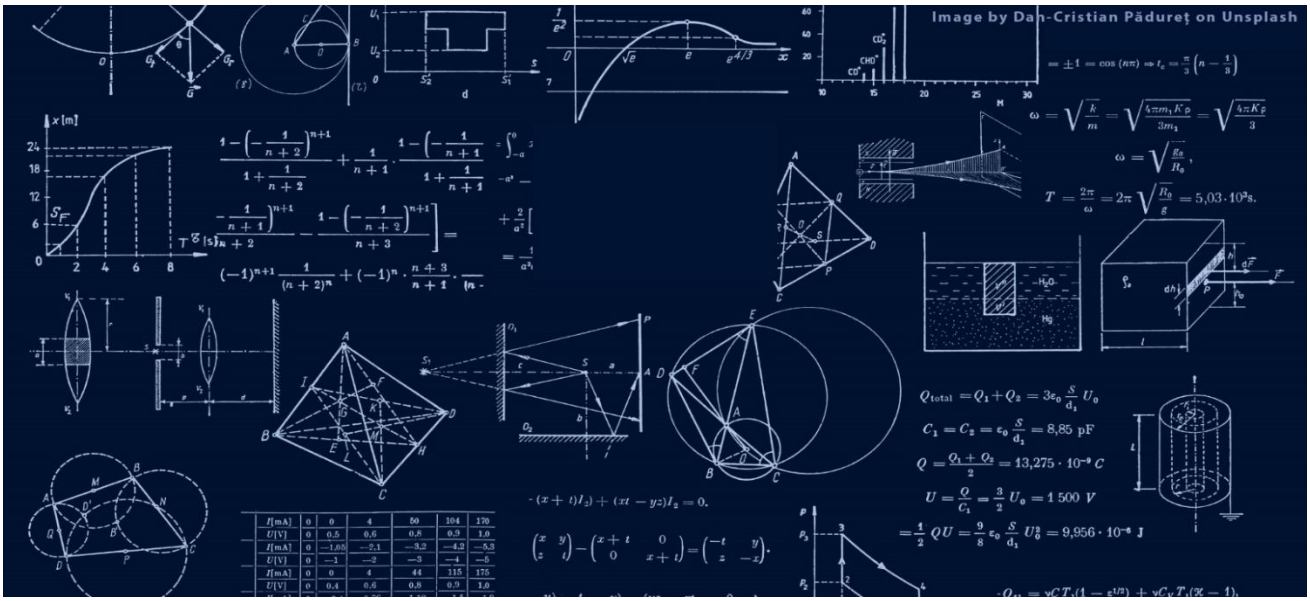
Faculty of Engineering Alexandria University

Electronics and Communication Engineering

Computational Mathematics

Term Project : **PART1**



الرقم الجامعي	الاسم
20010843	عبدالرحمن محمد عصام الدين حافظ خضر محمد الروبي
20010944	علي عز الدين علي عقل
20011178	محمد ابراهيم محمد مجاهد
20011949	مصطفي محمد عبدالعال عبدالهادي
20010109	أحمد صالح محمد محمود عبيد
19015430	أسماء حسن مختار ابوشادى
20010777	عبدالرؤوف فتحي المغربي
20010392	بريهان حسام الدين امام

Part I: Image processing:

Introduction:

For this part we used python specifically NumPy, NumPy is library used for working with matrices and arrays. We used it to divide the image into a matrix of pixels, then do basic processing.

We made a GUI to show the results of the processing on images and ease its use. For the interface we used OpenCV library **without the use of its built-in functions** for any of the image processing.

Code analysis:

We divided the main processing into functions to make the code cleaner. The main functions were rotate , flip , scale , shearX and shearY

- First of all, the rotation function,

```
def rotate(image):
    height=image.shape[0]
    width=image.shape[1]

    # intializing rotation matrix to rotate image 90 degrees clockwise
    rotation_mat = np.array([[0,-1],[1,0]])

    # defining blank image to output the rotated image on
    output=np.zeros((width,height,3))

    center_height = round(((height+1)/2)-1)
    center_width = round(((width+1)/2)-1)

    for i in range(height):
        for j in range(width):

            # defining a matrix containing co-ordinates of pixels with respect to original image
            xy = np.array([[width-1-j-center_width],[height-1-i-center_height]])

            # using dot product to calculate the new co-ordinates for the rotated image
            rotate_mat = np.dot(rotation_mat,xy)

            new_y=center_width-int(rotate_mat[1])
            new_x=center_height-int(rotate_mat[0])

            if (0<=new_x<height) and (0<=new_y<width) :
                output[new_y,new_x,:]=image[i,j,:]
    return np.uint8(output)
```

This function was used to rotate images 90 degrees clockwise. For the rotation process we used the rotation matrix which is $\begin{matrix} \cos\emptyset & \sin\emptyset \\ -\sin\emptyset & \cos\emptyset \end{matrix}$

Where $\emptyset$ is the angle of rotation , in our case we chose the $\emptyset$ to be $= 90°$

Therefore, the rotation matrix will be equal $\begin{matrix} 0 & -1 \\ 1 & 0 \end{matrix}$

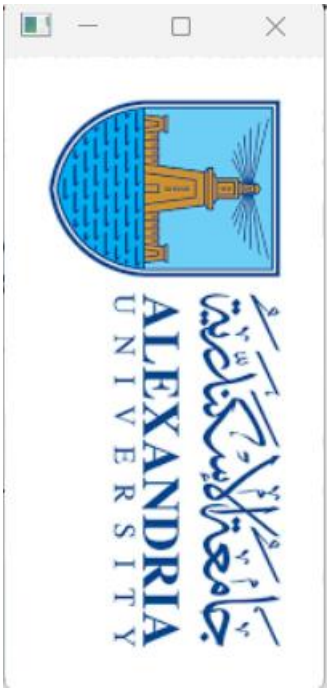
By using nested for loops in python to represent x and y axis on the image and be able to select each pixel and find its new position after rotation.

By using dot product, we multiply the rotation matrix with initial co-ordinates of the selected pixel to output the new co-ordinates.

Then we add the new co-ordinates to a blank image then we return it to the main loop to be showed.

Original image:

Rotated image:



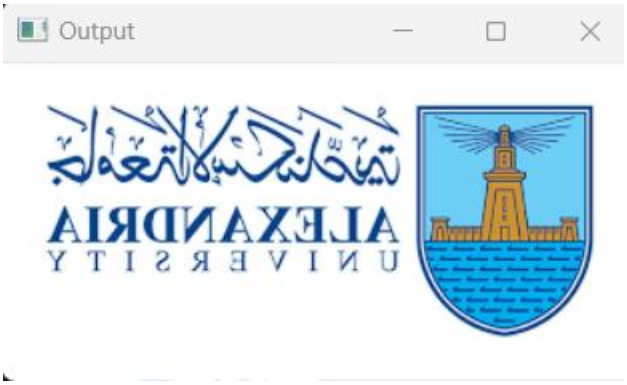
- Second function made was the flip function,

```python
def flip(image):
    height=image.shape[0]
    width=image.shape[1]

   # defining blank image to output the flipped image on
    output=np.zeros((height,width,3))

    for i in range(height):
        for j in range(width):

            # flipping pixels of the image line by line
            output[i,width-j-1,:]=image[i,j,:]

    return np.uint8(output)
```

This function was used to reflect the image around its vertical axis.

Where the function accesses each pixel and flips it on a new blank image until all pixels are flipped then the output image is returned to the main loop.

Flipped image:



- Next function is the scale function,

```
def scale(image,scale_factor) :

    height=image.shape[0]
    width=image.shape[1]

    # Compute the new dimensions of the scaled image
    new_width = round(width * scale_factor)
    new_height = round(height * scale_factor)

    # define a blank image to output the rotated image on
    output = np.zeros((new_height, new_width, 3))

    # Sample the original image and resize the subset to the desired size
    for i in range(new_height):
        for j in range(new_width):
            new_x = round(i / scale_factor)
            new_y = round(j / scale_factor)
            if (0<=new_x<height) and (0<=new_y<width) :
                output[i,j,:]=image[new_x,new_y,:]
    return np.uint8(output)
```

The scaling function allows you to upscale or downscale an image, Where the function uses nearest-neighbor interpolation to scale the image which means that every output pixel is replaced by its nearest pixel in the input image.

Scaled image:

Scale factor: 1.6

Scale factor: 0.8

- Last functions are the shear functions:

  Firstly, For the **ShearX** function:

```python
def shearX(image):

    height=image.shape[0]
    width=image.shape[1]

    # defining shearing angle
    angle=math.radians(20)

    # intializing shearing matrix
    shearingMatrix = np.array([[1, np.tan(angle), 0],[0, 1, 0],[0, 0, 1]])

    # defining blank image to output the sheared image on
    output=np.zeros((int(height),int(1.2*width),3))

    for i in range(height):
        for j in range(width):

            # defining a matrix containing old co-ordinates
            xy = np.array([[j],[i],[1]])

            # using dot product to calculate the new co-ordinates for the sheared image
            shear_mat = np.dot(shearingMatrix,xy)
            new_y=int(shear_mat[1])
            new_x=int(shear_mat[0])

            # checking to prevent any errors in the processing
            if (0<=new_x<3*height) and (0<=new_y<width) :
                output[new_y,new_x,:]=image[i,j,:]

    return np.uint8(output)
```

We are going to extract the height and width of the input image using Numpy's shape function, then define the shearing angle in degrees using the math.radians function to convert the angle to radians.

Then we will create a 3x3 shearing matrix that defines the transformation to be applied to the image,where the first row of the matrix defines the transformation in the X-axis direction, the second row defines the transformation in the Y-axis direction and the third row is a placeholder since we are not applying any transformation in the Z-axis direction.

Then create a blank output image with dimensions equal to the input image's height and 1.2 times the input image's width. The extra 0.2 times width is added to prevent any loss of data while shearing the image in the X-Axis.

After that we are then going to iterate over each pixel in the input image using two nested for loops, For each pixel it creates a matrix XY containing the old coordinates of the pixels in the

input image, then apply the shearing matrix to the old coordinates using Numpy's dot function to obtain the new coordinates of the pixel in the sheared image.

Since the output image is a 2D array, we need to convert the new x and y coordinates to integer values to access the corresponding pixel in the output image array. The int function is used to truncate the decimal part of the new coordinates and convert them to integers.

Then check if the new coordinates are within the bounds of the output image before assigning the pixel value from the input image to the corresponding pixel in the output image.

Finally, we will convert the output image to an 8-bit unsigned integer format using Numpy's uint8 function and return the sheared image.

Original Image:

Sheared image in X-axis:

Increasing the shear angle:

Secondly, For the **ShearY** function:

```
def shearY(image):

    height=image.shape[0]
    width=image.shape[1]

    # defining shearing angle
    angle = math.radians(20)

    # intializing shearing matrix
    shearingMatrix = np.array([[1, 0, 0],[-np.tan(angle), 1, 0],[0, 0, 1]])

    # defining blank image to output the sheared image on
    output=np.zeros((int(1.7*height),int(width),3))

    for i in range(height):
        for j in range(width):

            # defining a matrix containing old co-ordinates
            xy = np.array([[j],[i],[1]])

            # using dot product to calculate the new co-ordinates for the sheared image
            shear_mat = np.dot(shearingMatrix,xy)

            # adjusting the new co-ordinates since the y component is shifted
            new_y=int(shear_mat[1]) + int(0.7*height)
            new_x=int(shear_mat[0])

            # checking to prevent any errors in the processing
            if (0<=new_x<3*height) and (0<=new_y<width) :
                output[new_y,new_x,:]=image[i,j,:]

    return np.uint8(output)
```

This function will differ from that of the ShearX function in certain lines:

1-In the Shearing matrix array:

- The first row of the matrix [[1, 0, 0]] has a value of 1 in the first column, which means that there is no transformation in the X-axis direction The second and third columns have a value of 0, which means that there is no transformation in the Y-axis and Z-axis directions.
- The second row of the matrix [[-np.tan(angle), 1, 0]] defines the transformation in the Y-axis direction. The first column has a value of -tan(angle), which means that the transformation will shift the pixels along the Y-axis by an amount proportional to the tangent of the angle. The second column has a value of 1, which means that there is no transformation in the X-axis direction. The third column has a value of 0, which means that there is no transformation in the Z-axis direction.

- The third row of the matrix [[0, 0, 1]] is a placeholder since we are not applying any transformation in the Z-axis direction.

2- In the output image part:

The dimensions are defined as (int(1.7*height), int(width), 3), The first dimension is calculated by multiplying the input image's height by 1.7, which adds an extra 70% of the input image's height to the output image. This is done to prevent any loss of data while shearing the image, as shearing can cause the image to expand or contract along the vertical axis and the second dimension of the output image is equal to the width of the input image, which means that the output image will have the same width as the input image.

3-New Y value:

To accommodate this shift, the new_y value calculated for each pixel in the loop is adjusted by adding int(0.7*height) to the y-coordinate. The value int(0.7*height) is used because the output image's height is increased by 70% of the input image's height, as defined in the output array initialization.

Original Image:

Sheared image in Y-axis:

Main Loop :

```
path = 'download.png'
img = cv2.imread(path)
showed_Image = img.copy()
scale_factor = 1
while True :
    key = cv2.waitKey(1)

    # rotate image
    if key ==ord('F') or key ==ord('f'):
        showed_Image = rotate(showed_Image)

    # reflect image
    if key ==ord('e') or key ==ord('E'):
        showed_Image = flip(showed_Image)

    # Show original image
    if key ==ord('s') or key ==ord('S'):
        showed_Image = img.copy()
        scale_factor = 1

    # Scale image
    if key ==ord('H') or key ==ord('h'):
        if scale_factor < 4 :
            showed_Image = img.copy()
            scale_factor += 0.2
            showed_Image = scale(showed_Image,scale_factor)
    if key ==ord('L') or key ==ord('l'):
        if scale_factor > 0.6 :
            showed_Image = img.copy()
            scale_factor -= 0.2
            showed_Image = scale(showed_Image,scale_factor)

    # shear the image in x-axis Direction
    if key ==ord('K') or key ==ord('k'):
        showed_Image = shearX(showed_Image)
```

```
    # shear the image in y-axis Direction
    if key ==ord('Y') or key ==ord('y'):
        showed_Image = shearY(showed_Image)

    # break the loop
    if key ==ord('q') or key ==ord('Q'):
        break
    cv2.imshow("Output",showed_Image)
```

1. Using OpenCV's imread function , We are going to read an image from the specified file path and create a copy of the image to perform the manipulations on.
2. The waitKey function waits for a specified number of milliseconds for the user to input a key to perform an operation.
3. We will then display the current image on a window using OpenCV's imshow function.
4. If the user presses the 'F' or 'f' key, the program rotates the image by calling the rotate function.
5. If the user presses the 'E' or 'e' key, the program reflects the image by calling the flip function.
6. If the user presses the 'S' or 's' key, the program shows the original image by copying the input image and resetting the scale factor to 1.
7. If the user presses the 'H' or 'h' key, the program scales the image up by a factor of 0.2 using the scale function. The current scale factor is stored in the scale_factor variable and incremented by 0.2. The program ensures that the scalefactor does not exceed 4.
8. If the user presses the 'L' or 'l' key, the program scales the image down by a factor of 0.2 using the scale function. The current scale factor is stored in the scale_factor variable and decrements it by 0.2. The program ensures that the scale factor does not go below 0.6.
9. If the user presses the 'K' or 'k' key, the program shears the image in the X-axis direction by calling the shearX function.
10. If the user presses the 'Y' or 'y' key, the program shears the image in the Y-axis direction by calling the shearY function.
11. If the user presses the 'Q' or 'q' key, the program breaks out of the loop and exits the program.
12. The imshow function displays the output image on the window after each manipulation.

Part II: Cryptography:

Third parties or organizations are prevented from obtaining sensitive information through the study and use of cryptography, which uses special procedures and techniques to ensure secure communication. Concepts like secrecy, data integrity, authentication, etc. are essential to modern cryptography.

American mathematician Lester S. Hill created and refined the Hill Cipher technique in 1929. Hill Cipher employs a variety of mathematical techniques, simulating the employment of numerous key techniques in traditional encryption.

Hill Cipher uses a polygraphy substitution Cipher, which ensures consistent replacement over many layers of blocks, under the guise of classical cryptography. The Hill Cipher may easily combine digraphs (two-letter blocks), trigraphs (threeletter blocks), or any other multiple-sized blocks to create a consistent Cipher thanks to this polygraphy substitution Cipher.

The Hill Cipher is based on modulo arithmetic, matrix multiplication, and complex generic matrix operations (such as matrix inverses). Evidently, compared to other Ciphers, it is more mathematical.

Encryption:

Encryption in Hill Cipher involves using a key matrix to transform blocks of plaintext letters into blocks of ciphertext letters. The key matrix is typically generated randomly, and its size is determined by the number of letters in each block of plaintext.

Here are the steps involved in encrypting a message using Hill Cipher:

1. Choose a key matrix: The key matrix is a square matrix with a size determined by the number of letters in each block of plaintext. For example, if we are encrypting blocks of two letters, the key matrix will be a 2x2 matrix. The key matrix is typically chosen randomly, subject to the constraint that it must be invertible modulo a chosen modulus.

2. Divide the plaintext into blocks: The plaintext message is divided into blocks of the same size as the key matrix. Each block is represented as a column vector, where the entries of the vector correspond to the numerical values of the letters in the block. For example, if we are encrypting blocks of two letters, the first block "HI" might be represented as the column vector [7, 8], where "H" has numerical value 7 and "I" has numerical value 8 (assuming A=0, B=1, C=2, etc.).

3. Multiply each block by the key matrix: Each block of plaintext is multiplied by the key matrix modulo the chosen modulus. This is done by multiplying the key matrix by the column vector representing the block of plaintext. The resulting column vector represents the corresponding block of ciphertext.

4. Convert the ciphertext blocks back into letters: Each column vector representing a block of ciphertext is converted back into a block of letters by taking the numerical values modulo the chosen modulus and converting them back into their corresponding letters.

Decryption:

Decryption in Hill Cipher involves using the inverse of the key matrix to transform blocks of ciphertext letters into blocks of plaintext letters. The inverse of the key matrix is calculated modulo the chosen modulus, and it is used in a similar way to the key matrix in the encryption process.

Here are the steps involved in decrypting a message using Hill Cipher:
1. Calculate the inverse of the key matrix: The inverse of the key matrix is calculated modulo the chosen modulus. This is necessary for the decryption process, as it allows us to reverse the encryption process and recover the original plaintext. If the key matrix is not an invertible modulus, the decryption process cannot be performed.
2. Divide the ciphertext into blocks: The ciphertext message is divided into blocks of the same size as the key matrix. Each block is represented as a column vector, where the entries of the vector correspond to the numerical values of the letters in the block.

3. Multiply each block by the inverse of the key matrix: Each block of ciphertext is multiplied by the inverse of the key matrix modulo the chosen modulus. This is done by multiplying the inverse of the key matrix by the column vector representing the block of ciphertext. The resulting column vector represents the corresponding block of plaintext.

4. Convert the plaintext blocks back into letters: Each column vector representing a block of plaintext is converted back into a block of letters by taking the numerical values modulo the chosen modulus and converting them back into their corresponding letters. Modular Arithmetic: Because it enables us to work with a finite set of integers, which is required for many cryptographic techniques, modular arithmetic is utilized in cryptography. Modular arithmetic is employed in the Hill Cipher to guarantee that the integers produced by the encryption and decryption processes fall within a certain range, which is set by the modulus. This guarantees that the ciphertext and plaintext letters stay inside the same set and allows the Cipher to work on a finite set of integers that can be expressed using matrix algebra.