Farimah Rashidi(99222040)

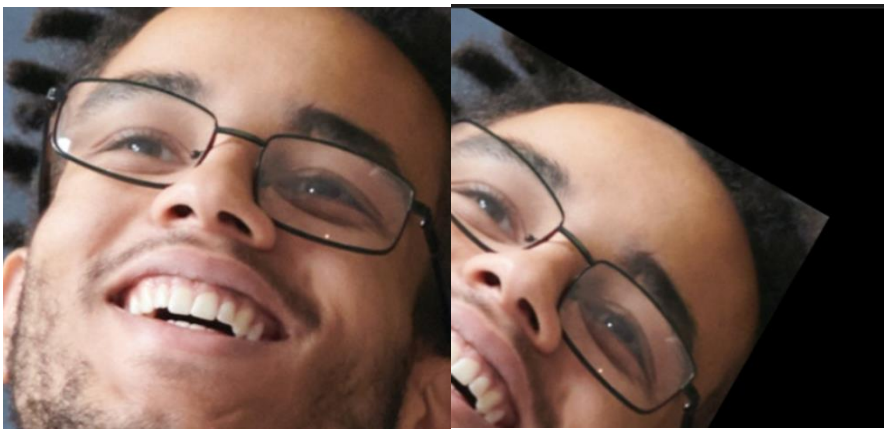
Question1

some linear transformations:

```
[ ] import cv2
    import numpy as np
    from google.colab.patches import cv2_imshow

[ ] #loading the image
    image = cv2.imread('image1.jpg')

[ ] #converting the image to grayscale
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

[ ] #detecting faces in the image
    face_cascade = cv2.CascadeClassifier(cv2.data.haarcascades + 'haarcascade_frontalface_default.xml')
    faces = face_cascade.detectMultiScale(gray, scaleFactor=1.1, minNeighbors=5, minSize=(30, 30))
```

```
for (x, y, w, h) in faces:
    #extracting the face region of interest (ROI)
    face = image[y:y+h, x:x+w]

    cv2_imshow(cv2.resize(face, (400, 400)))
    cv2.waitKey(0)

    #first linear transformation: Rotating faces by 30 degrees
    angle = 30
    angle_rad = np.deg2rad(angle)
    cos_theta = np.cos(angle_rad)
    sin_theta = np.sin(angle_rad)

    #rotation matrix
    M = np.array([[cos_theta, -sin_theta, 0], [sin_theta, cos_theta, 0]], dtype=np.float32)

    #finding center of rotation
    center = np.array([w/2, h/2])
    #implement the rotation on each face
    face_rotated = cv2.warpAffine(face, M, (w, h))

    #display new faces
    cv2_imshow(cv2.resize(face_rotated, (400, 400)))
    cv2.waitKey(0)
    cv2.destroyAllWindows()
```

One example(code includes all faces):



I applied a linear transformation to rotate the face by 30 degrees using an affine transformation matrix. The rotation is implemented using trigonometric functions to calculate the sine and cosine of the rotation angle in radians. Then I displayed rotated face using cv2_imshow, and the process is repeated for each detected face.

```python
#loading the image
image = cv2.imread('image1.jpg', cv2.IMREAD_GRAYSCALE)

face_cascade = cv2.CascadeClassifier(cv2.data.haarcascades + 'haarcascade_frontalface_default.xml')

#detecting faces in the image
faces = face_cascade.detectMultiScale(image, scaleFactor=1.1, minNeighbors=5, minSize=(30, 30))

shear_factor = 0.4

for (x, y, w, h) in faces:
    #extracting the face region from the original image
    face_region = image[y:y+h, x:x+w]

    #creating a blank canvas for the sheared face region
    sheared_face_region = np.zeros_like(face_region)

    height, width = face_region.shape
```

```python
for (x, y, w, h) in faces:
    #extracting the face region from the original image
    face_region = image[y:y+h, x:x+w]

    #creating a blank canvas for the sheared face region
    sheared_face_region = np.zeros_like(face_region)

    height, width = face_region.shape

    for y_face in range(height):
        for x_face in range(width):
            #calculating the new x-coordinate after shearing
            x_new = int(x_face + y_face * shear_factor)

            if x_new >= 0 and x_new < width:
                sheared_face_region[y_face, x_new] = face_region[y_face, x_face]

    image[y:y+h, x:x+w] = sheared_face_region

#showing image with sheared faces
from google.colab.patches import cv2_imshow
cv2_imshow(image)
```

I detected faces in the image using a Haar cascade classifier, and then I applied a shear transformation to the detected faces. The shear transformation is applied to each detected face region individually by looping through the pixels in the face region and calculating the new x-coordinate after applying the shear transformation.

some non-linear transformations:

```python
from google.colab.patches import cv2_imshow

image = cv2.imread('image1.jpg')

gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

face_cascade = cv2.CascadeClassifier(cv2.data.haarcascades + 'haarcascade_frontalface_default.xml')

faces = face_cascade.detectMultiScale(gray, scaleFactor=1.1, minNeighbors=5, minSize=(30, 30))

for (x, y, w, h) in faces:
    #extracting the face region from the image
    face = image[y:y+h, x:x+w]

    c = 15
    #performing the log transformation
    face_output = c * np.log(1 + face)

    face_output = np.uint8(face_output)

    #replacing the original face region with the transformed face region
    image[y:y+h, x:x+w] = face_output

cv2_imshow(image)
```

I converted image to grayscale, and then I detected faces in the image using a Haar cascade classifier. The detected faces are then processed using a log transformation. The log transformation is applied to each pixel in the face region by taking the natural logarithm of the pixel intensity value plus a constant (c) to avoid taking the log of zero. Then I replaced The transformed face region with the original face region in the original image.

```python
image = cv2.imread('image1.jpg')

gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

face_cascade = cv2.CascadeClassifier(cv2.data.haarcascades + 'haarcascade_frontalface_default.xml')

faces = face_cascade.detectMultiScale(gray, scaleFactor=1.1, minNeighbors=5, minSize=(30, 30))

for (x, y, w, h) in faces:
    #extracting the face region from the image
    face = image[y:y+h, x:x+w]

    k1 = 0.0000000001
    k2 = 0.0000000001

    #center of the face region
    cx = x + w/2
    cy = y + h/2
```

```python
    #iterating over each pixel in the face region
    for i in range(x, x+w):
        for j in range(y, y+h):
            #converting pixel coordinates to polar coordinates
            r = np.sqrt((i - cx)**2 + (j - cy)**2)
            theta = np.arctan2(j - cy, i - cx)


            rd = r * (1 + k1 * r**2 + k2 * r**4)
            xi = int(cx + rd * np.cos(theta))
            yi = int(cy + rd * np.sin(theta))

            if xi >= x and xi < x + w and yi >= y and yi < y + h:
                #replacing the pixel with the transformed pixel
                image[j, i] = face[yi - y, xi - x]

cv2_imshow(image)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

For each detected face, I applied a radial distortion transformation to the pixels within the face region. The transformation is based on polar coordinates, where the radial distance from the center of the face is modified by two constants (k1 and k2) and added to the original distance. The resulting transformed pixel coordinates are used to replace the original pixels in the face region.

```python
image = cv2.imread('image1.jpg')

gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

face_cascade = cv2.CascadeClassifier(cv2.data.haarcascades + 'haarcascade_frontalface_default.xml')

faces = face_cascade.detectMultiScale(gray, scaleFactor=1.1, minNeighbors=5, minSize=(30, 30))

for (x, y, w, h) in faces:
    #extracting the face region from the image
    face = image[y:y+h, x:x+w]

    #center of the face region
    cx = x + w/2
    cy = y + h/2

    k1 = 0.00000000001
    k2 = 0.00000000001
```

```python
    #iterating over each pixel in the face region
    for i in range(x, x+w):
        for j in range(y, y+h):
            #converting pixel coordinates to polar coordinates
            r = np.sqrt((i - cx)**2 + (j - cy)**2)
            theta = np.arctan2(j - cy, i - cx)

            rd = r / (1 + k1 * r**2 + k2 * r**4)
            xi = int(cx + rd * np.cos(theta))
            yi = int(cy + rd * np.sin(theta))

            if xi >= x and xi < x + w and yi >= y and yi < y + h:
                #replacing the pixel with the transformed pixel
                image[j, i] = face[yi - y, xi - x]

cv2_imshow(image)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

The transformation is based on polar coordinates, where the radial distance from the center of the face is divided by a polynomial function of r (r / (1 + k1 * r^2 + k2 * r^4)). The resulting transformed pixel coordinates are used to replace the original pixels in the face region.

how faces are detected by opencv: In this code, I used the Haar cascade classifier to detect faces in the grayscale image.The 'cv2.CascadeClassifier' class is used to load the Haar cascade classifier XML file, which contains the trained classifier for detecting faces. The XML file is provided by OpenCV and is stored in the 'cv2.data.haarcascades' directory. In this code, I used the haarcascade_frontalface_default.xml file, which is a trained classifier for detecting frontal faces. Then i called the cv2.CascadeClassifier.detectMultiScale() method on the grayscale image, passing several parameters such as scaleFactor, minNeighbors, and minSize to configure the detection process. The scaleFactor specifies the scale factor for reducing the image during the detection process to improve efficiency, while minNeighbors specifies the minimum number of neighbors required for a region to be considered as a face. minSize specifies the minimum size of the detected face region.

The detectMultiScale() method returns a list of rectangles, where each rectangle represents the coordinates (x, y), width (w), and height (h) of a detected face region in the image. These coordinates are then used to extract the face regions from the original image for further processing or manipulation, as done in the code by iterating over the detected faces and applying the radial undistortion transformation to each face region.