

Learnings from Assignment 4 - ID2090

Krishna Murari Chivukula, ME23B233

April 16, 2024

Abstract

This document is about the reflections and learning outcomes from Assignment 4 of the course ID2090, as a part of question 3.

1 Part A: Breaking The Spectre

1.1 Abstract of the Problem:

We were tasked with taking in a **YAML** serialized file as an input, with the file containing a matrix as an object. A script was written to operate on this matrix and obtain its eigenvalues and corresponding unitary and diagonal matrices. Furthermore, the spectral decomposition of the matrix was obtained and the matrix was classified into a certain domain. For this purpose, I used Python, with my main sub usage begin that of **SymPy**, a python library for symbolic mathematical computation, like that of matrix operations.

1.2 Initialising Functions:

After importing the necessary libraries from **SymPy**, I initialised functions for checking the necessary mathematical aspects of the matrix under consideration and classifying them.

- **Hermitian Matrix** : Given a matrix A , whose complex conjugate transpose is given by A^* , if $A = A^*$, A is said to be Hermitian.
- **Unitary Matrix** : Given a matrix A , whose complex conjugate transpose is given by A^* , if $AA^* = I$, where I is the identity matrix, it is Unitary.
- **Positive Semidefinite Matrix**: A positive semidefinite matrix is a symmetric matrix with non-negative eigenvalues (≥ 0).
- **Positive Definite Matrix**: A positive semidefinite matrix is a symmetric matrix with positive eigenvalues (> 0).

```
import yaml
from sympy import * #Imports all functions from the sympy library
import re
import sys

def check_hermitian(matrix): #Hermitian Matrices are ones which satisfy A = ComplexConjugateTranspose(A)
    return matrix.is_hermitian

def check_unitary(matrix):
    return matrix * matrix.adjoint() == eye(matrix.shape[0]) #Unitary Matrices are those that satisfy A*ComplexConjugateTranspose(A) = I

def check_positive_semidefinite(matrix):
    return all(eigenvalue >= 0 for eigenvalue in matrix.eigenvals().keys())

def check_positive_definite(matrix):
    return all(eigenvalue > 0 for eigenvalue in matrix.eigenvals().keys())

def matrix_classification(matrix):
    classifications = []
    if check_hermitian(matrix):
        classifications.append("Hermitian")
    if check_unitary(matrix):
        classifications.append("Unitary")
    if check_positive_semidefinite(matrix):
        classifications.append("Positively Semidefinite")
    if check_positive_definite(matrix):
        classifications.append("Positively Definite")
    if not classifications:
        classifications.append("Normal")
    return classifications
```

Figure 1: Initialising Functions

1.3 Preprocessing of YAML Object

This was followed by pre-processing the YAML file. I loaded the file into from the **YAML** object initialised as `obj.yaml`. This was followed by a series of conversions to **SymPy** matrices, namely Domain matrices or Mutable Dense Matrix. This was obtained from **SymPy** documentation, [here](#).

```
# Put the data in the YAML file from obj.yaml using the function yaml.load()U
with open('obj.yaml', "r") as file:
    yaml_data = yaml.load(file, Loader=yaml.Loader)

# Define a function to convert nested dictionaries to MutableDenseMatrix(in sympy)
def convert_to_sympy_matrix(data):
    if '_rep' in data and 'state' in data['_rep'] and 'rep' in data['_rep']['state'] and 'dictitems' in data['_rep']['state']['rep']:
        rows, cols = data['rows'], data['cols']
        matrix_data = data['_rep']['state']['rep']['dictitems']
        matrix_list = [[complex(matrix_data[i][j]) if isinstance(matrix_data[i][j], Number) else matrix_data[i][j] for j in range(cols)] for i in range(rows)]
        return MutableDenseMatrix(matrix_list)
    else:
        return None

# Define a function to convert dictionaries to DomainMatrix
def convert_to_domain_matrix(data):
    if 'args' in data and data['args']:
        args = data['args'][0]
        rows, cols = data['_rep']['state']['shape']
        matrix = DomainMatrix(args, rows=rows, shape=(rows, cols))
        return matrix
    else:
        return None

# Convert the YAML file to SymPy matrix or DomainMatrix so it can be operated on by matrix operations
if isinstance(yaml_data, MutableDenseMatrix):
    matrix = yaml_data
elif isinstance(yaml_data, dict):
    matrix = convert_to_sympy_matrix(yaml_data)
    if matrix is None:
        matrix = convert_to_domain_matrix(yaml_data)
else:
    matrix = None
```

Figure 2: Preprocessing YAML Object

1.4 Calculating Eigenvalues:

The Eigenvalue, λ of a matrix A , is the value which satisfies the equation, $Av = \lambda v$, for a column vector v . We use inbuilt **SymPy** functions for calculation. The print function was used to pretty print the output as how a human would draw a matrix.

```
#Eigenvalue List
eigen= matrix.eigenvects(simplify=True)
eigenvecs=[]
eigenvalslst=[]
for eigenvalue,multiplicity,vecs in eigen:
    for i in range(0,multiplicity):
        eigenvecs.append(vecs[i])
    for i in range(0,multiplicity):
        eigenvalslst.append(eigenvalue)

#Eigenvalues
eigenvals=Matrix([eigenvalslst])
print("Eigenvalues:\n")
pprint(eigenvals)
print("\n")
```

Figure 3: Eigenvalue Calculation

1.5 Spectral Decomposition:

Let A be an $n \times n$ matrix with distinct eigenvalues $\lambda_1, \lambda_2, \dots, \lambda_n$ and corresponding eigenvectors $\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_n$.

The spectral decomposition of A is given by:

$$A = Q\Lambda Q^{-1} \quad (1)$$

where $Q = [\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_n]$ is the matrix whose columns are the eigenvectors of A , and $\Lambda = \text{diag}(\lambda_1, \lambda_2, \dots, \lambda_n)$ is the diagonal matrix of eigenvalues. The matrix Q is orthogonal, i.e., $Q^T Q = I$, where I is the identity matrix, since the eigenvectors are linearly independent. Therefore, $Q^{-1} = Q^T$. Thus, the spectral decomposition of A can be written as:

$$A = Q \Lambda Q^T \quad (2)$$

The spectral decomposition provides a useful representation of A in terms of its eigenvalues and eigenvectors, which can simplify various computations and analyses involving A . The following code snippet can be referred to for Unitary Matrix and Spectral Decomposition.

```
eigenvecsmatrix = eigenvecs[0]
for col in eigenvecs[1:]:
    eigenvecsmatrix = eigenvecsmatrix.row_join(col)

U_cols=Matrix.orthogonalize(*eigenvecs,normalize=True)

#Unitary
U=U_cols[0]
for col in U_cols[1:]:
    U=U.row_join(col)
print("U:\n")
pprint(U)
print("\n")

Ut=U.conjugate().transpose()
D=Ut*matrix*U
print("D:\n")
pprint(D)
print("\n")
print("\nDecomposition:")

dict2={}
for i in range(0,3):
    if eigenvals[i] not in dict2:
        dict2[eigenvals[i]]=U_cols[i] * U_cols[i].transpose()
    else:
        dict2[eigenvals[i]]+=U_cols[i] * U_cols[i].transpose()

for key in dict2:
    print(key, '*')
    pprint(dict2[key])
```

Figure 4: Unitary Matrix and Spectral Decomposition

1.6 Final Classification and Defining Functions:

Here we define the functions that we initialised in the beginning. We also output the final classification.

```
# Check if the matrix is Hermitian
check_hermitian = matrix == matrix.conjugate().transpose()

# Check if the matrix is Unitary
check_unitary = matrix*matrix.conjugate().transpose() == matrix.conjugate().transpose()*matrix == eye(matrix.shape[0])

# Check if the matrix is Positively Semidefinite
check_pos_semidefinite = all(eigenvalue >= 0 for eigenvalue in matrix.eigenvals().keys())

# Check if the matrix is Positively Definite
check_pos_definite = all(eigenvalue > 0 for eigenvalue in matrix.eigenvals().keys())

# Check if the matrix is Normal
check_normal = matrix*matrix.transpose() == matrix.transpose()*matrix

# Output classification of the matrix
classification = "Classification:\n"
```

Figure 5: Final Classification

1.7 Conclusions:

The spectral decomposition is a potent tool in linear algebra, allowing matrices to be expressed in terms of their eigenvalues and eigenvectors. Its significance extends across multiple domains, including matrix diagonalization, stability analysis of dynamical systems, and dimensionality reduction methods like Principal Component Analysis in data science. Moreover, spectral decomposition facilitates the computation of matrix functions and offers valuable insights into the behavior of linear transformations. Its utility further extends to the analysis of Hermitian and unitary matrices, crucial in quantum mechanics and signal processing applications. The above mathematics, including eigenvectors and related concepts, form the fundamentals of Linear algebra, an extremely useful tool from an engineering perspective as well for machine learning.

1.8 Final Output:

I put the entire python code into a bash script titled question1.sh. The following output was obtained. I was not able to figure out how to format the spectral decomposition to obtain everything in a single line.

```
[me23b233@ID2090:~/assignment_4$ vi question_1.sh
[me23b233@ID2090:~/assignment_4$ ./question_1.sh obj.yml
Eigenvalues:

[0 0 3]

Matrix([[ -1, -1, 1], [1, 0, 1], [0, 1, 1]])
U:


$$\begin{bmatrix} \frac{-\sqrt{2}}{2} & \frac{-\sqrt{6}}{6} & \frac{\sqrt{3}}{3} \\ \frac{\sqrt{2}}{2} & \frac{-\sqrt{6}}{6} & \frac{\sqrt{3}}{3} \\ 0 & \frac{\sqrt{6}}{3} & \frac{\sqrt{3}}{3} \end{bmatrix}$$


D:


$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 3 \end{bmatrix}$$


Decomposition:
0 *

$$\begin{bmatrix} 2/3 & -1/3 & -1/3 \\ -1/3 & 2/3 & -1/3 \\ -1/3 & -1/3 & 2/3 \end{bmatrix}$$

3 *

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

Classification:
A is Hermitian, Positively Semidefinite.
```

Figure 6: Output of Bash Script

2 Part B:Optimaz

2.1 Abstract of the Problem:

We were given a set of (x, y, z) coordinates of points in a CSV file points.csv. The task was to find the optimal equation of the plane (more so the coefficients), fitting the behavior of the system of points, to the best possible extent, applying Newton's method. My main language of usage was **SageMath 10.2**, which I used through a **Jupyter Notebook**. In the end I compiled all code blocks into a **Bash** script for execution.

2.2 Initialising θ_0

We use the **Sage** commands as follows, to generate θ_0 as a matrix. For simplicity sake, I chose all elements of θ_0 to lie in between -10 and 10, hence declaring the randrange.

```
# Initialize theta with random values
initial_theta = matrix([[randrange(-10, 10)], [randrange(-10, 10)], [randrange(-10, 10)]])
theta = initial_theta
print('Initial Theta:', theta.transpose())

Initial Theta: [-6 -9 -8]
```

Figure 7: Initialising θ_0

2.3 Defining an Objective Function

The objective function(L) that was chosen to minimize, using **Newton's Method of Steepest Descent**, was the distance of an arbitrary point to the plane under consideration. However, upon some researching, I found out that for applying the above mathematical technique, the function being used must be a convex function. When including the denominator term in the formula for distance of a point from a plane, this criterion gets violated. I thought it best to omit that portion and proceed as follows.

```
# Function for calculating the distance of plane from a point
def distance_from_plane_square(x, y, z, points):
    squared_distances = [(x * point[0] + y * point[1] + z * point[2] - 1) ** 2 for point in points]
    return sum(squared_distances)
```

Figure 8: Defining the Objective Function

2.4 Reading From the CSV File:

We initialize an empty list names "points" and append the x,y,z coordinates of each point in the points.csv file to this list. Used strip to strip the file into individual lines and then split it based of commas. Some points to be noted:

```
# Read points from file
points = []
with open('$a', 'r') as f:
    next(f) # Skip the first line
    for line in f.readlines():
        coords = line.strip().split(',')
        point = vector([float(coord) for coord in coords])
        points.append(point)
```

Figure 9: Uploading points

- We skip the first line of the file, since that does not contain any data, rather only the column name.
- We turn "point" into a vector for ease of operation by the mathematical functions that follow.

2.5 Calculating Hessian and Gradient and Understanding Newton's Method:

- We are concerned with finding $\Delta\theta^t$, obtained from the formula $(H^{-1})^t g^t$, which forms the fundamental for **Newton's Method**.
- The Hessian, H_{ij} is given as $\frac{\partial^2 L}{\partial \theta_i \partial \theta_j}$ and g , the gradient is computed as $g_i = \frac{\partial L}{\partial \theta_i}$.
- **Newton's Method** is root-finding algorithm that uses the first and second derivatives of a function to iteratively improve an initial guess for the root. I referred to [this](#) article for reading up on the mathematics of it. This was from where I found out the necessary conditions required to operate on the objective function.
- We need to understand that this technique is in fact, different from the steepest descent method.

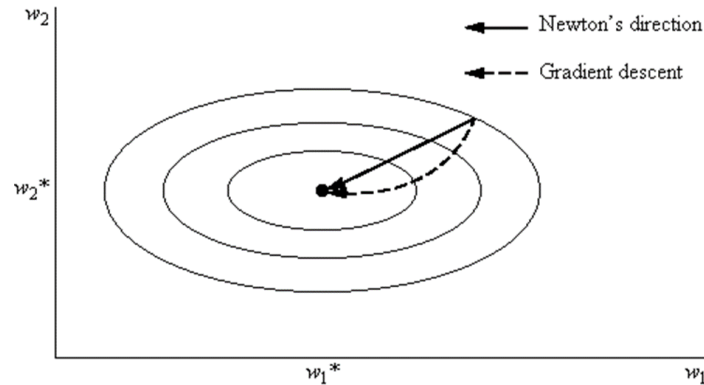


Figure 10: Difference in approaching optimal value

- [This](#) is another article from **Cornell** that I referred to, to understand the difference in intuition behind gradient(steepest) descent and Newton's method. Gradient descent possesses a wide application aspect in the field of machine learning, being the most fundamental algorithm to optimize loss functions.
- Notice that the Newton's method is theoretically the best 2nd order method due to its high convergence rate, specially when the initial guess for the optimal value is close to the actual value. However, it possesses its own disadvantages.
 1. **Requirement of Derivatives:** Newton's method requires the computation of both the first and second derivatives of the function, which may not always be readily available or computationally feasible, especially in complex or high-dimensional problems. The function under consideration needs to be differentiable.
 2. **Sensitivity to Initial Guess:** Newton's method may not converge or may converge to a different root if the initial guess is far from the actual root or lies in a region where the function behaves unexpectedly(oscillatory for example).
 3. **Computational Cost:** Computing derivatives, particularly second derivatives, can be computationally expensive, especially for large-scale problems.
 4. **Not Guaranteed to Converge:** Newton's method is not guaranteed to converge for all functions or initial guesses. It may diverge or oscillate in some cases. It can also cycle back to the initial guess after some steps.

- The following code block shows an initialisation of a certain number of epochs and the mathematical calculation of the Hessian and Gradient, as long as the error is less than a value of δ , which will be explained later.

```
# Initialize iteration count
iter = 0
f = distance_from_plane_square(x, y, z, points)
# Loop until error is less than delta or maximum iterations reached
while distance_from_plane_square(theta[0, 0], theta[1, 0], theta[2, 0], points) > $delta:
    # Calculate error
    error_value = f.subs({x: theta[0, 0], y: theta[1, 0], z: theta[2, 0]})

    # Calculate Hessian matrix and gradient matrix
    hessian_matrix = matrix([[diff(diff(f, x), x), diff(diff(f, x), y), diff(diff(f, x), z)],
                             [diff(diff(f, y), x), diff(diff(f, y), y), diff(diff(f, y), z)],
                             [diff(diff(f, z), x), diff(diff(f, z), y), diff(diff(f, z), z)]])

    gradient_matrix = matrix([[diff(f, x)], [diff(f, y)], [diff(f, z)]])

    # Substitute values into Hessian and gradient matrices
    hessian_matrix = hessian_matrix.subs({x: theta[0, 0], y: theta[1, 0], z: theta[2, 0]})
    gradient_matrix = gradient_matrix.subs({x: theta[0, 0], y: theta[1, 0], z: theta[2, 0]})
```

Figure 11: Hessian and Gradient

2.6 Updation of θ^t and Error tracking

- Before beginning, we initialise a value for a parameter known as δ . δ is the minimum value of error that the Newtons method must return, for the loop to break. Essentially, we treat it as a threshold and iteratively repeat all the calculations. After playing around with various values, for ideal computation time I chose $\delta = 0.000000001$ to be an optimal value, based of computational time, though it returned the answer in only 1 epoch.
- Upon further analysis, I realised that the number of epochs obtained depends on the δ value that is initialised. Changing it's value can change the number of epochs and time taken for execution.
- θ^t is updated using the rule, $\theta^{t+1} - \Delta\theta^t$. The error is also updated every step, as follows.

```
# Calculate delta theta and update theta
try:
    delta_theta = hessian_matrix.inverse() * gradient_matrix
    theta -= delta_theta
except Exception as e:
    print('Error occurred during theta update:', e)

# Calculate error after theta update
updated_error_value = f.subs({x: theta[0, 0], y: theta[1, 0], z: theta[2, 0]})

# Increment iteration count
iter += 1
```

Figure 12: Updation of θ and Error

2.7 Printing Optimal θ and Epochs

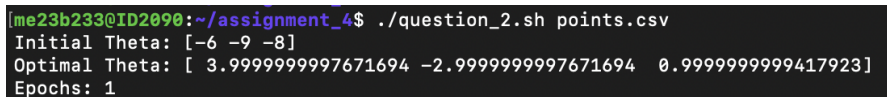
The final snippet is a simple print command to output the optimal value of our θ vector (outputted as a list) and the number of epochs taken to achieve it.

```
# Print the optimal matrix for theta and number of epochs required to achieve this
print('Optimal Theta:', theta.transpose())
print('Epochs:', iter)
EOF
```

Figure 13: Print Output

2.8 Compiling Code into a Bash Script and Sample Output

- I compiled all code snippets above into a single bash script, titled question2.sh that could be run on the VM. The script is run against a CSV file titled point.csv which contains x, y, z coordinates of a certain number of points as mentioned above. Sample output is shown. The attached image contains the script that was run on the VM terminal.



```
[me23b233@ID2090:~/assignment_4$ ./question_2.sh points.csv
Initial Theta: [-6 -9 -8]
Optimal Theta: [ 3.9999999997671694 -2.9999999997671694  0.9999999999417923]
Epochs: 1
```

Figure 14: Sample Output of Bash Script

3 References:

- <https://docs.sympy.org/latest/modules/matrices/dense.html> For SymPy documentation.
- https://amsi.org.au/ESA_Senior_Years/SeniorTopic3/3j/3j_2content_2.html For Newton's Method
- <https://www.cs.cornell.edu/courses/cs4780/2015fa/web/lecturenotes/lecturenote07.html> For Gradient Descent
- I referred to Wikipedia and 3b1b for reading more on the different types of matrices.