Spl-1 Project Report- 2023

Numeric Operations Library

Submitted by:

Salsabila Zaman
BSSE Roll:1443
BSSE Session:2021-22

Supervised By

Abdus Satter
Assistant Professor,
Institue Of Information Technology

Supervisor's Approval: ________________________
(signature)

Institute of Information Technology

University of Dhaka

Date:17/12/23

Contents

List Of Figures

Introduction

The Numerical Operations Library is an ambitious and innovative project that aims to create a comprehensive and easy-to-use software package that meets the demands of number theorists, mathematicians, researchers, and students. It will offer essential tools and algorithms for exploring prime numbers, factorization, modular arithmetic, and other fundamental concepts in number theory. The library will also feature matrix calculations along with equation solving and expression evaluation to allow users of all proficiency levels to interact and gain knowledge about mathematical functionalities.

Domain and Overview

Numeric Operations Library encompasses a wide range of mathematical functionalities, catering to diverse domains. In the realm of number theory, it tackles prime numbers, aiding in cryptography and algorithmic applications. The library delves into digit manipulations, providing tools for operations like summation, reversal, and palindrome checks. For algebraic pursuits, Numeric Operations Library computes the Greatest Common Divisor (GCD) and Least Common Multiple (LCM) and solves linear and quadratic equations. In the field of sequences, it efficiently generates Fibonacci sequences for various mathematical explorations.

Specializing in Matrix Manipulation, Numeric Operations Library provides essential tools for linear algebra and beyond. It handles matrix addition and multiplication, crucial in computer graphics and optimization, respectively. Determinants and inverses are computed, supporting applications in system stability analysis and linear system solutions. Matrix power calculations find use in dynamical systems, while transposition aids in diverse mathematical transformations. Eigenvalues and eigenvectors, vital in physics and scientific research, and matrix rank, fundamental in system analysis and optimization, complete the suite of matrix operations offered by Numeric Operations Library.

1 Prime Numbers

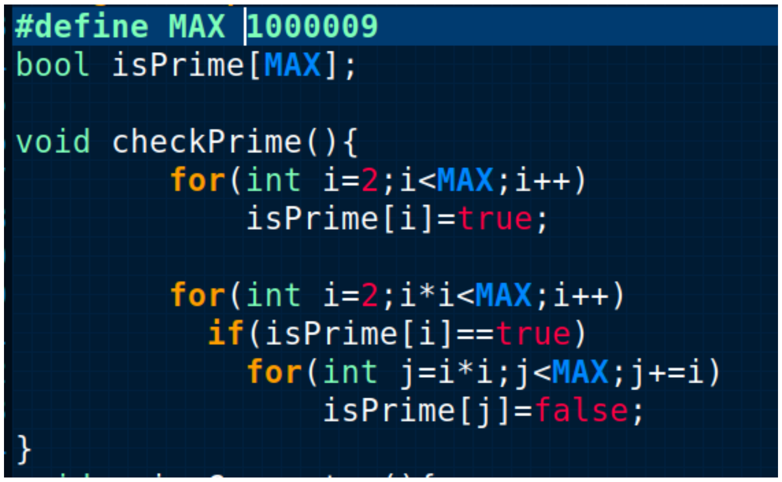
A prime number is *a* whole number greater than 1 whose only factors are 1 and itself. In this project, you can check

-Check number if it's prime or not

-Generate all prime numbers in a given range

1.1 Prime Checker

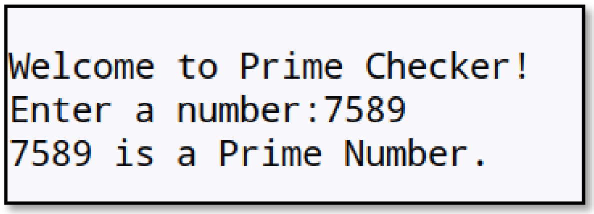
For both of the work I have implemented Sieve of Erathosthnenes and precalculated all prime numbers upto 1000009.Sieve of Eratosthenes is an almost mechanical procedure for separating out composite numbers and leaving the primes.



```
#define MAX 1000009
bool isPrime[MAX];

void checkPrime(){
        for(int i=2;i<MAX;i++)
            isPrime[i]=true;

        for(int i=2;i*i<MAX;i++)
          if(isPrime[i]==true)
            for(int j=i*i;j<MAX;j+=i)
                isPrime[j]=false;
}
```

Figure 1.1.a

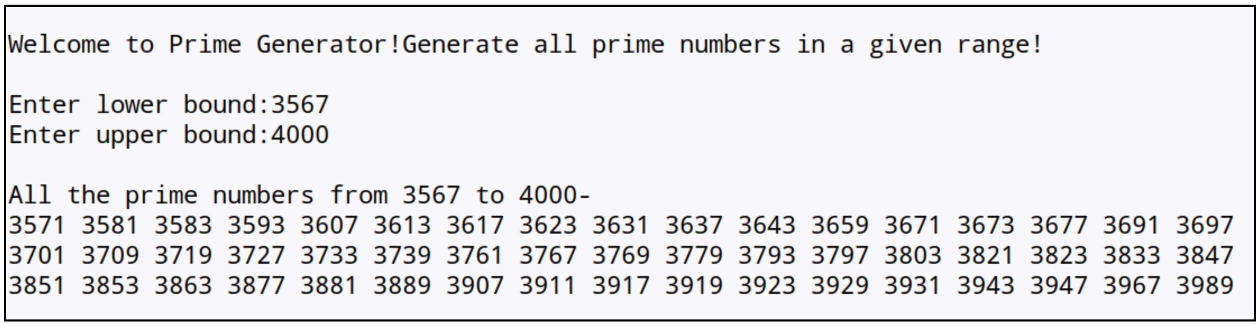
At the beginning of this code, it marks all numbers except zero and one as potential prime numbers, after that, it begins the process to sift composite numbers. For this it does iterate over all numbers from 2 to n.In the second iteration, if the current number is a prime number, it marks all the multiples of i as composite numbers starting from i^2. This is already an optimization over the naive way of implementing it, as all smaller numbers that are multiples of i, necessary also have a prime factor which is less than i, so all of them were already sifted by this stage.



```
Welcome to Prime Checker!
Enter a number:7589
7589 is a Prime Number.
```

Figure 1.1.b

1.2 Prime Generator



```
Welcome to Prime Generator!Generate all prime numbers in a given range!

Enter lower bound:3567
Enter upper bound:4000

All the prime numbers from 3567 to 4000-
3571 3581 3583 3593 3607 3613 3617 3623 3631 3637 3643 3659 3671 3673 3677 3691 3697
3701 3709 3719 3727 3733 3739 3761 3767 3769 3779 3793 3797 3803 3821 3823 3833 3847
3851 3853 3863 3877 3881 3889 3907 3911 3917 3919 3923 3929 3931 3943 3947 3967 3989
```

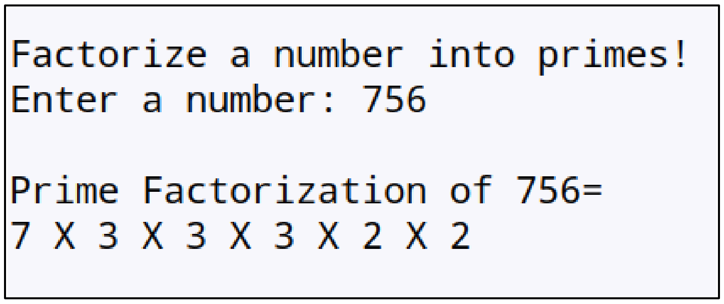
Figure 1.2

Although the sieve can be a tedious process for discovering large primes,it is a procedure that can be effectively turned over to a computer, using a language such as Fortran, BASIC, or Pascal. According to Ore, every table of primes has been constructed with the method described by Eratosthenes. This includes tables of all the primes up to one hundred million.

2 Prime Factorization

Prime factorization is a process of factoring a number in terms of prime numbers i.e. the factors will be prime numbers. Here we used the Division Method to calculate the prime factors. The steps to calculate the prime factors of a number is similar to the process of finding the factors of a large number.

- Step 1: Divide the given number by the smallest prime number. In this case, the smallest prime number should divide the number exactly.
- Step 2: Again, divide the quotient by the smallest prime number.
- Step 3: Repeat the process, until the quotient becomes 1.
- Step 4: Finally, multiply all the prime factors.



```
Factorize a number into primes!
Enter a number: 756

Prime Factorization of 756=
7 X 3 X 3 X 3 X 2 X 2
```

Figure 2

3 Digit Manipulations

Digit manipulation refers to operations or algorithms that focus on individual digits inside a numerical representation, such as extracting, modifying, or analyzing individual digits based on their position. Here our goal is to perform operations on individual digits. Digit manipulations can be used in-

1.Data validation

2.Mathematical Calculations

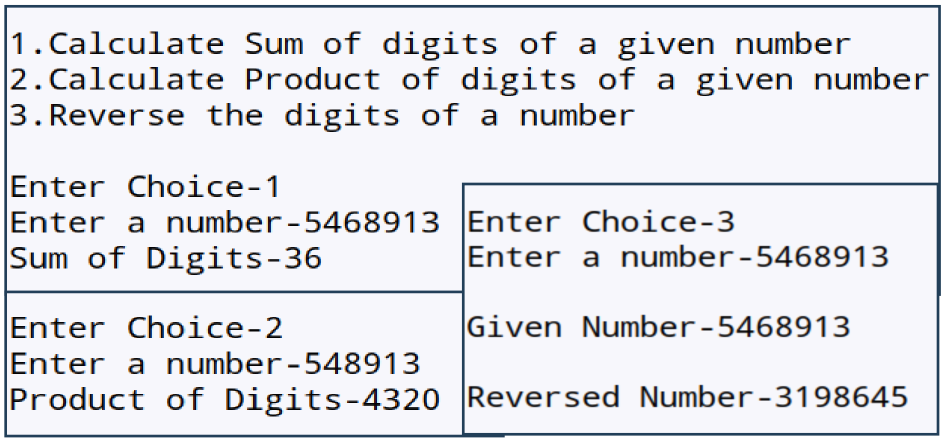
3.Data Validation

4.Encryption Algorithms

5.User Input Processing involving numerical dataset

6.Password Security: To ensure password strength. For instance, algorithms that check for the presence of numbers, their position, or their frequency can enhance password security.

7.Data Analysis: Processing and extracting relevant information from numerical datasets. For example, analyzing sales figures, population statistics, or any numerical data may involve digit manipulation.



```
1.Calculate Sum of digits of a given number
2.Calculate Product of digits of a given number
3.Reverse the digits of a number

Enter Choice-1
Enter a number-5468913
Sum of Digits-36
```

```
Enter Choice-2
Enter a number-548913
Product of Digits-4320
```

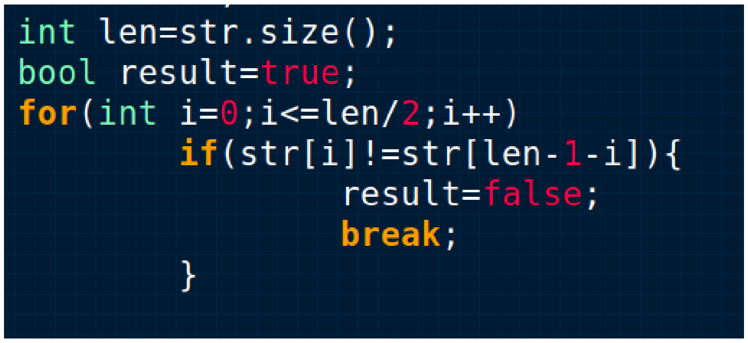
```
Enter Choice-3
Enter a number-5468913

Given Number-5468913

Reversed Number-3198645
```

Figure 3

4 Palindrome Checker

A **palindrome** is a word, number, phrase, or other sequence of symbols that reads the same backwards as forwards, such as *madam* or *racecar*, A palindrome checker is a program or algorithm that determines whether a given sequence of characters forms a palindrome. Developing an efficient palindrome checker can be a starting point for exploring different algorithms and optimization techniques in computer science.

In this program, we take the user input as a string then compare the leftmost first character of the string with the rightmost 1st, leftmost 2nd character with the rightmost 2nd and so on. Whenever it doesn't find a match, the loop breaks and the given string is declared as not a palindrome, otherwise a palindrome.

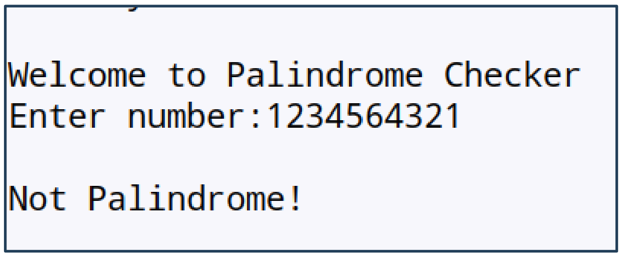


```
int len=str.size();
bool result=true;
for(int i=0;i<=len/2;i++)
        if(str[i]!=str[len-1-i]){
                result=false;
                break;
        }
```

Figure 4.1

The principle of a palindrome checker is straightforward, yet there are many uses for it. It provides a foundation for understanding and implementing basic string manipulation and comparison algorithms.

- Security and Cryptography: A palindrome checker can be utilized to verify the integrity of a sequence, and palindromes can be utilized in specific cryptographic applications or security techniques.
- Data Filtering: A palindrome checker can assist in locating palindromic patterns in alphanumeric data when it needs to be filtered or processed.



```
Welcome to Palindrome Checker
Enter number:1234564321

Not Palindrome!
```

Figure 4.2

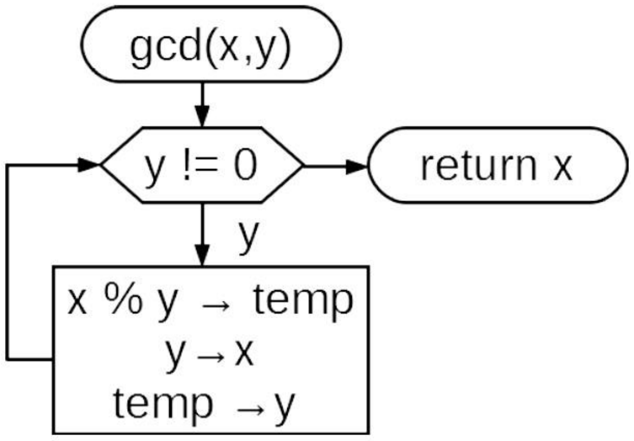
5 GCD Calculator

The greatest common divisor (GCD) of two numbers is the greatest factor that divides both the numbers. Here I used Euclidean algorithm.

Formal description of the Euclidean algorithm:

- ✓ **Input** Two positive integers, a and b.
- ✓ **Output** The greatest common divisor, g, of a and b.
- ✓ **Internal computation**

  1. If a<b, exchange a and b.
  2. Divide a by b and get the remainder, r. If r=0, report b as the GCD of a and b.
  3. Replace a by b and replace b by r. Return to the previous step.



Flowchart for gcd function

Figure 5.1

Example-Divide 210 by 45, and get the result 4 with remainder 30, so 210=4⋅45+30.

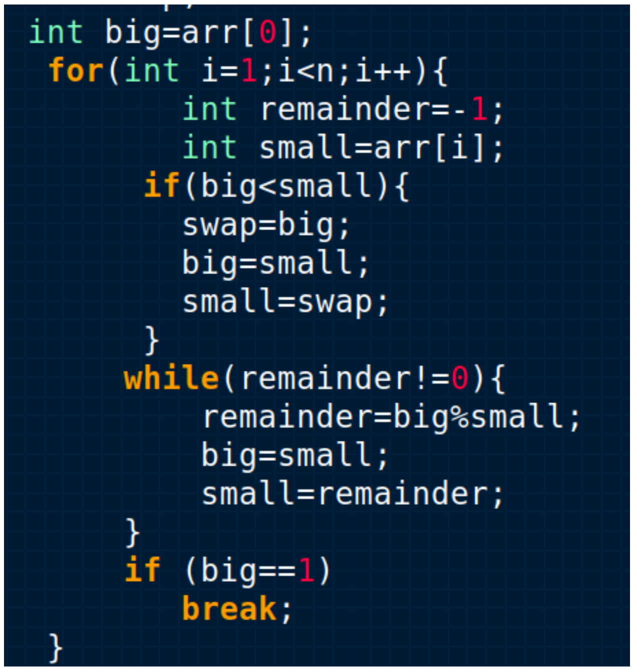
Divide 45 by 30, and get the result 1 with remainder 15, so 45=1⋅30+15.

Divide 30 by 15, and get the result 2 with remainder 0, so 30=2▪15+0.

The greatest common divisor of 210 and 45 is 15.

The program extends the euclidean algorithm for multiple numbers. Firstly,we calculated gcd for two numbers and then the ran the same algorithm on gcd of previous two numbers and another number ,until no numbers are left.

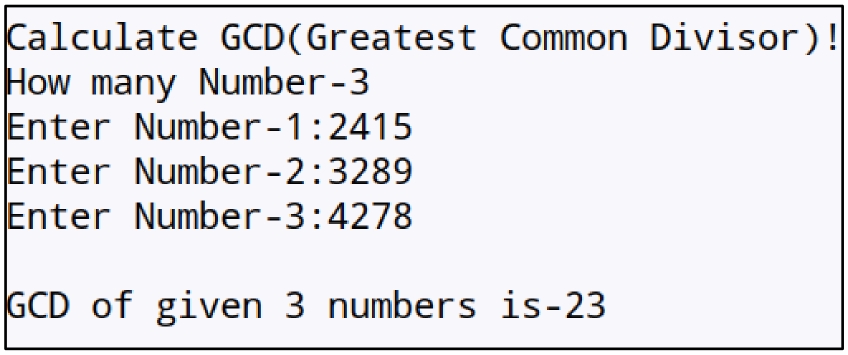
The large number always have to be divided by the small number, that’s why the program takes two variable named big and small, divides big by small and replaces big with small and small with the remainder until the remainder is zero or small is 1.



```
int big=arr[0];
 for(int i=1;i<n;i++){
        int remainder=-1;
        int small=arr[i];
      if(big<small){
        swap=big;
        big=small;
        small=swap;
      }
     while(remainder!=0){
         remainder=big%small;
         big=small;
         small=remainder;
     }
     if (big==1)
        break;
 }
```

Figure 5.2

Sample Input/Output:



```
Calculate GCD(Greatest Common Divisor)!
How many Number-3
Enter Number-1:2415
Enter Number-2:3289
Enter Number-3:4278

GCD of given 3 numbers is-23
```

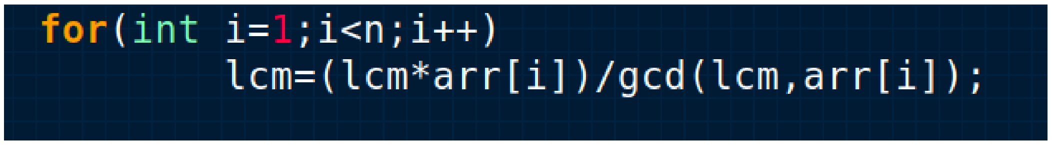
Figure 5.3

6 LCM Calculator

LCM (Least Common Multiple) of two numbers is the smallest number which can be divided by both numbers. In this program, we calculated LCM using GCD using this formula.

$$LCM(a,b) = (a \times b) / GCD(a,b)$$

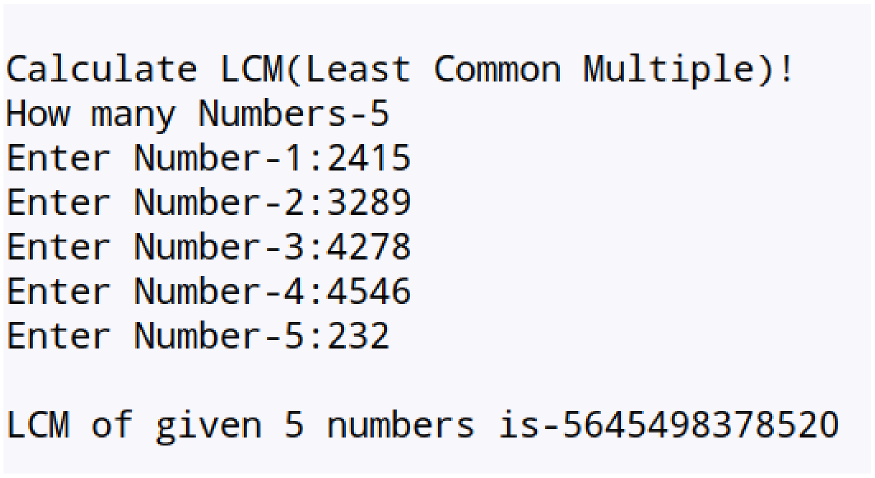
As the program calculates LCM for multiple numbers, we first initialize the lcm variable with first element of the array and then run this according to the equation until no other elements is left at the array for calculation,



```
for(int i=1;i<n;i++)
        lcm=(lcm*arr[i])/gcd(lcm,arr[i]);
```

Figure 6.1

Sample Input/Output:



```
Calculate LCM(Least Common Multiple)!
How many Numbers-5
Enter Number-1:2415
Enter Number-2:3289
Enter Number-3:4278
Enter Number-4:4546
Enter Number-5:232

LCM of given 5 numbers is-5645498378520
```

Figure 6.2

7 Fibonacci Sequences

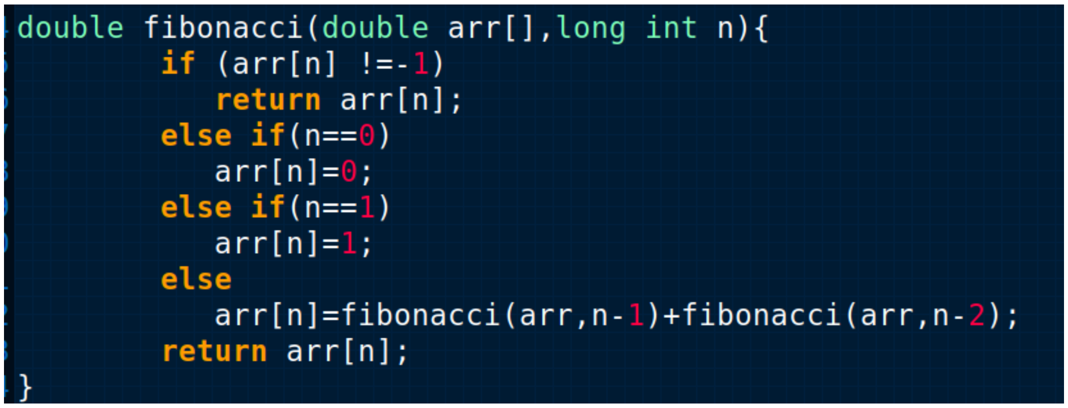
The Fibonacci sequence is a set of integers (the Fibonacci numbers) that starts with a zero, followed by a one, then by another one, and then by a series of steadily increasing numbers. The sequence follows the rule that each number is equal to the sum of the preceding two numbers.

The Fibonacci sequence begins with the following 14 integers:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233 …

The Fibonacci sequence can be defined by the following three equations:

- $F_0 = 0$ (applies only to the first integer)
- $F_1 = 1$ (applies only to the second integer)
- $F_n = F_{n-1} + F_{n-2}$ (applies to all other integers)

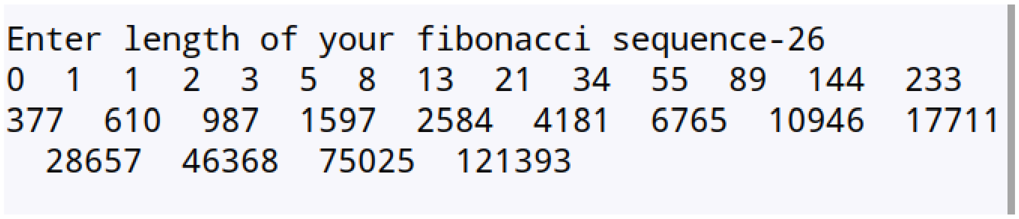


```
double fibonacci(double arr[],long int n){
        if (arr[n] !=-1)
           return arr[n];
        else if(n==0)
           arr[n]=0;
        else if(n==1)
           arr[n]=1;
        else
           arr[n]=fibonacci(arr,n-1)+fibonacci(arr,n-2);
        return arr[n];
}
```

Figure 7.1

The program generates Fibonacci sequence of numbers with dynamic programming.

Sample Input/Output:



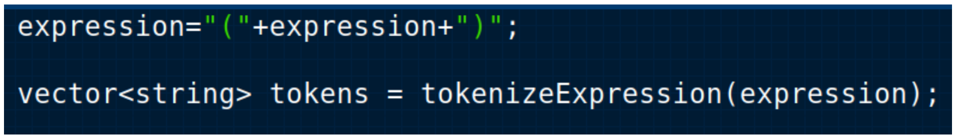
```
Enter length of your fibonacci sequence-26
0  1  1  2  3  5  8  13  21  34  55  89  144  233
377  610  987  1597  2584  4181  6765  10946  17711
  28657  46368  75025  121393
```

Figure 7.2

8 Expression Evaluation

An expression is a combination of operators, constants and variables. An expression may consist of one or more operands, and zero or more operators to produce a value. There are various types of expressions- Arithmetic, Logical, Relational, Conditional. The program works with arithmetic expressions.

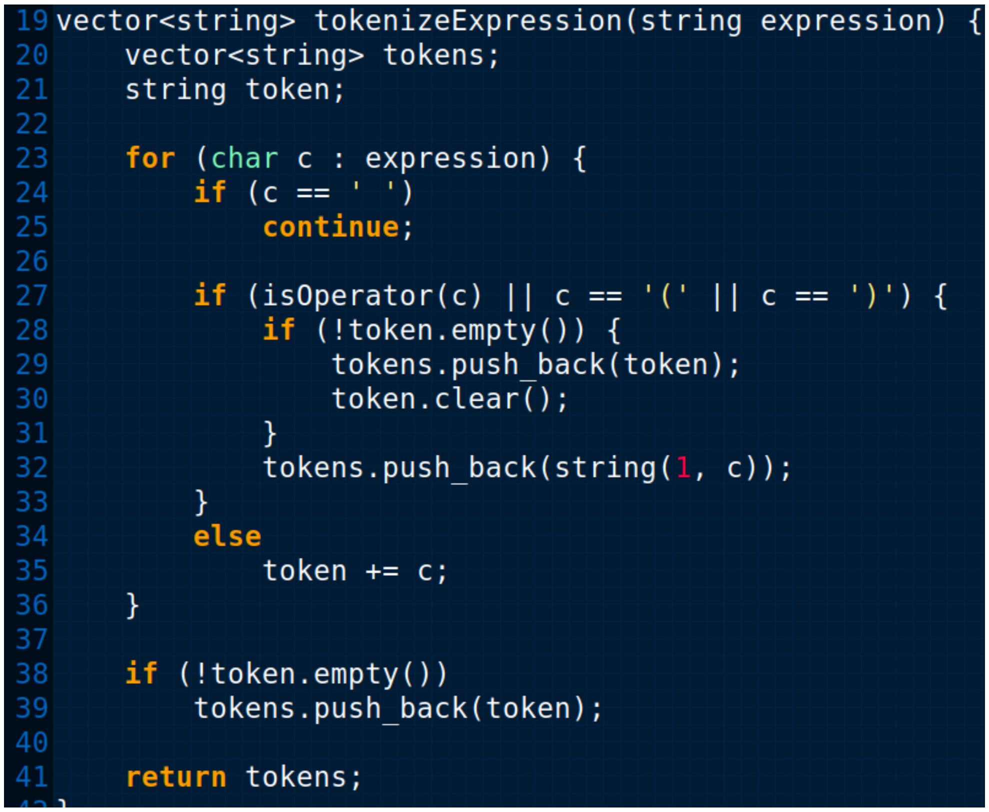
An *arithmetic expression* is an expression using additions **+**, subtractions **-**, multiplications *****, divisions **/**, and exponentials **^**.A parenthesis can be used to override the precedence of operators and force the evaluation of a sub-expression within an expression. The program is taking input the expression from the user and encapsulating it with brackets, the left and right bracket will work as a marker for the start and end of the expression(string). Then the program tokenizes the expression and convert it into a vector of strings, where each string is a token(operator/operand).



```cpp
expression="("+expression+")";

vector<string> tokens = tokenizeExpression(expression);
```

Figure 8.1

The program reads the expression character by character using for-loop, it ignores whitespace. It also keeps a string named "token", which is to put all parts of an operand to a single operand. Example- if there is an operand 3.1416, firstly the token=3, then token=3. , then token=3.14 ,it keeps adding until an operator is found. When an operator is found, it pushes the current token into tokens vector and empties the current token and then the operator is also push into the vector. Last if -is for the last operand to be push on to the vector. Lastly it returns the vector containing tokens.



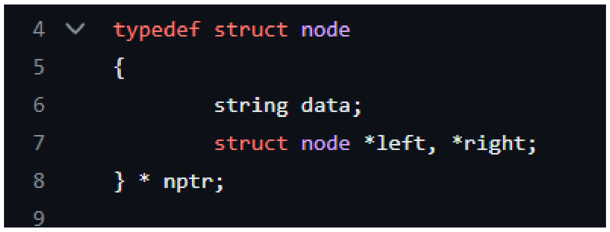
```cpp
vector<string> tokenizeExpression(string expression) {
    vector<string> tokens;
    string token;

    for (char c : expression) {
        if (c == ' ')
            continue;

        if (isOperator(c) || c == '(' || c == ')') {
            if (!token.empty()) {
                tokens.push_back(token);
                token.clear();
            }
            tokens.push_back(string(1, c));
        }
        else
            token += c;
    }

    if (!token.empty())
        tokens.push_back(token);

    return tokens;
```

Figure 8.2

Implemented the Shunting Yard Algorithm to build a binary expression tree from an infix expression represented as a vector of strings.

The idea behind this condition is to pop operators from the stack and process them when the operator at the top of the stack has higher precedence than or equal to the current operator (s[i]). This ensures that operators with higher precedence are processed first, following the rules of the Shunting Yard Algorithm.

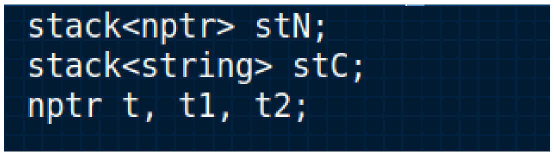
The program stores converted infix notation in a tree. To create the tree, the program uses an user defined data-type typedef and its pointer to create new nodes of the tree and add to the existing tree.



```
typedef struct node
{
        string data;
        struct node *left, *right;
} * nptr;
```

Figure 8.3

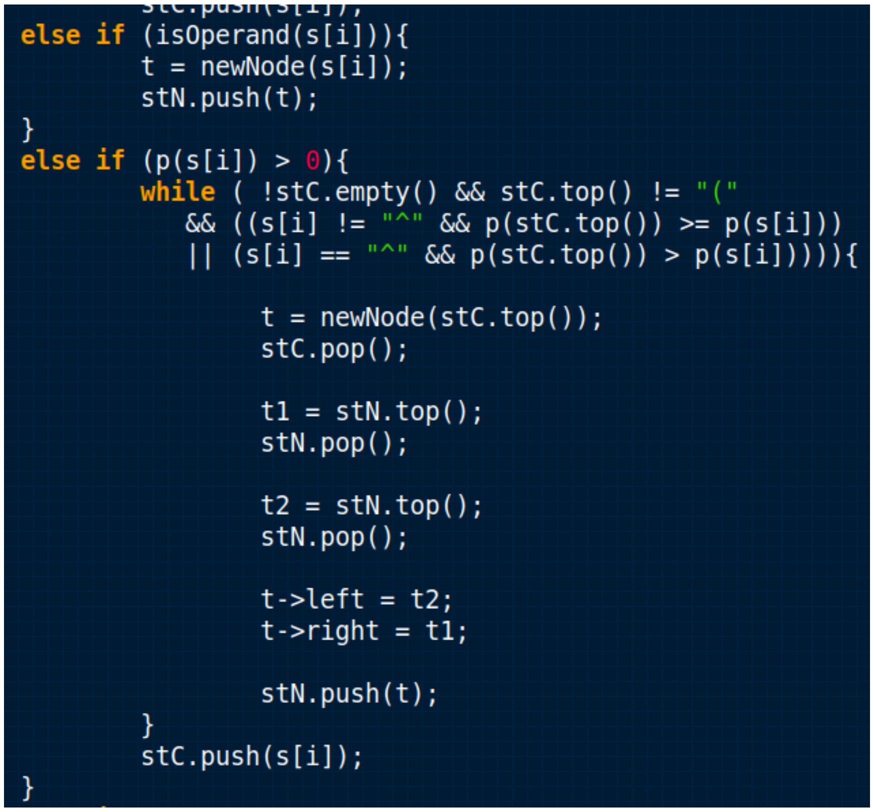
The program now starts building the tree from the existing tokens. The shunting yard algorithm uses stacks to process the associativity and precedence of operators. Here it takes two stacks ,one stack stN to store the processed nodes and another stack stC to store the unprocessed operators.



```
stack<nptr> stN;
stack<string> stC;
nptr t, t1, t2;
```

Figure 8.4

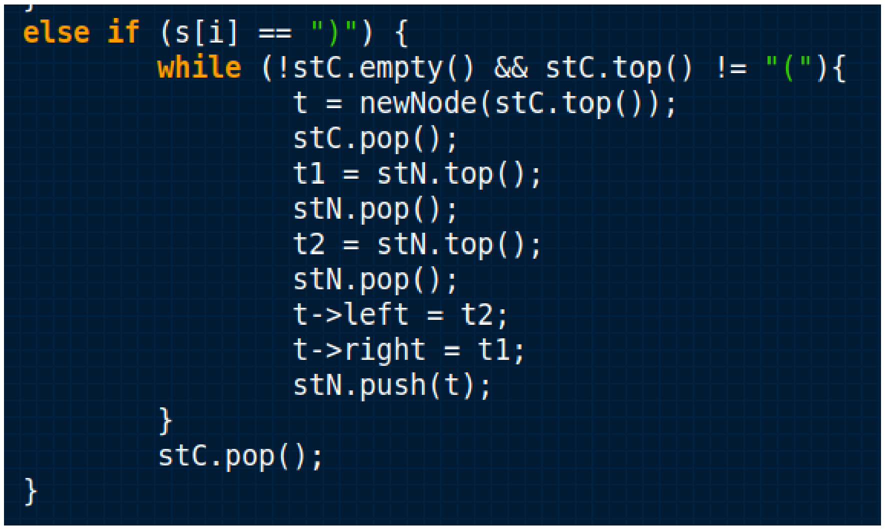
The loop continues as it pushes the operands to stN and whenever it finds the precedence of the top of operator stack is higher than the current operator, it pops out that operator from the stack and two nodes from stN stack, makes the operator a root and pushes the two nodes as a child of this root. All the operands of the expression are leaf’s of this tree.



```
else if (isOperand(s[i])){
        t = newNode(s[i]);
        stN.push(t);
}
else if (p(s[i]) > 0){
        while ( !stC.empty() && stC.top() != "("
           && ((s[i] != "^" && p(stC.top()) >= p(s[i]))
           || (s[i] == "^" && p(stC.top()) > p(s[i])))){

                t = newNode(stC.top());
                stC.pop();

                t1 = stN.top();
                stN.pop();

                t2 = stN.top();
                stN.pop();

                t->left = t2;
                t->right = t1;

                stN.push(t);
        }
        stC.push(s[i]);
}
```

Figure 8.5

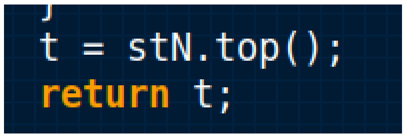
The program also takes account the precedence of the parentheses. It pushes "(" to the operator stack and whenever it finds ")",it pops out all the operators until it finds "(".



```
else if (s[i] == ")") {
        while (!stC.empty() && stC.top() != "("){
                t = newNode(stC.top());
                stC.pop();
                t1 = stN.top();
                stN.pop();
                t2 = stN.top();
                stN.pop();
                t->left = t2;
                t->right = t1;
                stN.push(t);
        }
        stC.pop();
}
```

Figure 8.6

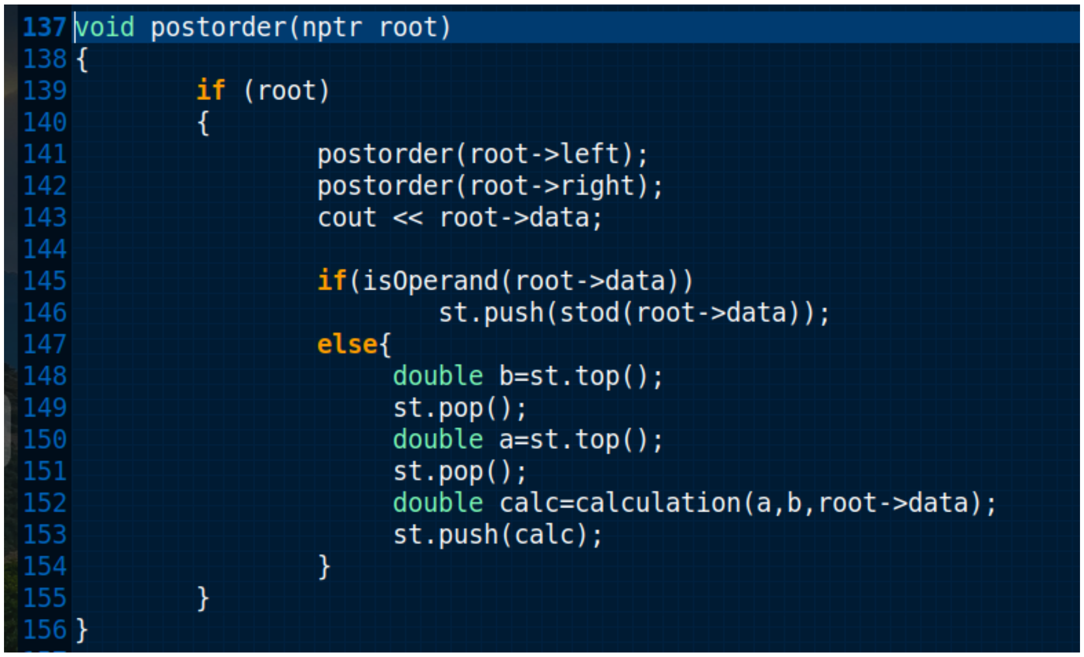
Lastly it returns the root of tree, which is the operator with lowest precedence among the expression.



```
t = stN.top();
return t;
```

Figure 8.7

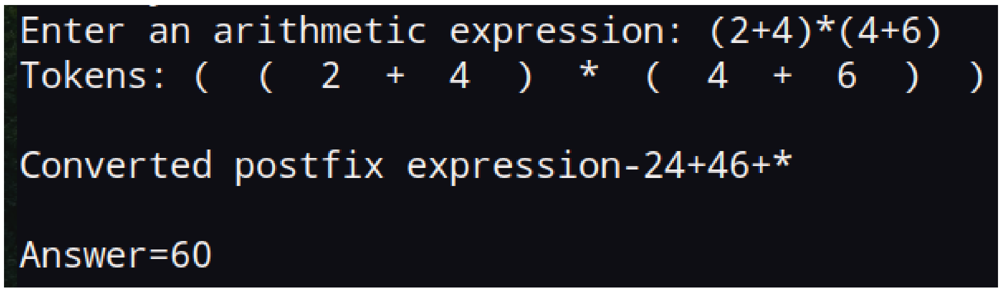
After building the tree, the program traverses the tree in post order and evaluates the expression with it. It takes a stack "st<double>" to evaluate the expression. Whenever we find an operand, we push this to stack by converting it from string to double. And when we find an operator, we pop two operands from the stack, do the calculation and again push the calculated value into the stack again.



```
void postorder(nptr root)
{
        if (root)
        {
                postorder(root->left);
                postorder(root->right);
                cout << root->data;

                if(isOperand(root->data))
                        st.push(stod(root->data));
                else{
                     double b=st.top();
                     st.pop();
                     double a=st.top();
                     st.pop();
                     double calc=calculation(a,b,root->data);
                     st.push(calc);
                }
        }
}
```

Figure 8.8

Sample Input/Output:



```
Enter an arithmetic expression: (2+4)*(4+6)
Tokens: (  (  2  +  4  )  *  (  4  +  6  )  )

Converted postfix expression-24+46+*

Answer=60
```

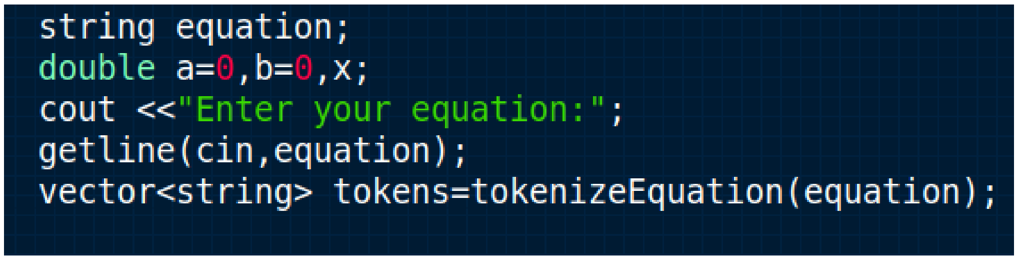
Figure 8.9

9 Equation Solver (One Variable)

9.1 Linear Equation Solver

An equation that has the highest degree of one is known as a **linear equation**. A linear equation in one variable is an equation in which there is only one variable present.

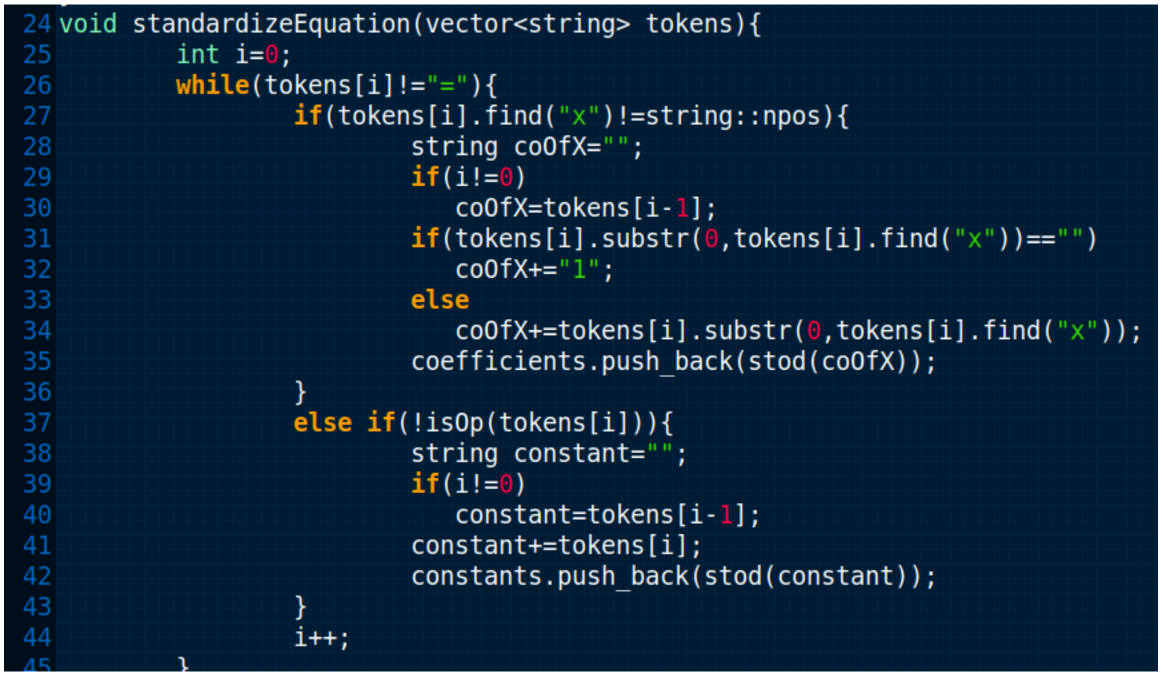
It is of the form **Ax+B=0,** where A and B are any two real numbers and x is an unknown variable that has only one solution.



```cpp
string equation;
double a=0,b=0,x;
cout <<"Enter your equation:";
getline(cin,equation);
vector<string> tokens=tokenizeEquation(equation);
```

Figure 9.1.a

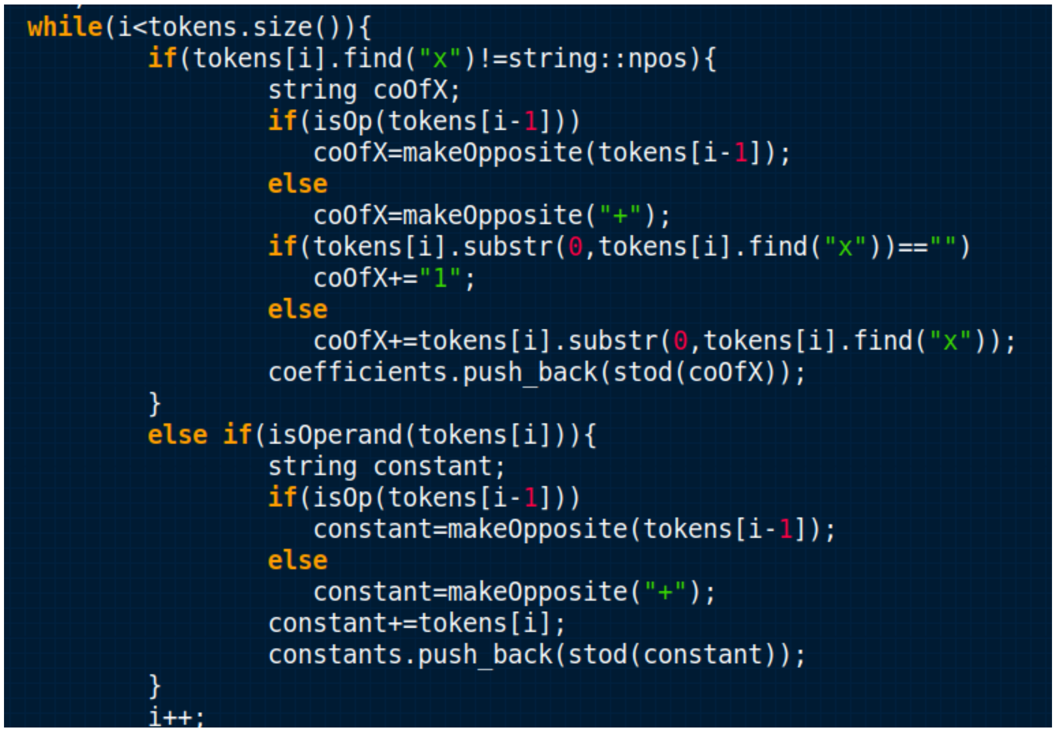
The program takes input the equation as a string it uses getline instead of cin, so that the input stream does not stop after getting whitespace. Then it tokenizes the equation the same way it tokenized the expression in the previous section except the fact now there is an extra operator "=".



```cpp
void standardizeEquation(vector<string> tokens){
        int i=0;
        while(tokens[i]!="="){
                if(tokens[i].find("x")!=string::npos){
                        string coOfX="";
                        if(i!=0)
                           coOfX=tokens[i-1];
                        if(tokens[i].substr(0,tokens[i].find("x"))=="")
                           coOfX+="1";
                        else
                           coOfX+=tokens[i].substr(0,tokens[i].find("x"));
                        coefficients.push_back(stod(coOfX));
                }
                else if(!isOp(tokens[i])){
                        string constant="";
                        if(i!=0)
                           constant=tokens[i-1];
                        constant+=tokens[i];
                        constants.push_back(stod(constant));
                }
                i++;
        }
```

Figure 9.1.b

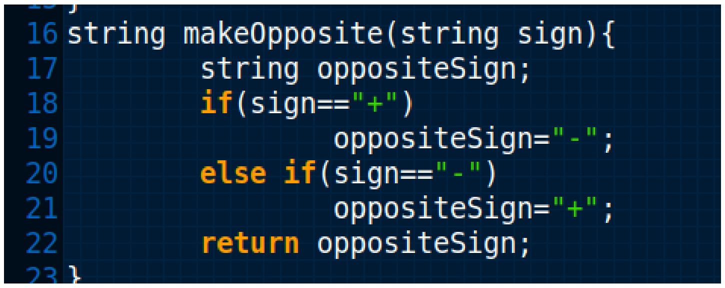
Example: Equation: 2x+5=x+2. Firstly, we have to convert this equation to a standard form. For that, we traverse the left side of the equal sign in one loop and right-side in another. If any token contains any variable, then the coefficient of that variable is converted into double and added to the coefficients stack along with its preceding sign otherwise the token is same way converted into double and added to the constants stack.



```
while(i<tokens.size()){
        if(tokens[i].find("x")!=string::npos){
                string coOfX;
                if(isOp(tokens[i-1]))
                   coOfX=makeOpposite(tokens[i-1]);
                else
                   coOfX=makeOpposite("+");
                if(tokens[i].substr(0,tokens[i].find("x"))=="")
                   coOfX+="1";
                else
                   coOfX+=tokens[i].substr(0,tokens[i].find("x"));
                coefficients.push_back(stod(coOfX));
        }
        else if(isOperand(tokens[i])){
                string constant;
                if(isOp(tokens[i-1]))
                   constant=makeOpposite(tokens[i-1]);
                else
                   constant=makeOpposite("+");
                constant+=tokens[i];
                constants.push_back(stod(constant));
        }
        i++;
```

Figure 9.1.c

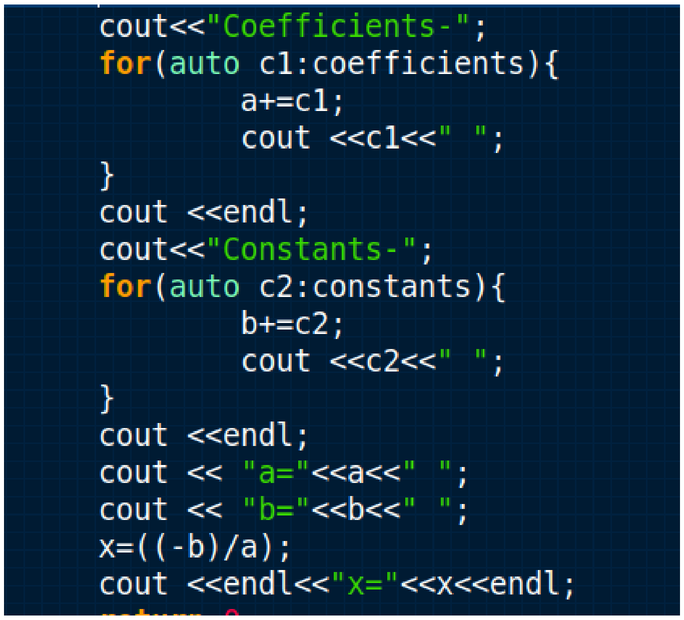
The implementation for the right side of the equation will be a bit different than the left side as the signs change according to the position. Coefficients and constants will be added as the same way, just while adding the preceding sign, we'll put the sign to a function "makeOppsite", which return the opposite sign of a given sign (Example: if '+' returns '-').



```
string makeOpposite(string sign){
        string oppositeSign;
        if(sign=="+")
                oppositeSign="-";
        else if(sign=="-")
                oppositeSign="+";
        return oppositeSign;
}
```

Figure 9.1.d

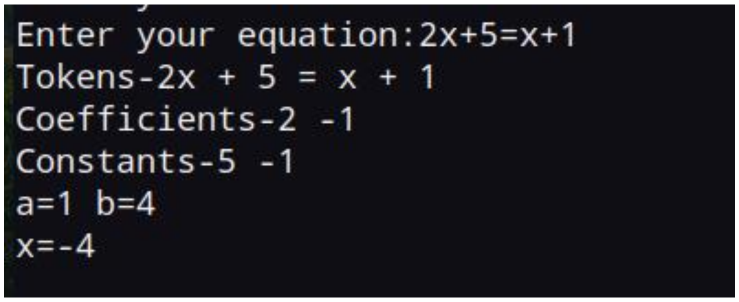
After the program is done traversing and standardizing the given equation, then it simply sums up the elements of coefficients vector and constants vector and stores it to two variables. Lastly, uses the standard formula $x = -b/a$ to calculate the value of x.



```
cout<<"Coefficients-";
for(auto c1:coefficients){
        a+=c1;
        cout <<c1<<" ";
}
cout <<endl;
cout<<"Constants-";
for(auto c2:constants){
        b+=c2;
        cout <<c2<<" ";
}
cout <<endl;
cout << "a="<<a<<" ";
cout << "b="<<b<<" ";
x=((-b)/a);
cout <<endl<<"x="<<x<<endl;
```

Figure 9.1.e

Sample Input/Output:



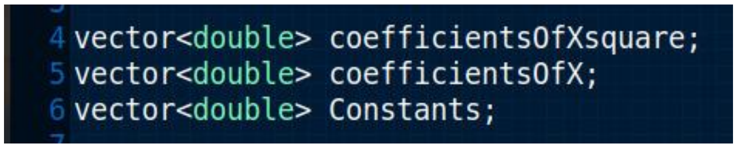
```
Enter your equation:2x+5=x+1
Tokens-2x + 5 = x + 1
Coefficients-2 -1
Constants-5 -1
a=1 b=4
x=-4
```

Figure 9.1.f

9.2 Quadratic Equation Solver

An equation that has the highest degree of two is known as a **quadratic equation**. A quadratic equation in one variable is an equation in which there is only one variable present. It is of the form **$Ax^2+Bx+C=0$,** where A,B and C are any three real numbers and x is an unknown variable that has two solution.

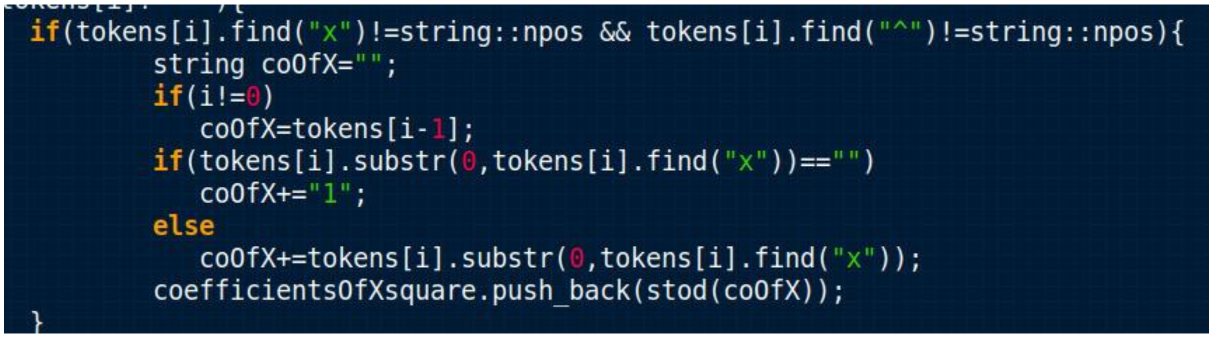
This program works exactly like the linear equation solver except with these additional steps-



```
vector<double> coefficientsOfXsquare;
vector<double> coefficientsOfX;
vector<double> Constants;
```

Figure 9.2.a

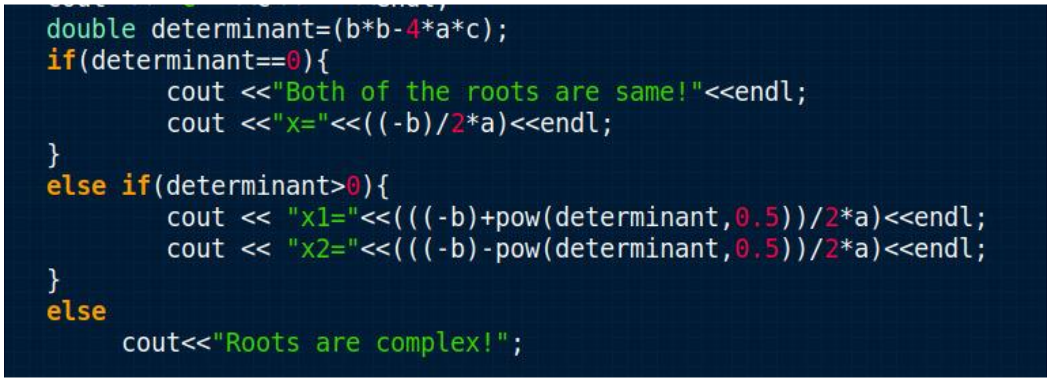
Below here, it finds the coefficient of x^2 from tokens.



```
if(tokens[i].find("x")!=string::npos && tokens[i].find("^")!=string::npos){
        string coOfX="";
        if(i!=0)
           coOfX=tokens[i-1];
        if(tokens[i].substr(0,tokens[i].find("x"))=="")
           coOfX+="1";
        else
           coOfX+=tokens[i].substr(0,tokens[i].find("x"));
        coefficientsOfXsquare.push_back(stod(coOfX));
}
```

Figure 9.2.b

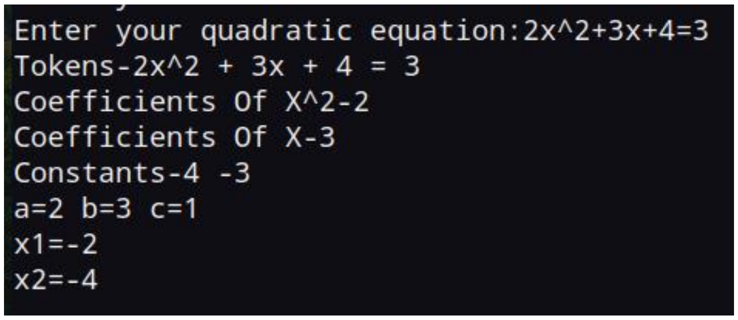
Roots are determined in this section using the formula-



```
double determinant=(b*b-4*a*c);
if(determinant==0){
        cout <<"Both of the roots are same!"<<endl;
        cout <<"x="<<((-b)/2*a)<<endl;
}
else if(determinant>0){
        cout << "x1="<<(((-b)+pow(determinant,0.5))/2*a)<<endl;
        cout << "x2="<<(((-b)-pow(determinant,0.5))/2*a)<<endl;
}
else
     cout<<"Roots are complex!";
```

Figure 9.2.c

Sample Input/Output:

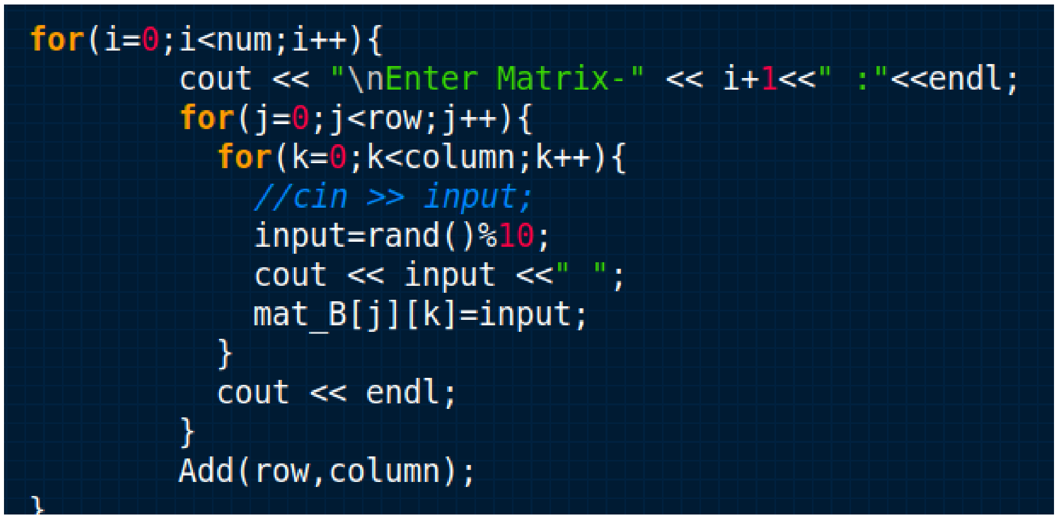


```
Enter your quadratic equation:2x^2+3x+4=3
Tokens-2x^2 + 3x + 4 = 3
Coefficients Of X^2-2
Coefficients Of X-3
Constants-4 -3
a=2 b=3 c=1
x1=-2
x2=-4
```

Figure 9.2.d

10 Matrix Addition

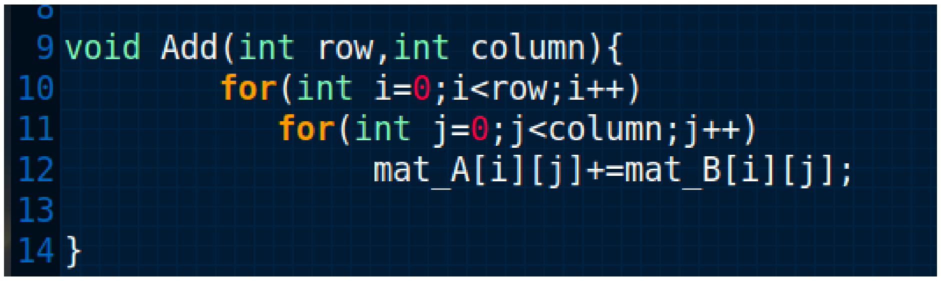
The addition of matrices is an operation on matrices where corresponding elements of two or more matrices are added. Matrices can be added only if they are of the same size, that is, they have the same dimension or order.



```
for(i=0;i<num;i++){
        cout << "\nEnter Matrix-" << i+1<<" :"<<endl;
        for(j=0;j<row;j++){
          for(k=0;k<column;k++){
            //cin >> input;
            input=rand()%10;
            cout << input <<" ";
            mat_B[j][k]=input;
          }
          cout << endl;
        }
        Add(row,column);
}
```

Figure 10.1

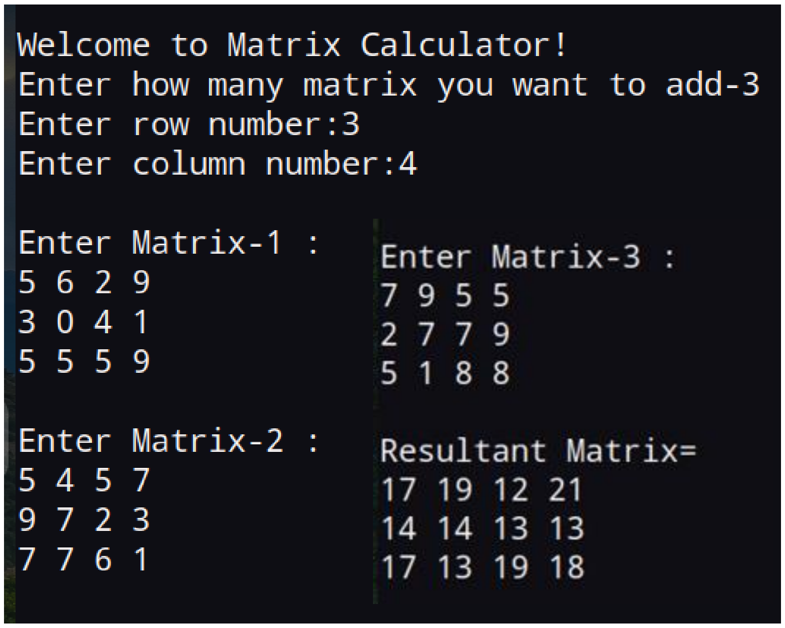
The program can take user input, for testing it takes dimensions of matrices and randomly generates all the entries of the matrix and stores them into mat_B. After taking input for a matrix, it runs the Add function which adds current matrix according to row and column to existing the mat_A,which stores the cumulative sum of all matrices.



```
void Add(int row,int column){
        for(int i=0;i<row;i++)
            for(int j=0;j<column;j++)
                mat_A[i][j]+=mat_B[i][j];

}
```

Figure 10.2

Sample Input/Output:



```
Welcome to Matrix Calculator!
Enter how many matrix you want to add-3
Enter row number:3
Enter column number:4

Enter Matrix-1 :
5 6 2 9
3 0 4 1
5 5 5 9

Enter Matrix-2 :
5 4 5 7
9 7 2 3
7 7 6 1

Enter Matrix-3 :
7 9 5 5
2 7 7 9
5 1 8 8

Resultant Matrix=
17 19 12 21
14 14 13 13
17 13 19 18
```

Figure 10.3

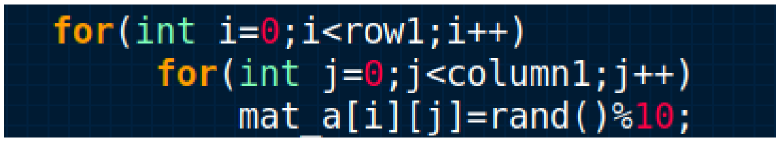
11 Matrix Multiplication

The product of two matrices A and B is defined if the number of columns of A is equal to the number of rows of B. Let A = $[a_{ij}]$ be an m × n matrix and B = $[b_{jk}]$ be an n × p matrix. Then the product of the matrices A and B is the matrix C of order m × p.

To get the $(i,k)^{th}$ element c_{ik} of the matrix C, we take the i^{th} row of A and k^{th} column of B, multiply them element-wise and take the sum of all these products.

$$AB=[c_{ij}],\text{ where } c_{ij}=a_{i1}b_{1j}+ a_{i2}b_{2j} +\ldots+ a_{in}b_{nj}$$

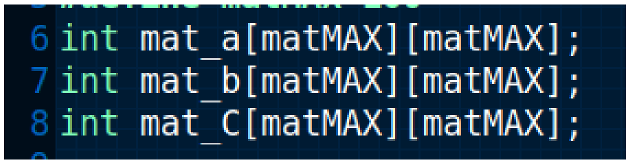
The entry in the *i*th row and *j*th column is denoted by the double subscript notation a_{ij}.
The program first takes input from user, how many matrices he wants to multiply. Then it asks the user to input the dimensions of the matrix and the matrix itself. We cannot just produce or introduce new variables as the number of matrices increases thus we take 4 variables to store the dimension of the current two matrices. Firstly, it takes input the dimension of 1st matrix as row1 and column 1, then randomly generates entries of that matrix.



```
for(int i=0;i<row1;i++)
        for(int j=0;j<column1;j++)
              mat_a[i][j]=rand()%10;
```

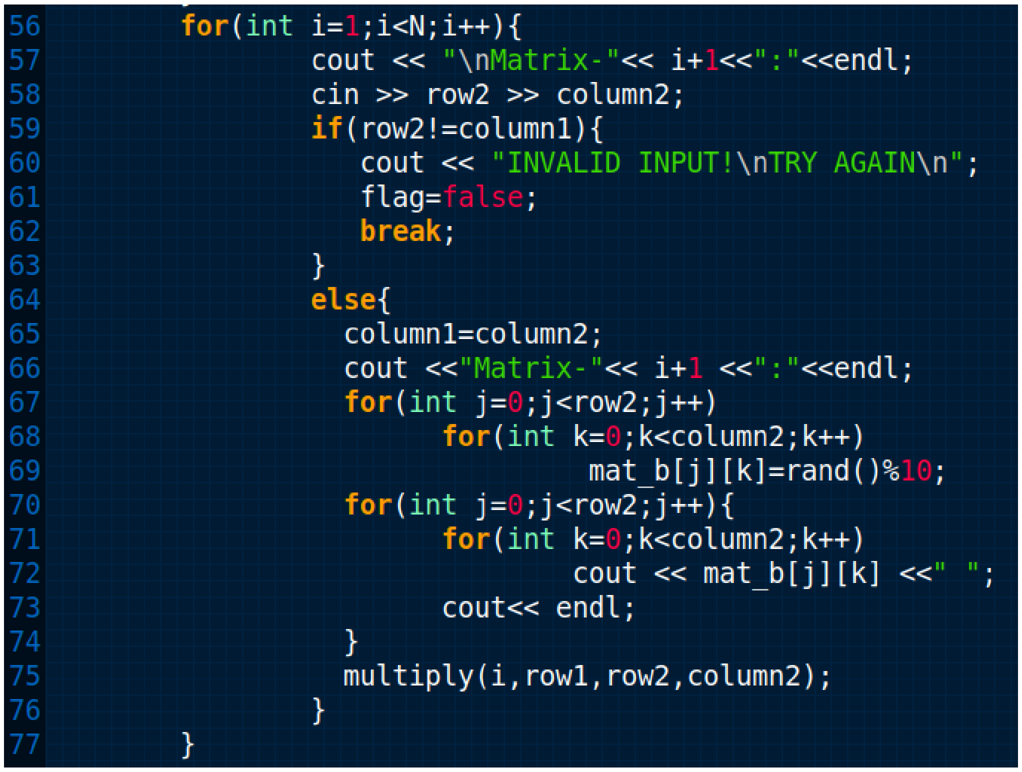
Figure 11.1

Then it asks the user to enter dimensions of the next matrices and if the column number of the previous matrix doesn't match with the row number of the current matrix, it shows error and redirects user to input valid dimensions. When matched, it replaces the column number of previous matrix with the column number of current matrix and randomly generates the current matrix using rand() function.



```
int mat_a[matMAX][matMAX];
int mat_b[matMAX][matMAX];
int mat_C[matMAX][matMAX];
```

Figure 11.2

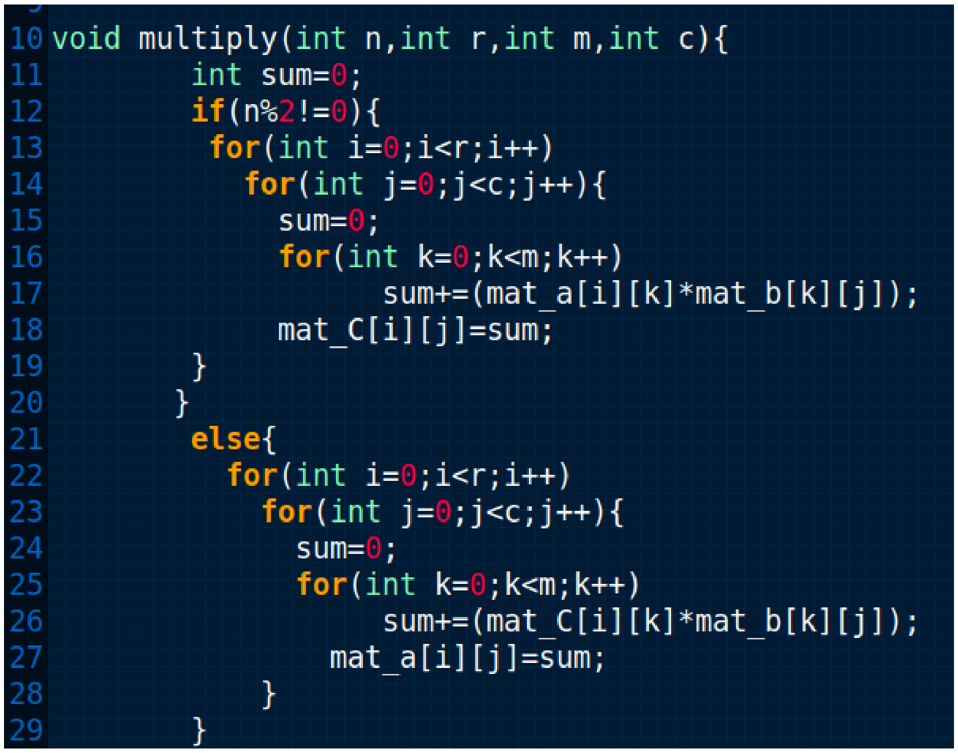


```
        for(int i=1;i<N;i++){
                cout << "\nMatrix-"<< i+1<<":"<<endl;
                cin >> row2 >> column2;
                if(row2!=column1){
                   cout << "INVALID INPUT!\nTRY AGAIN\n";
                   flag=false;
                   break;
                }
                else{
                  column1=column2;
                  cout <<"Matrix-"<< i+1 <<":"<<endl;
                  for(int j=0;j<row2;j++)
                        for(int k=0;k<column2;k++)
                                mat_b[j][k]=rand()%10;
                  for(int j=0;j<row2;j++){
                        for(int k=0;k<column2;k++)
                                cout << mat_b[j][k] <<" ";
                        cout<< endl;
                  }
                  multiply(i,row1,row2,column2);
                }
        }
```

Figure 11.3

New matrix is always stored in mat_b. Now there lies a tricky part, whenever we multiply two matrices, we always need to store that multiplication to a new matrix as we cannot edit the given two matrices. The program cannot generate 2D arrays by itself when the number of matrices is more than 2 thus we take three 2D arrays. One of them, mat_b stores given matrices from 2 to n. Other two matrices are used to store the multiplied matrix.

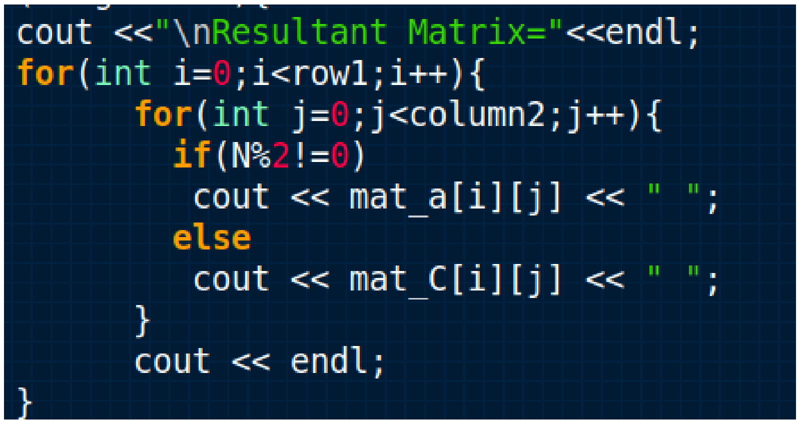
Suppose, the user want to multiply 5 matrices, So N=5. At i=0, user enters the 1st matrix, program stores that into mat_a. At i=1, user enters the 2nd matrix, program stores that into mat_b and runs the multiply function passing the value of i as an argument. Now i=1,which is odd, so the multiplication of mat_a and mat_b is stored in mat_c. Then i=2, user enters the 3rd matrix, program stores that into mat_b and runs the multiply function with i=2, which is even, thus now the previously calculated mat_c and new mat_b is multiplied and stored into mat_a. That's how we don't need more than 3 matrices to multiply multiple matrices.



```
void multiply(int n,int r,int m,int c){
        int sum=0;
        if(n%2!=0){
         for(int i=0;i<r;i++)
           for(int j=0;j<c;j++){
             sum=0;
             for(int k=0;k<m;k++)
                   sum+=(mat_a[i][k]*mat_b[k][j]);
             mat_C[i][j]=sum;
        }
       }
        else{
          for(int i=0;i<r;i++)
            for(int j=0;j<c;j++){
              sum=0;
              for(int k=0;k<m;k++)
                   sum+=(mat_C[i][k]*mat_b[k][j]);
                mat_a[i][j]=sum;
            }
        }
```

Figure 11.4

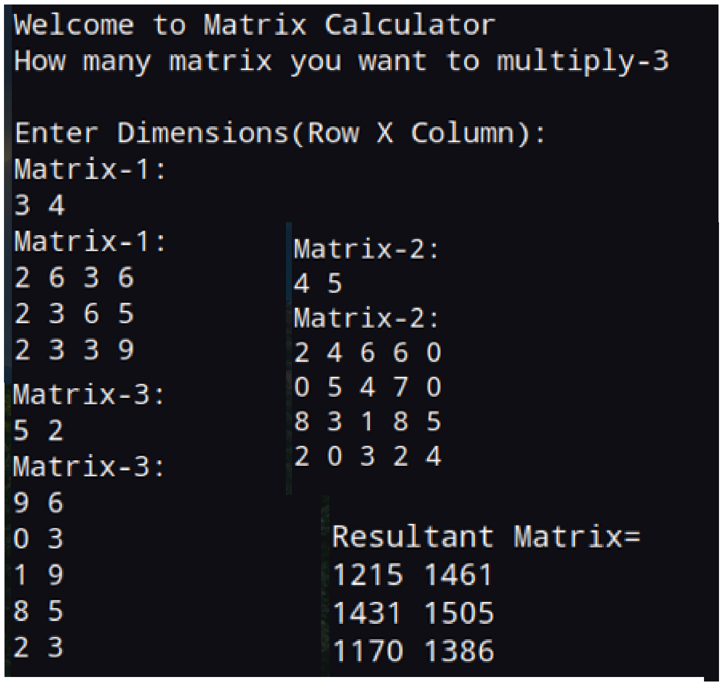
Now if we think logically, when the number of matrices multiplied is odd(greater than 1), the resultant matrix will be stored in mat_a, otherwise in mat_b.



```
cout <<"\nResultant Matrix="<<endl;
for(int i=0;i<row1;i++){
      for(int j=0;j<column2;j++){
        if(N%2!=0)
         cout << mat_a[i][j] << " ";
        else
         cout << mat_C[i][j] << " ";
      }
      cout << endl;
}
```

Figure 11.5

Sample Input/Output:



```
Welcome to Matrix Calculator
How many matrix you want to multiply-3

Enter Dimensions(Row X Column):
Matrix-1:
3 4
Matrix-1:
2 6 3 6
2 3 6 5
2 3 3 9
Matrix-3:
5 2
Matrix-3:
9 6
0 3
1 9
8 5
2 3
```

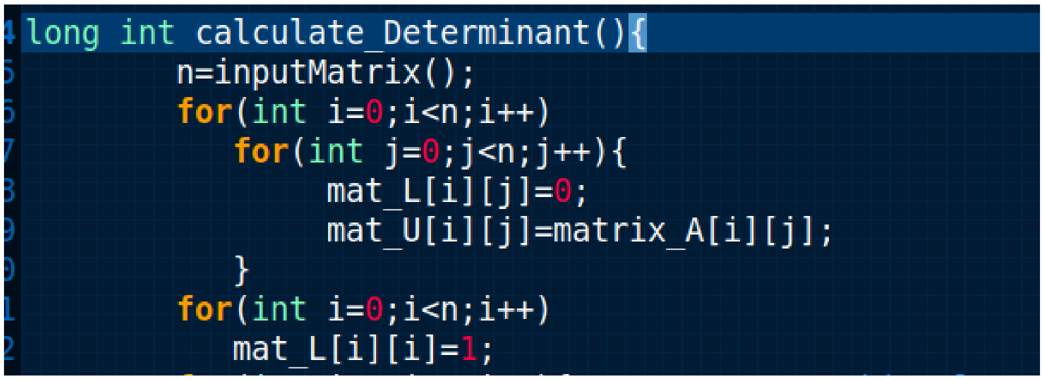
```
Matrix-2:
4 5
Matrix-2:
2 4 6 6 0
0 5 4 7 0
8 3 1 8 5
2 0 3 2 4

Resultant Matrix=
1215 1461
1431 1505
1170 1386
```

Figure 11.6

12 Matrix Determinant

The *determinant* of a matrix is a single numerical value which is used when calculating the inverse or when solving systems of linear equations. The determinant of a matrix A is denoted |A|, or sometimes det(A). The determinant is only defined for square matrices.

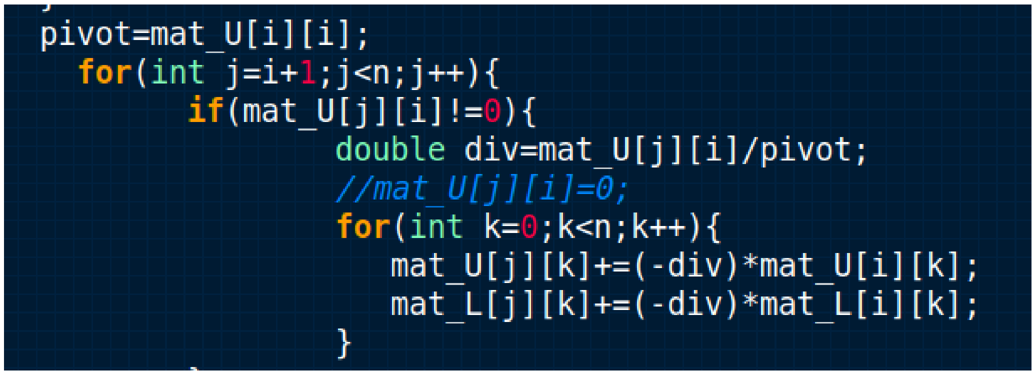
We calculated the determinant using Gauss-Elimination method. As we always need determinant to calculate inverse of a matrix thus we are taking two matrices, mat_L and mat_U. In mat_U, the program stores the given matrix and mat_L is just an identity matrix.



```
long int calculate_Determinant(){
        n=inputMatrix();
        for(int i=0;i<n;i++)
           for(int j=0;j<n;j++){
                mat_L[i][j]=0;
                mat_U[i][j]=matrix_A[i][j];
           }
        for(int i=0;i<n;i++)
           mat_L[i][i]=1;
```

Figure 12.1

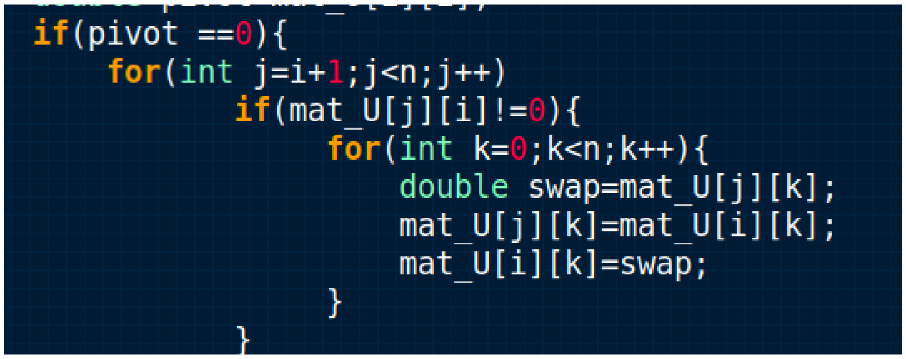
The program then used row reduction algorithm to transform the given matrix into an upper triangular matrix. For that, it takes diagonal element of every row as a pivot and with scaler multiplication with pivot, it makes all the entries of column below that pivot to zero.



```
pivot=mat_U[i][i];
  for(int j=i+1;j<n;j++){
        if(mat_U[j][i]!=0){
                double div=mat_U[j][i]/pivot;
                //mat_U[j][i]=0;
                for(int k=0;k<n;k++){
                   mat_U[j][k]+=(-div)*mat_U[i][k];
                   mat_L[j][k]+=(-div)*mat_L[i][k];
                }
```

Figure 12.2

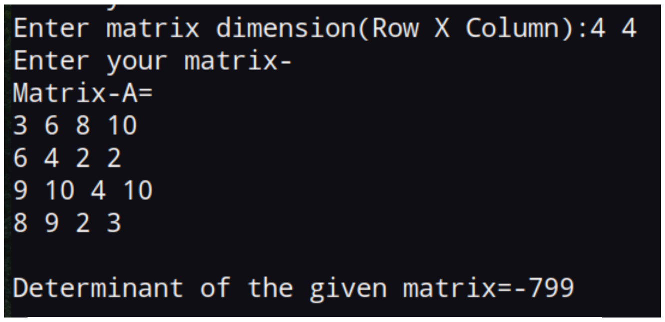
If the pivot equals to zero, the program checks for 1st non-zero element in that column below the pivot and when found, the pivot row and that row is exchanged.



```
if(pivot ==0){
    for(int j=i+1;j<n;j++)
          if(mat_U[j][i]!=0){
                for(int k=0;k<n;k++){
                    double swap=mat_U[j][k];
                    mat_U[j][k]=mat_U[i][k];
                    mat_U[i][k]=swap;
                }
          }
```

Figure 12.3

After transforming the matrix into upper triangular, the product of the diagonal entries of mat_L is the determinant of the given matrix.
Sample Input/Output:

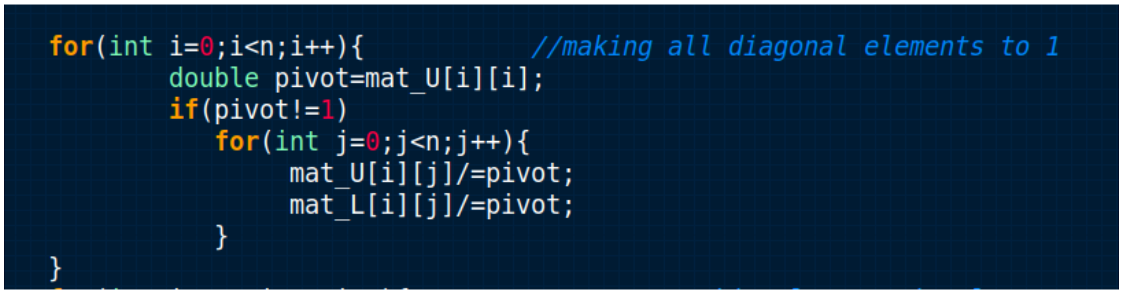


```
Enter matrix dimension(Row X Column):4 4
Enter your matrix-
Matrix-A=
3 6 8 10
6 4 2 2
9 10 4 10
8 9 2 3

Determinant of the given matrix=-799
```

Figure 12.4

13 Matrix Inversion

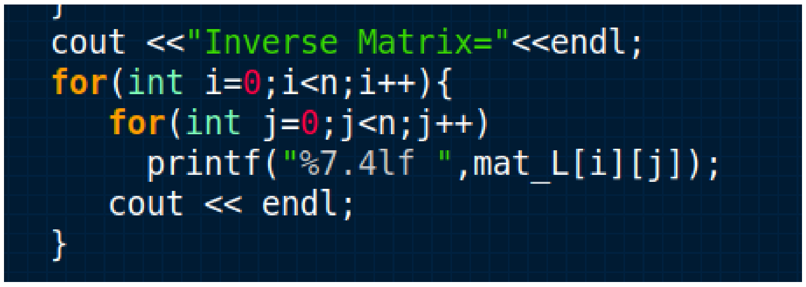
Inverse of a Matrix is a matrix that on multiplying with the original matrix result in the identity matrix. It is required to solve complex problems using matrix operations. For any matrix A its inverse is denoted as A^{-1}. Inverses only exist for square matrices. The program follows Gauss-Jordan Elimination method to find inverse of a matrix. For calculating inverse of a matrix, we need to make sure the determinant of that matrix is not zero. So firstly, the program transforms the given matrix into upper triangular and the scaler operations needed to that also applied to the mat_U, which later will be the inverse matrix. Then it converts all diagonal entries to 1 by dividing the entire row by the pivot, same operations are done in mat_U.



```
for(int i=0;i<n;i++){           //making all diagonal elements to 1
        double pivot=mat_U[i][i];
        if(pivot!=1)
           for(int j=0;j<n;j++){
                mat_U[i][j]/=pivot;
                mat_L[i][j]/=pivot;
           }
}
```

Figure 13.1

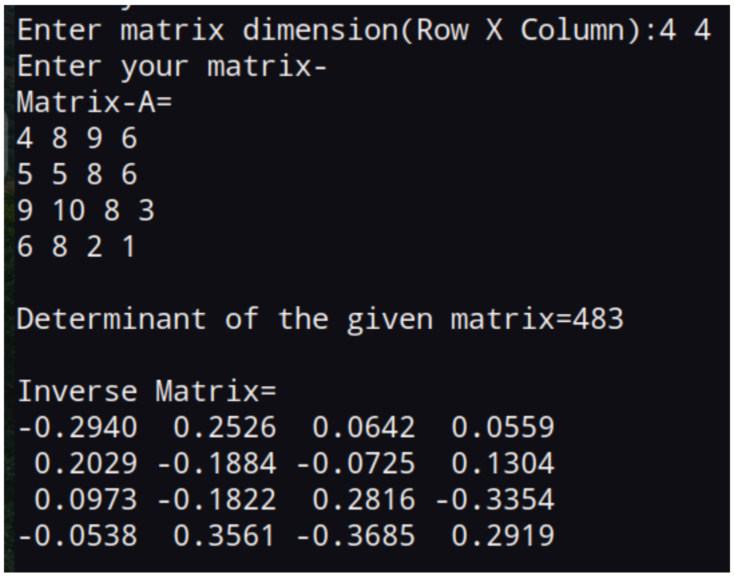
Finally, the program transforms the mat_U into an Identity matrix by making its upper triangular elements zero. As all the operations are also applied to mat_L, by then it is the resultant Inverse of the given matrix.



```
    cout <<"Inverse Matrix="<<endl;
    for(int i=0;i<n;i++){
       for(int j=0;j<n;j++)
         printf("%7.4lf ",mat_L[i][j]);
       cout << endl;
    }
```

Figure 13.2

Sample Input/Output:



```
Enter matrix dimension(Row X Column):4 4
Enter your matrix-
Matrix-A=
4 8 9 6
5 5 8 6
9 10 8 3
6 8 2 1

Determinant of the given matrix=483

Inverse Matrix=
-0.2940  0.2526  0.0642  0.0559
 0.2029 -0.1884 -0.0725  0.1304
 0.0973 -0.1822  0.2816 -0.3354
-0.0538  0.3561 -0.3685  0.2919
```

Figure 13.3

14 Matrix Power Calculation

Matrix power is obtained by multiplication matrix by itself 'n' times. The matrix must be square in order to raise it to a power.

If A is a square matrix and K is a positive integer, the kth power of A is given by

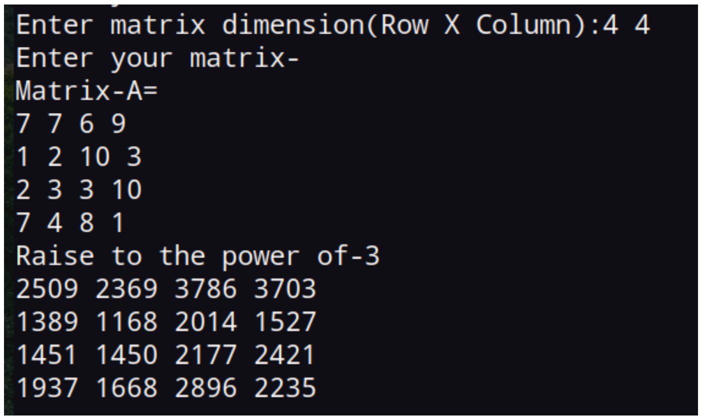
$A^K = A \times A \times A \times ....\times A$, where there are K copies of matrix A.

It is possible to compute A^k using A^{k-1},

$A^k = A^{k-1} \times A$

For instance, $A^4 = A^3 \times A$. Thus we applied matrix multiplication here to calculate A^n.

Sample Input/Output:



```
Enter matrix dimension(Row X Column):4 4
Enter your matrix-
Matrix-A=
7 7 6 9
1 2 10 3
2 3 3 10
7 4 8 1
Raise to the power of-3
2509 2369 3786 3703
1389 1168 2014 1527
1451 1450 2177 2421
1937 1668 2896 2235
```

Figure 14

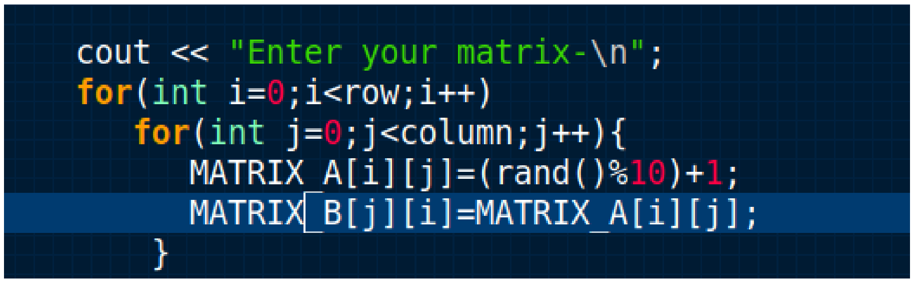
15 Matrix Transposition

Transpose of a Matrix is defined as "A Matrix which is formed by turning all the rows of a given matrix into columns and columns into rows."

General statement for matrix transpose:

If $A=[a_{ij}]_{m\times n}$, then $A' = [a_{ij}]_{n\times m}$

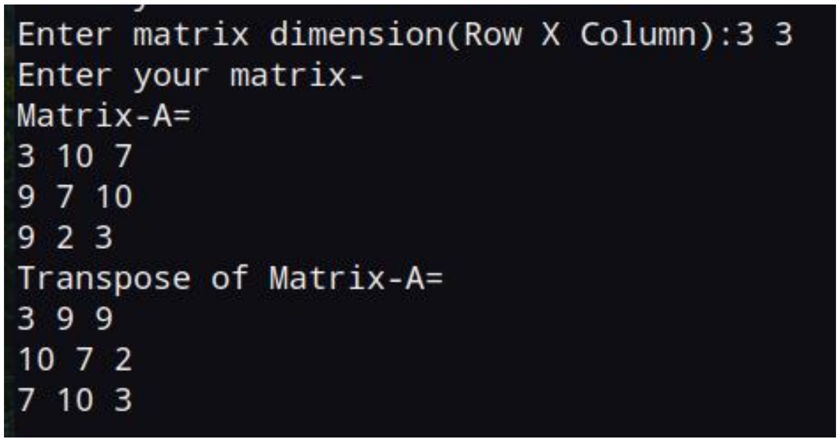
The program randomly generates MATRIX_A[i][j] entries and stores it into MATRIX_B[j][i].



```
    cout << "Enter your matrix-\n";
    for(int i=0;i<row;i++)
       for(int j=0;j<column;j++){
          MATRIX_A[i][j]=(rand()%10)+1;
          MATRIX_B[j][i]=MATRIX_A[i][j];
        }
```

Figure 15.1

Sample Input/Output:



```
Enter matrix dimension(Row X Column):3 3
Enter your matrix-
Matrix-A=
3 10 7
9 7 10
9 2 3
Transpose of Matrix-A=
3 9 9
10 7 2
7 10 3
```

Figure 15.2

16 Matrix Eigenvalues and Eigenvectors

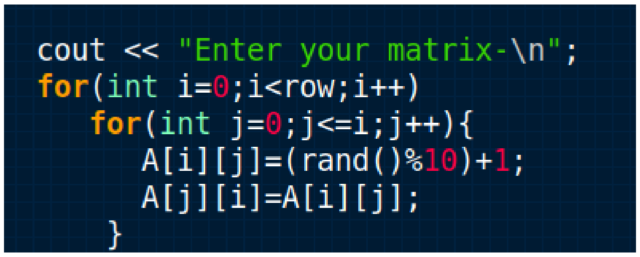
Eigenvalues are the special set of scalars associated with the system of linear equations. It is mostly used in matrix equations. The basic equation is

$$\mathbf{Ax = \lambda x}$$

The number or scalar value "**λ**" is an eigenvalue of A.

In Mathematics, an eigenvector corresponds to the real non zero eigenvalues which point in the direction stretched by the transformation whereas eigenvalue is considered as a factor by which it is stretched. In case, if the eigenvalue is negative, the direction of the transformation is negative. For every real matrix, there is an eigenvalue. Sometimes it might be complex.

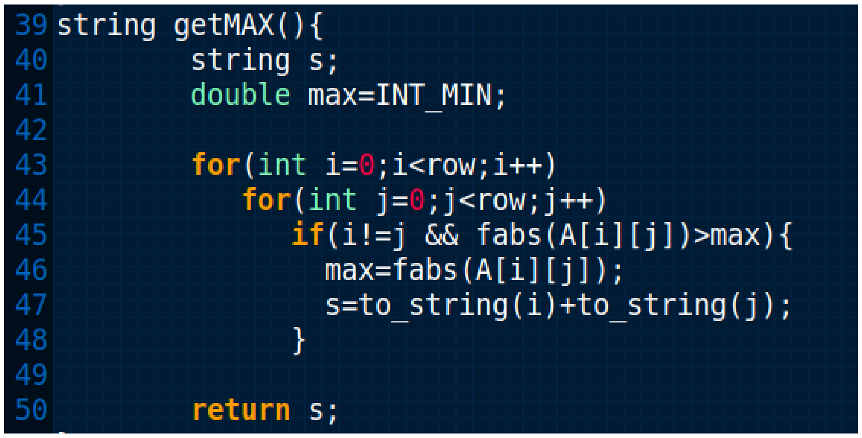
The program follows the Jacobi algorithm to calculate the eigenvalues and corresponding eigenvectors. Jacobi algorithm works on Symmetric matrix, so it randomly generates symmetric matrix by given dimensions.



```
cout << "Enter your matrix-\n";
for(int i=0;i<row;i++)
   for(int j=0;j<=i;j++){
      A[i][j]=(rand()%10)+1;
      A[j][i]=A[i][j];
   }
```

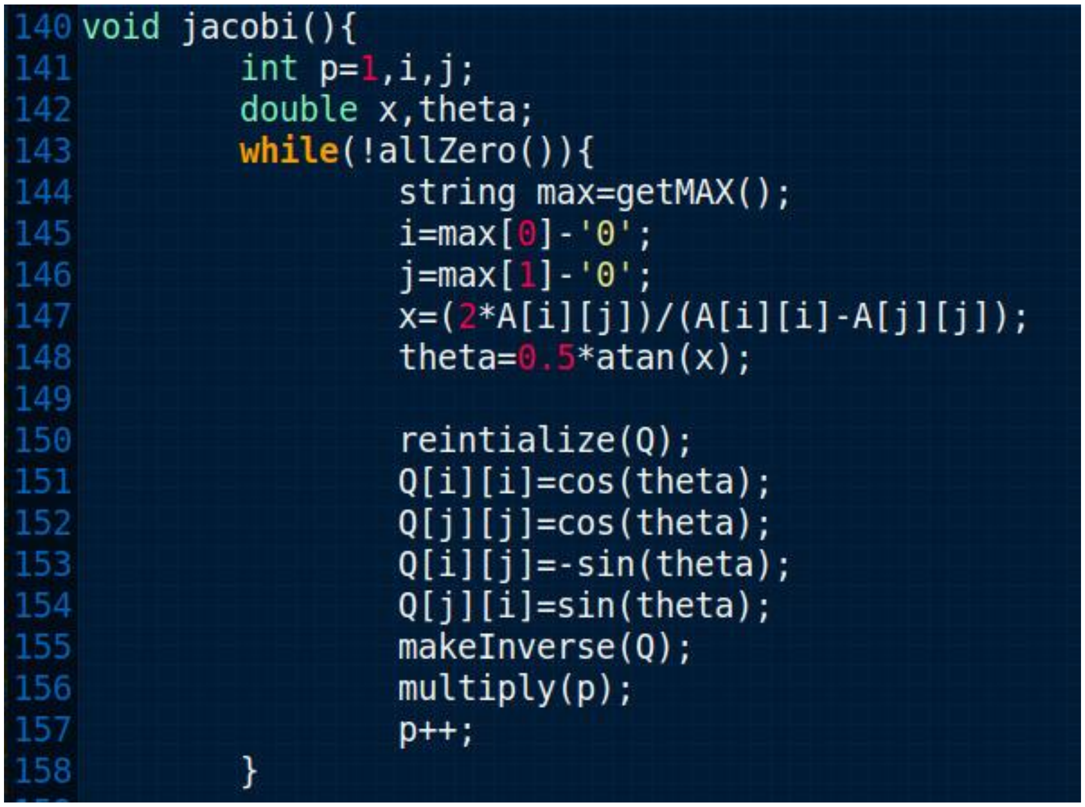
Figure 16.1

Firstly, the algorithm requires the largest absolute value among the entries along with its index. So it creates a function getMAX to retrieve the index of the largest value ignoring sign.



```
string getMAX(){
        string s;
        double max=INT_MIN;

        for(int i=0;i<row;i++)
           for(int j=0;j<row;j++)
              if(i!=j && fabs(A[i][j])>max){
                max=fabs(A[i][j]);
                s=to_string(i)+to_string(j);
              }

        return s;
```

Figure 16.2

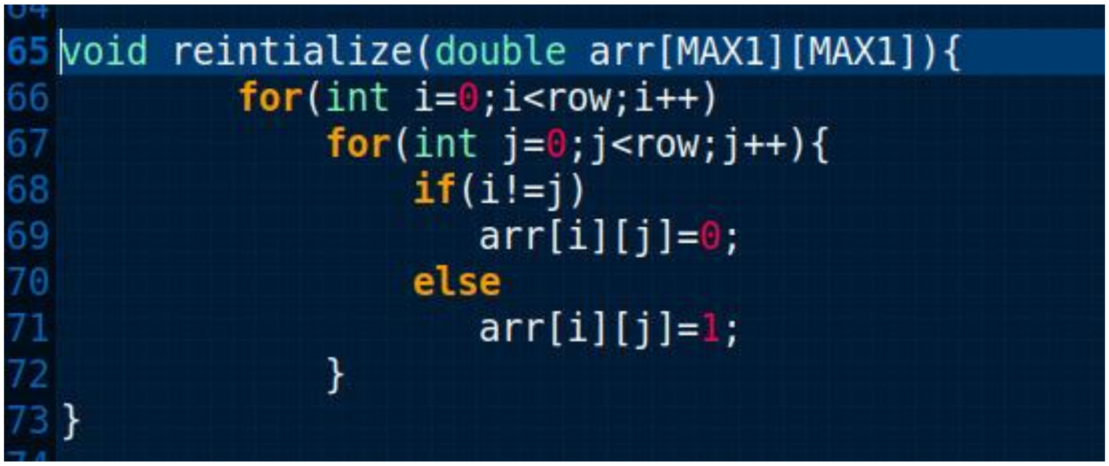


```
140 void jacobi(){
141         int p=1,i,j;
142         double x,theta;
143         while(!allZero()){
144                 string max=getMAX();
145                 i=max[0]-'0';
146                 j=max[1]-'0';
147                 x=(2*A[i][j])/(A[i][i]-A[j][j]);
148                 theta=0.5*atan(x);
149
150                 reintialize(Q);
151                 Q[i][i]=cos(theta);
152                 Q[j][j]=cos(theta);
153                 Q[i][j]=-sin(theta);
154                 Q[j][i]=sin(theta);
155                 makeInverse(Q);
156                 multiply(p);
157                 p++;
158         }
```

Figure 16.3

Then it calculates theta

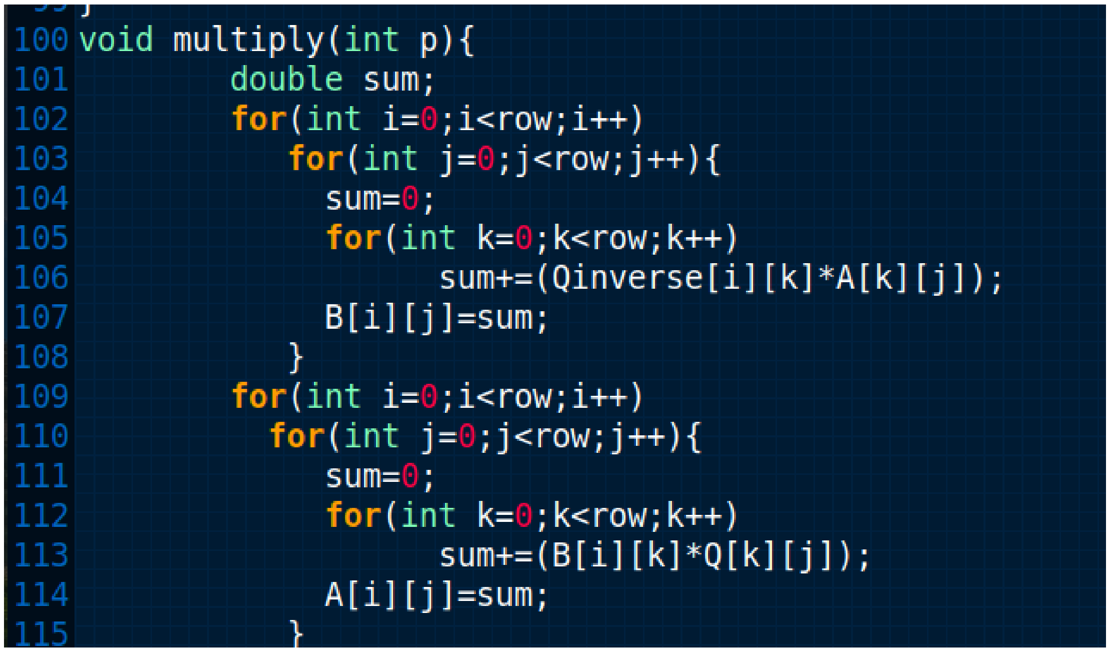
$$\tan(2\theta) = \frac{2S_{ij}}{S_{jj} - S_{ii}}$$ using this formula from line 144-158.



```
void reintialize(double arr[MAX1][MAX1]){
        for(int i=0;i<row;i++)
            for(int j=0;j<row;j++){
                if(i!=j)
                    arr[i][j]=0;
                else
                    arr[i][j]=1;
            }
}
```

Figure 16.4

In the above, the program reinitializes the 2D array Q as an identity matrix. And then transform this into an orthogonal matrix/rotational matrix using operations from line 151-154. Then it calculates the inverse of matrix Q using makeInverse function. Inverse of an orthogonal matrix is just the transpose of that matrix.



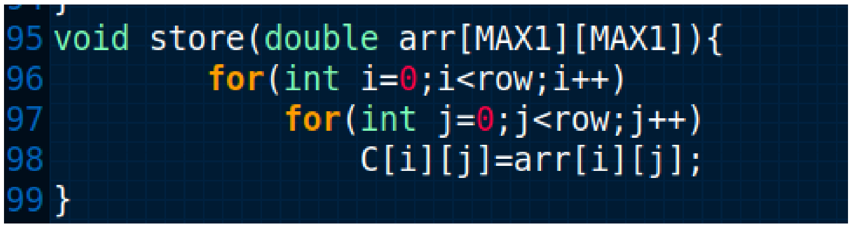
```
void multiply(int p){
        double sum;
        for(int i=0;i<row;i++)
           for(int j=0;j<row;j++){
             sum=0;
             for(int k=0;k<row;k++)
                   sum+=(Qinverse[i][k]*A[k][j]);
             B[i][j]=sum;
           }
        for(int i=0;i<row;i++)
          for(int j=0;j<row;j++){
             sum=0;
             for(int k=0;k<row;k++)
                   sum+=(B[i][k]*Q[k][j]);
             A[i][j]=sum;
           }
```

Figure 16.5

The above multiplication is performed according to this formula-

$$A' = QAQ^T$$

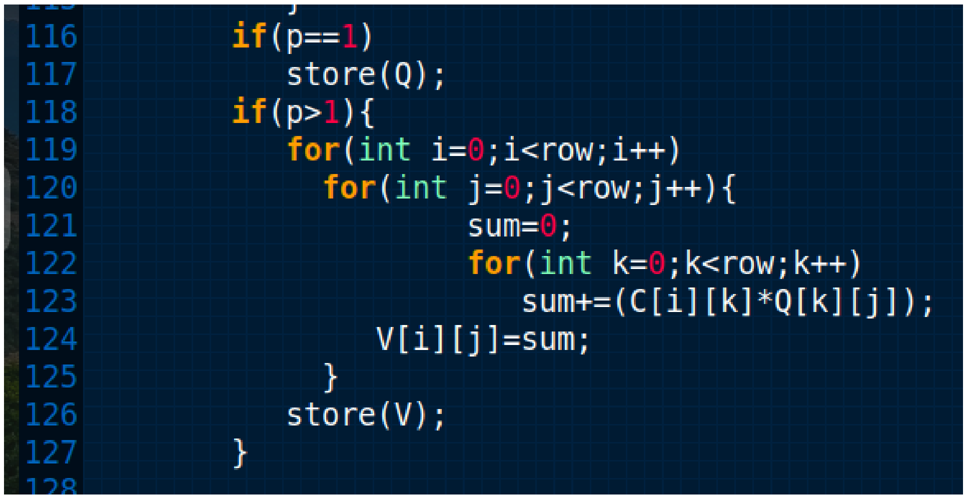
To get the corresponding eigenvectors, the program stores the product of Q matrices in C matrix. For instance, 1st iteration, $C=Q_1$. 2nd iteration, $C=Q_1\times Q_2$. It goes on until we find matrix A where all non-diagonal elements aro zero or less than its modulas value is $<10^{-6}$.



```
void store(double arr[MAX1][MAX1]){
        for(int i=0;i<row;i++)
            for(int j=0;j<row;j++)
                C[i][j]=arr[i][j];
}
```

Figure 16.6

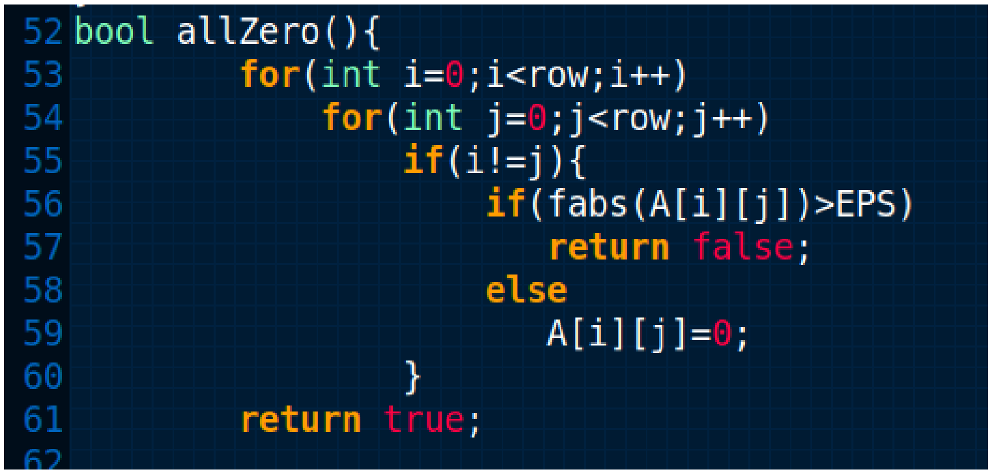
This part of the program calculates the cumulative product of Q matrices and stores it.



```
        if(p==1)
            store(Q);
        if(p>1){
            for(int i=0;i<row;i++)
              for(int j=0;j<row;j++){
                      sum=0;
                      for(int k=0;k<row;k++)
                        sum+=(C[i][k]*Q[k][j]);
                  V[i][j]=sum;
              }
            store(V);
        }
```

Figure 16.7

This function checks if all the non-diagonal entries are zero or not.



```
bool allZero(){
        for(int i=0;i<row;i++)
            for(int j=0;j<row;j++)
                if(i!=j){
                    if(fabs(A[i][j])>EPS)
                        return false;
                    else
                        A[i][j]=0;
                }
        return true;
```

Figure 16.8

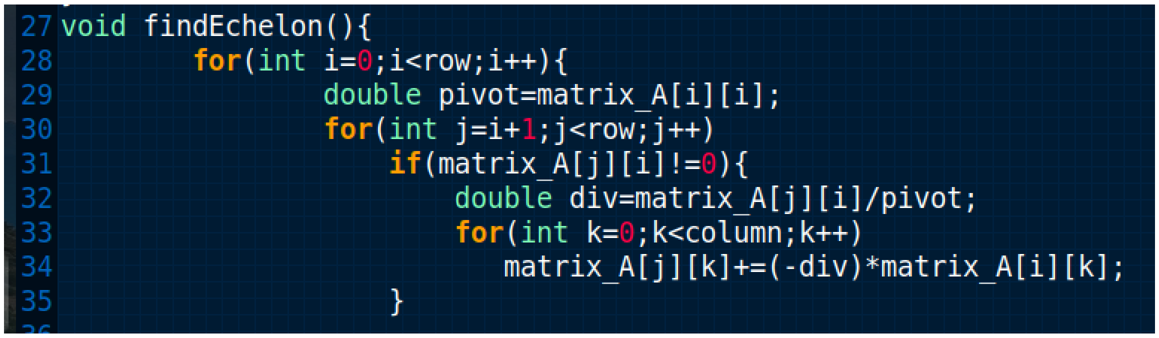
17 Matrix Rank

The rank of the matrix refers to the number of linearly independent rows or columns in the matrix. The program finds the Rank of a Matrix using the Echelon Form.

A matrix 'A' is said to be in Echelon form if it is either in upper triangular form or in lower triangular form. We can use elementary row/column transformations and convert the matrix into Echelon form.

A row (or column) transformation can be one of the following:

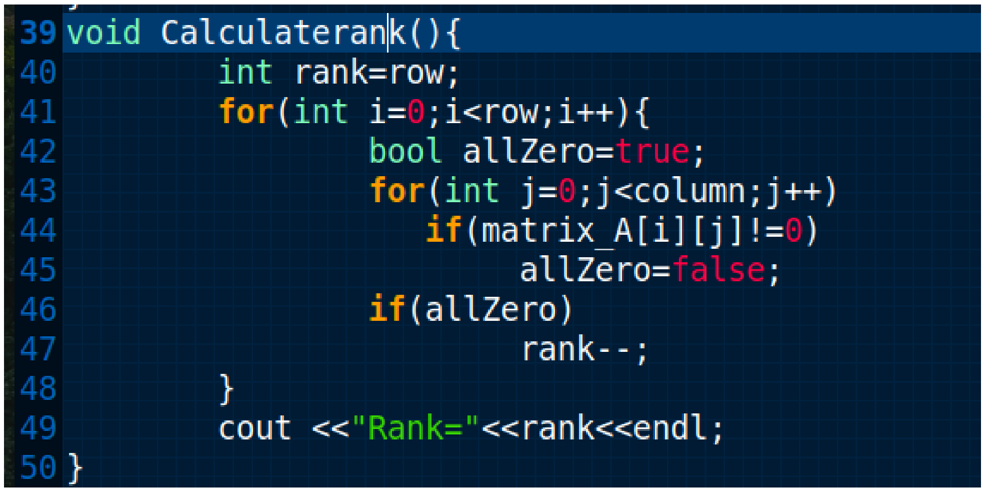
- Interchanging two rows.
- Multiplying a row by a scalar.
- Multiplying a row by a scalar and then adding it to the other row.



```
void findEchelon(){
        for(int i=0;i<row;i++){
                double pivot=matrix_A[i][i];
                for(int j=i+1;j<row;j++)
                    if(matrix_A[j][i]!=0){
                        double div=matrix_A[j][i]/pivot;
                        for(int k=0;k<column;k++)
                           matrix_A[j][k]+=(-div)*matrix_A[i][k];
                    }
```

Figure 17.1

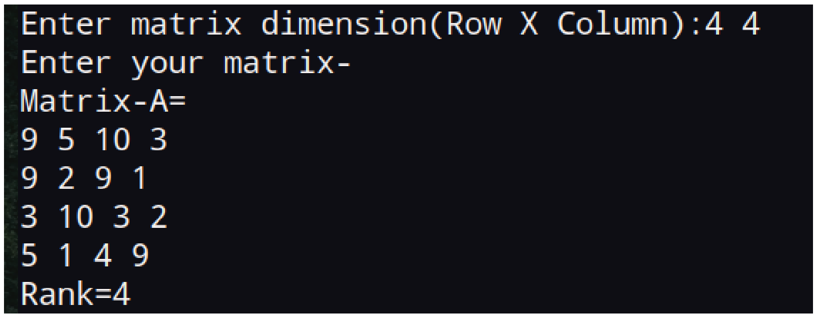
It converts the matrix into Echelon form using row/column transformations. Then the rank of the matrix is equal to the number of non-zero rows in the resultant matrix.



```
void Calculaterank(){
        int rank=row;
        for(int i=0;i<row;i++){
                bool allZero=true;
                for(int j=0;j<column;j++)
                   if(matrix_A[i][j]!=0)
                        allZero=false;
                if(allZero)
                        rank--;
        }
        cout <<"Rank="<<rank<<endl;
}
```

Figure 17.2

Sample Input/Output:



```
Enter matrix dimension(Row X Column):4 4
Enter your matrix-
Matrix-A=
9 5 10 3
9 2 9 1
3 10 3 2
5 1 4 9
Rank=4
```

Figure 17.3

Challenges Faced

Due to a lack of resources, standardizing linear and quadratic equations and implementing tokenization became difficult. Implementing other complex matrix operations was also challenging.

Ensuring a user-friendly interface and comprehensive documentation was crucial for the library's accessibility. Overcoming this challenge involved conducting user testing, gathering feedback, and refining the documentation to provide clear and concise guidance. Regular updates and communication with the user community allowed me to address user concerns promptly and improve the overall user experience.

Conclusion

The journey of developing Numeric Operations Library has been an ongoing process of refinement, learning from challenges, and implementing innovative solutions. The commitment to continuous improvement and adaptability has been key to overcoming obstacles and ensuring that Numeric Operations Library remains a reliable and powerful tool for mathematicians, researchers, and students alike.

References

https://www.encyclopedia.com/science/encyclopedias-almanacs-transcripts-and-maps/sieve-eratosthenes-0

https://byjus.com/maths/prime-factorization/

https://www.khanacademy.org/computing/computer-science/cryptography/modarithmetic/a/the-euclidean-algorithm

https://www.idomaths.com/hcflcm.php#google_vignette

https://www.prepbytes.com/blog/stacks/expression-evaluation/

https://brilliant.org/wiki/shunting-yard-algorithm/

https://www.cuemath.com/algebra/addition-of-matrices/

https://www.toppr.com/guides/maths/matrices/multiplication-of-matrices/

https://www.ncl.ac.uk/webtemplate/ask-assets/external/maths-resources/core-mathematics/pure-maths/matrices/matrix-determinant.html

https://people.richland.edu/james/lecture/m116/matrices/inverses.html

https://www.nagwa.com/en/explainers/432180315293/

https://www.vedantu.com/maths/matrix-rank

https://www.cmi.ac.in/~ksutar/NLA2013/iterativemethods.pdf

https://en.wikipedia.org/wiki/Jacobi_eigenvalue_algorithm#:~:text=In%20numerical%20linear%20algebra%2C%20the,a%20process%20known%20as%20diagonalization).