.67580142),(-0.9129491,),(0.09910944),(1.14457216),(0.43804614),(1.22382056)
7. Restore points from PCA points using $\begin{bmatrix} x \\ y \end{bmatrix} = x_{pca} * W + \begin{bmatrix} x_{avg} \\ y_{avg} \end{bmatrix}$
   Restored points: **(2.37125896,3.01870601),(0.60502558,1.10316089),(2.48258429,3.13944242),(1.99587995,2.61159364),(2.9459812,3.64201343),(2.42886391,3.08118069),(1.74281635,2.33713686),(1.03412498,1.56853498),(1.51306018,2.08795783),(0.9804046,1.51027325)**

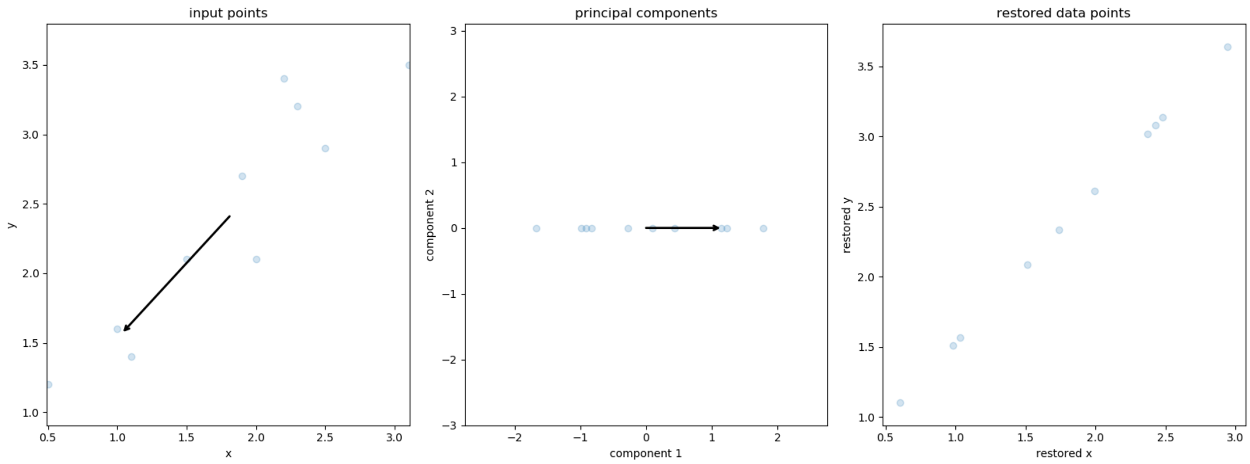


Figure 9- original points, PCA points and restored points with data loss

Main method code to be used

```
if __name__ == '__main__':
    input_points = np.asarray(
        [[2.5, 2.9], [0.5, 1.2], [2.2, 3.4], [1.9, 2.7], [3.1, 3.5], [2.3, 3.2],
          [2, 2.1], [1, 1.6], [1.5, 2.1], [1.1, 1.4]])
    print("input points:")
    print(input_points)
    print()
    X = input_points[:, 0]
    Y = input_points[:, 1]

    #get PCA points and componenets
    pca = sklearnPCA(n_components=1)  # 1-dimensional PCA
    points_pca = pca.fit_transform(input_points)
    print("sample covariance = (XtX)/(n-1): ")
    cov_matrix = pca.get_covariance()
    print(cov_matrix)
    print()

    print("PCA componenets (W) : ")
    print(pca.components_)
    print()

    print("PCA eigenvalues = (v_i)t*(cov_matrix*v_i) where v_i is an eigenvector: ")
    for eigenvector in pca.components_:
        print(np.dot(eigenvector.T, np.dot(cov_matrix, eigenvector)))
    print()
    # eigenvalues can be produces by solving det(covmatrix-lambada*I)=0
    # eigenvectors can be produces by solving covmatrix*v = lambada*v

    print("each point converted using (x_new)=Wt((x,y)-(x_avg,y_avg)):")
    print(points_pca)
    print()

    # plot input data
    fig, ax = plt.subplots(1, 3, figsize=(16, 6))
    fig.subplots_adjust(left=0.0625, right=0.95, wspace=0.1)
    ax[0].scatter(input_points[:, 0],input_points[:, 1], alpha=0.2)
    # draw PCA vectors on original scatter
    for length, vector in zip(pca.explained_variance_, pca.components_):
        v = vector * np.sqrt(length)
        draw_vector(pca.mean_, pca.mean_ + v, ax=ax[0])
        ax[0].axes.autoscale(enable=True, axis='both', tight=True)
```

```python
    ax[0].axis('equal')
    ax[0].set(xlabel='x', ylabel='y', title='input points')

    # plot principal components- 1d
    ax[1].scatter(points_pca[:, 0],np.zeros(len(points_pca[:, 0])), alpha=0.2)
    draw_vector([0, 0], [np.sqrt(pca.explained_variance_[0]), 0], ax=ax[1])
    ax[1].axis('equal')
    ax[1].set(xlabel='component 1', ylabel='component 2',
              title='principal components',
              xlim=(-5, 5), ylim=(-3, 3.1))

    # plot restored data
    print("each point restored using (x_restored,y_restored)=x_pca*W+(x_avg,y_avg)):")
    points_restoration = pca.inverse_transform(points_pca)
    print(points_restoration)

    ax[2].scatter(points_restoration[:, 0], points_restoration[:, 1], alpha=0.2)
    ax[2].axis('equal')
    ax[2].set(xlabel='restored x', ylabel='restored y', title='restored data points')
    plt.tight_layout()
    plt.show()
```