Ahmedabad University

STEADY STATE OF NETWORK FLOW

Using Linear Algebra Concepts

ABSTRACT
Each and Every system in the modern world holds a saturation point above which there is no possible way to optimize the system. Henceforth calculating the steady state of the network reflects the most important node in the network. Implementing the above idea with simple linear algebra concepts to find the steady state of the input populace matrix.

Ahmedabad University

Report: Linear Algebra

Name	Enrolment Number
Anshul Mehta	AU1940275
Kavan Desai	AU1940126
Harsh Patel	AU1940114
Nihal Aggarwal	AU1940217

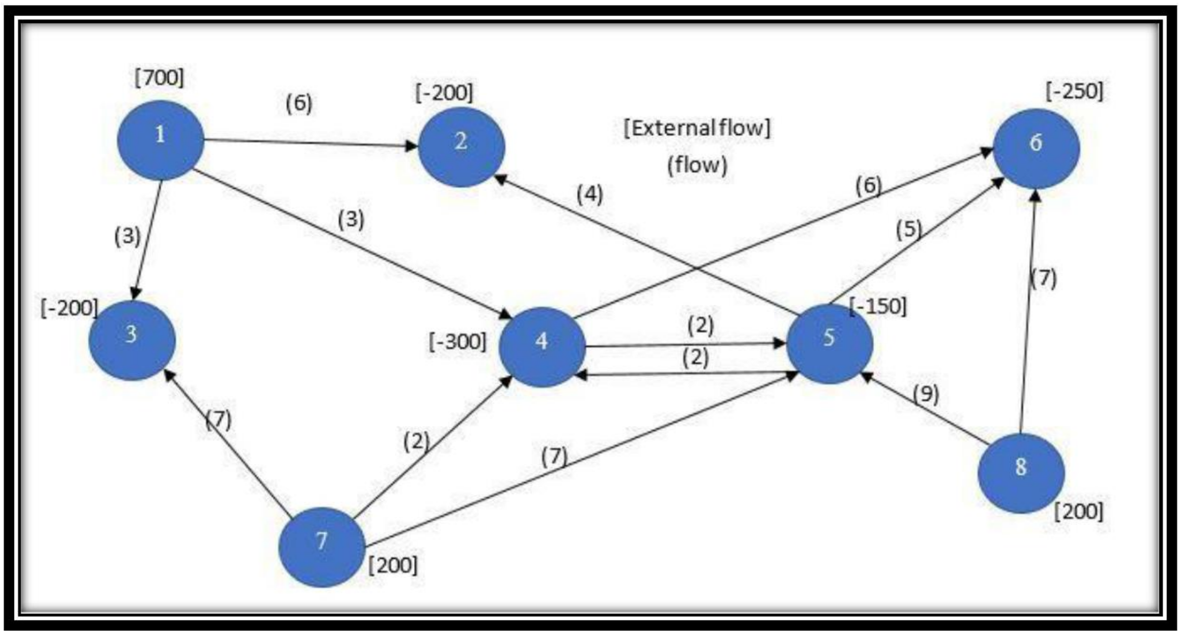
Submitted to Faculty: Professor Gaurav Goswami

- **Finding Steady State of a Network flow.**
  Page Rank to illustrate the Search Engine Results and asserting the fact that Steady state is one of the Eigenvectors of the Matrix.

Introduction

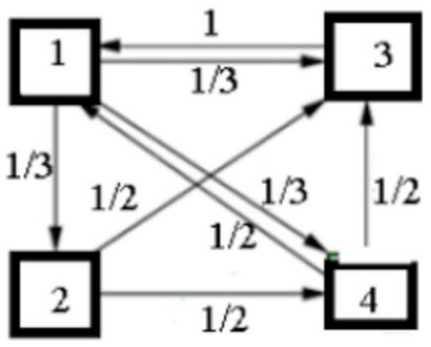
A network is a directed graph that forms a system of nodes and edges that connect them to each other. A network has multiple connotations across Graph Theory, Linear Algebra, Electric flow, Communication. Network flow is essentially a graph theory problem where it defines the capacity of each edge in a connected network to transfer data. Each edge has a given weight that indicates about what amount of data is viable to transfer through them. The summation of weights of edges connected to a particular node is called the weight of the node. The aim is to define a flow function for a given directed graph. Flow function is a model to the net amount of flow between pairs of nodes. Network flow can be used to model real life simulations of problems involving transfer of items, goods, data between locations and networks respectively. Examples include: Transportation networks, electrical networks and fluid networks. We in this case specifically use Markov Matrix to illustrate the stochastic process where probabilistic transitions between two nodes

Representation of a Network:



Representation of Graph/Network as a matrix:

The Network can be depicted as a graph where the nodes can be represented as spheres and the edges can be represented as arrows since it is a directed graph. The network in Linear Algebra can be shown as an incidence matrix which is a N x N matrix where N is the number of nodes. The edges in incidence matrix show us the values/weights. For example:



$$A = \begin{bmatrix} 0 & 0 & 1 & 1/2 \\ 1/3 & 0 & 0 & 0 \\ 1/3 & 1/2 & 0 & 1/2 \\ 1/3 & 1/2 & 0 & 0 \end{bmatrix}$$

Steady State of a network flow: The stabilized flow of the network results in such a flowthat the relative data, number of visitors at the nodes are plateaued according to the potential. Which implies that the percentage of visitors/data at the nodes are unchanged irrespective of the flow after a certain number of iterations.

For example: Matrix= [[0.7, 0.2], [0.3, 0.8]]

This matrix can be shown as the following network:

In this graph the flow of population at A is divided in to two parts where 70% of the population will stay at A and 30% will move to B. Whereas for B 20% will move to A and 80% stays at B.

Suppose Initial Population = 100 at A and B

Total=200

In the first iteration of the flow the respective populations are: 90 and 110

We go on iterating till a point where the exchange of flow does not affect the population of both the nodes which means:

30%A=20%B

At Steady state solution: [40.003381, 59.996619]

Final Population: 80 and 120

The dominant eigen vector of this matrix is: (2,3) which asserts our fact.

Eigen Centrality:

Degree Centrality is the number of nodes a node is connected with. This can be a good way to realize the importance of a node in the network. But this has a major drawback that if a let’s say a webpage created a lot of dummy websites that have links to a webpage it would increase the importance of that webpage although we know that isn’t the case. A good way to optimize this approach is Eigen Centrality. The approach measures a node’s centrality in the network by checking the number of influential nodes a node is connected with in the network. Thus, what happens in this case is that the dummy nodes become of no use and the ones with influence come into play.

$$C_e(v_j) = \frac{1}{\lambda}\sum_{J=1}^{n} A_j \cdot C_e C(v_j)$$

$$\lambda C_e = A^T C_e$$

Motivation:

Problems related to Network flow are extensively used in real life scenarios. One of the most prominent motivation was to understand the approach behind the famous PageRank algorithm that is used by Google in order to Rank pages. With ever increasing competition in Ranking Web-Pages on a search engine there arises a prominent question on how these pages are ranked. Whenever we look up for something on a search - engine we get millions and millions of website and webpages as the results. Since there are literally millions of pages that are retrieved as the result it becomes of prime importance to figure out which are the most relevant pages so that the user should not have to waste time looking up for results traversing web pages and should get desired result at the top. This process of ranking pages based on whether the user after searching key terms related to them stays on a webpage, or is directed to some other web page or simply moves to another webpage is useful info. This was the primary motivation to discover some subtle things about how concepts of linear algebra are used in order to rank the web page.

Overview:

We have defined our primary subject matter as steady state analysis of a network flow. Steady state analysis means repetitive analysis of a flow in network till it reaches the steady state where flow doesn’t affect the populations at nodes. This was the primary intuition behind ranking of web pages when the page rank Algorithm was designed. The World Wide Web is a diagraph where each webpage is a Node represented by a sphere in the graph and links from any given webpage are edges and we start off with the fact that the probability of getting redirected from page A to B where edge exist as a link from A to B. The relevancy of these pages was determined by how well is one page connected to other important pages and was given a rank and was displayed on the search engine accordingly.

Assumptions and Limitations:

As we have used Jacobi algorithm to find eigen vectors and eigen value, the limitation of Jacobi algorithm is that it only finds correct eigen vectors for symmetric matrix.
The sum of elements of each column should be unity as each column describes each node in network and network flow is always conserved.
Negative elements do not make sense in directional graph. So, each element of input matrix should be non-negative.

The test cases are preferably smaller matrices as we use matrix multiplication algorithm in O(n^3) time complexity which uses 3 For loops. For the diagonal matrix we scale our input matrix (repetitive analysis) till n=100 as for larger values there is a possibility of Integer overflow. Another assumption underpinning the network flow in general is that all the nodes have an equal rank.

Approach:

We have used python to code the Network flow and its’ steady state flow. We start off with the connectivity matrix of a given graph. The arrows of directed graphs are link that direct us from a webpage to the one it is pointing to. Since the entire web is taken as a network to illustrate network there are millions and millions of nodes in terms of webpages and it becomes a sparsely populated matrix as not all the webpages in the network are connected to other. This implies that the values need not be similar to each other and nodes carry different weights.

Why does this happen: This is because a rank of a matrix is dependent on its' connectivity to other important nodes and not merely number of connections. So, the uneven values become quite illustrative.

1) Ranks on a given node are always dependent on other webpages

We make use of the formula for calculating the rank of a node: R(j+1)=A*r(j)

According to our assumption as stated earlier our starting r = [1/n,1/n,1/n,1/n,1/n……….]

for n nodes

Then we find out the eigenvalues as eigen values and eigen vectors.
Now we decompose the matrix to $A = SDS^{-1}$where S contains the eigen vectors; Whereas D is diagonal matrix containing eigenvalues
Now we scale it to kth degree so that $A^k = SD^kS^{-1}$
Now once we get $A^k \cdot U_0 = U_k$ which was our equation

After a certain number of iterations when the value starts to plateau, we can say that the network has attained a steady state.

This means the vector that we finally get is the vector that represents the rank of the nodes in the given graph, which is useful in ranking them from the highest to lowest.

Perron Frobenius Theorem:

For every matrix there exists a maximum Eigen Value that is greater than all the other eigen values and the Eigen Vector Corresponding to it is the dominant Eigen Vector.
That values are called the Perron Frobenius values and it has a multiplicity 1.

Perron Frobenius for Ergodic Markov Chains:

In our specific approach we use Ergodic Markov Chains that are dynamical systems.
The PF value helps us determine the Steady state distribution of the Symmetric Markov Chain.
This can be intuitively established by P(x,y)=P(y,x) for transitions the Stochastic Process.
From Perron Frobenius Theorem we can establish that The PF values here will be 1

Why this approach works:

Steady State:
A process in Markov model is generally dependent on the past values. The given values are all transition probabilities.

In a network flow we start off with the assumption that larger volumes of data flows through a node that is highly connected. But a higher connectivity does not necessarily imply higher volumes of data/ higher number of visitors. What matters is how well is the node connected with other nodes of higher importance. We intend to do that with our approach.

This approach works because the probability of number of visitors/data going from one node to another depends upon the previous iteration. We capitalize on this fact that rank of a node only depends on the previous rank. So, we find the eigenvalues and eigenvectors to exponentiate in order to iterate. Then once we start iterating, we find out that the value becomes stagnant after a certain number of iterations. This is because eigen values play an important role in network stability. The buckling loads of objects to construction sites to vibrational measurements everything obtains a stability at the eigen values. This happens because sum of probability is 1 and in Markov Matrix the sum of probabilities of a column generally give out the values of dominant eigen value. Here in our case the dominant eigen values is 1. [2]

Suppose A has n Eigen Vectors that are Linearly Independent.

$$v1, v2, v3, \ldots\ldots vn$$

$$\lambda 1, \lambda 2, \lambda 3, \ldots\ldots \lambda n$$

Any vector in the sub space can be represented as

$$x = C_1 v_1 + C_2 v_2 + C_3 v_3 \cdots + C_n v_n$$

We keep on Multiplying A and get to the following result:

$$Av_1 = \lambda_1 v_1$$

$$Ax = C_1 \lambda_1 v_1 + \ C_2 \lambda_2 v_2 + C_3 \lambda_3 v_3 + \cdots + C_n \lambda_n v_n$$

$$A^2 x = C_1 \lambda_1^2 v_1 + C_2 \lambda_2^2 V_2 + \cdots + C_n \lambda_n^2 v_n$$

$$A^n x = c_1 \lambda_1^n v_1 + C_2 \lambda_2^n V_1 + \cdots + C_n \lambda_n^n V_n$$

$$A^n x = \lambda_1^n (c_1 v_1)$$

$$A^k n = C_1 \lambda_1^k v_1$$

$$A^k x = \lambda^2 \left(C_1 v_1 + C_2 v_2 \left(\frac{\lambda_2}{\lambda_1}\right)^2 \cdots C_n v_n \left(\frac{\lambda_2}{\lambda_1}\right)^k \right)$$

The dominant eigen value we take common and then we keep on iterating and get stabler values. It is a good approach that provides stability i.e. steady state. [1]

The eigen vector related to the dominant eigen vector is the steady state of the system. It is indeed an eigen vector of the matrix as expected in the results.

Proof using row operations:

$$m = \begin{bmatrix} a & d & g \\ b & e & h \\ c & f & i \end{bmatrix}$$

This is an input matrix of R3. Now we write the characteristic equation for the matrix

$$m = \begin{bmatrix} a-\lambda & d & y \\ b & e-\lambda & h \\ c & f & i-\lambda \end{bmatrix}$$

Here applying the row operation we add the elements.

$$m = \begin{bmatrix} a+b+c-\lambda & d+e+f-\lambda & y+h+i-\lambda \\ b & e-\lambda & h \\ c & f & i-\lambda \end{bmatrix}$$

$$s = a+b+c$$
$$s = d+e+f$$
$$s = g+h+i$$

$$m = \begin{bmatrix} s-\lambda & s-\lambda & s-\lambda \\ b & e-\lambda & h \\ c & f & i-\lambda \end{bmatrix}$$

Therefore, we can take s- λ common and it becomes the factor of the determinant. So, if s=1 then one of the eigenvalues of the input matrix will be 1.

Detailed Linear Algebra Concepts in context of our code:

- Matrix Multiplication
- Eigenvalues and Eigenvectors
- Inverse of a Matrix
- Jacobi Algorithm
- Diagonalization of a matrix

Matrix Multiplication:

In Calculation of eigen Values of Given Matrix
In Diagonalization to Multiply S, D and S^-1 Matrices

In finding Inverse of a matrix to multiply row reduced matrix

Eigen-Values:

Jacobi Method is used to find them

Then Eigen Values are placed into a diagonal matrix while diagonalization

They play an important role as the core topic of our code revolves around eigen-centrality at the steady state.

Eigen Vectors using Jacobi:

Jacobi method is an iterative method which is use to find out Eigen values and Eigenvector of real symmetric matrix only. To understand this method let's take a symmetric matrix A. The First step is that we find the largest non-diagonal element. Since this is symmetric there will be two of them. They can be represented as a[p][q] and a[q][p].Now we define a rotation matrix J1 in such a way that sin Θ will be at position A[q][p] and A[p][q] and cos Θ will be at position a[p][p] and a[q][q] and all other diagonal element will be one and all non-diagonal element will be zero.

Now we will multiply the A with J1^T and them again multiply the product of these two matrices with J matrix. Let the obtain matrix be A1(A1=J1t*A*J). Now A1 will be a symmetric matrix. If A1 is not a diagonal matrix, then perform the Jacobi method on it and obtain another rotation matrix J2, and then again try to diagonalized the matrix A2 by multiplying it with transpose of J2 and then multiplying this product with J2. If the obtain matrix is non diagonalized matrix then repeat same process until the resultant matrix is diagonalized. The resulting matrix consists of Eigen Values as the diagonal elements

Eigen vectors are simply the multiplication of J1*J2*…. Jn i.e. all the rotation matrices.[3]

$$A' = J(p_1 q_1 \theta)^T A J(p_1 q_1 \theta)$$

$$\phi = cot\,2\theta = \frac{Aqq - App}{2Apq}$$

$$tan\,\theta = \Phi + \sqrt{\emptyset^2 + 1}$$

$$= I_1 * I_2 * I_3 \cdots$$

[7]

Inverse of a Matrix: Invertible Matrix Theorem states that every n x n is row equivalent to an n x n Identity Matrix. The matrix A-Lambda(I) must be non-invertible in order to find eigen values is a subtle concept we use. For the second part we use Inverse function in order to invert the matrix S. We do that by row reduction. We start off with AX=I where I is Identity matrix. Now we row reduce it to echelon form using row operations till a point that the augmented matrix [A | I] is converted [E(A),A^-1]. [5]

Equations:

$$AA^{-} = I$$

$$Ax = I$$

$$Ax_1 = e_1$$

$$Ax_2 = e_2$$

$$Ax_3 = e_3$$

$$\cdots$$

$$Ax_n = e_n$$

$$[A|e_j]$$

$$[I|x]$$

$$x \to A^{-1}$$

Row Reduction: Although we do not have an explicit function defined for row reduction and we do it within the inverse function it becomes an indispensable concept of linear algebra that is used. According to Invertible Matrix theorem we the idea of inverting using row reduced echelon form.[4]

Diagonalization (Eigen-Decomposition) And Final Calculations for ranks: Diagonalization is used in the context of our project to split the input matrix into S, D AND S ^-1 so that we can with ease exponentiate the matrix and thus iterate with ease.

$$D = \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix}$$

$$A = SDS^{-1}$$

$$A(A) = SDS^{-1}(SDS^{-1})$$

$$A^2 = SD^2S^{-1}$$

$$A^3 = SD^3S^{-1}$$

$$\cdots$$

$$A^k = SD^kS^{-1}$$

$$A^k \cdot U_0 = U_k$$

$$SD^k S^{-1}U_0 = U_k$$

$$C = S^{-1}U_0$$

$$SD^kC = U_k$$

[6]

Approach to calculate steady state:

So here, as we want Uk which will be steady state of network flow. So U0 is simply U0. As U0 is a predefined matrix which is equalized distribution of each node. Here we can say U1=A*U0, where A is input matrix and U2=A*U1, so here every state depends on the past state. When we are iterating up to A^k the system experiences transition k times and then if we multiply with the probability distribution, we will get the final probability of transition which indicates the steady state. Assuming the Kth state is steady state. So, after K iterations we get steady state of network flow. Here comes Linear algebra concepts. Multiplying a matrix K times is not a good idea which is naïve approach indeed. So, to optimize, we diagonalize matrix and get A=SDS^{-1}. when we multiply SDS^{-1}K time S^{-1}S will become Identity so we get is S(D^k) S^{-1}multiplying diagonal matrix k time is far easier than multiplying A k times. Now we need S, D and S^{-1} so we need to find eigenvalues and eigen vectors. Here Jacobi algorithm is used to find eigenvalues and eigen vectors.
And finally, we get steady state of network flow.

How our code works

We take Input as a matrix which is a representation of network flow which has some predefined flow.

Firstly, we have found eigen vectors and eigen values with the use of Jacobi algorithm.

We have strategized our code to decrease time complexity as again and again multiplying a diagonal matrix K times is now and then a naïve approach, here the value k may also land up to 100-200 iterations, which is not a practical solution. So, a very trivial concept of linear algebra is that any diagonal matrix to the power k is equal to a diagonal matrix with all diagonal elements to the power k. So, we are scaling k iterations to reduce time complexity to O(n^3) which is quite efficient. And now finally by multiplying 4 matrix S, D^k, $S^{-1}$and U0 we get steady state of network flow which is our desired result.

Explanations:

This is our main method. Here we invoke our predefined methods and obtain the results.

1) Matrix Multiply and Inverse are our pre-defined functions.
2) We use Jacobi method to find eigenvalues and eigen vectors
3) We define a Max Element function in Jacobi method that gives us the largest non-diagonal element in the matrix. The largest element then plays a decisive role in deciding the rotation matrix
4) This method then reduces the off diagonal elements to 0 in order to obtain the eigen values using the Formula Off(A') ^2= Frobenius Norm -Square of trace of matrix
5) Eigen Vectors are given by J1*J2*……
   Eigen Vectors give S and S^{-1} (by invoking inverse function) Populate D with Eigen Values at the Position D[i][j] s.t i=j using for loops.
6) Now we have our S, D and S^{-1}Now here in our strategy to reduce time complexity we apply the fact that while scaling S*D*$S^{-1}$and multiplying it again with S*D*S^-1 the S*S^{-1}in between of the expression makes for an identity matrix.
7) This optimizes our approach as we use the naïve multiplication algorithm which takes O(n^3) complexity. We directly scale the values of Lambda-Eigenvalues up till k, which the desired number of iterations we would like to multiply S*D*S^{-1}in order to get A^k.
8) Then we multiply (A^k) ***P0 which is our** population matrix.

The P0 is the matrix which consists of columns as **the populations available at the Nodes**/ Webpages. The final Multiplication in the aforementioned step gives us the Steady State of the Network Flow
Shortcomings- Only available for symmetric matrices

This is the Final Answer that will give us our steady state of the Network.

Modules Used: Numpy (Only for making arrays, Linear algebra functions are defined separately)

Structure of the code:

Define function Jacobi
Define Max-Element(A)
Return Max Element

Define Rotate (A, r,d,e)
#Creates a Rotation matrix
Return Eigen values, Eigen Vectors

Define MatrixMultiply:
# Naïve algorithm using 3 for loops for Matrix Multiplication

Define Inverse (Matrix A)
Define ZeroMatrix:

Creates a matrix of N x N with all 0s in elements Define CopyMatrix:
Creates a duplicate of the Matrix

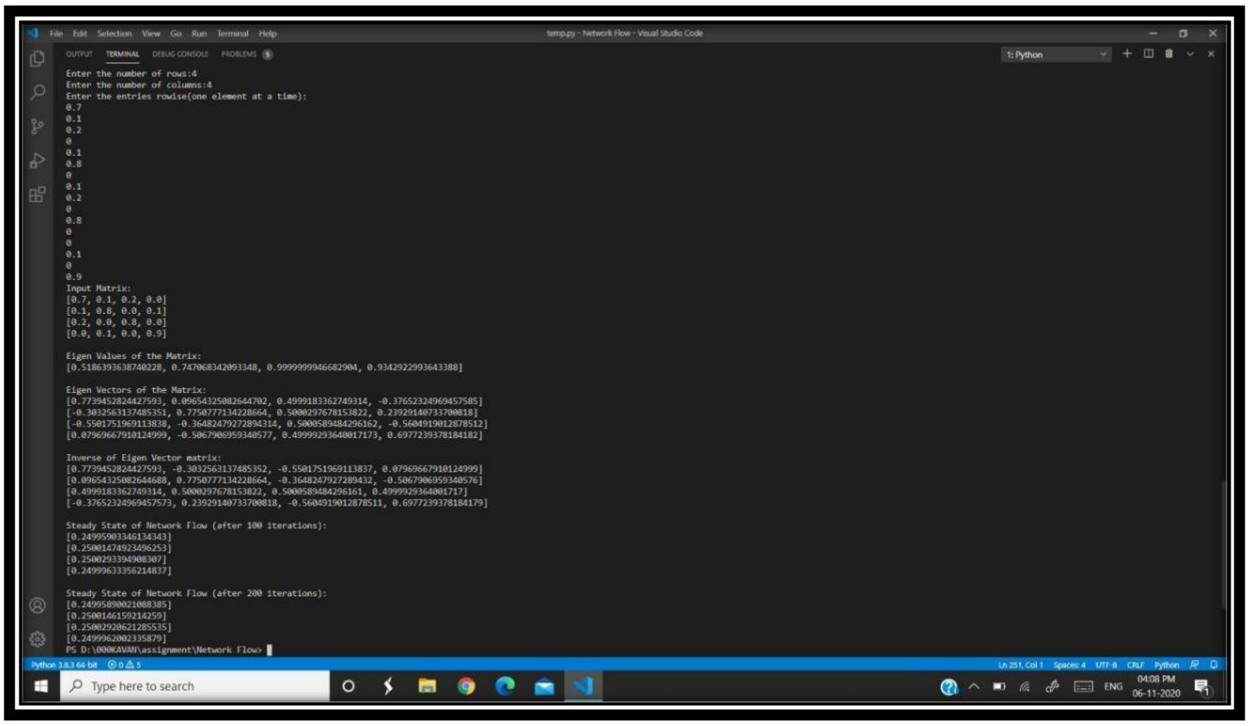
Creates an Identity Matrix with 1 as the diagonal elements of the matrix
Define MatrixMultiply:
Naïve algorithm using 3 for loops for Matrix Multiplication
Now we row reduce the Matrices in-order to get A^-1 and I

Scaling the values here directly by power k

Diagonal matrix of Eigen Values raised to k Return D Matrix

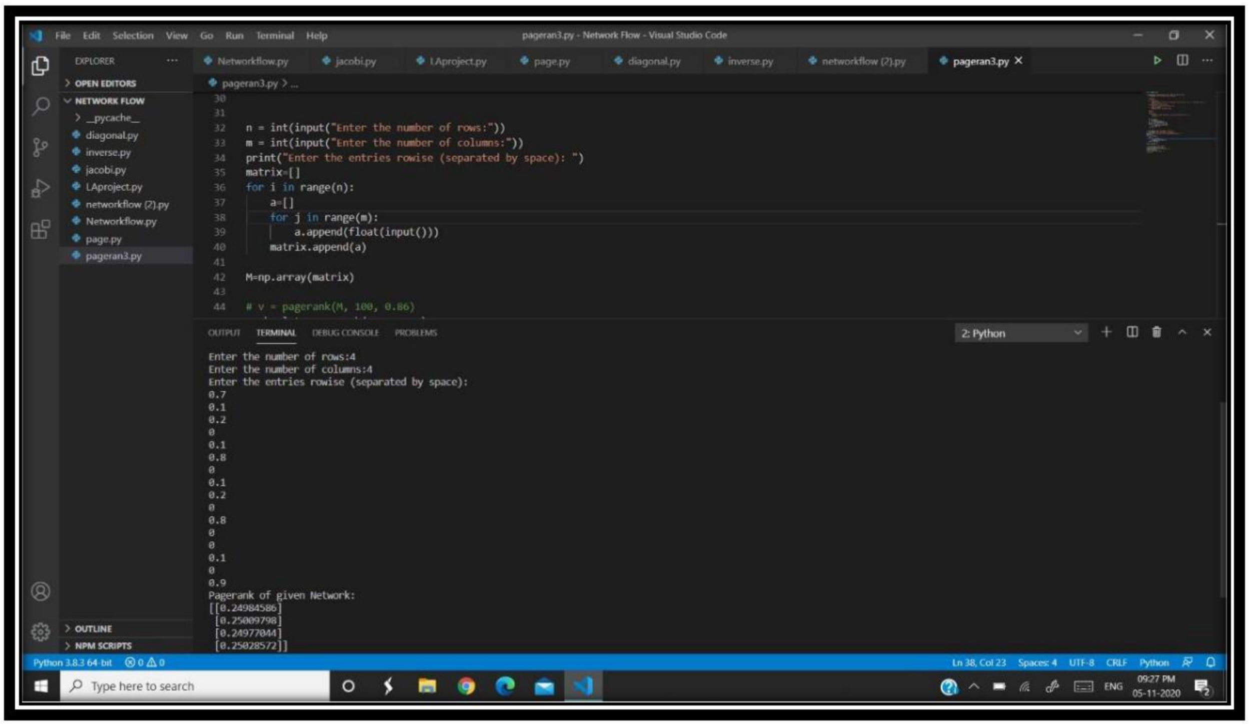
Using MatrixMultiply we Multiply S with D and then S^-1 and the P0
This give the value of our steady state in Network Flow.

Result of our code:



```
Enter the number of rows:4
Enter the number of columns:4
Enter the entries rowise(one element at a time):
0.7
0.1
0.2
0
0.1
0.8
0
0.1
0.2
0
0.8
0
0
0.1
0
0.9
Input Matrix:
[0.7, 0.1, 0.2, 0.0]
[0.1, 0.8, 0.0, 0.1]
[0.2, 0.0, 0.8, 0.0]
[0.0, 0.1, 0.0, 0.9]

Eigen Values of the Matrix:
[0.5186393638740228, 0.747068342093348, 0.9999999946682904, 0.9342922993643388]

Eigen Vectors of the Matrix:
[0.7739452824427593, 0.09654325082644702, 0.4999183362749314, -0.37652324969457585]
[-0.3032563137485351, 0.7750777134228664, 0.5000297678153822, 0.23929140733700818]
[-0.5501751969113838, -0.36482479272894314, 0.5000589484296162, -0.5604919012878512]
[0.07969667910124999, -0.5067906959340577, 0.49999293640017173, 0.6977239378184182]

Inverse of Eigen Vector matrix:
[0.7739452824427593, -0.3032563137485352, -0.5501751969113837, 0.07969667910124999]
[0.09654325082644688, 0.7750777134228664, -0.3648247927289432, -0.5067906959340576]
[0.4999183362749314, 0.5000297678153822, 0.5000589484296161, 0.4999929364001717]
[-0.37652324969457573, 0.23929140733700818, -0.5604919012878511, 0.6977239378184179]

Steady State of Network Flow (after 100 iterations):
[0.24995903346134343]
[0.25001474923496253]
[0.2500293394908307]
[0.24999633356214837]

Steady State of Network Flow (after 200 iterations):
[0.24995890021088385]
[0.2500146159214259]
[0.2500292062128553]
[0.2499962002335879]
PS D:\000KAVAN\assignment\Network Flow>
```

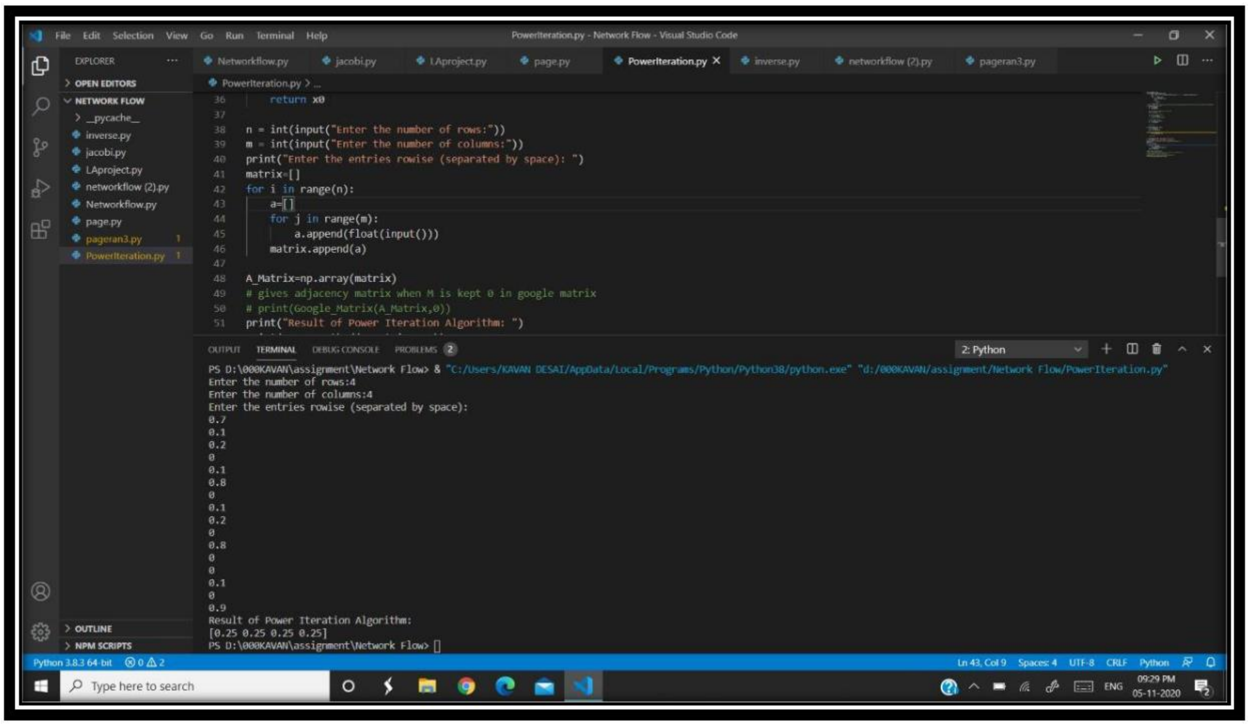
Result of PageRank Algorithm:



```python
n = int(input("Enter the number of rows:"))
m = int(input("Enter the number of columns:"))
print("Enter the entries rowise (separated by space): ")
matrix=[]
for i in range(n):
    a=[]
    for j in range(m):
        a.append(float(input()))
    matrix.append(a)

M=np.array(matrix)

# v = pagerank(M, 100, 0.86)
```

```
Enter the number of rows:4
Enter the number of columns:4
Enter the entries rowise (separated by space):
0.7
0.1
0.2
0
0.1
0.8
0
0.1
0.2
0
0.8
0
0
0.1
0
0.9
Pagerank of given Network:
[[0.24984586]
 [0.25009798]
 [0.24977044]
 [0.25028572]]
```

Result of Power Iteration Algorithm:



```python
        return x0

n = int(input("Enter the number of rows:"))
m = int(input("Enter the number of columns:"))
print("Enter the entries rowise (separated by space): ")
matrix=[]
for i in range(n):
    a=[]
    for j in range(m):
        a.append(float(input()))
    matrix.append(a)

A_Matrix=np.array(matrix)
# gives adjacency matrix when M is kept 0 in google matrix
# print(Google_Matrix(A_Matrix,0))
print("Result of Power Iteration Algorithm: ")
```

```
PS D:\000KAVAN\assignment\Network Flow> & "C:/Users/KAVAN DESAI/AppData/Local/Programs/Python/Python38/python.exe" "d:/000KAVAN/assignment/Network Flow/PowerIteration.py"
Enter the number of rows:4
Enter the number of columns:4
Enter the entries rowise (separated by space):
0.7
0.1
0.2
0
0.1
0.8
0
0.1
0.2
0
0.8
0
0
0.1
0
0.9
Result of Power Iteration Algorithm:
[0.25 0.25 0.25 0.25]
PS D:\000KAVAN\assignment\Network Flow> 
```

The results are equal which illustrate that indeed the steady state of our network Is equal if the network is considered as WWW and the Columns i.e. nodes are considered as Webpages.

The Page rank connection:

As in page rank we entertain random clicks/visits to a particular link/node of network or with the real time data sets we can get "more important page" as more scored page rank and less important page as less scored page rank. whereas in network flow we have some predefined flow, after over and over iterations we come across a steady state which is steady state of a network flow. In steady state of a network flow we get more network flow to a node which redirects us to page rank. [8]

Inferences:

Some very qualitative results can be drawn. The symmetric matrices do give a steady state where the vectors that represents the Steady state is (1, 1, 1, ….). The non-

symmetric matrices do give out the percentage of population/ data that is present at the saturation/ steady state of the flow. The steady state also speaks about page rank in which the value of each node which is a webpage shows what numbers of visitors are going to stay on a given node and what number are going to transit. This helps in deciding the relevancy of the pages. For example, if a webpage has a steady state solution value of 0.90 that shows us that while randomly clicking on webpages a surfer might find that the node given is very relevant and 90% chances are there that he/she might not transit to another website. Whereas if a node has 0.05 as the value then it means only 5% visitors find that node relevant and rest might jump on.

There can be one important inference drawn that the Page Rank directly iterates through matrix while Network flow uses a slightly different approach of iterating over the rank vector while Network flow depends on eigen values but give out the same result. Thus, in every iteration of PageRank we can also take out common Eigen Vector till the value which we keep on removing as common becomes redundant.

One very important inference that can be drawn from this is that the steady state is an eigen vector of the matrix. This tells us that eigen centrality plays a very important role in finding out the influence of a node in network and to extrapolating the subtle concept to a massive algorithm like Page Rank – GOOGLE

NOTE- The Page Rank Algorithm is essentially not a part of our definition. It has been used to illustrate the fact steady state is actually what we intended to find through page rank algorithm as well. The Page Rank Algorithm has been imported from Wikipedia References:

Conclusions:

The Steady State of a network is indeed an (proportional) eigen vector of the input Matrix

The Page Rank Algorithm and Power Iteration Algorithm give out similar results as our code because of eigen centrality.

Repetitive analysis does actually give out the steady state of the network flow and so does it give out convincing results for algorithms like Page rank

Contributions of Group mates:

Anshul

- Coding
- Research about Mathematical reason of steady state and Eigen Centrality
- Research about why steady state is one of the eigen vectors of the matrix and why the dominant one and how Page Rank Iteration is connected to it.
- Research about Jacobi method and how eigen values are obtained in a diagonal matrix through rotation matrices.
- Written and formatted the report content, equations, images and content.

- Designed Symmetric test cases ranging from 2*2 to 5*5 matrices based on the property probability (Markov) that we have used and analysed them.

Harsh

- Main approach for calculating steady state for the project.

- Research about the how steady state of network flow is linked with Page Rank.

- Coding the main content.

- Presentation

- Research about different algorithm to calculate eigen values and eigen vectors.

- Writing the Report

Kavan

- Coding the core part of the project

- Maintained the code through-out the project

- Researched about how to inculcate inverse in python.

- Approach for diagonalization in python Optimised approach for iterative call, reduced time complexity to O(n^3)

- Researched how Steady state of Network flow is linked with PageRank.
- Analysed why our code works.

- Writing the Report

Nihal

- Coding the Core of the Project

- Research about how to inculcate Matrix Multiply in the Code

- Analysis about the Eigen Vector and Eigen values and Diagonalization and inculcating them in the project

- Analysing the Steady State of Network Flow with a Mathematical Approach.

- Making the Presentation
- Writing the Report

- ## References

1) Dynneson, “Google’s PageRank algorithm powered by linear algebra,” 2010. [Online].Available:http://online.sfsu.edu/meredith/Linear_Algebra/725_F2010/PDF/Dynneson_FinalDraft_Linear-Algebra-Project.pdf.[Accessed: 2020].
2) Elsevier, “Markov Processes for Stochastic Modeling,” *Markov Processes for Stochastic Modeling - 2nd Edition*, 03-Jun-2013. [Online].Available:https://www.elsevier.com/_dynamic/product-display?isbn=978-0-12-407795-9.[Accessed: 29-Nov-2020].
3) A. T. Cruz, “Jacobi Method: Eigenvalues and Eigenvectors ,” *Jacobi Method: Eigenvalues and Eigenvectors*, 2015. [Online]. Available:https://www.patnauniversity.ac.in/econtent/science/physics/MScPhy58.pdf. [Accessed: 2020].
4) A. Vainchtein, “Using row reduction to calculate the inverse and the determinant of a squarematrix.”[Online].Available:http://www.math.pitt.edu/~annav/0290H/row_reduction.pdf. [Accessed: 2020].
5) Published by Thom Ives on November 1 and T. Ives, “Simple Matrix Inversion in Pure Python without Numpy or Scipy,” *Integrated Machine Learning and Artificial Intelligence*, 01-Nov-2018.

[Online]Available:https://integratedmlai.com/matrixinverse/. [Accessed: 29-Nov-2020].

6) P. D. Andrew Chamberlain, “Using Eigenvectors to Find Steady State Population Flows,” *Medium*, 04-Oct-2017. [Online]. Available: https://medium.com/@andrew.chamberlain/using-eigenvectors-to-find-steady-state-population-flows-cd938f124764. [Accessed: 29-Nov-2020].
7) B. Gonçalves, “Network Effects Explained: PageRank and Preferential Attachment,” *Medium*, 04-Oct-2016. [Online]. Available: https://bgoncalves.medium.com/network-effects-explained-pagerank-and-preferential-attachment-61fdf93d023a. [Accessed: 29-Nov-2020].
8) “Redirect Notice,” *Google*. [Online]. Available: https://www.researchgate.net/figure/Example-of-a-network-flow-programming-model_fig2_318840112 . [Accessed: 29-Nov-2020].
9) *Google Images*. [Online]. Available: https://www.google.com/imgres?imgurl=https%3A%2F%2Fwww.samford.edu%2Fsports-++analytics%2Ffans%2F2018%2Fimages%2FNBA-Directed-Network-Graph.png%2Cpagerank. [Accessed: 29-Nov-2020].
10) “Perron-Frobenius Theory,” 2008. [Online].Available:https://web.stanford.edu/class/ee363/lectures/pf.pdf. [Accessed: 2020]