Spl-1 Project Report- 2023

Leverage Score Sampling

Submitted by

Mahir Faisal

BSSE Roll No: 1316

Session- 2020-2021

Supervised by

Dr.Mohammad Shoyaib

Professor

Institute of Information Technology

Supervisor's Approval: ______________________

(signature)

Institute of Information Technology

University of Dhaka

21-05-2023

Table of Contents

List of Figures

1. Introduction

Leverage Score Sampling is a technique used in data analysis to extract a representative subset of data points from large datasets. The technique is necessary because dealing with large datasets that have a high dimensionality can be computationally expensive and time-consuming. By selecting a representative subset of data points using Leverage Score Sampling, it becomes possible to perform various analyses on the dataset while saving time and computational resources.

Furthermore, **Leverage Score Sampling** allows for the extraction of important features of the dataset. This can be particularly useful when dealing with complex datasets where it is difficult to identify the key features. By identifying the important features of the dataset, it becomes possible to gain insights that may not be possible with other methods.

It has numerous applications in real life. Here are some examples of its importance :

1. Medical Research: In medical research, large datasets are often used to identify risk factors for various diseases. Leverage Score Sampling can be used to extract a representative subset of data points from these datasets, making it easier and faster to identify the risk factors.

2. Finance: In finance, large datasets are used to analyze market trends and predict stock prices. Leverage Score Sampling can be used to extract a representative subset of data points, making it easier to identify patterns and make predictions.

3. Marketing: In marketing, large datasets are used to analyze consumer behavior and develop marketing strategies. Leverage Score Sampling can be used to extract a representative subset of data points,

making it easier to identify trends and develop effective marketing campaigns.

4. Image and Video Processing: In image and video processing, Leverage Score Sampling can be used to compress large images and videos, making them easier to store and transmit.

Overall, **Leverage Score Sampling** is an important technique in data analysis that can help to improve the efficiency and effectiveness of analyses.

1.1 Background

Before we deeply discuss the project, some important terms should be clarified for proper comprehension of the implementation.

➢ **EigenValue** Eigenvalue is a mathematical concept that is commonly used in linear algebra and other fields of mathematics. In simple terms, eigenvalues are a set of scalar values that are associated with a matrix. It is important because it provides information about the properties of a matrix.

$$|\boldsymbol{A}-\boldsymbol{\gamma I}|=\boldsymbol{0}; \text{ where } \gamma \ is\ the\ eigenvalue\ of\ A$$

➢ **EigenVector** Eigenvector is a nonzero vector that, when multiplied by a matrix, results in a scalar multiple of the original vector. For easy to say, an eigenvector is a vector that remains in the same direction after it is transformed by a matrix. It is used to represent the direction of the greatest variance in a dataset.

$$|\boldsymbol{A}-\boldsymbol{\gamma I}|\boldsymbol{x}=\boldsymbol{0}; \text{ where x is the eigenvectors for } \boldsymbol{\gamma}.$$

➢ **l2 norm** l2 norm also known as the euclidean norm, is a measure of the length or magnitude of a vector in a euclidean space. It is defined as the square root of the sum of square values of the vector component.

$$\textbf{L2 norm} = \sqrt[2]{\sum_{1}^{k}(x^{2})}$$

➢ **SVD** Singular value Decomposition(SVD) is the process to break down a matrix into its constituent parts. SVD involves decomposing a matrix A into three matrices : U, Σ & V. U contains the left singular vectors of A where V contains the right singular vectors, & Σ contains the singular values of A.

$$\mathbf{A = U\Sigma V^{t}}$$

➢ **Singular Value** It is very important concept in linear algebra. Basically they are positive square roots of the Eigenvalues of a data matrix. They indicate how much each direction in the input space of a linear transformation is a scaled or contracted transformation.

Singular values = $\sqrt[2]{\gamma i}$ **;** where $\gamma i \text{ is the eigen values}$

➢ **Leverage Score** Leverage score is a mathematical measure used in linear algebra to determine the importance of each row/column of a matrix. Leverage scores are important because they can be used to identify influential data points or outliers in a dataset.

$$\textbf{Leverage score} = \sum_{1}^{n}((Ui,j)^{2});$$

where Ui,j is the element of I'th row & I'th column of left singular factor U

1.2 Why SVD for Leverage Score Sampling

Singular Value Decomposition (**SVD**) is an important technique used in data analysis, and it plays a crucial role in computing Leverage Score Sampling. It provides a number of benefits, including efficient computation, data compression, dimension reduction & the identification of important features of a dataset.

1. Efficient computation: **SVD** provides an efficient way to decompose a matrix into its constituent parts. This can be particularly useful when dealing with large matrices, as SVD allows for the decomposition to be performed quickly and accurately.

2. Data compression: **SVD** can be used to compress data by reducing the number of dimensions. This can be useful when dealing with large datasets, as it can reduce the amount of memory required to store the data.

3. Dimension reduction: **SVD** can be used to identify the most important features of a dataset by reducing the number of dimensions. This can be particularly useful when dealing with high-dimensional datasets, as it can make it easier to analyze the data and identify patterns.

4. Leverage Score Sampling: **SVD** is used to compute Leverage Score Sampling by identifying the most important rows of a matrix. This can be useful when dealing with large datasets, as it allows for a representative subset of data points to be extracted while maintaining the important features of the dataset.

1.3 Challenges Faced

While accomplish that project, I had face some challenges that were not too easy to resolve. Some of these are :

➢ **Finding Appropriate Algorithm** : At the very beginning of this project, it was very difficult to find the algorithms that are needed or related to compute **SVD** (Singular Value Decomposition).

➢ **Implementation of the Algorithms :** There was a little bit difficulty to implement the Bairstow & Faddeev Leverrier algorithm.

➢ **Manage Higher order matrix :** For higher order matrix the Faddeev Leverrier shows the coefficients which are very large value. But Bairstow will fail for the large eigenvalues. So, then we had to use another algorithm, Jacobi Rotation.

➢ **Manage Large Codes** : Working with a large codebase is a tedious task. It becomes very difficult to track changes on different files.

➢ **Debugging codes** : Frequently, I got segmentation problems or dumped code errors but I was not sure where it happening. Then this issue was instantly solved by debugging the CPP files.

2 Description of the Project

My project is designed to compute Leverage Score Sampling using SVD(Singular Value Decomposition) to efficiently select a subset of rows from a large dataset. Basically, it divides the dataset into important & non-important features. It includes the following attributes:

- ➢ Read data matrix from file
- ➢ Transpose of the matrix
- ➢ Multiply transpose matrix with original matrix

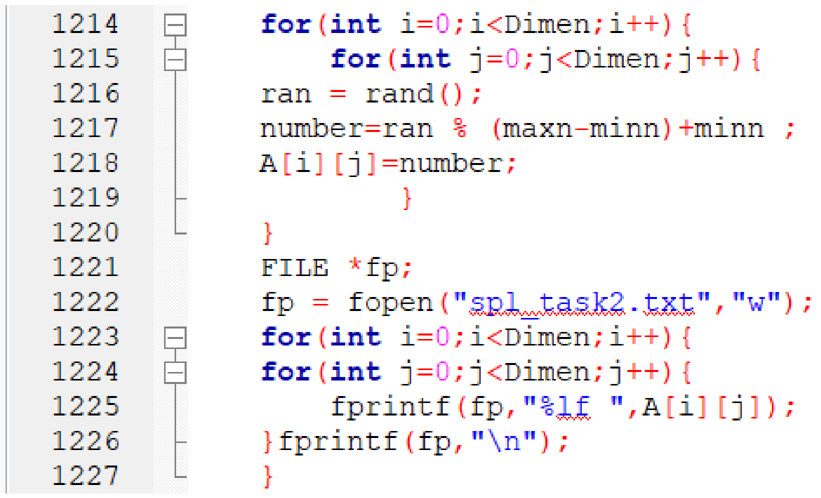
if(dimension< 10) {

- ➢ Faddeev Leverrier Algorithm
    - ➤ Calculate coefficients of the characteristic equation
- ➢ Bairstow Algorithm
    - ➤ Create new array
    - ➤ Delete array

```
            ➤ Calculate absolute value
            ➤ Remove error
            ➤ Root finding
                    ➡ Print real & complex root
                    ➡ Print real root only
            ➤ Calculate initial guess r & s
            ➤ Reduce power of the polynomial equation
            ➤ Get coefficient after every iterate
      ➢ Gauss Elimination
            ➤ Gauss elimination method
            ➤ forward elimination
            ➤ Back substitution
            ➤ Swap row
    }

else {
      ➢ SVD For Higher Order
      }

      ➢ Pseudo Inverse
      ➢ Leverage Score Sampling
      ➢ Matrix Projection
```

2.1 Read Data Matrix from file

Here, a user can generate random numbers as a matrix. We maintained two for loops that will continuously use the ‘rand’ method to generate the numbers up to the dimension. And then write the matrix into the file “ **spl_task2.txt** ”. Then read the matrix.



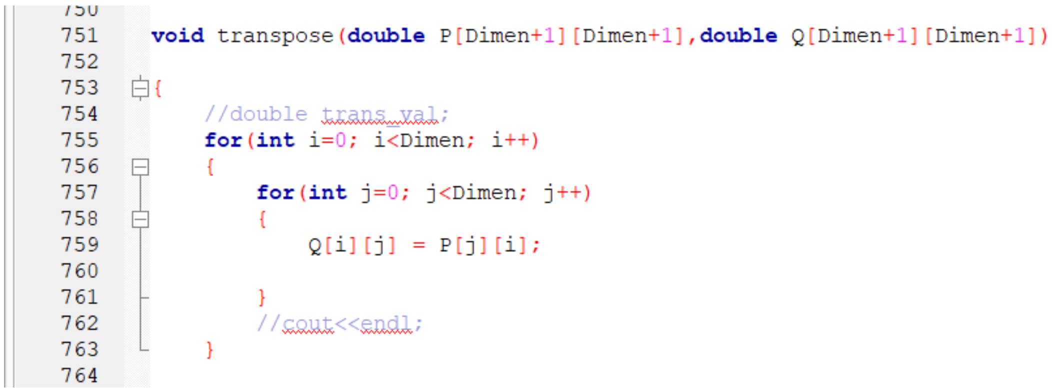
```
    for(int i=0;i<Dimen;i++){
        for(int j=0;j<Dimen;j++){
    ran = rand();
    number=ran % (maxn-minn)+minn ;
    A[i][j]=number;
            }
    }
    FILE *fp;
    fp = fopen("spl_task2.txt","w");
    for(int i=0;i<Dimen;i++){
    for(int j=0;j<Dimen;j++){
        fprintf(fp,"%lf ",A[i][j]);
    }fprintf(fp,"\n");
    }
```

Figure 1: generate random number

2.2 Transpose of the Matrix

This method calculates the transpose of the original matrix. If the original matrix is A, then the transpose of the matrix that the method calculates is Aт. Aт is found for each i’th row & j’th column as

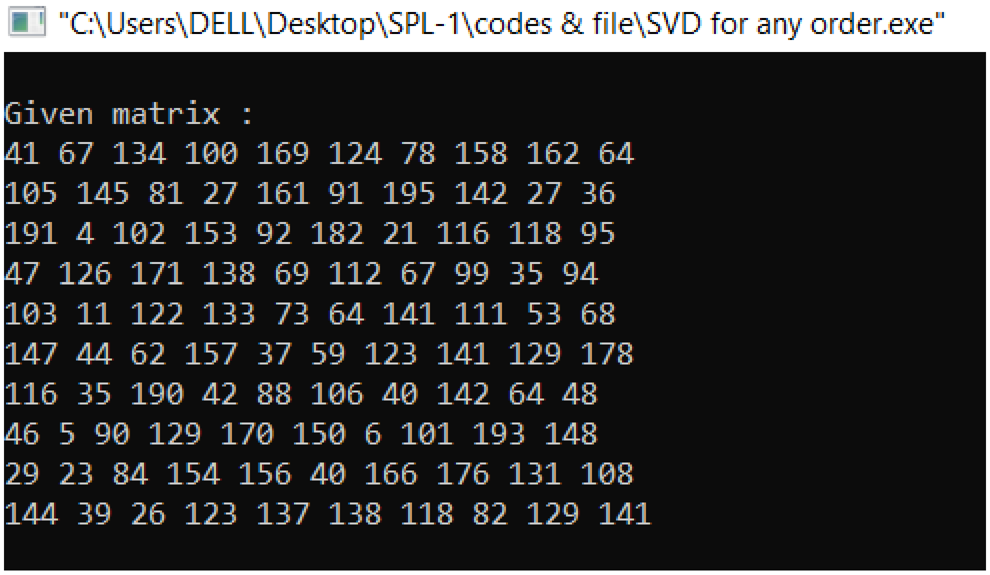
Aт[i][j] = A[j][i] ;



```
void transpose(double P[Dimen+1][Dimen+1],double Q[Dimen+1][Dimen+1])

{
    //double trans_val;
    for(int i=0; i<Dimen; i++)
    {
        for(int j=0; j<Dimen; j++)
        {
            Q[i][j] = P[j][i];

        }
        //cout<<endl;
    }
```

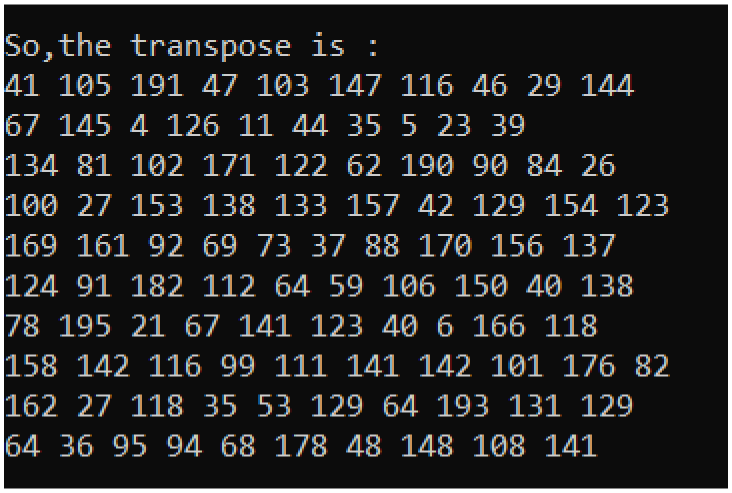
Figure 2: transpose of matrix

The input & output of the function :


"C:\Users\DELL\Desktop\SPL-1\codes & file\SVD for any order.exe"

```
Given matrix :
41 67 134 100 169 124 78 158 162 64
105 145 81 27 161 91 195 142 27 36
191 4 102 153 92 182 21 116 118 95
47 126 171 138 69 112 67 99 35 94
103 11 122 133 73 64 141 111 53 68
147 44 62 157 37 59 123 141 129 178
116 35 190 42 88 106 40 142 64 48
46 5 90 129 170 150 6 101 193 148
29 23 84 154 156 40 166 176 131 108
144 39 26 123 137 138 118 82 129 141
```

Figure 3: read the matrix



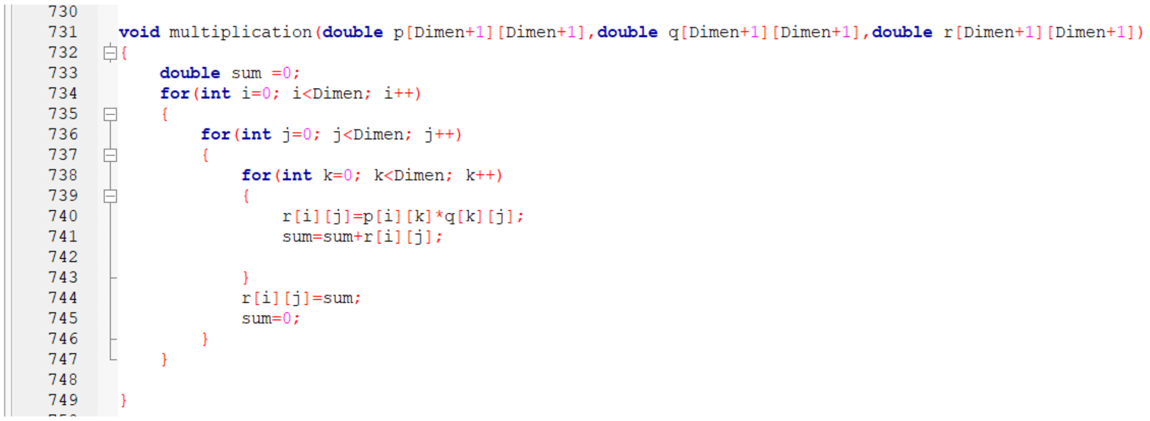
```
So,the transpose is :
41 105 191 47 103 147 116 46 29 144
67 145 4 126 11 44 35 5 23 39
134 81 102 171 122 62 190 90 84 26
100 27 153 138 133 157 42 129 154 123
169 161 92 69 73 37 88 170 156 137
124 91 182 112 64 59 106 150 40 138
78 195 21 67 141 123 40 6 166 118
158 142 116 99 111 141 142 101 176 82
162 27 118 35 53 129 64 193 131 129
64 36 95 94 68 178 48 148 108 141
```

Figure 4: output for transpose matrix

2.3 Multiplication

This function can multiply two different input matrices. In this program, this function multiply A_T with the original matrix A. Here, we use three for loops to compute the multiplication. The main calculative part of this function is : **r[i][j]=p[i][k]*q[k][j];** where **p[i][k]** represent the i'th row &

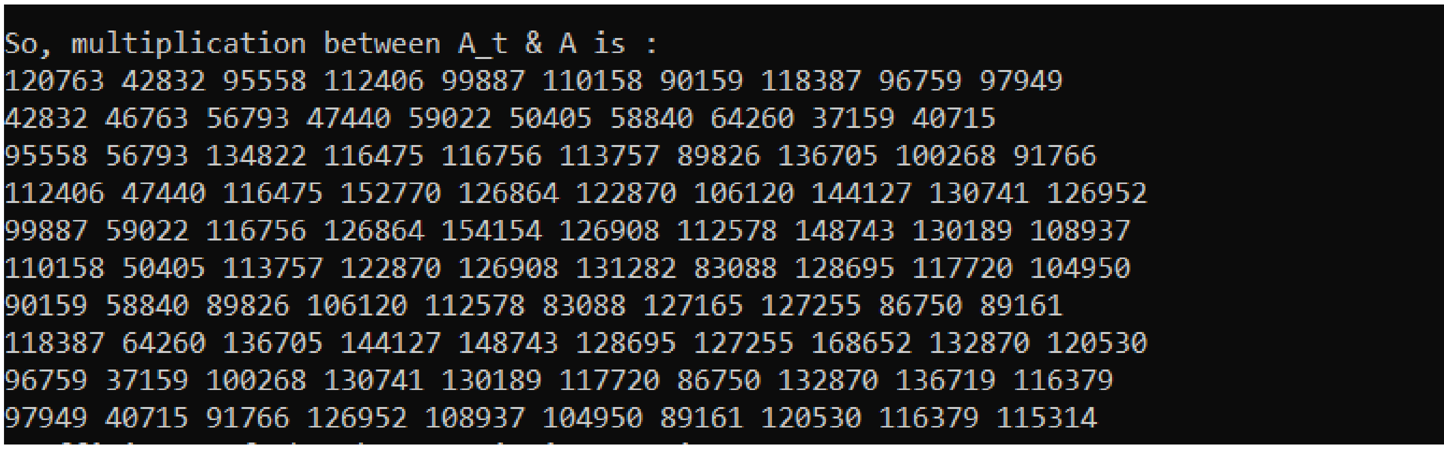
k'th column of A^T & **q[k][j]** represent the k'th row & j'th column of A.



```
void multiplication(double p[Dimen+1][Dimen+1],double q[Dimen+1][Dimen+1],double r[Dimen+1][Dimen+1])
{
    double sum =0;
    for(int i=0; i<Dimen; i++)
    {
        for(int j=0; j<Dimen; j++)
        {
            for(int k=0; k<Dimen; k++)
            {
                r[i][j]=p[i][k]*q[k][j];
                sum=sum+r[i][j];

            }
            r[i][j]=sum;
            sum=0;
        }
    }

}
```

Figure 5: Multiplication

Let, after multiply the resulting matrix W = ($A^T \times$ A).So, the output matrix W, is :

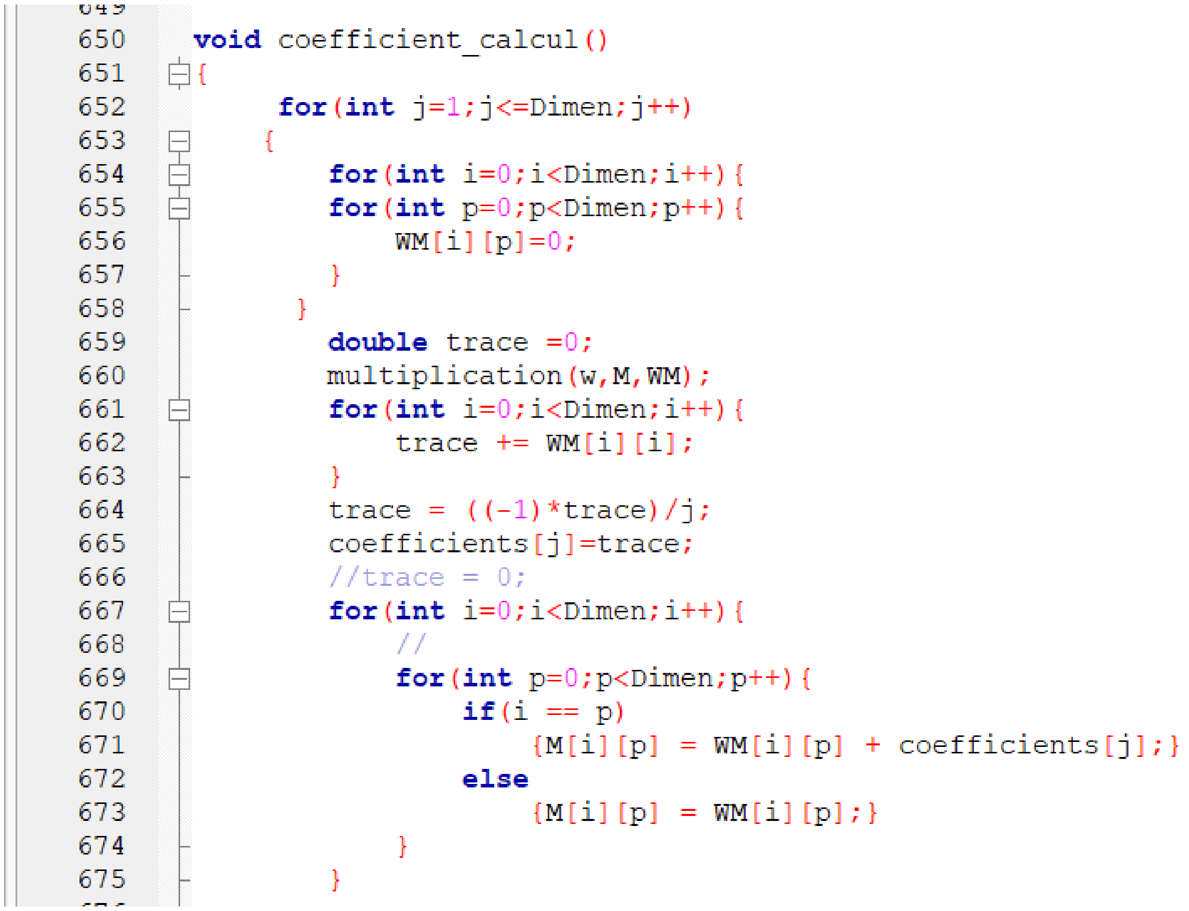


```
So, multiplication between A_t & A is :
120763 42832 95558 112406 99887 110158 90159 118387 96759 97949
42832 46763 56793 47440 59022 50405 58840 64260 37159 40715
95558 56793 134822 116475 116756 113757 89826 136705 100268 91766
112406 47440 116475 152770 126864 122870 106120 144127 130741 126952
99887 59022 116756 126864 154154 126908 112578 148743 130189 108937
110158 50405 113757 122870 126908 131282 83088 128695 117720 104950
90159 58840 89826 106120 112578 83088 127165 127255 86750 89161
118387 64260 136705 144127 148743 128695 127255 168652 132870 120530
96759 37159 100268 130741 130189 117720 86750 132870 136719 116379
97949 40715 91766 126952 108937 104950 89161 120530 116379 115314
```

Figure 6: output resulting matrix after multiplication

2.4 Calculate Coefficients

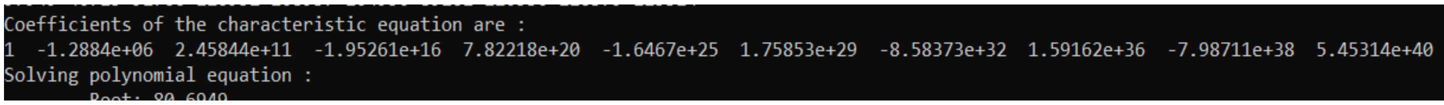
This function calculates the coefficients of the characteristic(polynomial) equation of the matrix which is found after multiplication A^T & A, using the **Faddeev Leverrier Algorithm**. For example if the characteristic equation is **x^3-3x^2+7x-2=0,** then the output of that function is **1,-3,7-2** which are the coefficients of the respective polynomial part. In every iteration we multiply the **W** with an identity matrix **M**. Then calculate the trace of resulting matrix by following that procedure : **trace = $\sum_1^n \sum_1^n WMi,j$;** Then coefficients can be calculated as : **coefficient[i] = (-1)* trace/j ;**



```
void coefficient_calcul()
{
   for(int j=1;j<=Dimen;j++)
  {
    for(int i=0;i<Dimen;i++){
    for(int p=0;p<Dimen;p++){
      WM[i][p]=0;
    }
   }
    double trace =0;
    multiplication(w,M,WM);
    for(int i=0;i<Dimen;i++){
      trace += WM[i][i];
    }
    trace = ((-1)*trace)/j;
    coefficients[j]=trace;
    //trace = 0;
    for(int i=0;i<Dimen;i++){
      //
      for(int p=0;p<Dimen;p++){
        if(i == p)
          {M[i][p] = WM[i][p] + coefficients[j];}
        else
          {M[i][p] = WM[i][p];}
      }
    }
```

Figure 7: calculating coefficients

After apply the Faddeev Leverrier Algorithm for the matrix W, the output is :

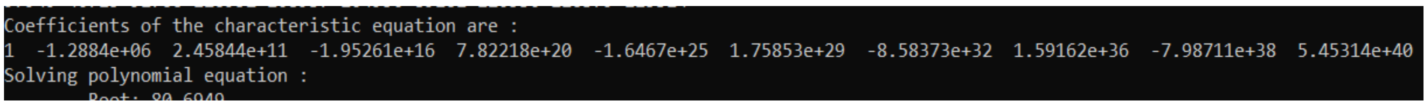


```
Coefficients of the characteristic equation are :
1  -1.2884e+06  2.45844e+11  -1.95261e+16  7.82218e+20  -1.6467e+25  1.75853e+29  -8.58373e+32  1.59162e+36  -7.98711e+38  5.45314e+40
Solving polynomial equation :
        Root: 80.6949
```

Figure 8: output for Faddeev Leverrier

2.5 Bairstow Algorithm

In that part we implement the Bairstow Algorithm to find the roots(**eigenvalues**) of the characteristic equation. Here, we put the coefficients that we found from the above(part-2.4) implementation and then we will find the output of the roots of the polynomial equation.
Input for this part of the project:

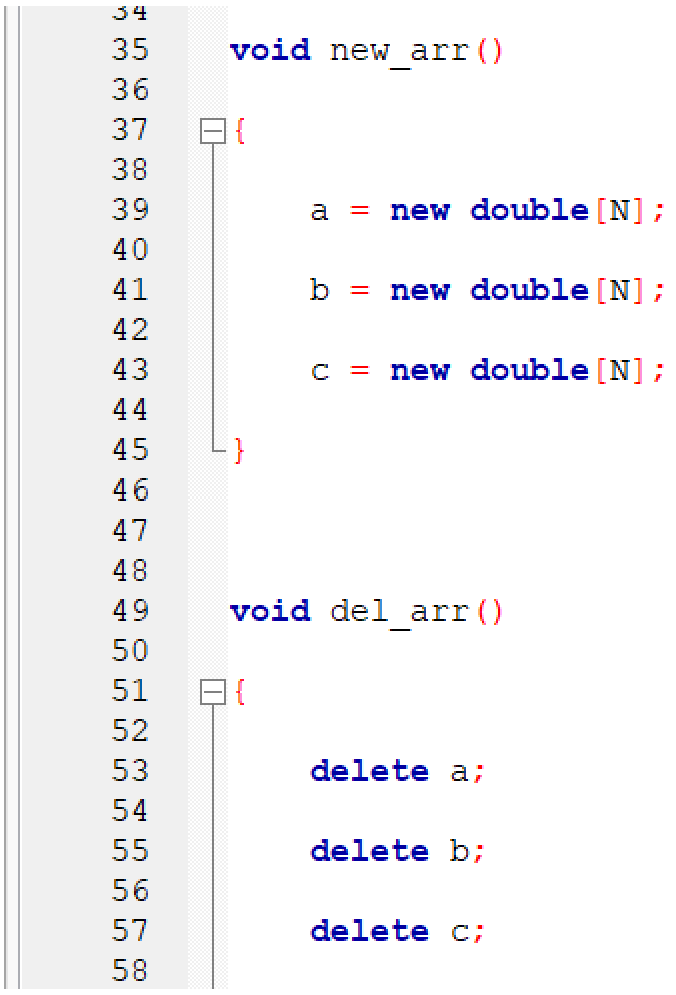


```
Coefficients of the characteristic equation are :
1  -1.2884e+06  2.45844e+11  -1.95261e+16  7.82218e+20  -1.6467e+25  1.75853e+29  -8.58373e+32  1.59162e+36  -7.98711e+38  5.45314e+40
Solving polynomial equation :
```

Figure 9: input for Bairstow

2.5.1 Create & Delete array

Here, we are allowed to create new arrays. And in the second part we deallocate those arrays.

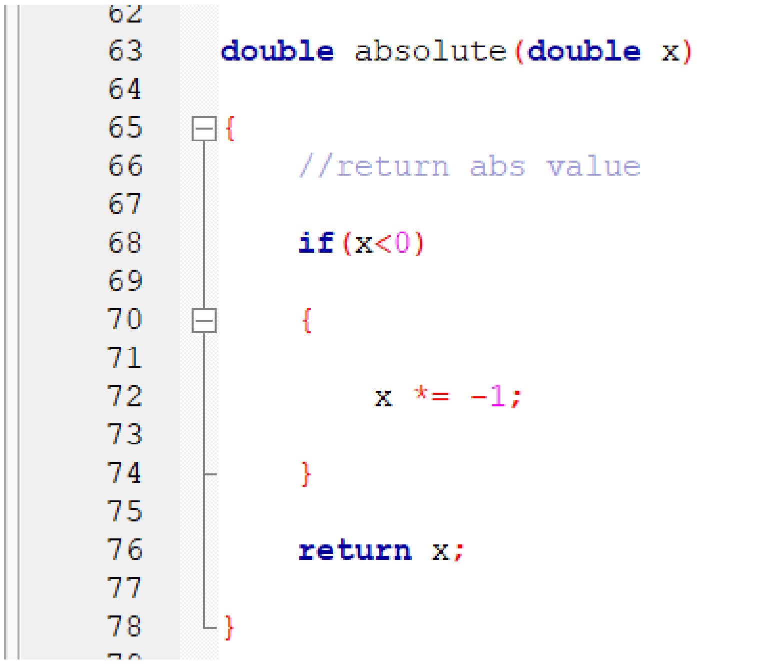


```
void new_arr()

{

    a = new double[N];

    b = new double[N];

    c = new double[N];

}



void del_arr()

{

    delete a;

    delete b;

    delete c;

```

Figure 10: create & delete array

2.5.2 Calculate absolute value

It will return the absolute value of any real number.

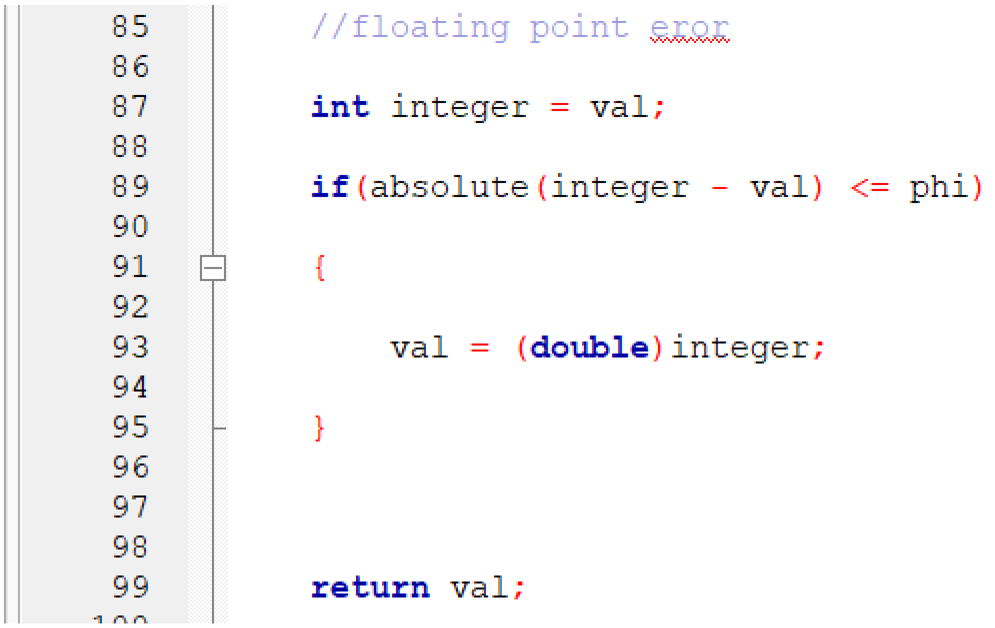


```
double absolute(double x)

{

    //return abs value

    if(x<0)

    {

        x *= -1;

    }

    return x;

}
```

Figure 11: calculating absolute value

2.5.3 Remove Error

When a double value is very close to a round figure, then it will return only the integer part of the value.

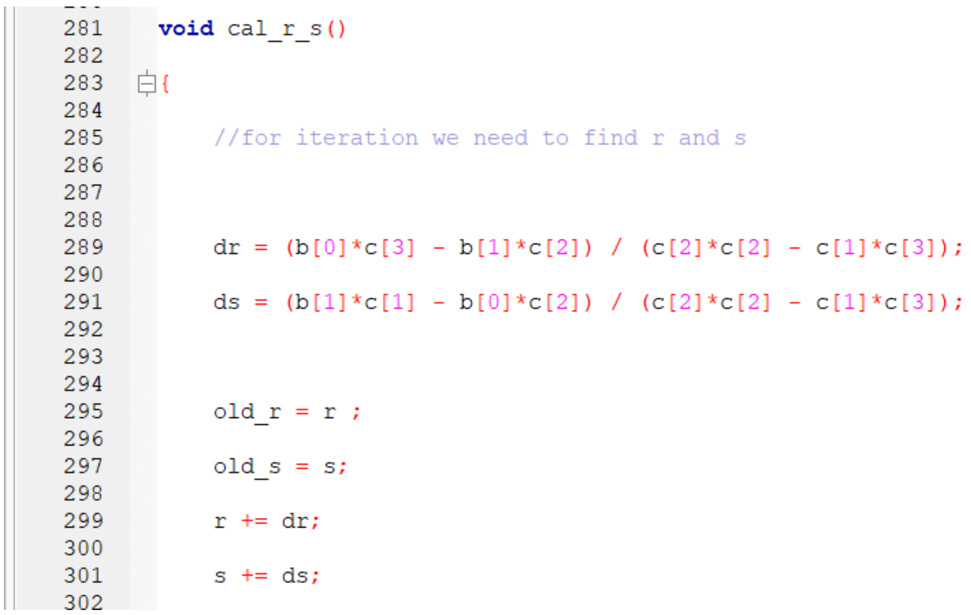


```
    //floating point eror

    int integer = val;

    if(absolute(integer - val) <= phi)

    {

        val = (double)integer;

    }



    return val;
```

Figure 12: remove error

2.5.4 Calculation r & s

For every iteration we need to find the value of r & s of the quadratic factor **x^2+rx+s** which initially starts with the value 0.1 & 0.1 respectively. And then every iteration they will change following procedure:

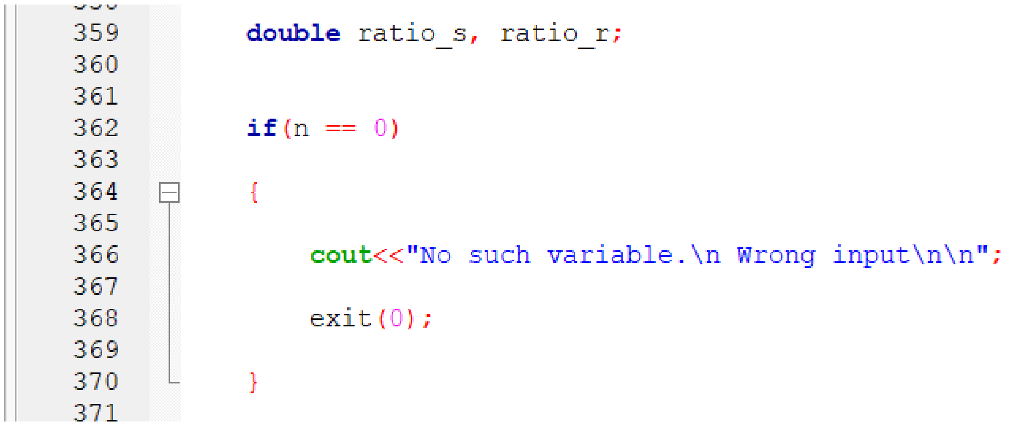


```
void cal_r_s()

{

    //for iteration we need to find r and s



    dr = (b[0]*c[3] - b[1]*c[2]) / (c[2]*c[2] - c[1]*c[3]);

    ds = (b[1]*c[1] - b[0]*c[2]) / (c[2]*c[2] - c[1]*c[3]);



    old_r = r ;

    old_s = s;

    r += dr;

    s += ds;

```

Figure 13: calculation r & s

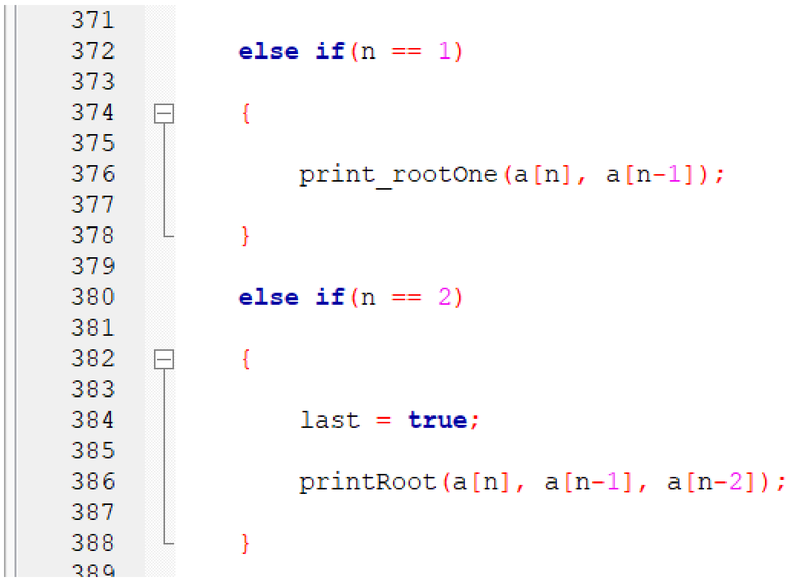
2.5.5 Root Finding

Basically, in this part we call two different functions **printroot** & **printroot_one** depending on the value of dimension(n) to calculate roots.



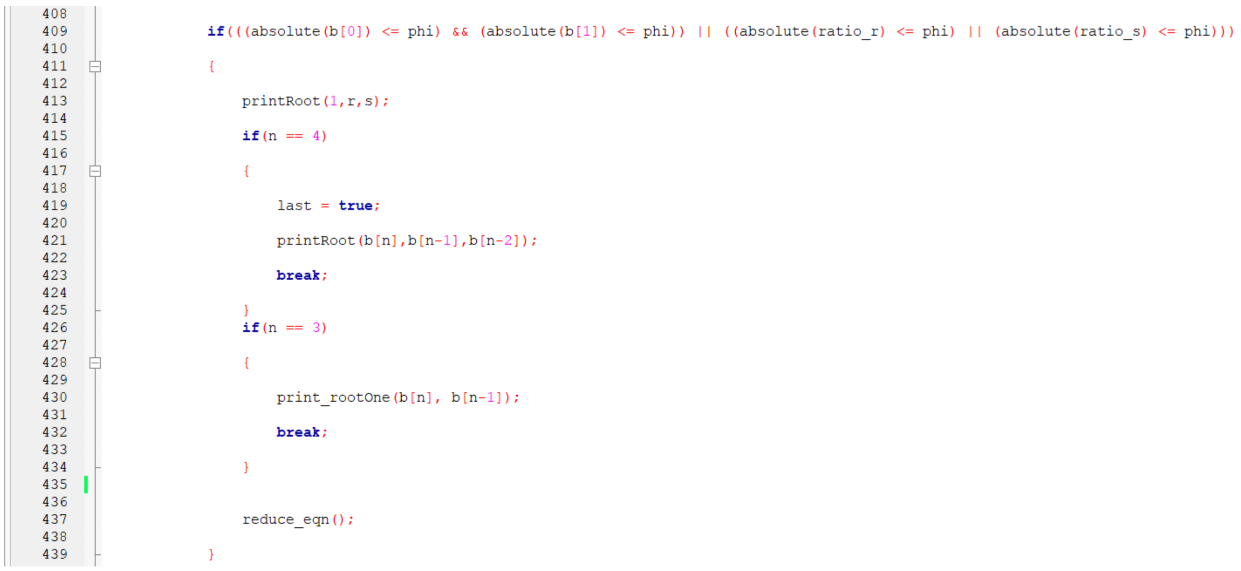
```
    double ratio_s, ratio_r;

    if(n == 0)

    {

        cout<<"No such variable.\n Wrong input\n\n";

        exit(0);

    }

```

Figure 14: root finding-1



```
else if(n == 1)

{

    print_rootOne(a[n], a[n-1]);

}

else if(n == 2)

{

    last = true;

    printRoot(a[n], a[n-1], a[n-2]);

}
```

Figure 15: root finding-2

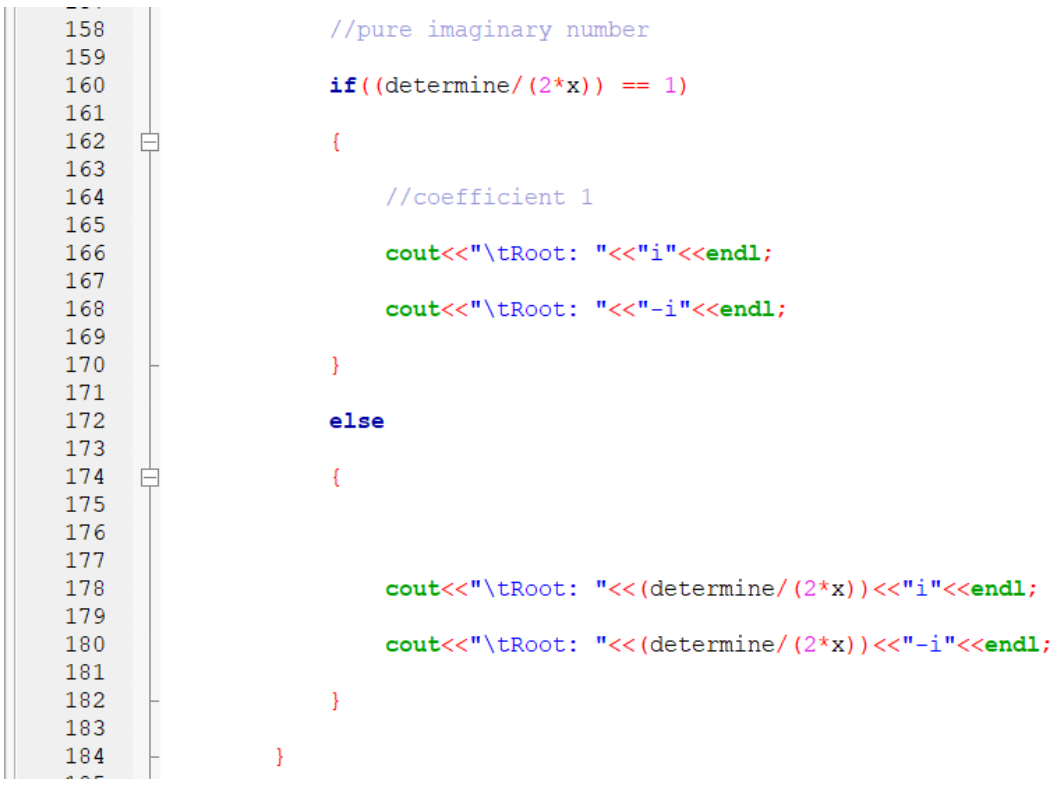


```
if(((absolute(b[0]) <= phi) && (absolute(b[1]) <= phi)) || ((absolute(ratio_r) <= phi) || (absolute(ratio_s) <= phi)))

{

    printRoot(1,r,s);

    if(n == 4)

    {

        last = true;

        printRoot(b[n],b[n-1],b[n-2]);

        break;

    }
    if(n == 3)

    {

        print_rootOne(b[n], b[n-1]);

        break;

    }


    reduce_eqn();

}
```

Figure 16:root finding-3

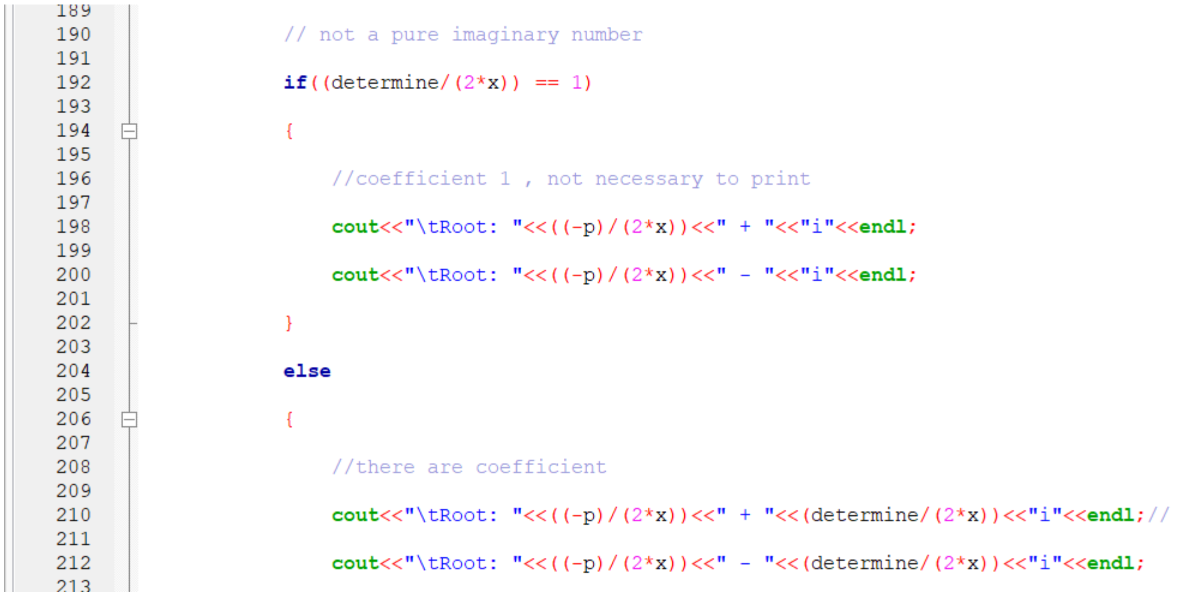
2.5.5.1 PrintRoot

First of all we determine whether the root is complex or real. For this we calculate a double value **determine = r*r - 4*r*s.** If **determine** <0, then the roots are complex. Again, whether the roots are pure imaginary or not, it is defined with the value of **r.**



```
        //pure imaginary number

        if((determine/(2*x)) == 1)

        {

            //coefficient 1

            cout<<"\tRoot: "<<"i"<<endl;

            cout<<"\tRoot: "<<"-i"<<endl;

        }

        else

        {



            cout<<"\tRoot: "<<(determine/(2*x))<<"i"<<endl;

            cout<<"\tRoot: "<<(determine/(2*x))<<"-i"<<endl;

        }

    }
```

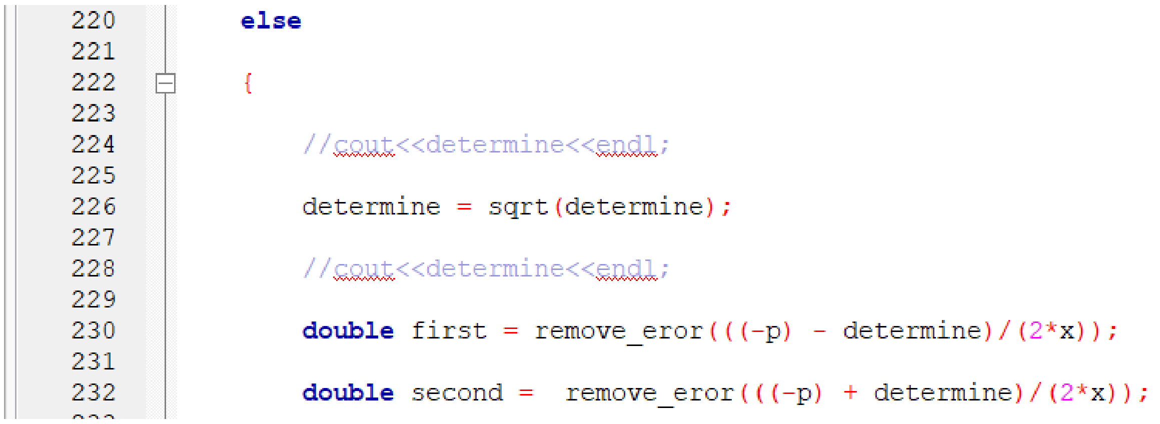
Figure 17:print root



```
    // not a pure imaginary number

    if((determine/(2*x)) == 1)

    {

        //coefficient 1 , not necessary to print

        cout<<"\tRoot: "<<((-p)/(2*x))<<" + "<<"i"<<endl;

        cout<<"\tRoot: "<<((-p)/(2*x))<<" - "<<"i"<<endl;

    }

    else

    {

        //there are coefficient

        cout<<"\tRoot: "<<((-p)/(2*x))<<" + "<<(determine/(2*x))<<"i"<<endl;//

        cout<<"\tRoot: "<<((-p)/(2*x))<<" - "<<(determine/(2*x))<<"i"<<endl;
```

Figure 18:printroot-2

Again, when **determine** >0, then the roots are real.

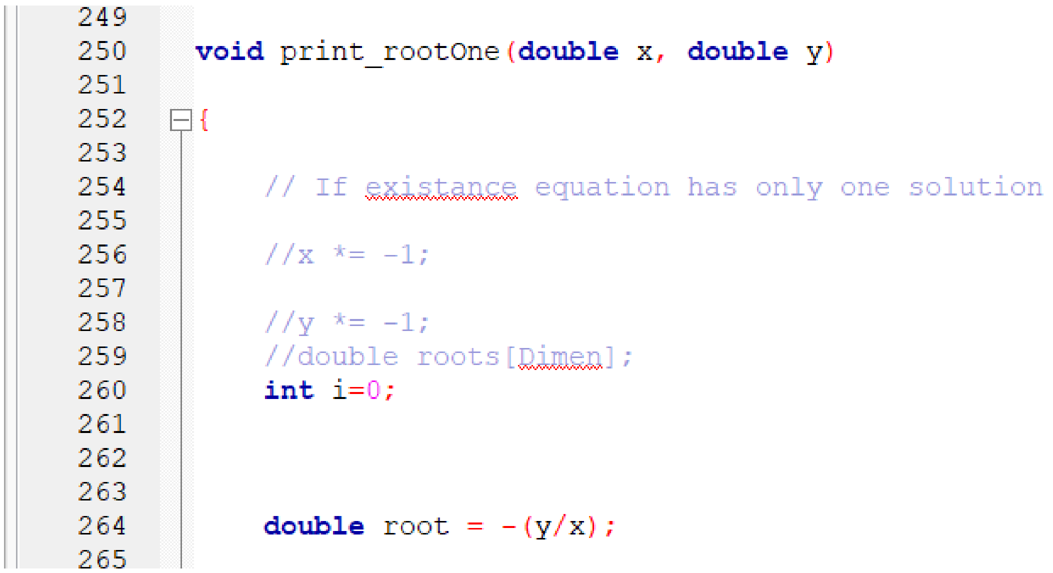


```
    else

    {

        //cout<<determine<<endl;

        determine = sqrt(determine);

        //cout<<determine<<endl;

        double first = remove_eror(((-p) - determine)/(2*x));

        double second =  remove_eror(((-p) + determine)/(2*x));
```

Figure 19:printroot-3

2.5.5.2 PrintRoot_one

If existance equation has only one solution, then that function might be helpfull.

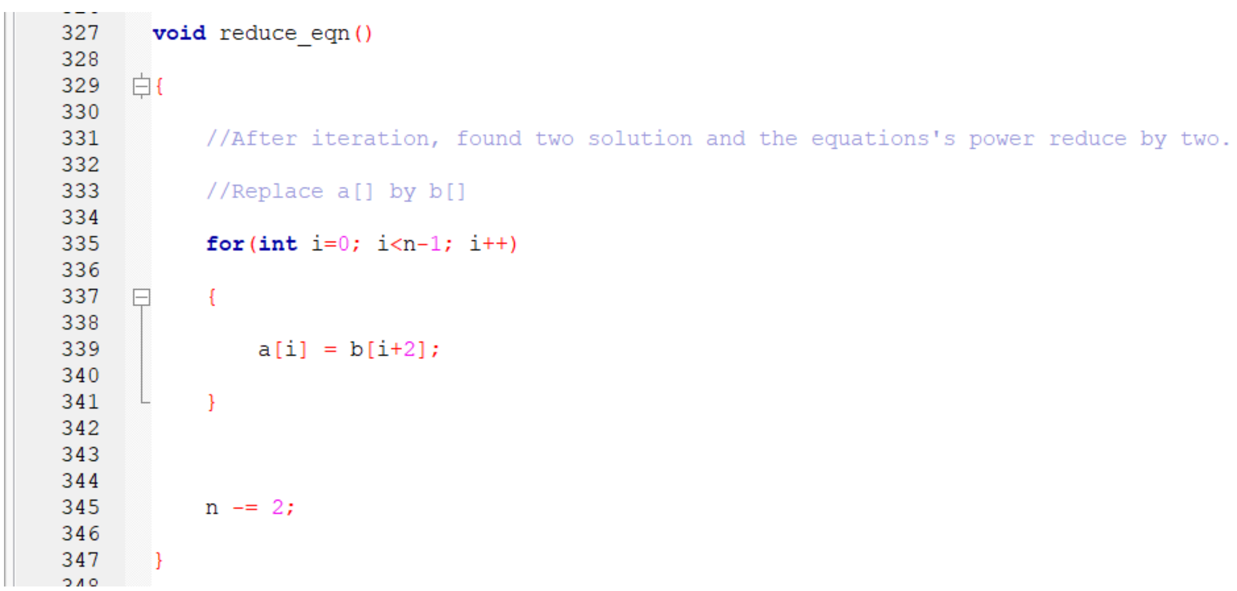


```

void print_rootOne(double x, double y)

{

    // If existance equation has only one solution

    //x *= -1;

    //y *= -1;
    //double roots[Dimen];
    int i=0;



    double root = -(y/x);

```

Figure 20: printroot_one

2.5.6 Reduce Power

After every iteration if we find two roots dividing with x^2+rx+s, then we have to reduce the equation. For this we may apply following procedure:

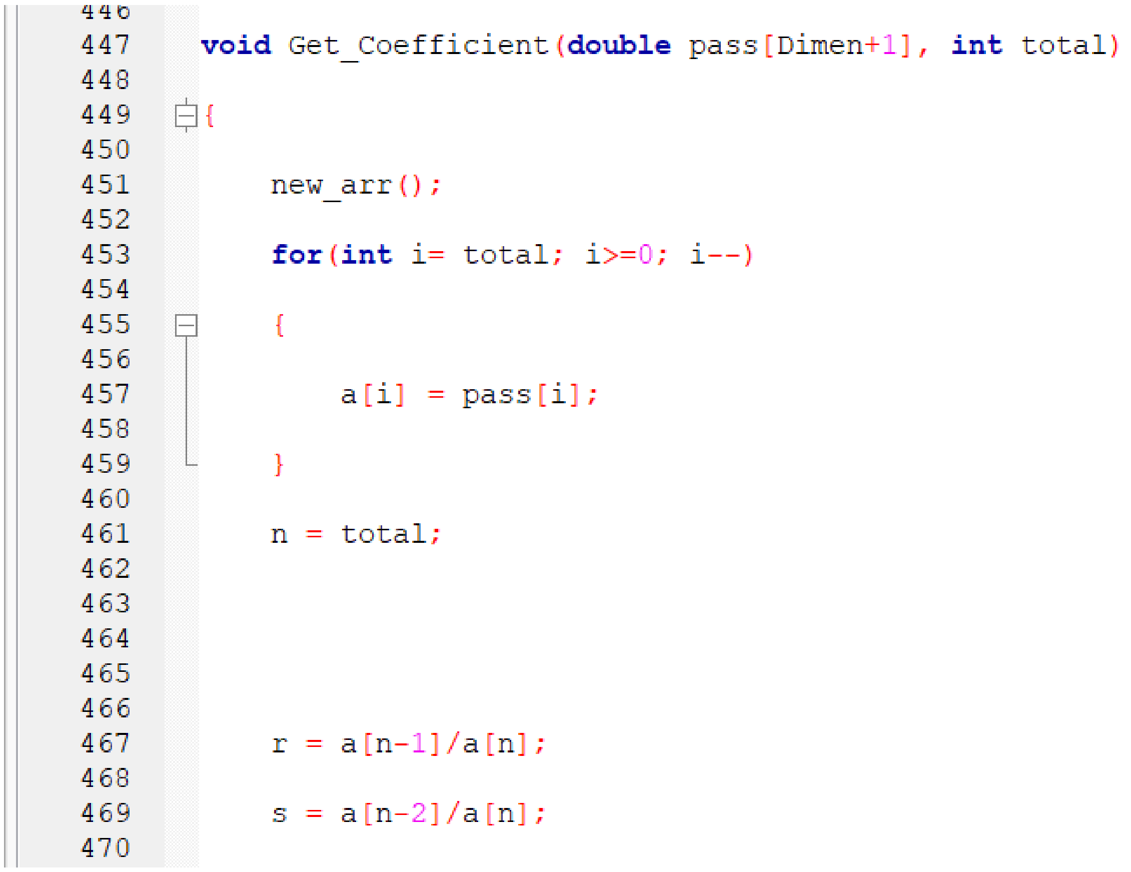


```
void reduce_eqn()
{

    //After iteration, found two solution and the equations's power reduce by two.

    //Replace a[] by b[]

    for(int i=0; i<n-1; i++)

    {

        a[i] = b[i+2];

    }


    n -= 2;

}
```

Figure 21:reduce equation

2.5.7 Get Coefficients

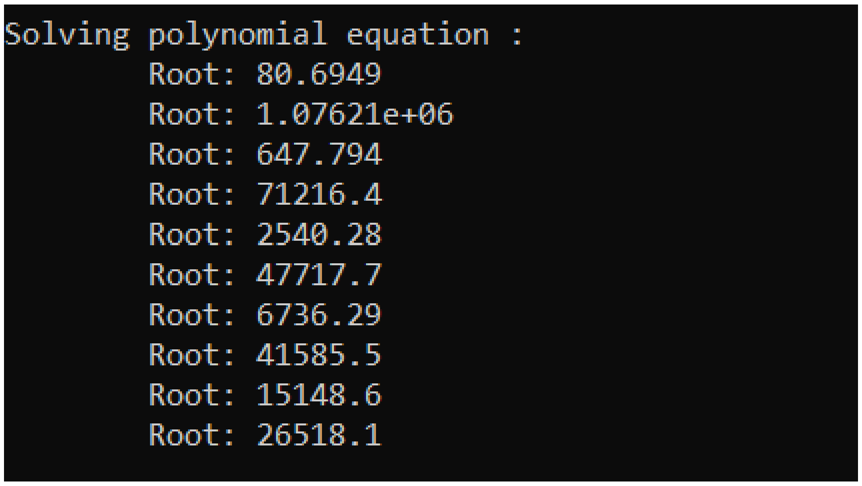
In every iteration we have to define the coefficients of the polynomial equation using the following procedure. Here, **r = a[n-1]/a[n];**
s = a[n-2]a[n];



```
void Get_Coefficient(double pass[Dimen+1], int total)

{

    new_arr();

    for(int i= total; i>=0; i--)

    {

        a[i] = pass[i];

    }

    n = total;




    r = a[n-1]/a[n];

    s = a[n-2]/a[n];

```

Figure 22: get coefficients

If we successfully complete the above procedure that means **Bairstow Algorithm,** then it will print all the roots(**eigenvalues**) of the characteristic equation. So, after put the coefficients(part-2.5) of W, then the output is:



```
Solving polynomial equation :
        Root: 80.6949
        Root: 1.07621e+06
        Root: 647.794
        Root: 71216.4
        Root: 2540.28
        Root: 47717.7
        Root: 6736.29
        Root: 41585.5
        Root: 15148.6
        Root: 26518.1
```

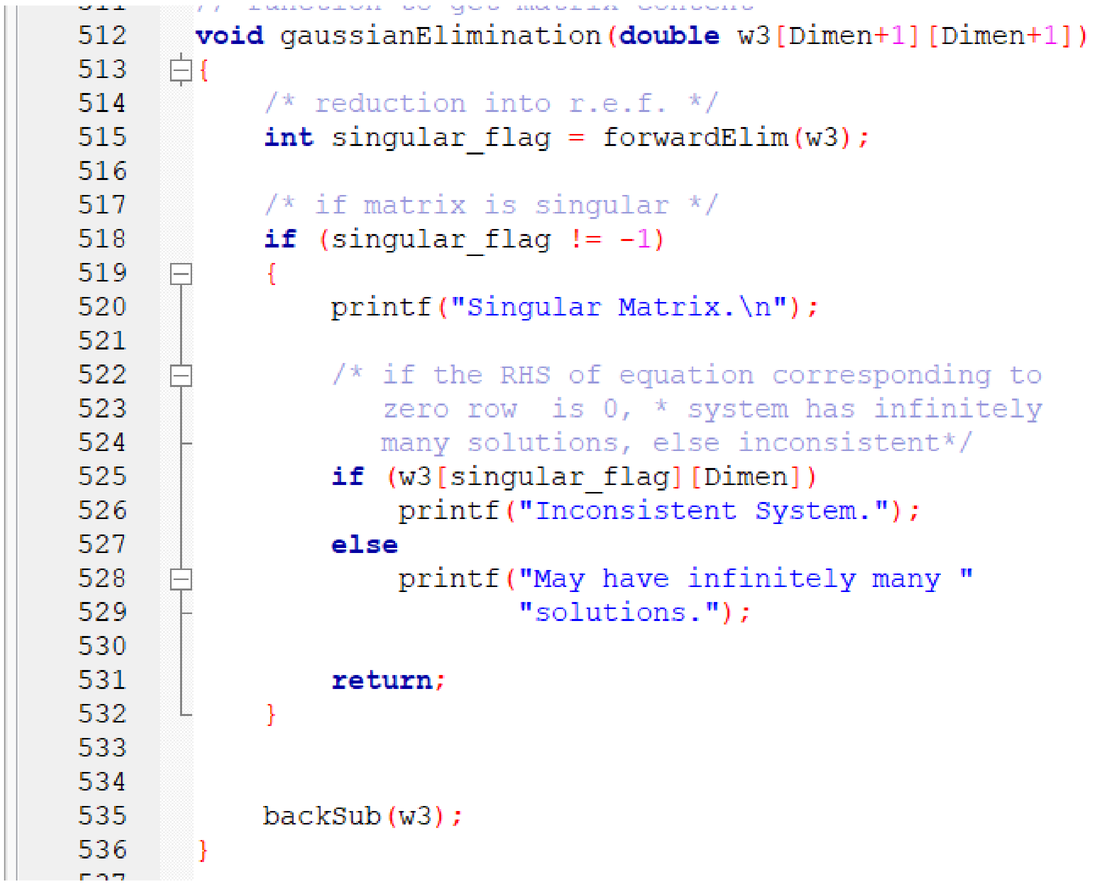
Figure 23: output for bairstow

2.6 Gauss Elimination

Here, we used Gaussian elimination to calculate the eigenvectors for each eigenvalue. That will perform using **backward substitution**. It will take the eigenvalues found from **part-2.5** as an input and then compute eigenvectors for each eigenvalue.

2.6.1 Gauss Elimination Method

This function will be called for every eigenvalue. It will starts with a matrix W3, which can be found for every eigenvalue **ev[i]** following process: **W3[i][i] = W[i][i] - ev[i]**
Then we apply Gaussian Elimination on **W3.**
It will call the **forward_elimination** & **backward_substitution** method.



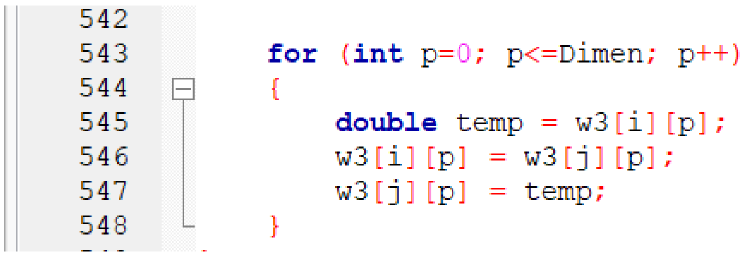
```
void gaussianElimination(double w3[Dimen+1][Dimen+1])
{
    /* reduction into r.e.f. */
    int singular_flag = forwardElim(w3);

    /* if matrix is singular */
    if (singular_flag != -1)
    {
        printf("Singular Matrix.\n");

        /* if the RHS of equation corresponding to
           zero row  is 0, * system has infinitely
           many solutions, else inconsistent*/
        if (w3[singular_flag][Dimen])
            printf("Inconsistent System.");
        else
            printf("May have infinitely many "
                   "solutions.");

        return;
    }


    backSub(w3);
}
```

Figure 24: Gauss elimination

2.6.2 Swap Row

When the principal diagonal element is zero, it denotes that the matrix is singular. Then we have to swap the greatest value row with the current row. Following equations are used to swap rows:

temp = W3[i][p]; W3[i][p] = W3[j][p]; W3[j][p] = temp ;

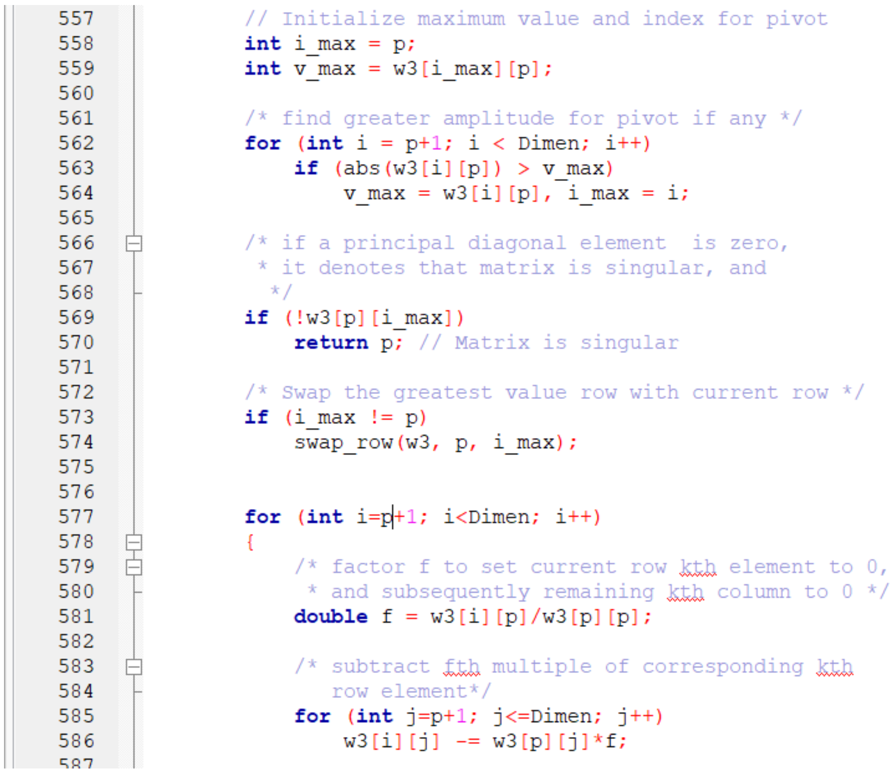


```
    for (int p=0; p<=Dimen; p++)
    {
        double temp = w3[i][p];
        w3[i][p] = w3[j][p];
        w3[j][p] = temp;
    }
```

Figure 25: swap row

2.6.3 Forward Elimination

For every iteration we have to define a **pivot** and have to find greater amplitude for pivot if any.

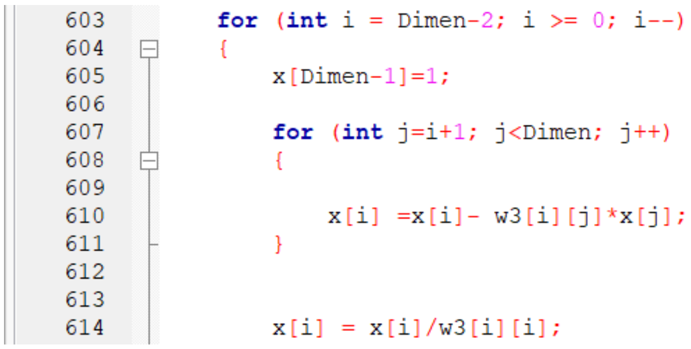


```
// Initialize maximum value and index for pivot
int i_max = p;
int v_max = w3[i_max][p];

/* find greater amplitude for pivot if any */
for (int i = p+1; i < Dimen; i++)
    if (abs(w3[i][p]) > v_max)
        v_max = w3[i][p], i_max = i;

/* if a principal diagonal element  is zero,
 * it denotes that matrix is singular, and
  */
if (!w3[p][i_max])
    return p; // Matrix is singular

/* Swap the greatest value row with current row */
if (i_max != p)
    swap_row(w3, p, i_max);


for (int i=p+1; i<Dimen; i++)
{
    /* factor f to set current row kth element to 0,
     * and subsequently remaining kth column to 0 */
    double f = w3[i][p]/w3[p][p];

    /* subtract fth multiple of corresponding kth
       row element*/
    for (int j=p+1; j<=Dimen; j++)
        w3[i][j] -= w3[p][j]*f;
```

Figure 26:forward elimination

2.6.4 Backward Substitution

Basically, here we calculate the eigenvectors for each eigenvalue. We initialize the last element as **1** and then compute others element following backward elimination:
x[i] = x[i] - W3[i][j] * x[j]; x[i] = x[i] /W3[i][i] ; for every i'th row & j'th column.

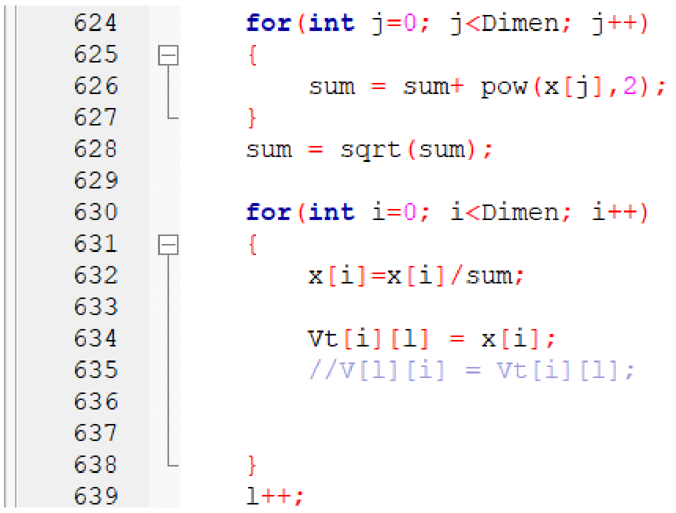


```
for (int i = Dimen-2; i >= 0; i--)
{
    x[Dimen-1]=1;

    for (int j=i+1; j<Dimen; j++)
    {

        x[i] =x[i]- w3[i][j]*x[j];
    }


    x[i] = x[i]/w3[i][i];
```

Figure 27: backward substitution

And then we normalize the eigenvectors by following that procedure:

$$\mathbf{xi} = xi/\sqrt{\sum_{1}^{n}(xj^2)}$$



```
        for(int j=0; j<Dimen; j++)
        {
            sum = sum+ pow(x[j],2);
        }
        sum = sqrt(sum);

        for(int i=0; i<Dimen; i++)
        {
            x[i]=x[i]/sum;

            Vt[i][l] = x[i];
            //V[l][i] = Vt[i][l];


        }
        l++;
```

Figure 28: backward substitution continued

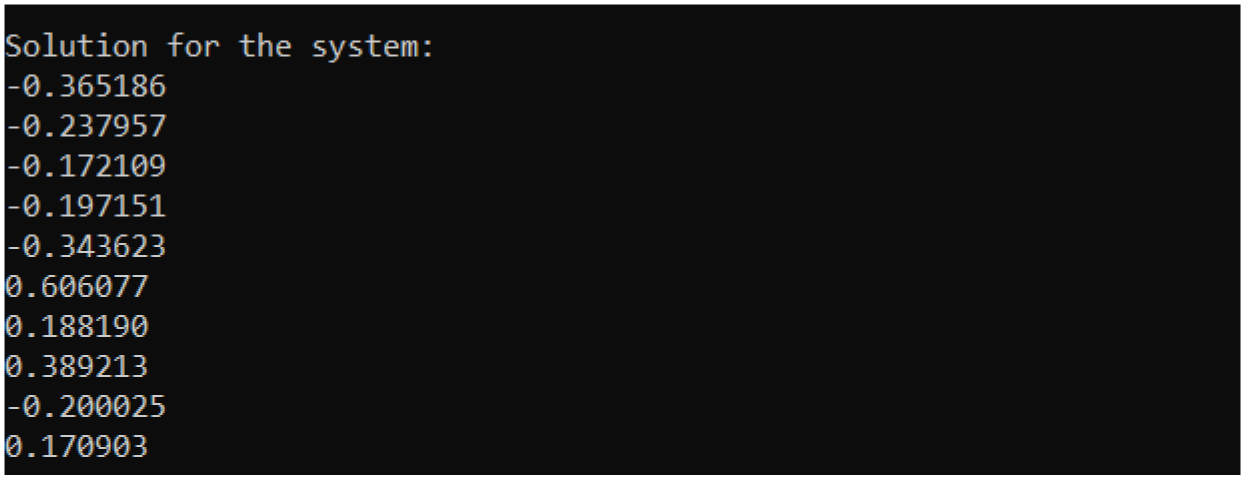
After normalize the eigenvectors, we find the transpose matrix of the right singular factor **V**, which is the core element of the **SVD** as SVD stands for:

$$\mathbf{SVD(A) = U*\Sigma*V_T}$$

After successfully completing the above(**part-2.6**) section, we are allowed to find eigenvectors.

Let's see the sample input and output for the above implementation:

If the eigenvalue is **80.6949,** then the eigenvectors(normalized) for this is:



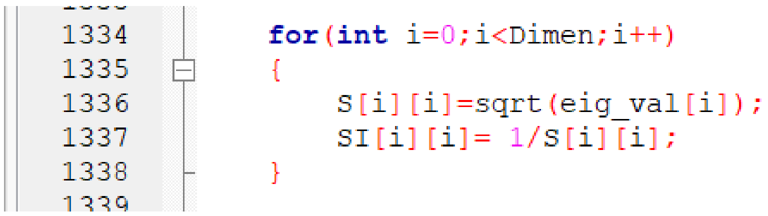
```
Solution for the system:
-0.365186
-0.237957
-0.172109
-0.197151
-0.343623
0.606077
0.188190
0.389213
-0.200025
0.170903
```

Figure 29: output for gauss elimination

2.7 Calculate U & Σ

Singular matrix **Σ** is found from the square root of the sorted eigenvalues. And put them as a diagonal element of that matrix.

Here, **Σ**ii **= sqrt(eigenvalue[i]) ; (Σ**ii)$^{-1}$ **= 1/Σ**ii ;



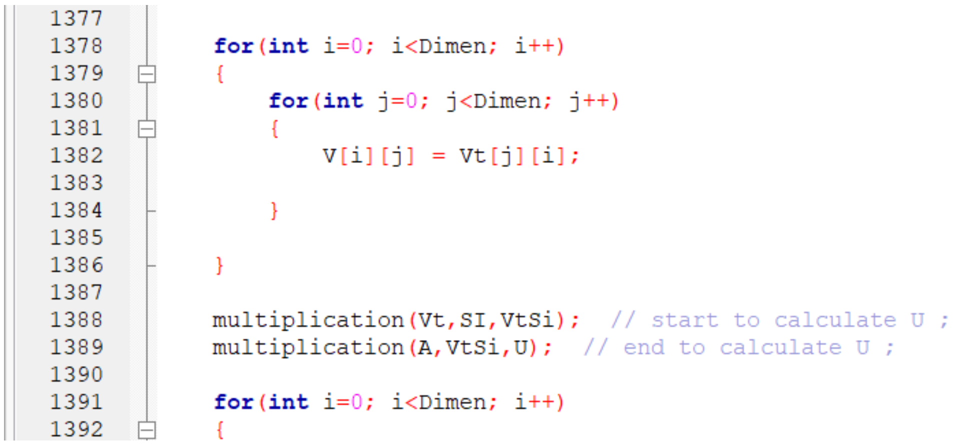
```
    for(int i=0;i<Dimen;i++)
    {
        S[i][i]=sqrt(eig_val[i]);
        SI[i][i]= 1/S[i][i];
    }
```

Figure 30: calculate U & S

And then the left singular factor **U** can be found from the following procedure:

U = A*Vт*Si ;

where **Si** is the inverse of the singular matrix **S.** We showed in the above figure the procedure how we compute **Si**.



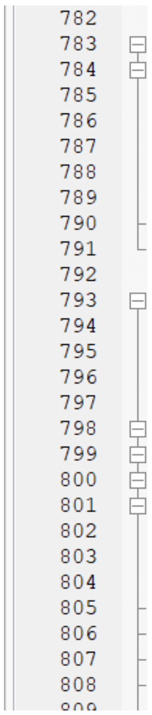
```

    for(int i=0; i<Dimen; i++)
    {
        for(int j=0; j<Dimen; j++)
        {
            V[i][j] = Vt[j][i];

        }

    }

    multiplication(Vt,SI,VtSi);  // start to calculate U ;
    multiplication(A,VtSi,U);  // end to calculate U ;

    for(int i=0; i<Dimen; i++)
    {
```

Figure 31: formulae for U

2.8 SVD For Higher Order

When the dimension of the data matrix is very large then we apply the **Jacobi Rotation** method to calculate the eigenvalues. First of all we

find the largest value of the matrix except the diagonal elements. Then store its index to compute the rest of the part.
Let’s see the details procedure of that particular part of:

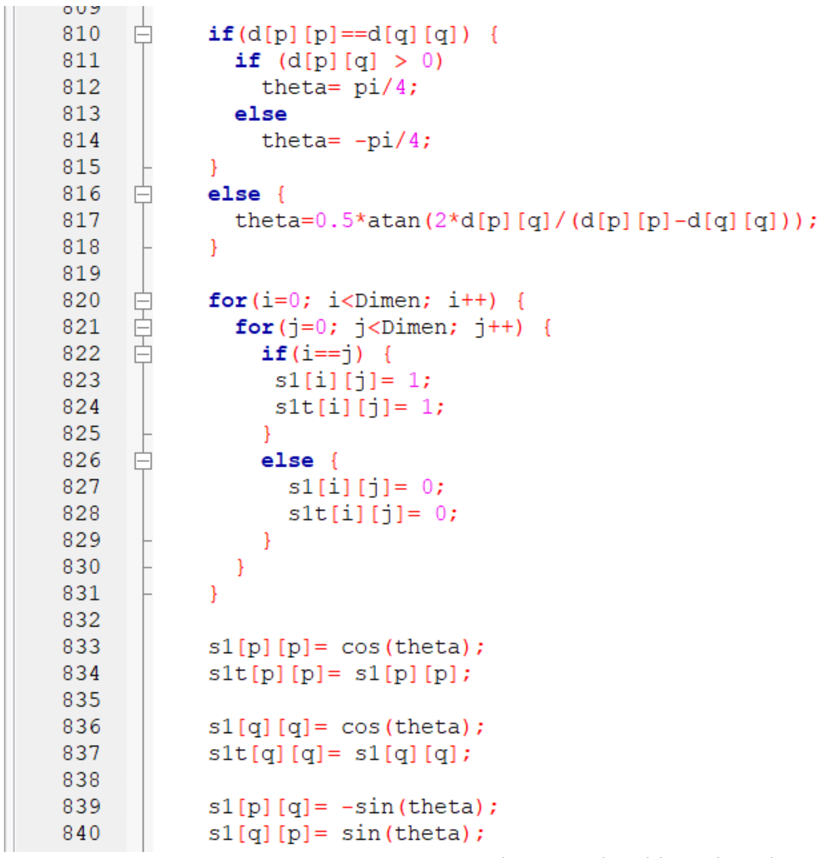


```
for(i=0; i<Dimen; i++) {
  for(j=0; j<Dimen; j++) {
    d[i][j]= w[i][j];
    if(i==j)
      s[i][j]= 1;
    else
      s[i][j]= 0;
  }
}

do {
  flag= 0;
  p=0; q=1;
  max= fabs(d[p][q]);

  for(i=0; i<Dimen; i++) {
    for(j=0; j<Dimen; j++) {
      if(i!=j) {
        if (max < fabs(d[i][j])) {
          max= fabs(d[i][j]);
          p= i;
          q= j;
        }
      }
    }
  }
```

Figure 32: jacobi iteration

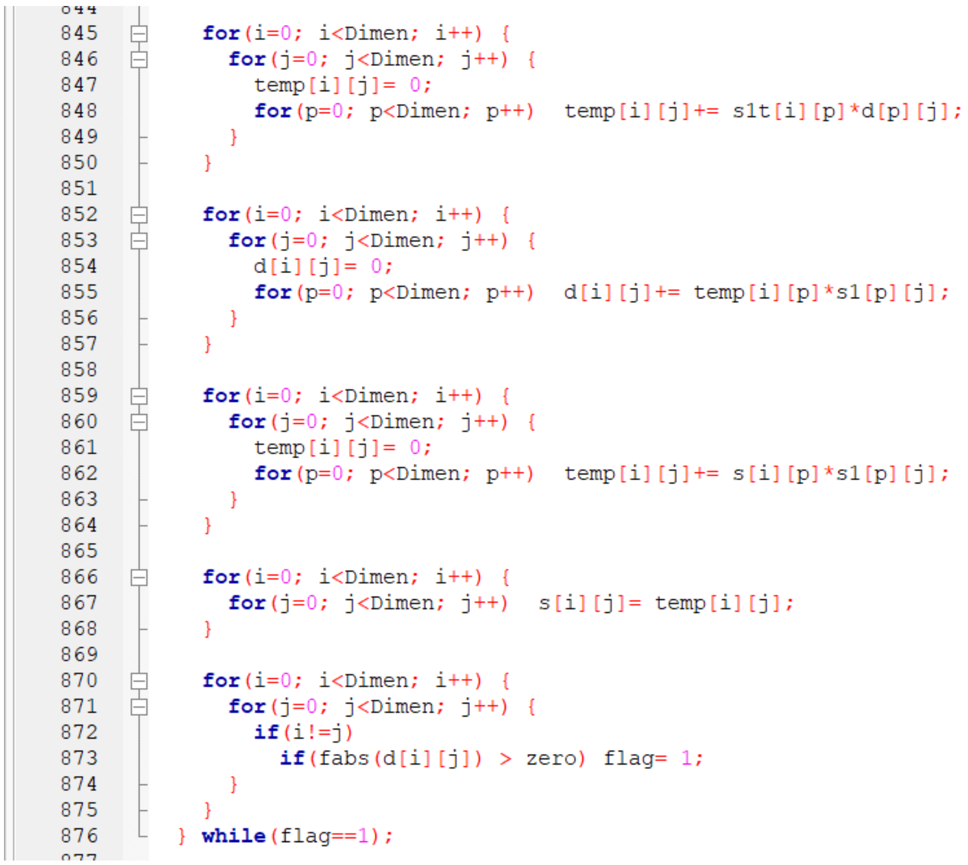
When **d[p][p]** is equal to **d[q][q]**, then **theta = π/4 or -π/4**.

if (**d[p][q]** > 0)
    theta = π/4
else
    theta = -π/4



```
    if(d[p][p]==d[q][q]) {
      if (d[p][q] > 0)
        theta= pi/4;
      else
        theta= -pi/4;
    }
    else {
      theta=0.5*atan(2*d[p][q]/(d[p][p]-d[q][q]));
    }

    for(i=0; i<Dimen; i++) {
      for(j=0; j<Dimen; j++) {
        if(i==j) {
         s1[i][j]= 1;
         s1t[i][j]= 1;
        }
        else {
          s1[i][j]= 0;
          s1t[i][j]= 0;
        }
      }
    }

    s1[p][p]= cos(theta);
    s1t[p][p]= s1[p][p];

    s1[q][q]= cos(theta);
    s1t[q][q]= s1[q][q];

    s1[p][q]= -sin(theta);
    s1[q][p]= sin(theta);
```

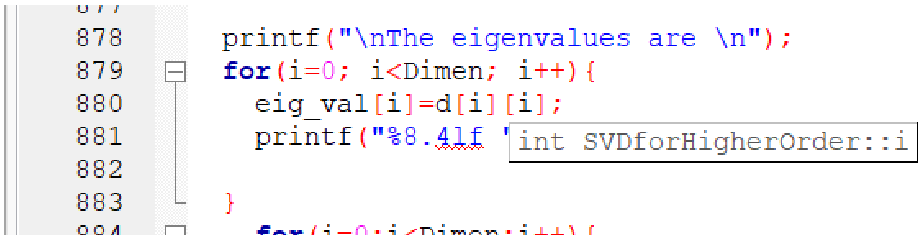
Figure 33: jacobi continued



```
    for(i=0; i<Dimen; i++) {
      for(j=0; j<Dimen; j++) {
        temp[i][j]= 0;
        for(p=0; p<Dimen; p++)   temp[i][j]+= s1t[i][p]*d[p][j];
      }
    }

    for(i=0; i<Dimen; i++) {
      for(j=0; j<Dimen; j++) {
        d[i][j]= 0;
        for(p=0; p<Dimen; p++)   d[i][j]+= temp[i][p]*s1[p][j];
      }
    }

    for(i=0; i<Dimen; i++) {
      for(j=0; j<Dimen; j++) {
        temp[i][j]= 0;
        for(p=0; p<Dimen; p++)   temp[i][j]+= s[i][p]*s1[p][j];
      }
    }

    for(i=0; i<Dimen; i++) {
      for(j=0; j<Dimen; j++)   s[i][j]= temp[i][j];
    }

    for(i=0; i<Dimen; i++) {
      for(j=0; j<Dimen; j++) {
        if(i!=j)
          if(fabs(d[i][j]) > zero)  flag= 1;
      }
    }
  } while(flag==1);
```

Figure 34: jacobi continued

After **Jacobi iteration**, the **eigenvalues** will be found from the diagonal element of the **d** matrix.



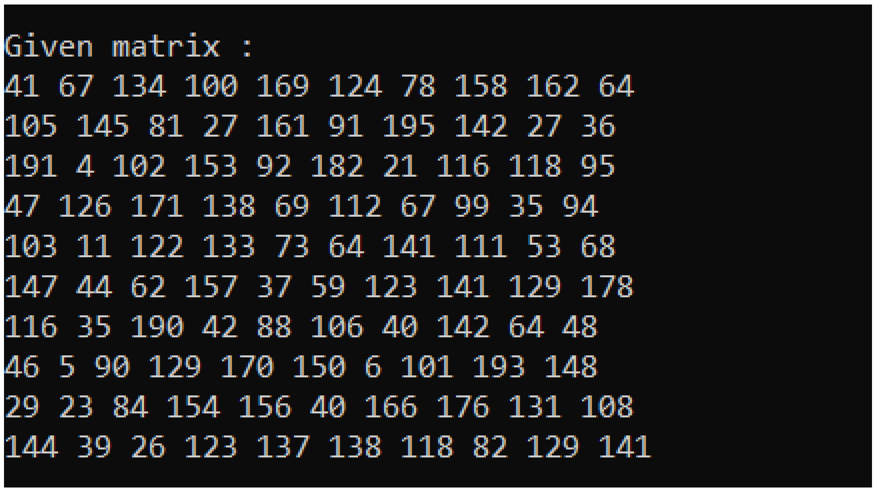
```
printf("\nThe eigenvalues are \n");
for(i=0; i<Dimen; i++){
  eig_val[i]=d[i][i];
  printf("%8.4lf '
                    int SVDforHigherOrder::i

}
```

Figure 35: jacobi eigenvalue

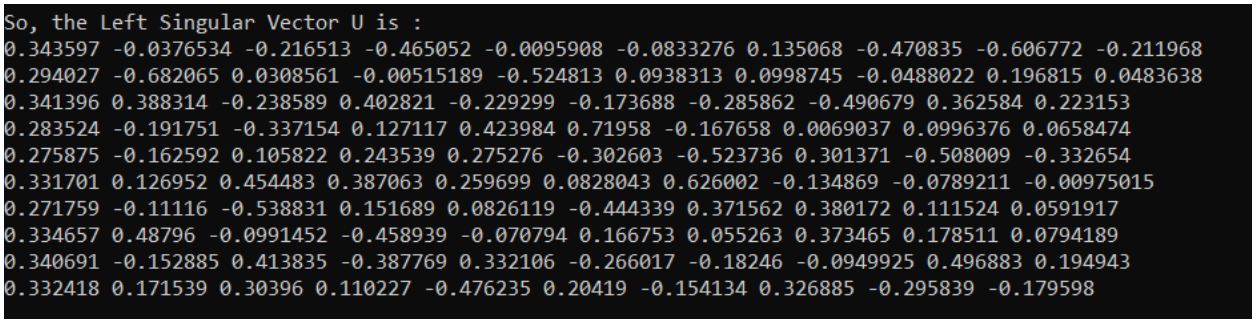
After finding the eigenvalues we calculate the **eigenvectors, singular matrix, left & right singular factor** as a same procedure that we discussed above(**part- 2.6.4, 2.7**).

So, by calculating the **U, Σ & V**, the procedure of computing **SVD** is completed.
Let's see the sample input & output of our computed **SVD** :

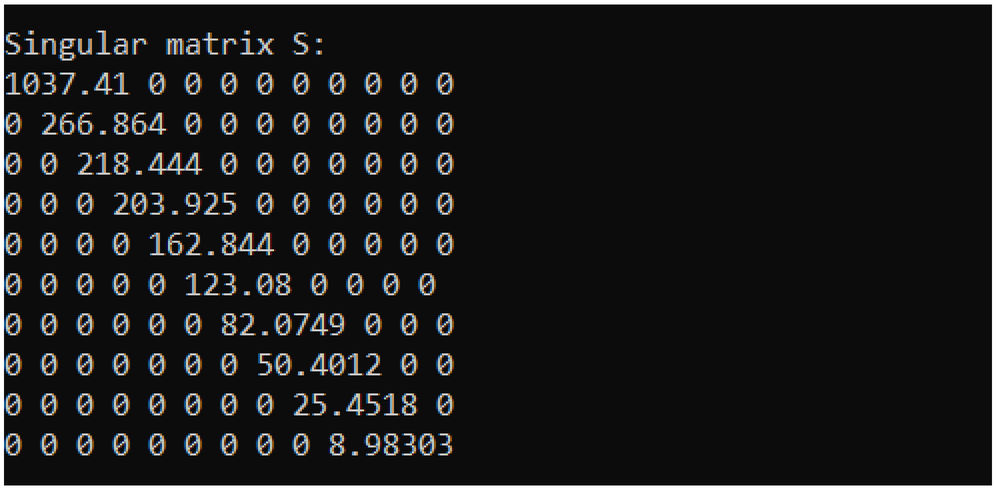
Input & Output:



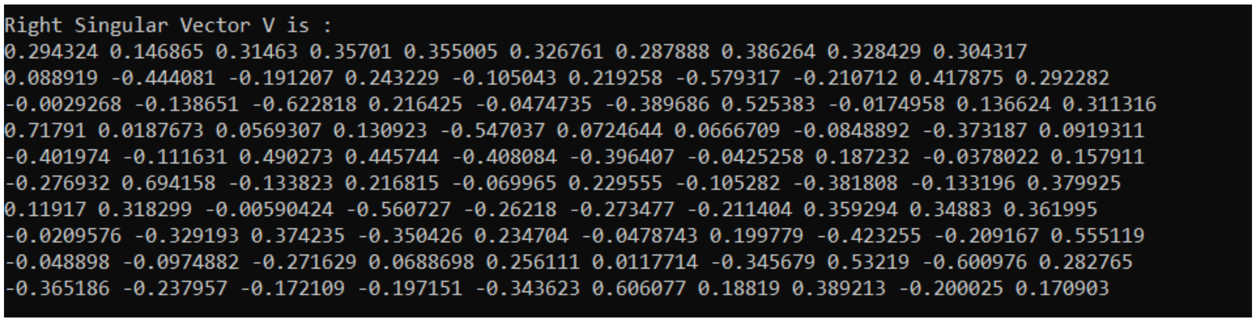
```
Given matrix :
41 67 134 100 169 124 78 158 162 64
105 145 81 27 161 91 195 142 27 36
191 4 102 153 92 182 21 116 118 95
47 126 171 138 69 112 67 99 35 94
103 11 122 133 73 64 141 111 53 68
147 44 62 157 37 59 123 141 129 178
116 35 190 42 88 106 40 142 64 48
46 5 90 129 170 150 6 101 193 148
29 23 84 154 156 40 166 176 131 108
144 39 26 123 137 138 118 82 129 141
```



```
So, the Left Singular Vector U is :
0.343597 -0.0376534 -0.216513 -0.465052 -0.0095908 -0.0833276 0.135068 -0.470835 -0.606772 -0.211968
0.294027 -0.682065 0.0308561 -0.00515189 -0.524813 0.0938313 0.0998745 -0.0488022 0.196815 0.0483638
0.341396 0.388314 -0.238589 0.402821 -0.229299 -0.173688 -0.285862 -0.490679 0.362584 0.223153
0.283524 -0.191751 -0.337154 0.127117 0.423984 0.71958 -0.167658 0.0069037 0.0996376 0.0658474
0.275875 -0.162592 0.105822 0.243539 0.275276 -0.302603 -0.523736 0.301371 -0.508009 -0.332654
0.331701 0.126952 0.454483 0.387063 0.259699 0.0828043 0.626002 -0.134869 -0.0789211 -0.00975015
0.271759 -0.11116 -0.538831 0.151689 0.0826119 -0.444339 0.371562 0.380172 0.111524 0.0591917
0.334657 0.48796 -0.0991452 -0.458939 -0.070794 0.166753 0.055263 0.373465 0.178511 0.0794189
0.340691 -0.152885 0.413835 -0.387769 0.332106 -0.266017 -0.18246 -0.0949925 0.496883 0.194943
0.332418 0.171539 0.30396 0.110227 -0.476235 0.20419 -0.154134 0.326885 -0.295839 -0.179598
```



```
Singular matrix S:
1037.41 0 0 0 0 0 0 0 0 0
0 266.864 0 0 0 0 0 0 0 0
0 0 218.444 0 0 0 0 0 0 0
0 0 0 203.925 0 0 0 0 0 0
0 0 0 0 162.844 0 0 0 0 0
0 0 0 0 0 123.08 0 0 0 0
0 0 0 0 0 0 82.0749 0 0 0
0 0 0 0 0 0 0 50.4012 0 0
0 0 0 0 0 0 0 0 25.4518 0
0 0 0 0 0 0 0 0 0 8.98303
```



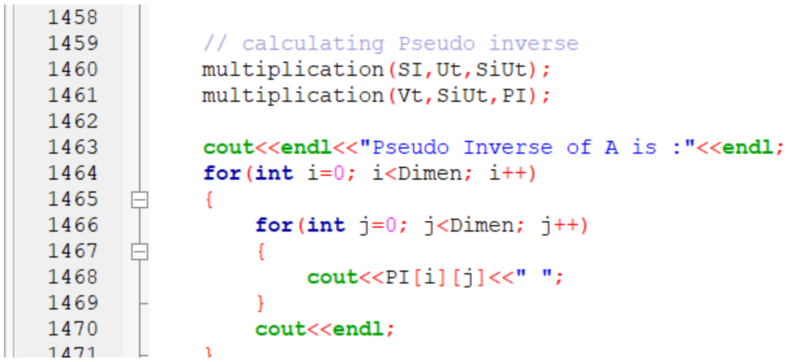
```
Right Singular Vector V is :
0.294324 0.146865 0.31463 0.35701 0.355005 0.326761 0.287888 0.386264 0.328429 0.304317
0.088919 -0.444081 -0.191207 0.243229 -0.105043 0.219258 -0.579317 -0.210712 0.417875 0.292282
-0.0029268 -0.138651 -0.622818 0.216425 -0.0474735 -0.389686 0.525383 -0.0174958 0.136624 0.311316
0.71791 0.0187673 0.0569307 0.130923 -0.547037 0.0724644 0.0666709 -0.0848892 -0.373187 0.0919311
-0.401974 -0.111631 0.490273 0.445744 -0.408084 -0.396407 -0.0425258 0.187232 -0.0378022 0.157911
-0.276932 0.694158 -0.133823 0.216815 -0.069965 0.229555 -0.105282 -0.381808 -0.133196 0.379925
0.11917 0.318299 -0.00590424 -0.560727 -0.26218 -0.273477 -0.211404 0.359294 0.34883 0.361995
-0.0209576 -0.329193 0.374235 -0.350426 0.234704 -0.0478743 0.199779 -0.423255 -0.209167 0.555119
-0.048898 -0.0974882 -0.271629 0.0688698 0.256111 0.0117714 -0.345679 0.53219 -0.600976 0.282765
-0.365186 -0.237957 -0.172109 -0.197151 -0.343623 0.606077 0.18819 0.389213 -0.200025 0.170903
```

Figure 36: Input & output for SVD

2.9 Pseudo Inverse

The Pseudo Inverse of a matrix is a generalization of the inverse for matrices that may not have an inverse. It allows us to solve systems of equations even when the matrix is not invertible. Following procedure can be used to compute Pseudo Inverse :

$$PI = V_T * SI * U_T ;$$

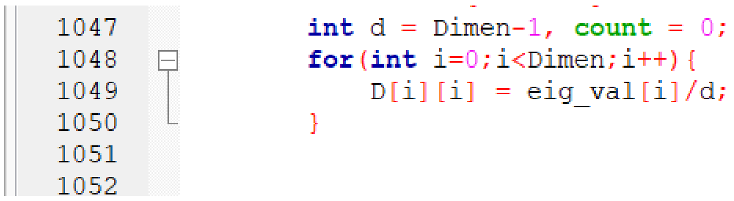


```
// calculating Pseudo inverse
multiplication(SI,Ut,SiUt);
multiplication(Vt,SiUt,PI);

cout<<endl<<"Pseudo Inverse of A is :"<<endl;
for(int i=0; i<Dimen; i++)
{
    for(int j=0; j<Dimen; j++)
    {
        cout<<PI[i][j]<<" ";
    }
    cout<<endl;
}
```

Figure 37: Pseudo Inverse

2.10 Leverage Score Sampling

In order to ignore the outlier in the dataset, Leverage Score Sampling is a better way than others. This method computes the leverage score for every row of left singular factor, **U.** And then identifies the highest leverage scores. At last it returns the corresponding row of the original matrix **A**, which is the representative sample of the original data matrix containing the most important features only.
At first create the diagonal matrix D, since **D[i][i] = eigenvalue[i]/d;** where **d = Dimension-1.**

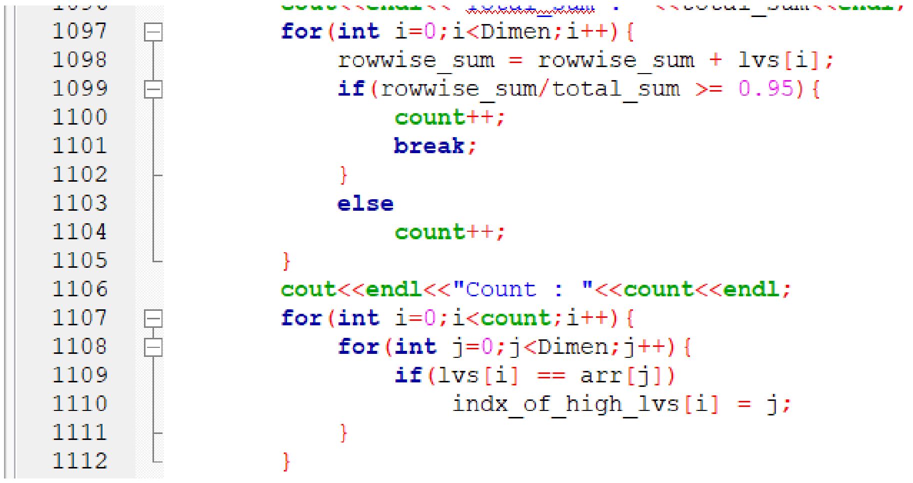


```
int d = Dimen-1, count = 0;
for(int i=0;i<Dimen;i++){
    D[i][i] = eig_val[i]/d;
}
```

Figure 38: Leverage score

Then we calculate the leverage scores following procedure:

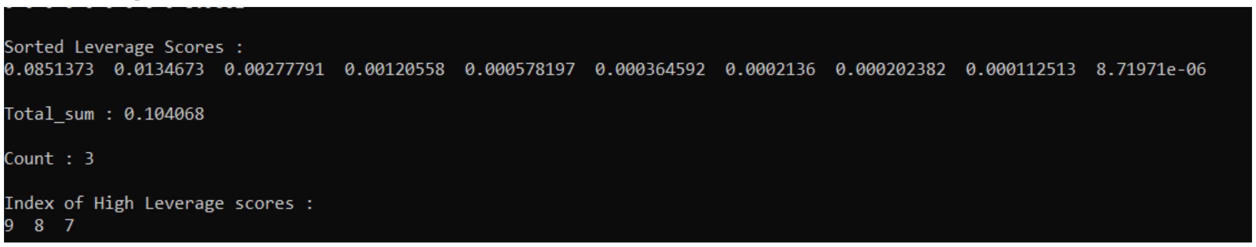
```
for(int i=0;i<Dimen;i++){
        double sum = 0;
    for(int j=0;j<Dimen;j++){
        sum = sum + pow(U[i][j],2);
    }
     lvs[i] = sum;

     lvs[i] = lvs[i] / D[i][i];
```



```
for(int i=0;i<Dimen;i++){
    rowwise_sum = rowwise_sum + lvs[i];
    if(rowwise_sum/total_sum >= 0.95){
        count++;
        break;
    }
    else
        count++;
}
cout<<endl<<"Count : "<<count<<endl;
for(int i=0;i<count;i++){
    for(int j=0;j<Dimen;j++){
        if(lvs[i] == arr[j])
            indx_of_high_lvs[i] = j;
    }
}
```

Figure 39: finding index of highest leverage scores

Leverage scores:



```
Sorted Leverage Scores :
0.0851373  0.0134673  0.00277791  0.00120558  0.000578197  0.000364592  0.0002136  0.000202382  0.000112513  8.71971e-06

Total_sum : 0.104068

Count : 3

Index of High Leverage scores :
9  8  7
```

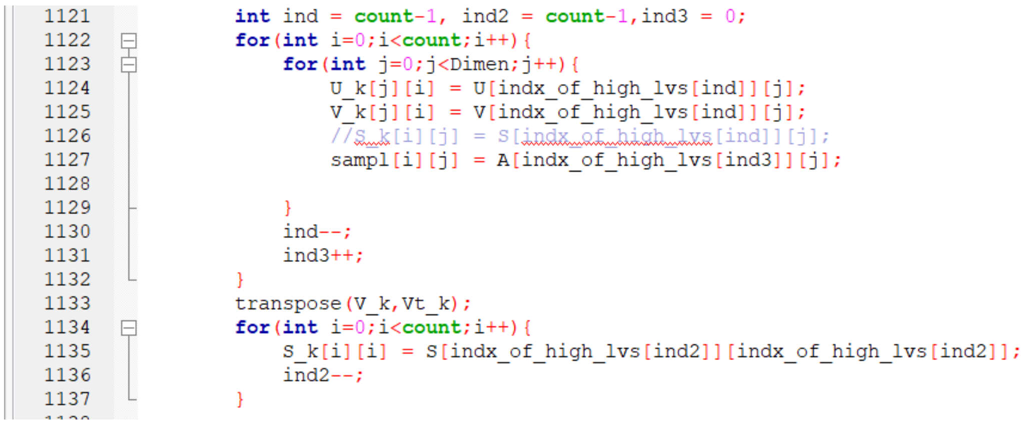
Figure 40: Output for leverage scores

2.11 Matrix Projection

Projection matrix is the matrix that represents the original matrix which can be created from only sample rows. From the sample row we compute the corresponding **UK,ΣK** & **Vk** for projection matrix.

Projection Matrix :

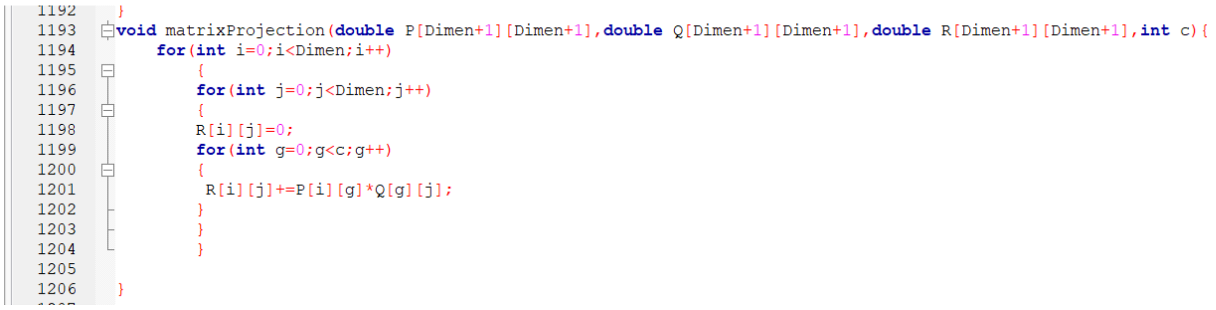
$$\mathbf{Ak = UK*\Sigma K*Vk^{T}\ ;}$$



```
        int ind = count-1, ind2 = count-1,ind3 = 0;
        for(int i=0;i<count;i++){
            for(int j=0;j<Dimen;j++){
                U_k[j][i] = U[indx_of_high_lvs[ind]][j];
                V_k[j][i] = V[indx_of_high_lvs[ind]][j];
                //S_k[i][j] = S[indx_of_high_lvs[ind]][j];
                sampl[i][j] = A[indx_of_high_lvs[ind3]][j];

            }
            ind--;
            ind3++;
        }
        transpose(V_k,Vt_k);
        for(int i=0;i<count;i++){
            S_k[i][i] = S[indx_of_high_lvs[ind2]][indx_of_high_lvs[ind2]];
            ind2--;
        }
```

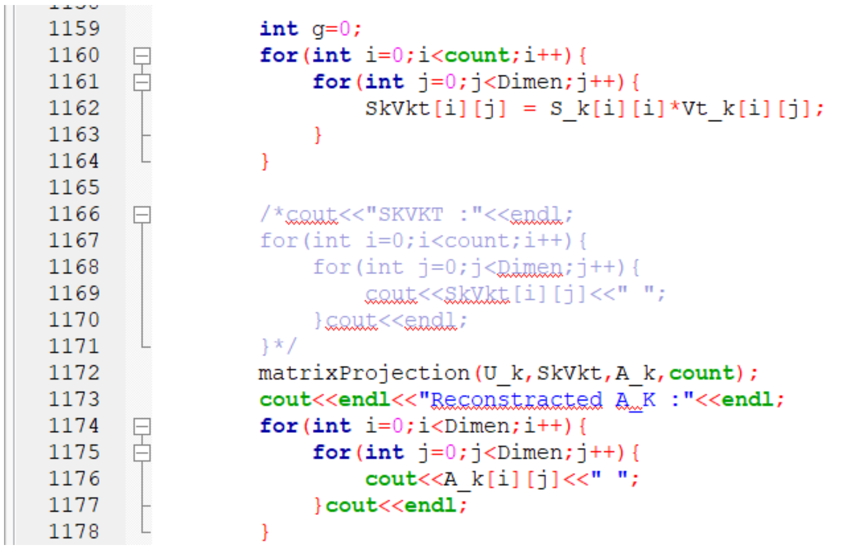
Figure 41: projection matrix

Projection matrix :



```
}
void matrixProjection(double P[Dimen+1][Dimen+1],double Q[Dimen+1][Dimen+1],double R[Dimen+1][Dimen+1],int c){
    for(int i=0;i<Dimen;i++)
        {
        for(int j=0;j<Dimen;j++)
        {
        R[i][j]=0;
        for(int g=0;g<c;g++)
        {
         R[i][j]+=P[i][g]*Q[g][j];
        }
        }
        }

}
```

Figure 42: projection matrix continued

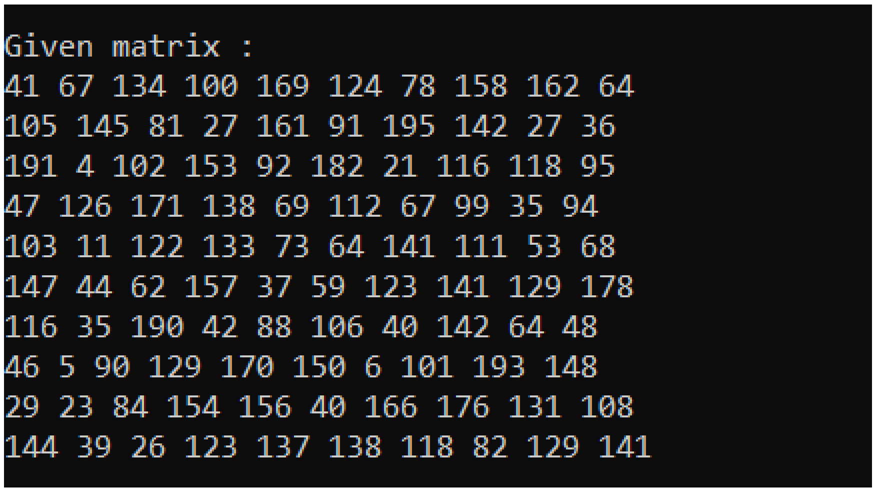


```
        int g=0;
        for(int i=0;i<count;i++){
            for(int j=0;j<Dimen;j++){
                SkVkt[i][j] = S_k[i][i]*Vt_k[i][j];
            }
        }

        /*cout<<"SKVKT :"<<endl;
        for(int i=0;i<count;i++){
            for(int j=0;j<Dimen;j++){
                cout<<SkVkt[i][j]<<" ";
            }cout<<endl;
        }*/
        matrixProjection(U_k,SkVkt,A_k,count);
        cout<<endl<<"Reconstracted A_K :"<<endl;
        for(int i=0;i<Dimen;i++){
            for(int j=0;j<Dimen;j++){
                cout<<A_k[i][j]<<" ";
            }cout<<endl;
        }
```

Figure 43: projection matrix continued

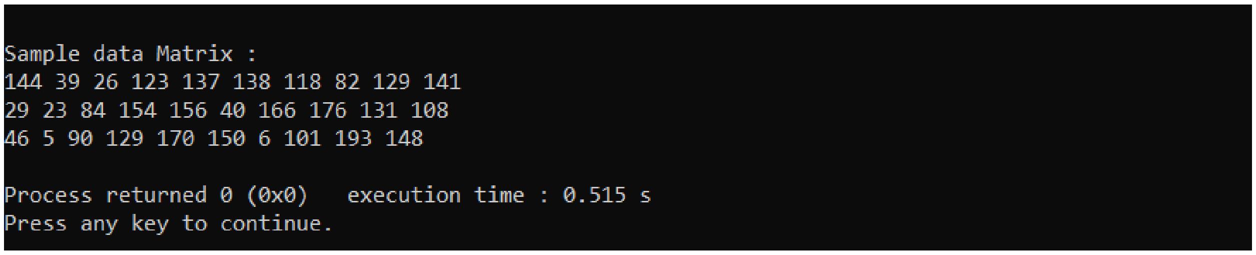
Let’s see the sample input & final output of that project :

Input :



```
Given matrix :
41 67 134 100 169 124 78 158 162 64
105 145 81 27 161 91 195 142 27 36
191 4 102 153 92 182 21 116 118 95
47 126 171 138 69 112 67 99 35 94
103 11 122 133 73 64 141 111 53 68
147 44 62 157 37 59 123 141 129 178
116 35 190 42 88 106 40 142 64 48
46 5 90 129 170 150 6 101 193 148
29 23 84 154 156 40 166 176 131 108
144 39 26 123 137 138 118 82 129 141
```

Output(Representative sample) :



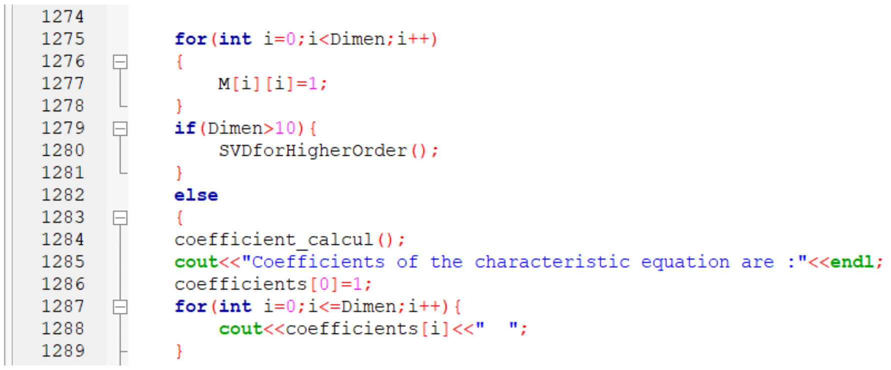
```
Sample data Matrix :
144 39 26 123 137 138 118 82 129 141
29 23 84 154 156 40 166 176 131 108
46 5 90 129 170 150 6 101 193 148

Process returned 0 (0x0)   execution time : 0.515 s
Press any key to continue.
```

Figure 44: final input 7 output for the project

Here, we see that only three of ten rows contain the most important feature in this dataset. So, we can store this 3x10 matrix instead of the original 10x10 matrix.

3 User Manual

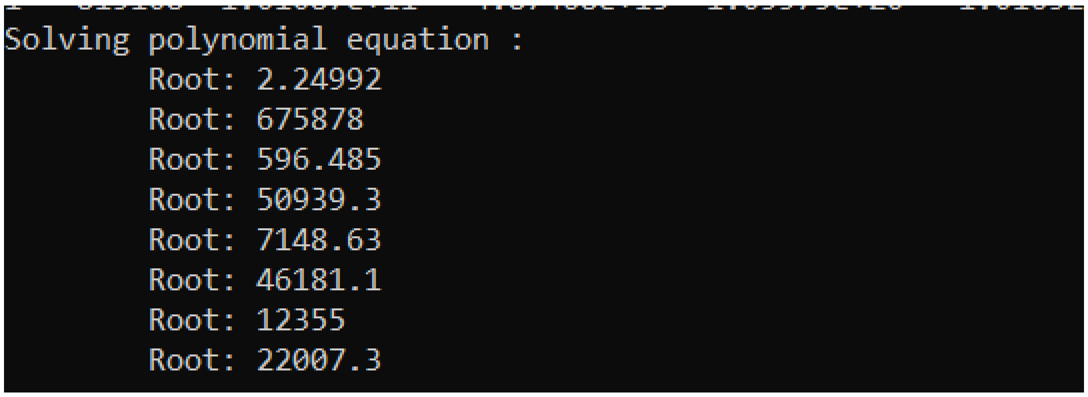
Our project will start from the main function. First of all it generates random numbers & writes them into the file, " **spl_task2.txt** ". Then program control goes to the transpose function & calculates the transpose of the matrix. Then it calculates the multiplication of Aᴛ & A. In order to compute **SVD**, we apply two different procedures based on the value of dimension. If n<10, then we use the **Bairstow Algorithm** to calculate the eigenvalues. Before Bairstow, we have to implement the **Faddeev Leverrier** algorithm to find the coefficients of the polynomial_equation.



```
    for(int i=0;i<Dimen;i++)
    {
        M[i][i]=1;
    }
    if(Dimen>10){
        SVDforHigherOrder();
    }
    else
    {
    coefficient_calcul();
    cout<<"Coefficients of the characteristic equation are :"<<endl;
    coefficients[0]=1;
    for(int i=0;i<=Dimen;i++){
        cout<<coefficients[i]<<"  ";
    }
```

Figure 45: user manual-1

In Bairstow, we will use the functions **GetCoefficients, reduce_equation, root_finding, printRoot** etc. After that implementation we will find the eigenvalues.

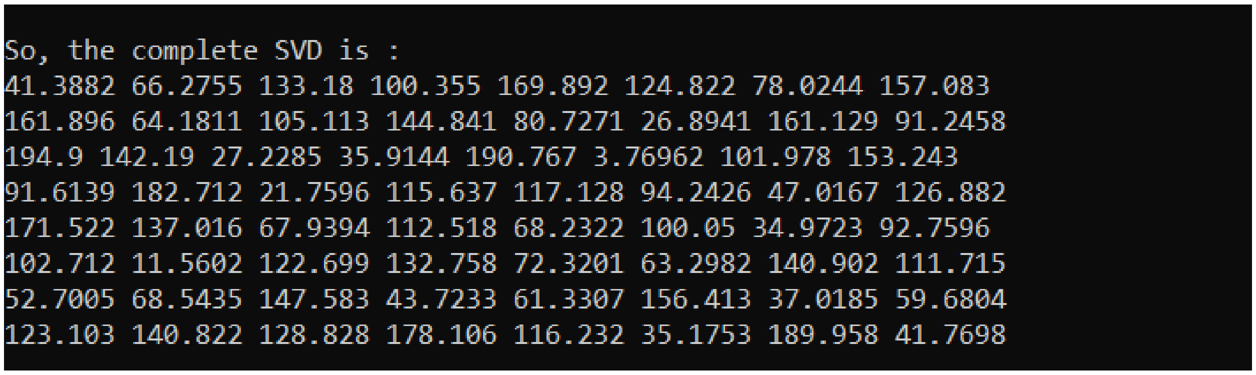


```
Solving polynomial equation :
        Root:  2.24992
        Root:  675878
        Root:  596.485
        Root:  50939.3
        Root:  7148.63
        Root:  46181.1
        Root:  12355
        Root:  22007.3
```

Figure 46: user manual-2

If n>10, program control goes to the function **SVDForHigherOrder**.

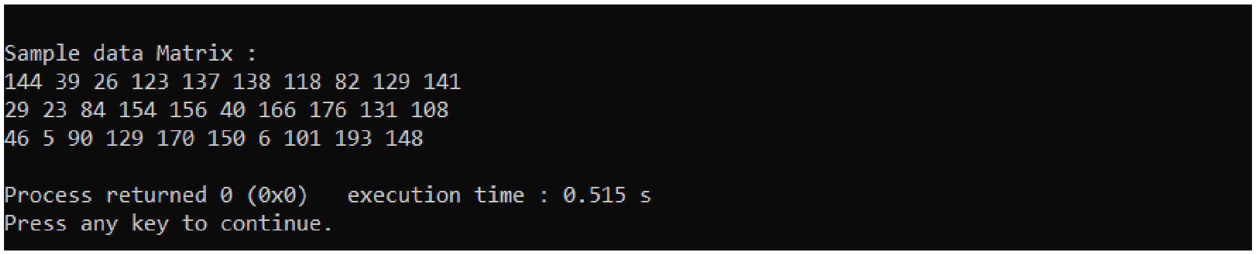
Here, we used the **Jacobi Rotation** method to calculate the eigenvalues. Then program control goes to **Gauss Elimination.** Here, we calculate the eigenvectors for each eigenvalue. Then we calculate the **U, Σ** & **V** matrix. After multiplication **U, Σ** & **Vт** we find the resulting matrix which is either equal or very close to the original matrix A.



```
So, the complete SVD is :
41.3882 66.2755 133.18 100.355 169.892 124.822 78.0244 157.083
161.896 64.1811 105.113 144.841 80.7271 26.8941 161.129 91.2458
194.9 142.19 27.2285 35.9144 190.767 3.76962 101.978 153.243
91.6139 182.712 21.7596 115.637 117.128 94.2426 47.0167 126.882
171.522 137.016 67.9394 112.518 68.2322 100.05 34.9723 92.7596
102.712 11.5602 122.699 132.758 72.3201 63.2982 140.902 111.715
52.7005 68.5435 147.583 43.7233 61.3307 156.413 37.0185 59.6804
123.103 140.822 128.828 178.106 116.232 35.1753 189.958 41.7698
```

Figure 47: user manual-3

Then program control goes to the function **LeverageScoreSampling.** That function calculates the leverage scores and finally returns the representative sample of the original matrix which contains the most important feature of the dataset.



```
Sample data Matrix :
144 39 26 123 137 138 118 82 129 141
29 23 84 154 156 40 166 176 131 108
46 5 90 129 170 150 6 101 193 148

Process returned 0 (0x0)   execution time : 0.515 s
Press any key to continue.
```

Figure 48: user manual-4

4 Conclusion

Leverage score sampling using SVD is a game-changer when it comes to analyzing high-dimensional datasets. Its ability to efficiently select representative subsets of data based on leverage scores unlocks new possibilities for deriving actionable insights. By leveraging the mathematical properties of SVD, this technique enhances efficiency, reduces noise, and improves decision-making processes. Whether in finance, marketing, or other fields, leverage score sampling using SVD empowers analysts and decision-makers to make the most of their data.

So, embrace the power of leverage score sampling using SVD and unlock the potential within our high-dimensional datasets

Reference

1.https://math.iitm.ac.in/public_html/sryedida/caimna/transcendental/polynomial%20methods/brs%20method.html, Bairstow Method, 19/05/2023
2. Faddeev–LeVerrier algorithm - Wikipedia, 19/05/2023
3.https://www.cs.cmu.edu/afs/cs/user/dwoodruf/www/teaching/15859-fall17/scribe8.pdf, David Woodruff, 19/05/2023
4. Regular Inverse & Pseudo Inverse Matrix Calculation using Singular Value Decomposition | by Antonius Freenergi | Medium, Regular vs Pseudo Inverse, 19/05/2023
5. Singular Values Decomposition (SVD) in C++11 by an Example - CodeProject, Arthur V.Ratz, 19/05/2023
6. eigen value & eigen vector.pdf, 19/05/2023
7. Iterative_methods_for_eigenvalue_problem.pdf, 19/05/2023