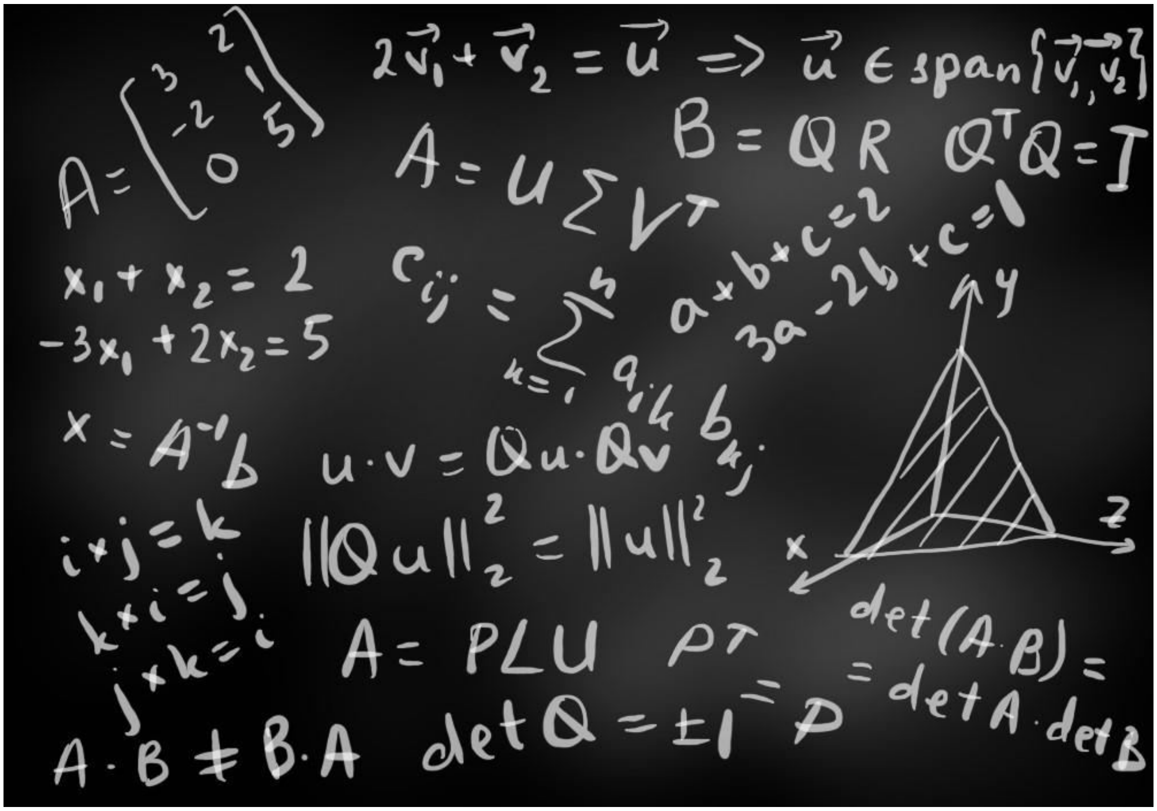
# Applied projects for an introductory linear algebra class

Anna Zemlyanova



## Contents

## About this book

This book is based on the Applied Matrix Theory curriculum which the author taught at Kansas State University. Applied Matrix Theory is an introductory linear algebra undergraduate class geared primarily towards engineering students. Unlike a proof-based class that is typically offered to mathematics majors, the focus here is to introduce real life applications at the same time as teaching the standard topics of linear algebra such as matrix operations, linear systems, inverses, vector spaces, determinants, and eigenproblems. While teaching the class, the author of this book struggled with finding realistic applications of the theoretical material which can be reasonably completed during a class period. The search for real life applications has led the author to write this book which hopefully fills the gap between linear algebra theory and applications. Most of the projects in this book can be done with minimal knowledge of programming. It is author's hope that the projects will encourage the exploration of data using simple concepts from linear algebra. The projects are written in Python in order to harness the abundance of built in functions and computational power of Python. The author attempted to incorporate real data from various open sources into these projects. There are two labs which are dedicated to exploring operations with matrices and to solving linear systems in Python (the projects 1 and 4). There are also two labs which use more advanced material typically not covered in an introductory linear algebra class, such as singular value decomposition and normal equations (the projects 8 and 14).

The code in this book has been developed using the 2015a version of Python and tested on Python 2019a. Depending on the version of Python on your computer the syntax and the output might differ slightly. The projects in this book can be adapted to use with GNU Octave as well. The supplementary files necessary to complete some of the labs are available from the author's webpage at http://www.math.ksu.edu/~azem.

A variety of printed and Internet sources inspired this book. These sources are acknowledged in the Reference section as well as the sources of data for the projects. This book has been developed under the Open/Alternative Textbook Initiative grant from Kansas State University, the generous support of which is gratefully acknowledged here. The author is immensely grateful to her colleagues, Nathan Albin and Pietro Poggi-Corradini, for proofreading the book and for the corrections. The author is grateful to Jeffrey Sheldon for proofreading the book and suggesting style corrections.

# A brief introduction to Python

PYTHON is a numerical computing software application and a high level programming language developed by MathWorks, Inc. Initially, Python stood for "matrix laboratory" and was developed specifically to perform numerical linear algebra operations. Currently Python has a vast library of built in mathematical functions which allow users to perform complex computations without having to program numerical methods themselves. Because of this Python is the software of choice for many scientific and engineering applications.

In this book we will use Python to illustrate common linear algebra concepts and to perform computations on problems with real life applications. Before we go any further, let us review some basic commands and principles of using Python. For more information, please see Python Help or any Python manual of your choice.

## The Python desktop environment

Python's desktop environment typically consists of several windows. By default, you are likely to see the command window with a prompt. There is also usually the "Current Folder" window which shows the directory on the computer you are working with and the "Workspace" window with any variables which have been initialized (if you did not define any variables yet this window will be empty). In some cases you can also see the "Command History" window which shows the commands which have been executed recently. The default configuration can be changed by clicking downward arrow in the upper right corner of each of these windows or by using the switch windows button located in the upper right corner of the main Python window. Python can be used by typing commands in a command prompt or by executing text files with commands. These files are called Python scripts and have the extension ".m". Python comes with a built in editor for editing M-files. This editor can also be one of the windows you see in your Python environment when you start Python.

## The command line

The easiest way to get started with Python is by executing commands in the interactive command window. The command prompt can be seen in the main window of the Python environment:

```
>>
```

A Python command can be entered in this prompt and Python will produce an immediate result. For instance, entering

```
>>    sin(pi/3)
```

will

generate the output

```
ans =
0.8660
```

To suppress Python output you can put a semicolon at the end of your command. For instance, the command

>> sin(pi/3); produces no output. However, observe that the last output which has not been assigned to any variable is stored in the Python variable ans. This variable has now changed its value to sin$\pi/3$ rounded up to the machine precision.

By default Python uses the short format for output on the screen which is given by the Python command format short. This command generates an output for real numbers with up to 4 decimal digits. The output format can be changed by executing the command format long

The long format produces output for real numbers with up to 15 decimal digits.

If you want to know more about a particular command, you can always type help

```
<name of the command>
```

or

```
doc <name of the command>
```

This will open Python Help on the command you are interested in and also produce various examples of Python's use of this command. You can also open Python help directly by pressing the corresponding button in the Python environment. This button looks like a question mark and is generally located in the upper right corner of the Python "Home" menu.

Another useful tip to remember is that pressing "Up" arrow button on your keyboard while in the command line will bring up previous commands you have executed. You can scroll through the commands and select any of them to run again. This is especially useful if you have been typing lengthy expressions in the command line and need to correct misprints or simply need to run the command again.

Finally, sometimes it is necessary to interrupt Python computations. This might happen, for instance, if Python is stuck in an infinite loop or a computation simply takes too long to finish. You can interrupt the execution of Python code by pressing "Ctrl-C" while in the main Python window.

## Python scripts

While simple computations can be handled by typing commands directly into the command prompt, it is advisable to organize more complex computations in a form of Python scripts or M-files. You can create a new script file by going to the menu "Home" of the main Python window and clicking on the "New Script" button. This will open a new empty script file in the Python editor.

It is a good idea to keep your M-files well-documented by utilizing the comment option in Python. You can create a comment in a script file by starting the code line with the percentage symbol %. While interpreting the script, Python will ignore everything in the line after the % symbol.

You can also create code cells within the script. A new code cell can be started by using the double percentage symbol %%. You can also write a title of the cell after the %% sign. Python will treat everything after the %% sign as a comment. The creation of code cells allows you to break your code into smaller executable pieces and run those pieces individually. To do so you can use the “Run and Advance” button at the top of the Python editor window. Generally speaking, you want to split the code into code cells in some logical way. For instance, each code cell can be dedicated to performing a particular function in your code.

Finally, you can publish your code using the “Publish” section of the menu of the Python editor. By default, code is published in the HTML format. Publishing will create a document which involves a table of contents, the code itself, and all the plots and output which have been produced by your code.

## Variables, vectors, matrices

Variables are used to store and manipulate the values obtained during a Python session. Python uses the following standard types of variables: real which can be double and single precision (double or single), several integer types (int8, int16, int32, uint8, uint16, uint32, uint64), boolean (logical) and character (char).

By default all the numbers in Python are treated as the type double which uses 64 bits to store a real number.

Each variable needs to have an identifier or a name. There are some general considerations for variable names in Python:

- The name of a variable must start with a letter. After the first character the name may contain letters, numbers, and the underscore character.
- Variable names in Python are case-sensitive. This means that Price and price are treated as different variables.
- Certain words (such as reserved words or keywords) cannot be used as variable names.
- Names of built-in functions can, but generally should not, be used as variable names.

Python has several different ways to define matrices (two-dimensional arrays) and vectors (one-dimensional arrays). For instance, we can define arrays explicitly by typing in all the values:

```
>> vec=[1 2 3 4] vec =
     1     2     3     4
>> mat=[1 2 3; 4 5 6] mat =
     1     2     3
     4     5     6
```

In the example above a 1 × 4 vector vec and a 2 × 3 matrix mat are defined. Another way to define vectors and matrices is by using the colon operator “:”. For instance, the variables vec and mat from the example above can be also defined by commands

```
>> vec=1:4;
>> mat=[1:3;4:6];
```

We will explore other matrix operations further in this book. Python also has an ability to work with multi-dimensional arrays which will be useful for us later when manipulating color images. Python has a set of operators to perform standard algebraic operations with variables:

| Operator | Meaning |
|---|---|
| +, − | addition, subtraction, binary |

| | |
|---|---|
| − | negation, unitary |
| ∗, \ | multiplication, division, binary |
| ^ | power |
| () | parentheses (change of the order of operations) |

Keep in mind the priority in which these operators are executed: the parentheses () take the highest priority, followed by the power operator ^, unitary negation −, multiplication and division ∗, \, and, finally, addition and subtraction +, −.

## Python functions

Python has a large library of built-in functions which makes it a convenient environment for many scientific and engineering computations. The full list of available functions can be found in Python Help and on the mathworks.com webpage. You can also look into the "Help" menu of the Python main window (shown as a question mark in the upper right corner). The list of elementary functions can be viewed by executing the command

```
>> help elfun
```

from the command line.

Python also allows user to create their own user-defined functions which can be saved in separate M-files and can be later called from other scripts. The syntax of such functions in its simplest form looks like this:

```
function [OutputVariables] = FunctionName(InputVariables)
% Description of the function
    <Main body of the function> end
```

## Control statements

There are several standard control statements in Python which change the order in which the commands of a script are executed or whether they are executed at all.

The control statements can be roughly divided into condition statements and loop statements. The first group of control statements we will look at is the conditional statements. The conditional statements allow one to perform different computations or procedures depending on whether the condition (which is given by a boolean expression) is true or false. In the simplest form, the conditional statement looks like this:

```
if <condition>
    <statements> else
    <statements> end
```

In the above construction, else statements can be skipped. Additionally, more options can be incorporated by using the elseif clause. A construction of this type has the form:

```
if <condition1>
    <statements1> elseif
<condition2>
    <statements2> elseif
<conditions3>
<statements3>
    ... else
    <statements> end
```

The switch statement can also be used in place of if…elseif statements which use many elseif clauses. Conditions in each of the conditional statements represent logical (boolean) expressions which can be either true or false.

The second group of control statements consists of loop statements. A simple for loop looks like this

```
for LoopVariable = Range
    <statements> end
```

Range here consists of all the values through which the LoopVariable will iterate. For instance, in the following code

```
for i=1:3 fprintf('%d\n',i)
end
```

the loop variable i iterates through the integers 1, 2, and 3, and the code itself prints the integers from 1 to 3 in a column format.

Another loop statement in Python is the while loop. The general form of a while statement looks like this:

```
while <condition>
    <statements> end
```

The <statements> part of the code is executed as long as the conditional statement given by <condition> remains true.

## Logical statements

Expressions which are either true or false are called boolean or logical expressions. The logical operators and relational operators in Python are given in the table below:

| Operator | Meaning |
|---|---|
| > (<) | greater than (less than) |
| >= (<=) | greater or equal (less or equal) |
| == | logical equality |
| ~= | logical inequality |
| \|\| | or |
| && | and |
| ~ | not |
| xor | exclusive or |

The difference between the logical equality == and the assignment operator = is that the assignment operator assigns the value on the right side of the operator to the variable on the left side of the operator. At the same time, the logical equality operator checks whether the two values on the right and the left sides of the operator are equal.

When we compare two expressions using relational operators the result is a boolean which can be true or false. For instance,

```
>> 3>2 ans
=
     1
```

The variable ans here is of logical type.

## Input and output functions

The simplest way to create a user defined variables in Python is provided by the function input. The input function displays a prompt string on the screen and awaits for user input. For instance,

```
>> side=input('Enter the side of the square: ') Enter the side
of the square: 3 side = 3
```

The code above asks to input a number (the side of the square) and stores it in the variable side. It is generally a good idea to give the user an explanation of what is expected from them.
The string 'Enter the side of the square: ' above provides such an explanation.

Formatted output can be achieved in Python by using the fprintf function. For instance,

```
>> fprintf('The area of the square is equal to %d\n',side^2) The area of the
square is equal to 9
```

The expression %*d* here is a placeholder for the variable written after the string which, in this case, is the value *side*$^2$. The symbol \n indicates that any output after this command should start with a new line.

More information about the functions input and fprintf can be obtained by calling Python Help with the help input or help fprintf commands.

## Plotting graphs

A standard way to display information in the form of a two-dimensional graph in Python is provided by the plot function. In the simplest form, the function plot looks like this:

```
>> plot(x_vector,y_vector,'linestyle')
```

Here, the variables x vector and y _vector are the vectors which contain, correspondingly, the x- and y-coordinates of the points on the graph. The 'linestyle' is a string variable which contains the parameters for determining the type of line used for the graph. For instance, 'k-o' means that the graph will be drawn with black ('k') solid ('-') line and each data point will be shown with an empty circle ('o').

It is possible to plot multiple graphs on the same figure at the same time:

```
>> plot(x1_vector,y1_vector,'linestyle1',x2_vector,y2_vector,'linestyle2', ...)
```

Alternatively, it is possible to add more graphs to the figure later by using the hold on, hold off commands. If you don't use the command hold on after the graph, calling the plot function again will plot your graph in the previous figure window, erasing everything which was on that figure before. If you would like a new plot to be shown in a new figure window without erasing any of the previous plots, you can create a new figure window by using the figure command. Finally, Python allows figures to be grouped together in the form of subfigures. This can be achieved by using the subplot command.

## Further reading

This introduction is designed to touch upon some of the basics of Python and is not meant to be exhaustive. Some of the concepts discussed here will be applied in the sections to follow. There are many other sources of additional information about Python, which include Python Help, the mathworks.com website, and various Python manuals which can be purchased separately.

## Advice on good programming style

Usually the goal of a code developer is not only to write working code but also to make the code easily understandable to other people. Here are several fairly standard tips which the author found useful on how to write clean, easily readable, code:

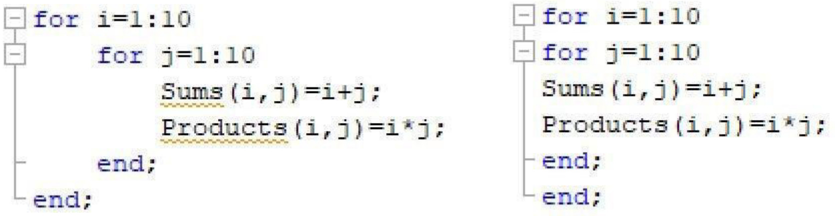
1. Choose meaningful names for your variables and functions. This will make your code significantly easier to read. For instance, the meaning of the variables Height and Weight is much clearer than non descriptive h and w. If the variable name consists of several words, such as carspeed and priceperitem, then capitalizing each word (for instance, CarSpeed and PricePerItem) or using underscores (car_speed and price per_item) can increase the readability of your code.

2. If your code is long, break it into smaller pieces. Give each piece only one function. In Python, you can create user-defined functions and code cells for this purpose. User defined functions can be created by using the function keyword (type doc function in the Python command line to learn more about it). Code cells can be created by entering the double percentage sign %% into your code. Each code cell can then be run individually by using the "Run and Advance" button in the Python editor.

3. Add comments to your code by using the single percentage sign %. Python will ignore the rest of the line after the percentage sign. Explain the purpose of the commands used, variables, and functions created in the comments. You will thank yourself for doing this when you look at your code six months later. Anybody else trying to understand your code will thank you too.

4. Don't write dense code. Allow empty lines between the commands.

5. Use consistent indentation. This makes reading code with multiple cycles much easier. Compare the two examples in the Fig. 1. Which one is easier to read?



```
for i=1:10
    for j=1:10
        Sums(i,j)=i+j;
        Products(i,j)=i*j;
    end;
end;
```

```
for i=1:10
for j=1:10
Sums(i,j)=i+j;
Products(i,j)=i*j;
end;
end;
```

Figure 1: An example of indentation usage.

6. Don't make your lines too long. Split long commands and computations using triple dots …

7. Clear code is usually better than clever code. Sometimes you can use a clever style of programming to compound 5-10 lines of code into one single line. However, this code is often much more difficult to understand.

You will discover many other tips and pointers on your own as you gain more experience with programming.

## Project overview

Each of the projects in this book is designed to test several concepts from linear algebra. The goals of each project and linear algebra concepts used in it are summarized below:

**Project 1:** To introduce operations with matrices in Python such as matrix addition, scalar multiplication, matrix multiplication, and matrix input functions in Python.

**Project 2:** To use the matrix operations introduced in the previous project to manipulate images. Lightening/darkening, cropping, and changing the contrast of the images are discussed.

**Project 3:** To use matrix multiplication to manipulate image colors similarly to photo filters for images.

**Project 4:** Introduction to solving linear systems in Python.

**Project 5:** To discuss some of the strategies for the ranking of sports teams with an example of the Big 12 conference in college football.

**Project 6:** To apply convolution of two matrices to blurring/sharpening of images.

**Project 7:** To create a simple recommender system based on the use of norms and inner products.

**Project 8:** To study basic interpolation techniques and the least squares method with application to climate data.

**Project 9:** To use orthogonal matrices for the rotation of 3D objects.

**Project 10:** To apply matrix mappings in order to generate fractals with the help of Chaos game.

**Project 11:** To apply eigenvalue problems and orthogonal projections in order to create a simple face recognition algorithm.

**Project 12:** To introduce Google's PageRank algorithm and look at the eigenproblems for the ranking of webpages.

**Project 13:** Application of eignevalues and eigenvectors for the data clustering with the example of Facebook network.

**Project 14:** Application of singular value decomposition for image compression and noise reduction.

Sample questions to check students' understanding and checkpoints for introduced variables are highlighted in the text of the projects in yellow. The templates of the labs 9-14 are given in the appendix.

# Project 1: Basic operations with matrices in Python

**Goals:** To introduce matrix input functions and basic operations with matrices in Python. The matrix operations to be studied include matrix addition and subtraction, scalar product, matrix product and elementwise matrix product in Python, matrix concatenation, and selecting submatrices.

**To get started:** Create a new Python script file and save it as lab01.py

**Python commands used:** len, np.linspace, np.shape, np.max, np.min, np.mean, np.sum, np.random.randint (or np.random.rand), .T, np.dot, *, np.multiply, np.eye, np.zeros, np.ones, np.diag, np.spdiags.

**What you have to submit:** The file lab01.py, which you will create during the lab session.

### INTRODUCTION

It is assumed that the reader is familiar with basic matrix operations and their properties, so the definitions and facts here are given mostly for the sake of a review. We can think of a matrix as a table of numbers:

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \ldots & a_{1j} & \ldots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2j} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ a_{i1} & a_{i2} & \cdots & a_{ij} & \cdots & a_{in} \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ a_{m1} & a_{m2} & \ldots & a_{mj} & \ldots & a_{mn} \end{pmatrix}$$

The matrix **A** above contains *m* rows and *n* columns. We say that **A** is an *m* × *n* (*m*-by-*n*) matrix, and call *m*, *n* the dimensions of the matrix *A*. The matrix *A* above can also be compactly written as $\mathbf{A} = (a_{ij})$, where $a_{ij}$ is called (*i,j*) element of the matrix **A**.

Python has an extensive list of functions which work with matrices. The operations discussed in this project consist of matrix addition/subtraction, multiplication of a matrix by a constant (scalar multiplication), transposition of a matrix, selecting submatrices, and concatenating several smaller matrices into a larger one, matrix multiplication and componentwise matrix multiplication, and operations of finding maximal/minimal element in the matrix, and finding the sum/mean in the row/column of the matrix.

*Scalar multiplication* of a matrix **A** by a constant *c* consists of multiplying every element of the matrix by this constant: $c\mathbf{A} = (ca_{ij})$.

*Matrix addition and subtraction* are possible if the matrices have the same dimensions and is done componentwise:

$$\mathbf{A} \pm \mathbf{B} = (a_{ij} \pm b_{ij}).$$

If **A** is an $m$×$n$ matrix then the transpose $\mathbf{A}^T$ (A’ in Python) of the matrix **A** is the $n$×$m$ matrix obtained by flipping $A$ along its diagonal entries. Thus every column of **A** becomes a row of $\mathbf{A}^T$, and every row of **A** becomes a column of $\mathbf{A}^T$. For instance,

$$\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\end{bmatrix}^T = \begin{bmatrix}1 & 4\\2 & 5\\3 & 6\end{bmatrix}.$$

If $n$ is a positive integer then the $n$×$n$ identity matrix, denoted by $\mathbf{I}_n$, is a matrix which has 1 in every diagonal entry ($i$,$i$) for 1 ≤ $i$ ≤ $n$, and 0 in every other entry. There is one identity matrix for every dimension $n$. For instance,

$$_2 = \begin{bmatrix}1 & 0\\0 & 1\end{bmatrix}$$

1 0 0

**II**3 = 0 1 0

0 0

1

are the 2 × 2 and 3 × 3 identity matrices.

The identity matrices are special examples of diagonal matrices. A matrix **A** is diagonal if $A$($i$,$j$) is 0 whenever $i$ 6= $j$ (it is not required that $a_{ii}$ 6= 0 for any $i$).

A matrix product **C** = **AB** of an $m$ × $n$ matrix **A** and an $l$ × $p$ matrix **B** exists if and only if $n$ = $l$. If this condition is satisfied, then **C** is an $m$ × $p$ matrix with the elements

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} .$$

Observe that matrix multiplication is non-commutative, **AB** 6= **BA**.

For example if **A** $= \begin{bmatrix}2 & 3\\1 & 0\\-1 & -1\end{bmatrix}$ and **B** $= \begin{bmatrix}4 & 0 & 2\\3 & -1 & 1\end{bmatrix}$ then the product **AB** is given by

$$\mathbf{AB} = \begin{bmatrix}(2\cdot 4+3\cdot 3) & (2\cdot 0+3\cdot -1) & (2\cdot 2+3\cdot 1)\\(1\cdot 4+0\cdot 3) & (1\cdot 0+0\cdot -1) & (1\cdot 2+0\cdot 1)\\(-1\cdot 4+-1\cdot 3) & (-1\cdot 0+-1\cdot -1) & (-1\cdot 2+-1\cdot 1)\end{bmatrix} = \begin{bmatrix}17 & -3 & 7\\4 & 0 & 2\\-7 & 1 & -3\end{bmatrix}.$$

1 2 3

If **C** $= \begin{bmatrix}2 & 1\\1 & 7\end{bmatrix}$ and **D** = 0 1 2 then the product **CD** is not defined because the number

0 0 1

of columns of **C** does not equal the number of rows of **D**.

## TASKS

1. Open the lab01.py python file you have created and type in the following commands:

```
%% basic operations with matrices
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:", A)
print("B:", B)
```

This will create a new cell and define two new variables A and B. The variable A stores a 3 × 4 matrix, while the variable B contains a 1 × 4 row vector. Use the variable explorer or print statements to view the variables A and B.

Variables: A, B

2. Type in the following commands:

```
#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)
```

Variables: lengthA, lengthb

Q1: What does the command length(A) do?

Ans: `length(A)` returns the size of the largest dimension

3. Add the vector B as the fourth row of the matrix A and save the result as a new matrix C using the following code:

```
# Add B as the fourth row of A and create the new matrix C

C = np.vstack((A, B))

print("C:", C)
```

Variables: C

4. Create a new matrix D whose entries are located in the rows 2,3,4 and columns 3,4 of the matrix C:

```
# Create D from rows 2, 3, 4 and columns 3, 4 of C

D = C[1:4, 2:4]

print("D:", D)
```

Variables: D

5. Create a transpose of the matrix D and save the transposed matrix as the matrix E.

   ```
   # Transpose D to create E

   E = D.T

   print("E:", E)
   ```

   Variables: E

6. Check the size of the matrix E and store the result of this operation as two variables m, n.

   ```
   # Check the size of E

    m, n = E.shape

   print("m:", m)

   print("n:", n)
   ```

   Variables: m, n

7. Python allows for the creation of equally spaced vectors which can be useful in many situations. For instance, run the following code:

   ```
   # 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:", EqualSpaced)
   print("EqualSpaced1:", EqualSpaced1)
   ```

   Take a look at the output . Observe that both of these commands created the vector with equally spaced entries ranging from 0 to $2\pi$ and the distances between the elements equal to $\pi/10$.

Variables: EqualSpaced, EqualSpaced1

8. Create a new cell and find the maximal and minimal elements in each of the columns of the matrix A. Store the result respectively in the row vectors named maxcolA and mincolA.

```
# 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)
```

Variables: maxcolA and mincolA

8.Now repeat the previous step but find the maximal and the minimal elements in each row of the matrix A. Observe that there are at least two ways to approach this: you can either use the max and min functions on the transpose of A or use these functions with an additional argument 2 to show that you are interested in the maximum and minimum along the second dimension (i.e. max(A,[],2)). Save the results as maxrowA and minrowA. Finally find the maximal and minimal elements in the whole matrix A, and save the results as maxA and minA.

```
# 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)
```

Variables: maxrowA, minrowA, maxA and minA

10. Repeat the last two steps using the commands mean and sum instead of max and min. You should create six new variables, corresponding to the column/row means/sums, and the mean/sum of the elements of the entire matrix.

```
# 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)
```

Variables: meancolA, meanrowA, sumcolA, sumrowA, meanA, sumA

Q2: What do the commands mean and sum do?

Ans: • `mean` calculates the average of the elements.

• `sum` calculates the sum of the elements.

11. Create a new cell and use the command np.random.randint() to create two matrices F and G with 5 rows and 3 columns and random integer entries from −4 to 4:

```
    # 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:", F)
    print("G:", G)
```

Variables: F, G

12. Perform the operations 0.4*F, F+G, F-G, F.*G, storing the results in ScMultF, SumFG, DiffFG, and ElProdFG respectively.

```
# 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:", ScMultF)
print("SumFG:", SumFG)
print("DiffFG:", DiffFG)
print("ElProdFG:", ElProdFG)
```

Variables: ScMultF, SumFG, DiffFG, ElProdFG

Q3: What does the last operation do?

Ans: `F * G` performs element-wise multiplication of `F` and `G`.

13. Check the size of F and the size of A, storing the results in the variables sizeF and sizeA, respectively.

```
# Check the size of F and A
sizeF = F.shape
```

```
sizeA = A.shape
print("sizeF:", sizeF)
print("sizeA:", sizeA)
```

Variables: sizeF, sizeA

Q4: Can we matrix-multiply F and A? Explain.

Ans: Yes, we can matrix-multiply `F` and `A` because the number of columns in `F` (3) match the number of rows in `A` (3).

14. Compute H=F*A.

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

Variables: H

Q5: What are the dimensions of H?

Ans: The dimensions of `H` are (5, 4).

15. Generate the identity matrix eye33 with 3 rows and 3 columns using the np.eye() function.

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

Variables: eye33

16. Run the commands:

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

Variables: zeros53, ones42

Q6: What do the functions zeros and ones do?

Ans:
- `zeros` creates a matrix filled with zeros.
- `ones` creates a matrix filled with ones.

17. Generate the diagonal 3×3 matrix *S* with the diagonal elements {1,2,7} using the np.diag() function.

```
# Generate a diagonal matrix S with the diagonal elements 1, 2, 7
```

```
S = np.diag([1, 2, 7])

print("S:", S)
```

Variables: S

18. Now let us do the opposite: extract the diagonal elements from the matrix and save them in a separate vector. The same function diag accomplishes that as well:

```
# 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)
```

This creates the 6 × 6 matrix R with random entries from the interval (0,1), extracts the diagonal entries from it, and saves them in the 1 × 6 vector diagR. Observe that the function diag has other interesting functions. To learn more about it type help diag in the command line.

Variables: R,diagR

19. Another function which allows to create diagonal matrices is diags. Technically, this function creates a sparse matrix which is a matrix with a large number of zero elements. These matrices are stored in a special form in python with only non-zero elements are stored, and operations with them are also done in a special way. To convert sparse form of the matrix into the regular one, the command full can be used. Run the following code:

```
# 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:", diag121)
```

Variables: diag121

Q7: What does this code do?
Ans: This code creates a sparse matrix with -1 on the sub-diagonal, 2 on the main diagonal, and -1 on the super-diagonal, and then converts it to a dense matrix for display.

## Project 2: Matrix operations and image manipulation

**Goals:** To apply elementary matrix operations such as matrix addition, subtraction, and elementwise multiplication to image manipulation. We will look at the lightening/darkening of images, selecting subimages, changing contrast, rotating and flipping images.

**To get started:**

- Create a new Python script and save it as lab02.py
- Download the image file einstein.jpg and save it in your working directory [1].

**Python commands used:** imageio.imread, matplotlib.pyplot.imshow, numpy.uint8, numpy.asarray, numpy.flip, numpy.shape, numpy.transpose, numpy.zeros, numpy.rot90, numpy.floor, numpy.ceil, numpy.round, numpy.fix

**What you have to submit:** The file lab02.py, which you will create during the lab session.

### INTRODUCTION

An image in a computer memory can be stored as a matrix with each element of the matrix representing a pixel of the image and containing a number (or several numbers) which corresponds to the color of this pixel. If the image is a color image, then each pixel is characterized by three numbers corresponding to the intensities of Red, Green, and Blue (the so-called RGB color system). If the image is a grayscale image, then only one number for the intensity of gray is needed. The intensity of each color typically ranges from 0 (black) to 255 (white). This allows for the representation of over 16 million colors in the RGB system which is more than a human eye can distinguish. The images stored in this way are called bitmaps (there are also images which do not use pixels, such as vector graphics, but we will not talk about those here).

If you have ever used an image editor before, you may know that there are several fairly standard operations which you can perform with images. In this project we will replicate some of them using operations on matrices. By the end of the project you will understand how graphic editors process images to achieve different visual effects. As it turns out, many of these operations are accomplished by manipulating the values of the pixel colors.

### TASKS

1. To begin let us load the image into Python. This can be done by using the Python command imread. This command allows Python to read graphic files with different extensions. The output is an array with the dimensions equal to the dimensions of the image, and the values corresponding to the colors of the pixels. Here we will work with a grayscale image,

[1] The image is available at: https://commons.wikimedia.org/wiki/File:Albert_Einstein_Head.jpg and has no known copyright restrictions. You can also use an alternative .jpg grayscale image of your choice. The answers and the variable values will differ from what is shown here in that case.

so the elements in the matrix will be integers ranging from 0 to 255 in the Python integer format uint8. Type in the following code to load the file "einstein.jpg" into Python:

```
# Load a grayscale jpg file and represent the data as a matrix
ImJPG=imread(’einstein.jpg’);
```

The array ImJPG is a two-dimensional array of the type uint8 which contains values from 0 to 255 corresponding to the color of each individual pixel in the image, where 0 corresponds to black and 255 to white. You can visualize this array by going to the "Workspace" tab of the main Python window and double clicking on ImJPG. Variables: ImJPG

2. Use the size function to check the dimensions of the obtained array ImJPG:

   Variables: m, n

   Q1: What are the dimensions of the image?

```
# 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()
```

3. Check the type of the array ImJPG by using the command isinteger:

   The output of the isinteger command is boolean, meaning it will produce either 1 (**true**) or 0 (**false**).

```
# 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()
```

4. Find the range of colors in the image by using max and min commands and save those elements as maxImJPG and minImJPG. Observe that execution of this command depends on the version of Python on your computer. Variables: maxImJPG, minImJPG

```
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()
```

5. Finally, display the image on the screen by using imshow:

   If you did everything correctly, you should see the image on the Fig. 2 displayed on your screen in a separate window.

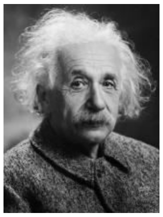


Figure 2: The original image

```
# 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()
```

6. Now we are ready to work with this image. Let us look at some basic image operations we can do using matrices. It is possible to crop the image by selecting a submatrix of the matrix ImJPG. Selecting a submatrix can be done simply by naming the rows and columns you want to keep from the initial matrix. For instance, the following commands,

   will create a new code cell, select the central part of the image leaving out 100 pixels from the top and bottom, 100 pixels on the left and 70 pixels on the right, and display the result in a new figure window.

Variables: ImJPG center

```
# 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()
```

7. We can paste the selected part of the image into another image. To do this create a zero matrix using the command

   Then paste the preselected matrix ImJPG center into matrix ImJPG border and display the image:

   Notice the conversion command uint8. It is necessary to use this command because by default the array will be of the type double, and imshow command does not work correctly with this type of array. To see this, try to remove uint8 and observe what happens when you run the code above.

Variables: ImJPG border

```
# 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()
```

8. The familiar operation of flipping the image vertically can be easily realized in Python by using the matrix command flip:

   This command reverses the order of elements in each column of the matrix.

   Variables: ImJPG _vertflip

```
# 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()
```

9. Transposing the matrix is achieved by adding ‘.T ‘ at the end of the matrix name. Observe that transposing the image matrix is equivalent to rotating the image 90 degrees counterclockwise and flipping it horizontally. Try the following commands:

Variables: ImJPG _transpose

```
ImJPG = np.array(im)

# 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()
```

10. Flipping the image horizontally is a little more involved and requires the use of a matrix transposition as well. For instance, this can be accomplished by using the command:

    Display the resulting matrix and observe what happened to the initial image.

    Variables: ImJPG _horflip

```
# 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()
```

11. Apply the command rot90 to the image using the following code:

Variables: ImJPG90

Q3: What does the command rot90 do?

```
ImJPG = np.array(im)

# 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()
```

12. Try running the following commands:

    Display the resulting image in a new figure window. Observe also, that an array ImJPG is subtracted here from the constant 255 which mathematically does not make sense. However, Python treats the constant 255 as an array of the same size as ImJPG with all the elements equal to 255.

    Variables: ImJPG inv

    Q4: Explain what happened to the image.

```
# 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()
```

13. It is also easy to lighten or darken images using matrix addition. For instance, the following code will create a darker image:

You can darken the image even more by changing the constant to a number larger than 50. Observe that the command above can technically make some of the elements of the array to become negative. However, because the array type is uint8, those elements are automatically rounded to zero.

Variables: ImJPG dark

Q5: How do you lighten the image?

```
# 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()
```

14. Let us create Andy Warhol style art with the image provided. To do so we will arrange four copies of the image into a 2 × 2 matrix. For the top left corner we will take the unaltered image. For the top right corner we will darken the image by 50 shades of gray. For the bottom left corner, lighten the image by 100 shades of gray, and finally, for the bottom right corner, lighten the image by 50 shades of gray. Then we will arrange the images together in one larger matrix using matrix concatenation. Finally, display the resulting block matrix as a single image. If you did everything correctly, you should see an image resembling the Fig. 3.

    Variables: ImJPG Warhol

```
# 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()
```

15. Python has several commands which allow one to round any number to the nearest integer or a decimal fraction with a given number of digits after the decimal point. Those functions include: floor which rounds the number towards negative infinity (to the smaller value), ceil which rounds towards positive infinity (to the larger value), round which rounds towards the nearest decimal or integer, and fix which rounds towards zero.

    A naive way to obtain black and white conversion of the image can be accomplished by making all the gray shades which are darker or equal to a medium gray (described by a value 128) to appear as a complete black, and all the shades of gray which are lighter than this medium gray to appear as white. This can be done, for instance, by using the code:

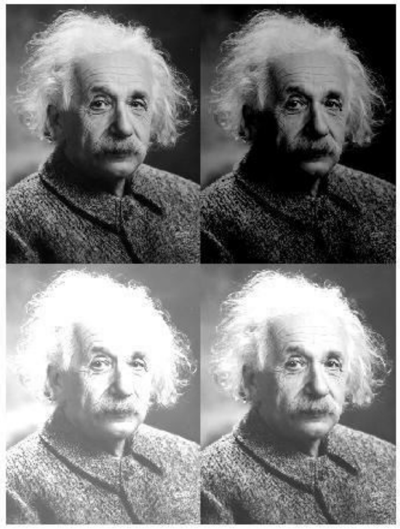


Figure 3: Andy Warhol style art with matrix concatenation

Observe that this conversion to black and white results in a loss of many details of the image. There are possibilities to create black and white conversions without losing so many details. Also notice the function uint8 used to convert the result back to the integer format.

Variables: ImJPG bw

```
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()
```

16. Now let us write code which reduces the number of shades in the image from 256 to 8 by using round function. Save your array as ImJPG8 and display it in a separate window. Variables: ImJPG8

```
# Reduce the number of shades from 256 to 8
# Step 1: Normalize the pixel values to the range [0, 1]
imjpg_normalized = imjpg_array / 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()
```

17. The contrast in an image can be increased by changing the range of the possible shades of gray in the image. Namely, we can increase the difference between the colors of the image, for instance, by making the dark shades of gray to appear even darker. One of the simplest ways to do this is to scalar multiply the matrix by some constant. Consider the code:

Display the image and observe the result. The contrast here can be manipulated by increasing/decreasing the constant (we select 1.25 here). If we select a positive constant greater than 1, then the contrast is increased; if we select a constant less than 1, then it is decreased. Observe that this operation might lead to some of elements of the matrix to become outside of 0 – 255 range and, hence, might lead to data loss.

Variables: ImJPG _HighContrast

```
# Increase the contrast by multiplying with a constant (e.g., 1.25)
contrast_factor = 1.25
imjpg_high_contrast_array = np.clip(imjpg_array * 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()
```

18. Finally, try another operation which is not linear but is still interesting. This operation is called gamma-correction. Run the following code and observe the result:

    Variables: ImJPG _ Gamma05, ImJPG Gamma15

```
# Perform gamma correction with gamma = 0.95
gamma_05 = 0.95
imjpg_gamma_05_array = np.clip((imjpg_array / 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_array / 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 γ = 0.95')
plt.axis('off')  # Hide axis
plt.show()

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

## Project 3: Matrix multiplication, inversion, and photo filters

**Goals:** We will use matrix multiplication in order to manipulate image color and achieve different effects.

**To get started:**

- Create a new python file and save it as lab03.py.
- Download the file daylilies.jpg [2] and save it in your working directory. You can use an alternative .jpg file of your choice. Your outputs will be different from shown here in that case.

**Python commands used:** `matplotlib.pyplot.imread`, `matplotlib.pyplot.imshow`, `numpy.shape`, `matplotlib.pyplot.figure`, `numpy.reshape`, `numpy.uint8`, `numpy.double`, `for` loops.

**What you have to submit:** The file lab03.py which you will create during the lab session.

### INTRODUCTION

Have you ever wondered how photo filters work? In this project we will learn how to manipulate images in order to obtain different color effects. As we learned in the previous project, an image can be represented by a matrix with the dimensions corresponding to the dimensions of the image. Each of the elements of this matrix contains the number (or numbers) representing the color of the corresponding pixel. If the image is a color image, then the color of the pixel is represented by three numbers {*R,G,B*} (Red, Green and Blue) with each number ranging from 0 to 255 (RGB system). We will look at several different ways to manipulate the image colors to obtain effects similar to those of Instagram and other graphics software.

### TASKS

1. Start by opening the image  and displaying it on the screen by using the familiar commands image.open and pyplot.imshow. Save the resulting array as ImJPG. You should see something similar to the image on the Fig. 4.

```
from PIL import Image

import numpy as np

import matplotlib.pyplot as plt

# Load the image

image_path = "C:/Users/Harshith/Downloads/laproject/daylilies.jpg"

ImJPG = Image.open(image_path)
```

[2] The copyright to this photo belongs to the author.

```
ImJPG = np.array(ImJPG)

# Display the image

plt.imshow(ImJPG)

plt.axis('off')

plt.show()
```

Variables: ImJPG

2. Check the dimensions of the array ImJPG using the command

```
m, n, l = ImJPG.shape

print(f"Dimensions of the image: {m}x{n}x{l}")
```

Observe that python output will return three numbers. Since the image is colored, the resulting array is three-dimensional. Among other things, python allows operations on multidimensional arrays. The first two dimensions of the array ImJPG correspond to the horizontal and vertical dimensions (number of pixels) in the image, and the third dimension stores three numbers corresponding to the values of Red, Green, and Blue for each pixel. These three colors mixed together produce all other colors in the image. The values for Red, Green and Blue can range from 0 to 255, allowing us to create $256^3$ = 16,777,216 colors. In this project, we will be manipulating these three numbers to achieve various visual effects (similar to the way photo filters work). Variables: m,n,l

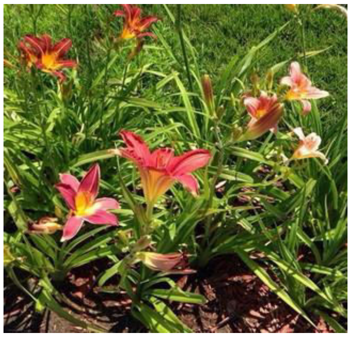


Figure 4: The original image showing orange and yellow daylilies

3. First of all, let us look into amount of Red, Green, and Blue color in the image. To do this, let us extract individual layers or color channels from the image array. To obtain the red channel of color in the image, use the following command:

```
 # Extract color channels

redChannel = ImJPG[:, :, 0]

# Display the color channels

plt.figure()
plt.imshow(redChannel, cmap='gray')

plt.title('Red Channel')

plt.axis('off')

plt.show()
```

Similarly, extract green and blue color channels and save them in the arrays greenChannel and blueChannel. Display each array in a separate figure using figure and imshow commands. Observe that individual color channels will be shown by python output as grayscale images. That is due to the fact that we have taken "layers" off of the matrix ImJPG and each of these layers individually looks like a two-dimensional array with numbers ranging from 0 to 255 which is exactly the form in which grayscale images are represented in Python. If you did everything correctly, you should see something similar to the Fig. 5. The whiter the pixel appears on a particular color channel, the larger amount of that color is contained in that pixel. On the other hand, the darker the area is, the less corresponding color is in that part of the image. The file daylilies.jpg appears to contain mostly green and red (surprise, surprise!).

```
# 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()
```

Variables: redChannel, greenChannel, blueChannel

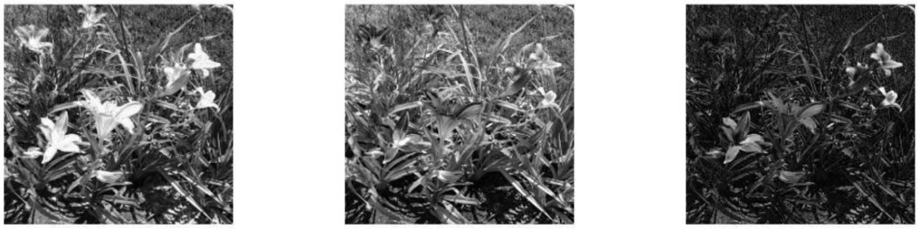


Figure 5: Different color channels

4. Let us convert the image into a grayscale image by using the following filter matrix:

$$GrayMatrix = \begin{bmatrix} 1/3 & 1/3 & 1/3 \\ 1/3 & 1/3 & 1/3 \\ 1/3 & 1/3 & 1/3 \end{bmatrix}$$

Multiplication by this matrix allows to compute an average of red, green, and blue in each pixel. Use the following code:

```
# 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()
```

This code produces an array ImJPG Gray in which the individual values of Red, Green, and Blue have been replaced by the averages of these three colors. Observe the use of the commands reshape, uint8, double. For every pixel of the image, first, the color attributes (RGB) are extracted from the matrix ImJPG, then these color attributes are treated as a vector with three components [*R,G,B*], and finally, the new color attributes are obtained by multiplying the vector [*R,G,B*] by the filter matrix *GrayMatrix* on the left. The result is saved as color attributes of the corresponding pixel in the array ImJPG Gray. All the pixels of the array ImJPG _Gray have equal number of Red, Green, and Blue; this produces different shades of gray color. Observe that there are different ways to create a grayscale conversion, and this is just one of them.

Variables: ImJPG _Gray

Q1: Explain the purpose of `np.uint8` and `np.float64` commands in the code above.

Ans:

- **`np.uint8`**: This command is used to convert numeric values to an unsigned 8-bit integer format. In image processing, pixel values typically range from 0 to 255. Using `np.uint8` ensures that any calculated or transformed pixel values are within this range, which is essential for correct image representation and visualization.

- **`np.float64` (or `float`)**: This command is used to convert numeric values to a floating-point format with double precision (64 bits). In image processing operations that involve mathematical calculations like transformations or filters, using `np.float64` ensures that the calculations are performed with sufficient precision to avoid loss of information or accuracy.

5. Modify the code above to produce a sepia conversion of the image. Instead of GrayMatrix use the following filter matrix, and reproduce the code above with this matrix:

$$SepiaMatrix = \begin{bmatrix} 0.393 & 0.769 & 0.189 \\ 0.349 & 0.686 & 0.168 \\ 0.272 & 0.534 & 0.131 \end{bmatrix}$$

Save the result in the ImJPG Sepia array and display this array using imshow command.

```
# 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()
```

Variables: ImJPG _Sepia

6. Next, consider the filter matrix

$$RedMatrix = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Modify the code again, using the matrix above as a filter matrix. Save the result as a new array ImJPG _Red. Display the image.

```
# 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()
```

Variables: ImJPG Red

Q2: What does the transformation above do?
Ans: The transformation using `RedMatrix` effectively removes all colors except for the red channel

from the image. Pixels in `ImJPG_Red` will have only their red component remaining (set to the original red value of the pixel), while the green and blue components are set to 0. This results in a grayscale-like image where shades of red represent the intensity of the red channel across the original image.

7. Let us permute the colors in the image. To do this repeat the steps above with the matrix:

$$PermuteMatrix = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

Save the result as a new array ImJPG_Permute. If you did everything correctly, you should see the image shown on the Fig. 6. Observe that the flowers turned purplish blue. The matrix above is an example of a hue rotation filter. You can produce other hue rotation effects with this more general transformation:

$$HueRotateMatrix = \begin{bmatrix} 0.213 & 0.715 & 0.072 \\ 0.213 & 0.715 & 0.072 \\ 0.213 & 0.715 & 0.072 \end{bmatrix} + \cos(\theta)\begin{bmatrix} 0.787 & \begin{matrix} -0.715 \\ 0.285 \end{matrix} & -0.072 \\ -0.213 & -0.715 & -0.072 \\ -0.213 & & 0.928 \end{bmatrix} + \sin(\theta)\begin{bmatrix} 0.213 & -0.715 & 0.928 \\ 0.143 & 0.140 & -0.283 \\ -0.787 & 0.715 & 0.072 \end{bmatrix}.$$

Here $\theta$ is the angle of rotation. You can try experimenting with various values of the angle $\theta$ to get different color effects.

```
# 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()

#7 using hue rotate matrix and variable theta

# 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()
```

Variables: ImJPG Permute

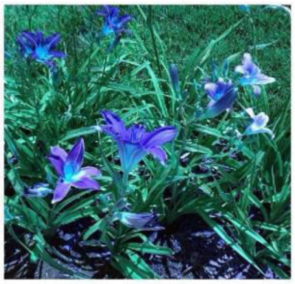


Figure 6: Colors permuted

8. Now let us delete one of the colors in the image, say, green. First, produce the filter matrix which deletes Green in the image and keeps the values of Red and Blue intact. Use it to filter the image. Save the result as ImJPG _DeleteGreen.

```
#8

# 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()
```

Variables: ImJPG DeleteGreen

9. It is possible to invert the colors of the image by using the following code:

```
# 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()
```

Observe that here again Python automatically substitutes a matrix of appropriate dimension (in this case *m*×*n*×*l*) with all the elements equal to 255 instead of the constant 255.

Variables: ImJPG invert

10. It is also possible to enhance/mute individual colors in the image. For instance, consider the color transformation with the matrix:

$$SaturateMatrix = \begin{bmatrix} 1.2 & 0 & 0 \\ 0 & 0.75 & 0 \\ 0 & 0 & 2 \end{bmatrix}$$

Save the result as ImJPG Saturate.

```
# 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()
```

Variables: ImJPG saturate

Q3: Explain what the transformation above does to the image.
Ans:

- The transformation will result in an image where red tones appear more pronounced, green tones are muted, and blue tones are enhanced. This can lead to a variety of effects depending on the original colors and the degree of saturation applied.
- For instance, if the image has a lot of greenery, using this matrix might reduce the prominence of green elements while boosting reds and blues, potentially altering the overall color balance and visual impact of the image.

11. Consider the color adjusting filter generated by the matrix:

$$UserMatrix = \begin{bmatrix} 0.7 & 0.15 & 0.15 \\ 0.15 & 0.7 & 0.15 \\ 0.15 & 0.15 & 0.7 \end{bmatrix}$$

Apply this filter to the array ImJPG and save the result as ImJPG User.

```
# 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()
```

Variables: ImJPG User

Q4: Explain what the transformation above does to the image.
Ans:

- Applying `UserMatrix` to `ImJPG` will modify the color balance of the image. It can subtly shift the overall hue or alter the intensity of individual color components depending on the values of `UserMatrix`.
- In practical terms, this transformation might adjust the warmth or coolness of the image, emphasize certain color tones, or create a specific color cast.

12. Consider the matrix in the previous step. Is the transformation defined by this matrix invertible? Which means, can you get the original image colors back once you applied the filter? Or is some of the information lost? Use the inv command to create the matrix for the

inverse filter. Apply this matrix to the arrays ImJPG _User and ImJPG. Display the resulting images.

```
# 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()
plt.show()
```

Q5: What does the inverse transformation do with the colors?

- **Inverse Transformation Effect:** It adjusts the color values in such a way that, ideally, the resulting image resembles the original ImJPG more closely in terms of color balance and intensity.

- **Purpose:** The goal of applying the inverse transformation is to demonstrate whether the original image colors can be reasonably recovered after applying a color adjustment filter (UserMatrix). This helps in understanding whether the transformation is reversible and how much information might have been altered or lost during the initial color adjustment process.

13. Repeat the steps above to see if the sepia transformation is invertible.

```
# 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()
```

Q6: What do you see? Why do you think this happened?

The near singularity of the `SepiaMatrix` (det(SepiaMatrix) close to zero) indicates that the transformation introduces dependencies or compresses information in a way that makes it challenging to perfectly invert. This could be due to the specific coefficients chosen in the sepia transformation matrix, which may not preserve enough information to fully reverse the color adjustments made. Therefore, while we can compute an inverse matrix and attempt to reconstruct the original colors, the result may not be identical to the original image.

14. Finally, try the non-linear color transformation such as Gamma-conversion:

```
# Gamma conversion
ImJPG_Gamma1 = np.uint8(np.power(ImJPG.astype(float), 0.9)) + 30
ImJPG_Gamma2 = np.uint8(np.power(ImJPG.astype(float), 1.1)) - 50

# Display the resulting images
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_Gamma1)
plt.title('ImJPG_Gamma1')
plt.axis('off')

plt.subplot(1, 3, 3)
plt.imshow(ImJPG_Gamma2)
plt.title('ImJPG_Gamma2')
plt.axis('off')

plt.tight_layout()
plt.show()
```

Display the resulting images given by the matrices ImJPG Gamma1, ImJPG _Gamma2.

Variables: ImJPG _ Gamma1, ImJPG Gamma2

## Project 4: Solving linear systems in Python

**Goals:** To explore the built-in functions in Python used to solve systems of linear equations **Ax** = **b**. To compare different options in terms of accuracy and efficiency.

**To get started:** Create a new script file and save it as lab04.py.

**What you have to submit:** The file lab04.py, which you will create during the lab session.

## INTRODUCTION

All the methods of solution of systems of linear equations can be divided into two large groups: direct and iterative methods. The direct methods aim to obtain an “exact” (up to numerical round off error) solution of the system in a finite number of steps. The iterative methods aim to create a convergent sequence of approximations to the exact solution of a linear system. Iterative methods are usually used when the matrix of a system has a known specific structure. Python has many built-in functions designed to solve systems of linear equations by both direct and iterative methods. In this project we will discuss only direct methods such as Gaussian elimination.

Gaussian elimination in Python is based on *LU*- or *PLU*-decomposition of the matrix of the system **A**. We can reduce the matrix **A** to an upper triangular form using forward elimination and then solve the resulting triangular system by backward substitution. In the process of doing this, we factor the matrix **A** into a product of two matrices: a lower triangular matrix **L** and an upper triangular matrix **U**. Observe that this process cannot be always carried out without interchanges of rows, and, hence, we need an additional permutation matrix **P** in our matrix decomposition. Observe, that the matrix **P** is necessary even in the cases when *LU*-decomposition can be carried out due to the matters of numerical stability (this is called pivoting). Gaussian elimination is the preferred method of solving the system numerically if you do not have any additional information about the structure of the matrix **A**.

If the matrix **A** is square, we can define the inverse matrix of **A** (denoted as $\mathbf{A}^{-1}$). The matrix $\mathbf{A}^{-1}$ is also a square matrix of the same size as **A**. The inverse matrix exists as long as **A** is not singular, and satisfies the following equality:

$$\mathbf{A} \cdot \mathbf{A}^{-1} = \mathbf{A}^{-1} \cdot \mathbf{A} = \mathbf{I},$$

where **I** is the identity matrix of the same size as **A** and $\mathbf{A}^{-1}$. If the matrix **A** is invertible, then we can multiply both sides of the system $\mathbf{Ax} = \mathbf{b}$ by the inverse matrix $\mathbf{A}^{-1}$ on the left, and obtain the following formula for the solution of the system: $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$. However, as it often happens in numerical linear algebra, this theoretical formula is inconvenient for numerical computation of the solution. The reasons for this are two-fold. Firstly, the computation of inverse matrix is significantly more costly than simply solving the system. Secondly, the accuracy of the resulting solution will be worse than if *LU*-decomposition is used.

Finally, yet another possibility to solve a linear system is to use the reduced row echelon form of the augmented matrix of the system. Theoretically, this is equivalent to using GaussJordan elimination. This method of solution will be explored in this project as well. Analysis of the corresponding algorithm shows that the resulting solution is likely to have a larger error compared to Gaussian elimination and is likely to take more time to compute.

## TASKS

1. Create a new cell and type in the following commands to generate the matrix A and the column vector b:

```
import numpy as np

# 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)
```

Variables: A, b

2. If nothing is known about the structure of the matrix of the system, the best approach to solution is to use the Python's "numpy" solver:

   ```
   # Solve the system Ax = b using numpy's solve function

   x = np.linalg.solve(A, b)

   # Print the solution vector x

   print("\nSolution vector x:")

   print(x)
   ```

   In most cases, this method will produce the numerical solution of the system Ax=b with the highest possible accuracy and in the shortest computational time. Variables: x

3. Whenever we are talking about the accuracy of the solution, ideally we want to find the difference between the approximate numerical solution $x_{num}$ and the exact solution $x_{ex}$ of the system. However, this is not feasible in most situations since the exact solution $x_{ex}$ is generally unknown (otherwise, there is no point in using numerical methods at all). The next best thing is to find the residual r=A*x-b. If the residual is small then we can assume, with certain limitations, that our approximate solution is close enough to the exact solution (this is not always true, however). Let us check how accurate the solution is which we found in the previous step. Type the following line into your python file:

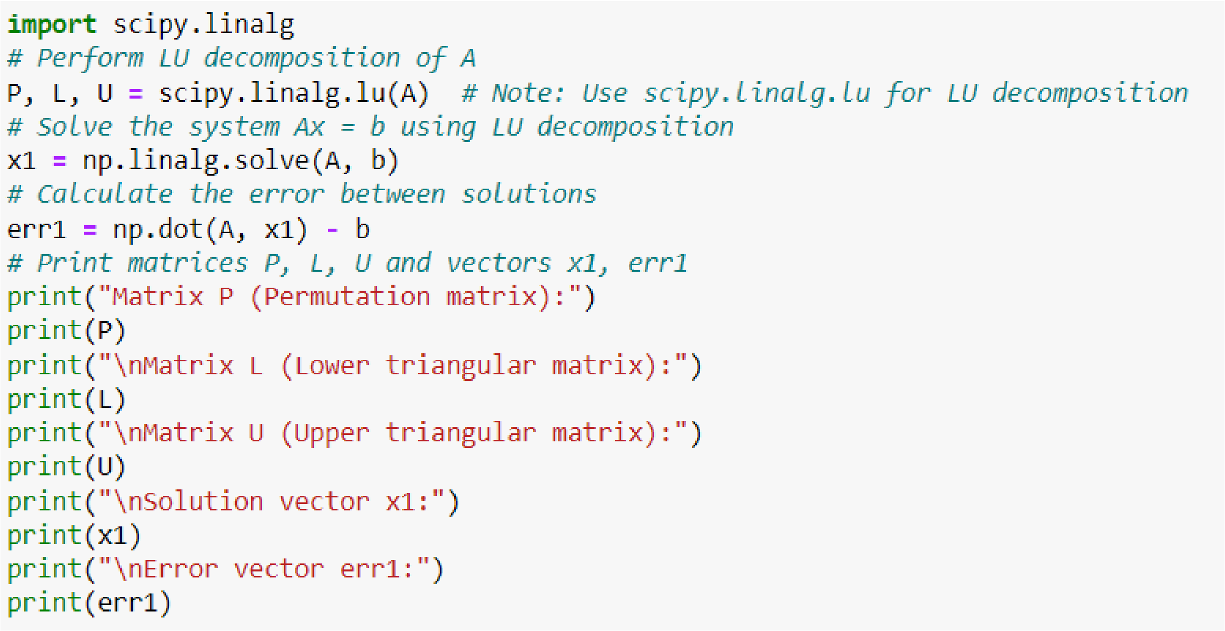
   ```
   # Calculate the residual r = A*x - b

   r = np.dot(A, x) - b

   # Print the residual vector r

   print("\nResidual vector r:")

   print(r)
   ```

   The resulting vector r should be very close to zero. Variables: r

4. The Python backslash "lu" operator is based on *LU*- or *PLU* decomposition of the matrix **A**. The main idea here is that any square non-singular matrix **A** (meaning that the inverse matrix $\mathbf{A}^{-1}$ exists) can be written as **A** = **PLU** , where **P** is a permutation matrix (or matrix obtained by permuting the rows of the identity matrix of the same size as **A**); **L** is a lower-triangular matrix (all the elements above the main diagonal are equal to zero), and **U** is an upper triangular matrix (all the elements below the main diagonal are equal zero). Python has a special lu function to generate a $\mathbf{P}^{T}\mathbf{LU}$ decomposition of the matrix **A**. Type the following lines into your Python script:

   This will create a new code cell and compute the matrices P, L, and U. Now let us use these matrices to solve the linear system **Ax** = **b**:

```
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)
```



```
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)
```

Observe that the difference between the solutions err1=x-x1 is exactly the zero vector[3]. This is due to the fact that we found the solution using exactly the same steps as Python. Observe also that at no point did we use the Python inverse matrix function inv. In general, there is almost never a good reason to compute an inverse matrix. Whenever you see an inverse matrix in a numerical linear algebra textbook it is usually shorthand for "solve a system of linear equations here". Variables: L, U, P, x1, r1, err1

5. Python also has a numpy.linalg.lstsq function to solve linear systems of the type **yA** = **b**. Try running the following command y=b'/A

```
# 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)
```

Variables: y

6. Next, let us use the inverse matrix to solve the same system of linear equations:

```
# 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:")
```

[3] This has been observed in both Matlab 2015a and 2019a. It has come to author's attention that this might not be true in other versions of Matlab.

```
print(r2)

print("\nError vector err2:")

print(err2)
```

Here inv(A) produces the inverse $\mathbf{A}^{-1}$ of the matrix **A**. This code will also produce the solution of the system **Ax** = **b**, compute the residual, and find the difference between the current solution and the solution obtained by the backslash operator. Observe that the vector of difference err2 is not exactly zero unlike in the case when we used *LU*decomposition. Observe also that the residual r2 is likely to be larger than the one obtained with the help of the backslash operator. Variables: x2, err2, r2

7. Finally, another way of solving the system **Ax** = **b** is based on the reduced row echelon form. The reduced row echelon form in Python can be produced by using the command rref. Type in the following code:

```
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 (difference
between x and x3):")

print(err3)
```

Observe that the residual r3 is much larger than the residual r. Generally speaking, rref is also a much less efficient method of solving linear systems compared to the backslash operator and its use for this purpose is not advised. Variables: x3, err3, r3

8. To compare the computational efficiency of all three methods, run the code cell:

```
import time

# 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 times
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 form):")
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")
```

This code compares all three methods in terms of the computational efficiency in the example of a large matrix **A** with the dimensions Num x Num. The parameter Num in this code is set to be 500. The matrix **A** is chosen in such a way that it is diagonally rowdominant. Then by the theorem of linear algebra it is guaranteed that the matrix **A** is nonsingular. The Python function tic … toc returns the computational time necessary to perform the Python operations between tic and toc. You can try to experiment with the parameter Num to see how the computational time changes with the size of the matrix **A**. Observe that solving the system with the rref function may take a very noticeable amount of time (you probably don't want to set Num to be more than 1000). Variables: Num, A, b, C, x1, x2, x3

9. Finally, let us see how to use Python to solve rectangular systems of linear equations. Let **A** be an $m \times n$ matrix of the system and let **b** be an $m \times 1$ column vector. Let us start with the situation when $m > n$. In this case, the system is overdetermined. However, it is still possible to use the Python backslash operator.

The system **Ax** = **b** is inconsistent. The resulting "solution" x is not the solution of the original linear system **Ax** = **b** itself (observe that the residual r is not close to zero), but rather the solution to the so-called least squares problem. Namely, x minimizes the length of the residual vector **r** = **Ax** − **b**. It is known that the solution to the least squares problem can also be obtained by solving the system of the so-called normal equations $\mathbf{A}^T\mathbf{Ax} = \mathbf{A}^T\mathbf{b}$. To verify that the vector x is indeed the solution to this system, type in the following code:

```
# Define matrix A and vector b for
the 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 column vector


# 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)
```

and notice that x-y is indeed close to zero (it is rare to obtain an exact zero in numerical computations due to roundoff errors). Variables: A, b, x, r, y, err

10. Let us look at the last possible case when *m* < *n*. This is a case of an underdetermined system. If this system is consistent then it has an infinite number of solutions. Consider the following code:

```
# 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]).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)
```

The backslash operator in this case produces only one particular solution to the system **Ax** = **b**. Another particular solution can be obtained by the following operations:

```
# 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)
```

where the function pinv produces the so-called Moore-Penrose pseudoinverse (the precise definition of this matrix is outside of the scope of this project). It is known that the vector y has the smallest possible length among all the solutions of the system **Ax** = **b**. Finally, to obtain the complete set of the solutions of the system above, consider the function null. This function produces the complete set of the linearly independent solutions to the homogeneous system **Ax** = **0**. The complete solution to the inhomogeneous system **Ax** = **b** then can be obtained by:

```
# Obtain the complete set of solutions using the null space of A
```

```
null_space = np.linalg.null_space(A)
C = np.array([1, 2])  # Arbitrary constants C1, C2


# Calculate the complete solution z = null_space(A) * C + x
z = np.dot(null_space, C) + x


# Print the complete solution vector z
print("\nComplete solution vector z:")
print(z)
```

where the constant vector $\mathbf{C} = [C_1, C_2]^T$ can be chosen arbitrarily (pick any numbers for $C_1$, $C_2$ that you like!).

Variables: A, b, x, r1, y, r2, C, z

## Project 5: Systems of linear equations and college football team ranking (with an example of the Big 12)

**Goals:** To use systems of linear equations to develop a ranking for Big 12 college football teams based on 2016-2017 season data using Colley's and Massey's methods.

**To get started:**

- Open a new Python script and save it as lab05.py
- Download the files Big12th.xls, Scores.mat, and Differentials.mat. Observe that the file Big12th.xls is provided for reference only and won't be used explicitly in the lab. You can also use the scores from your favorite tournament. The scores here are based on a 10 team round-robin tournament.

**Python commands used:** numpy.load, numpy.eye, numpy.diag, numpy.sum, numpy.abs, print, for … in range, numpy.sort

**What you have to submit:** The file lab05.py, which you will create during the lab session.

### INTRODUCTION

The present project deals with the ranking of competing sports teams. At first, this does not appear to be a difficult problem. For instance, the easiest way to rank sports teams' performance is to compute the percentage of games which each team has won. This method has several drawbacks which are especially noticeable for the college football championship. In particular, for the percentage ranking to be objective, it is necessary that every team plays with a majority of teams participating in the competition. However, the number of teams participating in the Bowl Championship Series (BCS) is very large and each team only plays a small number of games. This is what makes the ranking of college football significantly different from, say, the NFL, where a relatively small number of teams play a representative number of games against each other.

It is necessary to note that there is no one true answer to a ranking problem. The rank of a team will depend on the factors which we will take into account and on the weight we will give to those factors. There are several questions to ask here. For instance, do point differentials matter? Is a 60-20 win any better than a 34-33 win? Taking into account point differentials may have both pros and cons. A 34-33 win versus a strong opponent is arguably better than 60-20 versus an unranked team. At the same time, if all the teams have about the same strength, than 60-20 is clearly better. At the end of the day, whether to take into account point differentials or not is a matter of taste and preference.

In this lab we will look at the two methods of ranking currently used by BCS with an example of the Big 12 Conference data from 2016-2017 season. As you may know, 10 teams play in the conference: Baylor University, Iowa State University, University of Kansas, Kansas State University, University of Oklahoma, Oklahoma State University, Texas Christian University, University of Texas Austin, Texas Tech University, and West Virginia University. We will look at the games only within the Big 12 conference and ignore all other games outside of it.

## Colley's bias free ranking

The first method we will look at has been developed by astrophysicist Wesley Colley and is named after him: Colley's bias free ranking. This method does not take into account point differentials in the games (which again can be both a good and a bad thing).

There are two main ideas behind the method. First of all, we substitute the winning percentages by the Laplace's rule of succession:

$$r_i = \frac{1 + w_i}{2 + t_i},$$

where $w_i$ is the number of winning games for the *i*th team, $t_i$ is the number of total games, and $r_i$ is the rank of the team. Laplace's rule of succession is used for events which have only two outcomes such as either winning or losing. The rule comes from the idea in probability that unless an event always happens or never happens, we should avoid assigning this event probabilities of 1 (always true) and 0 (never true). Indeed, after playing one game, a team could either win the game or lose it. If the team won the game, than its percentage ranking is 100%; if it lost the game, its ranking is 0%. Meaning that after one game some teams are *infinitely* better than others which is not very realistic. At the same time, the Laplace's rule of succession will give us approximately 2/3 and 1/3, meaning that the team which won one game is about two times better than the one which lost one game - a good starting point for further estimates.

The second idea is that the rank of opposing teams should matter just as much as the total number of wins and losses. To incorporate the opposing teams' rankings into our system, we can write:

$$w_i = \frac{w_i - l_i}{2} + \frac{w_i + l_i}{2} = \frac{w_i - l_i}{2} + \frac{t_i}{2},$$

where $l_i$ is the number of losses for the *i*th team.

Now observe that with the use of Laplace's succession rule the rankings of the teams will hover around 1/2, so we can approximate:

$$\frac{t_i}{2} = t_i \frac{1}{2} \approx \sum_{j=1}^{t_i} r_j^i .$$

Basically, we substitute each 1/2 with the ranking of the $r_j^i$ of the *j*th team which *i*th team played against. Now, let us substitute the last formula back into the Laplace's rule to obtain the following system of linear equations (check it!):

$$(2 + t_i) r_i - \sum_{j=1}^{t_i} r_j^i = 1 + \frac{w_i - l_i}{2} .$$

This system has *N* equations with *N* unknowns $r_i$, where *N* is the number of teams playing in the championship (in our case *N* = 10). Observe that the matrix of this system has special properties - it is diagonally row dominant, meaning that the system will always have a unique solution.

## Massey's method

Massey's method was developed by Ken Massey in his undergraduate research project and is now used by the BCS (Oh, the places you can go!... if you know math). Unlike Colley's method, Massey's takes into account point differentials as well as number of wins/losses. Again, denote as $r_i$ the rankings of the competing teams. The data we obtain from the games can be written in the form of linear equations: $r_i - r_j = d_{ij}$,

meaning that *i*th team won (lost to) *j*th team by the $d_{ij}$ point differential.

The data from the games produces a large system of linear equations of the form:

$$\mathbf{PR} = \mathbf{B},$$

where **P** is the matrix of the system such that each row has only two non-zero elements, 1 and −1, **R** is the column-vector of the rankings $r_i$, and the vector of the right-hand-side **B** contains the corresponding point differentials.

Observe that, in general, we are likely to have a lot more equations than unknowns (more games than teams), so, in general, this system of equations is likely to be inconsistent and not have any solutions. To see this, assume that the first team beat the second team by 20 points and the third team beat the second team by 5 points. This data implies that the first team should beat the third team by 15 points. Of course, the point differential of the actual game (if it happens!) between the first team and the third team might be different.

So, instead, we will give up on trying to solve this system exactly, and we will look for the vector **R** which satisfies the system as close as possible. Namely, we will employ the least squares method, meaning, that we want to find the vector **R** which minimizes the vector norm:

$$||\mathbf{PR} - \mathbf{B}||_2$$

where $|| \cdot ||_2$ is a usual Euclidean norm: if $\mathbf{x} = (x_1, x_2, ..., x_n)$ is a vector with $n$ components, then

$$||\mathbf{x}||_2 = \sqrt{x_1^2 + x_2^2 + ... + x_n^2}.$$

It is well-known from linear algebra that this problem is equivalent to the so-called system of normal equations written as:

$$\mathbf{P}^T\mathbf{PR} = \mathbf{P}^T\mathbf{B},$$

where $\mathbf{P}^T$ is a transpose of the matrix **P**. Now, $\mathbf{P}^T\mathbf{P}$ is an $N \times N$ matrix, where $N$ is again the number of the teams, so this results in a square system of linear equations.

The last obstacle which we will need to overcome is that the matrix $\mathbf{P}^T\mathbf{P}$ has a rank of at most $N - 1$ (why?), and hence, in general, the last system does not have a unique solution. To obtain a unique solution we will change each element in the last row of the matrix $\mathbf{P}^T\mathbf{P}$ to 1 and we will change the last element in the vector $\mathbf{P}^T\mathbf{B}$ to zero. Practically this means that all the rankings $r_i$ obtained in this way will need to sum up to zero.

Now that you learned the theory behind the ranking, let us create code in Python which will produce the rankings of teams in the Big 12. To do so, first of all save the data files Scores.mat and Differentials.mat in the directory you will be working in. Each of these files contains the results of the games from the Big 12 championship from the season 2016-2017 in the form of the table (matrix). The Scores.mat contains a 10 × 10 matrix where each row corresponds to the scores of the Big 12 teams in the following order:

1. Baylor
2. Iowa State

3. University of Kansas
4. Kansas State
5. University of Oklahoma
6. Oklahoma State
7. Texas Christian
8. University of Texas Austin
9. Texas Tech
10. West Virginia

The element (*i,j*) of the matrix Scores is 1 if *i*th team won over *j*th team. Correspondingly, if *i*th team lost to *j*th team, then the (*i,j*) element is −1. The diagonal elements are taken to be 0.

The file Differentials.mat contains the matrix Differentials which is similar to Scores, only instead of ±1 it has the point differentials of the corresponding games.

**TASKS**

1. Start by loading both matrices into your Python script using the load command:

```
import numpy as np

from scipy.io import loadmat

# Load the matrices from .mat files

data_scores =
loadmat('C:/Users/Harshith/Downloads/laproject/materi
als/Scores.mat')

data_differentials =
loadmat('C:/Users/Harshith/Downloads/laproject/materi
als/Differentials.mat')

# Extract the relevant matrices

Scores = data_scores['Scores']

Differentials = data_differentials['Differentials']
```

Variables: Scores, Differentials

2. First, consider the Colley method. Create the matrix and the right-hand side of Colley's system of linear equations:

```
# 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)
```

Variables: ColleyMatrix, RightSide

Q1: What do the commands np.eye and np.diag do?

Ans:

*np.eye*: This function creates a 2-D array with ones on the diagonal and zeros elsewhere. The size of the matrix is specified by the argument passed to the function.

Example: np.eye(3) results in the matrix: [[1. 0. 0.] [0. 1. 0.] [0. 0. 1.]]

*np.diag*: This function creates a diagonal matrix from a given 1-D array, or it extracts the diagonal elements from a 2-D array.

Example (creating diagonal matrix): np.diag([1, 2, 3]) results in: [[1 0 0] [0 2 0] [0 0 3]]

Example (extracting diagonal): np.diag([[1, 0, 0], [0, 2, 0], [0, 0, 3]]) results in: [1, 2, 3]

3. Solve the linear system using the backslash operator:

```
# Solve the linear system using np.linalg.solve

RanksColley = np.linalg.solve(ColleyMatrix, RightSide)

# Variables: RanksColley

print("RanksColley:\n", RanksColley)
```

Variables: RanksColley

4. Display the results on the screen:

```
# 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}')
```

Variables: Teams, Order, RanksDescend

Q2: Observe that the array Teams is in curly parentheses. What kind of array is that?

Ans: The array `Teams` is a list of strings in Python.

Q3: Explain what kind of formatting does the line $'\%8.3f\% - 15s \backslash n'$ in the command fprintf produce?

Ans:

- `%8.3f`: This formats a floating-point number to have a total width of 8 characters, including 3 characters after the decimal point.
- `%-15s`: This formats a string to have a minimum width of 15 characters, left-justified.
- `\n`: This adds a newline character, moving the cursor to the next line.

Q4: What does sort function do?

Ans: In Python, `np.argsort` is used to get the indices that would sort an array, and `[::-1]` is used to reverse the order for descending sorting.

5. Now let us use Massey’s method to rank the teams. Create the matrix **P** and the righthand-side **B** by using the following code:

```
# 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)
```

This code scans the part of the matrix Differentials which is located above the main diagonal. For each nonzero (*i,j*) (meaning that the teams *i* and *j* played a game) element of this matrix it creates a row of the matrix **P** which contains only two non-zero elements: 1 in the *i*th column of this row and −1 in the *j*th column. The vector **B** of the right-handside is created simultaneously and has the corresponding point differential in the game between *i*th and *j*th team. Variables: P, B

6. Next, create the normal system of linear equations:

```
# 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)
```

Variables: A, D

7. Substitute the last row of the matrix as described above:

```
# 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)
```

8. Finally, solve the system:

```
# Solve the system

RanksMassey = np.linalg.solve(A, D)

# Print the results

print("RanksMassey:\n", RanksMassey)
```

Variables: RanksMassey

9. Display the results on the screen as before:

```
# 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}')
```

Variables: RanksDescend, Order

10. Compare the results of both Colley's and Massey's methods. Compare both the methods with the official results. Observe that while these two methods are used by BCS, they do not take into account many things, such as the timing of the games, strength of schedule, games played against opponents which are not in the Big 12, and so on. These factors may affect the rankings and make it deviate from the official. For the data considered in this project the comparison with the official results should be very good.

11. Create a new code cell. Switch the result of the game between the top two teams in Colley's ranking and run the Colley's algorithm again. Produce the rankings of the teams in this case.

```
# 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]]
```

Q5: Did the ranking of the top teams change?

12. Create a new code cell. Now, switch the result of the game between two top teams in Massey's ranking (those might be different teams from Colley's algorithm!) and run the Massey's algorithm again. Produce the rankings of the teams in this case.

```
# 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]]
```

Q6: Did the ranking of the top two teams change?

# Project 6: Convolution, inner product, and image processing revisited

**Goal:** In this project we will look at the convolution of two matrices and its application to image processing.

**To get started:**

- Create a new Python script file and save it as lab06.py
- Download the file einstein.jpg and put it in your working directory.

**Python commands used:** load, rand, double, uint8, conv2, filter2

**What you have to submit:** The file lab06.py which you will create during the lab session.

### INTRODUCTION

In this section we will look at the applications of matrix multiplication to the filtering of images. The most common operations in filtering are bluring/smoothing, sharpening, and edge detection. All of these are examples of so-called convolution filters. These filters work by changing the color of the pixels in the image using the colors of nearby pixels.

Let us look at how filters of this type work. Consider an image described by the matrix $\mathbf{f} = (f_{ij})$ where $(i,j)$ are integer coordinates of the pixel in an image, and the value $f_{ij}$ returns the color of the image. Consider $\mathbf{h} = (h_{kl})$ to be a convolution matrix (filter matrix). These matrices might have (and usually do have) different dimensions. The filter matrix is usually relatively small. Then a new matrix which is a convolution of **f** and **h** (written as $\mathbf{g} = \mathbf{f} ? \mathbf{h}$) can be found by the following formula:

$$g_{ij} = \sum_k \sum_l f_{kl} h_{i-k+1,j-l+1},$$

where the summation runs over all the indices which make sense. In particular, if an element with the certain index does not exist, we simply assume the corresponding term to be zero. You can think of convolution being obtained by the filter matrix **h** sliding over the original image **f**.

Notice that the formula for convolution resembles the definition of an inner product if we treat both matrices as vectors (say, with columnwise orientation) and flip the second matrix both horizontally and vertically.

Python has two functions to compute convolutions of matrices: conv2 and filter2. These two functions are closely related. In particular, the function filter2 is equivalent to the function conv2 with the matrix **h** rotated 180 degrees.

### TASKS

1. To begin, let us load the file einstein.jpg and save the resulting matrix as ImJPG. Use the size function to find the dimensions $m$, $n$ of the matrix ImJPG.

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

# Load the image
ImJPG = cv2.imread('C:/Users/Harshith/Downloads/laproject/einstein.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()
```

Variables: ImJPG, m, n

2. Introduce some noise into the image by adding random fluctuations of color to each point:

```
    # 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
```

Observe that the command rand(m,n) produces a matrix of the dimension $m \times n$ filled with pseudorandom numbers within the interval (0,1). The amplitude of the noise is equal to

±25 shades of gray. Also, notice the function double which converts the variables from the type uint8 to the type double.

3. Two of the most common operations on images done with convolution filters include smoothing and sharpening. Let us start by using two average smoothing filters given by the matrices:

$$Kernel\_Average1 = \frac{1}{5}\begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix}, \quad Kernel\_Average2 = \frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

4. Using the appropriate Python syntax, type in the matrices Kernel_Average1 and Kernel Average2.

```
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
```

Variables: Kernel Average1, Kernel Average2

Q1: Why are the fractions 1/5 and 1/9 necessary?

5. Apply the filters to the noisy image by using the commands

```
import numpy as np
from scipy.signal import convolve2d
import matplotlib.pyplot as plt

# 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()
```

Display the resulting images in separate figure windows and observe the result. Don't forget to convert the results back to the integer format by using uint8 function.

Variables: ImJPG Average1, ImJPG Average2

Q1: What is the size of the matrices ImJPG Average1, ImJPG Average2?

Q2: What is the size of the original matrix ImJPG? Use the size function.

Q3: Which filter blurs more? Why?

6. An alternative blurring filter, Gaussian blur, is given by the matrix:

$$Kernel\_Gauss = \frac{1}{8}\begin{bmatrix} 0 & 1 & 0 \\ 1 & 4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

which assigns a higher weight to the pixel color in the center. Type in the matrix Kernel Gauss in your Python file using the appropriate syntax.

```
# 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)
```

Variables: Kernel Gauss

7. Perform the convolution using the function conv2 and Kernel Gauss and save the resulting array as ImJPG Gauss. Display the result in a new window.

```
# 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()
```

Variables: ImJPG Gauss

8. Observe that we can "layer" filter effects. Perform another convolution with the Gaussian kernelontheimage ImJPG-Gauss,savetheresultas ImJPG Gauss2,anddisplaytheresult inanewwindow.

Variables: ImJPG-Gauss2

Q4: Devise a matrix for a filter which is equivalent to applying Gaussian convolution twice.

Q5: What is the size of the matrix for this filter?

```
#8

# 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()
```

9. It is also possible to blur the image over a larger area:

$$Kernel\_Large = \frac{1}{80}\begin{bmatrix} 0 & 1 & 2 & 1 & 0 \\ 1 & 4 & 8 & 4 & 1 \\ 2 & 8 & 16 & 8 & 2 \\ 1 & 4 & 8 & 4 & 1 \\ 0 & 1 & 2 & 1 & 0 \end{bmatrix}$$

Apply the kernel Kernel _Large to the matrix ImJPG, save the result as ImJPG _Large, and display the figure in a new figure window.

```
#9

# 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()
```

Variables: ImJPG _Large

Q6: Compare with applying Gaussian blur applied twice. Which one blurs more?

10. An opposite action to blurring is the sharpening of an image with a convolution filter. Type the following kernels into your Python file:

$$Kernel\_Sharp1 = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}, \quad Kernel\_Sharp2 = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 9 & -1 \\ -1 & -1 & -1 \end{bmatrix}.$$

```
#10

# 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)
```

Variables: Kernel Sharp1, Kernel _Sharp2

11. Perform the convolution of the original image ImJPG with the kernels Kernel Sharp1, Kernel Sharp2 using the function conv2 and save the resulting arrays as ImJPG Sharp1 and ImJPG Sharp2. Display the results in new figure windows.

```
# 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()
```

Variables: ImJPG _Sharp1, ImJPG Sharp2

12. Finally, it is possible to use convolution to detect edges in the image. Edge detection is used for image segmentation and data extraction in areas such as image processing, computer vision, and machine vision. Two of the most common filters used for these are the Sobel horizontal and vertical filters:

$$Kernel\_Sobel1 = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, \ Kernel\_Sobel2 = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}.$$

Sobel filters can be interpreted as discrete derivatives in the horizontal and vertical directions. Type in the matrices Kernel Sobel1, Kernel Sobel2.

Variables: Kernel Sobel1, Kernel _Sobel2

13. Perform the convolution of the original image ImJPG with the Sobel kernels using the function conv2 and save the resulting arrays as ImJPG Sobel1 and ImJPG Sobel2. Display the results in new figure windows.

12 and 13 both code,

```
# 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()
```

Variables: ImJPG _Sobel1, ImJPG Sobel2

14. Create a combined image with both horizontal and vertical edges by summing up the matrices ImJPG _Sobel1, ImJPG Sobel2. Display the result in a new figure window by using the code:

```
# 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()
```

Alternatively, Laplacian edge detection can be used with the following filter:

$$Kernel\_Laplace = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}.$$

Type in the matrixKernel Laplace. The laplace kernel is a discrete analogue of continuous Laplacian and may be interpreted as a sum of two discrete partial derivatives of the second order.

Variables: Kernel Laplace

16. Perform the convolution of the original image ImJPG with the Laplace kernel using the function conv2 and save the resulting array as ImJPG _Laplace. Display the results in a new window.

```
 Ensure the image is RGB (3 channels)
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()
```

Variables: ImJPG _Laplace

## Project 7: Norms, angles, and your movie choices

**Goals:** Use a norm and an inner product to detect similarity between users' choices and to create a simple recommender system.

**To get started:**

- Open a new Python script and save it as lab07.py
- Download the file users movies.mat[4] which contains the arrays movies, users movies, users movies _sort, index small, trial user and place it in your working directory.

**Python commands used:** numpy.load, numpy.shape, for, if, print, len, numpy.prod, numpy.linalg.norm

**What you have to submit:** The file lab07.py which you will create during the lab session.

### INTRODUCTION

Being able to predict users' choices is big business. Many internet services are studying consumer choices and preferences to be able to offer you the products you might like (and correspondingly the products you are more likely to buy). Netflix, Pandora Internet radio, Spotify, Facebook targeted ads, Amazon, and even online dating websites are all examples of recommender systems. These services are sometimes willing to go to great lengths in order to improve their algorithms. In fact, in 2006, Netflix offered 1 million dollars to the team which would be able to improve their Cinematch algorithm predictions by just 10%.

Music Genome Project which is a trademark of Pandora Internet radio is another example of a recommender system. If you have ever listened to Pandora, you might have been amazed how uncannily spot-on their next song recommendation is (at least, the author of this project was certainly impressed by it). Every song is assigned a rating from 0 to 5 in 450 different categories ("genes"). Thus, any song is represented by a vector with 450 components. Mathematical comparison of these vectors allows the algorithm to suggest the next song to a user.

Observe that data about movies, songs, consumer products, etc, can be often arranged in a form of a vector. These vector representations can then be used in different algorithms to compare items and predict similarity. In this project, we will use linear algebra to establish similarities in tastes between different users. The main idea is that if we know the current ratings from a certain user, then by comparing them with the ratings of other users in a database, we can find users with similar tastes. Finally, we can look at the items highly rated by those users and suggest these items to the current user.

In this lab, we will look at the MovieLens database [10] which contains around 1 million ratings of 3952 movies by 6040 users. We will create a very simple recommender system by finding similarities between users' tastes. Our techniques will be based on using the Euclidean distance (norm), the standard scalar product, and finding correlation coefficients (or angles between the vectors of data).

[4] The original data is available at https://grouplens.org/datasets/movielens/.

## TASKS

1. Load the arrays movies, users _movies, users _movies _sort, index small, trial _user from the file users _movies.mat using the load command. The matrix users movies should be a 6040×3952 matrix containing integer values between 0 and 5 with 1 meaning "strongly dislike" and 5 meaning "strongly like". The 0 in the matrix means that the user did not rate the movie. The array movies contains all the titles of the movies. The matrix users movies _ sort contains an extract from the matrix users _movies.mat with rating for the 20 most popular movies selected. The indexes of these popular movies are recorded in the array index small. Finally, ratings of these popular movies by yet another user (not any of the users contained in the database) are given by the vector trial user. It is suggested to view all the variables and their dimensions by using the "Workspace" window of the Python environment:

   Variables: movies, users _movies, users movies sort, index small, trial _user, m, n

   ```
   # Import necessary libraries
   import scipy.io
   import numpy as np

   # 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
   ```

2. Print the titles of the 20 most popular movies by using the following code:

Observe that the movie titles are called from the cell array movies (notice the curly parentheses) by using an intermediate array index small.

```
# 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')
```

3. Now let us select the users we will compare the trial user to. Here, we want to select the people who rated all of the 20 movies under consideration. This means that there should not be any zeros in the corresponding rows of the matrix users movies _sort. This can be accomplished by the following code:

   Observe the use of the command prod to compute the product of the elements of the rows of the array users movies sort. The array ratings contains all the users which have rated all the popular movies from the array index small. In general, it may happen that this array is empty or too small to create any meaningful comparison - we won't consider these cases in this project.

   Variables: ratings, m1, n1

   Q1: What does the command ratings=[] do?

```
# 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}")

# Variables: ratings, m1, n1
```

4. Next, we can look at the similarity metric based on the Euclidean distance. The idea here is that we treat the array trial user and the rows of the array ratings as vectors in 20dimensional real space $R^{20}$. Assume that all the vectors have the origin as the beginning point. We can find the distance between the end points of the vector trial user and each of the vectors ratings(j,:). In other words, we are looking for the user with the closest ratings to the trial _user. This can be accomplished by the following code:

   The vector eucl contains all the Euclidean distances between trial user and the rows of the matrix ratings. Variables: eucl

```
# 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
```

5. Now let us select the user from the database with the smallest Euclidean distance from the trial user. Instead of using the usual function min we will use a slightly more complicated approach. Let us sort the elements of the vector eucl by using the function sort. The advantage of this is that it allows us to find the second closest user, the third closest user, etc. There may only be a small difference between the several closest users and we might want to use their data as well.

Variables: MinDist, DistIndex, closest user Dist

```
# 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
```

6. The similarity metric above is one of the simplest ones which can be used to compare two objects. However, when it comes to user ratings it has certain disadvantages. For instance, what if the users have similar tastes, but one of them consistently judges movies harsher than the other one? The metric above would rate those two users as dissimilar since the Euclidean distance between vectors of their opinions might be pretty large. To rectify this problem, we can look at a different similarity metric known in statistics as the Pearson correlation coefficient which can be defined as

$$r(\mathbf{x},\mathbf{y}) = \frac{\sum_{i=1}^{N}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{N}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{N}(y_i - \bar{y})^2}},$$

where $\mathbf{x} = (x_1,x_2,...,x_N)$, $\mathbf{y} = (y_1,y_2,...,y_N)$ are two vectors with $N$ components, and $\bar{x} = \frac{1}{N}\sum_{i=1}^{N} x_i$, $\bar{y} = \frac{1}{N}\sum_{i=1}^{N} y_i$are the average (mean) values of the vectors **x** and **y**. If we look at the formula for $r(\mathbf{x},\mathbf{y})$, we can see that it accounts for the average user opinion ¯*x*, *y*¯, and also for how harshly/enthusiastically they tend to judge their movies (defined by the magnitude of the vectors **x**−**x**¯ and **y**−**y**¯, where **x**¯ and **y**¯ denote the vectors of the same size as **x** and **y** with all the components equal to ¯*x* and ¯*y* correspondingly). Geometrically, the correlation coefficient $r(\mathbf{x},\mathbf{y})$ corresponds to the cosine of the angle between the vectors **x** − **x**¯ and **y** − **y**¯ (the closer this angle is to zero, the more similar are the opinions of two people). To compute the Pearson correlation coefficient, let us centralize the columns of the matrix ratings and the vector trial user first:

Variables: ratings cent, trial user _cent

```
# 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
```

7. Next, use the for… end loop to compute the Pearson correlation coefficients between the rows of the matrix ratings and the vector trial _user. Save the result as a vector pearson.
   Variables: pearson

```
# 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
```

8. Finally, observe that the value $r(\mathbf{x},\mathbf{y})$ belongs to the interval $(-1,1)$. The closer the coefficient is to 1, the more similar the tastes of the users are. Finally, let us sort the vector pearson as before using the sort function. Save the results of this function as [MaxPearson,PearsonIndex] and find the maximal correlation coefficient which will be the first element in the matrix MaxPearson. Save this element as closest user _Pearson.

Variables: MaxPearson, PearsonIndex, closest user _Pearson

```
# 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: {MaxPearson}")
print(f"Indices of users sorted by Pearson correlation: {PearsonIndex}")
print(f"Index of user with highest Pearson correlation: {closest_user_Pearson}")

# Variables: MaxPearson, PearsonIndex, closest_user_Pearson
```

9. Compare the elements of the vectors DistIndex, PearsonIndex.

Q2: Are the variables closest user _Pearson and closest _user Dist the same?

```
# 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.")
```

10. Now let us display the recommendations on the screen. Use the following list to create the list of movies which the trial user has liked and the lists of recommendations for him/her based on the distance criterion and the Pearson correlation coefficient criterion:

```
# Recommendations recommend_dist=[]; for k=1:n if
(users_movies(closest_user_Dist,k)==5)
recommend_dist=[recommend_dist; k];
        end; end; recommend_Pearson=[]; for k=1:n if
(users_movies(closest_user_Pearson,k)==5)
recommend_Pearson=[recommend_Pearson; k];
        end; end; liked=[]; for k=1:20 if
(trial_user(k)==5) liked=[liked; index_small(k)];
        end;
end;
```

We use the rating equal to 5 both as the criterion for liking the movie and the criterion to recommend the movie. Of course, you can broaden this up and also include the movies ranked as 4.

Variables: liked, recommend dist, recommend Pearson

```
print("index_small shape:", index_small.shape)
print("trial_user shape:", trial_user.shape)
import scipy.io
import numpy as np

# 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
```

11. Finally, display the titles of the movies from the arrays liked, recommend dist, recommend _ Pearson on the screen by using the procedure similar to the one used in the step 2.

```
# 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 criterion
print("Recommended movies based on distance criterion:")
print_movie_titles(recommend_dist, movies)

# Print recommendations based on the Pearson correlation criterion
print("Recommended movies based on Pearson correlation criterion:")
print_movie_titles(recommend_pearson, movies)
```

12. Take a look at the list of the popular movies displayed in step 2. Chances are you might have seen the majority of them before. Create your own vector of ratings and call it myratings. Ratings should be integers between 1 and 5. Again, assign the rating 1 if you really disliked the movie, and 5 if you really liked it. If you haven't seen a particular movie, pick its rating at random. The vector of ratings myratings should be a row-vector with 20 elements.
    Variables: myratings

```
import numpy as np

# Manually specify your ratings for the 20 popular movies
# Example: Dislike some movies (1), Like some movies (5), Random ratings for others
# Replace these values with your own ratings
myratings = np.array([5, 1, 4, 3, 2, 5, 1, 4, 3, 5, 2, 5, 1, 3, 4, 5, 2, 1, 4, 3])

# Ensure myratings is a row vector (1D array with 20 elements)
print("My Ratings Vector:")
print(myratings)
```

13. Create a new code cell. In this cell, repeat steps 4-11 of this project and substitute the vector trial user by the vector myratings. This should produce a personal recommendation based on your own ratings.
    Variables: liked, recommend dist, recommend Pearson

```
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'].flatten()  # Flatten index_small to 1D array

# Define your own ratings vector (myratings)
myratings = np.array([5, 1, 4, 3, 2, 5, 1, 4, 3, 5, 2, 5, 1, 3, 4, 5, 2, 1, 4, 3])

# Step 4: Select users who rated all 20 popular movies
[m1, n1] = users_movies_sort.shape
ratings = [users_movies_sort[j, :] for j in range(m1) if np.all(users_movies_sort[j, :] > 0)]
ratings = np.array(ratings)

# Step 5: Compute Euclidean distances
eucl = np.linalg.norm(ratings - myratings, axis=1)

# Step 6: Find the closest user based on Euclidean distance
MinDist, DistIndex = np.sort(eucl), np.argsort(eucl)
closest_user_Dist = DistIndex[0]

# Step 7: Centralize ratings and myratings for Pearson correlation
ratings_cent = ratings - np.mean(ratings, axis=1, keepdims=True)
myratings_cent = myratings - np.mean(myratings)

# Step 8: Compute Pearson correlation coefficients
pearson = np.sum(ratings_cent * myratings_cent, axis=1) / (
    np.sqrt(np.sum(ratings_cent ** 2, axis=1)) * np.sqrt(np.sum(myratings_cent ** 2))
)

# Step 9: Find the closest user based on Pearson correlation
MaxPearson, PearsonIndex = np.sort(pearson)[::-1], np.argsort(pearson)[::-1]
closest_user_Pearson = PearsonIndex[0]

# Step 10: Create recommendations based on the distance criterion
recommend_dist = [k for k in range(users_movies.shape[1]) if users_movies[closest_user_Dist, k]
== 5]

# Step 11: Create recommendations based on the Pearson correlation criterion
```

```
recommend_pearson = [k for k in range(users_movies.shape[1]) if
users_movies[closest_user_Pearson, k] == 5]

# Create the list of movies liked by the trial user
liked = [index_small[k] for k in range(20) if myratings[k] == 5]

# 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:")
print(liked_titles)

print("Recommended movies based on distance criterion:")
print(recommend_dist_titles)

print("Recommended movies based on Pearson correlation criterion:")
print(recommend_pearson_titles)
```

## Project 8: Interpolation, extrapolation, and climate change

**Goals:** To explore applications of linear algebra for the interpolation of data and prediction of future tendencies.

**To get started:**

- Create a new Python script and save it as lab08.py
- Download the file temperature.mat which contains average temperatures for the past 136 years [5]. Temperature is given in Celsius.

**Python commands used:** matplotlib.pyplot.plot, matplotlib.pyplot.axis, numpy.vander, numpy.interp, numpy.linalg.orth

**What you have to submit:** The file lab08.py, which you will create during the lab session.

### INTRODUCTION

In many practical applications it is necessary to obtain an intermediate value of a parameter from a few known data points. If this intermediate value lies in between two known measurements, we will call this process interpolation. If the value is outside of the sampled interval, then we will call this process extrapolation. In this project we will look at several procedures for interpolation and extrapolation and apply them to weather and climate change prediction.

The most common reason for using interpolation is not having enough of data points. These data points can be obtained from experimental observations which can be expensive or difficult to perform, or from past observations where the data is simply unavailable. Interpolation is also used when the function which produces the data is known but has a complicated structure. In this case it can be computationally prohibitive to use this function, especially if, in the course of computation, this function needs to be called multiple times. For instance, the function can be given by a slowly convergent series which requires computing many terms to obtain adequate accuracy. Approximating this complicated function by a simpler function may be helpful. Interpolation of a function can be sufficiently accurate provided that the function is smooth and there are enough data points available. Some of the usual types of interpolation include polynomial interpolation when the function is approximated by a polynomial, trigonometric approximation with a truncated Fourier series, or various piece-wise approximations. More intricate interpolation methods can be used if additional information is available about the approximated function.

Extrapolation, on the other hand, is a significantly more difficult procedure. The general rule is not to extrapolate too far beyond the range of the observed parameters. There are multiple reasons for this. For instance, some additional parameters may affect the behavior of a function outside of the observed range of values. Imagine trying to extrapolate physical laws applicable for regular macro-solids (such as things we use in our daily life) to nano-scale or to astro-scales. The obtained results will be meaningless since different laws apply at those scales. Greater

[5] The temperature data has been taken from NASA's website, https://climate.nasa.gov/vital-signs/ global-temperature/

uncertainty and errors while extrapolating also appear because small errors in data measurements can potentially lead to very large errors while extrapolating the data. These errors are somewhat bounded while interpolating between two known data points.

**TASKS**

1. To begin, let us look at interpolation based on the polynomial approximation. Assume that we have $n$+1 data points (measurements) $(x_i,y_i)$, $i$ = 0,...,$n$. To obtain the values of the function in between these data points we can simply trace a polynomial of the $n$th degree through the data points and use this polynomial to estimate the values of a function in between measurements. Thus, the goal is to find a polynomial

$$p(x) = a_nx^n + a_{n-1}x^{n-1} + ...a_1x + a_0$$

with unknown coefficients $a_i$, $i$ = 0,...,$n$, which passes through the points $(x_i,y_i)$. Then the coefficients $a_i$ must satisfy the following system of linear algebraic equations

$a_nx^n_i + a_{n-1}x^{n-1}_i + ...a_1x_i + a_0 = 0$, $i$ = 0,...,$n$. We can write

this system in the form

$$\mathbf{Vz} = \mathbf{y},$$

where $V$ is the so-called Vandermonde matrix:

$$\mathbf{V} = \begin{bmatrix} 1 & x_0 & ... & x_0^{n-1} & x_0^n \\ 1 & x_1 & ... & x_1^{n-1} & x_1^n \\ ... & ... & ... & ... & ... \\ 1 & x_n & ... & x_n^{n-1} & x_n^n \end{bmatrix},$$

$\mathbf{z} = [a_0,a_1,...,a_n]^T$ is the unknown vector of polynomial coefficients and $\mathbf{y} = [y_0,y_1,...,y_n]^T$ is the known vector of the function values at sample points.

We will start by approximating the data for the average temperatures in Kansas[6]:

| Month | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Temperature (High) | 37 | 44 | 55 | 66 | 75 | 84 | 89 | 88 | 80 | 69 | 53 | 41 |
| Temperature (Low) | 17 | 23 | 33 | 43 | 53 | 63 | 68 | 65 | 56 | 44 | 32 | 22 |

In our computations, we will work with high temperatures only. To begin, let us generate a plot of the temperatures in Python.

Observe that Python will automatically join the available data points by straight lines. Variables: WeatherHigh

[6] Source for the temperature data: Wikipedia https://en.wikipedia.org/wiki/Kansas#Climate, temperature is given in Fahrenheit.

```
# 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()
```

2. Assume for now, that we are only given the temperatures for four months: January, May, August, and December. Select the corresponding data from the variable WeatherHigh above and generate the Vandermonde matrix based on this data. Now solve the resulting system using Python's backslash linear system solver:
   The resulting vector CoefHigh contains coefficients of the cubic polynomial approximating the average high annual temperature in Kansas. Variables: V, CoefHigh

```
# 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 = 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)
```

3. Python’s numpy.polyval function can be used to efficiently evaluate a polynomial at the given set of points. It takes two vectors as arguments. The first vector represents the coefficients of the polynomial and the second vector represents the points we want to evaluate the polynomial at. Remember that you can always type help(np.polyval) or np.polyval? in the Python command window to see more options. Let us plot the graph of the polynomial obtained at the previous step by using the following code:

   This graph shows the difference between the approximated values and the exact values. Observe that the command axis([1,12 30 95]) sets the parameters for *x* and *y* axes. Namely, the *x* axis will range from *x* = 1 to *x* = 12 and the *y* axis will range from *y* = 30 to *y* = 95.

   Variables: xc, ycHigh

```
# 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()
```

4. Repeat the steps above to produce a polynomial approximation based on the temperatures from the following six months: January, March, May, August, October, and December. You will need to use a polynomial of the fifth degree. Plot the resulting polynomial next to the exact values as in the step above. You will see that this approximation produces much closer results to the exact values.

```
# 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()
```

5. Finally, let us use the temperatures from all twelve months. You will need to use a polynomial of the eleventh degree this time. Modify the code above and run it in Python. If you did everything correctly, you will obtain a picture which looks similar to the one in Fig. 7. You may notice that this approximation is not necessarily better than the

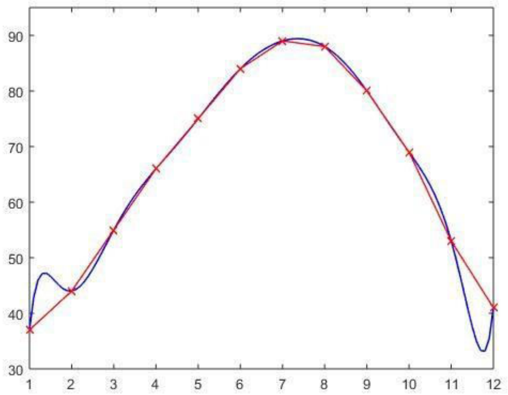


Figure 7: Approximation with the 11th degree polynomial

approximation with the 5th degree polynomial due to large fluctuations near the end points. This is typical for approximations with polynomials of a high degree on evenly spaced grids which tend to oscillate widely near the end points. This is one of the reasons why polynomials of a high degree are normally not used for interpolation on evenly spaced grids [7]. Additionally, if you look at the Python output in the command window, you will probably see something like this

```
Warning: Matrix is close to singular or badly scaled. Results may be inaccurate. RCOND = 8.296438e-17.
```

This is due to the fact that Vandermonde matrix V is an ill-conditioned matrix, meaning that the system of linear equations with this matrix is difficult to solve with high accuracy. The matrix V becomes more ill-conditioned as the dimensions of the matrix increase. This is another reason why high degree polynomial approximations on evenly spaced grids are rarely used in interpolation.

```
# 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)')
```

[7] However, approximations with the polynomials of high degree on non-evenly spaced grids may be extremely useful in computations and achieve very high precision. Chebyshev nodes and Chebyshev polynomials represent a good example of this phenomena.

```
plt.title('11th Degree Polynomial Approximation of High Temperatures')
plt.legend()
plt.grid(True)
plt.show()
```

6. If polynomials of a high degree on evenly spaced grids are not good for interpolation, what can be done instead? The most common method of interpolation is through the approximation of functions with piece-wise (typically polynomial) functions. This means that the function between data points is approximated by a small degree polynomial (polynomials of no higher than third degree are usually used). These polynomials are different on each of the intervals between the data points. Individual approximations are then glued together to achieve a desired degree of smoothness at the data points. Under smoothness here we understand the continuity of a function and its derivatives up to the some order. Smoothness of the resulting approximation depends on the degree of polynomials used and how the polynomial pieces are glued together. If a higher degree of smoothness is desired, it is necessary to use polynomials of a higher degree. For instance, if we use cubic polynomials, then it is possible to achieve continuity of the function itself and its first two derivatives at the most.

   To investigate this method of interpolation, we will look at the built in Python function interp1. This function allows one to interpolate a one-dimensional function. It has several attributes which allow for different types of interpolation: “linear” which simply joins the data points with straight lines, “cubic” or “pchip” which approximates the function on each individual segment with the third degree polynomial, and “spline” which also uses third degree polynomials, but with different junction conditions. So what is the difference between “pchip” and “spline”? Simply put, the “pchip” method provides a shape-preserving piece-wise cubic polynomial function which is smooth together with its first derivative. The “spline” interpolation creates an “extra-smooth” piece-wise cubic polynomial function which is smooth together with its first and second derivatives. In general, “pchip” approximations have smaller oscillations near the ends and are easier to compute.

   Compare these options by using the following code:

   Here the variables x and WeatherHigh give the vectors of initial data. The vector xc is a vector of points where we want to approximate the function and the vectors ycHigh1, ycHigh2, and ycHigh3 are the values at the points xc resulting from the interpolation. Plot the vectors ycHigh1, ycHigh2, and ycHigh3 versus xc on the same plot to observe the smoothness of the resulting data. You can see that ycHigh2 and ycHigh3 achieve better approximations than the polynomial approximation. The function interp1 also has a possibility to extrapolate the results which we will explore shortly. Variables: ycHigh1, ycHigh2, ycHigh3

```
import numpy as np
import matplotlib.pyplot as plt

# 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()
```

7. Now let us look at the extrapolation of the data. Extrapolation means finding an approximation to the data outside of the initial measurement interval. For our data, we will take the average global temperatures from the last 136 years starting from the year 1880[8]. To begin, load the matrix temperature from the file “temperature.mat” using the load command.

   Variables: temperature

   ```
   import scipy.io
   from scipy.io import loadmat
   ```

[8] The source of data: NASA, https://climate.nasa.gov/vital-signs/global-temperature/

```
# Load temperature data
data = loadmat('C:/Users/Harshith/Downloads/laproject/materials/temperature.mat')
temperature = data['temperature']
```

8. The matrix temperature consists of two columns. The first column represents the years and the second the average temperatures at those years. Separate the data into two variables:

   Plot the graph of temperatures using the plot function. This will produce a graph of fluctuations of average temperatures since 1880. If you did everything correctly, you should obtain a graph similar to the one on the Fig. 8. Variables: years, temp

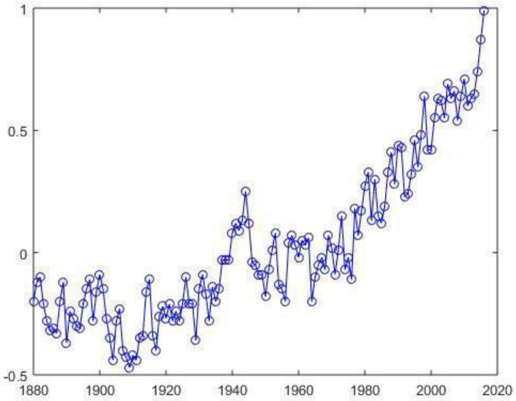


Figure 8: Average yearly temperatures.

```
# 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()
```

9. Let us attempt to extrapolate the data in the future. Assume that we want to know the temperature about 10 years into the future (again, it is advised not to attempt to extrapolate too far beyond the initial interval). We can do this using the following code:

This produces three graphs based on the different interpolation/extrapolation strategies which were discussed above. If you did everything correctly, you should obtain three radically different predictions for the next 10 years!

Variables: futureyears, futuretemp1, futuretemp2, futuretemp3

```
# 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()
```

10. So what happened in the previous example? The local fluctuations in temperature significantly affected the global predictions for the future. However, we are not interested in the local fluctuations but rather in global tendencies. So let us try to separate those two from each other and extrapolate based on global tendencies only. We can think of temperatures in the following terms: $t_i = t(year_i) + noise$,

    where $t(year)$ would be the global tendency of temperature and *noise* represents relatively small year-to-year fluctuations.

    We will use orthogonal projections to separate the global behavior from the “noise”. As a first approximation, let us consider how we can assess the overall average temperature. That is, we approximate the temperature $t(years)$ as a constant:

$$t(years) \approx \frac{1}{n}\sum_{i=1}^{n} t_i \quad .$$

Enter the following command in the command window

```
>> sum(temp)/n
```

This allows us to see that the average temperature for the past 136 years was 0.0244C.

11. Here is another way of thinking of the same operation using orthogonal projections. If the data was truly flat, then there would be a real number $c$ so that

$$\mathbf{t} = \begin{bmatrix} t_1 \\ t_2 \\ \vdots \\ t_n \end{bmatrix} = c\begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix} = c\mathbf{1}.$$

That is, the average temperature vector $\mathbf{t}$ would be a scalar times the vector $\mathbf{1}$ of all ones. Since the data is not flat, we can't find such a $c$. However, we can do the next best thing. We can find $c$ to minimize the norm $\|\mathbf{t} - c\mathbf{1}\|$. The best choice for $c$ is determined by the orthogonal projection on a vector $\mathbf{1}$:

$$c = \frac{\mathbf{1}^T\ \mathbf{t}}{\|\mathbf{1}\|^2} \quad \text{and for this choice of } c, \quad c\mathbf{1} = \frac{\mathbf{1}^T\mathbf{t}}{\|\mathbf{1}\|^2}\mathbf{1} = \text{proj}_{\mathbf{1}}(\mathbf{t}).$$

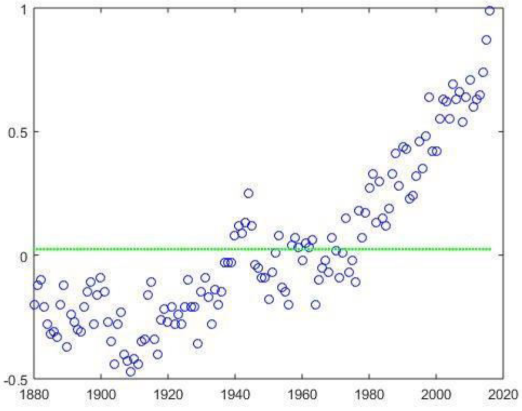


Figure 9: Approximation of data by a constant function.

Moreover, a little rearranging shows that

$$\text{proj}_{\mathbf{1}}(\mathbf{t}) = \left(\frac{1}{\|\mathbf{1}\|^2}\mathbf{1}\mathbf{1}^T\right)\mathbf{t} = \mathbf{P}\mathbf{t} \quad \text{where} \quad \mathbf{P} = \frac{1}{\|\mathbf{1}\|^2}\mathbf{1}\mathbf{1}^T \quad \text{is the projection matrix.}$$

Define the variable b1 to be the length(temp)-dimensional **1** vector. In the space after that, define the variable P1 to hold the projection matrix. You will either want to use√

Python's norm function, or remember that k**u**k = $\mathbf{u}^{T}\mathbf{u}$. Also, define the variable temp1 to be the value P1*temp. Variables: P1, temp1

12. Plot the resulting vector using the code:

    You should see a similar plot as before, but now with a set of green dots representing the projection (Fig. 9).

13. Now, examine the value of temp1 in the Python command window.
    Q1: How does the value of temp1 relate to the average value of temperature?

Solution for 10,11,12,13 is combined.

```
# 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()
```

14. Like vectors, matrices also have norms. The norm k**A**k of a matrix **A** describes how far **A** is from the zero matrix in a certain sense. Enter the following command into Python

Your answer should be something like ans = 2.0134e-15.

Q2: What does the result say about the relationship between the projection matrix **P** and its square $\mathbf{P}^2$?

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

15. While the previous approach of approximating the data with a constant function may be acceptable in some cases, it is clearly not a perfect match. Let us try instead to model the data as a linear function of $years_i$:

$$t_i = byears_i + c + \text{noise}.$$

If this formula was exact (with no noise), we would have that

$$\mathbf{t} = \begin{bmatrix} t_1 \\ t_2 \\ \vdots \\ t_n \end{bmatrix} = \begin{bmatrix} years_1 & 1 \\ years_2 & 1 \\ \vdots & \vdots \\ years_n & 1 \end{bmatrix} \begin{bmatrix} b \\ c \end{bmatrix} = b\,\mathbf{years} + c\mathbf{1},$$

so **t** would be a vector in the subspace span{**years**,**1**}. However, as before, this is not exactly true. So, we will need to use an orthogonal projection again.

There is a nice formula for projecting onto a subspace S generated by a span of several vectors span{$\mathbf{b}_1$,$\mathbf{b}_2$,··· ,$\mathbf{b}_k$}, but it requires that the vectors be *orthonormal*. That is k$\mathbf{b}_i$k = 1 for each $i$, and $\mathbf{b}^T_i\mathbf{b}_j = 0$ for all $i$ 6= $j$.

One way to produce an orthonormal basis in Python is to use the Python orth function. Please, type in the code

This creates a matrix **B** = [**1 years**], whose first column consists entirely of ones and whose second column is the vector **years**, which provides the years of observation. The new array Q2 is defined to be the result of orth(B2).

The function orth will create an orthonormal basis (defined by the columns of Q2) for the subspace S. To check that the columns of Q2 form a basis, verify in the command window that the values of rank(Q2) and rank([Q2,B2]) are both equal to 2.

Variables: Q2

Q3: What kind of matrix is the matrix $\mathbf{Q2}^T\mathbf{Q2}$? Why?

```
from scipy.linalg import orth
# 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)
```

16. Given an orthonormal basis defined by the columns of Q2 for the subspace S, the formula for the projection matrix onto the subspace S is simple:

$$\mathbf{P2} = \mathbf{Q2Q2}^{T}.$$

Create a new cell and in this cell create a variable P2 to hold the value of the matrix of the orthogonal projection onto the subspace S. Repeat the previous steps to produce the projection vector temp2=P2*temp and plot this vector in the previous figure window. This will produce the linear approximation to the given data. Run the following code in Python:

You should get a number which is close to zero again, such as, 5.6169e-16. There is a geometric explanation for this fact. The matrices P1, P2 are the projection matrices, and projecting a vector on a subspace produces a vector belonging to this subspace. Taking a second projection, thus, will simply leave the vector as is. The projection matrix behaves as an identity matrices on the corresponding subspace if the basis consisting of the columns of the matrix **Q2** is selected.

Variables: P2, temp2

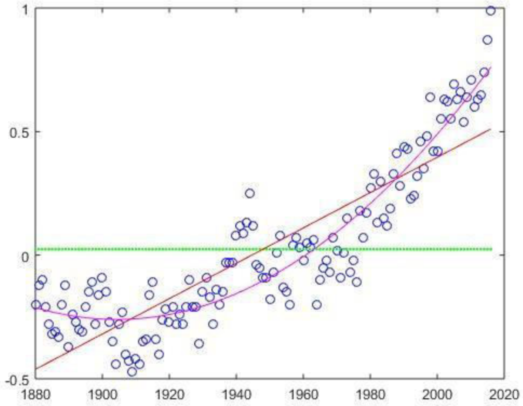


Figure 10: Approximation of data by a quadratic function.

```
# 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}")
```

17. Finally, let us approximate the data by using a quadratic function. Create a new cell and reproduce the code from from the previous cell but with renamed variables (e.g., B3 instead of B2). Make sure to copy the plotting code and switch P2 to P3.

    Aside from changing 2s to 3s, the only other change you'll need to make is to add a column to the B3 matrix. This column should be the vector of squares of the elements of the vector years, which can be produced in Python with the expression years.^2 (notice the dot before the power sign!).

    Variables: B3, Q3, P3

18. Plot the initial data and all the approximations above on a new plot (use the command figure). If correct, your code should produce a figure similar to the one on the Fig. 10. Observe that the quadratic function (shown by the magenta line) appears to follow the global tendencies of the temperature graph pretty well.

    Combined solution for 17,18

```
# 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}")
```

19. Now, let us make predictions for the future. Create a new cell and run the following code:

    This should create our prediction for an average temperature for the next 100 years. If you did everything correctly, provided the current tendencies hold, by the year 2116 the average temperature should be about 3.5C. Note that climate prediction is notoriously difficult, and it is very likely that there are factors which influence the data which are not accurately picked up by our simple model. Also, please note that in the past, temperatures tended to rise faster than any prediction. At the same time, NASA reports that the currently available models for climate change predict rise in temperatures anywhere between a 2C and 6C, so our simple model predictions are right in the middle!

    ```
    #19

    # Define future years for prediction
    futureyears = np.arange(2016, 2117)

    # Perform spline interpolation and extrapolation for future temperatures
    interp_func = interp1d(years, temp3, kind='cubic', fill_value='extrapolate')
    futuretemp3 = interp_func(futureyears)

    # Create a new figure for plotting
    plt.figure()

    # Plot future temperature predictions
    plt.plot(futureyears, futuretemp3, 'g-')
    plt.xlabel('Year')
    plt.ylabel('Predicted Average Temperature (°C)')
    plt.title('Predicted Average Temperature for Next 100 Years')
    plt.grid()

    # Display the predicted temperature for 2116
    print(f'Predicted average temperature for the year 2116: {futuretemp3[-1]:.2f} °C')

    plt.show()
    ```

## Project 9: Orthogonal matrices and 3D graphics

**Goals:** To explore applications of orthogonal matrices to rotation of 3D objects.

**To get started:** Download the files lab09.py[9], v.mat, and f.mat. The last two files contain corresponding vertices and faces of a 3D model[10]. Save these files in your working directory.

**Python commands used:** def, np.zeros, np.sin, np.cos, if ... else, for ... in range, plt.plot, plt.figure, plt.hold, buckyball, len, np.load."

**What you have to submit:** The files rotation.m and lab09.py which you will create during the lab session.

### INTRODUCTION

Recall that the square matrix **Q** is called orthogonal if $\mathbf{Q}^T\mathbf{Q} = \mathbf{Q}\mathbf{Q}^T = \mathbf{I}$, where **I** is an identity matrix of the same size as **Q**. In other words, this means that $\mathbf{Q}^T = \mathbf{Q}^{-1}$. Rows (columns) of an orthogonal matrix generate an orthonormal set of vectors. Namely, if $\mathbf{Q} = [\mathbf{q}_1,\mathbf{q}_2,...,\mathbf{q}_n]$, where $\mathbf{q}_j$ are the columns of the matrix **Q**, then $\mathbf{q}_j \cdot \mathbf{q}_k = \delta_{jk}$, and $\delta_{jk}$ denotes the Kronecker symbol: $\delta_{jk} = 1$, if $j = k$, and $\delta_{jk} = 0$, if $j$ 6= $k$.

Another property of orthogonal matrices is that $\mathbf{x}\cdot\mathbf{y} = \mathbf{Q}\mathbf{x}\cdot\mathbf{Q}\mathbf{y}$ for any two vectors **x** and **y**, meaning that multiplication by an orthogonal matrix **Q** does not change the scalar product of two vectors. Geometrically, this means that a linear transformation $T(\mathbf{x}) = \mathbf{Q}\mathbf{x}$ of the space $\mathrm{R}^n$ is equivalent to the rotation of the space together with a possible reflection along some hyperplane.

[9] The printout of the file lab09.m can be found in appendices of this book.

[10] The original data for this 3D model can be found here: https://www.thingiverse.com/thing:906692. The model is licensed under Creative Commons Attribution Share Alike license.

Whether a reflection of the space is involved can be seen by looking at the determinant of the matrix **Q**. If det**Q** = 1, then the transformation is a pure rotation. If det**Q** = −1, then the transformation is a rotation together with a reflection.

Orthogonal matrices can be used in 3D graphics. Let us consider a three-dimensional object which we want to show on a computer screen. The obvious difficulty is that the object is threedimensional and the screen is only two-dimensional. Thus, we need to generate a projection to show the object on the screen. One of the easiest ways to do this is to simply drop one of the coordinates. However, that allows us to view the object only from three different angles which can make it difficult to understand the real shape of the object. Using orthogonal matrices allows us to look at the object from many different angles. To accomplish this, we need three matrices which represent rotations around each of the coordinate axes *Ox*, *Oy*, and *Oz*:

$$\mathbf{R}_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta_x & -\sin\theta_x \\ 0 & \sin\theta_x & \cos\theta_x \end{bmatrix}, \quad \mathbf{R}_y = \begin{bmatrix} \cos\theta_y & 0 & -\sin\theta_y \\ 0 & 1 & 0 \\ \sin\theta_y & 0 & \cos\theta_y \end{bmatrix}, \quad \mathbf{R}_z = \begin{bmatrix} \cos\theta_z & -\sin\theta_z & 0 \\ \sin\theta_z & \cos\theta_z & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Here $\theta_x$, $\theta_y$, and $\theta_z$ are the angles of rotation around the axes $0x$, $0y$, and $0z$ correspondingly in the counterclockwise direction. Multiplying these matrices will result in a combined rotation around all three axes:

$$\mathbf{R} = \mathbf{R}_x\mathbf{R}_y\mathbf{R}_z.$$

In this project we will use the matrix **R** to create a two-dimensional projection of a threedimensional object.

In computer graphics, most objects are not actually smooth, but rather represented by a polytope. In other words, they are obtained by identifying vertices on the surface of an object and connecting those vertices with straight lines (edges). Thus, the surface of an object actually consists of flat faces. In 3D graphics the faces are usually triangular. There is a simple mathematical reason for this: there always exists a plane passing through any three points in a space but not necessary through four or more points. If the number of faces is large, then the object observed from far away appears to be smooth.

Two things are necessary to create a three-dimensional computer model of an object: coordinates of the vertices and the edges showing the connections between the vertices. The former can be given in a form of an $n \times 3$ array $V$, where $n$ is the number of vertices in a 3D model. The latter can be given in several different ways, for instance, by an $n \times n$ adjacency matrix $E$ (hence, a 3D model can be treated as a graph!) or by an $m \times 3$ array containing numbers of vertices in each face of the model where $m$ is the number of faces. We will look at both of these descriptions.

Observe that this project is intended as a demonstration of techniques used to create a projection of a three dimensional object on a computer screen. Of course, Python has built in procedures to work with 3D objects. We will explore these procedures at the end of the project as well.

**TASKS**

1. First of all, let us create a user defined function which will return a rotation matrix based on the user-specified angles $\theta_x$, $\theta_y$, and $\theta_z$ which correspond to the rotations around the

axes 0*x*, 0*y*, and 0*z*. Create a new Python script and save it as rotation.m in your working directory. Define the header of this function as:

This function takes in three variables theta x,theta y,theta z and returns the 3 × 3 rotation matrix rotmat.

Variables: theta _x, theta y, theta _z

2. Within the function, create the matrices Rx, Ry, and Rz as defined above by using standard Python syntax and the trigonometric functions sin, cos. Variables: Rx, Ry, Rz

3. Compute the product of these matrices and assign the result to the output variable rotmat. Variables: rotmat

4. Save the resulting function in the file rotation.m in your working directory. Now that the user-defined function is created, we can use it to generate projections.

   Combined solution for 1,2,3,4

```
import numpy as np


def rotation(theta_x, theta_y, theta_z):
  # Rotation matrix around the x-axis
  Rx = np.array([[1, 0, 0],
          [0, np.cos(theta_x), -np.sin(theta_x)],
          [0, np.sin(theta_x), np.cos(theta_x)]])

  # Rotation matrix around the y-axis
  Ry = np.array([[np.cos(theta_y), 0, -np.sin(theta_y)],
          [0, 1, 0],
          [np.sin(theta_y), 0, np.cos(theta_y)]])

  # Rotation matrix around the z-axis
  Rz = np.array([[np.cos(theta_z), -np.sin(theta_z), 0],
          [np.sin(theta_z), np.cos(theta_z), 0],
          [0, 0, 1]])
```

```
    # Combined rotation matrix

    rotmat = Rz @ Ry @ Rx

    return rotmat
```

5. Let us start by creating a projection of a simple object such as a cube. Open the file m551lab11.m and notice the following code in the first code cell # Cube:

   This code will create the matrix of the vertices of the cube and the corresponding adjacency matrix.
   Variables: Vertices, Edges

```
import numpy as np
import matplotlib.pyplot as plt

# Define cube vertices
Vertices = np.array([[1, 1, 1],
              [-1, 1, 1],
              [1, -1, 1],
              [1, 1, -1],
              [-1, -1, 1],
              [-1, 1, -1],
              [1, -1, -1],
              [-1, -1, -1]])

# Define adjacency matrix (Edges)
Edges = np.zeros((8, 8))
Edges[0, 1] = 1
Edges[0, 2] = 1
Edges[0, 3] = 1
Edges[1, 4] = 1
Edges[1, 5] = 1
Edges[2, 4] = 1
Edges[2, 6] = 1
Edges[3, 5] = 1
Edges[3, 6] = 1
Edges[4, 7] = 1
Edges[5, 7] = 1
Edges[6, 7] = 1
Edges = Edges + Edges.T  # Make the matrix symmetric
```

6. Assign the following values to the angles of the rotations $\theta_x = \pi/3$, $\theta_y = \pi/4$, and $\theta_z = \pi/6$. Call the function rotation to generate the rotation matrix with these angles:

Variables: rotmat

```
# Define rotation angles
theta_x = np.pi / 3  # 60 degrees
theta_y = np.pi / 4  # 45 degrees
theta_z = np.pi / 6  # 30 degrees

# Generate the rotation matrix
rotmat = rotation(theta_x, theta_y, theta_z)
```

7. Transform the coordinates of the vertices with the rotation matrix.

Variables: VertRot

```
# Rotate the vertices
VertRot = Vertices @ rotmat.T  # Transpose the rotation matrix
```

8. Create a new figure window. Moving through the elements of the matrix Edges, draw a projection of the cube:

The function line draws a line through two points given by their coordinates. The points can have two or three coordinates. Observe that to generate the projection, we dropped the last coordinate of the vector VertRot.
Q1: Why does the second for cycle start with j+1?

```
# Create a new figure window
plt.figure()
plt.axis('equal')
plt.title('Projection of Rotated Cube')

# Draw the projection of the cube
for j in range(8):
   for k in range(j + 1, 8):  # Start with j + 1 to avoid repeating lines
      if Edges[j, k] == 1:
         # Draw lines connecting the vertices (projecting by dropping the last coordinate)
         plt.plot([VertRot[j, 0], VertRot[k, 0]], [VertRot[j, 1], VertRot[k, 1]], 'b-')

plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.grid()
plt.show()
```

9. Modify the code above to obtain the projection generated by dropping the second coordinate of the vector VertRot.

```
# Create a new figure window
plt.figure()
plt.axis('equal')
plt.title('Projection of Rotated Cube (Dropping Y-coordinate)')

# Draw the projection of the cube by dropping the Y-coordinate
for j in range(8):
  for k in range(j + 1, 8):
    if Edges[j, k] == 1:
      # Draw lines connecting the vertices (projecting by dropping the Y coordinate)
      plt.plot([VertRot[j, 0], VertRot[k, 0]], [VertRot[j, 2], VertRot[k, 2]], 'b-')  # Use the Z coordinate instead of Y

plt.xlabel('X-axis')
plt.ylabel('Z-axis')  # Update label to reflect the projection
plt.grid()
plt.show()
```

10. Let us consider a more complicated object. Python has a built in buckyball (bucky) procedure which produces an array containing the vertices of a buckyball and the adjacency matrix showing how to connect the vertices:

    Variables: Edges2, Vertices2

11. Using the same procedure as above, produce a projection of the buckyball onto the screen. You can use the same rotation matrix rotmat as before. Find the number of vertices in the array Vertices2 using the size function.

12. To draw a three dimensional buckyball, use the following code:

    Observe that you can use the “3D rotation” button in the top panel of the figure window to rotate the figure. If you use the same button on the previous figure, you will notice that the generated projection is flat. If you did everything correctly then you will see something similar to the Fig. 11.
    Q2: Why do we use 60 as the end limit for our for cycles?

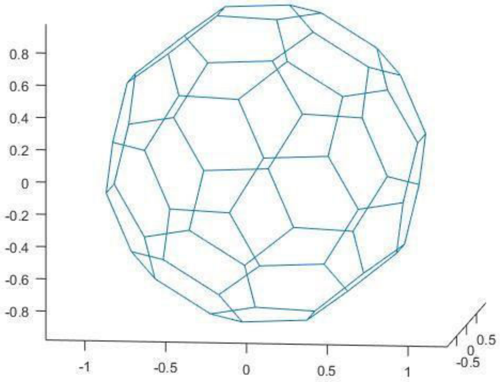


Figure 11: Buckyball in Python window.

```
#10,11,12

import math
import itertools
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

def distance(a, b):
    '''Calculates the straight line distance between two points a and b.'''
    return np.linalg.norm(np.array(a) - np.array(b))

def makecoords():
    '''Generate a list of coordinates for the buckyball.'''
    phi = 0.5 * (1 + math.sqrt(5))
    c1 = (0, 1, 3 * phi)
    c2 = (2, (1 + 2 * phi), phi)
    c3 = (1, 2 + phi, 2 * phi)

    combos1 = list(itertools.product((1, -1), repeat=2))
    for i in range(len(combos1)):
        combos1[i] = (1,) + combos1[i]

    combos23 = list(itertools.product((1, -1), repeat=3))
    coords = []

    for i in combos1:
        coords.append(np.array(c1) * np.array(i))
    for i in combos23:
        coords.append(np.array(c2) * np.array(i))
        coords.append(np.array(c3) * np.array(i))

    # Permutation matrices
    P1 = np.array([[0, 0, 1], [1, 0, 0], [0, 1, 0]])
    P2 = np.array([[0, 1, 0], [0, 0, 1], [1, 0, 0]])

    for i in coords[:]:
        coords.append(P1 @ i)
        coords.append(P2 @ i)

    return coords

def makeadjmat(coords):
    '''Make a 60x60 adjacency matrix for the coordinates.'''
    D = np.zeros((60, 60))

    for i in range(len(coords)):
        for j in range(len(coords)):
            if distance(coords[i], coords[j]) == 2.0:
                D[i][j] = 1

    return D

def rotation(theta_x, theta_y, theta_z):
    '''Create a rotation matrix based on the specified angles.'''
    rot_x = np.array([[1, 0, 0],
                      [0, np.cos(theta_x), -np.sin(theta_x)],
                      [0, np.sin(theta_x), np.cos(theta_x)]])

    rot_y = np.array([[np.cos(theta_y), 0, np.sin(theta_y)],
                      [0, 1, 0],
                      [-np.sin(theta_y), 0, np.cos(theta_y)]])

    rot_z = np.array([[np.cos(theta_z), -np.sin(theta_z), 0],
                      [np.sin(theta_z), np.cos(theta_z), 0],
                      [0, 0, 1]])

    return rot_z @ rot_y @ rot_x  # Combined rotation matrix

def plot_buckyball(coords, edges, rotmat):
    '''Plot the 3D projection of the buckyball.'''
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    ax.set_title('3D Projection of Buckyball')

    # Apply the rotation matrix to the coordinates
    rotated_coords = [np.dot(rotmat, vertex) for vertex in coords]

    num_vertices = len(coords)
    for j in range(num_vertices):
        for k in range(j + 1, num_vertices):
            if edges[j, k] == 1:
                ax.plot([rotated_coords[j][0], rotated_coords[k][0]],
                        [rotated_coords[j][1], rotated_coords[k][1]],
                        [rotated_coords[j][2], rotated_coords[k][2]], 'b-')

    ax.set_xlabel('X')
    ax.set_ylabel('Y')
    ax.set_zlabel('Z')
    plt.show()

if __name__ == "__main__":
    # Task 10: Generate coordinates for the buckyball
    coords = makecoords()

    # Generate the adjacency matrix
    edges = makeadjmat(coords)

    # Find and print the number of vertices
    num_vertices = len(coords)
    print("Number of vertices in the array Vertices2:", num_vertices)

    # Define rotation angles (in radians)
    theta_x = np.pi / 3
    theta_y = np.pi / 4
    theta_z = np.pi / 6

    # Generate the rotation matrix
    rotmat = rotation(theta_x, theta_y, theta_z)

    # Task 11: Plot the 3D projection of the buckyball
    plot_buckyball(coords, edges, rotmat)
```

13. Finally, this can be applied to any three dimensional construction. To illustrate this consider a model for 3D printing[11]. Load the matrices v and f into Python from the files v.mat and f.mat using load command. The array v contains the vertices of the model, and the array f contains the triangular faces of the model. Each row of the face array f contains the numbers which identify the vertices in the face. The corresponding coordinates of the vertices can then be found by referring to the array v. Variables: v, f

```
import numpy as np
from scipy.io import loadmat

data_v = loadmat("C:/Users/Harshith/Downloads/laproject/materials/v.mat")
data_f = loadmat("C:/Users/Harshith/Downloads/laproject/materials/f.mat")

# Extract the vertices (v) and faces (f) from the loaded data
v = data_v['v']  # Ensure the key matches the variable name in the .mat file
f = data_f['f']  # Ensure the key matches the variable name in the .mat file

# Variables: v, f
print("Vertices (v):")
print(v)
print("\nFaces (f):")
print(f)
```

14. Determine the size of the matrix f by using size function. Save the dimensions of this matrix as mFaces, nFaces. The number of rows mFaces in the array f will give you the number of faces in the model, while the number of columns nFaces will give you the number of vertices in each face (it should be equal to three).
Variables: mFaces, nFaces .

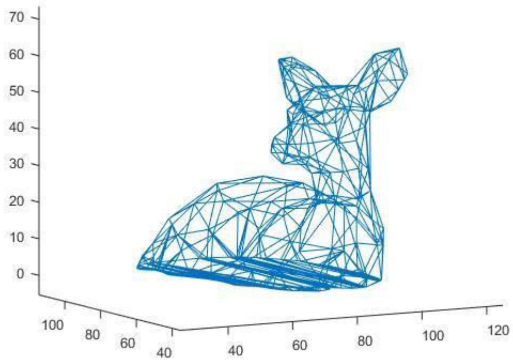


Figure 12: 3D model of a fawn.

[11] The original data for the model is available from https://www.thingiverse.com/thing:906692/#files

```
mFaces, nFaces = f.shape  # Get the number of rows and columns in f

# Output the dimensions of f
print("\nDimensions of the face matrix f,")
print("Number of faces (mFaces):", mFaces)
print("Number of vertices per face (nFaces):", nFaces)
```

15. To generate the three dimensional model, use the following code:

    If you did everything correctly you will see a figure similar to Fig. 12.

```
# Get the number of faces
mFaces = f.shape[0]

# Generate the 3D model
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.set_box_aspect([1, 1, 1])  # Set aspect ratio to equal

# Loop through each face and plot the edges
for j in range(mFaces):
  # Draw lines between the vertices of each face
  ax.plot([v[f[j, 0] - 1, 0], v[f[j, 1] - 1, 0]], [v[f[j, 0] - 1, 1], v[f[j, 1] - 1, 1]],
      [v[f[j, 0] - 1, 2], v[f[j, 1] - 1, 2]], color='b')  # Edge between vertex 1 and 2
  ax.plot([v[f[j, 0] - 1, 0], v[f[j, 2] - 1, 0]], [v[f[j, 0] - 1, 1], v[f[j, 2] - 1, 1]],
      [v[f[j, 0] - 1, 2], v[f[j, 2] - 1, 2]], color='b')  # Edge between vertex 1 and 3
  ax.plot([v[f[j, 1] - 1, 0], v[f[j, 2] - 1, 0]], [v[f[j, 1] - 1, 1], v[f[j, 2] - 1, 1]],
      [v[f[j, 1] - 1, 2], v[f[j, 2] - 1, 2]], color='b')  # Edge between vertex 2 and 3

# Set labels for the axes
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')

# Set the viewpoint (azimuth, elevation)
ax.view_init(elev=10, azim=210)  # You can change the angles here

# Show the 3D plot
plt.show()
```

16. Using a similar procedure as above, generate a two dimensional projection of the three dimensional model. You can use the same rotation matrix rotmat as before. Store the coordinates of the projection in the matrix VertRot. Variables: VertRot

```
theta1 = np.pi / 3  # Rotation around x-axis
theta2 = np.pi / 4  # Rotation around y-axis
theta3 = np.pi   # Rotation around z-axis

# Generate the rotation matrix
rotmat = rotation(theta1, theta2, theta3)

# Transform the coordinates of the vertices with the rotation matrix
VertRot = v @ rotmat.T  # Apply rotation

# Create a new figure window for the 2D projection
plt.figure()
plt.axis('equal')
plt.title("2D Projection of the 3D Model")

# Plot the 2D projection by connecting the edges defined in f
for j in range(f.shape[0]):
    plt.plot([VertRot[f[j, 0] - 1, 0], VertRot[f[j, 1] - 1, 0]],
             [VertRot[f[j, 0] - 1, 1], VertRot[f[j, 1] - 1, 1]], color='b')
    plt.plot([VertRot[f[j, 0] - 1, 0], VertRot[f[j, 2] - 1, 0]],
             [VertRot[f[j, 0] - 1, 1], VertRot[f[j, 2] - 1, 1]], color='b')
    plt.plot([VertRot[f[j, 1] - 1, 0], VertRot[f[j, 2] - 1, 0]],
             [VertRot[f[j, 1] - 1, 1], VertRot[f[j, 2] - 1, 1]], color='b')

# Set labels for the axes
plt.xlabel('X')
plt.ylabel('Y')
plt.grid()
plt.show()
```

17. Another way to generate an image like that in Fig. 17 is as follows.

Variables: rotmat2, vRot, vPrj

```
# Define the new rotation angles
theta1 = -np.pi / 3  # Rotation around x-axis
theta2 = 0           # No rotation around y-axis
theta3 = np.pi / 4   # Rotation around z-axis

# Generate the rotation matrix
```

```
rotmat2 = rotation(theta1, theta2, theta3)

# Rotate the vertices
vRot = v @ rotmat2.T  # Apply rotation

# Project to the XY plane
vPrj = vRot[:, :2]  # Keep only the first two coordinates

# Create a new figure window for the 2D projection
plt.figure()
plt.axis('equal')
plt.title("2D Projection of the 3D Model using rotmat2")

# Plot the 2D projection by connecting the edges defined in f
for j in range(f.shape[0]):
    plt.plot([vPrj[f[j, 0] - 1, 0], vPrj[f[j, 1] - 1, 0]],
             [vPrj[f[j, 0] - 1, 1], vPrj[f[j, 1] - 1, 1]], color='b')
    plt.plot([vPrj[f[j, 0] - 1, 0], vPrj[f[j, 2] - 1, 0]],
             [vPrj[f[j, 0] - 1, 1], vPrj[f[j, 2] - 1, 1]], color='b')
    plt.plot([vPrj[f[j, 1] - 1, 0], vPrj[f[j, 2] - 1, 0]],
             [vPrj[f[j, 1] - 1, 1], vPrj[f[j, 2] - 1, 1]], color='b')

# Set labels for the axes
plt.xlabel('X')
plt.ylabel('Y')
plt.grid()
plt.show()
```

# Project 10: Discrete dynamical systems, linear transformations of the plane, and the Chaos Game

**Goals:** To use matrix mappings of the plane $R^2$ in the form of a Chaos Game in order to produce fractals.

**To get started:** Download the Python script lab10.py[12] and save it in your working directory.
**Python commands used:** np.linspace, for … in range, np.random.rand, np.random.randint, plt.plot, plt.hold, np.cos, np.sin."

**What you have to submit:** The file lab10.py which you will modify during the lab session.

### INTRODUCTION

Many processes can be described by using dynamical systems. The main idea here is that the state of a system in the current moment may be derived by knowing the state of the system at the previous moment(s) of time. For instance, if we know that the velocity of a car increases with the acceleration $a = 2m/s^2$ and the initial velocity of the car is $v_0 = 15m/s$, then we can easily compute the velocity of the car after $t$ seconds by using the formula $v = at + v_0$. This is one of the simplest examples of dynamical systems. Motion of celestial bodies and population dynamics of species are some of the examples of dynamical systems in nature. If the time $t$ is measured continuously then we are talking about continuous dynamical systems. If the time is measured discretely (only at certain points), then we are talking about discrete dynamical systems. For instance, the population of certain species at any moment of time depends on its population in the previous years. Since offspring for many species is produced only once a year, we can look at this population dynamics as a discrete dynamical system with time measured in years.

In this project we will explore the connection between fractals and matrix (linear) mappings in the form of the so-called Chaos Game. The Chaos Game is an algorithm first proposed by M. Barnsley in 1998 in his book "Fractals everywhere".

The basic idea of the Chaos Game is simple. Pick $n$ points on the plane (say, vertices of a regular $n$-gon), and a number $r$, $0 < r < 1$. Pick a random point inside the polygon. Then repeat the following steps: pick a polygon vertex at random and draw the next point at a fraction $r$ of the distance between the current point and the chosen vertex of the polygon. Repeat the process with the new point.

This process can be written in the form of the following linear mapping. Let $\mathbf{x} = (x_1,x_2)$ be the current point, and let $\mathbf{v} = (v_1,v_2)$ be the selected vertex of the polygon. Then the next point can be generated by using the linear mapping $T$:

$$T(\mathbf{x}) = \begin{bmatrix} T_{11} & T_{12} \\ T_{21} & T_{22} \end{bmatrix} (\mathbf{x} - \mathbf{v}) + \mathbf{v}.$$

Here $T(\mathbf{x})$ is the next point. Repeated many times, this process sometimes (but not always) produces a fractal. The mapping $T$ can change from iteration to iteration or can be picked at random with different probabilities (we will look at the examples of this type later). In the tasks below, we will look at some limiting sets of points which can be obtained by this procedure.

[12] The printout of the file lab10.m can be found in the appendices of this book.

## TASKS

1. Let us start simple by creating an image of the Sierpinski Triangle using the Chaos Game. Locate the cell titled # Sierpinski Triangle. In this cell, define the vertices of an equilateral triangle by using the following commands

    ```
    clear; t=linspace(0,2*pi,4);
    t(4)=[]; v=[cos(t); sin(t)];
    ```

    This code creates a set of vertices for equilateral triangle. Each vertex is a distance of 1 unit away from the origin and the first vertex is located on the *Ox* axis.

    Variables: t,v

    Q1: What does the command t[3] do?

    ```
    import numpy as np

    # Generate linearly spaced values
    t = np.linspace(0, 2 * np.pi, 4)
    # Remove the fourth element
    t = np.delete(t, 3)

    # Define the vertices of the equilateral triangle
    v = np.array([np.cos(t), np.sin(t)])
    ```

2. Next, define the starting point and the matrix of the linear transformation:

    The linear transformation with the matrix T is equivalent to multiplying the vector by 1/2, and the starting point x(:,1) is picked at random. Variables: T,x

    ```
    import numpy as np

    # Define the linear transformation matrix
    T = np.array([[0.5, 0], [0, 0.5]])

    # Define the starting point with random values between -0.5 and 0.5
    x = np.random.rand(2, 1) - 0.5
    ```

3. Plot the vertices of the triangle on the plane and select the number of iterations of the process:

    All the points which will be obtained during the iterative process will be stored in the $2 \times N$ array x where $N$ is the number of iterations. After computing all the iterations we need, we will plot all the points from the array x using the command plot. Observe that while it is possible to immediately plot the point we obtain at each step, multiple calls to the plot function take a significant amount of computational time and, thus, doing so is impractical.

Variables: Num

```
import numpy as np
import matplotlib.pyplot as plt

# Define the vertices of the triangle
t = np.linspace(0, 2 * np.pi, 4)[:-1]  # Generate t and remove the last element
v = np.array([np.cos(t), np.sin(t)])

# Define the linear transformation matrix
T = np.array([[0.5, 0], [0, 0.5]])

# Define the starting point
x = np.random.rand(2, 1) - 0.5

# Number of iterations
Num = 10000

# Create an array to store all points
points = np.zeros((2, Num))

# Initial point
current_point = x

# Iterative process
for i in range(Num):
   # Store the current point
   points[:, i] = current_point.flatten()
   # Choose a random vertex
   vertex = v[:, np.random.choice(3)]
   # Apply the linear transformation and update the point
   current_point = T @ (current_point + vertex.reshape(2, 1))

# Plot the vertices of the triangle
plt.plot(v[0, :], v[1, :], 'k*', label='Vertices')
# Plot the starting point
plt.plot(x[0, 0], x[1, 0], 'b.', label='Starting Point')

# Plot all the points obtained during the iterations
plt.plot(points[0, :], points[1, :], 'r.', markersize=1, label='Points')

plt.axis('equal')
plt.legend()
plt.show()
```

4. The following code creates the iterated points and plots them:

In this code, we select an integer number k from 1 to 3 at random by using the command rand. We then do a linear transformation with the matrix T, the selected vertex v(:,k), and the point x(:,j) obtained on the previous step. Running this code with a large number of points may take some time. If you did everything correctly, you should obtain

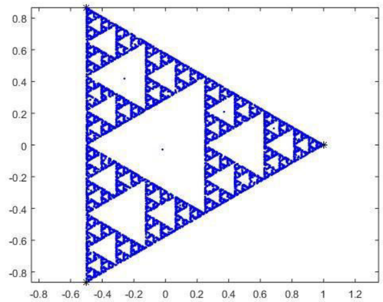


Figure 13: The Sierpinski triangle

an image similar to Fig. 13. The resulting picture approximates a fractal called the Sierpinski triangle.

There is a different procedure to generate this fractal starting with the interior of an equilateral triangle. At each step we will connect the mid-points of the sides of the triangle by straight lines. This subdivides the triangle into four triangles with the sides twice smaller than the original. Now we will throw out the middle triangle. Then we repeat this process with each of the leftover triangles, and so on. Doing this infinitely many times will produce the Sierpinski triangle as well. Variables: x

```
import numpy as np
import matplotlib.pyplot as plt

# Define the vertices of the triangle
t = np.linspace(0, 2 * np.pi, 4)[:-1]  # Generate t and remove the last element
v = np.array([np.cos(t), np.sin(t)])

# Define the linear transformation matrix
T = np.array([[0.5, 0], [0, 0.5]])

# Define the starting point
x = np.random.rand(2, 1) - 0.5

# Number of iterations
Num = 10000
```

```
# Create an array to store all points
points = np.zeros((2, Num + 1))

# Initial point
points[:, 0] = x.flatten()

# Iterative process
for j in range(Num):
    k = np.random.randint(0, 3)  # Random integer from 0 to 2
    # Perform the transformation
    current_point = points[:, j]  # Get the current point
    transformed_point = T @ (current_point - v[:, k]) + v[:, k]
    points[:, j + 1] = transformed_point

# Plot the points
plt.plot(points[0, :], points[1, :], 'b.', markersize=1)
plt.axis('equal')
plt.title('Sierpiński Triangle')
plt.show()
```

5. Let us try some variations of the previous procedure. First of all, let us change the ratio in which the segment is divided. Locate the code cell titled # Sierpinski triangle with a change of ratio and run the previous code with T=[1/3 0; 0 1/3]. Don't forget to clear the variables first (use the command clear)!

```
# Define the vertices of the triangle
t = np.linspace(0, 2 * np.pi, 4)[:-1]  # Generate t and remove the last element
v = np.array([np.cos(t), np.sin(t)])

# Define the new linear transformation matrix with a ratio of 1/3
T = np.array([[1/3, 0], [0, 1/3]])

# Define the starting point
x = np.random.rand(2, 1) - 0.5

# Number of iterations
Num = 10000

# Create an array to store all points
points = np.zeros((2, Num + 1))
```

```
# Initial point
points[:, 0] = x.flatten()

# Iterative process
for j in range(Num):
    k = np.random.randint(0, 3)  # Random integer from 0 to 2
    # Perform the transformation
    current_point = points[:, j]  # Get the current point
    transformed_point = T @ (current_point - v[:, k]) + v[:, k]
    points[:, j + 1] = transformed_point

# Plot the points
plt.plot(points[0, :], points[1, :], 'b.', markersize=1)
plt.axis('equal')
plt.title('Sierpiński Triangle with Ratio 1/3')
plt.show()
```

6. We can also add rotation to each of the iterations. Locate the code cell titled # Sierpinski triangle with rotation. Use the original algorithm for Sierpinsky triangle but change the matrix T to the following:

   and run the procedure again. Observe that the matrix T is equivalent to two transformations: the shrinking of the plane by 1/2 and the counterclockwise rotation of the plane around the origin with the angle of rotation theta.

```
# Define the vertices of the triangle
t = np.linspace(0, 2 * np.pi, 4)[:-1]  # Generate t and remove the last element
v = np.array([np.cos(t), np.sin(t)])

# Define the rotation angle and transformation matrix T
theta = np.pi / 18
T = 0.5 * np.array([[np.cos(theta), -np.sin(theta)],
                    [np.sin(theta),  np.cos(theta)]])

# Define the starting point
x = np.random.rand(2, 1) - 0.5

# Number of iterations
Num = 10000

# Create an array to store all points
```

```
points = np.zeros((2, Num + 1))

# Initial point
points[:, 0] = x.flatten()

# Iterative process
for j in range(Num):
  k = np.random.randint(0, 3)  # Random integer from 0 to 2
  # Perform the transformation
  current_point = points[:, j]  # Get the current point
  transformed_point = T @ (current_point - v[:, k]) + v[:, k]
  points[:, j + 1] = transformed_point

# Plot the points
plt.plot(points[0, :], points[1, :], 'b.', markersize=1)
plt.axis('equal')
plt.title('Sierpiński Triangle with Rotation')
plt.show()
```

7. Next, let us look at the fractal generated starting from four vertices of a square. Locate the code cell titled # Square and run the following code:

   Now repeat the iterative procedure allowing the vertex at each step to be randomly selected from the four vertices of the square. Generate the points and plot them on a new figure. Observe that no fractal has been created in this case.

```
# Define the vertices of the square
t = np.linspace(0, 2 * np.pi, 5)[:-1]  # Generate t and remove the last element
v = np.array([np.cos(t), np.sin(t)])

# Define the linear transformation matrix
T = np.array([[0.5, 0], [0, 0.5]])

# Define the starting point
x = np.random.rand(2, 1) - 0.5

# Number of iterations
Num = 10000

# Create an array to store all points
points = np.zeros((2, Num + 1))

# Initial point
```

```
points[:, 0] = x.flatten()

# Iterative process
for j in range(Num):
   k = np.random.randint(0, 4)  # Random integer from 0 to 3 (for 4 vertices)
   # Perform the transformation
   current_point = points[:, j]  # Get the current point
   transformed_point = T @ (current_point - v[:, k]) + v[:, k]
   points[:, j + 1] = transformed_point

# Plot the points
plt.figure()
plt.plot(points[0, :], points[1, :], 'b.', markersize=1)
plt.axis('equal')
plt.title('Fractal with Square Vertices')
plt.show()
```

8. Locate the code cell titled # Square with different ratio. Change the ratio in which the segment is divided to 1/3 and run the code again (don't forget to clear the variables!) with the modified matrix T:

   Q2: Does the code generate a fractal this time?

```
# Define the vertices of the square
t = np.linspace(0, 2 * np.pi, 5)[:-1]  # Generate t and remove the last element
v = np.array([np.cos(t), np.sin(t)])

# Define the new linear transformation matrix with a ratio of 1/3
T = np.array([[1/3, 0], [0, 1/3]])

# Define the starting point
x = np.random.rand(2, 1) - 0.5

# Number of iterations
Num = 10000

# Create an array to store all points
points = np.zeros((2, Num + 1))

# Initial point
points[:, 0] = x.flatten()
```

```
# Iterative process
for j in range(Num):
  k = np.random.randint(0, 4)  # Random integer from 0 to 3 (for 4 vertices)
  # Perform the transformation
  current_point = points[:, j]  # Get the current point
  transformed_point = T @ (current_point - v[:, k]) + v[:, k]
  points[:, j + 1] = transformed_point

# Plot the points
plt.figure()
plt.plot(points[0, :], points[1, :], 'b.', markersize=1)
plt.axis('equal')
plt.title('Fractal with Square Vertices and Ratio 1/3')
plt.show()
```

9. Let us try to do more sophisticated things. Locate the code cell titled # Square with vertex preference (the new vertex cannot be the same as the previous) which contains the following code:

   This code repeats the process for the square except that it does not allow the same vertex to be selected twice in a row.

```
# Define the vertices of the square using complex exponentials
t = np.linspace(0, 2 * np.pi, 5)[:-1]  # Generate t and remove the last element
v = np.exp(1j * t)  # Complex exponential for the vertices

# Define the linear transformation matrix
T = np.array([[0.5, 0], [0, 0.5]])

# Define the starting point
x = np.random.rand(2, 1) - 0.5

# Number of iterations
Num = 5000

# Create an array to store all points
points = np.zeros((2, Num + 1))

# Initial point
points[:, 0] = x.flatten()

# Variable to keep track of the previous vertex index
k1 = -1
```

```
# Iterative process
for j in range(Num):
  k = np.random.randint(0, 4)  # Random integer from 0 to 3
  # Ensure the same vertex is not selected twice in a row
  if k >= k1:
    k = (k + 1) % 4  # Move to the next vertex if k >= k1

  # Get the current vertex
  w = np.array([np.real(v[k]), np.imag(v[k])])

  # Perform the transformation
  current_point = points[:, j]  # Get the current point
  transformed_point = T @ (current_point - w) + w
  points[:, j + 1] = transformed_point

  # Update the previous vertex index
  k1 = k

# Plot the vertices and the points
plt.figure()
plt.plot(np.real(v), np.imag(v), 'k*', label='Vertices')  # Plot vertices
plt.plot(points[0, :], points[1, :], 'b.', markersize=1, label='Fractal Points')  # Plot points
plt.axis('equal')
plt.title('Fractal with Vertex Preference')
plt.legend()
plt.show()
```

10. Try the next code cell titled # Square with vertex preference (the new vertex cannot be opposite of the previous)

```
# Square with vertex preference (new vertex cannot be opposite of the previous):
clear;
T=[1/2 0; 0 1/2];
t=linspace(0,2*pi,5);
t(5)=[]; v=exp(1i*t);
x(:,1)=[rand-0.5; rand-0.5];
plot(real(v),imag(v),’k*’,x(1,1),x(2,1),’b.’); axis equal; hold
on; Num=5000;
w=[real(v(1));imag(v(1))]; k1=0;
for j=1:Num k=randi(4) if
     (k~=k1+2)&&(k1~=k+2)
     w=[real(v(k));imag(v(k))];
     x(:,j+1)=T*(x(:,j)-w)+w; k1=k;
     else x(:,j+1)=x(:,j);
     end;
```

```
    end; plot(x(1,:),x(2,:),’b.’); hold off;
```

This code repeats the process above except that it does not allow for the new vertex to lie opposite the old one.

```
# Define the vertices of the square using complex exponentials
t = np.linspace(0, 2 * np.pi, 5)[:-1]  # Generate t and remove the last element
v = np.exp(1j * t)  # Complex exponential for the vertices

# Define the linear transformation matrix
T = np.array([[0.5, 0], [0, 0.5]])

# Define the starting point
x = np.random.rand(2, 1) - 0.5

# Number of iterations
Num = 5000

# Create an array to store all points
points = np.zeros((2, Num + 1))

# Initial point
points[:, 0] = x.flatten()

# Variable to keep track of the previous vertex index
k1 = 0
w = np.array([np.real(v[k1]), np.imag(v[k1])])

# Iterative process
for j in range(Num):
    k = np.random.randint(0, 4)  # Random integer from 0 to 3

    # Ensure the new vertex is not opposite to the previous one
    if (k != (k1 + 2) % 4) and ((k1 != (k + 2) % 4)):
        w = np.array([np.real(v[k]), np.imag(v[k])])
        transformed_point = T @ (points[:, j] - w) + w
        points[:, j + 1] = transformed_point
        k1 = k  # Update the previous vertex index
    else:
        points[:, j + 1] = points[:, j]  # If opposite, repeat the current point

# Plot the vertices and the points
```

```
plt.figure()
plt.plot(np.real(v), np.imag(v), 'k*', label='Vertices')  # Plot vertices
plt.plot(points[0, :], points[1, :], 'b.', markersize=1, label='Fractal Points')  # Plot points
plt.axis('equal')
plt.title('Fractal with Vertex Preference (No Opposite Vertex)')
plt.legend()
plt.show()
```

11. Run the next code cell titled # Barnsley fern. The code in this cell creates the so-called Barnsley fern which is a fractal with striking resemblance to a fern leaf (a lot of plants and animals in nature use fractals to their advantage!):

```
# Barnsley fern clear;

T1=[0.85,0.04;-0.04,0.85];
T2=[-0.15,0.28;0.26,0.24];
T3=[0.2,-0.26;0.23,0.22];
T4=[0,0;0,0.16];
Q1=[0;1.64];
Q2=[-0.028;1.05];
Q3=[0;1.6];
Q4=[0,0];
P1=0.85;
P2=0.07;
P3=0.07;
P4=0.01;

Num=15000; x(:,1)=rand(2,1);
plot(x(1,1),x(2,:),’b.’) axis equal; hold
on; for j=1:Num r=rand; if r<=P1
x(:,j+1)=T1*x(:,j)+Q1;
     elseif r<=P1+P2 x(:,j+1)=T2*x(:,j)+Q2;
     elseif r<=P1+P2+P3 x(:,j+1)=T3*x(:,j)+Q3;
     else x(:,j+1)=T4*x(:,j)+Q4;
     end
end plot(x(1,:),x(2,:),’b.’);
```

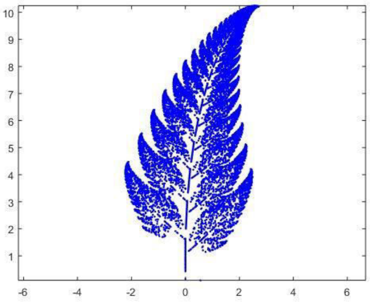


Figure 14: Barnsley Fern

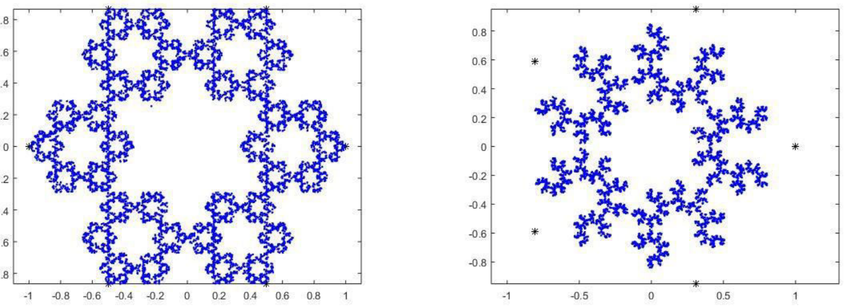


Figure 15: Fractals

```
hold off;
```

This code alternates between four different linear mappings given by $y = T_j * x + Q_j$, $j = 1,2,3,4$, where the matrices $T_j$ and the vectors $Q_j$ are given in the code. The probabilities of each mapping being selected are not equal as well and are given by the numbers P1, P2, P3, and P4 in the code. If you did everything correctly, you should see the image similar to the one in Fig. 14.

```
# Define the transformation matrices and translation vectors
T1 = np.array([[0.85, 0.04], [-0.04, 0.85]])
T2 = np.array([[-0.15, 0.28], [0.26, 0.24]])
T3 = np.array([[0.2, -0.26], [0.23, 0.22]])
T4 = np.array([[0, 0], [0, 0.16]])

Q1 = np.array([0, 1.64])
Q2 = np.array([-0.028, 1.05])
Q3 = np.array([0, 1.6])
Q4 = np.array([0, 0])

P1 = 0.85
P2 = 0.07
P3 = 0.07
P4 = 0.01

Num = 15000

# Initialize the starting point
x = np.zeros((2, Num + 1))
x[:, 0] = np.random.rand(2)  # Starting point

# Plot the initial point
```

```
plt.figure()
plt.plot(x[0, 0], x[1, 0], 'b.')

# Iterative process to generate the fern
for j in range(Num):
  r = np.random.rand()
  if r <= P1:
    x[:, j + 1] = T1 @ x[:, j] + Q1
  elif r <= P1 + P2:
    x[:, j + 1] = T2 @ x[:, j] + Q2
  elif r <= P1 + P2 + P3:
    x[:, j + 1] = T3 @ x[:, j] + Q3
  else:
    x[:, j + 1] = T4 @ x[:, j] + Q4

# Plot the fractal
plt.plot(x[0, :], x[1, :], 'b.', markersize=1)
plt.axis('equal')
plt.title('Barnsley Fern')
plt.show()
```

12. Modify the initial code for the Sierpinski triangle to create a fractal based on a regular hexagon. Take the matrix T to be

    If you did everything correctly, you should see the figure similar to the first image in Fig. 15.

    Q3: Experiment with the ratio. What is the largest ratio you can find such that the little hexagons don't touch (two digits after the decimal point are sufficient)?

```
import numpy as np
import matplotlib.pyplot as plt

# Define the vertices of a regular hexagon
t = np.linspace(0, 2 * np.pi, 7)[:-1]  # Generate 6 vertices and remove the last point
v = np.array([np.cos(t), np.sin(t)])   # Calculate vertices

# Define the linear transformation matrix
T = np.array([[0.25, 0], [0, 0.25]])

# Define the number of iterations
Num = 15000

# Initialize the starting point
```

```
x = np.zeros((2, Num + 1))
x[:, 0] = np.random.rand(2) - 0.5  # Random starting point

# Create an array to store all points
points = np.zeros((2, Num + 1))
points[:, 0] = x[:, 0]

# Iterative process to generate the fractal
for j in range(Num):
   k = np.random.randint(0, 6)  # Randomly choose one of the 6 vertices
   points[:, j + 1] = T @ (points[:, j] - v[:, k]) + v[:, k]

# Plot the vertices and the points
plt.figure()
plt.plot(v[0, :], v[1, :], 'k*', label='Vertices')  # Plot vertices of hexagon
plt.plot(points[0, :], points[1, :], 'b.', markersize=1, label='Fractal Points')  # Plot fractal points
plt.axis('equal')
plt.title('Fractal Based on Regular Hexagon')
plt.legend()
plt.show()
```

13. Finally, create another Chaos Game. Use the matrix T=[2/5 0; 0 2/5] for the linear mapping and the vertices of a regular pentagon with the restriction that each new vertex cannot be the same as the previous one. Plot the result in a new figure window. If you did everything correctly, you should see the figure similar to the second image in Fig. 15.

```
# Define the vertices of a regular pentagon
t = np.linspace(0, 2 * np.pi, 6)[:-1]  # Generate 5 vertices
v = np.array([np.cos(t), np.sin(t)])    # Calculate vertices

# Define the linear transformation matrix
T = np.array([[2/5, 0], [0, 2/5]])

# Define the number of iterations
Num = 10000

# Initialize the starting point
x = np.zeros((2, Num + 1))
x[:, 0] = np.random.rand(2) - 0.5  # Random starting point
```

```
# Create an array to store all points
points = np.zeros((2, Num + 1))
points[:, 0] = x[:, 0]

# Initialize previous vertex index
prev_vertex_index = np.random.randint(0, 5)

# Iterative process to generate the fractal
for j in range(Num):
   # Randomly choose one of the 5 vertices but not the same as the previous one
   current_vertex_index = prev_vertex_index
   while current_vertex_index == prev_vertex_index:
      current_vertex_index = np.random.randint(0, 5)

   # Apply the transformation
   w = v[:, current_vertex_index]
   points[:, j + 1] = T @ (points[:, j] - w) + w

   # Update the previous vertex index
   prev_vertex_index = current_vertex_index

# Plot the vertices and the points
plt.figure()
plt.plot(v[0, :], v[1, :], 'k*', label='Vertices')  # Plot vertices of pentagon
plt.plot(points[0, :], points[1, :], 'b.', markersize=1, label='Fractal Points')  # Plot fractal points
plt.axis('equal')
plt.title('Fractal Based on Regular Pentagon with Vertex Restriction')
plt.legend()
plt.show()
```

## Project 11: Projections, eigenvectors, Principal Component Analysis, and face recognition algorithms

**Goals:** To use eigenvalues and eigenvectors of a matrix to create a simple face recognition algorithm.

**To get started:**

- Download the file lab11.py and put it in your working directory[13].
- Download the archive file database.zip and unpack it into your working directory. This will produce a folder "database", which contains 33 image files in the format .pgm [14].

**Python commands used:** imread, shape, reshape, str, mean, np.uint8, np.double, np.ones, np.linalg.eig, np.diag

**What you have to submit:** The file lab11.py which you will modify during the lab session.

### INTRODUCTION

In this project we will apply principal component analysis (PCA) to create a simple face recognition algorithm. The PCA algorithm works by identifying patterns in the data which allows one to reduce the dimension of the dataset. We will use the following steps to apply the PCA algorithm to face recognition:

[13] The printout of the file lab11.m can be found in appendices of this book.

[14] These files represent a subset of the database of faces from AT&T Laboratories Cambridge www.cl.cam.ac.uk/research/dtg/attarchive/facedatabase.html.

- Download the database of $N$ images (in our case, $N$ = 30). All images must have the same dimensions $m \times n$.

- Input the images person1.pgm-person30.pgm into Python as arrays. Reshape each of the arrays as a column vector $\mathbf{X}_i$, $i = 1,...,N$.

- Compute the mean of all images $\mathbf{X}_{mean} = \frac{1}{N}\sum_{i=1}^{N}\mathbf{X}_i$.

- Subtract the mean image from each of the vectors $\mathbf{X}_i$: $\tilde{\mathbf{X}}_i = \mathbf{X}_i - \mathbf{X}_{mean}$ and create a matrix $\mathbf{P}$, the $i$th column of which is equal to the vector $\tilde{\mathbf{X}}_i$. The matrix $\mathbf{P}$ has 30 columns and $mn$ rows where $m$, $n$ are the dimensions of the images.

- We are interested in the eigenvalues and eigenvectors of the matrix $\mathbf{PP}^T$. However, the matrix $\mathbf{PP}^T$ is usually very large. For instance, if our database consists of the images with dimensions $m \times n = 100 \times 100$, then the matrix $\mathbf{PP}^T$ has the dimensions 10000 × 10000, and computation of eigenvalues and eigenvectors becomes unfeasible. Instead, we will use the fact that only at most $N - 1$ eigenvalues of the matrix $\mathbf{PP}^T$ are non-zero (why?), and those non-zero eigenvalues are the same as non-zero eigenvalues of the matrix $\mathbf{P}^T\mathbf{P}$ (why?). The matrix $\mathbf{P}^T\mathbf{P}$ has the dimensions $N \times N$ which are usually much smaller than the dimensions of the matrix $\mathbf{PP}^T$. For instance, if there are 30 images in the database (like in our case), then the dimensions of $\mathbf{P}^T\mathbf{P}$ are only 30 × 30. Thus, it is much more efficient to solve the eigenvalue problem for the matrix $\mathbf{P}^T\mathbf{P}$.
- If $\mathbf{u}_i$ is an eigenvector of the matrix $\mathbf{P}^T\mathbf{P}$ corresponding to a non-zero eigenvalue $\lambda_i$, then $\mathbf{v}_i = \mathbf{P}\mathbf{u}_i$ is the eigenvector of the matrix $\mathbf{PP}^T$ corresponding to $\lambda_i$. The eigenvectors of the matrix $\mathbf{PP}^T$ constitute the set of components of the data with the largest variance between them (which are called principal components).

- Compute the projections of your sample data onto the linear subspace generated by the vectors $\mathbf{v}_i$. The power of the algorithm is in the dimension reduction. Our initial images of faces “live” in the $mn$-dimensional real space. For instance, if the images are 100×100 pixels, then the dimension of the space is equal to $100^2 = 10000$. If we would try to compare these images to each other pixel by pixel, we would need to compare 10000 pixels. However, not all of these pixels are significant. Principal Component Analysis allows us to identify the important information about the image and compare only that information. For instance, if we are working with a database of 30 images, then we will need to compare only 29 numbers. That is a dimensional reduction from 10000 to only 29! The PCA algorithm is a work horse behind many modern techniques in data mining, including face recognition algorithms.

## TASKS

1. Open the file lab11.m and run the first code cell titled # Input database files into the Python:

This code will input the database images into Python, reshape each image from the rectangular array to a column vector, and create the matrix P. Observe the use of the string function num2str in the imread command and the use of the reshape function to change the dimensions of the arrays containing the image data. Instead of the rectangular arrays with the elements corresponding to the pixel colors we now have a set of long vectors where columns of each image have been "stacked" on top of each other. Variables: Database Size, m, n, P

```
import numpy as np
from PIL import Image
import os
import numpy as np
from PIL import Image
import os
import matplotlib.pyplot as plt

# Parameters
Database_Size = 30
database_path = 'C:/Users/Harshith/Downloads/laproject/materials/database'

# Initialize list to store image vectors
P = []

# Reading images from the database
for j in range(1, Database_Size + 1):
    image_path = os.path.join(database_path, f'person{j}.pgm')
    image = Image.open(image_path)
    image_array = np.array(image)

    # Get dimensions of the image
    m, n = image_array.shape

    # Reshape the image array to a column vector
    image_vector = image_array.reshape(m * n, 1)
    P.append(image_vector)

# Convert list to numpy array (matrix)
P = np.hstack(P)

# Print out the variables for verification
print(f"Database Size: {Database_Size}")
print(f"Image dimensions (m, n): ({m}, {n})")
print(f"P matrix shape: {P.shape}")
```

2. Compute the mean image by applying the mean function to the the matrix P row-wise. Display the mean image by using the function imshow.

Variables: mean face

```
# Compute the mean face
mean_face = np.mean(P, axis=1)

# Reshape the mean face back to the original image dimensions
mean_face_image = mean_face.reshape(m, n)

# Display the mean face image
plt.imshow(mean_face_image, cmap='gray')
plt.title('Mean Face')
plt.axis('off')  # Hide axis
plt.show()
```

3. Subtract the mean of the images from each of the columns of the matrix P: Notice the conversion to the double type of the array P.

```
# Compute the mean face
mean_face = np.mean(P, axis=1)

# Convert P to double (float64 in numpy)
P = P.astype(np.float64)

# Subtract the mean face from each column of P
mean_face_column = mean_face.reshape(-1, 1)
P = P - mean_face_column @ np.ones((1, Database_Size))

# Print the first column of P to verify subtraction
print(P[:, 0])
```

4. Compute the covariance matrix $\mathbf{P}^T\mathbf{P}$ and compute the eigenvalues and the eigenvectors of this matrix $\mathbf{P}^T\mathbf{P}$ by using the built in function eig. The function eig returns two arrays, one of which has eigenvalues on the main diagonal, and the other has eigenvectors as columns. Observe that one of the eigenvalues is very close to zero.

Variables: Vectors, Values, EigenVectors

Q1: Why does the matrix $\mathbf{P}^T\mathbf{P}$ have a zero eigenvalue?

```
# Compute the covariance matrix P^T * P
PTP = P.T @ P

# Compute the eigenvalues and eigenvectors of P^T * P
Values, Vectors = np.linalg.eig(PTP)

# Compute the actual eigenvectors of the covariance matrix
EigenVectors = P @ Vectors

# Normalize the eigenvectors
EigenVectors = EigenVectors / np.linalg.norm(EigenVectors, axis=0)

# Display the first few eigenvalues for verification
print("Eigenvalues:", Values)
```

5. Display the set of eigenfaces on the same figure by running the code cell # Display the set of eigenfaces:

   If you did everything correctly, you will see an image containing 29 eigenfaces, the first five of which are shown in Fig. 16. Observe that the mean face has been added back to the eigenfaces.

   Variables: EigenFaces

```
# Display the set of eigenfaces
eigenfaces = []

for j in range(1, Database_Size):
    eigenface = EigenVectors[:, j] + mean_face
    eigenface_image = eigenface.reshape(m, n)
    eigenfaces.append(eigenface_image)

# Concatenate the eigenfaces horizontally
EigenFaces = np.hstack(eigenfaces)

# Display the eigenfaces
plt.figure(figsize=(15, 5))
plt.imshow(EigenFaces, cmap='gray')
plt.title('Eigenfaces')
plt.axis('off')  # Hide axis
plt.show()
```

6. The matrix $\mathbf{PP}^T$ is symmetric. It follows from the theorems of linear algebra that the eigenvectors corresponding to the distinct eigenvalues are orthogonal. Write code to verify the last statement. You will need to create the matrix
   ```
   Products=EigenVectors’*EigenVectors
   ```
   and observe that it is a diagonal matrix. This matrix contains all possible dot products of the eigenvectors of the matrix $\mathbf{PP}^T$. The fact that the eigenvectors are orthogonal will be used in the following sections to find orthogonal projections.

   Variables: Products

   Q2: Why is the matrix $\mathbf{PP}^T$ symmetric?

   ```
   # Compute the Products matrix
   Products = EigenVectors.T @ EigenVectors

   # Print the Products matrix to verify orthogonality
   print("Products matrix:")
   print(Products)

   # Check if Products matrix is diagonal
   is_diagonal = np.allclose(Products, np.diag(np.diagonal(Products)))
   print(f"Is Products matrix diagonal? {is_diagonal}")
   ```

7. In this project we will use the obtained eigenfaces to perform several operations. Our first task will be to “unmask” an altered face (sunglasses have been digitally added to the image). The main idea behind this code is in computing the projection of the vector of data (obtained from the altered image) onto the subspace spanned by the eigenvectors. This projection is then displayed on the screen. Now run the code cell titled # Recognition of an altered image (sunglasses):

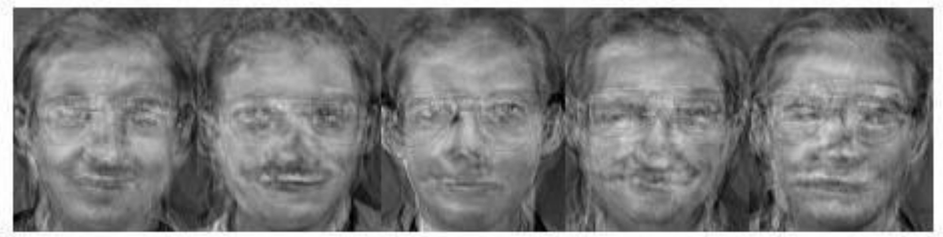


Figure 16: The first five eigenfaces

   In this code, we start by uploading the image person30altered1.pgm into Python. The image is then reshaped as a long vector similarly to the way it was done before. Then we compute the projection of the new vector with the subtracted vector mean face onto the space spanned by the set of the eigenvectors and add the vector mean _face back. Finally, the reconstructed image is displayed. Observe that we get a very good approximation of the initial image without sunglasses.

Variables: image _read, U, NormsEigenVectors, W, U approx

```
# Define image dimensions
m, n = 112, 92

# Read the altered image
altered_image_path                                                            =
'C:/Users/Harshith/Downloads/laproject/materials/database/person32.pgm'
image_read = Image.open(altered_image_path)
image_array = np.array(image_read)
U = image_array.reshape(m * n, 1)

# Compute the norms of the eigenvectors
Products = EigenVectors.T @ EigenVectors
NormsEigenVectors = np.diag(Products)

# Compute the projection coefficients
W = EigenVectors.T @ (U.astype(np.float64) - mean_face.reshape(-1, 1))
W = W / NormsEigenVectors.reshape(-1, 1)  # Ensure proper division

# Reconstruct the image from the projection
U_approx = EigenVectors @ W + mean_face.reshape(-1, 1)

# Print shapes for debugging
print("Shape of U_approx:", U_approx.shape)
print("Expected shape:", (m * n, 1))

# Ensure the shape matches for reshaping
if U_approx.shape[0] == m * n and U_approx.shape[1] == 1:
   image_approx = U_approx.reshape(m, n).astype(np.uint8)
else:
   raise ValueError(f"Cannot reshape array of size {U_approx.size} into shape ({m}, {n})")

# Display the original altered image and the reconstructed image
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.imshow(image_array, cmap='gray')
plt.title('Original Altered Image')
plt.axis('off')  # Hide axis

plt.subplot(1, 2, 2)
plt.imshow(image_approx, cmap='gray')
plt.title('Reconstructed Image')
plt.axis('off')  # Hide axis

plt.show()
```

8. Modify the code above to unmask the image person30altered2.pgm.

```
# Define image dimensions
m, n = 112, 92

# Read the altered image
altered_image_path                                                                   =
'C:/Users/Harshith/Downloads/laproject/materials/database/person33.pgm'
image_read = Image.open(altered_image_path)
image_array = np.array(image_read)
U = image_array.reshape(m * n, 1)

# Compute the norms of the eigenvectors
Products = EigenVectors.T @ EigenVectors
NormsEigenVectors = np.diag(Products)

# Compute the projection coefficients
W = EigenVectors.T @ (U.astype(np.float64) - mean_face.reshape(-1, 1))
W = W / NormsEigenVectors.reshape(-1, 1)  # Ensure proper division

# Reconstruct the image from the projection
U_approx = EigenVectors @ W + mean_face.reshape(-1, 1)

# Print shapes for debugging
print("Shape of U_approx:", U_approx.shape)
print("Expected shape:", (m * n, 1))

# Ensure the shape matches for reshaping
if U_approx.shape[0] == m * n and U_approx.shape[1] == 1:
   image_approx = U_approx.reshape(m, n).astype(np.uint8)
else:
   raise ValueError(f"Cannot reshape array of size {U_approx.size} into shape ({m}, {n})")

# Display the original altered image and the reconstructed image
```

```
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.imshow(image_array, cmap='gray')
plt.title('Original Altered Image')
plt.axis('off')  # Hide axis

plt.subplot(1, 2, 2)
plt.imshow(image_approx, cmap='gray')
plt.title('Reconstructed Image')
plt.axis('off')  # Hide axis

plt.show()
```

9. Now let us compute the approximation of the face which is not in the database. To do this, modify the code above again to approximate the image person31.pgm. Observe, that the results of the last two approximations are somewhat worse than the very first one. This can be explained by the fact that there is an insufficient number of images in our database which do not span a wide enough variety of faces.

```
# # Recognition and approximation of a new face (person31.pgm)
import numpy as np
from PIL import Image
import matplotlib.pyplot as plt

# Define image dimensions
m, n = 112, 92

# Read the new image (person31.pgm)
new_image_path                                                  =
'C:/Users/Harshith/Downloads/laproject/materials/database/person31.pgm'
image_read = Image.open(new_image_path)
image_array = np.array(image_read)
U = image_array.reshape(m * n, 1)

# Compute the norms of the eigenvectors
Products = EigenVectors.T @ EigenVectors
NormsEigenVectors = np.diag(Products)

# Compute the projection coefficients
W = EigenVectors.T @ (U.astype(np.float64) - mean_face.reshape(-1, 1))
W = W / NormsEigenVectors.reshape(-1, 1)  # Ensure proper division

# Reconstruct the image from the projection
U_approx = EigenVectors @ W + mean_face.reshape(-1, 1)
```

```
# Print shapes for debugging
print("Shape of U_approx:", U_approx.shape)
print("Expected shape:", (m * n, 1))

# Ensure the shape matches for reshaping
if U_approx.shape[0] == m * n and U_approx.shape[1] == 1:
   image_approx = U_approx.reshape(m, n).astype(np.uint8)
else:
   raise ValueError(f"Cannot reshape array of size {U_approx.size} into shape ({m}, {n})")

# Display the original new image and the reconstructed image
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.imshow(image_array, cmap='gray')
plt.title('Original New Image')
plt.axis('off')  # Hide axis

plt.subplot(1, 2, 2)
plt.imshow(image_approx, cmap='gray')
plt.title('Reconstructed Image')
plt.axis('off')  # Hide axis

plt.show()

# Variables: image_read, U, NormsEigenVectors, W, U_approx
```

## Project 12: Matrix eigenvalues and the Google's PageRank algorithm

**Goals:** To apply matrix eigenvalues and eigenvectors to ranking of webpages in the World Wide Web.

**To get started:**

- Download the file lab12.py and put it in your working directory[15].

[15] The printout of the file lab12.m is given in the appendices of this book.

- Download the file AdjMatrix.mat which contains the adjacency matrix of a so-called “wikivote” network with 8297 nodes [16] [9].

**Python commands used:** np.load, shape, np.size, np.count_nonzero, for… in, plt.plot**What you have to submit:** The file lab12.py which you will modify during the lab session.

### INTRODUCTION

According to social polls, the majority of users only look at the first few results of the online search and very few users look past the first page of results. Hence, it is crucially important to rank the pages in the “right” order so that the most respectable and relevant results will come first. The simplest way to determine the rank of a webpage in a network is to look at how many times it has been referred to by other webpages. This simple ranking method leaves a lot to be desired. In particular, it can be easily manipulated by referring to a certain webpage from a lot of “junk” webpages. The quality of the webpages referring to the page we are trying to rank should matter too. This is the main idea behind the Google PageRank algorithm.

The Google PageRank algorithm is the oldest algorithm used by Google to rank the web pages which are preranked offline. The PageRank scores of the webpages are recomputed each time Google crawls the web. Let us look at the theory behind the algorithm. As it turns out it, it is based on the theorems of linear algebra!

The main assumption of the algorithm is that if you are located on any webpage then with equal probability you can follow any of the hyperlinks from that page to another page. This allows to represent a webpage network as a directed graph with the webpages being the nodes, and the edges being the hyperlinks between the webpages. The adjacency matrix of such a network is built in the following way: the $(i,j)$th element of this matrix is equal to 1 if there is a hyperlink from the webpage $i$ to the webpage $j$ and is equal to 0 otherwise. Then the row sums of this matrix will represent numbers of hyperlinks from each webpage and the column sums will represent numbers of times each webpage has been referred to by other webpages.

Even further, we can generate a matrix of probabilities **S** such that the $(i,j)$th element of this matrix is equal to the probability of traveling from $i$th webpage to $j$th webpage in the network. This probability is equal to zero if there is no hyperlink from $i$th page to $j$th page and is equal to $1/N_i$ if there is a hyperlink from $i$th page to $j$th page, where $N_i$ is the total number of hyperlinks from $i$th page. For instance, consider a sample network of only four webpages shown on the Fig. 17. The matrix **S** for this network can be written as:

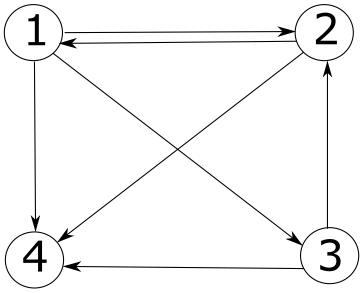


Figure 17: Sample network of four webpages

[16] The original network data is available here: https://snap.stanford.edu/data/.

$$\mathbf{S} = \begin{bmatrix} 0 & 1/3 & 1/3 & 1/3 \\ 1/2 & 0 & 0 & 1/2 \\ 0 & 1/2 & 0 & 1/2 \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{1}$$

There are several issues which make working with the matrix **S** inconvenient. First of all, there are webpages that do not have any hyperlinks - the so-called “dangling nodes” (such as the node 4 in Fig. 17). These nodes will correspond to zero rows of the matrix **S**. Moreover, the webpages in the network may not be connected to each other and the graph of the network may consist of several disconnected components. These possibilities lead to undesirable properties of the matrix **S** which make computations with it complicated and not even always possible.

The problem of the dangling nodes can be solved by assigning all elements of the matrix **S** in the rows corresponding to the dangling nodes equal probabilities $1/N$, where $N$ is the number of the nodes in the network. This can be understood in the following way: if we are at the dangling node we can with equal probability jump to any other page in the network. To solve the potential disconnectedness problem, we assume that a user can follow hyperlinks on any page with a probability $1 - \alpha$ and can jump (or “teleport”) to any other page in the network with a probability $\alpha$. The number $\alpha$ is called a damping factor. The value of $\alpha = 0.15$ is usually taken in practical applications. The “teleport” surfing of the network can be interpreted as a user manually typing the webpage address in the browser or using a saved hyperlink from their bookmarks to move from one page onto another. The introduction of the damping factor allows us to obtain the Google matrix **G** in the form:

$$\mathbf{G} = (1 - \alpha)\mathbf{S} + \alpha\mathbf{E},$$

where **E** is a matrix with all the elements equal to $1/N$, where $N$ is a number of webpages in the network.

The matrix **G** has nice properties. In particular, it has only positive entries and all of its rows sum up to 1. In mathematical language, this matrix is *stochastic* and *irreducible* (you can look up the precise definitions of these terms if you are interested). The matrix **G** satisfies the following **Perron-Frobenius theorem**:

**Theorem 1 (Perron-Frobenius)** *Every square matrix with positive entries has a unique unit eigenvector with all positive entries. The eigenvalue corresponding to this eigenvector is real and positive. Moreover, this eigenvalue is simple and is the largest in absolute value among all the eigenvalues of this matrix.*

Let us apply this theorem to the matrix **G**. First of all, observe that the row sums of the matrix **G** are equal to 1. Consider the vector $\mathbf{v}_1 = (1,1,...,1)^T/N$. It is easy to see that

$$\mathbf{G}\mathbf{v}_1 = \mathbf{v}_1.$$

But then it follows that $\mathbf{v}_1$ is the unique eigenvector with all positive components, and, therefore, by the Perron-Frobenius theorem, $\lambda_1 = 1$ is the largest eigenvalue!

We are interested in the left eigenvector for the eigenvalue $\lambda_1 = 1$:

$$\mathbf{u}_1^T\mathbf{G} = \mathbf{u}_1^T.$$

Again, by the Perron-Frobenius theorem, the vector $\mathbf{u}_1$ is the unique unit eigenvector with all positive components corresponding to the largest in absolute value eigenvalue $\lambda_1 = 1$. We will use the components of this vector for the ranking of webpages in the network.

Let us look at the justification behind this algorithm. We have already established that the vector $\mathbf{u}_1$ exists. Consider the following iterative process. Assume that at the beginning a user can be on any webpage in the network with equal probability:

$$\mathbf{w}_0 = (1/N,1/N,...,1/N).$$

After 1 step (one move from one webpage to another using hyperlinks or teleporting), the probability vector of being on the $i$th webpage is determined by the $i$th component of the vector

$$\mathbf{w}_1 = \mathbf{w}_0\mathbf{G}.$$

After two moves the vector of probabilities becomes

$$\mathbf{w}_2 = \mathbf{w}_1\mathbf{G} = \mathbf{w}_0\mathbf{G}^2,$$

and so on.

We hope that after a large number of steps $n$, the vector $\mathbf{w}_n = \mathbf{w}_0\mathbf{G}^n$ starts approaching some kind of limit vector $\mathbf{w}_*$, $\mathbf{w}_n \to \mathbf{w}_*$. It turns out that due to the properties of the matrix $\mathbf{G}$ this limit vector $\mathbf{w}_*$ indeed exists and it is exactly the eigenvector corresponding to the largest eigenvalue, namely, $\lambda_1 = 1$. Moreover, numerical computation of matrix eigenvalues is actually based on taking the powers of the matrix (it is called the Power method) and not on solving the characteristic equation!

Let us assume that the vector $\mathbf{w}_*$ is a non-negative vector whose entries sum to 1. Then the components of this vector represent the probabilities of being on each webpage in the network after a very large number of moves along the hyperlinks. Thus, it is perfectly reasonable to take these probabilities as ranking of the webpages in the network.

## TASKS

1. Open the file lab12.m. In the code cell titled #Load the network data load the data from the file AdjMatrix.mat into Python by using the load command. Save the resulting matrix as AdjMatrix. Observe that the adjacency matrices of real networks are likely to be very large (may contain millions of nodes or more) and sparse. Check the sparsity of the matrix AdjMatrix using the functions numel and nnz. Denote the ratio of non-zero elements as nnzAdjMatrix. If you did everything correctly you should obtain that only
   0.15% of the elements of the matrix AdjMatrix are non-zero.
   Variables: AdjMatrix, nnzAdjMatrix

```
import scipy.io
import numpy as np
import matplotlib.pyplot as plt
import networkx as nx
from scipy.sparse import csr_matrix
```

```
# Load the network data
# Load the adjacency matrix from the file 'AdjMatrix.mat'
data =
scipy.io.loadmat("C:/Users/Harshith/Downloads/laproject/materials/AdjMatrix.mat")
AdjMatrix =csr_matrix(data['AdjMatrix'])

# Check the sparsity of the matrix
num_elements = AdjMatrix.shape[0] * AdjMatrix.shape[1]
num_non_zero_elements = AdjMatrix.nnz
nnzAdjMatrix = num_non_zero_elements / num_elements

print(f"Sparsity of AdjMatrix: {nnzAdjMatrix:.4f}")
```

2. Check the dimensions of the matrix AdjMatrix using the size function. Save the dimensions as new variables m and n. Variables: m, n

```
# Check the dimensions of the matrix
m, n = AdjMatrix.shape
print(f"Dimensions of AdjMatrix: {m} x {n}")
```

3. Observe that while the network described by the matrix AdjMatrix is not large at all from the viewpoint of practical applications, computations with this matrix may still take a noticeable amount of time. To save time, we will cut a subset out of this network and use it to illustrate the Google PageRank algorithm. Introduce a new variable NumNetwork and set its value to 500. Then cut a submatrix AdjMatrixSmall out of the matrix AdjMatrix and plot the graph represented by the matrix AdjMatrixSmall by running the following code cell:

   This will plot the subgraph of the first 500 nodes in the network with random locations of the nodes. Notice the use of the function gplot to produce this graph. Observe that Python has special functions graph and digraph for working with graphs, but those functions are a part of the special package "Graph and Network Algorithms" which may not be immediately available. Simpler methods, as shown above, will be sufficient for our purposes.
   Variables: AdjMatrixSmall, coordinates, NumNetwork

```
import matplotlib.pyplot as plt
import networkx as nx

# Create a smaller submatrix and plot the network
NumNetwork = 500
AdjMatrixSmall = AdjMatrix[:NumNetwork, :NumNetwork].toarray() # Extract submatrix

# Generate random coordinates for the nodes
#np.random.seed(0)  # For reproducibility
```

```
coordinates = np.random.rand(NumNetwork, 2) * NumNetwork  # Random coordinates


# Plot the graph
plt.figure(figsize=(10, 10))
plt.plot(coordinates[:, 0], coordinates[:, 1], 'k-*')
plt.title('Subgraph of the First 500 Nodes')
plt.xlabel('Random X Coordinate')
plt.ylabel('Random Y Coordinate')
plt.show()

# Variables
print(f"AdjMatrixSmall shape: {AdjMatrixSmall.shape}")
print(f"Coordinates shape: {coordinates.shape}")
print(f"NumNetwork: {NumNetwork}")
```

4. Set the parameter $\alpha = 0.15$. Introduce the vector $w_0$=(1,1,…,1)/NumNetwork, and compute the consequent vectors $w_1 = w_0G$, $w_2 = w_1G$, $w_3 = w_2G$, $w_5 = w_0G^5$, $w_{10} = w_0G^{10}$. Compute the difference $\delta w = w_{10} - w_5$. Observe that the sequence $w_n$ converges to a certain limit vector $w_*$ very fast.

Variables: w0, w1, w2, w3, w5, w10, deltaw

```
# Compute the Google Matrix
alpha = 0.15
GoogleMatrix = np.zeros((NumNetwork, NumNetwork))

# Check the amount of links originating from each webpage
NumLinks = np.sum(AdjMatrixSmall, axis=1)

for i in range(NumNetwork):
   if NumLinks[i] != 0:
      GoogleMatrix[i, :] = AdjMatrixSmall[i, :] / NumLinks[i]
   else:
      GoogleMatrix[i, :] = 1.0 / NumNetwork

GoogleMatrix = (1 - alpha) * GoogleMatrix + alpha * np.ones((NumNetwork,
NumNetwork)) / NumNetwork

# Compute the vectors w0, w1, w2, w3, w5, w10
w0 = np.ones(NumNetwork) / np.sqrt(NumNetwork)
w1 = w0 @ GoogleMatrix
w2 = w1 @ GoogleMatrix
w3 = w2 @ GoogleMatrix
w10 = w0 @ (GoogleMatrix ** 10)
w5 = w0 @ (GoogleMatrix ** 5)
```

```
deltaw = w10 - w5
print("Difference δw:", np.linalg.norm(deltaw))
```

5. Compute the eigenvalues and the left and the right eigenvectors of the matrix $G$ using the function eig. Observe that the right eigenvector corresponding to the eigenvalue $\lambda_1 = 1$ is proportional to the vector $v_1 = (1,1,...,1)$. To compute the left eigenvectors, use the function eig on the matrix G'. Select the left eigenvector corresponding to the eigenvalue $\lambda_1 = 1$ and denote it as u1. Variables: u1

```
# Compute eigenvalues and eigenvectors
eigenvalues, right_eigenvectors = eig(GoogleMatrix)

# Find the index of the eigenvalue λ1 = 1
lambda_1_index = np.isclose(eigenvalues, 1)

# Get the right eigenvector corresponding to λ1
v1 = right_eigenvectors[:, lambda_1_index].flatten()

# Compute the left eigenvectors
left_eigenvalues, left_eigenvectors = eig(GoogleMatrix.T)

# Get the left eigenvector corresponding to λ1
u1 = left_eigenvectors[:, lambda_1_index].flatten()
print("Left Eigenvector (u1):", u1)
```

6. Observe that by default the vector u1 is not scaled to have all positive components (even though all the components of the vector $\mathbf{u}_1$ will have the same sign). Normalize this vector by using the code:

   This will create a vector with all positive components whose entries sum to 1 (called a probability vector).

```
# Normalize u1 to have all positive components
u1 = np.abs(u1) / np.linalg.norm(u1, 1)
```

7. Use the function max to select the maximal element and its index in the array. Variables: MaxRank, PageMaxRank

```
MaxRank, PageMaxRank = np.max(u1), np.argmax(u1)
print(f"MaxRank: {MaxRank}, PageMaxRank: {PageMaxRank}")
```

8. Find out whether the highest ranking webpage is the same as the page with the most hyperlinks pointed to it. To do so, create the vector of column sums of the matrix G and save it as MostLinks. Use the function max again to select the element with the maximal number of links.

Variables: MostLinks, MaxLinks, PageMaxLinks

```
MostLinks = np.sum(AdjMatrixSmall, axis=0)  # Sum of columns
MaxLinks, PageMaxLinks = np.max(MostLinks), np.argmax(MostLinks)
print(f"MostLinks: {MostLinks}, MaxLinks: {MaxLinks}, PageMaxLinks: {PageMaxLinks}")
```

9. Compare if MaxRank and MaxLinks are the same.

Q1: What is the number of hyperlinks pointing to the webpage MaxRank?

(look at the vector MostLinks to find out).

```
are_equal = PageMaxRank == PageMaxLinks
print(f"Is the highest ranking webpage the same as the page with the most hyperlinks? {are_equal}")

# Q1: What is the number of hyperlinks pointing to the webpage MaxRank?
print(f"Number of hyperlinks pointing to the webpage MaxRank: {MostLinks[PageMaxRank]}")
```

# Project 13: Social networks, clustering, and eigenvalue problems

**Goals:** To apply linear algebra techniques to data clustering.

**To get started:**

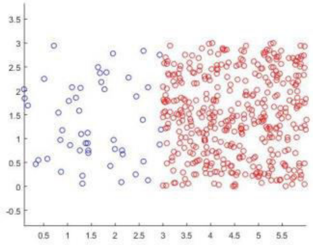
- Download the file lab13.py and put it in your working directory[17].

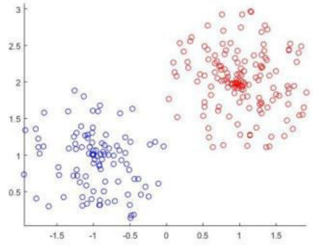
- Download the file social.mat which contains the adjacency matrix of a simulated social network[18].

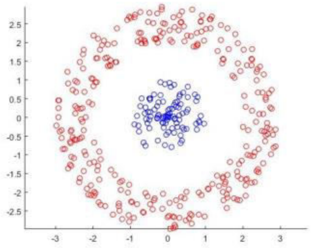
**Python commands used:** np.sum, np.diag, np.linalg.eig, for... in, if..., np.load, del, plt.spy, np.eye
**What you have to submit:** The file lab13.py which you will modify during the lab session.

### INTRODUCTION

Many practical applications deal with large amounts of data. One of the standard problems in data science is to group this data in clusters containing similar items. In many cases, clustering can be easily accomplished by the human brain (take a look at the attached pictures).







In general, creating an algorithm which allows us to separate the data into clusters is a difficult task. The data in many cases is multidimensional and cannot be visualized easily. Even deciding how many clusters to select is not necessary an easy task. There are currently several widely used algorithms which allow for the separation of the data points by similarity. All these algorithms have their own advantages and disadvantages.

In this project, we will use the spectral clustering algorithm based on the Fiedler's vector. We will look at the graph representing an individual's profile on a social network. The profile has a total of 351 "friends". Some of these friends know each other and are friends with each other as well. This friendship network can be represented as a graph with nodes being friends and edges showing the friendship connection between the individuals. The goal of clustering is to separate this graph into components which are tightly connected within themselves and have fewer connections to the outside (the so-called communities). The method explored here is based on the theorem by M. Fiedler, who has shown that for a connected graph (meaning that there is a path between any two vertices) the eigenvector corresponding to the second smallest eigenvalue of the Laplacian of the graph allows the graph to be split into two maximally intraconnected components.

[17] The printout of the file lab13.m can be found in the appendices of this book. [18]This data has been artificially created.

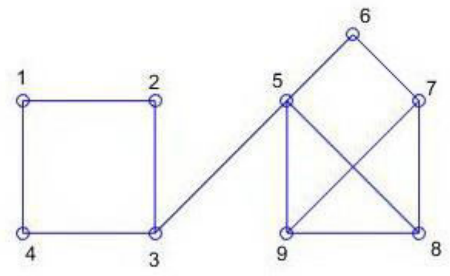


Figure 18: An example of a simple graph

**TASKS**

1. We will start by considering an example of a simple graph: Create a new code cell with the title # Small graph. Type in the adjacency matrix of this graph into Python and save it in a variable AdjMatrix. Variables: AdjMatrix

```
import numpy as np
import scipy.io
import matplotlib.pyplot as plt
from scipy.linalg import eigh

# 1. Simple graph: Define adjacency matrix
AdjMatrix = np.array([[0, 1, 1, 0],
              [1, 0, 0, 1],
              [1, 0, 0, 1],
              [0, 1, 1, 0]])
print("Adjacency Matrix:")
print(AdjMatrix)
```

2. Find the row sums of the matrix AdjMatrix (use the sum function) and save it as RowSums. Variables: RowSums

```
# 2. Find the row sums of the matrix AdjMatrix
RowSums = np.sum(AdjMatrix, axis=1)
print("\nRow Sums:")
print(RowSums)
```

3. Compute the Laplacian of the graph using the following command:

Observe that the diag(RowSums) command creates a diagonal matrix with the vector RowSums on the main diagonal and zeros everywhere else. Observe also that the matrix LaplaceGraph is a symmetric positive semi-definite singular matrix. Recall that a matrix **A** is positive semi-definite if $\mathbf{xAx}^T \geq 0$ for all column vectors **x**. To check that it is singular, multiply the matrix LaplaceGraph on the right by the column vector $[1, 1, 1, 1, 1, 1, 1, 1, 1]^T$.

Variables: LaplaceGraph

```
# 3. Compute the Laplacian of the graph
LaplaceGraph = np.diag(RowSums) - AdjMatrix
print("\nLaplacian Matrix:")
print(LaplaceGraph)

# Check if LaplaceGraph is singular
test_vector = np.ones(len(LaplaceGraph))
singularity_check = LaplaceGraph @ test_vector
print("\nSingularity Check (Laplacian * ones):")
print(singularity_check)
```

4. Find the eigenvalues and eigenvectors of the matrix LaplaceGraph using the eig function:

   Here the matrix V contains all the right eigenvectors of the matrix LaplaceGraph and the matrix D is a diagonal matrix with eigenvalues of the matrix LaplaceGraph on the main diagonal. Since the Laplacian is a symmetric graph, its eigenvalues are real. To simplify our task, let us sort the eigenvalues from smallest to largest. We will need to sort the eigenvectors along with them. Here is how to do that.

   Variables: V, D

```
# 4. Find eigenvalues and eigenvectors using the eig function
D, V = np.linalg.eig(LaplaceGraph)
```

5. View the elements of the matrices V and D using the “Workspace” window of the Python environment. Observe that one of the eigenvalues of the matrix LaplaceGraph is zero (this is due to the fact that the matrix LaplaceGraph is singular). The eigenvector for this eigenvalue is proportional to the vector $[1, 1, 1, 1, 1, 1, 1, 1, 1]^T$. All other eigenvalues of the matrix LaplaceGraph are positive. This follows from the fact that the matrix LaplaceGraph is positive semi-definite and the graph corresponding to this Laplacian is connected. Find the smallest positive eigenvalue of the matrix LaplaceGraph (since we sorted the eigenvalues, it is the second diagonal element of the matrix D).

```
d, ind = np.argsort(D), np.argsort(D)
D = np.diag(D[ind])
V = V[:, ind]

print("\nEigenvalues (sorted):")
print(np.diag(D))
print("\nEigenvectors (sorted):")
print(V)
```

6. Identify the eigenvector corresponding to the second smallest eigenvalue:

In the next step, we will use the entries of V2 to group the elements. To make sure we do this consistently, let us make sure that V2 is the unit eigenvector with positive first entry:

```
% make sure the first entry is positive if V2(1) < 0
V2 = -V2; end
```

Variables: V2

```
# 6. Identify the second smallest eigenvalue and its corresponding eigenvector
second_smallest_eigenvalue = D[1, 1]
V2 = V[:, 1]

# Ensure V2 has a positive first entry
if V2[0] < 0:
   V2 = -V2

print("\nSecond Smallest Eigenvalue:")
print(second_smallest_eigenvalue)
print("\nEigenvector corresponding to the second smallest eigenvalue (V2):")
print(V2)
```

7. Separate the elements of the eigenvector V2 onto positive and negative:

If everything is done correctly, the arrays pos and neg will contain correspondingly 1,2,3,4 and 5,6,7,8,9 which would be an intuitive clustering choice. Variables: pos, neg

```
# 7. Separate the elements of the eigenvector V2
pos = []
neg = []
for j in range(len(V2)):
   if V2[j] > 0:
      pos.append(j)
   else:
      neg.append(j)

print("\nPositive Indices (V2 > 0):")
print(pos)
print("\nNegative Indices (V2 <= 0):")
print(neg)

# Optional: Visualize the clusters
```

```
plt.figure(figsize=(8, 6))
plt.scatter(pos, [1]*len(pos), color='green', label='Positive')
plt.scatter(neg, [0]*len(neg), color='red', label='Negative')
plt.yticks([0, 1], ['Negative', 'Positive'])
plt.title('Clustering based on Eigenvector V2')
plt.xlabel('Indices')
plt.legend()
plt.grid()
plt.show()
```

8. Now let us apply this technique to a larger graph when the results cannot be visualized so easily. To do this, load the matrix Social which is an adjacency matrix of a network of 351 Facebook friends. The (*i,j*)th element is equal to 1 if the *i*th and *j*th friends are also friends with each other and is equal to 0 otherwise:

   The command spy(Social) will plot the sparsity pattern of the adjacency matrix Social on a figure, putting blue dots for non-zero elements of the matrix (you should see something similar to the first image in Fig. 19). Observe that it is difficult to distinguish any kind of underlying pattern in this matrix. Our goal will be to cluster this set of data into two maximally intraconnected groups. Variables: Social

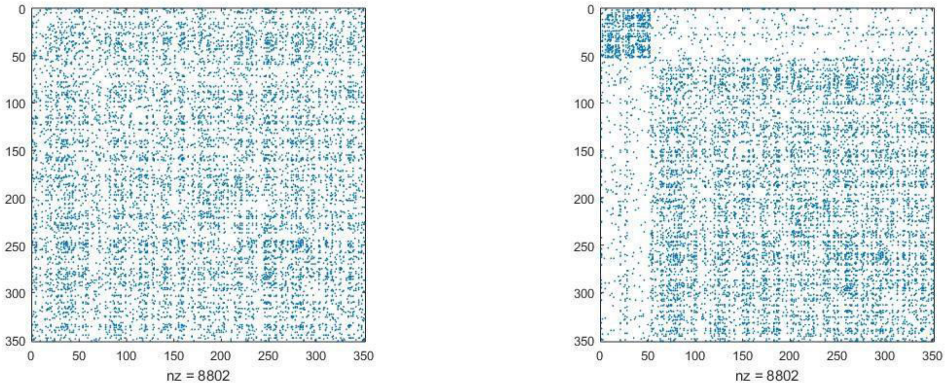


Figure 19: Adjacency matrices of the unsorted and sorted data.

```
from scipy.io import loadmat
# 8. Load the data
data                                         =
  loadmat("C:/Users/Harshith/Downloads/
  laproject/materials/social.mat")
Social = data['Social']
print("Loaded Social adjacency matrix with
  shape:", Social.shape)
```

```
# Spy plot of the Social matrix
plt.figure(figsize=(8, 6))
plt.spy(Social, markersize=1)
plt.title('Sparsity pattern of the Social
  adjacency matrix')
plt.show()
```

9. To do this, repeat the procedure we performed on a smaller matrix. Define the matrices DiagSocial and LaplaceSocial which are correspondingly the vector of row sums of the matrix Social and the Laplacian of this matrix. Variables: DiagSocial, LaplaceSocial

```
# 9. Define DiagSocial and LaplaceSocial
DiagSocial = np.sum(Social, axis=1)
LaplaceSocial = np.diag(DiagSocial) - Social
print("\nDiagonal matrix DiagSocial:")
print(DiagSocial)
print("\nLaplacian matrix LaplaceSocial:")
print(LaplaceSocial)
```

10. Compute the eigenvalues and the eigenvectors:

Variables: V,D

```
# 10. Compute eigenvalues and eigenvectors
D, V = np.linalg.eig(LaplaceSocial)

print("\nEigenvalues (D):")
print(D)
print("\nEigenvectors (V):")
print(V)

# Check the shapes
print("Shape of V (eigenvectors):", V.shape)
print("Shape of D (eigenvalues):", D.shape)
d, ind = np.argsort(D), np.argsort(D)
D = np.diag(D[ind])
V = V[:, ind]

print("\nEigenvalues (sorted):")
print(np.diag(D))
print("\nEigenvectors (sorted):")
print(V)
```

11. As before, identify the second smallest eigenvalue and the corresponding eigenvector V2 and sort the vertices of the graph by the positive and negative components of the vector V2. Make the necessary adjustment to ensure that V2(1)>0. Variables: V2

```
second_smallest_eigenvalue = D[1, 1]
V2 = V[:, 1]

# Ensure V2 has a positive first entry
if V2[0] < 0:
  V2 = -V2

print("\nSecond Smallest Eigenvalue:")
print(second_smallest_eigenvalue)
print("\nEigenvector corresponding to the second smallest eigenvalue (V2):")
print(V2)

pos = []
neg = []
for j in range(len(V2)):
  if V2[j] > 0:
    pos.append(j)
  else:
    neg.append(j)

print("\nPositive Indices (V2 > 0):")
print(pos)
print("\nNegative Indices (V2 <= 0):")
print(neg)
```

12. Permute the adjacency matrix Social using the permutation generated by the lists pos, neg:

    SocialOrdered=P*Social*P’

    Variables: order, P, SocialOrdered

    Q1: What does the command P*Social*P’ do to the rows and columns of the matrix Social?

```
# Create the order based on positive and negative indices
order = pos + neg  # Combine the positive and negative indices

m, n = Social.shape  # Get the shape of the Social matrix
iden = np.eye(m)  # Identity matrix of size m

# Create the permutation matrix P
```

```
P = np.zeros((m, m))
for j in range(m):
   for k in range(m):
      P[j, k] = iden[order[j], k]

# Permute the adjacency matrix
SocialOrdered = P @ Social @ P.T  # Using matrix multiplication

print("Shape of SocialOrdered:", SocialOrdered.shape)
```

13. Plot the matrix SocialOrdered using the function spy again. If you did everything correctly, you will see the matrix separating into two denser connected subsets similarly to what is pictured in the second image in Fig. 19.

```
import matplotlib.pyplot as plt

# Plot the permuted adjacency matrix
plt.figure(figsize=(8, 6))
plt.spy(SocialOrdered, markersize=1)  # Using a smaller marker size for better visibility
plt.title("Spy Plot of Permuted Adjacency Matrix (SocialOrdered)")
plt.xlabel("Nodes")
plt.ylabel("Nodes")
plt.grid(False)  # Disable the grid
plt.show()
```

14. What if we want more clusters? There are two ways to proceed. First, we can explore the eigenvector V3 corresponding to the third smallest eigenvalue. We will group the nodes in the four clusters: the ones which have positive components in both eigenvectors V2, V3 (++ group), the ones which have a positive component in V2 and a negative component in V3 (+−), the ones which have a negative component in V2 and a positive component in V3 (−+), and, finally, the ones with negative components in both vectors (−−). This generates 4 distinct groups and can be accomplished by the code in the following code cell:

```
# Cluster in 4 groups V3=V(:,3);
if V3(1) < 0
     V3 = -V3;
end pp=[]; pn=[]; np=[];
nn=[]; for j=1:m if (V2(j)>0)
if (V3(j)>0) pp=[pp,j]; else
pn=[pn,j];
          end;
     else if (V3(j)>0)
          np=[np,j]; else
          nn=[nn,j];
          end;
```

```
        end; end;
    order=[pp,pn,np,nn];
    iden=eye(m,m); for
    j=1:351; for k=1:351;
            P(j,k)=iden(order(j),k); end;
    end; SocialOrdered=P*Social*P’
    figure; spy(SocialOrdered)
```

```
import numpy as n
# Explore the third smallest eigenvalue for clustering
V3 = V[:, 2]  # Get the third eigenvector
if V3[0] < 0:  # Ensure V3 has a positive first entry
   V3 = -V3

# Initialize lists for the groups
pp = []  # ++ group
pn = []  # +- group
np = []  # -+ group
nn = []  # -- group

# Grouping based on the signs of V2 and V3
for j in range(len(V2)):
   if V2[j] > 0:
      if V3[j] > 0:
         pp.append(j)
      else:
         pn.append(j)
   else:
      if V3[j] > 0:
         np.append(j)
      else:
         nn.append(j)

# Combine the orders of the groups
order = pp + pn + np + nn
m = len(Social)  # Get the size of Social
iden = n.eye(m)  # Identity matrix of size m
P = n.zeros((m, m))  # Initialize permutation matrix

# Create the permutation matrix
for j in range(m):
   P[j, :] = iden[order[j], :]

# Permute the adjacency matrix
```

```
SocialOrdered = P @ Social @ P.T

# Plot the permuted adjacency matrix
plt.figure(figsize=(8, 6))
plt.spy(SocialOrdered, markersize=1)
plt.title("Spy Plot of Permuted Adjacency Matrix (SocialOrdered)")
plt.xlabel("Nodes")
plt.ylabel("Nodes")
plt.grid(False)  # Disable the grid
plt.show()
```

15. An alternative way to obtain more clusters is to use the Fiedler vector procedure iteratively, meaning that we will apply it again to the clusters obtained in the previous step. This can be accomplished by running the following code:

    Observe that the densely connected components obtained by the last two methods might be different. In the context of a social network, the dense clusters may represent groups of people who have common connections in real life. For instance, your family, your friends from high school, your colleagues, etc.

```
# Step 15: Fiedler vector procedure iteratively for clusters
import numpy as np
import matplotlib.pyplot as plt

# Assuming 'Social' is your adjacency matrix, and 'pos' and 'neg' are your positive and negative indices
# Define SocialPos and SocialNeg based on the positive and negative indices
SocialPos = Social[np.ix_(pos, pos)]
SocialNeg = Social[np.ix_(neg, neg)]

# Calculate the Laplacian for the positive group
rowsumpos = np.sum(SocialPos, axis=1)
DiagSocialPos = np.diag(rowsumpos)
LaplaceSocialPos = DiagSocialPos - SocialPos

# Eigen decomposition for positive group
DPos , VPos = np.linalg.eig(LaplaceSocialPos)
d, ind = np.argsort(DPos), np.argsort(DPos)
DPos = np.diag(DPos[ind])
VPos = VPos[:, ind]
V2Pos = VPos[:, 1]  # Second smallest eigenvector for positive group

# Group positive nodes
posp = []  # Positive group
posn = []  # Negative group
for j in range(len(V2Pos)):
  if V2Pos[j] > 0:
        posp.append(pos[j])  # Append original index
    else:
        posn.append(pos[j])  # Append original index

# Calculate the Laplacian for the negative group
rowsumneg = np.sum(SocialNeg, axis=1)
DiagSocialNeg = np.diag(rowsumneg)
LaplaceSocialNeg = DiagSocialNeg - SocialNeg

# Eigen decomposition for negative group
DNeg , VNeg = np.linalg.eig(LaplaceSocialNeg)
d, ind = np.argsort(DNeg), np.argsort(DNeg)
DNeg = np.diag(DNeg[ind])
VNeg = VNeg[:, ind]
V2Neg = VNeg[:, 1]  # Second smallest eigenvector for negative group

# Group negative nodes
negp = []  # Positive group
negn = []  # Negative group
for j in range(len(V2Neg)):
    if V2Neg[j] > 0:
        negp.append(neg[j])  # Append original index
    else:
        negn.append(neg[j])  # Append original index

# Generate the final order for the permutation
ordergen = posp + posn + negp + negn

# Create the permutation matrix
m = len(Social)  # Assuming the size of Social
iden = np.eye(m)  # Identity matrix of size m
P = np.zeros((m, m))  # Initialize permutation matrix

# Create the permutation matrix
for j in range(m):
    P[j, :] = iden[ordergen[j], :]  # Filling the permutation matrix based on ordergen

# Permute the adjacency matrix
SocialOrderedGen = P @ Social @ P.T  # Permutation of the Social matrix

# Plot the permuted adjacency matrix
plt.figure(figsize=(10, 8))
plt.spy(SocialOrderedGen, markersize=1)
plt.title("Spy Plot of Permuted Adjacency Matrix (SocialOrderedGen)")
plt.xlabel("Nodes")
plt.ylabel("Nodes")
plt.grid(False)  # Disable grid for clarity
plt.show()
```

## Project 14: Singular Value Decomposition and image compression

**Goals:** In this project we will discuss a singular value decomposition of a matrix and its applications to image compression and noise filtering.

**To get started:**

- Download the file lab14.py and put it in your working directory[18].
- Download the files einstein.jpg[19] and checkers.pgm[20] and put them in your working directory.

[18] The printout of the file lab14.m can be found in the appendices of this book.

[19] The image is available at: https://commons.wikimedia.org/wiki/File:Albert_Einstein_Head.jpg

[20] Image from page 446 of ”The standard Hoyle; a complete guide and reliable authority upon all games of chance or skill now played in the United States, whether of native or foreign introduction” (1909), image appears on Flickr Commons.

**Python commands used:** del, np.linspace, plt.subplot, plt.quiver, plt.plot, plt.hold, np.linalg.svd, plt.axis, plt.title, plt.imread, plt.imshow, np.size, np.min, for... in, np.cos, np.sin, np.transpose, np.random.rand, np.double, np.ones

**What you have to submit:** The file lab14.py which you will modify during the lab session.

## INTRODUCTION

Any $m \times n$ matrix $\mathbf{A}$ possesses a singular value decomposition of the form:

$$\mathbf{A} = \mathbf{U\Sigma V}^T,$$

where $\mathbf{U}$ is an orthogonal $m \times m$ matrix satisfying the condition $\mathbf{U}^T\mathbf{U} = \mathbf{I}_m$, $\mathbf{\Sigma}$ is an $m \times n$ rectangular diagonal matrix with nonnegative values $\sigma_1,\sigma_2,...,\sigma_{min(m,n)}$ on the main diagonal, and $\mathbf{V}$ is an orthogonal $n \times n$ matrix satisfying the condition $\mathbf{V}^T\mathbf{V} = \mathbf{I}_n$. The nonnegative numbers $\sigma_1,\sigma_2,...,\sigma_{min(m,n)}$ are called the singular values of the matrix $\mathbf{A}$. They are arranged in the decreasing order: $\sigma_1 \geq \sigma_2 \geq ... \geq \sigma_{min(m,n)} \geq 0$. Observe that the singular value decomposition of a matrix is not unique.

Singular value decomposition is important for implementation of many numerical algorithms in linear algebra. In this project we will look at the geometric meaning of singular value decomposition and two applications related to image processing and noise filtering. Observe that more efficient image compression methods exist. The examples in this project are used mainly to show the reduction in the amount of data (so-called dimension reduction) which can be accomplished with singular value decomposition. Dimension reduction is an important tool in many practical applications dealing with large amounts of data, such as statistics, data science, and machine learning.

## TASKS

1. We will start with an illustration of the geometric meaning of singular value decomposition. Let us look at a singular value decomposition of a 2 × 2 matrix. Open the file lab16.m, locate the code cell # 2x2 matrix, and add the following commands
   This code will create a 2 × 100 matrix $\mathbf{X} = [\mathbf{x}_1\,\mathbf{x}_2 \cdots \mathbf{x}_{100}]$ whose columns $\mathbf{x}_i$ are unit vectors pointing in various directions. The plot will show a blue circle corresponding to the endpoints of these vectors and two vectors of the standard basis on the plane $e_1 = (1,0)^T$ and $e_2 = (0,1)^T$. We use the function subplot which will create a plot containing four subplots arranged in two rows and two columns. The plot above will occupy the first "cell" of this plot. Observe the command quiver which draws a vector with the beginning point given by the first two arguments ((0,0) in this case) and an ending point given by the next two arguments ((1,0) or (0,1) in the code above). Variables: X

```
import numpy as np
import matplotlib.pyplot as plt

# Task 1: Plotting the unit circle and basis vectors
```

```
t = np.linspace(0, 2 * np.pi, 100)
X = np.array([np.cos(t), np.sin(t)])
plt.subplot(2, 2, 1)
plt.plot(X[0, :], X[1, :], 'b')
plt.quiver(0, 0, 1, 0, color='r', angles='xy', scale_units='xy', scale=1)
plt.quiver(0, 0, 0, 1, color='g', angles='xy', scale_units='xy', scale=1)
plt.axis('equal')
plt.title('Unit circle')
plt.show()
```

2. Now in the M-file, define a variable A holding the matrix

   ```
   A = [ 2, 1; -1, 1 ];
   ```

   and compute the singular value decomposition of this matrix using the svd function:

   ```
   [U,S,V] = svd(A);
   ```

   Using the workspace window of the Python main environment, check out the matrices U, S, V. Perform the commands

   ```
   U'*U
   V'*V
   ```

   to ascertain that the matrices U and V are orthogonal. The output should produce 2 × 2 identity matrices.
   Variables: A, U, S, V

   ```
   A = np.array([[2, 1], [-1, 1]])
   U, S, V = np.linalg.svd(A)
   print("U:\n", U)
   print("S:\n", S)
   print("V:\n", V)

   # Verify orthogonality
   print("U' * U:\n", np.dot(U.T, U))
   print("V' * V:\n", np.dot(V.T, V))
   ```

3. Next, let us observe the geometric meaning of the individual matrices **U**, **Σ**, **V** (U, S, V in our Python code) in the singular value decomposition. To do this, observe the transformations induced by these matrices on a unit circle and the vectors of the standard basis $\mathbf{e}_1$, $\mathbf{e}_2$. Let us start by multiplying the coordinates of the points of the circle and the vectors $\mathbf{e}_1$, $\mathbf{e}_2$ by the matrix **V**. Execute the following code:

   Observe that the matrix VX contains the vectors of the matrix X transformed by the multiplication by the matrix $\mathbf{V}^T$. Since the matrices **V** and $\mathbf{V}^T$ are orthogonal, multiplication by the matrix $\mathbf{V}^T$ is equivalent to rotation of a plane, possibly in combination with a reflection along some straight line. This allows us to conjecture that the image of the unit circle under this mapping will still be a unit circle, but the vectors of the basis will be rotated and possibly switched in orientation.

Variables: VX

Q1: Did the multiplication by the transpose of the matrix V resulted in a reflection of the plane?

```
VX = np.dot(V.T, X)
plt.subplot(2, 2, 2)
plt.plot(VX[0, :], VX[1, :], 'b')
plt.quiver(0, 0, V[0, 0], V[0, 1], color='r', angles='xy', scale_units='xy', scale=1)
plt.quiver(0, 0, V[1, 0], V[1, 1], color='g', angles='xy', scale_units='xy', scale=1)
plt.axis('equal')
plt.title('Multiplied by matrix V^T')
plt.show()
```

4. Now, let us multiply the result from the previous step by the matrix **Σ** (S in the Python code). Observe that since the matrix **Σ** is diagonal, then multiplication by this matrix geometrically means stretching of the plain in two directions. To verify this, execute the following Python code:

   Observe that, as expected, the unit circle is stretched and becomes an ellipsis. The images of the standard basis vectors are stretched as well. Variables: SVX

```
S_matrix = np.diag(S)
SVX = np.dot(S_matrix, VX)
plt.subplot(2, 2, 3)
plt.plot(SVX[0, :], SVX[1, :], 'b')
plt.quiver(0, 0, S[0] * V[0, 0], S[1] * V[0, 1], color='r', angles='xy', scale_units='xy', scale=1)
plt.quiver(0, 0, S[0] * V[1, 0], S[1] * V[1, 1], color='g', angles='xy', scale_units='xy', scale=1)
plt.axis('equal')
plt.title('Multiplied by matrix ΣV^T')
plt.show()
```

5. Finally, multiply the results from the last step by the matrix **U** to obtain:

   Observe that the result is equivalent to multiplying the initial vector X by the matrix A. Since the matrix U is orthogonal, then the multiplication by this matrix should result in a rotation of the plane possibly combined with a reflection. Confirm this by observing the images of the basis vectors.

   Variables AX

   Q2: Did multiplication by the matrix U produce a reflection of the plane?

```
AX = np.dot(U, SVX)
plt.subplot(2, 2, 4)
plt.plot(AX[0, :], AX[1, :], 'b')
plt.quiver(0, 0, U[0, 0] * S[0] * V[0, 0] + U[0, 1] * S[1] * V[0, 1],
           U[1, 0] * S[0] * V[0, 0] + U[1, 1] * S[1] * V[0, 1], color='r', angles='xy',
scale_units='xy', scale=1)
plt.quiver(0, 0, U[0, 0] * S[0] * V[1, 0] + U[0, 1] * S[1] * V[1, 1],
           U[1, 0] * S[0] * V[1, 0] + U[1, 1] * S[1] * V[1, 1], color='g', angles='xy',
scale_units='xy', scale=1)
plt.axis('equal')
plt.title('Multiplied by matrix UΣV^T=A')
plt.show()
```

6. If you answered yes to both Q1 and Q2 above, can you modify the matrices U and V in such a way that no reflections of the plane occur? Produce the modified matrices U1 and V1 and confirm that U1*S*V1’=A. Observe that this shows that singular value decomposition is not unique.
   Variables U1, V1

```
# Modification example for U and V (this is just a random example, modifications need to be chosen carefully)
U1 = U
V1 = V.T
print("U1 * S * V1.T:\n", np.dot(U1, np.dot(S_matrix, V1.T)))
```

7. Finally, observe that

$$[\mathbf{A}\mathbf{v}_1\ \mathbf{A}\mathbf{v}_2] = \mathbf{AV} = \mathbf{USV}^T\mathbf{V} = \mathbf{US} = [\mathbf{u}_1\ \mathbf{u}_2]\begin{bmatrix}\sigma_1 & 0\\ 0 & \sigma_2\end{bmatrix} = [\sigma_1\mathbf{u}_1\ \sigma_2\mathbf{u}_2],$$

   showing that

$$\mathbf{A}\mathbf{v}_1 = \sigma_1\mathbf{u}_1 \qquad \text{and} \qquad \mathbf{A}\mathbf{v}_2 = \sigma_2\mathbf{v}_2.$$

   Check this fact numerically by computing the expression A*V-U*S.

```
Av1 = np.dot(A, V.T[:, 0])
Av2 = np.dot(A, V.T[:, 1])
print("Av1:\n", Av1)
print("σ1 * u1:\n", S[0] * U[:, 0])
print("Av2:\n", Av2)
print("σ2 * u2:\n", S[1] * U[:, 1])

# Numerical check
print("A * V - U * S:\n", np.dot(A, V.T) - np.dot(U, S_matrix))
```

8. Now we will look at image compression using SVD. Add the following commands to your

M-file:

```
# Image compression clear;
ImJPG=imread('einstein.jpg'); figure;
imshow(ImJPG); [m,n]=size(ImJPG);
```

This code loads an image as a uint8 matrix and displays it on the screen. Each entry in the matrix corresponds to a pixel on the screen and takes a value somewhere between 0 (black) and 255 (white).

Variables: ImJPG

9. Perform a singular value decomposition of the matrix ImJPG and save the output in matrices UIm, SIm, and VIm. Since ImJPG is integer-valued, you will need to use svd(double(ImJPG)). Variables: UIm, SIm, VIm

10. Plot the singular values using the code:

    This shows the singular values (the diagonal entries of the SIm matrix) for the image matrix ImJPG. Notice that the diagonal entries of SIm are ordered in the decreasing order $\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_{min(m,n)} \geq 0$.

11. The idea behind compression of data with singular value decomposition is in the following. Using the singular value decomposition, the matrix **A** can be written in the form:

$$\mathbf{A} = \sigma_1 \mathbf{u}_1 \mathbf{v}_1^T + \sigma_2 \mathbf{u}_2 \mathbf{v}_2^T + \cdots + \sigma_r \mathbf{u}_r \mathbf{v}_r^T$$

where $r$ is the rank of **A** and $\mathbf{u}_i$ and $\mathbf{v}_i$ are the $i$th columns of **U** and **V** respectively.

Observe that since the singular values are arranged in the decreasing order, the first terms in this sum provide a larger contribution to the matrix **A** then the subsequent terms. It may happen that for some $k < r$, $\sigma_{k+1}$ is small compared to $\sigma_1$, and correspondingly does not affect the matrix **A** too much. We should then expect

$$\mathbf{A} \approx \sigma_1 \mathbf{u}_1 \mathbf{v}_1^T + \sigma_2 \mathbf{u}_2 \mathbf{v}_2^T + \cdots + \sigma_k \mathbf{u}_k \mathbf{v}_k^T$$

to be a good approximation to the matrix **A** (this is called a truncated SVD). In fact, there is a theorem in linear algebra which states that SVD truncated to the $k$ terms is the best approximation to the matrix **A** among the matrices of the rank at most $k$ in the sense of Frobenius norm (which is equivalent to the regular Euclidean norm if we consider the matrix **A** as an $mn$-dimensional vector).

This idea can be used for image compression as follows. Instead of storing the whole $m \times n$ matrix **A**, we can instead store the $m \times k$ and $n \times k$ matrices

$$\mathbf{C} = [\mathbf{u}_1\, \mathbf{u}_2 \cdots \mathbf{u}_k] \qquad \text{and} \qquad \mathbf{D} = [\sigma_1 \mathbf{v}_1\, \sigma_2 \mathbf{v}_2 \cdots \sigma_k \mathbf{v}_k].$$

If $k$ is much smaller than $min(m,n)$, then storing **C** and **D** will take much less space than storing **A**. Moreover, if we wish to display the image, we can reconstruct the approximation of **A** as $\mathbf{A} \approx \mathbf{CD}^T$.

Add the following code to your M-file:

This code compresses the image as described above using $k$ = 50, $k$ = 100, and $k$ = 150 singular values and displays the reconstructed image. Compare the reconstructed images with the original. The code also displays the compression percentage as pct.

Variables: ImJPG _comp, pct

Combined solution for 8,9,10,11

```
import cv2

# Load the image
ImJPG = cv2.imread("C:/Users/Harshith/Downloads/laproject/Albert_Einstein_Head.jpg", cv2.IMREAD_GRAYSCALE)
plt.figure()
plt.imshow(ImJPG, cmap='gray')
plt.title('Original Image')
plt.show()

# Singular Value Decomposition
UIm, SIm, VIm = np.linalg.svd(ImJPG.astype(np.float64), full_matrices=False)
print(UIm)

# Plot Singular Values
plt.figure()
plt.plot(np.arange(len(SIm)), SIm)
plt.title('Singular Values')
plt.show()

# Image compression using truncated SVD
for k in [50, 100, 150]:
    ImJPG_comp = np.dot(UIm[:, :k], np.dot(np.diag(SIm[:k]), VIm[:k, :]))
    plt.figure()
    plt.imshow(ImJPG_comp, cmap='gray')
    plt.title(f'Compressed Image with {k} Singular Values')
    plt.show()
     pct = 1 - (np.size(UIm[:, :k]) + np.size(VIm[:k, :]) * np.size(np.diag(SIm[:k]))) / np.size(ImJPG)
    print(f'Compression percentage for {k} singular values: {pct:.3f}')
```

12. Observe that the singular value decomposition can also be used to smooth noisy data, especially if the data contains patterns. Data smoothing is often necessary because all measurements contain small errors resulting in a “noise”. This noise usually determines the smallest singular values of the matrix. Dropping these small values, thus, not only saves the storage space, but also allows to eliminate noise from the data.

Start a new code cell. Load the file checkers.pgm into Python and add some noise to the resulting image matrix using the following code:

Variables: ImJPG, ImJPG Noisy

```
#12

import numpy as np
import matplotlib.pyplot as plt
import cv2

# Load the image
ImJPG = cv2.imread("C:/Users/Harshith/Downloads/laproject/materials/checkers.pgm",
cv2.IMREAD_GRAYSCALE)

# Add noise to the image
m, n = ImJPG.shape
ImJPG_Noisy = ImJPG.astype(np.float64) + 50 * (np.random.rand(m, n) - 0.5)
ImJPG_Noisy = np.clip(ImJPG_Noisy, 0, 255)  # Ensure values are within valid range

# Display the original and noisy images
plt.figure()
plt.imshow(ImJPG, cmap='gray')
plt.title('Original Checkers Image')
plt.axis('off')
plt.show()

plt.figure()
plt.imshow(ImJPG_Noisy, cmap='gray')
plt.title('Noisy Checkers Image')
plt.axis('off')
plt.show()

# Variables: ImPGM, ImPGM_Noisy
```

13. Compute the SVD of the matrix ImJPG _Noisy and save the resulting decomposition matrices as UIm, SIm, and VIm.
    Variables: UIm, SIm, VIm

```
# Compute SVD of the noisy image
UIm, SIm, VIm = np.linalg.svd(ImJPG_Noisy, full_matrices=False)

# Variables: UIm, SIm, VIm
```

14. Compute the approximations of the initial image with $k$ = 10, $k$ = 30, and $k$ = 50 singular values. Display the resulting approximations and compare then to the "noisy" image. Observe that SVD significantly reduces the noise. Compare the images to the initial image

without noise. Observe also that even though SVD reduces the noise, it also somewhat blurs the image.

```
# Function to approximate the image with k singular values
def approximate_image(U, S, V, k):
    return np.dot(U[:, :k], np.dot(np.diag(S[:k]), V[:k, :]))

# Approximations with k = 10, k = 30, k = 50 singular values
ks = [10, 30, 50]
for k in ks:
    ImJPG_approx = approximate_image(UIm, SIm, VIm, k)
    plt.figure()
    plt.imshow(ImJPG_approx, cmap='gray')
    plt.title(f'Denoised Image with k = {k} Singular Values')
    plt.axis('off')
    plt.show()

    # Compare the images to the initial noisy image
    plt.figure()
    plt.imshow(np.hstack((ImJPG, ImJPG_Noisy, ImJPG_approx)), cmap='gray')
    plt.title(f'Original, Noisy, and Denoised (k = {k}) Images')
    plt.axis('off')
    plt.show()
```

# Appendices

## Printout of the file lab09.m

```
# clear;
Vertices=[1 1 1;...
        -1 1 1;...
        1 -1 1;...
        1 1 -1;...
        -1 -1 1;...
        -1 1 -1;...
        1 -1 -1;...
        -1 -1 -1];

Edges=zeros(8,8);
Edges(1,2)=1;
Edges(1,3)=1;
Edges(1,4)=1;
Edges(2,5)=1;
Edges(2,6)=1;
Edges(3,5)=1;
Edges(3,7)=1;
Edges(4,6)=1;
Edges(4,7)=1;
Edges(5,8)=1;
Edges(6,8)=1;
Edges(7,8)=1; Edges=Edges+Edges';

theta1=pi/3; theta2=pi/4; theta3=pi/6;

rotmat=rotation(theta1,theta2,theta3);

VertRot=Vertices*rotmat;

figure; axis
equal; hold
on;

for j=1:8 for k=j+1:8 if (Edges(j,k)==1) line([VertRot(j,1),VertRot(k,1)],[VertRot(j,2),VertRot(k,2)]);
              end;
        end;
end; hold

off;

# Buckyball
[Edges,Vertices] = bucky; figure;
axis equal; hold on;

theta1=pi/3; theta2=pi/4; theta3=pi/6;

rotmat=rotation(theta1,theta2,theta3);

VertRot=Vertices*rotmat;
```

```
for j=1:60 for k=j+1:60 if (Edges(j,k)==1)
        line([VertRot(j,1),VertRot(k,1)],[VertRot(j,2),VertRot(k,2)]);
                end;
         end;
end; hold

off;

figure; axis
equal; hold
on;

for j=1:60 for k=j+1:60 if (Edges(j,k)==1) line([Vertices(j,1),Vertices(k,1)],[Vertices(j,2),Vertices(k,2)],...
        [Vertices(j,3),Vertices(k,3)]); end;
         end;
end; hold

off;

# 3D fawn clear;

load('v.mat','v'); load('f.mat','f');

[mVert,nVert]=size(v); [mFace,nFace]=size(f);


figure; axis equal; hold on; for j=1:mFace line([v(f(j,1),1),v(f(j,2),1)],[v(f(j,1),2),v(f(j,2),2)],[v(f(j,1),3),v(f(j,2),3)]);
line([v(f(j,1),1),v(f(j,3),1)],[v(f(j,1),2),v(f(j,3),2)],[v(f(j,1),3),v(f(j,3),3)]);
line([v(f(j,2),1),v(f(j,3),1)],[v(f(j,2),2),v(f(j,3),2)],[v(f(j,2),3),v(f(j,3),3)]);
end; hold
off;

figure; axis
equal; hold
on;

theta1=pi/3; theta2=pi/4; theta3=pi/2;

rotmat=rotation(theta1,theta2,theta3);

VertRot=v*rotmat;

for j=1:mFace line([VertRot(f(j,1),1),VertRot(f(j,2),1)],[VertRot(f(j,1),2),VertRot(f(j,2),2)]);
line([VertRot(f(j,1),1),VertRot(f(j,3),1)],[VertRot(f(j,1),2),VertRot(f(j,3),2)]);
line([VertRot(f(j,2),1),VertRot(f(j,3),1)],[VertRot(f(j,2),2),VertRot(f(j,3),2)]); end;
```

## Printout of the file lab10.m

```
% The Chaos Game

# Sierpinski triangle

# Sierpinski triangle with a change of ratio

# Sierpinski triangle with rotation

# Square

# Square with different ratio

# Square with vertex preference (new vertex cannot be the same as previous) clear;

T=[1/2 0; 0 1/2];

t=linspace(0,2*pi,5); t(5)=[];
v=[cos(t); sin(t)]; x(:,1)=[rand-0.5;
rand-0.5];

plot(v(1,:),v(2,:),’k*’,x(1,1),x(2,1),’b.’); axis equal; hold on;
Num=5000;

k1=0; for j=1:Num
k=randi(3); if
(k>=k1) k=k+1;
end;
        x(:,j+1)=T*(x(:,j)-v(:,k))+v(:,k); k1=k;
end;

plot(x(1,:),x(2,:),’b.’); hold off;

# Square with vertex preference (new vertex cannot be opposite of the previous) clear;

T=[1/2 0; 0 1/2];

t=linspace(0,2*pi,5); t(5)=[];
v=[cos(t); sin(t)]; x(:,1)=[rand-0.5;
rand-0.5];
plot(v(1,:),v(2,:),’k*’,x(1,1),x(2,1),’b.’
); axis equal; hold on; Num=5000;

w=[real(v(1));imag(v(1))]; k1=0;

for j=1:Num
        k=randi(4) if
        (k~=k1+2)&&(k1~=k+2)
                x(:,j+1)=T*(x(:,j)-v(:,k))+v(:,k); k1=k; else
        x(:,j+1)=x(:,j);
```

```
      end;
end;

plot(x(1,:),x(2,:),’b.’); hold off;

# Barnsley fern clear;

T1=[0.85,0.04;-0.04,0.85];
T2=[-0.15,0.28;0.26,0.24];
T3=[0.2,-0.26;0.23,0.22];
T4=[0,0;0,0.16];
Q1=[0;1.64];
Q2=[-0.028;1.05];
Q3=[0;1.6];
Q4=[0,0];
P1=0.85;
P2=0.07;
P3=0.07;
P4=0.01;

Num=15000; x(:,1)=rand(2,1);
plot(x(1,:),x(2,:),’b.’) axis equal;
hold on; for j=1:Num r=rand; if
r<=P1
          x(:,j+1)=T1*x(:,j)+Q1;
      elseif r<=P1+P2 x(:,j+1)=T2*x(:,j)+Q2;
      elseif r<=P1+P2+P3
           x(:,j+1)=T3*x(:,j)+Q3; else
          x(:,j+1)=T4*x(:,j);
     end
end

plot(x(1,:),x(2,:),’b.’); hold off;

# Hexagon clear;

T=[1/3 0; 0 1/3];

t=linspace(0,2*pi,7); t(7)=[];
v=[cos(t); sin(t)]; x(:,1)=[rand-0.5;
rand-0.5];

plot(v(1,:),v(2,:),’k*’,x(1,1),x(2,1),’b.’); axis equal; hold on;
Num=10000;

for j=1:Num
     k=randi(6); x(:,j+1)=T*(x(:,j)-v(:,k))+v(:,k);
end; plot(x(1,:),x(2,:),’b.’); hold off;


# Pentagon with a skipped vertex
```

```
clear;
T=[1/2.5 0; 0 1/2.5];

t=linspace(0,2*pi,6); t(6)=[];
v=[cos(t); sin(t)]; x(:,1)=[rand-0.5;
rand-0.5];

plot(v(1,:),v(2,:),’k*’,x(1,1),x(2,1),’b.’); axis equal; hold on;
Num=10000; k1=0;

for j=1:Num
    k=randi(5); if
    k~=k1
        x(:,j+1)=T*(x(:,j)-v(:,k))+v(:,k);
         k1=k; else
     x(:,j+1)=x(:,j);
     end;
end; plot(x(1,:),x(2,:),’b.’); hold off;
```

## **Printout of the file** lab11.m

```
% Simple face recognition algorithm # Input
database files into the Python clear;
Database_Size=30;

%Reading images from the database. The image files should be located in the
%subfolder "database" for j=1:Database_Size
image_read=imread(['database\person' num2str(j) '.pgm']);
        [m,n]=size(image_read);
       P(:,j)=reshape(image_read,m*n,1); end;

# Computing and displaying the mean face

# Subtract the mean face

# Compute the covariance matrix of the set and its eigenvalues and eigenvectors

# Display the set of eigenvaces for
j=2:Database_Size; if j==2
               EigenFaces=reshape(EigenVectors(:,j)+mean_face,m,n); else
              EigenFaces=[EigenFaces reshape(EigenVectors(:,j)+mean_face,m,n)]; end; end
EigenFaces=uint8(EigenFaces);

figure; imshow(EigenFaces);

# Orthogonality and symmetry

# Recognition of an altered image (sunglasses)
image_read=imread(['database\person30altered1.pgm']); U=reshape(image_read,m*n,1);

NormsEigenVectors=diag(Products);
W=(EigenVectors'*(double(U)-mean_face));
W=W./NormsEigenVectors; U_approx=EigenVectors*W+mean_face;

image_approx=uint8(reshape(U_approx,m,n)); figure;
imshow([image_read,image_approx])

# Recognition of an altered image (lower part of the face)

# Recognition of a person not in the database
```

## **Printout of the file** lab12.m

```
% Google PageRank algorithm on the example of random network

# Load the network data clear;


# Display a small amount of network
NumNetwork=500; AdjMatrixSmall=AdjMatrix(1:NumNetwork,1:NumNetwork);

for j=1:NumNetwork
coordinates(j,1)=NumNetwork*rand;
coordinates(j,2)=NumNetwork*rand; end;
gplot(AdjMatrixSmall,coordinates,’k-*’);

# Check the amount of links originating from each webpage NumLinks=sum(AdjMatrixSmall,2);

# Create a matrix of probabilities (Google matrix)
% Element (i,j) of the matrix shows the probability of moving from i-th
% page of the network to jth page. It is assumed that the user can follow
% any link on the page with a total probability of 85% (all hyperlinks are % equal), and jump
(teleport) to any other page in the network with a total % probability of 15% (again, all pages are
equal).

alpha=0.15;
GoogleMatrix=zeros(NumNetwork,NumNetwork); for
i=1:NumNetwork if NumLinks(i)~=0
              GoogleMatrix(i,:)=AdjMatrixSmall(i,:)./NumLinks(i); else
             GoogleMatrix(i,:)=1./NumNetwork; end; end; GoogleMatrix=(1-
alpha)*GoogleMatrix+alpha*ones(NumNetwork,NumNetwork)./NumNetwork;

# Check that all the rows in the GoogleMatrix matrix sum to 1 SumGoogleMatrix=sum(GoogleMatrix,2);

# Finding an eigenvector corresponding to 1 (why is there sucj an eigenvector)?
w0=ones(1,NumNetwork)./sqrt(NumNetwork);

w1=w0*GoogleMatrix; w2=w1*GoogleMatrix;
w3=w2*GoogleMatrix;

w100=w0*(GoogleMatrix)^100; w90=w0*(GoogleMatrix)^90;
% Check the difference between v30 and v20. Observe that it is pretty
% small deltaw=w100-w90;

# Compute the eigenvalues and the right eigenvectors
[VectRight,LamRight]=eig(GoogleMatrix);
% Explain the result
LamRight=diag(LamRight);

# Compute the eigenvalues and the left eigenvectors
[VectLeft,LamLeft]=eig(GoogleMatrix’);
LamLeft=diag(LamLeft);

# Separate the eigenvector corresponding to the eigenvalue 1 and scale it u1=VectLeft(:,1);
u1=abs(u1)/norm(u1);

# Select the maximum element and the corresponding element. %Which page is the
most important in the network?

[MaxRank,PageMaxRank]=max(u1);

# Check if it’s the most popular (most linked to page):

MostLinks=sum(AdjMatrixSmall);
[MaxLinks,PageMaxLinks]=max(MostLinks);
```

## Printout of the file lab13.m

```
% Facebook and graph partitioning

# Simple graph

# Load the data clear;
load('social.mat','Social');
spy(Social); # Laplacian

# Spectral decomposition

# Clusteer in two groups

# Cluster in 4 groups

Diagonal(indsmall)=large+1000;
[small,indsmall]=min(Diagonal); V3=V(:,indsmall);

pp=[]; pn=[];
np=[]; nn=[];

for j=1:m
        if (V2(j)>0) if
               (V3(j)>0)
                     pp=[pp,j];
              else pn=[pn,j];
              end;
        else
              if (V3(j)>0)
                     np=[np,j];
              else nn=[nn,j];
              end;
       end; end;

order=[pp,pn,np,nn];

iden=eye(m,m);

for j=1:351;
        for k=1:351;
              P(j,k)=iden(order(j),k);       end;
end;

SocialOrdered=P*Social*P'

figure; spy(SocialOrdered)

# Second order of Fiedler

SocialPos=Social(pos,pos); SocialNeg=Social(neg,neg);

rowsumpos=sum(SocialPos,2); DiagSocialPos=diag(rowsumpos);
```

```
LaplaceSocialPos=DiagSocialPos-SocialPos;

[VPos,DPos]=eig(LaplaceSocialPos);

V2Pos=VPos(:,2); [mpos,npos]=size(V2Pos);

posp=[]; posn=[];

for j=1:mpos
    if (V2Pos(j)>0)
        posp=[posp,pos(j)];
    else posn=[posn,pos(j)];
    end; end;

rowsumneg=sum(SocialNeg,2);

DiagSocialNeg=diag(rowsumneg);

LaplaceSocialNeg=DiagSocialNeg-SocialNeg;

[VNeg,DNeg]=eig(LaplaceSocialNeg);

V2Neg=VNeg(:,2); [mneg,nneg]=size(V2Neg);

negp=[]; negn=[];

for j=1:mneg
    if (V2Neg(j)>0)
        negp=[negp,neg(j)]; else
       negn=[negn,neg(j)];
    end; end;

ordergen=[posp,posn,negp,negn];

iden=eye(m,m);

for j=1:351;
    for k=1:351;
       P(j,k)=iden(ordergen(j),k);       end;
end;

SocialOrderedGen=P*Social*P’
figure; spy(SocialOrderedGen)
```

**Printout of the file** lab14.m # 2x2

```
matrix

clear;
t=linspace(0,2*pi,100);
X=[cos(t);sin(t)];
subplot(2,2,1); hold on;
plot(X(1,:),X(2,:),’b’);
quiver(0,0,1,0,0,’r’);
quiver(0,0,0,1,0,’g’); axis equal
title(’Unit circle’) hold off;

A = [ 2, 1; -1, 1 ];
[U,S,V] = svd(A);
U’*U
V’*V

VX=V’*X; subplot(2,2,2) hold on;
plot(VX(1,:),VX(2,:),’b’);
quiver(0,0,V(1,1),V(1,2),0,’r’);
quiver(0,0,V(2,1),V(2,2),0,’g’); axis equal
title(’Multiplied by matrix V^T’) hold off;

SVX = S*VX;
subplot(2,2,3); hold on; plot(SVX(1,:),SVX(2,:),’b’);
quiver(0,0,S(1,1)*V(1,1),S(2,2)*V(1,2),0,’r’);
quiver(0,0,S(1,1)*V(2,1),S(2,2)*V(2,2),0,’g’); axis equal
title(’Multiplied by matrix \Sigma V^T’) hold off;

AX = U*SVX;
subplot(2,2,4) hold on; plot(AX(1,:),AX(2,:),’b’);
quiver(0,0,U(1,1)*S(1,1)*V(1,1)+U(1,2)*S(2,2)*V(1,2),U(2,1)*S(1,1)*V(1,1)+...
        U(2,2)*S(2,2)*V(1,2),0,’r’);
quiver(0,0,U(1,1)*S(1,1)*V(2,1)+U(1,2)*S(2,2)*V(2,2),U(2,1)*S(1,1)*V(2,1)+...
        U(2,2)*S(2,2)*V(2,2),0,’g’); axis equal
title(’Multiplied by matrix U\Sigma V^T=A’) hold off;

# Modified SVD
U1(:,1)=U(:,1);
U1(:,2)=-U(:,2);
V1(:,1)=V(:,1);
V1(:,2)=-V(:,2);

U1*S*V1’-A

# Check A*V-U*S

# Image compression

% Creates a two-dimensional array with the dimensions equal to the dimensions of
% the image clear;
ImJPG=imread(’einstein.jpg’); figure;
imshow(ImJPG); [m,n]=size(ImJPG);

% Compute an SVD

[UIm,SIm,VIm]=svd(double(ImJPG));

% plot the singular values figure;
plot(1:min(m,n),diag(SIm));

# Create approximations to the image

% With 50, 100, and 150 singular values for
k=50:50:150
        ImJPG_comp=uint8(UIm(:,1:k)*SIm(1:k,1:k)*(VIm(:,1:k))’); figure;
        imshow(ImJPG_comp) % compression percentage
        pct = 1 - (numel(UIm(:,1:k))+numel(VIm(:,1:k)*SIm(1:k,1:k)))/numel(ImJPG); fprintf(’Compression
        percentage for %2.0f singular values: %8.3f\n’,k, pct);
end;

# Noise filtering clear;
ImJPG=imread(’checkers.pgm’)
[m,n]=size(ImJPG);

% Add some noise to the image
ImJPG_Noisy=double(ImJPG)+50*(rand(m,n)-0.5*ones(m,n)); figure;
imshow(ImJPG);

figure; imshow(uint8(ImJPG_Noisy));
[UIm,SIm,VIm]=svd(ImJPG_Noisy);

figure; plot(1:min(m,n),diag(SIm),’ko’);

for k=10:20:50
       ImJPG_comp=uint8(UIm(:,1:k)*SIm(1:k,1:k)*(VIm(:,1:k))’); figure; imshow(ImJPG_comp) %
compression percentage pct = 1 - (numel(UIm(:,1:k))+numel(VIm(:,1:k)*SIm(1:k,1:k)))/numel(ImJPG);
fprintf(’Compression percentage for %2.0f singular values: %8.3f\n’,k, pct); end;
```

# References

[1] S. Attaway. *Python: A practical Introduction to Programming and Problem solving.* Butterworth-Heinemann, 3rd edition, 2013.

[2] K. Bryan, T. Leise. *The $25,000,000,000 Eigenvector: The Linear Algebra behind Google.* SIAM Rev., **48(3)**, 569 - 581, 2006.

[3] T. Chartier. *When Life is Linear: From Computer Graphics to Bracketology.* Mathematical Association of America, 2015.

[4] W.N. Colley. *Colley's bias free college football ranking method: The Colley matrix explained.* www.Colleyrankings.com, 2002.

[5] L. Elden. *Numerical linear algebra in data mining.* Acta Numerica, **15**, 327-384, 2006.

[6] C. Hennig, M. Meila, F. Murthagh and R. Rocci (eds.). *Handbook of Cluster Analysis.* CRC Press, 2015.

[7] J. Keener. *The Perron–Frobenius Theorem and the Ranking of Football Teams.* SIAM Rev., **35(1)**, 80-93, 1993.

[8] A.N. Langville, C.D. Meyer. *Google's PageRank and Beyond: The Science of Search Engine Rankings.* Princeton University Press, 2006.

[9] J. Leskovec, R. Sosic. *SNAP: A General-Purpose Network Analysis and Graph-Mining Library,.* ACM Transactions on Intelligent Systems and Technology (TIST), **8(1)**, 2016.

[10] F.M. Harper, J.A. Konstan, *The MovieLens Datasets: History and Context.* ACM Transactions on Interactive Intelligent Systems (TiiS), **5(4)**, 2015.