Linear Algebra

UE23CS241B

4th Semester, Academic Year 2023

Date: 08/03/2025

Name: Pranav Hemanth	SRN: PES1UG23CS433	Section: G

```
# Project 1: Basic operations with matrices in Python import numpy as np
```

```
A = np.array([[1, 2, -10, 4], [3, 4, 5, -6], [3, 3, -2, 5]]) B = np.array([3, 3, 4, 2])
print("A:\n", A) print("B:\n", B)
```

```
A:
  [[  1   2 -10   4]  [  3   4   5  -6]
  [  3   3  -2   5]] B:
  [3 3 4 2]
```

```
#function to determine
length def length(matrix):
return max(matrix.shape)

# Length of A and B
lengthA = length(A)
lengthB = length(B)
print("lengthA:", lengthA)
print("lengthB:", lengthB)
```

```
lengthA: 4 lengthB: 4
```

```
# Add B as the fourth row of A and create the new matrix C C = np.vstack((A, B)) print("C:\n", C)
```

```
C:
  [[  1   2 -10   4]  [  3   4   5  -6]  [  3   3  -2   5]  [  3   3   4   2]]
```

```
# Create D from rows 2, 3, 4 and columns 3, 4 of C D = C[1:4, 2:4] print("D:\n", D)
```

```
D:
  [[ 5 -6]  [-2  5]  [ 4  2]]
```

```
# Transpose D to create E

E = D.T print("E:\n", E)
```

```
E:
  [[ 5 -2  4]  [-6  5  2]]
```

```
# Check the size of E m, n = E.shape print("m:", m) print("n:",
n)
```

```
m: 2 n: 3
```

```
# Create equally spaced vectors using arange and linspace
EqualSpaced = np.arange(0, 2 * np.pi, np.pi / 10) EqualSpaced1 = np.linspace(0, 2 * np.pi, 21) print("EqualSpaced:\n",
EqualSpaced) print("EqualSpaced1:\n", EqualSpaced1)
```

```
EqualSpaced:
  [0.         0.31415927 0.62831853 0.9424778  1.25663706 1.57079633  1.88495559 2.19911486 2.51327412 2.82743339
 3.14159265 3.45575192  3.76991118 4.08407045 4.39822972 4.71238898 5.02654825 5.34070751
  5.65486678 5.96902604] EqualSpaced1:
  [0.         0.31415927 0.62831853 0.9424778  1.25663706 1.57079633  1.88495559 2.19911486 2.51327412 2.82743339
 3.14159265  3.45575192   3.76991118 4.08407045  4.39822972  4.71238898  5.02654825  5.34070751   5.65486678  5.96902604
 6.28318531]
```

```
# Find the maximum and minimum in each column of A maxcolA = np.max(A, axis=0) mincolA =
np.min(A, axis=0) print("maxcolA:", maxcolA) print("mincolA:", mincolA)
```

```
maxcolA: [3 4 5 5] mincolA: [  1   2 -10  -6]
```

```
# Find the maximum and minimum in each row of A
maxrowA = np.max(A, axis=1) minrowA = np.min(A,
axis=1)
# Find the maximum and minimum elements in the entire matrix A
maxA = np.max(A) minA = np.min(A) print("maxrowA:", maxrowA)
print("minrowA:", minrowA) print("maxA:", maxA) print("minA:",
minA)
```

```
maxrowA: [4 5 5] minrowA: [-10  -6  -2] maxA: 5 minA: -10
```

```
# Calculate mean and sum in each column and row of A
meancolA = np.mean(A, axis=0) meanrowA = np.mean(A,
axis=1) sumcolA = np.sum(A, axis=0) sumrowA =
np.sum(A, axis=1)
# Calculate mean and sum of all elements in A
meanA = np.mean(A) sumA = np.sum(A)
print("meancolA:", meancolA)
print("meanrowA:", meanrowA)
print("sumcolA:", sumcolA) print("sumrowA:",
sumrowA) print("meanA:", meanA)
print("sumA:", sumA)
```

```
meancolA: [ 2.33333333  3.         -2.33333333  1.        ] meanrowA: [-0.75  1.5   2.25] sumcolA: [
  7  9 -7  3] sumrowA: [-3  6  9] meanA: 1.0 sumA: 12
```

```
# Create matrices F and G with random integers from -4 to 4
F = np.random.randint(-4, 5, (5, 3))
G = np.random.randint(-4, 5, (5, 3))
print("F:\n", F) print("G:\n", G)
```

```
F:
  [[-2 -4  2]  [ 2 -4 -2]  [ 4  2  3]  [ 0  3 -1]
  [-4  4  1]] G:
  [[ 1  0  0]  [ 0  2  0]  [ 1  2 -4]  [-3 -1 -4]  [ 0  4 -4]]
```

```
# Perform scalar multiplication, addition, subtraction, and element-wise multiplication on F and G
ScMultF = 0.4 * F
SumFG = F + G
DiffFG = F - G
```

```
ElProdFG = F * G print("ScMultF:\n", ScMultF) print("SumFG:\n", SumFG)
print("DiffFG:\n", DiffFG) print("ElProdFG:\n", ElProdFG)
```

```
ScMultF:
  [[-0.8 -1.6  0.8]  [ 0.8 -1.6 -0.8]  [ 1.6  0.8  1.2]  [ 0.   1.2 -0.4]
  [-1.6  1.6  0.4]] SumFG:
  [[-1 -4  2]  [ 2 -2 -2]  [ 5  4 -1]  [-3  2 -5]
  [-4  8 -3]] DiffFG:
  [[-3 -4  2]  [ 2 -6 -2]  [ 3  0  7]  [ 3  4  3]
  [-4  0  5]] ElProdFG:
  [[ -2   0   0]  [  0  -8   0]  [  4   4 -12]  [  0  -3   4]  [  0  16  -4]]
```

```
# Check the size of F and A sizeF = F.shape sizeA = A.shape
print("sizeF:", sizeF) print("sizeA:", sizeA)
```

```
sizeF: (5, 3) sizeA: (3, 4)
```

```
# Perform matrix multiplication of F and A if dimensions are compatible if sizeF[1] == sizeA[0]:     H = F @ A
    print("H:\n", H) else:
    print("Cannot multiply F and A due to incompatible dimensions.")
```

```
H:
  [[ -8 -14  -4  26]  [-16 -18 -36  22]  [ 19  25 -36  19]  [  6   9  17 -23]  [ 11  11  58 -35]]
```

```
# Generate the identity matrix with 3 rows and 3 columns eye33 = np.eye(3) print("eye33:\n",
eye33)
```

```
eye33:
  [[1. 0. 0.]  [0. 1. 0.]  [0. 0. 1.]]
```

```
# Generate matrices of zeros with size 5x3 and ones with size 4x2 zeros53 = np.zeros((5, 3)) ones42 =
np.ones((4, 2)) print("zeros53:\n", zeros53) print("ones42:\n", ones42)
```

```
zeros53:  [[0. 0. 0.]  [0. 0. 0.]  [0. 0. 0.]  [0. 0.
 0.]
 [0.         0. 0.]]ones42:  [[1. 1.]
 [1.         1.]
  [1. 1.]  [1. 1.]]
```

```
# Generate a diagonal matrix S with the diagonal elements 1, 2, 7 S = np.diag([1, 2, 7]) print("S:\n", S)
```

```
S:
  [[1 0 0]  [0 2 0]  [0 0 7]]
```

```
# Extract the diagonal elements from a random 6x6 matrix R = np.random.rand(6, 6) diagR =
np.diag(R) print("R:", R) print("diagR:", diagR)
```

```
R: [[0.40151202 0.62282625 0.35680583 0.01422376 0.05999239 0.32926742]  [0.10496773 0.99996929 0.95455657 0.29959249
0.6798288   0.15754804]   [0.39796597  0.07202853  0.43580223  0.25549625  0.77360268  0.52160439]   [0.01999679  0.75971532
0.98973635  0.8658178    0.71202421  0.90482223]   [0.79462579  0.37573559  0.46681164  0.18296374  0.87420623  0.53091008]
[0.05852918 0.53290773 0.23654192 0.14048134 0.69536968 0.92474325]] diagR: [0.40151202 0.99996929 0.43580223 0.8658178
0.87420623 0.92474325]
```

```
# Create a sparse diagonal matrix and convert to dense from scipy.sparse import
diags diag121 = diags([-np.ones(10), 2*np.ones(10), -np.ones(10)], [-1, 0, 1],
shape=(10,
10)).todense()
print("diag121:\n", diag121)
```

```
diag121:
  [[ 2. -1.  0.  0.  0.  0.  0.  0.  0.  0.]  [-1.  2. -1.  0.  0.  0.  0.  0.  0.  0.]  [ 0. -1.  2. -1.  0.  0.  0.
 0.  0.  0.]  [ 0.  0. -1.  2. -1.  0.  0.  0.  0.  0.]  [ 0.  0.  0. -1.  2. -1.  0.  0.  0.  0.]  [ 0.  0.  0.  0.
 -1.  2. -1.  0.  0.  0.]  [ 0.  0.  0.  0.  0. -1.  2. -1.  0.  0.]  [ 0.  0.  0.  0.  0.  0. -1.  2. -1.  0.]  [ 0.
 0.  0.  0.  0.  0.  0. -1.  2. -1.]  [ 0.  0.  0.  0.  0.  0.  0.  0. -1.  2.]]
```

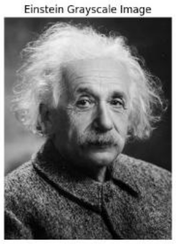
```
# Project 2: Matrix operations and image manipulation

import imageio.v3 as iio  # imageio.v3 is the latest version for imread import matplotlib.pyplot as plt import numpy as np from PIL import Image
```

```
# Load a grayscale jpg file and represent the data as a matrix
ImJPG=iio.imread("Albert_Einstein_Head.jpg");
''' plt.imshow(ImJPG, cmap="gray")  # Ensure grayscale
display plt.show()
'''
```

```
'\nplt.imshow(ImJPG, cmap="gray")  # Ensure grayscale display\nplt.show()\n'
```

```
# Get the dimensions of the image m, n = ImJPG.shape
# Print the dimensions print(f'The dimensions of the
image are {m} x {n}')
# Optional: Visualize the image
plt.imshow(ImJPG, cmap='gray')
plt.title('Einstein Grayscale Image')
plt.axis('off') plt.show()
```

```
The dimensions of the image are 1200 x 900
```


Einstein Grayscale Image

```
# Check if the array is of integer type isInt =
np.issubdtype(ImJPG.dtype, np.integer)
# Print the result print(f'Is ImJPG of
integer type? {isInt}')
# Optional: Visualize the image '''
plt.imshow(ImJPG, cmap='gray')
plt.title('Einstein Grayscale Image')
plt.axis('off') plt.show()
'''
```

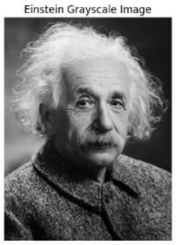
```
Is ImJPG of integer type? True
 '\nplt.imshow(ImJPG, cmap='gray')\nplt.title('Einstein Grayscale Image')\nplt.axis('off')\nplt.show()\n'
```

```
isInt = np.issubdtype(ImJPG.dtype, np.integer) # Find the range
of colors in the image maxImJPG = np.max(ImJPG) minImJPG =
np.min(ImJPG) # Print the results print(f'The dimensions of the
image are {m} x {n}') print(f'Is ImJPG of integer type?
{isInt}') print(f'The maximum pixel value in the image is
{maxImJPG}') print(f'The minimum pixel value in the image is
{minImJPG}')
# Optional: Visualize the image '''
plt.imshow(ImJPG, cmap='gray')
plt.title('Einstein Grayscale Image')
plt.axis('off') plt.show()
'''
```

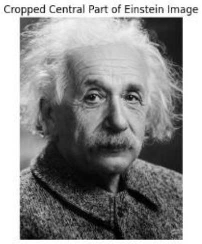
```
The dimensions of the image are 1200 x 900
 Is ImJPG of integer type? True
 The maximum pixel value in the image is 255
 The minimum pixel value in the image is 0
 '\nplt.imshow(ImJPG, cmap='gray')\nplt.title('Einstein Grayscale Image')\nplt.axis('off')\nplt.show()\n'
```

```
# Check if the array is of integer type isInt =
np.issubdtype(ImJPG.dtype, np.integer) # Find the range of
colors in the image maxImJPG = np.max(ImJPG) minImJPG =
np.min(ImJPG) # Print the results print(f'The dimensions of the
image are {m} x {n}') print(f'Is ImJPG of integer type?
{isInt}') print(f'The maximum pixel value in the image is
{maxImJPG}') print(f'The minimum pixel value in the image is
{minImJPG}')
# Display the image plt.imshow(ImJPG,
cmap='gray') plt.title('Einstein
Grayscale Image') plt.axis('off')
plt.show()
```

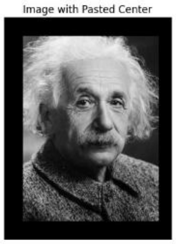
```
The dimensions of the image are 1200 x 900
Is ImJPG of integer type? True
The maximum pixel value in the image is 255
The minimum pixel value in the image is 0
```



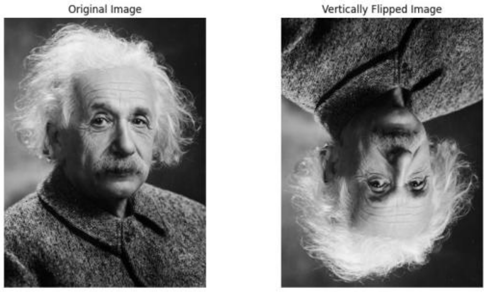
```
# Get the dimensions of the image m,
n = ImJPG.shape
# Crop the central part of the image
ImJPG_center = ImJPG[100:m-100, 100:n-70]
# Display the cropped image plt.figure()
plt.imshow(ImJPG_center, cmap='gray')
plt.title('Cropped Central Part of Einstein Image')
plt.axis('off') plt.show()
```



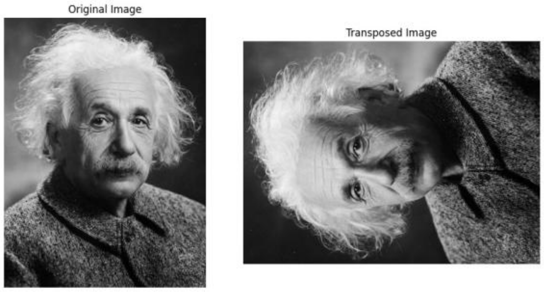
```
# Get the dimensions of the image m, n = ImJPG.shape
# Crop the central part of the image
ImJPG_center = ImJPG[100:m-100, 100:n-70]
# Create a zero matrix of type uint8 with the same dimensions as the original image
ImJPG_border = np.zeros((m, n), dtype=np.uint8)
# Paste the cropped image into the zero matrix
ImJPG_border[100:m-100, 100:n-70] = ImJPG_center
# Display the resulting image
plt.figure() plt.imshow(ImJPG_border,
cmap='gray') plt.title('Image with
Pasted Center') plt.axis('off')
plt.show()
```



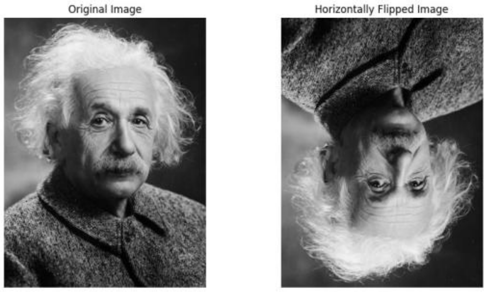
```
# Perform vertical flipping by reversing the rows of the matrix
ImJPG_vertflip = np.flipud(ImJPG)
# Display the original and flipped images side by side for comparison plt.figure(figsize=(10, 5)) plt.subplot(1, 2, 1) plt.imshow(ImJPG, cmap='gray') plt.title('Original Image') plt.axis('off') plt.subplot(1, 2, 2) plt.imshow(ImJPG_vertflip, cmap='gray') plt.title('Vertically Flipped Image') plt.axis('off')
plt.tight_layout()
plt.show()
```



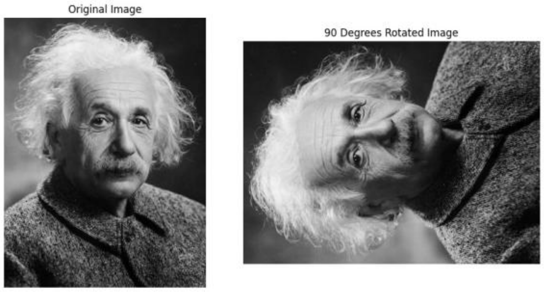
```
ImJPG = np.array(ImJPG)
# Transpose the image matrix
ImJPG_transpose = ImJPG.T
# Display the original and transposed images side by side for comparison plt.figure(figsize=(10, 5)) plt.subplot(1, 2, 1) plt.imshow(ImJPG, cmap='gray') plt.title('Original Image') plt.axis('off') plt.subplot(1, 2, 2) plt.imshow(ImJPG_transpose, cmap='gray') plt.title('Transposed Image') plt.axis('off') plt.tight_layout()
 plt.show()
```



```
# Transpose the image matrix
ImJPG_transpose = ImJPG.T
# Flip the transposed image horizontally (along the vertical axis)
ImJPG_horflip = np.fliplr(ImJPG_transpose)
# Transpose the flipped image back to its original orientation
ImJPG_horflip = ImJPG_horflip.T
# Display the original and horizontally flipped images side by side for comparison plt.figure(figsize=(10, 5)) plt.subplot(1, 2, 1) plt.imshow(ImJPG, cmap='gray') plt.title('Original Image') plt.axis('off') plt.subplot(1, 2, 2) plt.imshow(ImJPG_horflip, cmap='gray') plt.title('Horizontally Flipped Image') plt.axis('off')
 plt.tight_layout()
 plt.show()
```

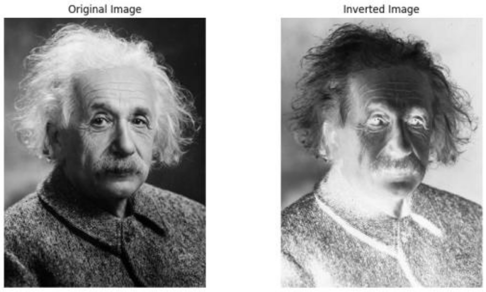


```
ImJPG = np.array(ImJPG)
# Rotate the image matrix by 90 degrees counterclockwise
ImJPG90 = np.rot90(ImJPG)
# Display the original and rotated images side by side for comparison plt.figure(figsize=(10, 5)) plt.subplot(1, 2, 1) plt.imshow(ImJPG, cmap='gray')
 plt.title('Original Image') plt.axis('off') plt.subplot(1, 2, 2) plt.imshow(ImJPG90, cmap='gray') plt.title('90 Degrees Rotated Image') plt.axis('off')
 plt.tight_layout()
 plt.show()
```

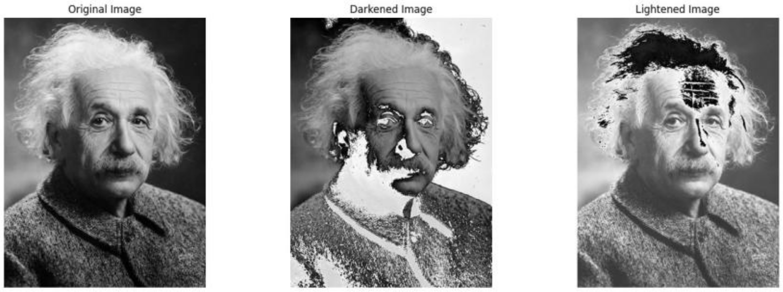


```
# Perform color inversion
ImJPG_inv = 255 - ImJPG
# Display the original and inverted images side by side for comparison plt.figure(figsize=(10, 5)) plt.subplot(1, 2, 1) plt.imshow(ImJPG, cmap='gray') plt.title('Original Image') plt.axis('off') plt.subplot(1, 2, 2) plt.imshow(ImJPG_inv, cmap='gray') plt.title('Inverted Image') plt.axis('off') plt.tight_layout()
```

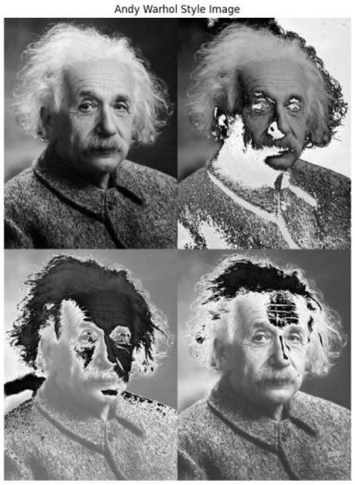
```
plt.show()
```



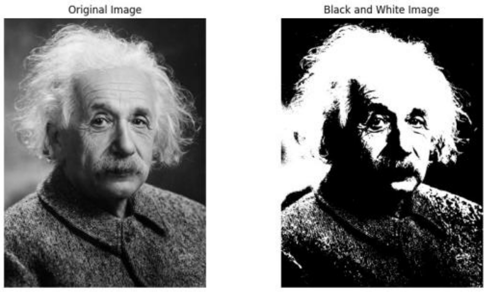
```
# Darken the image by subtracting a constant value
ImJPG_dark = ImJPG - 50
ImJPG_dark[ImJPG_dark < 0] = 0 # Ensure no negative values
# Lighten the image by adding a constant value
ImJPG_light = ImJPG + 50
ImJPG_light[ImJPG_light > 255] = 255 # Ensure values do not exceed 255 # Display the original, darkened, and lightened images side by side for comparison plt.figure(figsize=(15, 5)) plt.subplot(1, 3, 1) plt.imshow(ImJPG, cmap='gray') plt.title('Original Image') plt.axis('off') plt.subplot(1, 3, 2) plt.imshow(ImJPG_dark,
cmap='gray') plt.title('Darkened Image') plt.axis('off') plt.subplot(1, 3, 3) plt.imshow(ImJPG_light, cmap='gray') plt.title('Lightened Image') plt.axis('off') plt.tight_layout()
 plt.show()
```



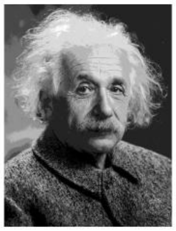
```
# Darken the image by subtracting 50
ImJPG_dark = ImJPG - 50
ImJPG_dark[ImJPG_dark < 0] = 0 # Ensure no negative values
# Lighten the image by adding 100
ImJPG_light_100 = ImJPG + 100
ImJPG_light_100[ImJPG_light_100 > 255] = 255 # Ensure values do not exceed 255
# Lighten the image by adding 50
ImJPG_light_50 = ImJPG + 50
ImJPG_light_50[ImJPG_light_50 > 255] = 255 # Ensure values do not exceed 255
# Arrange the images in a 2x2 matrix top_row =
np.concatenate((ImJPG, ImJPG_dark), axis=1)
bottom_row = np.concatenate((ImJPG_light_100, ImJPG_light_50), axis=1)
ImJPG_Warhol = np.concatenate((top_row, bottom_row), axis=0) #
Display the resulting block matrix as a single image
plt.figure(figsize=(10, 10)) plt.imshow(ImJPG_Warhol,
cmap='gray') plt.title('Andy Warhol Style Image')
plt.axis('off') plt.show()
```



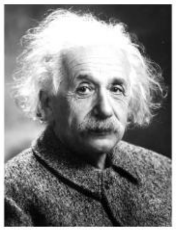
```
image_path = "Albert_Einstein_Head.jpg"
im = Image.open(image_path).convert('L') # Convert to grayscale
ImJPG = np.array(im)
# Naive conversion to black and white
ImJPG_bw = np.uint8(255 * np.floor(ImJPG / 128)) #
Display the original and black and white images
plt.figure(figsize=(10, 5)) plt.subplot(1, 2, 1)
plt.imshow(ImJPG, cmap='gray') plt.title('Original
Image') plt.axis('off') plt.subplot(1, 2, 2)
plt.imshow(ImJPG_bw, cmap='gray') plt.title('Black
and White Image') plt.axis('off')
plt.tight_layout() plt.show()
```



```
# Reduce the number of shades from 256 to 8
# Step 1: Normalize the pixel values to the range [0, 1] imjpg_normalized =
ImJPG / 255.0
# Step 2: Scale the pixel values to the range [0, 7] and round them imjpg_reduced =
np.round(imjpg_normalized * 7)
# Step 3: Scale back to the range [0, 255] and convert to uint8 imjpg8_array =
np.uint8(imjpg_reduced * (255 / 7))
# Convert the numpy array back to an image
imjpg8 = Image.fromarray(imjpg8_array) #
Display the image in a separate window
plt.imshow(imjpg8, cmap='gray')
plt.axis('off') # Hide axis plt.show()
```

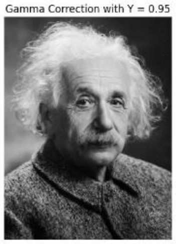


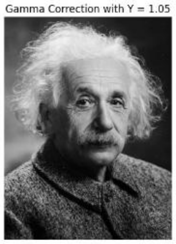
```
# Increase the contrast by multiplying with a constant (e.g., 1.25) contrast_factor =
1.25 imjpg_high_contrast_array = np.clip(ImJPG * contrast_factor, 0,
255).astype(np.uint8)
# Convert the numpy array back to an image imjpg_high_contrast =
Image.fromarray(imjpg_high_contrast_array)
# Display the high contrast image
plt.imshow(imjpg_high_contrast, cmap='gray')
plt.axis('off') # Hide axis plt.show()
```



```
# Perform gamma correction with gamma = 0.95 gamma_05 = 0.95
imjpg_gamma_05_array = np.clip((ImJPG / 255.0) ** gamma_05 * 255, 0,
255).astype(np.uint8)
# Perform gamma correction with gamma = 1.05 gamma_15 = 1.05
imjpg_gamma_15_array = np.clip((ImJPG / 255.0) ** gamma_15 * 255, 0,
255).astype(np.uint8)
# Convert the numpy arrays back to images imjpg_gamma_05
= Image.fromarray(imjpg_gamma_05_array) imjpg_gamma_15 =
Image.fromarray(imjpg_gamma_15_array)
# Display the gamma-corrected images plt.figure()
```

```
plt.imshow(imjpg_gamma_05, cmap='gray') plt.title('Gamma Correction with Y = 0.95') plt.axis('off') # Hide axis plt.show() plt.figure()
plt.imshow(imjpg_gamma_15, cmap='gray') plt.title('Gamma Correction with Y = 1.05') plt.axis('off') # Hide axis plt.show()
```





Start coding or generate with AI.

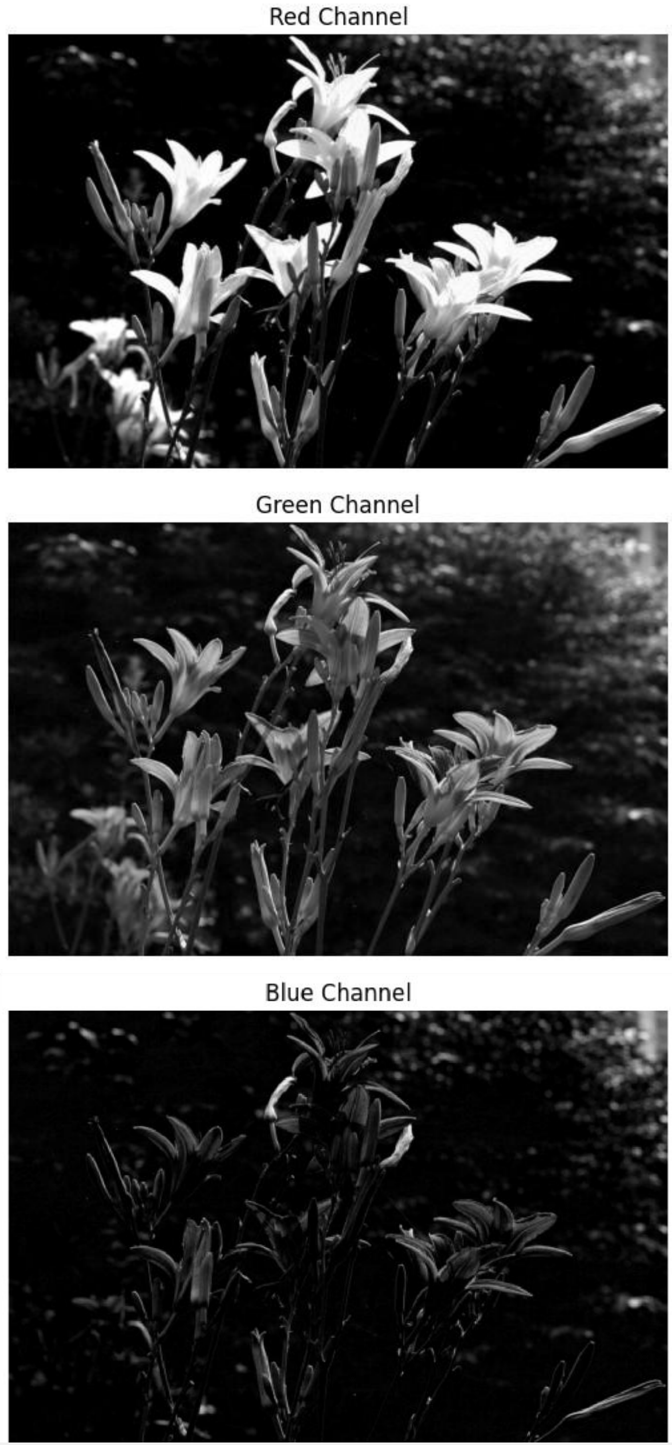
```
# Project 3: Matrix multiplication, inversion, and photo filters import matplotlib.pyplot as plt import
numpy as np
```

```
from PIL import Image
import numpy as np
import matplotlib.pyplot as plt
# Load the image
image_path = "Backlit_daylilies.jpg"
ImJPG = Image.open(image_path)
ImJPG = np.array(ImJPG)
# Display the image
plt.imshow(ImJPG)
plt.axis('off')
plt.show()
```

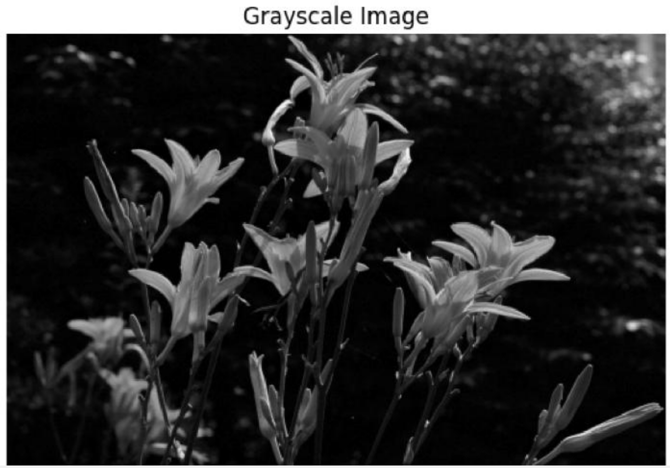
```
m, n, l = ImJPG.shape
print(f"Dimensions of the image: {m}x{n}x{l}")
```

```
Dimensions of the image: 1050x1600x3
```

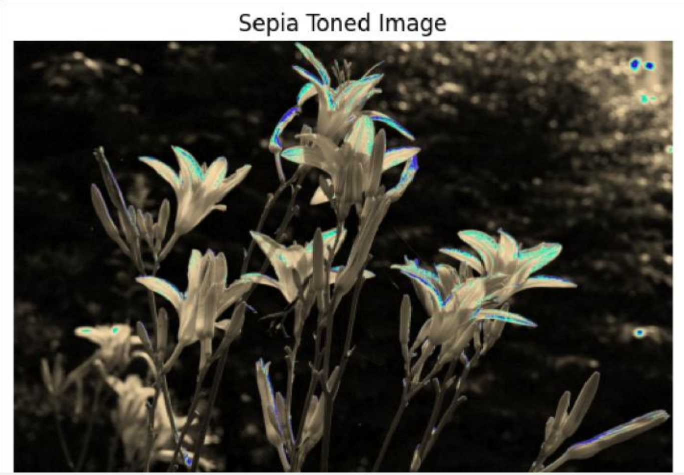
```
# Extract color channels
redChannel = ImJPG[:, :, 0]
greenChannel = ImJPG[:, :, 1]
blueChannel = ImJPG[:, :, 2] #
Display the color channels
plt.figure()
plt.imshow(redChannel,
cmap='gray') plt.title('Red
Channel') plt.axis('off')
plt.show() plt.figure()
plt.imshow(greenChannel,
cmap='gray') plt.title('Green
Channel') plt.axis('off') plt.show()
plt.figure()
plt.imshow(blueChannel,
cmap='gray') plt.title('Blue
Channel') plt.axis('off')
plt.show()
```


Red Channel
Green Channel
Blue Channel

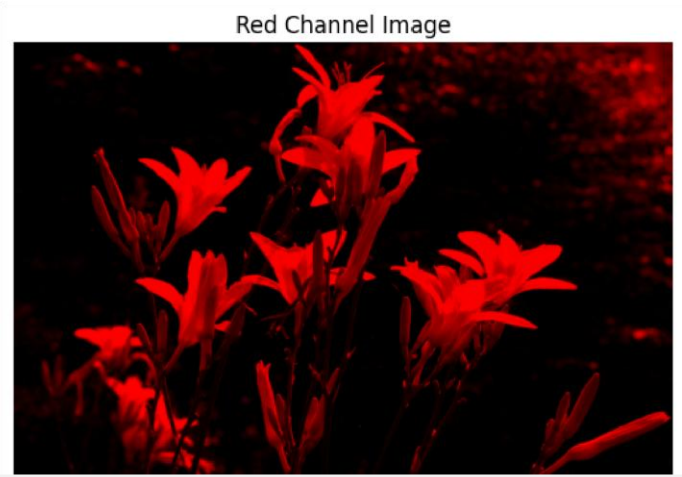
```
# Define the GrayMatrix filter
GrayMatrix = np.array([[1/3, 1/3, 1/3],
[1/3, 1/3, 1/3],
[1/3, 1/3, 1/3]])
# Initialize ImJPG_Gray with the same shape as ImJPG
ImJPG_Gray = np.zeros_like(ImJPG, dtype=np.uint8)
# Convert each pixel to
grayscale for i in range(m):
for j in range(n):
        PixelColor = ImJPG[i, j, :].reshape(3, 1) # Reshape to (3, 1) for matrix multiplication
        ImJPG_Gray[i, j, :] = np.dot(GrayMatrix, PixelColor).flatten().astype(np.uint8)
# Display the grayscale
image plt.figure()
plt.imshow(ImJPG_Gray)
plt.title('Grayscale Image')
plt.axis('off') plt.show()
```



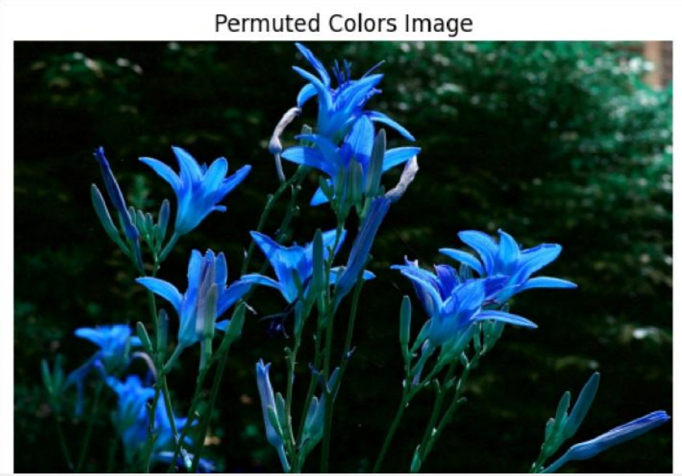
```
# Define the SepiaMatrix filter
SepiaMatrix = np.array([[0.393, 0.769, 0.189],
[0.349, 0.686, 0.168],
[0.272, 0.534, 0.131]])
# Initialize ImJPG_Sepia with the same shape as ImJPG
ImJPG_Sepia = np.zeros_like(ImJPG, dtype=np.uint8)
# Convert each pixel to sepia
tone for i in range(m):     for j
in range(n):
        PixelColor = ImJPG[i, j, :].reshape(3, 1) # Reshape to (3, 1) for matrix multiplication
        ImJPG_Sepia[i, j, :] = np.dot(SepiaMatrix, PixelColor).flatten().astype(np.uint8)
# Display the sepia toned
image plt.figure()
plt.imshow(ImJPG_Sepia)
plt.title('Sepia Toned Image')
plt.axis('off') plt.show()
```



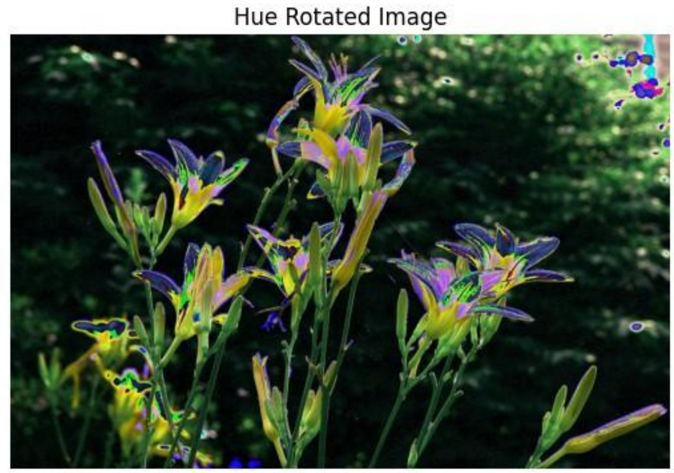
```
# Define the RedMatrix filter
RedMatrix = np.array([[1, 0, 0],
[0, 0, 0],
[0, 0, 0]])
# Initialize ImJPG_Red with the same shape as ImJPG
ImJPG_Red = np.zeros_like(ImJPG, dtype=np.uint8)
# Convert each pixel to red channel only
for i in range(m):
    for j in range(n):
        PixelColor = ImJPG[i, j, :].reshape(3, 1) # Reshape to (3, 1) for matrix multiplication
        ImJPG_Red[i, j, :] = np.dot(RedMatrix, PixelColor).flatten().astype(np.uint8)
# Display the red channel
image plt.figure()
plt.imshow(ImJPG_Red)
plt.title('Red Channel Image')
plt.axis('off') plt.show()
```

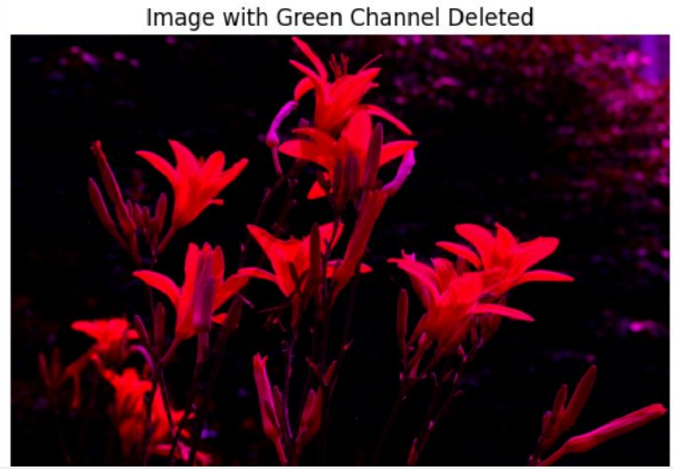


```
# Define the PermuteMatrix filter
PermuteMatrix = np.array([[0, 0, 1],
[0, 1, 0],
[1, 0, 0]])
# Initialize ImJPG_Permute with the same shape as
ImJPG ImJPG_Permute = np.zeros_like(ImJPG,
dtype=np.uint8) # Apply the PermuteMatrix to permute
the color channels for i in range(m):     for j in
range(n):
        PixelColor = ImJPG[i, j, :].reshape(3, 1) # Reshape to (3, 1) for matrix multiplication
        ImJPG_Permute[i, j, :] = np.dot(PermuteMatrix, PixelColor).flatten().astype(np.uint8)
# Display the permuted image
plt.figure()
plt.imshow(ImJPG_Permute)
plt.title('Permuted Colors
Image') plt.axis('off')
plt.show()
```

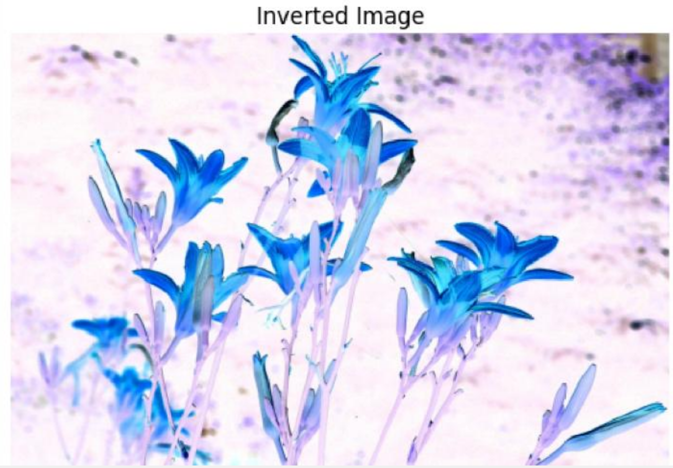


```
# Define rotation angle θ (in radians)
theta = np.radians(45) # Example angle, adjust as desired
# Define the HueRotateMatrix
HueRotateMatrix = np.array([[0.213 + np.cos(theta), 0.715 - 0.715*np.cos(theta) -
0.072*np.sin(theta), 0.072 + 0.928*np.sin(theta)],
[0.213 - 0.213*np.cos(theta) + 0.285*np.sin(theta), 0.715 + np.cos(theta),
0.072 - 0.283*np.sin(theta)],
[0.213 - 0.787*np.sin(theta), 0.715 - 0.715*np.cos(theta) +
0.928*np.sin(theta), 0.072 + np.cos(theta)]])
# Initialize ImJPG_HueRotate with the same shape as ImJPG
ImJPG_HueRotate = np.zeros_like(ImJPG, dtype=np.uint8) #
Apply the HueRotateMatrix to rotate the hue of the image
for i in range(m):     for j in range(n):
        PixelColor = ImJPG[i, j, :].reshape(3, 1) # Reshape to (3, 1) for matrix multiplication
        ImJPG_HueRotate[i, j, :] = np.dot(HueRotateMatrix,
PixelColor).flatten().astype(np.uint8)
# Display the hue rotated image
plt.figure()
plt.imshow(ImJPG_HueRotate)
plt.title('Hue Rotated Image')
plt.axis('off') plt.show()

# Define the filter matrix to delete the green channel
DeleteGreenMatrix = np.array([[1, 0, 0],
[0, 0, 0],
[0, 0, 1]])
# Initialize ImJPG_DeleteGreen with the same shape as ImJPG
ImJPG_DeleteGreen = np.zeros_like(ImJPG, dtype=np.uint8)
# Apply the DeleteGreenMatrix to remove the green
channel for i in range(m):     for j in range(n):
        PixelColor = ImJPG[i, j, :].reshape(3, 1) # Reshape to (3, 1) for matrix multiplication
        ImJPG_DeleteGreen[i, j, :] = np.dot(DeleteGreenMatrix,
PixelColor).flatten().astype(np.uint8)
# Display the image with green channel
deleted plt.figure()
plt.imshow(ImJPG_DeleteGreen)
plt.title('Image with Green Channel Deleted')
plt.axis('off') plt.show()
```



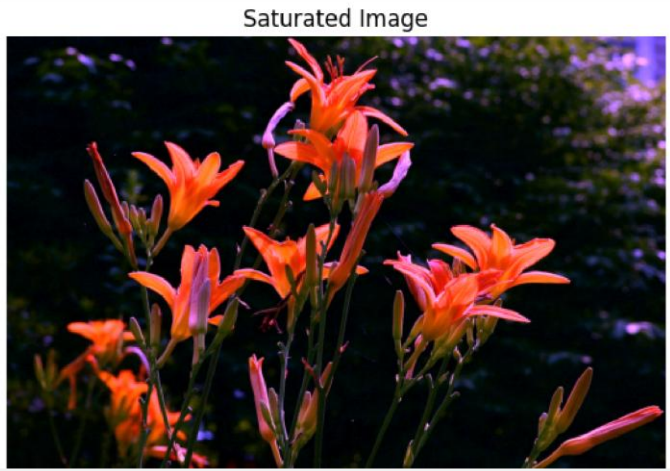


```
# Invert the colors
ImJPG_Invert = 255 - ImJPG # Display the inverted image plt.figure()
plt.imshow(ImJPG_Invert) plt.title('Inverted Image')
plt.axis('off')
plt.show()
```

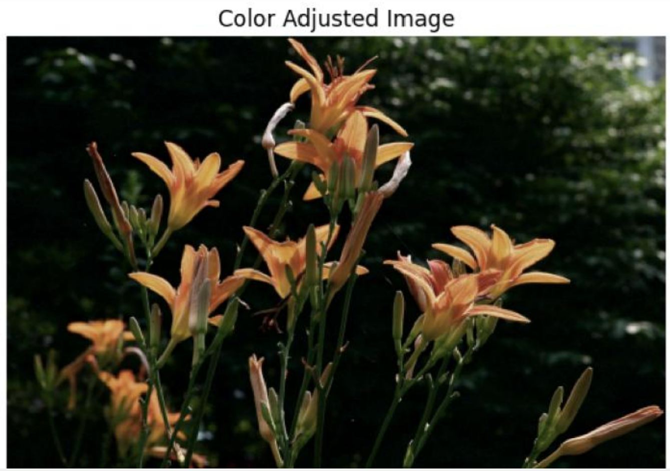


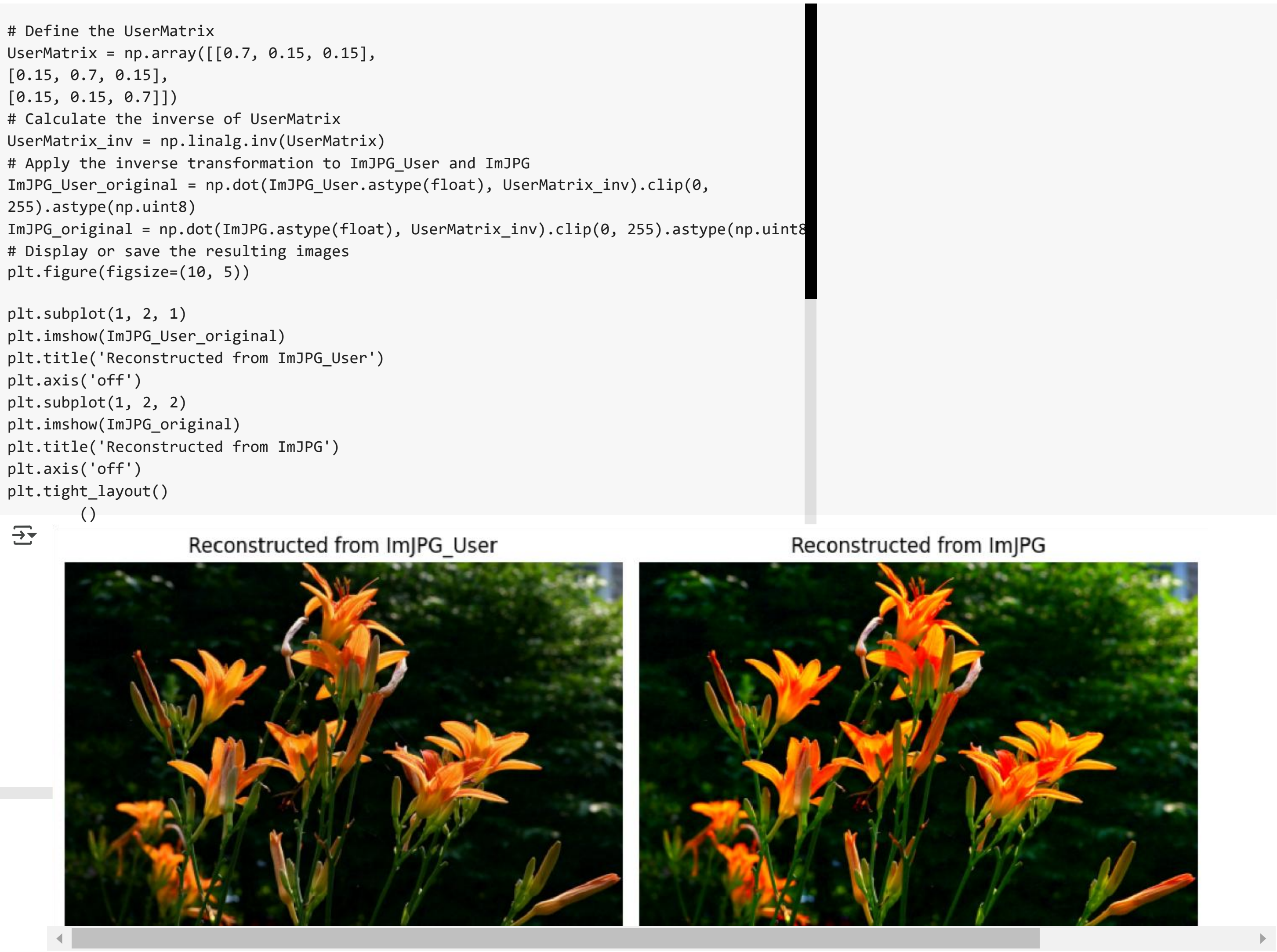
```
# Define the SaturateMatrix
SaturateMatrix = np.array([[1.2, 0, 0],
[0, 0.75, 0],
[0, 0, 2]])
# Apply the color transformation
ImJPG_Saturate = np.dot(ImJPG.astype(float), SaturateMatrix)
# Ensure values are within valid range (0-255) and convert to uint8
```

```
ImJPG_Saturate = np.clip(ImJPG_Saturate, 0, 255).astype(np.uint8)
# Display or save the resulting image
plt.figure()
plt.imshow(ImJPG_Saturate)
plt.title('Saturated
Image') plt.axis('off')
plt.show()
```



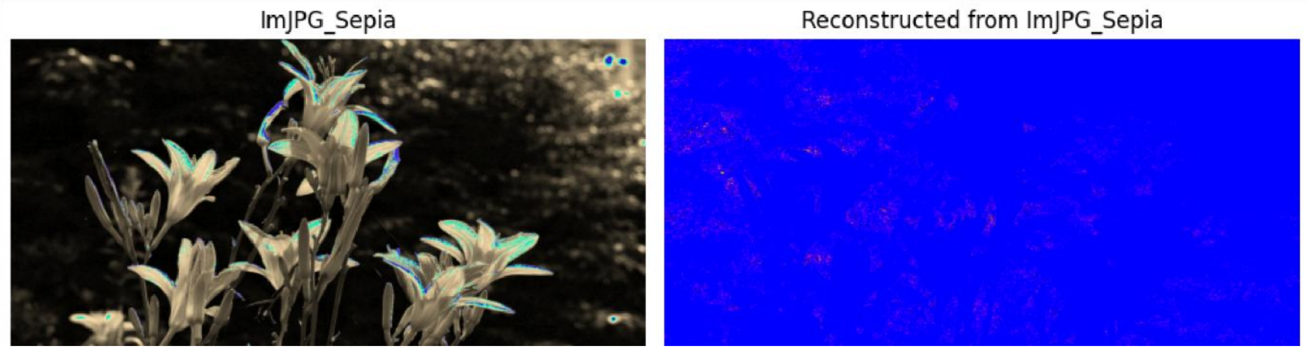
```
# Define the UserMatrix
UserMatrix = np.array([[0.7, 0.15, 0.15],
[0.15, 0.7, 0.15],
[0.15, 0.15, 0.7]])
# Apply the color adjustment transformation
ImJPG_User = np.dot(ImJPG.astype(float), UserMatrix)
# Ensure values are within valid range (0-255) and convert to uint8
ImJPG_User = np.clip(ImJPG_User, 0, 255).astype(np.uint8)
# Display or save the resulting
image plt.figure()
plt.imshow(ImJPG_User)
plt.title('Color Adjusted Image')
plt.axis('off') plt.show()
```





```
# Define the UserMatrix
UserMatrix = np.array([[0.7, 0.15, 0.15],
[0.15, 0.7, 0.15],
[0.15, 0.15, 0.7]])
# Calculate the inverse of UserMatrix
UserMatrix_inv = np.linalg.inv(UserMatrix)
# Apply the inverse transformation to ImJPG_User and ImJPG
ImJPG_User_original = np.dot(ImJPG_User.astype(float), UserMatrix_inv).clip(0,
255).astype(np.uint8)
ImJPG_original = np.dot(ImJPG.astype(float), UserMatrix_inv).clip(0, 255).astype(np.uint8
# Display or save the resulting images
plt.figure(figsize=(10, 5))

plt.subplot(1, 2, 1)
plt.imshow(ImJPG_User_original)
plt.title('Reconstructed from ImJPG_User')
plt.axis('off')
plt.subplot(1, 2, 2)
plt.imshow(ImJPG_original)
plt.title('Reconstructed from ImJPG')
plt.axis('off')
plt.tight_layout()
        ()
```

```
# Define the SepiaMatrix
SepiaMatrix = np.array([[0.393, 0.769, 0.189],
[0.349, 0.686, 0.168],
[0.272, 0.534, 0.131]])
# Calculate the inverse of SepiaMatrix
SepiaMatrix_inv = np.linalg.inv(SepiaMatrix)
# Apply the inverse transformation to ImJPG_Sepia
ImJPG_Sepia_original = np.dot(ImJPG_Sepia.astype(float), SepiaMatrix_inv).clip(0,
255).astype(np.uint8)
# Display or save the resulting images
plt.figure(figsize=(10, 5)) plt.subplot(1,
2, 1) plt.imshow(ImJPG_Sepia)
plt.title('ImJPG_Sepia') plt.axis('off')
plt.subplot(1, 2, 2)
plt.imshow(ImJPG_Sepia_original)
plt.title('Reconstructed from
ImJPG_Sepia') plt.axis('off')
plt.tight_layout() plt.show()
```



```
# Project 4: Solving linear systems in Python import numpy as np import time
```

```
# Define a function to create a magic-like matrix
def create_magic_like(n):     magic_square =
np.zeros((n, n), dtype=int)     row, col = 0, n //
2     num = 1     while num <= n*n:
        magic_square[row, col] = num         num += 1
new_row, new_col = (row - 1) % n, (col + 1) % n
if magic_square[new_row,new_col]:
            row += 1
else:
            row, col = new_row, new_col
return magic_square
# Generate a 5x5 "magic- like" matrix
A = create_magic_like(5) #
Define column vector b b =
np.array([10, 26, 42, 59, 38])
# Print matrix A and vector b
print("Matrix A:") print(A)
print("\nVector b:", b)
```

```
Matrix A:
[[17 24  1  8 15]
 [23  5  7 14 16]
 [ 4  6 13 20 22]
[10 12 19 21  3]
[11 18 25  2  9]]

Vector b: [10 26 42 59 38]
```

```
# Solve the system Ax = b using numpy's solve function x = np.linalg.solve(A, b) # Print the solution vector x print("Solution vector x:\n") print(x)
```

```
Solution vector x:

[-0.00833333  0.0724359   1.47179487  1.48782051 -0.33141026]
```

```
# Calculate the residual r = A*x - b r = np.dot(A, x) - b # Print the residual vector r print("\nResidual vector r:") print(r)
```

```
Residual vector r:
[ 0.00000000e+00  3.55271368e-15  0.00000000e+00  0.00000000e+00
 -7.10542736e-15]
```

```
import scipy.linalg
# Perform LU decomposition of A
P, L, U = scipy.linalg.lu(A) # Note: Use scipy.linalg.lu for LU decomposition
# Solve the system Ax = b using LU decomposition x1
= np.linalg.solve(A, b)
# Calculate the error between solutions err1
= np.dot(A, x1) - b
# Print matrices P, L, U and vectors x1, err1
print("Matrix P (Permutation matrix):") print(P)
print("\nMatrix L (Lower triangular matrix):")
print(L) print("\nMatrix U (Upper triangular
matrix):") print(U) print("\nSolution vector
x1:") print(x1) print("\nError vector err1:")
print(err1)
```

```
Matrix P (Permutation matrix):
[[0. 1. 0. 0. 0.]
 [1. 0. 0. 0. 0.]
 [0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 1.]
[0. 0. 1. 0. 0.]]

Matrix L (Lower triangular matrix):
[[1.         0.         0.         0.         0.        ]
 [0.73913043 1.         0.         0.         0.        ]
 [0.47826087 0.76873662 1.         0.         0.        ]
 [0.17391304 0.25267666 0.5163652  1.         0.        ]
 [0.43478261 0.48394004 0.72308355 0.92307692 1.        ]]

Matrix U (Upper triangular matrix):
[[ 23.           5.           7.          14.          16.        ]
 [  0.          20.30434783  -4.17391304  -2.34782609   3.17391304]
 [  0.           0.          24.8608137   -2.89079229  -1.09207709]
 [  0.           0.           0.          19.65116279  18.97932817]
 [  0.           0.           0.           0.         -22.22222222]]

Solution vector x1:
[-0.00833333  0.0724359   1.47179487  1.48782051 -0.33141026]

Error vector err1:
[ 0.00000000e+00  3.55271368e-15  0.00000000e+00  0.00000000e+00
 -7.10542736e-15]
```

```
# Solve the system Ax = b using numpy's least squares function y =
np.linalg.lstsq(A, b, rcond=None)[0]
# Print the solution vector y
print("\nSolution vector y:") print(y)
```

```
Solution vector y:
[-0.00833333  0.0724359   1.47179487  1.48782051 -0.33141026]
```

```
# Solve the system Ax = b using matrix inverse
A_inv = np.linalg.inv(A) x2 = np.dot(A_inv, b)
# Calculate the residual r2 and error err2
r2 = np.dot(A, x2) - b err2 = x - x2
# Print solution vector x2, residual r2 and error err2
print("\nSolution vector x2 using inverse:") print(x2)
print("\nResidual vector r2:") print(r2) print("\nError
vector err2:") print(err2)
```

```
    Solution vector x2 using inverse:
    [-0.00833333  0.0724359   1.47179487  1.48782051 -0.33141026]

    Residual vector r2:
    [-3.55271368e-15 -3.55271368e-15 -7.10542736e-15  0.00000000e+00
     -7.10542736e-15]

    Error vector err2:
    [-7.63278329e-17  4.16333634e-17  0.00000000e+00  4.44089210e-16
      5.55111512e-17]
```

```
def rref(A):
    '''
    Computes the reduced row echelon
form (RREF) of a matrix A.     Parameters:
    A : numpy.ndarray
    Input matrix of shape (m, n)
Returns:
    R : numpy.ndarray
    Reduced row echelon form of matrix
    A
    '''
    A = A.astype(float)
m, n = A.shape     lead
= 0     for r in
range(m):         if
lead >= n:
          break         if A[r, lead] ==
0:             for i in range(r + 1, m):
if A[i, lead] != 0:
A[[r, i]] = A[[i, r]]
break         if A[r, lead] != 0:
          A[r] = A[r] / A[r, lead]
for i in range(m):             if i
!= r:
              A[i] -= A[i, lead] * A[r]
lead += 1






    return A
# Compute reduced row echelon form of [A | b]
C = np.hstack((A, b[:, np.newaxis]))
R = rref(C)
# Extract solution vector x3 from the RREF matrix x3
= R[:, -1]
# Calculate residuals r3 = np.dot(A, x3) - b err3 = x - x3
print("\nSolution vector x3 (from RREF):") print(x3)
print("\nResidual vector r3 (from RREF):") print(r3)
print("\nError vector err3 (differencebetween x and x3):")
print(err3)
```

```
    Solution vector x3 (from RREF):
    [-0.00833333  0.0724359   1.47179487  1.48782051 -0.33141026]

    Residual vector r3 (from RREF):
    [1.77635684e-15 7.10542736e-15 0.00000000e+00 2.13162821e-14
     0.00000000e+00]

    Error vector err3 (differencebetween x and x3):
    [-2.67147415e-16  9.71445147e-17  0.00000000e+00 -4.44089210e-16
      2.22044605e-16]
```

```
# Initialize Num
Num = 500
# Generate matrix A and vector b
A = np.random.rand (Num, Num) + Num * np.eye (Num)
b = np.random.rand (Num, 1)
# Method 1: Solve using backslash operator
start_time = time.time ()
x1 = np.linalg.solve (A, b)
end_time = time.time ()
time_backslash = end_time - start_time
# Method 2: Solve using matrix inverse
start_time = time.time ()
x2 = np.linalg.inv (A) @ b
end_time = time.time ()
time_inv = end_time - start_time
# Method 3: Solve using reduced row echelon form (   rref)
start_time = time.time ()
C = np.hstack ((A, b))
R = rref (C)
x3 = R [:, -1]
end_time = time.time ()
time_rref = end_time - start_time
# Print the solution vectors and computational tim   es
print ("\nSolution vector x1 (using backslash operator):"   )
print (x1)
print ("\nSolution vector x2 (using matrix inverse):"   )
print (x2)
print ("\nSolution vector x3 (using reduced row echelon f   orm):" )
print (x3)
print ("\nComputational times:" )
print (f"Backslash operator:  {time_backslash } seconds" )
print (f"Matrix inverse:  {time_inv } seconds" )
print (f"Reduced row echelon form (rref):   {time_rref } seconds" )
```

```
    -2.68246835e-04  1.51297525e-03  1.45351443e-03  1.06302800e-04
     1.01075961e-03  1.42084062e-03  4.68192135e-04  2.80103134e-05
     2.10954301e-04 -3.32553568e-04  1.24174234e-03  1.59176876e-03
    -1.79033025e-05 -2.99217591e-04  5.09368048e-04  1.19054571e-03
     9.69917259e-04  3.26273522e-04  1.63816027e-03  1.64502107e-03
     3.69509520e-04  1.39063556e-03  4.00426121e-04 -1.88794041e-04
     4.00601007e-04  1.33443250e-03  7.87311478e-04  1.56807783e-03
     7.97229373e-04  1.60427818e-03  1.15552830e-04  1.60232322e-03
     3.69579563e-05  7.46292208e-05  1.97108430e-04 -2.67622289e-04
    -1.42346542e-04 -3.22321680e-04 -1.05568377e-04  1.29743246e-03
     3.38995378e-04  1.09939993e-04  4.61682643e-05  1.36280545e-03
     1.18010790e-03  1.92661091e-04  8.32944013e-04  3.58053181e-04
     1.17268902e-03  8.10003265e-04  6.66713216e-05  4.40828512e-04
    -3.19269444e-04 -2.61798024e-05 -1.57664100e-04 -7.01599528e-05
     5.01031914e-04  9.45884566e-04  1.59665360e-03  1.44995252e-03
    -2.30043538e-04  1.31197578e-03  1.44177220e-03  5.50510085e-04
    -2.28755036e-04  1.17851812e-03 -1.16651960e-04  1.51728551e-03
     1.40188552e-03  7.47387282e-04 -1.39381987e-04  1.42206316e-03
     6.01144115e-04  2.72623128e-04  1.07258449e-03  1.07648701e-04
    -2.45630367e-04  5.58192783e-04  1.24516380e-03 -2.89240068e-04
     1.44400990e-03 -2.33175973e-04 -1.65326421e-04 -3.12802419e-04
    -1.64553262e-04  4.99296975e-04  7.34121349e-04  9.07972364e-04
     3.25327590e-04  7.67718343e-04  1.29487919e-03  8.58486255e-04
    -8.69092570e-05 -2.82214767e-04  5.25075714e-04  1.51532806e-03
    -2.53773893e-04 -6.62131108e-05  6.55775111e-04  5.79493580e-04
     9.51640616e-04  6.82264059e-04  1.01321584e-03 -1.19300815e-04
     7.76166580e-04  1.06252044e-03  2.00220297e-04  1.66002398e-03
     1.13479574e-03 -3.07257151e-04  1.35269629e-04  4.97503812e-04
     1.39493004e-03  2.85993472e-04 -1.75564786e-04  1.31380005e-03
     3.86000590e-04  1.23299943e-03  3.38636729e-04 -2.10672374e-05
     6.61977076e-04  1.41596663e-03  1.01852607e-03  7.56131847e-04
     1.32674406e-03 -2.83816716e-04 -2.90668683e-04 -2.07339890e-04
     1.32459994e-03  2.59847627e-04  1.41875317e-03  1.30093630e-03
     5.04563097e-04  1.39535975e-03  2.89793590e-05  5.11765766e-04
     8.67913756e-04  4.00503303e-04  1.60148318e-03 -1.32270249e-04
     5.39506483e-04  9.86954667e-04  9.81264676e-04  1.33561767e-03
     1.34041012e-03 -1.89687347e-04  2.55313654e-04  5.78583558e-04
     1.45834018e-03  2.05420017e-04  9.86902604e-04 -2.00438157e-05
     9.57694522e-04  8.67081608e-04  1.13405951e-03  1.20508978e-04
    -2.66563849e-04  1.34975082e-03  2.71353117e-04  1.63525165e-03
     1.16886220e-03  8.40376728e-04  1.23777458e-04  6.00392634e-04
     2.96709022e-04  1.72744272e-05  9.30341682e-04  2.64702414e-04
     6.77742595e-04  1.10000361e-03  1.59720842e-03  7.51499191e-04
     1.84362439e-04 -2.40171503e-06  6.67325429e-04  1.24677550e-03
     1.31604975e-03  9.99553753e-04  1.66082831e-03  1.03883160e-03
    -2.02580745e-04  1.08665824e-03  8.34285379e-04  5.40981116e-04
    -8.25161454e-05  1.36626790e-03 -1.34932626e-04  7.52559367e-04
     7.18613024e-04  8.16857542e-04  3.72440985e-04 -3.12629842e-04
     8.71680786e-04  9.36316674e-05  1.44944940e-03  1.17274333e-03
     1.38016521e-03  1.41216179e-03  2.24156688e-05 -1.66692137e-04
     1.55745466e-04  6.84047718e-04  1.30672440e-03  7.12461139e-04
     3.95319659e-04  8.13162973e-04  3.70210105e-04  1.09274914e-03
     1.39610118e-03  7.20668878e-04  1.07560256e-03  9.00280045e-04]

   Computational times:
   Backslash operator: 0.011322498321533203 seconds
   Matrix inverse: 0.026056528091430664 seconds
   Reduced row echelon form (rref): 1.3512084484100342 seconds
```

```
# Define matrix A and vector b forthe overdetermined system
A = np.array([[1, 2, 3],
                [4, 5, 6],
                [7, 8, 10],
[9, 11, 12]])

b = np.array([1, 2, 3, 4]).reshape(-1, 1) # Reshape b to be a columnvector
# Solve the system Ax = b using the backslash operator x
= np.linalg.lstsq(A, b, rcond=None)[0]
# Calculate the residual r = Ax - b r
= np.dot(A, x) - b
# Print the solution vector x
print("\nSolution vector x:") print(x)
# Print the residual vector r
print("\nResidual vector r:") print(r)
# Calculate and print the solution using normal equations for comparison
ATA_inv = np.linalg.inv(np.dot(A.T,A)) # Calculate (A.T * A)^(-1) ATb =
np.dot(A.T, b) #Calculate A.T * b y = np.dot(ATA_inv, ATb) #Solve normal
equations A.T * A * y= A.T * b
# Calculate the difference between x and y to verify accuracy err =
x - y
# Print the solution vector y from normal equations
print("\nSolution vector y (from normal equations):") print(y)
# Print the difference vector err (should be close to zero)
print("\nDifference vector err (x - y):") print(err)
```

```
    Solution vector x:
    [[-0.24630542]
     [ 0.38916256]  [ 0.16256158]]

    Residual vector r:
    [[ 0.01970443]
     [-0.06403941]
     [ 0.01477833]
     [ 0.01477833]]

    Solution vector y (from normal equations):
    [[-0.24630542]
     [ 0.38916256]
     [ 0.16256158]]

    Difference vector err (x - y):
    [[-1.13797860e-15]
     [-3.75255382e-14]
     [ 1.19071419e-14]]
```

```
# Define matrix A and vector b for the underdetermined system
A = np.array([[1, 2, 3],
[4, 5, 6],
[7, 8, 9],
 [           ]])
 [10, 11, 12]]) b = np.array([1, 3, 5, 7]).reshape(-1, 1) #Remove the reshape, as it's
unnecessary
# Solve the system Ax = b using the backslash operator x
= np.linalg.lstsq(A, b, rcond=None)[0] # Calculate the
residual r1 = Ax - b r1 = np.dot(A, x) - b
# Print the solution vector x
print("\nSolution vector x:") print(x)
# Print the residual vector r1
print("\nResidual vector r1:") print(r1)
# Obtain another particular solution using the pseudoinverse pinv(A)
A_pinv = np.linalg.pinv(A) y = np.dot(A_pinv, b)
# Calculate the residual r2 = Ax - b for solution y r2
= np.dot(A, y) - b
# Print the solution vector y from pseudoinverse
print("\nSolution vector y (from pseudoinverse):") print(y)
# Print the residual vector r2
print("\nResidual vector r2:") print(r2)
```

```
    Solution vector x:
    [[0.38888889]
     [0.22222222]  [0.05555556]]

    Residual vector r1:
    [[8.8817842e-16]
     [8.8817842e-16]
     [8.8817842e-16]
     [0.0000000e+00]]

    Solution vector y (from pseudoinverse):
    [[0.38888889]
     [0.22222222]  [0.05555556]]

    Residual vector r2:
    [[ 1.55431223e-15]
     [-4.44089210e-16]
     [-2.66453526e-15]
     [-4.44089210e-15]]
```

Start coding or generate with AI.

```
# Project 5: Systems of linear equations and college football team ranking (with an example of the Big 12) import numpy as np from scipy.io import loadmat
```

```
# Load the matrices from .mat files data_scores =
loadmat("Scores.mat") data_differentials =
loadmat("Differentials.mat")
# Extract the relevant matrices
Scores = data_scores['Scores']
Differentials = data_differentials['Differentials']


# Colley's method # Define
variables games =
np.abs(Scores) total =
np.sum(games, axis=1)
# Construct Colley's matrix and the right-hand side vector
ColleyMatrix = 2 * np.eye(10) + np.diag(total) - games
RightSide = 1 + 0.5 * np.sum(Scores, axis=1)
# Print to verify print("ColleyMatrix:\n",
ColleyMatrix) print("\nRightSide:\n",
RightSide)
```

```
ColleyMatrix:
 [[11. -1. -1. -1. -1. -1. -1. -1. -1. -1.]
 [-1. 11. -1. -1. -1. -1. -1. -1. -1. -1.]
 [-1. -1. 11. -1. -1. -1. -1. -1. -1. -1.]
 [-1. -1. -1. 11. -1. -1. -1. -1. -1. -1.]
 [-1. -1. -1. -1. 11. -1. -1. -1. -1. -1.]
 [-1. -1. -1. -1. -1. 11. -1. -1. -1. -1.]
 [-1. -1. -1. -1. -1. -1. 11. -1. -1. -1.]
 [-1. -1. -1. -1. -1. -1. -1. 11. -1. -1.]
 [-1. -1. -1. -1. -1. -1. -1. -1. 11. -1.]
 [-1. -1. -1. -1. -1. -1. -1. -1. -1. 11.]]

RightSide:
 [-0.5 -1.5 -2.5  2.5  5.5  3.5  0.5 -0.5 -0.5  2.5]
```

```
# Solve the linear system using np.linalg.solve
RanksColley = np.linalg.solve(ColleyMatrix, RightSide)
# Variables: RanksColley print("RanksColley:\n",
RanksColley)
```

```
RanksColley:
 [0.33333333 0.25       0.16666667 0.58333333 0.83333333 0.66666667
 0.41666667 0.33333333 0.33333333 0.58333333]
```

```
# Teams list
Teams = [
'Baylor', 'Iowa State', 'University of Kansas', 'Kansas State',
'University of Oklahoma', 'Oklahoma State', 'Texas Christian',
'University of Texas Austin', 'Texas Tech', 'West Virginia' ]
# Sort the ranks in descending order and get the order of indices
Order = np.argsort(RanksColley)[::-1]
RanksDescend = RanksColley[Order]
# Display the results
print('\n') for j in
range(10):
    print(f'{RanksColley[Order[j]]:8.3f} {Teams[Order[j]]:15s}')
```

```
   0.833 University of Oklahoma
   0.667 Oklahoma State    0.583 Kansas State      0.583 West Virginia     0.417 Texas Christian    0.333 Baylor
   0.333 Texas Tech        0.333 University of Texas Austin
   0.250 Iowa State
   0.167 University of Kansas
```

```
# Massey's method
l = 0 P = []
B = []
# Loop through the upper triangular part of the Differentials matrix
for j in range(9):     for k in range(j + 1, 10):         if
Differentials[j, k] != 0:
            l += 1
row = np.zeros(10)
row[j] = 1             row[k] =
-1
            P.append(row)
            B.append(Differentials[j, k])
# Convert lists to numpy arrays
P = np.array(P)
B = np.array(B) #
Variables: P, B
print("Matrix P:\n", P)
print("Vector B:\n", B)
```

```
Matrix P:
 [[ 1. -1.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 1.  0. -1.  0.  0.  0.  0.  0.  0.  0.]
 [ 1.  0.  0. -1.  0.  0.  0.  0.  0.  0.]
 [ 1.  0.  0.  0. -1.  0.  0.  0.  0.  0.]
 [ 1.  0.  0.  0.  0. -1.  0.  0.  0.  0.]
 [ 1.  0.  0.  0.  0.  0. -1.  0.  0.  0.]
 [ 1.  0.  0.  0.  0.  0.  0. -1.  0.  0.]
 [ 1.  0.  0.  0.  0.  0.  0.  0. -1.  0.]
 [ 1.  0.  0.  0.  0.  0.  0.  0.  0. -1.]
 [ 0.  1. -1.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  1.  0. -1.  0.  0.  0.  0.  0.  0.]
 [ 0.  1.  0.  0. -1.  0.  0.  0.  0.  0.]
 [ 0.  1.  0.  0.  0. -1.  0.  0.  0.  0.]
 [ 0.  1.  0.  0.  0.  0. -1.  0.  0.  0.]
 [ 0.  1.  0.  0.  0.  0.  0. -1.  0.  0.]
 [ 0.  1.  0.  0.  0.  0.  0.  0. -1.  0.]
 [ 0.  1.  0.  0.  0.  0.  0.  0.  0. -1.]
 [ 0.  0.  1. -1.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  1.  0. -1.  0.  0.  0.  0.  0.]
 [ 0.  0.  1.  0.  0. -1.  0.  0.  0.  0.]
 [ 0.  0.  1.  0.  0.  0. -1.  0.  0.  0.]
 [ 0.  0.  1.  0.  0.  0.  0. -1.  0.  0.]
 [ 0.  0.  1.  0.  0.  0.  0.  0. -1.  0.]
 [ 0.  0.  1.  0.  0.  0.  0.  0.  0. -1.]
 [ 0.  0.  0.  1. -1.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  1.  0. -1.  0.  0.  0.  0.]
 [ 0.  0.  0.  1.  0.  0. -1.  0.  0.  0.]
 [ 0.  0.  0.  1.  0.  0.  0. -1.  0.  0.]
 [ 0.  0.  0.  1.  0.  0.  0.  0. -1.  0.]
 [ 0.  0.  0.  1.  0.  0.  0.  0.  0. -1.]
 [ 0.  0.  0.  0.  1. -1.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  1.  0. -1.  0.  0.  0.]
 [ 0.  0.  0.  0.  1.  0.  0. -1.  0.  0.]
 [ 0.  0.  0.  0.  1.  0.  0.  0. -1.  0.]
 [ 0.  0.  0.  0.  1.  0.  0.  0.  0. -1.]
 [ 0.  0.  0.  0.  0.  1. -1.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  1.  0. -1.  0.  0.]
 [ 0.  0.  0.  0.  0.  1.  0.  0. -1.  0.]
 [ 0.  0.  0.  0.  0.  1.  0.  0.  0. -1.]
 [ 0.  0.  0.  0.  0.  0.  1. -1.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  1.  0. -1.  0.]
 [ 0.  0.  0.  0.  0.  0.  1.  0.  0. -1.]
 [ 0.  0.  0.  0.  0.  0.  0.  1. -1.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  1.  0. -1.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  1. -1.]] Vector B:
 [  3  42 -21 -21  11 -40  -1 -19  -3   7  -5 -10  -7 -21 -21  56 -30 -15
 -53 -24  -1   3 -36 -27 -21  -7  24   3   6  -1  18   6   5   7  28  25
  18   1  17  22  -3 -24   8  -4 -31]
```

```
# Create the normal system of linear equations
A = np.dot(P.T, P)
D = np.dot(P.T, B) #
Variables: A, D
print("Matrix A:\n", A)
print("Vector D:\n", D)
```

```
Matrix A:
 [[ 9. -1. -1. -1. -1. -1. -1. -1. -1. -1.]
 [-1.  9. -1. -1. -1. -1. -1. -1. -1. -1.]
 [-1. -1.  9. -1. -1. -1. -1. -1. -1. -1.]
 [-1. -1. -1.  9. -1. -1. -1. -1. -1. -1.]
 [-1. -1. -1. -1.  9. -1. -1. -1. -1. -1.]
 [-1. -1. -1. -1. -1.  9. -1. -1. -1. -1.]
 [-1. -1. -1. -1. -1. -1.  9. -1. -1. -1.]
 [-1. -1. -1. -1. -1. -1. -1.  9. -1. -1.]
 [-1. -1. -1. -1. -1. -1. -1. -1.  9. -1.]
 [-1. -1. -1. -1. -1. -1. -1. -1. -1.  9.]] Vector D:
 [ -49.  -34. -202.   45.  169.   70.    2.  -25.  -51.   75.]
```

```
# Substitute the last row of the matrix and the last element of the vector
A[9, :] = np.ones(10)
D[9] = 0
# Print the updated matrix and vector
print("Updated Matrix A:\n", A) print("Updated
Vector D:\n", D)
```

```
Updated Matrix A:
 [[ 9. -1. -1. -1. -1. -1. -1. -1. -1. -1.]
 [-1.  9. -1. -1. -1. -1. -1. -1. -1. -1.]
 [-1. -1.  9. -1. -1. -1. -1. -1. -1. -1.]
 [-1. -1. -1.  9. -1. -1. -1. -1. -1. -1.]
 [-1. -1. -1. -1.  9. -1. -1. -1. -1. -1.]
 [-1. -1. -1. -1. -1.  9. -1. -1. -1. -1.]
 [-1. -1. -1. -1. -1. -1.  9. -1. -1. -1.]
 [-1. -1. -1. -1. -1. -1. -1.  9. -1. -1.]
 [-1. -1. -1. -1. -1. -1. -1. -1.  9. -1.]
 [ 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.]] Updated
Vector D:
 [ -49.  -34. -202.   45.  169.   70.    2.  -25.  -51.    0.]
```

```
# Solve the system
RanksMassey = np.linalg.solve(A, D) # Print the results
print("RanksMassey:\n", RanksMassey)
```

```
RanksMassey:
 [ -4.9  -3.4 -20.2   4.5  16.9   7.    0.2  -2.5  -5.1   7.5]
```

```
# Teams list
Teams = ['Baylor', 'Iowa State', 'University of Kansas', 'Kansas State',
'University of Oklahoma', 'Oklahoma State', 'Texas Christian',
'University of Texas Austin', 'Texas Tech', 'West Virginia']
# Sort the ranks in descending order
Order = np.argsort(RanksMassey)[::-1]
RanksDescend = RanksMassey[Order]
# Print the results
print("\nMassey Rankings:") for
j in range(10):
    print(f'{RanksDescend[j]:8.3f} {Teams[Order[j]]:<15}')
```

```
Massey Rankings:
  16.900 University of Oklahoma
   7.500 West Virginia
7.000 Oklahoma State    4.500
Kansas State
   0.200 Texas Christian   -2.500
University of Texas Austin
  -3.400 Iowa State
  -4.900 Baylor
  -5.100 Texas Tech
 -20.200 University of Kansas
```

```
print("Compare results of past 2 tasks"  )
```

```
Compare results of past 2 tasks
```

```
# Identify the current top two teams according to Colley's rankings top_teams_colley =
sorted(range(len(RanksColley)), key=lambda i: RanksColley[i], reverse=True)[:2]
# Simulate switching the result of the game between the top two teams
# For example, if team 0 (top_teams_colley[0]) played against team 1 (top_teams_colley[1])
# and team 0 lost, we switch it to a win.
Scores[top_teams_colley[0], top_teams_colley[1]] = -Scores[top_teams_colley[0], top_teams_colley[1]]
# Recalculate Colley's rankings
games = np.abs(Scores) total =
np.sum(games, axis=1)
ColleyMatrix = 2 * np.eye(10) + np.diag(total) - games
RightSide = (1 + 0.5 * np.sum(Scores, axis=1))
RanksColley_updated = np.linalg.solve(ColleyMatrix, RightSide)
# Display the updated rankings teams = ['Baylor', 'Iowa State', 'University
of Kansas', 'Kansas State',
'University of Oklahoma', 'Oklahoma State', 'Texas Christian',
'University of Texas Austin', 'Texas Tech', 'West Virginia']
# Sort and print rankings order_updated =
np.argsort(RanksColley_updated)[::-1] print("\nUpdated Colley's
Rankings After Game Result Switch:") for j in range(10):
    print(f"{RanksColley_updated[order_updated[j]]:8.3f} {teams[order_updated[j]]}")
# Reset the game result for future calculations
Scores[top_teams_colley[0], top_teams_colley[1]] = -Scores[top_teams_colley[0], top_teams_colley[1]]
```

```
Updated Colley's Rankings After Game Result Switch:
   0.708 University of Oklahoma
   0.625 Oklahoma State
   0.542 Kansas State
   0.542 West Virginia
   0.375 Texas Christian
   0.292 Baylor
   0.292 Texas Tech    0.292
University of Texas Austin
   0.208 Iowa State
   0.125 University of Kansas
```

```
# Identify the current top two teams according to Massey's rankings top_teams_massey =
sorted(range(len(RanksMassey)), key=lambda i: RanksMassey[i], reverse=True)[:2]
# Simulate switching the result of the game between the top two teams for Massey's method
# Adjust the differential matrix Differentials for the switched game result
# For example, if team 0 (top_teams_massey[0]) played against team 1
(top_teams_massey[1])
# and team 0 lost, we switch it to a win. if
Differentials[top_teams_massey[0], top_teams_massey[1]] != 0:
    Differentials[top_teams_massey[0], top_teams_massey[1]] = - Differentials[top_teams_massey[0], top_teams_massey[1]]
# Reset the initial matrix P to be large enough to accommodate all possible rows
P = np.zeros((45, 10)) B = np.zeros(45) l = -1 # Initialize l to -1 because it will be
incremented at the beginning of the loop
# Populate P and B based on the conditionals in your loop
for j in range(9):     for k in range(j + 1, 10):
if Differentials[j, k] != 0:
            l += 1
P[l, j] = 1
            P[l, k] = -1
            B[l] = Differentials[j, k]
# Adjust for the last row substitution as described in the previous steps
P[44, :] = np.ones(10)
B[44] = 0
# Recalculate Massey's rankings
A = np.dot(P.T, P)
D = np.dot(P.T, B)
# Solve the system again
RanksMassey_updated = np.linalg.solve(A, D)
# Display the updated rankings print("\nUpdated Massey's Rankings
After Game Result Switch:") order_updated_massey =
np.argsort(RanksMassey_updated)[::-1] for j in range(10):
    print(f"{RanksMassey_updated[order_updated_massey[j]]:8.3f} {teams[order_updated_massey[j]]}")
# Reset the game result for future calculations if
Differentials[top_teams_massey[0], top_teams_massey[1]] != 0:
    Differentials[top_teams_massey[0], top_teams_massey[1]] = - Differentials[top_teams_massey[0], top_teams_massey[1]]
```

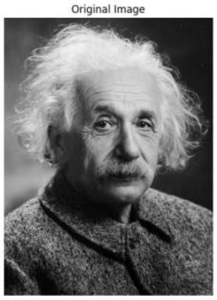
```
Updated Massey's Rankings After Game Result Switch:
  11.500 West Virginia
  11.300 University of Oklahoma
   7.000 Oklahoma State
   4.500 Kansas State
   0.200 Texas Christian   -2.500
University of Texas Austin
  -3.400 Iowa State
  -3.500 Texas Tech
  -4.900 Baylor
 -20.200 University of Kansas
```

Start coding or generate with AI.

```
# Project 6: Convolution, inner product, and image processing revisited import numpy as
np  # For numerical operations import cv2  # For loading and processing images from
scipy.signal import convolve2d  # For 2D convolution (conv2 equivalent) from
scipy.ndimage import convolve  # Alternative for filtering (filter2 equivalent) import
matplotlib.pyplot as plt
```

```
# Load the image
ImJPG = cv2.imread("Albert_Einstein_Head.jpg")
# Convert to RGB (OpenCV loads images in BGR format)
ImJPG = cv2.cvtColor(ImJPG, cv2.COLOR_BGR2RGB)
# Get dimensions m, n, _ = ImJPG.shape
# Display image dimensions
print(f"Image dimensions: {m} x {n}")
# Display the image
plt.figure(figsize=(6, 6))
plt.imshow(ImJPG) plt.axis('off')
plt.title('Original Image')
plt.show()
```

```
Image dimensions: 1200 x 900
```


Original Image

```
# Generate noise matrix with the same dimensions as ImJPG noise = 50 *
(np.random.rand(m, n, 3) - 0.5) # Add noise to each channel of the image
ImJPG_Noisy = np.double(ImJPG) + noise
```

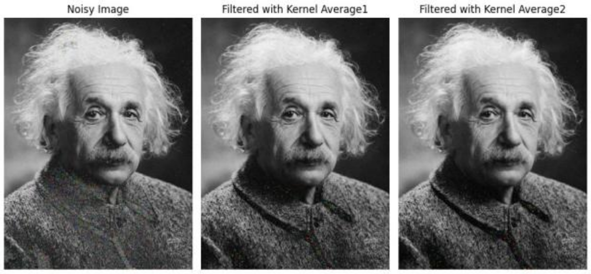
```
print("Smoothing filter matrices" )
```

```
Smoothing filter matrices
```

```
Kernel_Average1 = np.array([[0, 1, 0],
[1, 1, 1],
[0, 1, 0]]) / 5
Kernel_Average2 = np.array([[1, 1, 1],
[1, 1, 1], [1, 1, 1]])
/ 9
print(Kernel_Average1)
print(Kernel_Average2)
```

```
[[0.  0.2 0. ]
  [0.2 0.2 0.2]
 [0.  0.2 0. ]]
 [[0.11111111 0.11111111 0.11111111]
  [0.11111111 0.11111111 0.11111111]
  [0.11111111 0.11111111 0.11111111]]
```

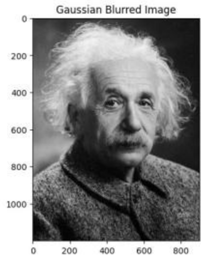
```
# Initialize arrays to store filtered results
ImJPG_Average1 = np.zeros_like(ImJPG_Noisy, dtype=np.float64)
ImJPG_Average2 = np.zeros_like(ImJPG_Noisy, dtype=np.float64) #
Apply the convolution filter to each channel separately for
channel in range(3):
    ImJPG_Average1[:, :, channel] = convolve2d(ImJPG_Noisy[:, :, channel], Kernel_Average1,
mode='same', boundary='symm')
    ImJPG_Average2[:, :, channel] = convolve2d(ImJPG_Noisy[:, :, channel],
Kernel_Average2, mode='same', boundary='symm')
# Convert the results back to uint8 format for display
ImJPG_Average1 = np.uint8(ImJPG_Average1)
ImJPG_Average2 = np.uint8(ImJPG_Average2)
# Display the resulting images
plt.figure(figsize=(10, 5)) plt.subplot(1,
3, 1) plt.imshow(np.uint8(ImJPG_Noisy))
plt.title('Noisy Image') plt.axis('off')
plt.subplot(1, 3, 2)
plt.imshow(ImJPG_Average1)
plt.title('Filtered with Kernel Average1')
plt.axis('off') plt.subplot(1, 3, 3)
plt.imshow(ImJPG_Average2)
plt.title('Filtered with Kernel Average2')
plt.axis('off') plt.tight_layout()
plt.show()
```


Noisy Image
Filtered with Kernel Average1
Filtered with Kernel Average2

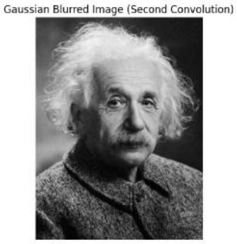
```
# Define the Gaussian blur kernel matrix
Kernel_Gauss = np.array([[0, 1, 0],
[1, 4, 1], [0, 1, 0]])
/ 8 print("Kernel
Gauss:")
print(Kernel_Gauss)
```

```
Kernel Gauss:
 [[0.    0.125 0.    ]
  [0.125 0.5   0.125]
  [0.    0.125 0.    ]]
```

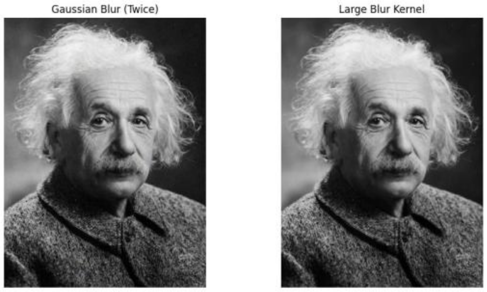
```
# Initialize arrays to store filtered results
ImJPG_Gauss = np.zeros_like(ImJPG_Noisy, dtype=np.float64)
#Apply the convolution filter to each channel separately for
channel in range(3):
    ImJPG_Gauss[:, :, channel] = convolve2d(ImJPG_Noisy[:, :, channel], Kernel_Gauss, mode='same', boundary='symm')
# Convert the results back to uint8 format for display
ImJPG_Gauss = np.uint8(np.clip(ImJPG_Gauss, 0, 255))
# Display the resulting image
plt.figure() plt.imshow(ImJPG_Gauss)
plt.title('Gaussian Blurred Image')
plt.show()
```



```
# Perform Gaussian blur convolution on ImJPG_Gauss
ImJPG_Gauss2 = np.zeros_like(ImJPG_Gauss, dtype=np.float64) for
channel in range(3):
    ImJPG_Gauss2[:, :, channel] = convolve2d(ImJPG_Gauss[:, :, channel], Kernel_Gauss, mode='same', boundary='symm')
ImJPG_Gauss2 = np.uint8(np.clip(ImJPG_Gauss2, 0, 255))
# Display the resulting image plt.figure()
plt.imshow(ImJPG_Gauss2) plt.title('Gaussian Blurred Image
(Second Convolution)') plt.axis('off') plt.show()
```



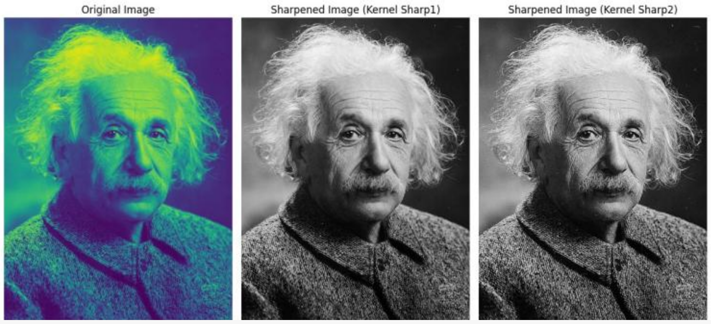
```
# Define the larger blur kernel (Kernel Large)
Kernel_Large = np.array([[0, 1, 2, 1, 0],
[1, 4, 8, 4, 1],
[2, 8, 16, 8, 2],
[1, 4, 8, 4, 1],
[0, 1, 2, 1, 0]]) / 80
# Apply the larger blur kernel to the image
ImJPG_Large = np.zeros_like(ImJPG, dtype=np.float64) for
channel in range(3):
    ImJPG_Large[:, :, channel] = convolve2d(ImJPG[:, :, channel], Kernel_Large, mode='same', boundary='symm')
ImJPG_Large = np.uint8(np.clip(ImJPG_Large, 0, 255))
# Display the resulting images plt.figure(figsize=(10,
5))
# Display Gaussian blurred image (second convolution)
plt.subplot(1, 2, 1) plt.imshow(ImJPG_Gauss2)
plt.title('Gaussian Blur (Twice)') plt.axis('off')
# Display larger blur image
plt.subplot(1, 2, 2)
plt.imshow(ImJPG_Large)
plt.title('Large Blur Kernel')
plt.axis('off')
plt.tight_layout() plt.show()
```



```
# Define the sharpening kernels
Kernel_Sharp1 = np.array([[0, -1, 0],
[-1, 5, -1],
[0, -1, 0]])
Kernel_Sharp2 = np.array([[-1, -1, -1],
[-1, 9, -1],
[-1, -1, -1]])
# Display the kernels (optional) print("Kernel
Sharp1:\n", Kernel_Sharp1) print("\nKernel
Sharp2:\n", Kernel_Sharp2)
```

```
Kernel Sharp1:
 [[ 0 -1  0]
 [-1  5 -1]
[ 0 -1  0]]

Kernel Sharp2:
 [[-1 -1 -1]
 [-1  9 -1]
 [-1 -1 -1]]
```

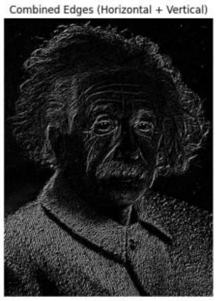
```
# Check the dimensions of ImJPG if
ImJPG.ndim == 3:
# Convert RGB to grayscale if necessary
    ImJPG = np.mean(ImJPG, axis=2).astype(np.uint8)
# Define the sharpening kernels
Kernel_Sharp1 = np.array([[0, -1, 0],
[-1, 5, -1],
[0, -1, 0]])
Kernel_Sharp2 = np.array([[-1, -1, -1],
[-1, 9, -1],
[-1, -1, -1]])
# Apply convolution with Kernel_Sharp1
ImJPG_Sharp1 = convolve2d(ImJPG, Kernel_Sharp1, mode='same', boundary='symm') #
Apply convolution with Kernel_Sharp2
ImJPG_Sharp2 = convolve2d(ImJPG, Kernel_Sharp2, mode='same', boundary='symm')
# Convert the results back to uint8 format for display
ImJPG_Sharp1 = np.clip(ImJPG_Sharp1, 0, 255).astype(np.uint8)
ImJPG_Sharp2 = np.clip(ImJPG_Sharp2, 0, 255).astype(np.uint8)
# Display the results plt.figure(figsize=(12,
6)) plt.subplot(1, 3, 1) plt.imshow(ImJPG)
plt.title('Original Image') plt.axis('off')
plt.subplot(1, 3, 2) plt.imshow(ImJPG_Sharp1,
cmap='gray') plt.title('Sharpened Image
(Kernel Sharp1)') plt.axis('off')
plt.subplot(1, 3, 3) plt.imshow(ImJPG_Sharp2,
cmap='gray') plt.title('Sharpened Image
(Kernel Sharp2)') plt.axis('off')
plt.tight_layout() plt.show()
```


Original Image
Sharpened Image (Kernel Sharp1)
Sharpened Image (Kernel Sharp2)

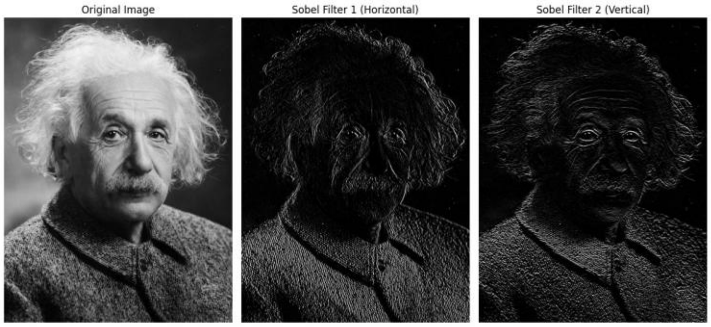
```
print("Kernel sobel theory" )
```

```
Kernel sobel theory
```

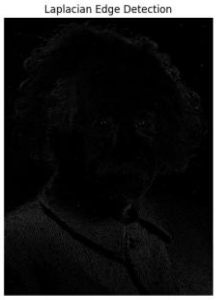
```
# Check the dimensions of ImJPG if
ImJPG.ndim == 3:
# Convert RGB to grayscale if necessary
    ImJPG = np.mean(ImJPG, axis=2).astype(np.uint8)
# Define Sobel kernels
Kernel_Sobel1 = np.array([[-1, 0, 1],
[-2, 0, 2],
[-1, 0, 1]])
Kernel_Sobel2 = np.array([[-1, -2, -1],
[0, 0, 0],
[1, 2, 1]])
# Perform convolution with Sobel kernels
ImJPG_Sobel1 = convolve2d(ImJPG, Kernel_Sobel1, mode='same', boundary='symm')
ImJPG_Sobel2 = convolve2d(ImJPG, Kernel_Sobel2, mode='same', boundary='symm')
# Clip and convert the results back to uint8 for display
ImJPG_Sobel1 = np.uint8(np.clip(ImJPG_Sobel1, 0, 255))
ImJPG_Sobel2 = np.uint8(np.clip(ImJPG_Sobel2, 0, 255))
# Display the results
plt.figure(figsize=(12, 6))
plt.subplot(1, 3, 1)
plt.imshow(ImJPG, cmap='gray')
plt.title('Original Image')
plt.axis('off') plt.subplot(1,
3, 2)
plt.imshow(ImJPG_Sobel1, cmap='gray')
plt.title('Sobel Filter 1 (Horizontal)')
plt.axis('off') plt.subplot(1, 3, 3)
plt.imshow(ImJPG_Sobel2, cmap='gray')
plt.title('Sobel Filter 2 (Vertical)')
plt.axis('off') plt.tight_layout()
plt.show()
```



```
# Check if the image is RGB and convert to grayscale if necessary if
ImJPG.ndim == 3:
    ImJPG = np.mean(ImJPG, axis=2).astype(np.uint8)
# Define Sobel kernels
Kernel_Sobel1 = np.array([[-1, 0, 1],
[-2, 0, 2],
[-1, 0, 1]])
Kernel_Sobel2 = np.array([[-1, -2, -1],
[0, 0, 0],
[1, 2, 1]])
# Perform convolution with Sobel kernels
ImJPG_Sobel1 = convolve2d(ImJPG, Kernel_Sobel1, mode='same', boundary='symm')
ImJPG_Sobel2 = convolve2d(ImJPG, Kernel_Sobel2, mode='same', boundary='symm')
# Combine horizontal and vertical edge images
ImJPG_SobelCombined = ImJPG_Sobel1 + ImJPG_Sobel2
# Clip and convert the combined result back to uint8 for display
ImJPG_SobelCombined = np.uint8(np.clip(ImJPG_SobelCombined, 0, 255))
# Display the combined edge-detected image
plt.figure(figsize=(8, 6))
plt.imshow(ImJPG_SobelCombined, cmap='gray')
plt.title('Combined Edges (Horizontal + Vertical)')
plt.axis('off') plt.show()
```



```
if ImJPG.ndim == 2:
    ImJPG = np.stack([ImJPG] * 3, axis=-1) # Convert grayscale to RGB
# Define Laplace kernel
Kernel_Laplace = np.array([[0, -1, 0],
[-1, 4, -1],
[0, -1, 0]])
# Apply convolution with Laplace kernel to each channel separately
ImJPG_Laplace = np.zeros_like(ImJPG, dtype=np.float64) for channel in
range(3):
    ImJPG_Laplace[:, :, channel] = convolve2d(ImJPG[:, :, channel], Kernel_Laplace, mode='same', boundary='symm')
# Clip and convert the result back to uint8 for display
ImJPG_Laplace = np.uint8(np.clip(ImJPG_Laplace, 0, 255))
# Display the Laplacian edge-detected image
plt.figure(figsize=(8, 6))
plt.imshow(ImJPG_Laplace)
plt.title('Laplacian Edge Detection')
plt.axis('off') plt.show()
```



Start coding or generate with AI.

```
# Project 7: Norms, angles, and your movie choices
import numpy as np import scipy.io
```

```
# Load the .mat file data = scipy.io.loadmat('users_movies.mat') # Extract variables from
the loaded data movies = data['movies'] # Array of movie titles users_movies =
data['users_movies'] # Matrix of user ratings for movies users_movies_sort =
data['users_movies_sort'] # Extracted ratings for 20 most popular movies index_small =
data['index_small'] # Indexes of the popular movies trial_user = data['trial_user'] #
Ratings of the popular movies by a trial user
# Get the dimensions of the users_movies matrix m,
n = users_movies.shape
# Print the variables and their dimensions to verify
print(f"Movies: {movies.shape}") print(f"Users Movies:
{users_movies.shape}") print(f"Users Movies Sort:
{users_movies_sort.shape}") print(f"Index Small:
{index_small.shape}") print(f"Trial User:
{trial_user.shape}") print(f"Dimensions of users_movies: {m}
rows, {n} columns")
# Variables: movies, users_movies, users_movies_sort, index_small, trial_user, m, n
```

```
Movies: (3952, 1)
 Users Movies: (6040, 3952)
 Users Movies Sort: (6040, 20)
 Index Small: (1, 20)
 Trial User: (1, 20)
 Dimensions of users_movies: 6040 rows, 3952 columns
```

```
# Print the titles of the 20 most popular movies print('Rating
is based on movies:')
# Loop through the index_small array and print the corresponding movie titles for
idx in index_small.flatten():
    print(movies[idx][0]) print('\n')
```

```
Rating is based on movies:
 ['Search for One-eye Jimmy, The (1996)']
 ['Little Women (1994)']
 ['Princess Bride, The (1987)']
 ['Wings of Desire (Der Himmel über Berlin) (1987)']
 ['Kalifornia (1993)']
 ["Billy's Hollywood Screen Kiss (1997)"]
 ['Dances with Wolves (1990)']
 ['10 Things I Hate About You (1999)']
 ['Fried Green Tomatoes (1991)']
 ['Snow White and the Seven Dwarfs (1937)']
 ['Love Letter, The (1999)']
 ['Brazil (1985)']
 ['Homeward Bound II']
 ['Thomas Crown Affair, The (1999)']
 ['Taxi Driver (1976)']
 ['Mass Appeal (1984)']
 ['Raiders of the Lost Ark (1981)']
 ['Scout, The (1994)']
 ['Shallow Grave (1994)']
 ['Unforgiven (1992)']
```

```
print("\nSubtask 3\n")
# Get the dimensions of the users_movies_sort matrix m1,
n1 = users_movies_sort.shape
# Initialize an empty list to store the ratings of users who have rated all 20 popular movies ratings =
[]
# Loop through each row in users_movies_sort for
j in range(m1):
# Check if the product of the elements in the row is not zero (meaning no zeros in the row)
if np.prod(users_movies_sort[j, :]) != 0:
# Append the row to the ratings list
ratings.append(users_movies_sort[j, :]) # Convert
the ratings list to a NumPy array ratings =
np.array(ratings)
# Print the resulting ratings array print(f"Ratings:
{ratings.shape}")
```

```
Subtask 3

Ratings: (125, 20)
```

```
print("\nSubtask 4\n")
# Get the dimensions of the ratings matrix m2,
n2 = ratings.shape
# Initialize an empty list to store the Euclidean distances eucl =
[]
# Loop through each row in ratings for
i in range(m2):
# Calculate the Euclidean distance between the trial_user vector and the current row of ratings     distance
= np.linalg.norm(ratings[i, :] - trial_user.flatten())
# Append the distance to the eucl list eucl.append(distance)
# Convert the eucl list to a NumPy array eucl
= np.array(eucl)
# Print the resulting Euclidean distances print(f"Euclidean
distances: {eucl}")
# Variables: eucl
```

```
Subtask 4

Euclidean distances: [570.2499452]
```

```
print("\nSubtask 5\n")
# Sort the Euclidean distances in ascending order
DistIndex = np.argsort(eucl)
MinDist = np.sort(eucl)
# Find the index of the closest user closest_user_Dist
= DistIndex[0]
# Print the results print(f"Sorted Euclidean distances:
{MinDist}") print(f"Indices of users sorted by distance:
{DistIndex}") print(f"Index of closest user:
{closest_user_Dist}") # Variables: MinDist, DistIndex,
closest_user_Dist
```

```
Subtask 5

Sorted Euclidean distances: [570.2499452]
Indices of users sorted by distance: [0]
Index of closest user: 0
```

```
print("\nSubtask 6\n")
# Centralize the columns of the matrix ratings ratings_cent =
ratings - np.mean(ratings, axis=1).reshape(-1, 1)
# Centralize the trial_user vector trial_user_cent =
trial_user - np.mean(trial_user) # Print the centralized
ratings and trial_user vectors print(f"Centralized
ratings: \n{ratings_cent}") print(f"Centralized
trial_user: \n{trial_user_cent}")
# Variables: ratings_cent, trial_user_cent
```

```
Subtask 6

Centralized ratings:
[[-0.4   0.6   0.6  ...  0.6  -0.4  -1.4 ]
 [-0.55  0.45  0.45 ...  0.45  0.45  0.45]
 [ 0.6   0.6   0.6  ...  0.6   0.6  -0.4 ]  ...
 [ 0.55  0.55  0.55 ...  0.55  0.55 -0.45]
 [-0.5   0.5   0.5  ...  0.5   0.5   0.5 ]
 [-0.4  -0.4  -0.4  ...  0.6   0.6   0.6 ]] Centralized
 trial_user:
[[-0.45  1.55  1.55  1.55 -0.45 -2.45  0.55  1.55 -1.45  1.55 -1.45  1.55
  -0.45  0.55  0.55 -1.45 -1.45 -2.45 -0.45  1.55]]
```

```
# Initialize the pearson array pearson
= np.zeros(m2)
# Compute Pearson correlation coefficients for
i in range(m2):
    pearson[i] = np.corrcoef(ratings_cent[i, :], trial_user_cent.flatten())[0, 1]
# Print the resulting Pearson correlation coefficients print(f"Pearson
correlation coefficients: {pearson}")
# Variables: pearson
```

```
Pearson correlation coefficients: [ 0.07561935  0.36369309  0.0245783  -0.16556179  0.03582872  0.2140846
   0.14763779  -0.02178972  -0.0100936     0.09702463   0.05321628  -0.2062086
  0.16821448   0.60769118   0.32403366   0.19674451   0.36226722   0.33180702   -
  0.0053019   0.12647862  0.30960763  0.44063497 -0.06205716  0.4257963
  -0.183647   -0.15140397  0.24374604  0.14284374  0.19674451  0.39302868
   0.33727631  0.46562671  0.13151475 -0.01950268  0.15690109 -0.34105647
 0.24896632  0.21031844  0.15617376 -0.01194291 -0.11582368 -0.08011545
   0.10748615  0.39938107  0.21573203  0.16556179 -0.24313206  0.20259223
 0.27942446  0.33700654  0.08682524  0.13122024  0.32557134  0.0533586
   0.14811562  0.36933894 -0.00739965  0.28900603  0.13876403 -0.14800564
 0.33653018 -0.15539269  0.19444977 -0.10748615  0.35165724  0.06931045   0.28756254
 0.00267796  0.21716794 -0.32403366  0.13291394  0.24573705  -0.11582368  0.42128131
 -0.07397705  0.00267796  0.1170161   0.16556179
   0.3024463   0.23837048  0.28858643  0.43115223  0.04744598  0.10551014
 -0.01194291 -0.03570538  0.05623725  0.62896156 -0.0100936  -0.16677361
 -0.01791436  0.1800582   0.21691509  0.08682524  0.13545965  0.32924378
  0.23837048 -0.11781529  0.51951641  0.09871488  0.12869663  0.15107865
  0.21452952  0.06541399  0.05472927 -0.17929818  0.31402432  0.23580013   0.2549193
0.01917923  0.24222691  0.01126761  0.59988588  0.2533473
  0.09057869  0.12459704  0.05175259 -0.01748262  0.47449447 -0.07505349
  0.30217235 -0.10238909 -0.36958744  0.16909002 -0.1170161 ]
```

```
# Sort the Pearson correlation coefficients in descending order
PearsonIndex = np.argsort(pearson)[::-1]
MaxPearson = np.sort(pearson)[::-1]
# Find the index of the user with the highest correlation coefficient closest_user_Pearson = PearsonIndex[0] # Print the results print(f"Sorted Pearson correlation coefficients:\n {MaxPearson}") print(f"Indices of users sorted by Pearson correlation:\n {PearsonIndex}") print(f"Index of user with highest
 Pearson correlation: {closest_user_Pearson}")
# Variables: MaxPearson, PearsonIndex, closest_user_Pearson
```

```
Sorted Pearson correlation coefficients:
 [ 0.62896156  0.60769118  0.59988588  0.51951641  0.47449447  0.46562671
  0.44063497  0.43115223  0.4257963   0.42128131  0.39938107  0.39302868   0.36933894
0.36369309  0.36226722  0.35165724  0.33727631  0.33700654   0.33653018  0.33180702
0.32924378  0.32557134  0.32403366  0.31402432   0.30960763  0.3024463   0.30217235
0.28900603  0.28858643  0.28756254   0.27942446  0.2549193   0.2533473   0.24896632
0.24573705  0.24374604   0.24222691  0.23837048  0.23837048  0.23580013  0.21716794
0.21691509   0.21573203  0.21452952  0.2140846   0.21031844  0.20259223  0.19674451
0.19674451  0.19444977  0.1800582   0.16909002  0.16821448  0.16556179   0.16556179
0.15690109  0.15617376  0.15107865  0.14811562  0.14763779   0.14284374  0.13876403
0.13545965  0.13291394  0.13151475  0.13122024   0.12869663  0.12647862  0.12459704
0.1170161   0.10748615  0.10551014   0.09871488  0.09702463  0.09057869  0.08682524
0.08682524  0.07561935   0.06931045  0.06541399  0.05623725  0.05472927  0.0533586
0.05321628
 0.05175259   0.04744598   0.03582872   0.0245783    0.01917923   0.01126761
0.00267796   0.00267796  -0.0053019   -0.00739965  -0.0100936   -0.0100936   -
0.01194291 -0.01194291 -0.01748262 -0.01791436 -0.01950268 -0.02178972
 -0.03570538 -0.06205716 -0.07397705 -0.07505349 -0.08011545 -0.10238909
 -0.10748615 -0.11582368 -0.11582368 -0.1170161  -0.11781529 -0.14800564
 -0.15140397 -0.15539269 -0.16556179 -0.16677361 -0.17929818 -0.183647
 -0.2062086  -0.24313206 -0.32403366 -0.34105647 -0.36958744] Indices
of users sorted by Pearson correlation:
 [ 87  13 112  98 118  31  21  81  23  73  43  29  55   1  16  64  30  49
  60  17  95  52  14 106  20  78 120  57  80  66  48 108 113  36  71  26
 110  96  79 107  68  92  44 102   5  37  47  28  15  62  91 123  12  45
  77  34  38 101  54   6  27  58  94  70  32  51 100  19 115  76  42  83
  99   9 114  93  50   0  65 103  86 104  53  10 116  82   4   2 109 111
  67  75  18  56   8  88  39  84 117  90  33   7  85  22  74 119  41 121
  63  72  40 124  97  59  25  61   3  89 105  24  11  46  69  35 122]
Index of user with highest Pearson correlation: 87
```

```
# Compare the elements of the vectors DistIndex and PearsonIndex
print("Indices sorted by Euclidean distance:", DistIndex) print("Indices
sorted by Pearson correlation:", PearsonIndex)
# Check if the variables closest_user_Pearson and closest_user_Dist are the same if
closest_user_Pearson == closest_user_Dist:
    print("The variables closest_user_Pearson and closest_user_Dist are the same.") else:
    print("The variables closest_user_Pearson and closest_user_Dist are different.")
```

```
Indices sorted by Euclidean distance: [0]
Indices sorted by Pearson correlation: [ 87  13 112  98 118  31  21  81  23  73  43  29  55   1  16  64  30  49
  60  17  95  52  14 106  20  78 120  57  80  66  48 108 113  36  71  26
 110  96  79 107  68  92  44 102   5  37  47  28  15  62  91 123  12  45
  77  34  38 101  54   6  27  58  94  70  32  51 100  19 115  76  42  83
  99   9 114  93  50   0  65 103  86 104  53  10 116  82   4   2 109 111
  67  75  18  56   8  88  39  84 117  90  33   7  85  22  74 119  41 121
  63  72  40 124  97  59  25  61   3  89 105  24  11  46  69  35 122] The
 variables closest_user_Pearson and closest_user_Dist are different.
```

```
print("index_small shape:", index_small.shape) print("trial_user
shape:", trial_user.shape)
# Load the .mat file data = scipy.io.loadmat('users_movies.mat') # Extract variables from
the loaded data movies = data['movies'] # Array of movie titles users_movies =
data['users_movies'] # Matrix of user ratings for movies users_movies_sort =
data['users_movies_sort'] # Extracted ratings for 20 most popular movies index_small =
data['index_small'].flatten() # Flatten index_small to 1D array trial_user =
data['trial_user'].flatten() # Ensure trial_user is 1D array # Variables: movies,
users_movies, users_movies_sort, index_small, trial_user m, n = users_movies.shape
# Recommendations based on the distance criterion
recommend_dist = [] for k in range(n):     if
users_movies[closest_user_Dist, k] == 5:
        recommend_dist.append(k)
# Recommendations based on the Pearson correlation coefficient criterion
recommend_pearson = [] for k in range(n):     if
users_movies[closest_user_Pearson, k] == 5:
        recommend_pearson.append(k)
# Movies liked by the trial user
liked = [] for k in range(20):
if trial_user[k] == 5:
# Convert 2D index_small to 1D index and add to liked list         if k
< len(index_small):             liked.append(index_small[k]) # Convert
indices to movie titles liked_titles = [movies[i][0] for i in liked]
recommend_dist_titles = [movies[i][0] for i in recommend_dist]
recommend_pearson_titles = [movies[i][0] for i in recommend_pearson]
# Print the results print("Movies liked by the trial user:", liked_titles)
print("Recommended movies based on distance criterion:", recommend_dist_titles)
print("Recommended movies based on Pearson correlation criterion:",
recommend_pearson_titles)
# Variables: liked, recommend_dist, recommend_pearson
```

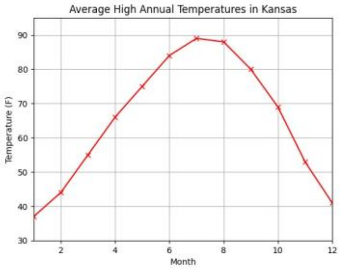
```
index_small shape: (20,)
trial_user shape: (20,)
Movies liked by the trial user: [array(['Little Women (1994)'], dtype='<U19'), array(['Princess Bride, The (1987)'], dtype='<U26'), array(['Wings of Desire (Der Himmel über Berlin) (1987)'], dtype='<U47'), array(['10 Things I Hate About You (1999)'], dtype='<U33'), array(['Snow White and the Seven Dwarfs (1937)
Recommended movies based on distance criterion: [array(['Toy Story (1995)'], dtype='<U16'), array(['Pocahontas (1995)'], dtype='<U17'), array(['Apollo 13 (1995)'], dtype='<U16'), array(["Schindler's List (1993)"], dtype='<U23'), array(['Beauty and the Beast (1991)'], dtype='<U27'), array(['Cinderella (1950)'],
Recommended movies based on Pearson correlation criterion: [array(['Taxi Driver (1976)'], dtype='<U18'), array(["Schindler's List (1993)"], dtype='<U23'), array(['Fargo (1996)'], dtype='<U12'), array(['Godfather, The (1972)'], dtype='<U21'), array(['North by Northwest (1959)'], dtype='<U25'), array(['Casablanca
```

```
# Function to print movie titles based on indices
def print_movie_titles (indices, movie_titles ):
    print("Movie Titles:" )
    for index  in indices :
        print(movie_titles [index])
    print()
# Print titles of movies liked by the trial user
print("Movies liked by the trial user:"  )
print_movie_titles (liked, movies )
# Print recommendations based on the distance crit   erion
print("Recommended movies based on distance criterion:"   )
print_movie_titles (recommend_dist , movies )
# Print recommendations based on the Pearson corre   lation criterion
print("Recommended movies based on Pearson correlation c   riterion:" )
print_movie_titles (recommend_pearson , movies )
```

```
[array(['Unforgiven (1992)'], dtype='<U17'  )]

Recommended movies based on distance criterion:
Movie Titles:
[array(['Toy Story (1995)'], dtype='<U16'  )]
[array(['Pocahontas (1995)'], dtype='<U17'  )]
[array(['Apollo 13 (1995)'], dtype='<U16'  )]
[array(["Schindler's List (1993)"], dtype='<U23'   )]
[array(['Beauty and the Beast (1991)'], dtype='<U27'   )]
[array(['Cinderella (1950)'], dtype='<U17'  )]
[array(['Mary Poppins (1964)'], dtype='<U19'  )]
[array(['Dumbo (1941)'], dtype='<U12'  )]
[array(['Sound of Music, The (1965)'], dtype='<U26'   )]
[array(['Paris Is Burning (1990)'], dtype='<U23'   )]
[array(['Back to the Future (1985)'], dtype='<U25'   )]
[array(['Ben-Hur (1959)'], dtype='<U14'  )]
[array(['City of Angels (1998)'], dtype='<U21'   )]
[array(['Rain Man (1988)'], dtype='<U15'  )]
[array(['Saving Private Ryan (1998)'], dtype='<U26'   )]
[array(["Bug's Life, A (1998)"], dtype='<U20'  )]
[array(['Christmas Story, A (1983)'], dtype='<U25'   )]
[array(['Awakenings (1990)'], dtype='<U17'  )]

Recommended movies based on Pearson correlation criterion:
Movie Titles:
[array(['Taxi Driver (1976)'], dtype='<U18'  )]
[array(["Schindler's List (1993)"], dtype='<U23'   )]
[array(['Fargo (1996)'], dtype='<U12'  )]
[array(['Godfather, The (1972)'], dtype='<U21'   )]
[array(['North by Northwest (1959)'], dtype='<U25'   )]
[array(['Casablanca (1942)'], dtype='<U17'  )]
[array(['Citizen Kane (1941)'], dtype='<U19'  )]
[array(['Mr. Smith Goes to Washington (1939)'], dtype='<U35'    )]
[array(['Bonnie and Clyde (1967)'], dtype='<U23'   )]
[array(['Bob Roberts (1992)'], dtype='<U18'  )]
```

```
# Project 8: Interpolation, extrapolation, and climate change
 import numpy as np  import matplotlib.pyplot as plt import
 scipy.io from scipy.io import loadmat from scipy.interpolate
 import interp1d from scipy.linalg import orth
```

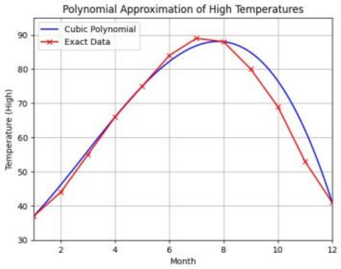
```
# High temperatures in Kansas (in Fahrenheit)
WeatherHigh = np.array([37, 44, 55, 66, 75, 84, 89, 88, 80, 69, 53, 41])
# Plot the temperatures plt.figure() plt.plot(range(1,
13), WeatherHigh, 'r-x') plt.axis([1, 12, 30, 95])
plt.title('Average High Annual Temperatures in Kansas')
plt.xlabel('Month') plt.ylabel('Temperature (F)')
plt.grid(True) plt.show()
```



```
# Months: January, May, August, December x
= np.array([1, 5, 8, 12]) V = np.vander(x,
increasing=True) # Select corresponding
temperatures y = WeatherHigh[[0, 4, 7,
11]]
# Solve for polynomial coefficients
CoefHigh = np.linalg.lstsq(V, y, rcond=None)[0] # Given
months: January, May, August, and December x =
np.array([1, 5, 8, 12]) y = WeatherHigh[x-1] # Select
corresponding temperatures
# Generate the Vandermonde matrix V
= np.vander(x, increasing=True)
# Solve for polynomial coefficients CoefHigh =
np.linalg.solve(V, y) print("Vandermonde Matrix (V):")
print(V) print("Coefficients of the cubic polynomial
(CoefHigh):") print(CoefHigh) print("Coefficients of the
cubic polynomial:", CoefHigh)
```

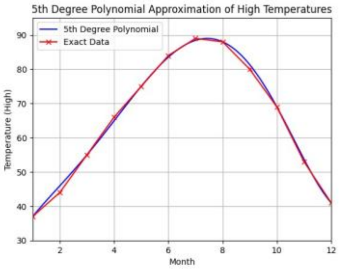
```
Vandermonde Matrix (V):
[[   1    1    1    1]
 [   1    5   25  125]
 [   1    8   64  512]
 [   1   12  144 1728]] Coefficients of the cubic
 polynomial (CoefHigh):
[29.48051948  6.41450216  1.24675325 -0.14177489]
Coefficients of the cubic polynomial: [29.48051948  6.41450216  1.24675325 -0.14177489]
```

```
# Evaluate the polynomial at the given set of points xc = np.arange(1, 12.1, 0.1) ycHigh =
np.polyval(CoefHigh[::-1], xc) # CoefHigh needs to be reversed for np.polyval
# Plot the polynomial and the original data plt.figure()
plt.plot(xc, ycHigh, 'b-', label='Cubic Polynomial')
plt.plot(range(1, 13), WeatherHigh, 'r-x', label='Exact Data')
plt.axis([1, 12, 30, 95]) plt.xlabel('Month')
plt.ylabel('Temperature (High)') plt.title('Polynomial
Approximation of High Temperatures') plt.legend() plt.grid(True)
plt.show()
```



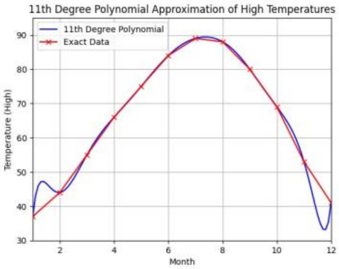
```
# Given months: January, March, May, August, October, and December
x_six = np.array([1, 3, 5, 8, 10, 12]) y_six = WeatherHigh[x_six - 1]
# Select corresponding temperatures
# Generate the Vandermonde matrix
V_six = np.vander(x_six, increasing=True)
# Solve for polynomial coefficients
CoefHigh_six = np.linalg.solve(V_six, y_six) print("Vandermonde
Matrix for six months (V_six):") print(V_six) print("Coefficients of
the 5th degree polynomial (CoefHigh_six):") print(CoefHigh_six)
# Evaluate the polynomial at the given set of points ycHigh_six =
np.polyval(CoefHigh_six[::-1], xc) # CoefHigh_six needs to be reversed for np.polyval #
Plot the polynomial and the original data plt.figure() plt.plot(xc, ycHigh_six, 'b-',
label='5th Degree Polynomial') plt.plot(range(1, 13), WeatherHigh, 'r-x', label='Exact
Data') plt.axis([1, 12, 30, 95]) plt.xlabel('Month') plt.ylabel('Temperature (High)')
plt.title('5th Degree Polynomial Approximation of High Temperatures') plt.legend()
plt.grid(True) plt.show()
```

```
Vandermonde Matrix for six months (V_six):
[[     1      1      1      1      1      1]
 [     1      3      9     27     81    243]
 [     1      5     25    125    625   3125]
 [     1      8     64    512   4096  32768]
 [     1     10    100   1000  10000 100000]
 [     1     12    144   1728  20736 248832]] Coefficients
 of the 5th degree polynomial (CoefHigh_six):
[ 2.51341991e+01  1.49771284e+01 -4.19289322e+00  1.21778499e+00
 -1.41305916e-01  5.08658009e-03]
```

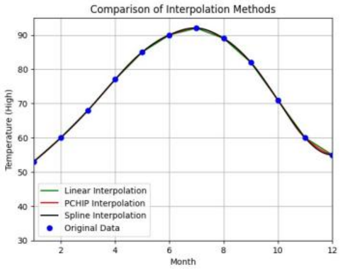


```
# All twelve months x_all =
np.arange(1, 13) y_all =
WeatherHigh # Generate the
Vandermonde matrix
V_all = np.vander(x_all, increasing=True)
# Solve for polynomial coefficients
CoefHigh_all = np.linalg.solve(V_all, y_all) print("Vandermonde Matrix
for all twelve months (V_all):") print(V_all) print("Coefficients of
the 11th degree polynomial (CoefHigh_all):") print(CoefHigh_all)
# Evaluate the polynomial at the given set of points ycHigh_all =
np.polyval(CoefHigh_all[::-1], xc) # CoefHigh_all needs to be reversed for np.polyval #
Plot the polynomial and the original data plt.figure() plt.plot(xc, ycHigh_all, 'b-',
label='11th Degree Polynomial') plt.plot(range(1, 13), WeatherHigh, 'r-x', label='Exact
Data') plt.axis([1, 12, 30, 95]) plt.xlabel('Month') plt.ylabel('Temperature (High)')
plt.title('11th Degree Polynomial Approximation of High Temperatures') plt.legend()
plt.grid(True) plt.show()
```

```
Vandermonde Matrix for all twelve months (V_all):
[[           1            1            1            1            1
             1            1            1            1            1
             1            1]
 [           1            2            4            8           16
            32           64          128          256          512
          1024         2048]
 [           1            3            9           27           81
           243          729         2187         6561        19683
         59049       177147]
 [           1            4           16           64          256
          1024         4096        16384        65536       262144
       1048576      4194304]
 [           1            5           25          125          625
          3125        15625        78125       390625      1953125
       9765625     48828125]
 [           1            6           36          216         1296
          7776        46656       279936      1679616     10077696
      60466176    362797056]
 [           1            7           49          343         2401
         16807       117649       823543      5764801     40353607
     282475249   1977326743]
 [           1            8           64          512         4096
         32768       262144      2097152     16777216    134217728
    1073741824   8589934592]
 [           1            9           81          729         6561
         59049       531441      4782969     43046721    387420489
    3486784401  31381059609]
 [           1           10          100         1000        10000
        100000      1000000     10000000    100000000   1000000000
   10000000000 100000000000]
 [           1           11          121         1331        14641
        161051      1771561     19487171    214358881   2357947691
   25937424601 285311670611]
 [           1           12          144         1728        20736
        248832      2985984     35831808    429981696   5159780352
   61917364224 743008370688]] Coefficients of the 11th degree
 polynomial (CoefHigh_all):
[-6.76999987e+02  2.08197828e+03 -2.49615549e+03  1.65367436e+03
 -6.79659628e+02  1.84081461e+02 -3.38253406e+01  4.24628379e+00
 -3.58862425e-01  1.95298716e-02 -6.18110654e-04  8.64297716e-06]
```

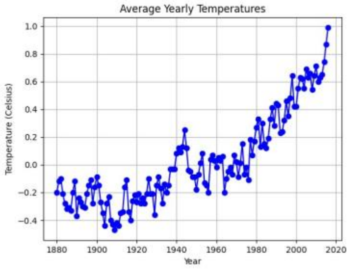


```
# Given data
x = np.arange(1, 13)
WeatherHigh = np.array([53, 60, 68, 77, 85, 90, 92, 89, 82, 71, 60, 55])
# Points for interpolation xc = np.arange(1, 12.1, 0.1) # Linear interpolation
ycHigh1 = np.interp(xc, x, WeatherHigh, left=None, right=None, period=None)
# Piecewise cubic Hermite interpolating polynomial (PCHIP)
from scipy.interpolate import PchipInterpolator
pchip_interpolator = PchipInterpolator(x, WeatherHigh)
ycHigh2 = pchip_interpolator(xc) # Cubic spline
interpolation from scipy.interpolate import CubicSpline
spline_interpolator = CubicSpline(x, WeatherHigh) ycHigh3 =
spline_interpolator(xc)
# Plot the interpolations plt.figure() plt.plot(xc,
ycHigh1, 'g-', label='Linear Interpolation') plt.plot(xc,
ycHigh2, 'r-', label='PCHIP Interpolation') plt.plot(xc,
ycHigh3, 'k-', label='Spline Interpolation') plt.plot(x,
WeatherHigh, 'bo', label='Original Data') plt.axis([1, 12,
30, 95]) plt.xlabel('Month') plt.ylabel('Temperature
(High)') plt.title('Comparison of Interpolation Methods')
plt.legend() plt.grid(True) plt.show()
```

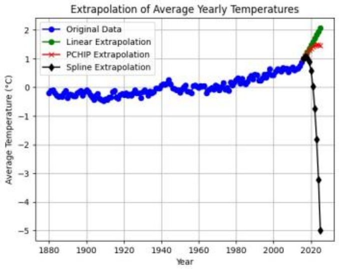


```
# Load temperature data data =
loadmat("temperature.mat")
temperature = data['temperature']
```

```
# Separate data into years and temperatures
years = temperature[:, 0] temp =
temperature[:, 1] # Plot the temperature data
plt.figure() plt.plot(years, temp, 'b-o')
plt.xlabel('Year') plt.ylabel('Temperature
(Celsius)') plt.title('Average Yearly
Temperatures') plt.grid(True)
plt.show ()
```



```
# Define the future years
futureyears = np.arange(2016, 2026)
# Linear extrapolation linear_interpolator = interp1d(years, temp, kind='linear',
fill_value='extrapolate') futuretemp1 = linear_interpolator(futureyears)
# Piecewise cubic Hermite interpolating polynomial (PCHIP) pchip_interpolator =
PchipInterpolator(years, temp, extrapolate=True) futuretemp2 =
pchip_interpolator(futureyears)
# Cubic spline interpolation spline_interpolator =
CubicSpline(years, temp, extrapolate=True) futuretemp3 =
spline_interpolator(futureyears)
# Plot the extrapolated data plt.figure() plt.plot(years, temp, 'b-o',
label='Original Data') plt.plot(futureyears, futuretemp1, 'g-o',
label='Linear Extrapolation') plt.plot(futureyears, futuretemp2, 'r-x',
label='PCHIP Extrapolation') plt.plot(futureyears, futuretemp3, 'k-d',
label='Spline Extrapolation') plt.xlabel('Year') plt.ylabel('Average
Temperature (°C)') plt.title('Extrapolation of Average Yearly
Temperatures') plt.legend() plt.grid(True) plt.show()
```



```
# Enter in command window : >> sum(temp)/n
```

```
# Find the orthagonal vectors
```

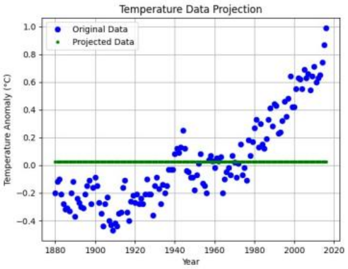
```
# Plot vectors in 11
```

```
# Calculate the average temperature average_temp = np.mean(temp) # Print the average
temperature print(f"The average temperature for the past 136 years was
{average_temp:.4f}°C.")
# Calculate orthogonal projection
n = len(temp) b1 = np.ones(n)
P1 = np.outer(b1, b1) / np.dot(b1, b1) temp1
= P1 @ temp
# Print projection matrix and projected temperatures
print("Projection matrix P1:") print(P1)
print("Projected temperatures (temp1):") print(temp1)
# Plot the results plt.figure() plt.plot(years, temp,
'bo', label='Original Data') plt.plot(years, temp1,
'g.', label='Projected Data') plt.xlabel('Year')
plt.ylabel('Temperature Anomaly (°C)')
plt.title('Temperature Data Projection') plt.legend()
plt.grid(True) plt.show()
```

```
The average temperature for the past 136 years was 0.0244°C.
Projection matrix P1:
[[0.00729927 0.00729927 0.00729927 ... 0.00729927 0.00729927 0.00729927]
 [0.00729927 0.00729927 0.00729927 ... 0.00729927 0.00729927 0.00729927]
 [0.00729927 0.00729927 0.00729927 ... 0.00729927 0.00729927 0.00729927]  ...
 [0.00729927 0.00729927 0.00729927 ... 0.00729927 0.00729927 0.00729927]
 [0.00729927 0.00729927 0.00729927 ... 0.00729927 0.00729927 0.00729927]
 [0.00729927 0.00729927 0.00729927 ... 0.00729927 0.00729927 0.00729927]]
 Projected temperatures (temp1):
[0.02437956 0.02437956 0.02437956 0.02437956 0.02437956 0.02437956
 0.02437956 0.02437956 0.02437956 0.02437956 0.02437956 0.02437956
0.02437956 0.02437956 0.02437956 0.02437956 0.02437956 0.02437956
0.02437956 0.02437956 0.02437956 0.02437956 0.02437956 0.02437956
0.02437956 0.02437956 0.02437956 0.02437956 0.02437956 0.02437956
0.02437956 0.02437956 0.02437956 0.02437956 0.02437956 0.02437956
0.02437956 0.02437956 0.02437956 0.02437956 0.02437956 0.02437956
0.02437956 0.02437956 0.02437956 0.02437956 0.02437956 0.02437956
0.02437956 0.02437956 0.02437956 0.02437956 0.02437956 0.02437956
0.02437956 0.02437956 0.02437956 0.02437956 0.02437956 0.02437956
0.02437956 0.02437956 0.02437956 0.02437956 0.02437956 0.02437956
0.02437956 0.02437956 0.02437956 0.02437956 0.02437956 0.02437956
0.02437956 0.02437956 0.02437956 0.02437956 0.02437956 0.02437956
```

```
0.02437956 0.02437956 0.02437956 0.02437956 0.02437956 0.02437956
0.02437956 0.02437956 0.02437956 0.02437956 0.02437956 0.02437956
0.02437956 0.02437956 0.02437956 0.02437956 0.02437956 0.02437956
0.02437956 0.02437956 0.02437956 0.02437956 0.02437956 0.02437956
0.02437956 0.02437956 0.02437956 0.02437956 0.02437956 0.02437956
0.02437956 0.02437956 0.02437956 0.02437956 0.02437956 0.02437956
0.02437956 0.02437956 0.02437956 0.02437956 0.02437956 0.02437956
```

```
 0.02437956 0.02437956 0.02437956 0.02437956 0.02437956 0.02437956
  0.02437956 0.02437956 0.02437956 0.02437956 0.02437956 0.02437956
  0.02437956 0.02437956 0.02437956 0.02437956 0.02437956]
```



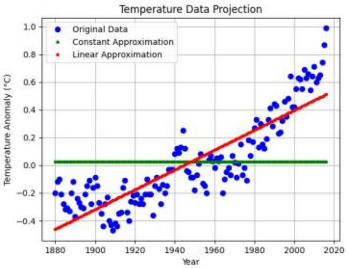
```
# Norm of P1^2 - P1
norm_P1 = np.linalg.norm (P1 @ P1 - P1 )
print(f"norm(P1 * P1 - P1) =  {norm_P1 :.4e}")
```

```
norm(P1 * P1 - P1) = 2.0132e-15
```

```
# Create the matrix B2 m
= len(years)
B2 = np.column_stack((np.ones(m), years)) #
Create the orthonormal basis using orth
Q2 = orth(B2) # Verify the ranks rank_Q2 =
np.linalg.matrix_rank(Q2) rank_Q2_B2 =
np.linalg.matrix_rank(np.column_stack((Q2, B2))) print(f"Rank of
Q2: {rank_Q2}") print(f"Rank of [Q2, B2]: {rank_Q2_B2}")
# Q3: What kind of matrix is Q2^T Q2? Why?
Q2_T_Q2 = np.dot(Q2.T, Q2) print("Q2.T @
Q2:") print(Q2_T_Q2)
```

```
Rank of Q2: 2
 Rank of [Q2, B2]: 2 Q2.T
 @ Q2:
 [[1.00000000e+00 7.07279798e-17]  [7.07279798e-17
 1.00000000e+00]]
```

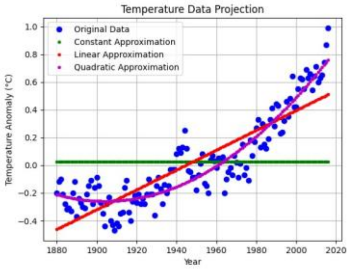
```
# Projection matrix onto the subspace S
P2 = Q2 @ Q2.T # Project the temperature data temp2 = P2 @ temp # Plot
the original and projected temperatures plt.figure() plt.plot(years,
temp, 'bo', label='Original Data') plt.plot(years, temp1, 'g.',
label='Constant Approximation') plt.plot(years, temp2, 'r.',
label='Linear Approximation') plt.xlabel('Year') plt.ylabel('Temperature
Anomaly (°C)') plt.title('Temperature Data Projection') plt.legend()
plt.grid(True) plt.show() # Display the projected temperatures
print("Projected temperatures (temp2):") print(temp2) # Norm of P2^2 -
P2 norm_P2 = np.linalg.norm(P2 @ P2 - P2)
print(f"norm(P2 * P2 - P2) =  {norm_P2 :.4e}")
```



```
Projected temperatures (temp2):
[-0.46194224 -0.45479045 -0.44763866 -0.44048687 -0.43333508 -0.42618328
 -0.41903149 -0.4118797  -0.40472791 -0.39757612 -0.39042433 -0.38327254
 -0.37612075 -0.36896895 -0.36181716 -0.35466537 -0.34751358 -0.34036179
 -0.33321    -0.32605821 -0.31890642 -0.31175463 -0.30460283 -0.29745104
 -0.29029925 -0.28314746 -0.27599567 -0.26884388 -0.26169209 -0.2545403
 -0.2473885  -0.24023671 -0.23308492 -0.22593313 -0.21878134 -0.21162955
 -0.20447776 -0.19732597 -0.19017417 -0.18302238 -0.17587059 -0.1687188
 -0.16156701 -0.15441522 -0.14726343 -0.14011164 -0.13295984 -0.12580805
 -0.11865626 -0.11150447 -0.10435268 -0.09720089 -0.0900491  -0.08289731
 -0.07574551 -0.06859372 -0.06144193 -0.05429014 -0.04713835 -0.03998656
 -0.03283477 -0.02568298 -0.01853119 -0.01137939 -0.0042276   0.00292419
  0.01007598  0.01722777  0.02437956  0.03153135  0.03868314  0.04583494
 0.05298673  0.06013852  0.06729031  0.0744421   0.08159389  0.08874568
 0.09589747  0.10304927  0.11020106  0.11735285  0.12450464  0.13165643
 0.13880822  0.14596001  0.1531118   0.1602636   0.16741539  0.17456718
 0.18171897  0.18887076  0.19602255  0.20317434  0.21032613  0.21747792
 0.22462972  0.23178151  0.2389333   0.24608509  0.25323688  0.26038867
 0.26754046  0.27469225  0.28184405  0.28899584  0.29614763  0.30329942
 0.31045121  0.317603    0.32475479  0.33190658  0.33905838  0.34621017
 0.35336196  0.36051375  0.36766554  0.37481733  0.38196912  0.38912091
 0.39627271  0.4034245   0.41057629  0.41772808  0.42487987  0.43203166
  0.43918345  0.44633524  0.45348703  0.46063883  0.46779062  0.47494241
  0.4820942   0.48924599  0.49639778  0.50354957  0.51070136] norm(P2 *
P2 - P2) = 5.6043e-16
```

```
# Approximate using quadratic function
```

```
# Add a column of squared years to the matrix B3
B3 = np.column_stack((np.ones(m), years, years**2))
# Create the orthonormal basis Q3 using the orth function
Q3 = np.linalg.qr(B3)[0] # Using QR decomposition to get an orthonormal basis
# Projection matrix onto the quadratic subspace
P3 = Q3 @ Q3.T
# Project the temperature data onto the quadratic subspace temp3
= P3 @ temp # Plot the original and projected temperatures
plt.figure() plt.plot(years, temp, 'bo', label='Original Data')
plt.plot(years, temp1, 'g.', label='Constant Approximation')
plt.plot(years, temp2, 'r.', label='Linear Approximation')
plt.plot(years, temp3, 'm.', label='Quadratic Approximation')
plt.xlabel('Year')
plt.ylabel('Temperature Anomaly (°C)')
 plt.title('Temperature Data Projection')
 plt.legend() plt.grid(True) plt.show() #
 Display the projected temperatures
 print("Projected temperatures (temp3):")
 print(temp3) # Norm of P3^2 - P3 norm_P3 =
 np.linalg.norm(P3 @ P3 - P3) print(f"norm(P3
 * P3 - P3) = {norm_P3:.4e}")
```



```
Projected temperatures (temp3):
[-0.21200149 -0.2158765  -0.21958814 -0.22313643 -0.22652136 -0.22974292
 -0.23280113 -0.23569598 -0.23842746 -0.24099559 -0.24340036 -0.24564176
 -0.24771981 -0.2496345  -0.25138582 -0.25297379 -0.2543984  -0.25565965
 -0.25675753 -0.25769206 -0.25846323 -0.25907104 -0.25951548 -0.25979657
 -0.2599143  -0.25986866 -0.25965967 -0.25928732 -0.25875161 -0.25805253
 -0.2571901  -0.25616431 -0.25497516 -0.25362265 -0.25210677 -0.25042754
 -0.24858495 -0.246579   -0.24440968 -0.24207701 -0.23958098 -0.23692159
 -0.23409884 -0.23111272 -0.22796325 -0.22465042 -0.22117423 -0.21753468
 -0.21373176 -0.20976549 -0.20563586 -0.20134287 -0.19688652 -0.19226681
 -0.18748373 -0.1825373  -0.17742751 -0.17215436 -0.16671785 -0.16111798
 -0.15535474 -0.14942815 -0.1433382  -0.13708489 -0.13066822 -0.12408819
 -0.1173448  -0.11043804 -0.10336793 -0.09613446 -0.08873763 -0.08117744
 -0.07345389 -0.06556698 -0.05751671 -0.04930308 -0.04092608 -0.03238573
 -0.02368202 -0.01481495 -0.00578452  0.00340927  0.01276642  0.02228693
       0.0319708   0.04181803  0.05182862  0.06200257  0.07233988  0.08284056
      0.09350459  0.10433198  0.11532273  0.12647684  0.13779431  0.14927514
      0.16091933  0.17272688  0.18469779  0.19683206  0.20912969  0.22159068
      0.23421503  0.24700274  0.25995381  0.27306824  0.28634603  0.29978718
      0.31339169  0.32715956  0.34109079  0.35518538  0.36944333  0.38386464
      0.39844931  0.41319734  0.42810873  0.44318348  0.45842159  0.47382306
      0.48938789  0.50511608  0.52100763  0.53706254  0.55328081  0.56966243
  0.58620742  0.60291577  0.61978748  0.63682255  0.65402098  0.67138277
  0.68890792  0.70659643  0.7244483   0.74246353  0.76064212]
norm(P3 * P3 - P3) = 7 8790e-16
```