Numerical Linear Algebra

done by

Jean-Claude S. MITCHOZOUNOU

Africa Business School, UM6P, Morocco

Master of Quantitative and Financial Modelling (QFM)

Practical work

Teacher: Prof. Lahcen Laayouni

Academic Year 2023-2024

CONTENTS

TP1

0.1 Exercise 1

• Let find the LU-factorization of the following matrix A in $\mathbb{R}^{n\times n}$.

$$A = \begin{bmatrix} 2 & -1 & 0 & 0 & \cdots & 0 \\ -1 & 2 & -1 & 0 & \cdots & 0 \\ 0 & -1 & 2 & -1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & 2 & -1 \\ 0 & 0 & 0 & \cdots & -1 & 2 \end{bmatrix}.$$

First of all we are going to proof the existence of this factorization.
Define A_k to be the $k\times k$ main sub matrixes of A.
One has: $det(A_1) = 2 \neq 0$ and $det(A_2) = det\left(\begin{bmatrix} 2 & -1 \\ -1 & 2 \end{bmatrix}\right) = 3 \neq 0$
Remark that

$$\begin{aligned} det(A_3) = det\left(\begin{bmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 2 \end{bmatrix}\right) &= 2det\left(\begin{bmatrix} 2 & -1 \\ -1 & 2 \end{bmatrix}\right) + (-1)(-1)^{3+2}det\left(\begin{bmatrix} 2 & 0 \\ -1 & -1 \end{bmatrix}\right) \\ &= 2det(A_2) - 2 \\ &= 2det(A_2) - det(A_1) \end{aligned}$$

We then have, by using cofactor expansion twice,

$$\det(A_n) = 2\det(A_{n-1}) - \det(A_{n-2}).$$

Calling $det(A_n) = d_n$, we have the following linear homogeneous recurrence relation:

$$d_n = 2d_{n-1} - d_{n-2}$$

The characteristic equation is

$$\begin{aligned} x^2 - 2x + 1 = 0 &\implies (x-1)^2 = 0 \\ &\implies x = 1 \end{aligned}$$

Since $x = 1$ is a double root of characteristic polynomial there exist constants k_1, k_2 such that:

$$d_n = (k_1 + nk_2)(1)^n = k_1 + nk_2$$

$$\begin{cases} d_1 = 2 \\ d_2 = 3 \end{cases} \iff \begin{cases} k_1 + k_2 = 2 \\ k_1 + 2k_2 = 3 \end{cases} \iff k_1 = k_2 = 1; \text{then } d_n = 1 + n.\text{①}$$

$d_n = det(A_n) = 1 + n \neq 0\ \forall n \implies$ A can be factorized en LU

Since A is triadiagonal matrix, the LU-factorization is reduced to a fews operations. We will refer to the Crout factorization wich requires that the matrix U has 1's on its diagonal. By imposing that the diagonal of U is composed only of 1 the decomposition takes the following form:

$$\begin{bmatrix} 2 & -1 & 0 & 0 & \cdots & 0 \\ -1 & 2 & -1 & 0 & \cdots & 0 \\ 0 & -1 & 2 & -1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & 2 & -1 \\ 0 & 0 & 0 & \cdots & -1 & 2 \end{bmatrix} = \begin{bmatrix} l_{11} & 0 & 0 & 0 & \cdots & 0 \\ l_{21} & l_{22} & 0 & 0 & \cdots & 0 \\ 0 & l_{32} & l_{33} & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & l_{(n-1)(n-1)} & 0 \\ 0 & 0 & 0 & \cdots & l_{nn-1} & l_{nn} \end{bmatrix} \begin{bmatrix} 1 & u_{12} & 0 & 0 & \cdots & 0 \\ 0 & 1 & u_{23} & 0 & \cdots & 0 \\ 0 & 0 & 1 & u_{34} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & 1 & u_{n-1n} \\ 0 & 0 & 0 & \cdots & 0 & 1 \end{bmatrix}.$$

We will do the LU-factorization for n = 5 before generalizing.

$$\begin{bmatrix} 2 & -1 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & -1 & 2 \end{bmatrix} = \begin{bmatrix} l_{11} & 0 & 0 & 0 & 0 \\ l_{21} & l_{22} & 0 & 0 & 0 \\ 0 & l_{32} & l_{33} & 0 & 0 \\ 0 & 0 & l_{43} & l_{44} & 0 \\ 0 & 0 & 0 & l_{54} & l_{55} \end{bmatrix} \begin{bmatrix} 1 & u_{12} & 0 & 0 & 0 \\ 0 & 1 & u_{23} & 0 & 0 \\ 0 & 0 & 1 & u_{34} & 0 \\ 0 & 0 & 0 & 1 & u_{45} \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}.$$

$l_{11} = 2 = \frac{1+1}{1}$, $l_{2,1} = l_{3,2} = l_{4,3} = l_{5,4} = -1$;
$l_{11}u_{12} = -1 \iff u_{12} = -\frac{1}{2}$;
$l_{22} + l_{21}u_{12} = 2 \iff l_{22} = 2 - l_{21}u_{12} \iff l_{22} = \frac{3}{2} = \frac{2+1}{2}$;
$l_{22}u_{23} = -1 \iff u_{23} = -\frac{1}{l_{22}} = -\frac{2}{3}$;
$l_{33} + l_{32}u_{23} = 2 \iff l_{33} = 2 - l_{32}u_{23} \iff l_{33} = 2 - \frac{2}{3} = \frac{4}{3} = \frac{3+1}{3}$;

$l_{33}u_{34} = -1 \iff u_{34} = -\frac{1}{l_{33}} = -\frac{3}{4}$;
Similarly one obtains: $l_{44} = \frac{5}{4} = \frac{4+1}{3}$, $l_{55} = \frac{6}{5} = \frac{5+1}{5}$ and $u_{4,5} = -\frac{4}{5}$.

By generalizing, one deduces that:

the matrix $L = (l_{ij})i, j$ is such that: $l_{ij} = \begin{cases} -1 \text{ if } i = j+1 \\ \frac{i+1}{i} \text{ if } i = j \\ 0 \text{ if not} \end{cases}$

and

the matrix $U = (u_{ij})i, j$ is such that: $u_{ij} = \begin{cases} -\frac{i}{j} \text{ if } j = i+1 \\ 1 \text{ if } i = j \\ 0 \text{ if not} \end{cases}$

$$\boxed{L = \begin{bmatrix} 2 & 0 & 0 & 0 & \cdots & 0 \\ -1 & \frac{3}{2} & 0 & 0 & \cdots & 0 \\ 0 & -1 & \frac{4}{3} & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \frac{n}{n-1} & 0 \\ 0 & 0 & 0 & \cdots & -1 & \frac{n+1}{n} \end{bmatrix}; U = \begin{bmatrix} 1 & -\frac{1}{2} & 0 & 0 & \cdots & 0 \\ 0 & 1 & -\frac{2}{3} & 0 & \cdots & 0 \\ 0 & 0 & 1 & -\frac{3}{4} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & 1 & -\frac{n-1}{n} \\ 0 & 0 & 0 & \cdots & 0 & 1 \end{bmatrix}, A = LU}$$

• Let verify if the matrix A is positive definite, or positive semi-definite

From the previous question we have shown that for all A_k be $k \times k$ main sub matrix of A, $det(A_k) = 1 + k$ (from ①). But $(1 + k) > 0\ \forall k$. Thus $det(A_k) > 0\ \forall k$. From Sylvester's criterion on concludes that the matrix A is positive-definite. So its also positive semi-definite.

- **Python script to solve a linear system associated with the above matrix.**

Steps of solving using LU-factorization

$$Ax = b \iff LUx = b \iff Ly = b \text{ with } y = Ux$$

Step 1: Solve $Ly = b$ for y using forward substitution.
Step 2: Solve $Ux = y$ for x using backward substitution

Displaying of matrix A with n= 10

```
import numpy as np

# Generating of the tridiagonal Matrix A
n = 10
A = 2*np.eye(n) - np.diag(np.ones(n-1),1) - np.diag(np.ones(n-1),-1)

print(f"The matrix A)

>>Output
The solutio n of linear system Ax = b with A=
[[ 2. -1.  0.  0.  0.  0.  0.  0.  0.  0.]
 [-1.  2. -1.  0.  0.  0.  0.  0.  0.  0.]
 [ 0. -1.  2. -1.  0.  0.  0.  0.  0.  0.]
 [ 0.  0. -1.  2. -1.  0.  0.  0.  0.  0.]
 [ 0.  0.  0. -1.  2. -1.  0.  0.  0.  0.]
 [ 0.  0.  0.  0. -1.  2. -1.  0.  0.  0.]
 [ 0.  0.  0.  0.  0. -1.  2. -1.  0.  0.]
 [ 0.  0.  0.  0.  0.  0. -1.  2. -1.  0.]
 [ 0.  0.  0.  0.  0.  0.  0. -1.  2. -1.]
 [ 0.  0.  0.  0.  0.  0.  0.  0. -1.  2.]]

```

Using LU factorization

```
import numpy as np
from scipy import linalg

# Generating of the tridiagonal Matrix A
n = 10
A = 2*np.eye(n) - np.diag(np.ones(n-1),1) - np.diag(np.ones(n-1),-1)

# Choice of the vector b
b = np.ones(10)

# LU factprization of A


def LU_Factorization(A):
    n = len(A)
    L = [[0.0] * n for i in range(n)]
    U = [[0.0] * n for i in range(n)]
    for j in range(n):
        L[j][j] = 1.0
        for i in range(j+1):
            s1 = sum(U[k][j] * L[i][k] for k in range(i))
            U[i][j] = A[i][j] - s1
        for i in range(j, n):
            s2 = sum(U[k][j] * L[i][k] for k in range(j))
            L[i][j] = (A[i][j] - s2) / U[j][j]
    return L, U

L,U = LU_Factorization(A)

# Solving the equation Ly = b with y = Ux
y = linalg.solve(L,b)

# Solving the equation Ux = y
x = linalg.solve(U,y)

print(f"The solution of linear system using LU factorization is\n x = {x}")

>>Output
The solution of linear system using LU factorization is
x = [ 5.  9. 12. 14. 15. 15. 14. 12.  9.  5.]
```

0.2 Exercise 2

Consider the following boundary value problem modeling the heat flow in a long pipe:

$$(P)\begin{cases} y''(x) - p(x)y'(x) - q(x)y(x) = r(x), \ x \in [a,b] \\ y(a) = \alpha, \ y(b) = \beta \end{cases}$$

1. **Let use an uniform discretization of the interval [a,b] to derive the linear system corre- sponding to the model problem**

   Let $N+1$ be the numbers of discretization points of the segment $[a,b]$. and $h = \dfrac{b-a}{N}$ the uniform step. Then for all $i \in \{1, \cdots, N-1\}$, $x_i = jh$. $x_0 = a < x_1 < \cdots < x_N = b$. From finite difference methods on obtains:

   $$\begin{cases} y''(x) = \dfrac{y_{i-1} - 2y_i + y_{i+1}}{h^2} \\ y' = \dfrac{y_i - y_{i-1}}{h} \end{cases} \quad \text{where } y_i = y(x_i)$$

   Denote by p_i, q_i, r_i the images $p(x_i), q(x_i), r(x_i)$ of the node x_i.
   So the discretized form of (p) is writen by:

   $$\begin{cases} \dfrac{y_{i-1} - 2y_i + y_{i+1}}{h^2} - p_i\dfrac{y_i - y_{i-1}}{h} - q_i y_i = r_i \\ y_0 = \alpha \\ y_N = \beta \end{cases}$$

   $$\Longleftrightarrow$$

   $$\begin{cases} (1 + hp_i)y_{i-1} - (h^2 q_i + hp_i + 2)y_i + y_{i+1} = h^2 r_i \\ y_0 = \alpha \\ y_N = \beta \end{cases}$$

For $i = 1$ one obtains

$$\begin{aligned} &(1+hp_1)y_0 - (h^2q_1 + hp_1 + 2)y_1 + y_2 = h^2r_1 \\ \iff &-(h^2q_1 + hp_1 + 2)y_1 + y_2 = h^2r_1 - \alpha(1+hp_1) \text{ ⓐ} \end{aligned}$$

;
For $i = N-1$ one obtains

$$\begin{aligned} &(1+hp_{N-1})y_{N-2} - (h^2q_{N-1} + hp_{N-1} + 2)y_{N-1} + y_N = h^2r_{N-1} \\ \iff &(1+hp_{N-1})y_{N-2} - (h^2q_{N-1} + hp_{N-1} + 2)y_{N-1} = h^2r_{N-1} - \beta \text{ ⓑ} \end{aligned}$$

For $2 \le i \le N-2,\ (1+hp_i)y_{i-1} - (h^2q_i + hp_i + 2)y_i + y_{i+1} = h^2r_i$

Then, from ⓐ, ⓑ, ⓐ the linear system corresponding is written by:
$AY = b$ with A equals

$$\begin{bmatrix} -a_{11} & 1 & 0 & 0 & \cdots & 0 \\ (1+hp_2) & -a_{22} & 1 & 0 & \cdots & 0 \\ 0 & (1+hp_3) & -a_{33} & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \cdots & \\ 0 & 0 & 0 & \cdots & (1+hp_{N-1}) & -a_{N-1,N-1} \end{bmatrix}; a_{ii} = (h^2q_i + hp_i + 2)$$

$$; Y = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ \vdots \\ y_{N-1} \end{bmatrix} \text{ and } b = \begin{bmatrix} h^2r_1 - \alpha(1+hp_1) \\ h^2r_2 \\ h^2r^3 \\ \vdots \\ \vdots \\ h^2r_{N-1} - \beta \end{bmatrix}$$

For the following question we will take $p_i = q_i = r_i = 1$ and $\alpha = \beta = 0$, to solve the system. In this case the the matrix A,and the vector b are written by:

$$A = \begin{bmatrix} -a_{11} & 1 & 0 & 0 & \cdots & 0 \\ (1+h) & -a_{22} & 1 & 0 & \cdots & 0 \\ 0 & (1+h) & -a_{33} & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \cdots & \\ 0 & 0 & 0 & \cdots & (1+h) & -a_{N-1,N-1} \end{bmatrix}; a_{ii} = (h^2 + h + 2)$$

and

$$b = \begin{bmatrix} h^2 \\ h^2 \\ h^2 \\ \vdots \\ \vdots \\ h^2 \end{bmatrix}$$

Python code to generate the matrix A and the vector b under the previous conditions.

```python
def Tridiag(n,a1,a2,a3):
    matrix = a2*np.eye(n) - a3*np.diag(np.ones(n-1),1) -
    ↪  a1*np.diag(np.ones(n-1),-1)
    return matrix

# Les bords du domaine
# a = 0
# b = 1
N = 100
###--------------Discretisation en N+1 points ------------###
h = 1/N

###------------Coeficients de la matrice-----------------###

a1, a2, a3 = 1+h, -(2+h+h**2), 1

###------------Vector b-------------------###
b = h**2 * np.ones(N) # second membre
A = Tridiag(N,a1,a2,a3)
```

2. **Let solve the linear system using SVD-decomposition.**
   We want to solve $Ax = b$, by use the SVD of A.

$$A = U\Sigma V^T \iff Ax = U\Sigma V^T x$$

. Then

$$Ax = b \iff U\Sigma V^T x = b$$

To find x, we can solve the system:

$$\Sigma V^T x = U^T b \qquad \text{because } U \text{ is an orthogonal matrix}$$

This simplifies to:

$$\Sigma y = U^T b \quad (\text{where } y = V^T x)$$

Since Σ is a diagonal matrix, this system $\Sigma y = U^T b$ is easy to solve for y.
Finally, we will able to find the solution x by using the orthogonality of V. This solution is given by: $x = Vy$

Python script to do it

```python
def SVD_Solver(A,b):
    # Singular Value Decomposition to solve the system of linear equations
    U, S, Vt = np.linalg.svd(A)
    S = np.diag(S)
    y = np.dot(np.linalg.inv(S), U.T @ b)
    x = Vt.T @ y
    return x
SVD_Solver(A,b)

>>Output

```

```
array([-3.93141953e-05, -2.09745360e-05, -3.78856594e-05, -2.99466516e-05,
↪  -4.06810522e-05, -3.50043272e-05, -4.27678712e-05,
↪  -3.81570669e-05,-4.42021992e-05, -4.02593827e-05, -4.52030206e-05,
↪  -4.17383046e-05, -4.59238916e-05, -4.28253686e-05, -4.64605883e-05,
↪  -4.36537696e-05, -4.68723798e-05, -4.43040550e-05, -4.71966768e-05,
↪  -4.48271366e-05, -4.74577744e-05, -4.52564890e-05, -4.76719378e-05,
↪  -4.56149064e-05, -4.78503825e-05, -4.59183970e-05, -4.80010564e-05,
↪  -4.61785125e-05, -4.81297333e-05, -4.64037885e-05, -4.82407012e-05,
↪  -4.66006649e-05, -4.83372081e-05, -4.67740903e-05, -4.84217573e-05,
↪  -4.69279289e-05, -4.84963088e-05, -4.70652421e-05, -4.85624184e-05,
↪  -4.71884865e-05, -4.86213365e-05, -4.72996561e-05, -4.86740796e-05,
↪  -4.74003860e-05, -4.87214822e-05, -4.74920303e-05, -4.87642359e-05,
↪  -4.75757193e-05, -4.88029187e-05, -4.76524042e-05, -4.88380176e-05,
↪  -4.77228914e-05, -4.88699453e-05, -4.77878687e-05, -4.88990543e-05,
↪  -4.78479266e-05, -4.89256475e-05, -4.79035748e-05, -4.89499863e-05,
↪  -4.79552555e-05, -4.89722978e-05, -4.80033545e-05, -4.89927799e-05,
↪  -4.80482097e-05, -4.90116061e-05, -4.80901189e-05, -4.90289284e-05,
↪  -4.81293457e-05, -4.90448801e-05, -4.81661261e-05, -4.90595755e-05,
↪  -4.82006756e-05, -4.90731074e-05, -4.82332019e-05, -4.90855338e-05,
↪  -4.82639312e-05, -4.90968413e-05, -4.82931728e-05, -4.91068447e-05,
↪  -4.83214871e-05, -4.91149188e-05, -4.83501256e-05, -4.91192852e-05,
↪  -4.83821948e-05, -4.91151207e-05, -4.84257385e-05, -4.90895426e-05,
↪  -4.85019042e-05, -4.90083167e-05, -4.86665828e-05, -4.87806321e-05,
↪  -4.90677526e-05, -4.81657547e-05, -5.00974341e-05, -4.65256752e-05,
↪  -5.27943301e-05, -4.21696992e-05, -5.99106806e-05, -3.06178782e-05,
↪  -7.87389310e-05])
```

3. **Let solve the linear system using the Gaussian elimination method.**
We want to solve $Ax = b$. The Gauss elimination method consists of eliminating all terms below the diagonal of the matrix A. Basically, a sequence of operations is performed on a matrix of coefficients. The operations involved are:

- Swapping two rows
- Multiplying a row by a nonzero number
- Adding a multiple of one row to another row

These operations are performed until the lower left-hand corner of the matrix is filled with zeros, as much as possible.The elementary operations must be performed on both the rows of matrix A and the vector b

The obtained triangular matrix is denoted by U. The operations performed to obtain U are equivalent to multiplying the original system by a sequence of invertible matrices. In fact, we have:

$$U = T_p \cdots T_3 T_2 T_1 A$$

where the matrices T_i correspond to the various operations performed on the rows of the matrix. Thus,
$Ax = b \iff T_p \cdots T_3 T_2 T_1 Ax = T_p \cdots T_3 T_2 T_1 b \iff Ux = B$ with $B = T_p \cdots T_3 T_2 T_1 b$.
Since U is triangular matrix, the system is easy to solve for x by using backward substitution.

Python script to do it

```python
def GAUSS_Solver(A, b):
    n = len(A)
## --------- Elementary operations -----------##
    for k in range(n-1):
        for i in range(k+1, n):
            A[i,k] = A[i,k]/A[k,k]
            for j in range(k+1, n):
                A[i,j] = A[i,j] - A[i,k]*A[k,j]
            b[i] = b[i] - A[i,k]*b[k]
    x = np.zeros(shape=(n, 1))
##----------Backward substitution.---------##
    x[n-1] = b[n-1]/A[n-1,n-1]
    for i in range(n-2, -1, -1):
        s = 0
        for j in range(i+1, n):
            s = s + A[i,j]*x[j]
        x[i] = (b[i] - s)/A[i,i]
    return x

GAUSS_Solver(A,b)
```

4. **Let solve the linear system using QR-factorization.**
We want to solve $Ax = b$, by use the QR of A.

$$A = QR \iff Ax = UQRx$$

. Then

$$Ax = b \iff QRx = b$$

To find x, we can solve the system:

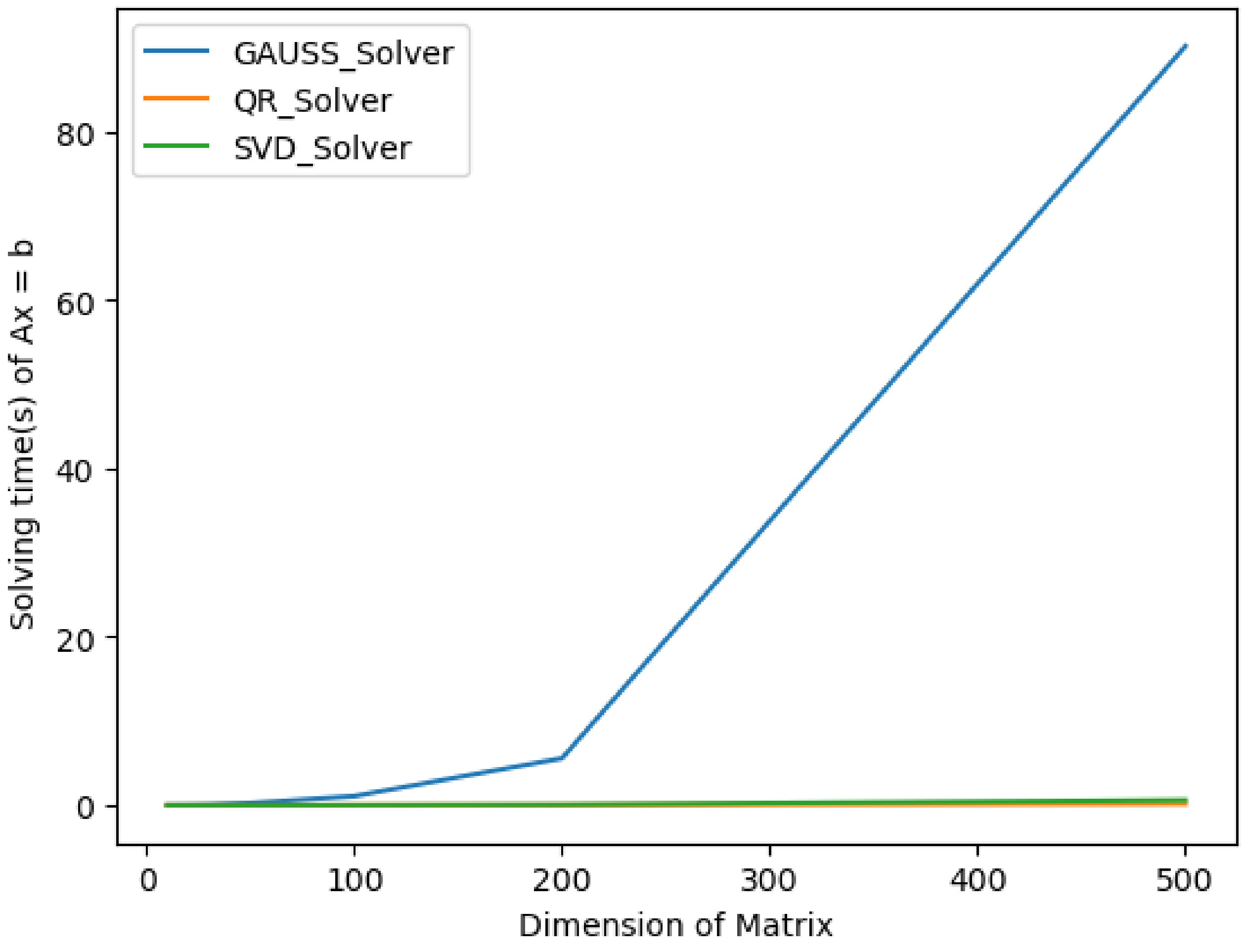
$$Rx = Q^T b \qquad \text{because } Q \text{ is an orthogonal matrix}$$

Since R is trapezoïdal matrix (upper triangular in our case because A is square matrix) this system $Rx = Q^T b$ is easy to solve for x. Indeed, once we compute $Q^T b$ (suppose its equals B), then we just going to solve $Rx = B$ for x by using backward substitution.
Python script to do it

```python
def QR_Solver(A, b):
    n = len(A)
    Q, R = np.linalg.qr(A)
    y = np.dot(Q.T, b)
    x = np.zeros(n)
    for i in range(n-1, -1, -1):
        x[i] = (y[i] - np.dot(R[i,i+1:], x[i+1:])) / R[i,i]
    return x
```

5. **Let compare the three methods.**
To compare the three methods we will base on the time of solving with respect each method by increasing the dimension of the matrix A.

```python
def timers(Ns,Solver):
    """
    Ns: it is the list of dimensions of the matrix A
    Solver: Should be one of the previous
    """
    times = []
    for N in Ns:
        A = Tridiag(N,a1,a2,a3)
        b = h**2 * np.ones(N)
        start = time.time()
        Solver(A,b)
        end = time.time()
        times.append(end-start)
    return times

Dims = [10,20,30,50,100,200,300,500]

times_Gauss = timers(Dims,GAUSS_Solver)
times_QR = timers(Dims,QR_Solver)
times_SVD = timers(Dims,SVD_Solver)
plt.figure()
plt.plot(Dims,times_Gauss, label = "GAUSS_Solver")
plt.plot(Dims,times_QR, label = "QR_Solver")
plt.plot(Dims, times_SVD, label = "SVD_Solver")
plt.xlabel("Dimension of Matrix")
plt.ylabel("Solving time of Ax = b")
plt.legend()


```



0.3 Exercise 3

Let solve the problem of fitting a polynomial $p(x) = \sum_{k=1}^{d} c_k x^{k-1}$ to the data points $(x_i, y_i), i = 1, 2, \cdots, m$ where $y_i = f(x_i)$ for f defined by:$f : [0, 1] \to \mathbb{R}; x \mapsto \cos(\frac{\pi x}{7}) + \frac{\pi x}{7} + 1$.
For each observation (x_i, y_i) one has: $p(x_i) = \sum_{k=1}^{d} c_k x_i^{k-1}$.
So the goal is to find $c_k, k = 1, 2, \cdots, d$, such that $\forall\ 1 \le i \le m,\ p(x_i) \approx y_i$.
Its comes to solve system $PC = Y$ where

$$P = \begin{bmatrix} 1 & x_1 & x_1^2 & x_1^3 & \cdots & x_1^{d-1} \\ 1 & x_2 & x_2^2 & x_2^3 & \cdots & x_2^{d-1} \\ 1 & x_3 & x_3^2 & x_3^3 & \cdots & x_3^{d-1} \\ \vdots & \vdots & \vdots & \vdots & \cdots & \vdots \\ 1 & x_m & x_m^2 & x_m^3 & \cdots & x_m^{d-1} \end{bmatrix}, C = \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ \vdots \\ c_d \end{bmatrix} \text{ and } Y = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_m \end{bmatrix}$$

Python script to generate the matrix P **and the observations** (x_i, y_i)

```
# Numbers d'observation N
N = 10# ou N = 20 points

# Lenght of vecteur of  unknows C
d = 5 # ou d= 10

# Generating of x_i
X = np.linspace(-10,10,N)

# Function to generate y_i
f = lambda x: np.cos(np.pi*x/7)+x/7 +1

# Generating of y_i
Y = f(X)

# Function wich generate the matrix P
def P_matrix(X,d):
    M = np.zeros((len(X),d))
    for k in range(d):
        M[:,k] = np.round(X**k,4)
    return M
P = P_matrix(X,d)
print(f"Matrix P = \n {P}\n and \n the vector Y = \n {Y}")

>>Output

Matrix P =
 [[ 1.0000000e+00 -1.0000000e+01  1.0000000e+02 -1.0000000e+03 1.0000000e+04]
 [ 1.0000000e+00 -7.7778000e+00  6.0493800e+01 -4.7050750e+02 3.6595031e+03]
 [ 1.0000000e+00 -5.5556000e+00  3.0864200e+01 -1.7146780e+02  9.5259870e+02]
 [ 1.0000000e+00 -3.3333000e+00  1.1111100e+01 -3.7037000e+01  1.2345680e+02]
 [ 1.0000000e+00 -1.1111000e+00  1.2346000e+00 -1.3717000e+00 1.5242000e+00]
 [ 1.0000000e+00  1.1111000e+00  1.2346000e+00  1.3717000e+00  1.5242000e+00]
 [ 1.0000000e+00  3.3333000e+00  1.1111100e+01  3.7037000e+01  1.2345680e+02]
 [ 1.0000000e+00  5.5556000e+00  3.0864200e+01  1.7146780e+02  9.5259870e+02]
 [ 1.0000000e+00  7.7778000e+00  6.0493800e+01  4.7050750e+02  3.6595031e+03]
 [ 1.0000000e+00  1.0000000e+01  1.0000000e+02  1.0000000e+03  1.0000000e+04]]
 and
```

```
the vector Y =
[-0.65109236 -1.05080373 -0.5907833   0.59853962  1.71949141  2.03695173
↪  1.55092057  0.99651829  1.17141849  2.20605049]
```

1. **Solving by using Normal equations.**

$$\begin{aligned} PC = Y &\iff P^T(PC) = P^TY \\ &\iff (P^TP)C = P^TY \end{aligned}$$

Since the matrix (P^TP) is invertible, it is so easy to solve the system $PC = Y$ for C by inverting the matrix P^TP. The solution C is given by: $C = (P^TP)^{-1}P^TY = P^{\dagger}Y$. with $P^{\dagger} = (P^TP)^{-1}P^T$ the pseudo-inverse of P

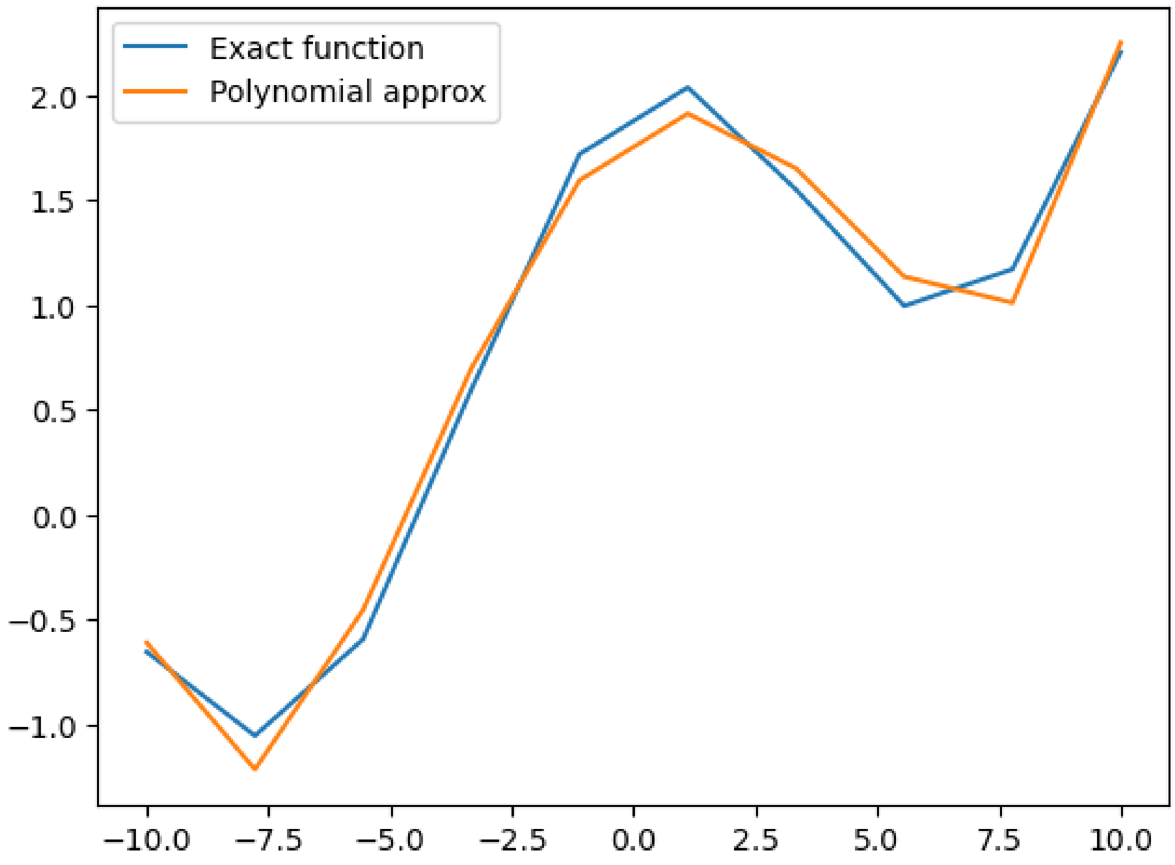
Python script to apply normal equation's method

```python
## ----P^TP
A = np.dot(P.T,P)
## -----Pseudo inverse of P ------
P_inv =  np.dot(np.linalg.inv(A),P.T)
## ------- Solution C ------------
C = np.dot(P_inv,Y)
print(f"Pour d = 5, C = {C}")
>>Output
C = [ 1.83378490e+00  1.42856679e-01 -6.53858466e-02  3.54037142e-09
 ↪  5.52620903e-04]
```

Approximatly, $C = (1.83378490e + 00, 1.42856679e - 01, 6.53858466e - 02, 3.54037142e - 09, 5.52620903e - 04)$ for $d = 5$.

By using the obtained coeficient C for $d = 5$, we are going to plot both the polynom we fit and the exact function f.

```python
##--------Plotting
# d=5
Y_fit = np.dot(P,C)
plt.plot(X,Y, label= "Exact function")
plt.plot(X,Y_fit, label = "Polynomial approx")
plt.legend()
```



2. **Solving by using QR decomposition** Suppose that $P = QR$.Then

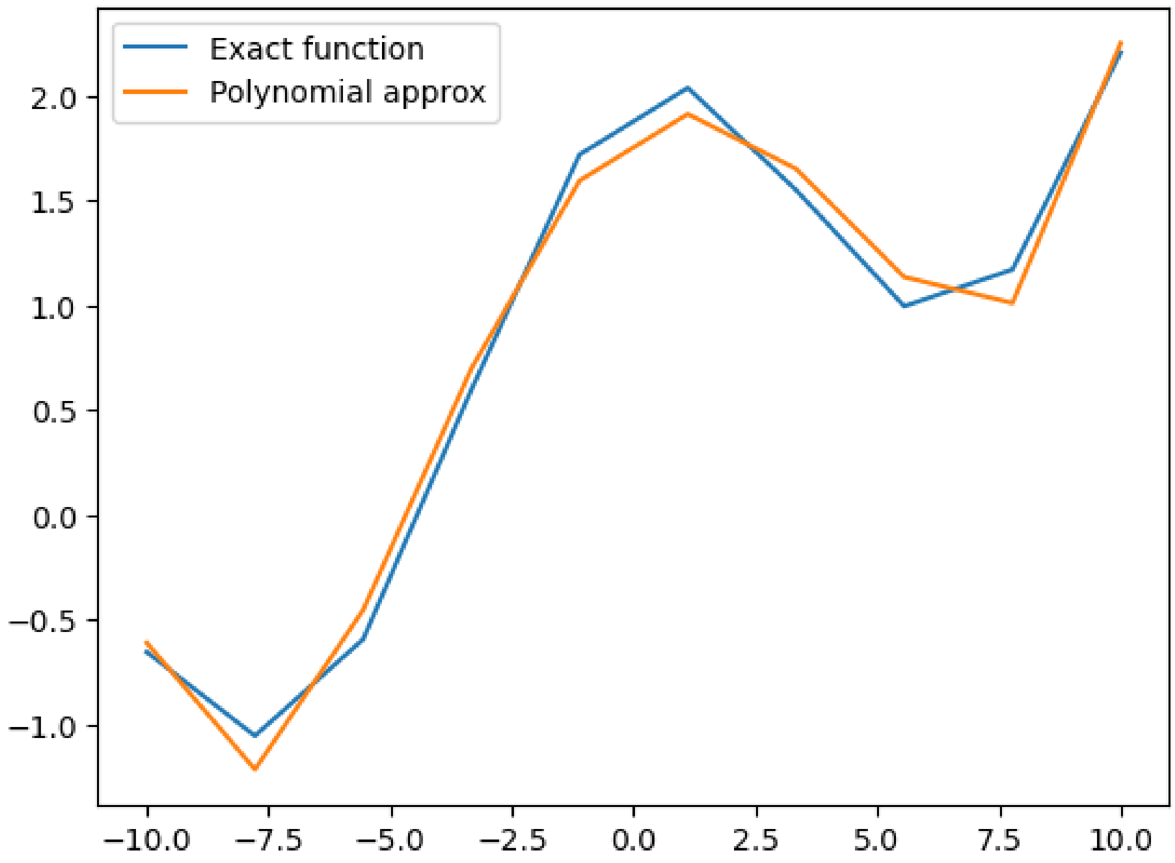
$$\begin{aligned} PC = Y &\iff P^T PC = P^T Y \\ &\iff (QR)^T(QR)C = P^T Y \\ &\iff R^T Q^T QRC = P^T Y \\ &\iff R^T RC = P^T Y \text{ because } Q \text{ is an orthogonal matrix} \end{aligned}$$

Since the matrix $(R^T R)$ is invertible, it is so easy to solve the system $PC = Y$ for C by inverting the matrix $R^T R$. The solution C is given by: $C = (R^T R)^{-1} P^T Y$.

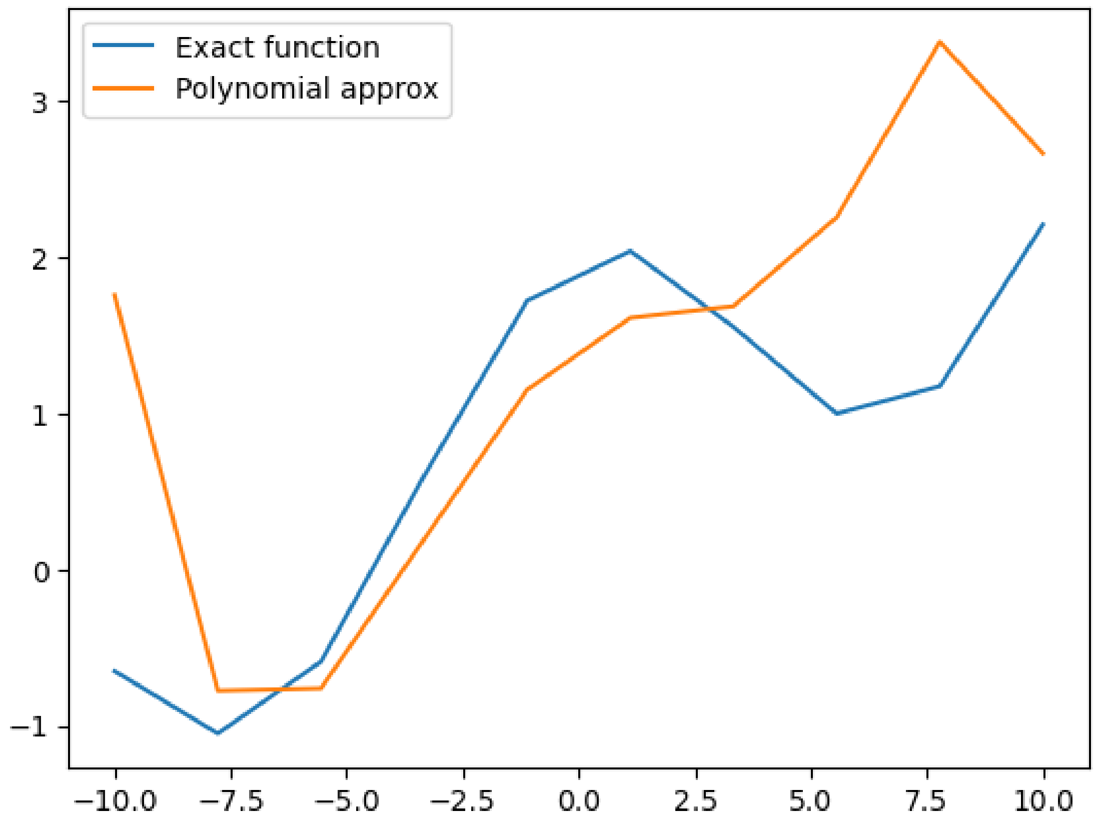
Python script to apply QR decomposition

```python
    ## ----QR
Q,R = np.linalg.qr(P)
## -----R^T*R ------
RtR =  np.dot(R.T,R)
##------invert of RtR times P^T
A = np.dot(np.linalg.inv(RtR),P.T)
## ------- Solution C ------------
C = np.dot(A,Y)
print(f"Pour d=5, C = {C}")
>>Output
C = [ 1.83378490e+00  1.42856679e-01 -6.53858466e-02  3.54037142e-09
 ↪  5.52620903e-04]
```

Approximatly $C = (1.83378490e+00, 1.42856679e-01, 6.53858466e-02, 3.54037142e-09, 5.52620903e-04)$ for $d = 5$

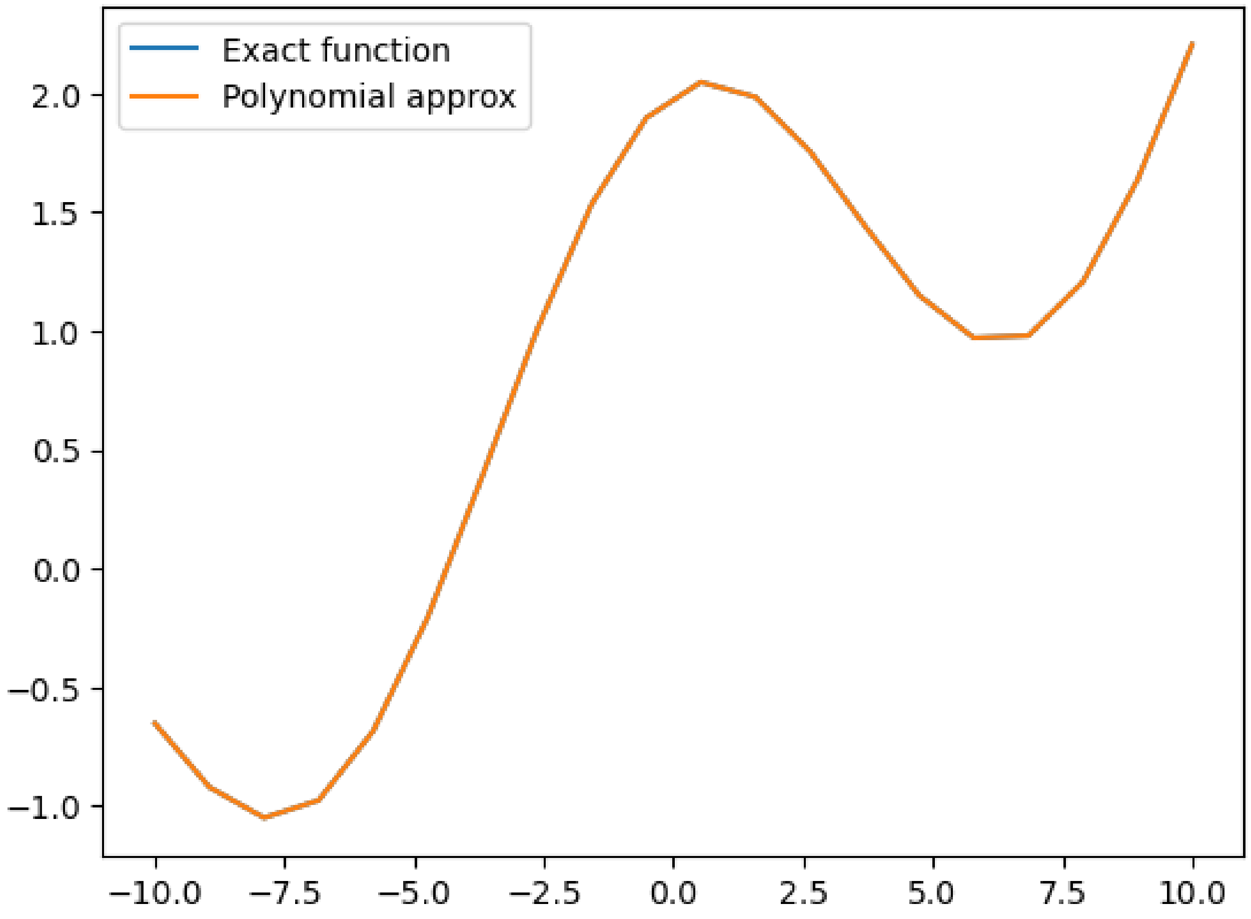


For $n = 10$ and $d = 15$ one obtaines a fitting as described by the following plot.



This behavior can be justify by the fact that the system is underdetermined because there is more unknowns than equations. In this case, the QR-factorization will not be interesting. Obviously the solution will be very biased.

For $n = 20$ and $d = 15$ one obtains a fitting as described by the following plot.



We become more accurate by increasing the numbers of points of interpolation.

0.4 Exercise 4

1. **Let explain how Singular Value Decomposition (SVD) can be applied to compress color images by discussing the process in the context of an RGB image.**

   Applying Singular Value Decomposition (SVD) to compress color images involves breaking down the image data into its singular values and corresponding matrices. In the context of an RGB image, we need to consider that an RGB image is composed of three color channels: red, green, and blue. The process in this case follows the following steps:

   •*Step 1:* **Image Representation:** An RGB image is represented as a 3D matrix, where each layer corresponds to one of the color channels (red, green, and blue). Let's denote this matrix as M;

   •*Step 2:* **SVD Decomposition:** Perform SVD on each color channel separately. For each channel, the SVD decomposition yields three matrices:U, Σ and V^T. These matrices represent the left singular vectors, singular values, and right singular vectors of M, respectively;

•*Step 3:* **Rank Reduction:** After obtaining the SVD matrices for each color channel, we can apply rank reduction by keeping only the dominant singular values and truncating the corresponding columns in matrices U and V^T. This effectively reduces the size of the matrices and hence compresses the image;

•*Step 4:* **Reconstruction:** To reconstruct the compressed image, we use the truncated matrices U_k, Σ_k and V_k^T,perform matrix multiplication. The reconstructed matrices for each color channel are combined to form the compressed color image;

•*Step 4:* **Compression ratio:** The compression ratio is determined by the number of singular values retained.Here's the formula for computing the compression ratio;

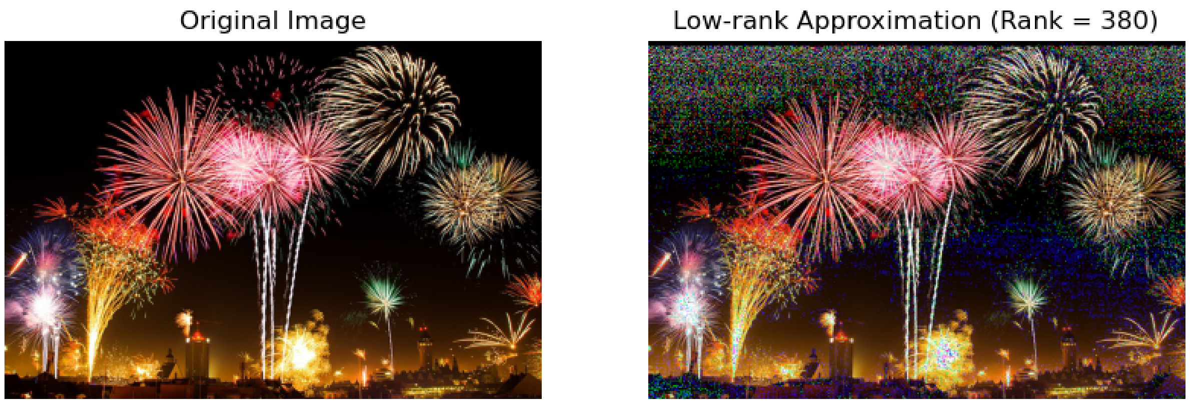
$$Compression_{Ratio} = \frac{Originalimagesize}{Compressedimagesize}$$

where Image size = Numbers of Pixels × Bit per Pixel.

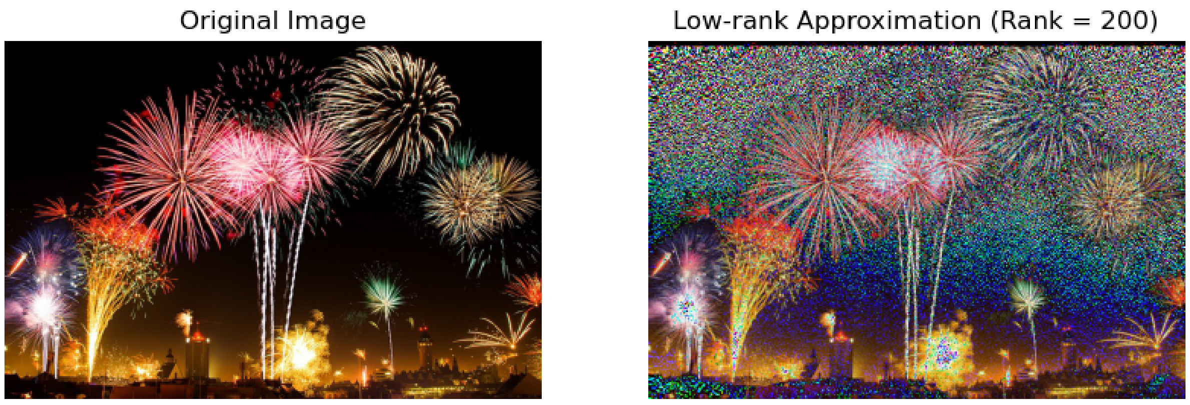
2. **Implementation of an SVD-based compression algorithm in Python**

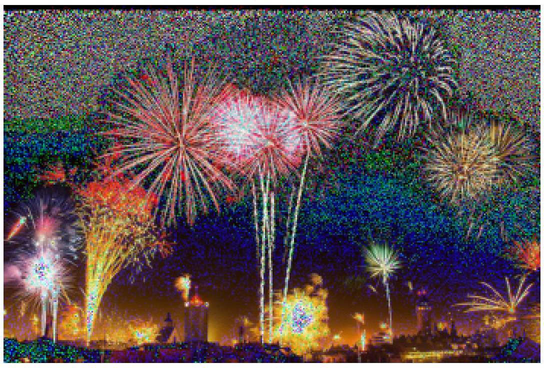
```python
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image

# Load the image
image_path = "OrigImage.jpg"  # Replace "image.jpg" with the path to your
↪  image
image = Image.open(image_path)

# Convert the image to a numpy array
image_array = np.array(image)
U,S,Vt = np.linalg.svd(image_array)
# Separate color channels
red_channel = image_array[:, :, 0]
green_channel = image_array[:, :, 1]
blue_channel = image_array[:, :, 2]

# Perform Singular Value Decomposition (SVD) for each color channel
U_red, S_red, V_red = np.linalg.svd(red_channel)
U_green, S_green, V_green = np.linalg.svd(green_channel)
U_blue, S_blue, V_blue = np.linalg.svd(blue_channel)

# Set the rank for low-rank approximation
rank = 380 # Adjust the rank as needed for your approximation

# Reconstruct the color channels with low-rank approximation
red_approximation = np.dot(U_red[:, :rank], np.dot(np.diag(S_red[:rank]),
↪  V_red[:rank, :]))
green_approximation = np.dot(U_green[:, :rank],
↪  np.dot(np.diag(S_green[:rank]), V_green[:rank, :]))
blue_approximation = np.dot(U_blue[:, :rank],
↪  np.dot(np.diag(S_blue[:rank]), V_blue[:rank, :]))

# Combine the color channels
```

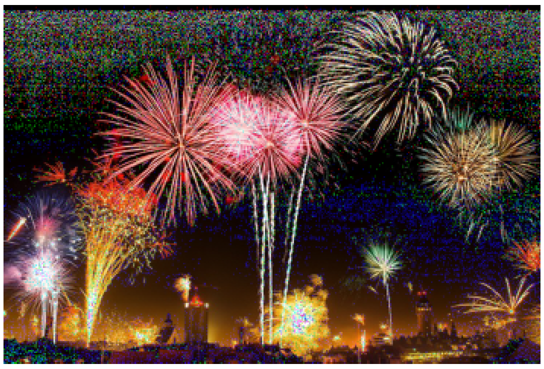
```
approximation = np.stack((red_approximation, green_approximation,
↪  blue_approximation), axis=-1)

# Display the original and the approximation
plt.figure(figsize=(10, 5))

plt.subplot(1, 2, 1)
plt.imshow(image_array)
plt.title('Original Image')
plt.axis('off')

plt.subplot(1, 2, 2)
plt.imshow(np.uint8(approximation))
plt.title(f'Low-rank Approximation (Rank = {rank})')
plt.axis('off')
plt.show()
>>Output
```

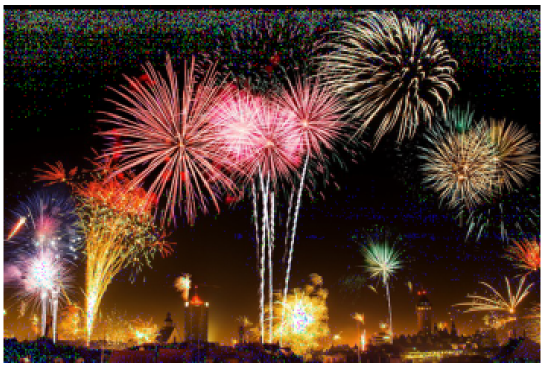


3. **Analyzing the effect of varying the number of singular values (k) on the compression ratio and image quality**







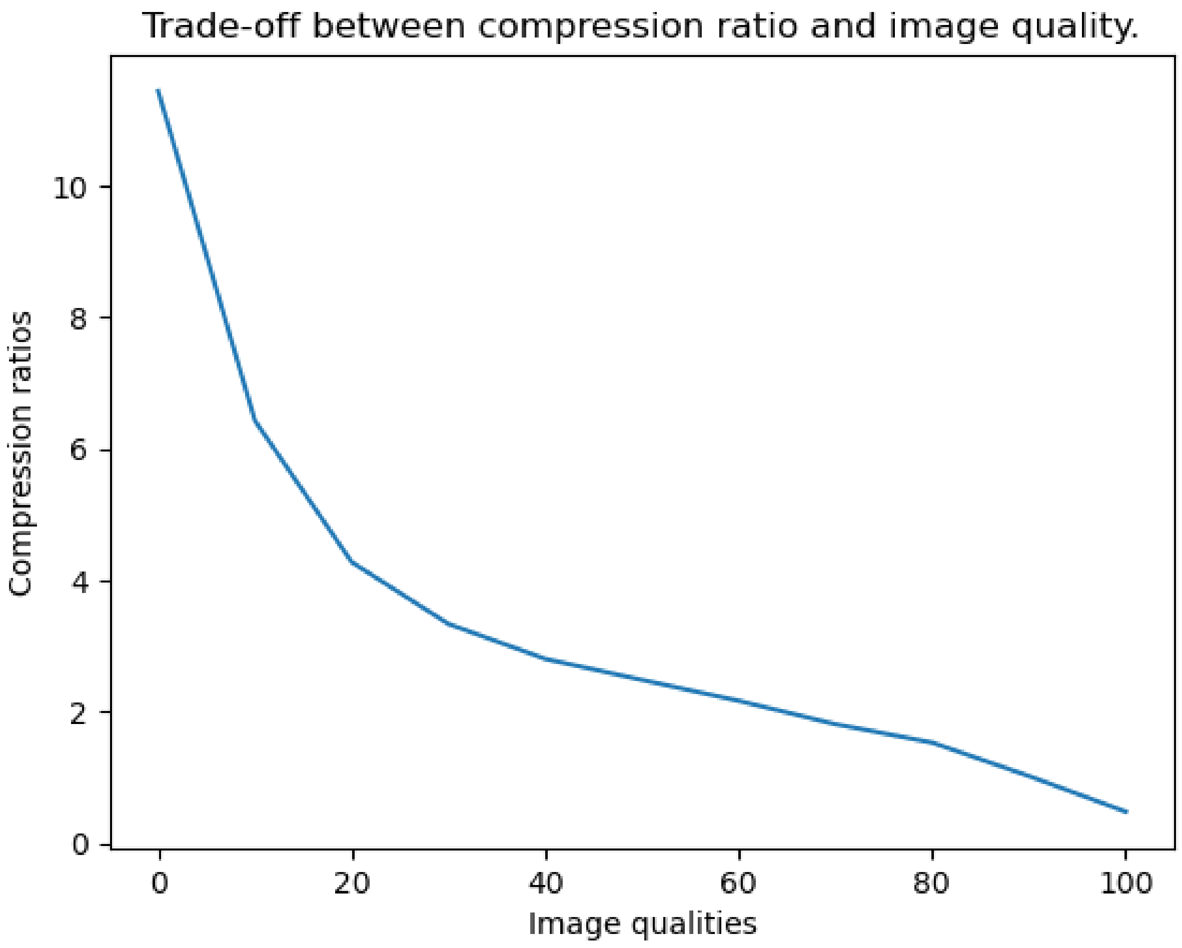


By analysing the previous display, one remarks that when we increase the rank of matrix(numbers of singular values) the quality of compressed image becomes so close to the quality of original image. The opposite happens when we decrease the rank of M.

Computing of the ratio of an image

```python
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
import os

def compression_ratio(original_file, compressed_file):
    original_size = os.path.getsize(original_file)
    compressed_size = os.path.getsize(compressed_file)
    ratio = original_size / compressed_size
    return ratio
```

```

# Load original image
original_image_path = "OrigImage.jpg"
original_image = Image.open(original_image_path)

# Qualities of Image
Qualities = [0,10,20,30,40,50,60,70,80,90,100]
# The corresponding ratios
Ratios = []
for compress_quality in Qualities:

    original_image.save("compressed_image.jpg", quality=compress_quality)

    compressed_image_path = "compressed_image.jpg"
    ratio = compression_ratio(original_image_path, compressed_image_path)
    Ratios.append(ratio)

print(f"Compression ratio: {Ratios} with respect to the qualities
 ↪  {Qualities}")
plt.plot(Qualities,Ratios, label = "")
plt.xlabel("Image qualities")
plt.ylabel("CCompression ratios")
plt.title("Trade-off between compression ratio and image quality.")
>>Output
Compression ratio: [11.439700243987453, 6.425942241801272,
 ↪  4.274059122281547, 3.332453357025003, 2.8018183370326106,
 ↪  2.4818889897156686, 2.1637631236299506, 1.8078437852873943,
 ↪  1.5287569141193595, 1.019254359404357, 0.47629966367835025] with
 ↪  respect to qualities
[0,10,20,30,40,50,60,70,80,90,100]
```



There's a trade-off between image quality and compression ratio. Higher compression ratios result in more loss of image information and potentially lower image quality. Choosing an appropriate number of singular values to retain is crucial to balancing compression and maintaining image fidelity. A compression ratio of 2, for example, means the compressed image is half the size of the original image.

0.5 Exercise 5

Let consider the biharmonic equation

$$\begin{cases} \Delta^2 u(s,t) := \Delta(\Delta u(s,t)) = f(s,t) \ (s,t) \in \Omega \\ u(s,t) = 0 \ \Delta u(s,t) = 0 \ (s,t) \in \partial\Omega \end{cases} \qquad (3)$$

where Ω is an open unit square and $\Delta^2 u = u_{xxxx} + 2u_{xxyy} + u_{yyyy}$

a) For $v = -\Delta u$, let shows that (3) can be written as a system $\begin{cases} -\Delta v(s,t) = f(s,t) \ (s,t) \in \Omega \\ -\Delta u(s,t) = v(s,t) \ (s,t) \in \Omega \\ u(s,t) = v(s,t) = 0 \ (s,t) \in \partial\Omega \end{cases}$

From assumption one has $v = -\Delta u$.

$$\begin{aligned} v = -\Delta u &\iff v(s,t) = -\Delta u(s,t) \ \forall (s,t) \in supp(v) = supp(-\Delta u) \\ &\iff -v(s,t) = \Delta u(s,t) \ \forall (s,t) \in \Omega \\ &\iff \Delta(-v(s,t)) = \Delta(\Delta u(s,t)) \\ &\iff \Delta(-v(s,t)) = \Delta^2 u(s,t) \text{ by definition} \\ &\iff -\Delta v(s,t) = \Delta^2 u(s,t) \ \text{ since } \Delta \text{ is linear(Laplacian operator) } \textcircled{1} \end{aligned}$$

$v = -\Delta u \iff v(s,t) = -\Delta u(s,t)$, particularly $v(s,t) = \Delta u(s,t)$ on $\partial\Omega$. But by assumption, $\Delta u(s,t) = 0$ $(s,t) \in \partial\Omega$, then $v(s,t) = 0$ $(s,t) \in \partial\Omega$ ②
Furthermore, $u(s,t) = 0$ (assumption) and $(-\Delta u(s,t) = v(s,t))$ is already contained in the previous equivalence: It is a result of equality of two function ③.
From ①, ②, ③ one deduces that (3) can be written as

$$\begin{cases} -\Delta v(s,t) = f(s,t) \ (s,t) \in \Omega \\ -\Delta u(s,t) = v(s,t) \ (s,t) \in \Omega \\ u(s,t) = v(s,t) = 0 \ (s,t) \in \partial\Omega \end{cases}$$

b) From the following questions one obtains an equivalent expression of (3) as follows:

$$\begin{cases} -\Delta v(s,t) = f(s,t) \ (s,t) \in \Omega \\ -\Delta u(s,t) = v(s,t) \ (s,t) \in \Omega \\ u(s,t) = v(s,t) = 0 \ (s,t) \in \partial\Omega \end{cases} \qquad (4)$$

(4) is two Poisson's equation in $(2D)$ which can be decoupled by:

$$\begin{cases} -\Delta v(s,t) = f(s,t) \ (s,t) \in \Omega \\ v(s,t) = 0 \ (s,t) \in \partial\Omega \end{cases} \qquad (\star)$$

and

$$\begin{cases} -\Delta u(s,t) = v(s,t) \ (s,t) \in \Omega \\ u(s,t) = 0 \ (s,t) \in \partial\Omega \end{cases} \qquad (\star\star)$$

Let consider $(\star)$;
The discretized domain is $\overline{\Omega}_h = \{(jh, kh) : j,k = 0,1,\dots,m+1\}$ (5).
The points Ω_h represent the interior points of the domain, while $\overline{\Omega} \setminus \Omega_h$ are the points on the boundary of Ω. Approximating the second partial derivatives using the finite difference method provides

$$\frac{\partial^2 u(jh,kh)}{\partial s^2} \approx \frac{v_{j-1,k} - 2v_{j,k} + v_{j+1,k}}{h^2},$$

$$\frac{\partial^2 u(jh,kh)}{\partial t^2} \approx \frac{v_{j,k-1} - 2v_{j,k} + v_{j,k+1}}{h^2}.$$

Inserting this into (*), we obtain the discrete Poisson problem

$$\begin{cases} -\Delta_h v_{j,k} = f_{j,k}, & (jh,kh) \in \Omega_h \\ v_{j,k} = 0, & (jh,kh) \in \partial\Omega_h \end{cases} \quad (5)$$

for $j,k = 1,\dots,m$ and $f_{j,k} = f(jh,kh)$. where discrete Poisson operator is given by

$$\Big\{ -\Delta_h v_{j,k} = -\tfrac{v_{j-1,k} - 2v_{j,k} + v_{j+1,k}}{h^2} - \tfrac{v_{j,k-1} - 2v_{j,k} + v_{j,k+1}}{h^2}, \quad \text{for } (jh,kh) \text{ in } \Omega_h,$$

The linear system (5) has $n := m^2$ equations and n unknowns. The values at the boundary points are known, which are zeros, and the unknowns are the n numbers $v_{j,k}$ at the interior points. The corresponding linear system can be written in matrix form as:

$TV + VT = h^2F$ (6), with $h = \frac{1}{m+1}$, where $T = \text{tridiag}(-1, 2, -1) \in \mathbb{R}^{m\times m}$ is the second derivative matrix;

$$V := \begin{pmatrix} v_{1,1} & \cdots & v_{1,m} \\ \vdots & \ddots & \vdots \\ v_{m,1} & \cdots & v_{m,m} \end{pmatrix},$$

and

$$F := \begin{pmatrix} f_{1,1} & \cdots & f_{1,m} \\ \vdots & \ddots & \vdots \\ f_{m,1} & \cdots & f_{m,m} \end{pmatrix}$$

Similarly, by considering $(\star\star)$ one shows that $TU + UT = h^2V$ (7) where

$$U := \begin{pmatrix} u_{1,1} & \cdots & u_{1,m} \\ \vdots & \ddots & \vdots \\ u_{m,1} & \cdots & u_{m,m} \end{pmatrix},$$

Hence the results.
(6) can be write in standard form $Ax = b$ by applying the vectorization operation on the matrices T, V and F as follows:

$$-x_{i-m} - x_{i-1} + 4x_i - x_{i+1} - x_{i+m} = b_i,$$

where $x_i = v_{j,k}$ and $b_i = h^2 f_{j,k}$, with $i = j + m(k-1)$.
Then,

$$TV + VT = h^2F \iff Ax = b, \quad A \in \mathbb{R}^{n\times n}, \quad b \in \mathbb{R}^n, \quad n = m^2.$$

where $A = T \bigotimes I + I \bigotimes T$, $I \in \mathbb{R}^{m\times m}$.
Since our matrix A is positive definite and symetric, we can solve the system $Ax = b$ by using Cholesky decomposition as follows:

$$Ax = b \iff CC^Tx = b \iff Cy = b \text{ with } y = C^Tx$$

where C is lower-triangular matrix

$$\text{Step 1: Solve } Cy = b \text{ for } y \text{ using forward substitution.}$$
$$\text{Step 2: Solve } C^Tx = y \text{ for } x \text{ using backward substitution}$$

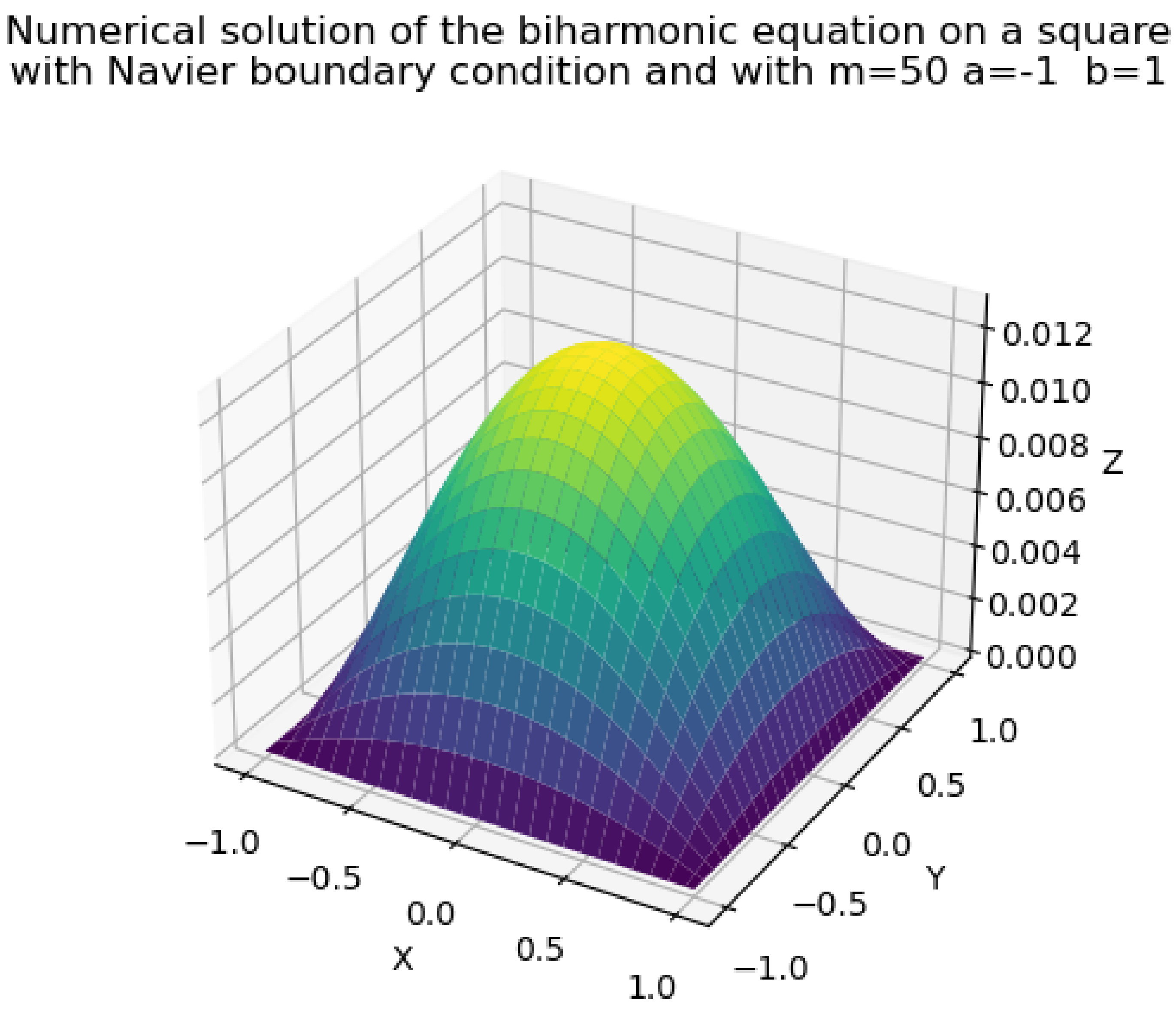
Once we get V by reshaping the obtained solution x, we will do the same operation to solve the second system$(**)$. In this case the second member of the equation is h^2V. Here,

$$\begin{cases} A = T \bigotimes I + I \bigotimes T \\ b_i = h^2 v_{j,k} & \text{with } i = j + m(k-1) \\ x_i = u_{j,k} \end{cases}$$

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



# Function which vectorizes the matrix

Vect = lambda Mat: Mat.flatten('F')

```

```python
# Example of second members of the equation Ax = b

f = lambda x,y:np.sin(x**2 + y**2)*np.exp(-x**2 - y**2)


# Function wich returns a tridiagonal matrix.

def Tridiag(n,a1,a2,a3):
    matrix = a2*np.eye(n) + a3*np.diag(np.ones(n-1),1) +
    ↪  a1*np.diag(np.ones(n-1),-1)
    return matrix


def solve_biharmonic_equation(f,l,L,m):
    """
    This function solve the biharmonic equation on a square
    with Navier boundary condition.
    It takes as parameters
    * f: the second member of the equation
    *m the number of interior points
    * l and L: the bounds

    It returns a tuples (X, fullF, fullU) where
    * X is an 1D array representing the discrete values
    * fullF is a 2d array that represents the discrete values of the second
    ↪  member f, evaluated in X
    * fullU is a 2D array that represents the solution of the equation
    """
    assert l<L
    assert m>0

    # Domain discretization
    X,h=np.linspace(l,L,m+2,retstep=True)

    Xp=X[1:m+1] # The interior points of the domain [a,b]
    Yp=Xp.copy()

    # Discrete values of the second member
    fullF=np.zeros((m+1,m+1))
    for i in range(m+1):
        for j in range(m+1):
            fullF[i,j]=f(X[i],X[j])

    # Extract just he interior values of the second member
    F=fullF[1:m+1,1:m+1]

    # Initialize the test matrix T

    T = Tridiag(m,-1,2,-1)

    # Computing the poisson matrix A
    I=np.eye(m)
    A=np.kron(I,T)+np.kron(T,I)

    # Vectorizing the matrix h²F to obtain b the second member of the standard
     ↪  form Ax = b
    b=Vect(np.power(h,2)*F)

    # Perform the Colesky decomposition of A to get C
    C=np.linalg.cholesky(A)


    # Solve the two triangular systems and get y
    yp=scipy.linalg.solve_triangular(C,b,lower=True)
    y=scipy.linalg.solve_triangular(C.T,yp,lower=False)

    # Reshape y to get the matrix V
    V=y.reshape(m,m)

    # Once we get V we will do the same operation to solve the second system
    # in this case the second member of the equation is h²V

    # Vectorize h²V to get b
    c=Vect(np.power(h,2)*V)

    # Solve the two triangular system and get x
    xp=scipy.linalg.solve_triangular(C,c,lower=True)
    x=scipy.linalg.solve_triangular(C.T,xp,lower=False)

    # Reshape x to get the matrix U containing only the interior discrete
     ↪  solutions
    U=x.reshape(m,m)

    # Add the solution on the bounds to get the whole solution
    fullU=np.zeros((m+2,m+2))
    fullU[1:m+1,1:m+1]= U.copy()

    return X, fullF, fullU




# Number of interior nodes
m=50 # So N=m+2
a=-1
b=1


X,F,U=solve_biharmonic_equation(f,a,b,m)
print(f'The discrete points are X:\n {X}')
print(f'The second member used is F:\n {F}')
print(f'The solution of the biharmonic equation is U:\n {U}')




# Plot the solution in 3D

```

```
x, y = np.meshgrid(X, X)
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(x, y, U, cmap='viridis')


# Set labels and title
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')
ax.set_title('Numerical solution of the biharmonic equation on a square\n'+
f'with Navier boundary condition and with m={m} a={a}  b={b}')

# Show the plot
plt.show()

>>Output

The discrete points are X:
 [0.         0.02439024 0.04878049 0.07317073 0.09756098 0.12195122
 0.14634146 0.17073171 0.19512195 0.2195122  0.24390244 0.26829268
 0.29268293 0.31707317 0.34146341 0.36585366 0.3902439  0.41463415
 0.43902439 0.46341463 0.48780488 0.51219512 0.53658537 0.56097561
 0.58536585 0.6097561  0.63414634 0.65853659 0.68292683 0.70731707
 0.73170732 0.75609756 0.7804878  0.80487805 0.82926829 0.85365854
 0.87804878 0.90243902 0.92682927 0.95121951 0.97560976 1.        ]
The second member used is F:
 [[0.         0.00059453 0.00237388 ... 0.31815429 0.3144178  0.        ]
 [0.00059453 0.00118835 0.00296558 ... 0.31811365 0.31436392 0.        ]
 [0.00237388 0.00296558 0.00473646 ... 0.31799066 0.31420134 0.        ]
 ...
 [0.31815429 0.31811365 0.31799066 ... 0.15906618 0.14986012 0.        ]
 [0.3144178  0.31436392 0.31420134 ... 0.14986012 0.14084836 0.        ]
 [0.         0.         0.         ... 0.         0.         0.        ]]
The solution of the biharmonic equation is U:
 [[ 0.00000000e+00  0.00000000e+00  0.00000000e+00 ...  0.00000000e+00
 ↪   0.00000000e+00  0.00000000e+00]
 [ 0.00000000e+00 -5.13701819e-12  2.93809905e-11 ...  1.30434277e-09
 ↪   4.40060405e-09  0.00000000e+00]
 [ 0.00000000e+00  2.93809905e-11 -1.58985623e-12 ...  2.51012747e-09
 ↪   1.25823080e-09  0.00000000e+00]...
 [ 0.00000000e+00  1.30434277e-09  2.51012747e-09 ...  1.13299657e-09...
 [ 0.00000000e+00  4.40060405e-09  1.25823080e-09 ...  6.86719774e-10
 ↪   1.86470568e-09  0.00000000e+00]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  ...  0.00000000e+00
 ↪   0.00000000e+00  0.00000000e+00]]
```


Numerical solution of the biharmonic equation on a square
with Navier boundary condition and with m=50 a=-1 b=1
0.012
0.010
0.008
0.006
0.004
0.002
0.000
Z
−1.0
−0.5
0.0
0.5
1.0
X
−1.0
−0.5
0.0
0.5
1.0
Y