Matrix and Fibonacci Algorithms

Introduction

This report includes functions for matrix calculations and explanations for these functions, as well as functions to calculate the terms of Fibonacci series both with recursive and iterative methods, as well as showing the time difference between these methods when they are utilised.

Matrix Program

A struct with a name myMat is used to represent matrices. This struct has two integer variables numRows and numCols to store the size of the matrix, which can go up to 4x4 maximum. It also has an array of integers named data to store the actual elements of the array. Row vectors can be represented if the numRows valuable is set to 1 and column vectors can be represented if the numCols variable is set to 1. This struct is used to represent 2D matrices.

1 - Function to create a matrix being a column vector from given matrix: Gets the column in the specified index end returns it as a 1 column matrix.

```
myMat mGetCol(myMat m, int col) {
    myMat res = zeroMat(m.numRows, 1); //Empty matrix of 1 column in size
    for (int row = 0; row < m.numRows; row++) //Iterate through the elements in column
        res.data[row] = getElem(m, row, col); //For each row, copy the element
    return res;
}
```

2 - Function to calculate the dot product of two vectors: Calculates the dot product of 2 vectors given. Returns the result as an integer value.

```
int dotProd(myMat v1, myMat v2) {
    int res = 0;
    for (int ct = 0; ct < v1.numRows; ct++) //For each row in the vectors
        res += v1.data[ct] * v2.data[ct]; //Add the product of the corresponding elements to the res
value
    return res;
}
```

3 - Function to create a matrix which is the transpose of given matrix: Takes a matrix and returns the transpose of it.

```
myMat mTranspose(myMat m) {
    myMat res = zeroMat(m.numCols, m.numRows); //Empty matrix with the number of rows as the original's
number of columns and columns as the original's rows
    for (int col = 0; col < m.numCols; col++) //For each column of the original matrix
        for (int i = 0; i < m.numRows; i++) //For each row of the original matrix
            res.data[col*res.numCols+i] = mGetCol(m, col).data[i]; //Get the 'col'th column's i'th data
and set the corresponding row's i'th data of the result.
    return res;
}
```

4 - Function to add two matrices: Takes 2 matrices and returns the sum of them.

```
myMat mAdd(myMat m1, myMat m2) {
    for (int ct = 0; ct < m1.numRows*m1.numCols; ct++) //For each element for the matrices
        m1.data[ct] += m2.data[ct]; //Add the corresponding element to the first matrix.
    return m1;  //Return the first matrix with added values.
}
```

5 - Function to multiply two matrices: Takes and multiplies 2 matrices and retuns the result matrix.

```
myMat mMult(myMat m1, myMat m2) {
```

```
    myMat res = zeroMat(m1.numRows, m2.numCols);    //Set an empty matrix with the size m1's rows and
m2's columns.
    int ct = 0;
    for (int m1r = 0; m1r < m1.numRows; m1r++){     //For each row of m1
        for (int m2c = 0; m2c < m2.numCols; m2c++){ //For each column of m2
            for (int i = 0; i < m1.numCols; i++){   //For each element in m1's column
                res.data[ct] += getElem(m1, m1r, i) * getElem(m2, i, m2c); //Get the corresponding values
from the corresponding rows and columns and multiply them, then add the value to the corresponding
element in the result matrix.
            }
            ct++;
        }
    }
    return res;
}
```

Outputs so far:

```
Matrix Program
A =
2       6       7
9       1       6

C =
8       7
10      10
2       9
```

```
A + C' =
10      16      9
16      11      15

A * C =
90      137
94      127

C * A =
79      55      98
110     70      130
85      21      68
```

```
Column vector from C's second column =
7
10
9

v1 =
3
6
9

v2 =
2
4
8

v1.v2 (Dot Product) =
102
```

The fallowing functions are optionals.

6 - To multiple each element in a matrix by a scalar: Takes an integer and a matrix and calculates and returns the scalar multiplication of them.

```
myMat mScalarMult(int k, myMat m){
    for (int ct = 0; ct < m.numRows*m.numCols; ct++) //For each element in the matrix
        m.data[ct] *= k; //Multiply the element by the value
    return m;
}
```

Output for scalar multiplication:

```
9A - 1C' (Scalar Multiplication, From Q1) =
10      44      61
74      -1      45
```

7 - To solve a two variable matrix equation using a 'magic' matrix:

7.1 - Finds the magic matrix for the given 2x2 matrix and returns it.

```
myMat findMagic(myMat m){
    myMat res = m; //Copies the original matrix
    setElem(res, 0, 0, -1 * getElem(m, 1, 1)); //Sets the element in 1st row 1st column to the negative
of the element in 2nd row 2nd column.
    setElem(res, 1, 1, -1 * getElem(m, 0, 0)); //Sets the element in 2nd row 2nd column to the negative
of the element in 1st row 1st column (from the original matrix).
    return res;
```

```
}
```

7.2 - Takes 2 matrices A and b. Solves and returns the x values as a matrix in the equation Ax=b by using 'magic' matrix on 2x2 matrices.

```
myMat solveByMagic(myMat A, myMat b){
    myMat res = zeroMat(2,1);
    myMat M = findMagic(A); //Finds and assigns the magic matrix for the matrix.
    myMat MA = mMult(M, A); //Assigns the product of magic matrix and A to MA
    myMat Mb = mMult(M, b); //Assigns the product of magic matrix and b to Mb
    res.data[0] = Mb.data[0]/getElem(MA, 0, 0); //Divides the 1st element of Mb to the element in the 1st row 1st column of MA and stores the result as x1
    res.data[1] = Mb.data[1]/getElem(MA, 1, 1); //Divides the 2nd element of Mb to the element in the 2nd row 2nd column of MA and stores the result as x2
    return res;
}
```

Outputs for 'magic' matrix:

```
Q2 A values =
1       6
1       10
```

```
Q2 b values =
57
89
```

```
Solving Q2 using 'magic' Matrix =
9
8
```

8 - To calculate the determinant of a 2D matrix and solve an equation using Cramer's Rule:

8.1 - Takes a matrix and an integer for the column and returns a matrix with the first row and the chosen column removed.

```
myMat subMat(myMat m, int col){
    myMat res = zeroMat(m.numRows - 1, m.numCols - 1);
    int k = 0; //'k' is to track the index of the data of the result matrix.
    for(int i = m.numCols; i <= (m.numCols * m.numRows); i++){ //Loops the data of the given matrix. 'i' is to track the index of the given matrix.
        if (i % m.numCols == col) i++; //If element is on the column to be removed increase 'i' to skip the element.
        res.data[k] = m.data[i]; //Assigns the element from the given matrix to the result matrix.
        k++; //Increases 'k' for the next iteration.
    }
    return res;
}
```

8.2 - Finds and returns the determinant of a given square matrix of any size.

```
int findDet(myMat m){
    if (m.numRows == 1){ //If the matrix has only one row
        return m.data[0]; //Returns the first element on the matrix.
    }
    else if (m.numRows == 2){ //If the matrix has 2 rows
        return ((getElem(m, 0, 0) * getElem(m, 1, 1)) - (getElem(m, 0, 1) * getElem(m, 1, 0))); /*Returns the difference between the product of the elements in 1st row 1st column and 2nd row 2nd column and the product of the elements in the 1st row 2nd column and the 2nd row 1st column.*/
    }
    else{
        int sum = 0;
        for (int i = 0; i < m.numCols; i++){ //For each element on the top row of the matrix
            sum += m.data[i] * pow(-1, i) * findDet(subMat(m,i));   /*Adds the product of the i'th element on the top row and the cofactor of that element (which is - or + determinant of the sub-matrix without the first row and the i'th column) to 'sum'.*/
        }
        return sum;
    }
}
```

8.3 - Takes 2 matrices A and b. Solves and returns the x values as a matrix in the equation Ax=b by using Cramer's Rule on square A matrices of any size.

```
myMat solveByCramer(myMat A, myMat b){
    myMat res = zeroMat(A.numRows, 1);
    int detA = findDet(A);  //Finds and assigns the determinant of the matrix
    myMat temp = A;         //Copies A to a temporary matrix
    for (int i = 0; i < A.numRows; i++){ //For each row in A
        for (int k = 0; k < A.numCols; k++){ //For each column in A
            setElem(temp, k, i, getElem(b, k, 0)); //Assigns the corresponding element on b to A
        }
        res.data[i] = findDet(temp) / detA; //Finds the determinant of temp and divides it to determinant
of A then assigns it to the corresponding element in the result
        temp = A; //Copies A to temp again for the next iteration
    }
    return res;
}
```

Outputs:

```
Q8 A values =
9       8
10      6

Q8 b values =
138
136

Solving Q8 using Cramer's Rule =
10
6
```

```
3x3 example A values =
6       7       1
10      6       8
7       10      2

3x3 example b values =
106
132
141

Solving 3x3 example using Cramer's Rule =
7
9
1
```

```
4x4 example A values =
2       9       8       9
10      6       7       13
1       10      6       3
8       14      2       19

4x4 example b values =
188
226
122
279

Solving 4x4 example using Cramer's Rule =
3
5
7
9
```

9 - The main function calling these: The main function to call the other functions.

```
int main(int argc, char *argv[]){
    cout << "Matrix Program\n";
    myMat m1, m2, m3, m4, m5, m6, m7, m8, m9, m10, m11, mA, mb, mA2, mb2, mA3, mb3, mA4, mb4, v1, v2;

    m1 = mFromStr("2,6,7;9,1,6");   //From question 1 A
    m2 = mFromStr("8,7;10,10;2,9"); //From question 1 C
    printMat("A", m1);
    printMat("C", m2);
    m3 = mAdd(m1, mTranspose(m2));
    printMat("A + C'", m3);
    m4 = mMult(m1, m2);
    printMat("A * C", m4);
    m5 = mMult(m2, m1);
    printMat("C * A", m5);

    ///To create a matrix being a column vector from given matrix
    m6 = mGetCol(m2, 1);
    printMat("Column vector from C's second column", m6);

    ///Next are random vectors to show the dot product.
    v1 = mFromStr("3;6;9"); //Random values
    v2 = mFromStr("2;4;8"); //Random values
    printMat("v1", v1);
    printMat("v2", v2);
    cout << "v1.v2 (Dot Product) =\n" << dotProd(v1, v2) << "\n\n";

    //Next are the optional functions.

    ///To multiply each element in a matrix by a scalar
    m7 = mAdd(mScalarMult(9, m1), mScalarMult(-1, mTranspose(m2)));//Values from question 1
    printMat("9A - 1C' (Scalar Multiplication, From Q1)", m7);

    ///Solving equation using 'magic' Matrix
    mA = mFromStr("1,6;1,10");  //Values from question 2
    mb = mFromStr("57;89");     //Values from question 2
    m8 = solveByMagic(mA, mb);
    printMat("Q2 A values", mA);
    printMat("Q2 b values", mb);
    printMat("Solving Q2 using 'magic' Matrix", m8);
```

```
    ///Solving equations using Cramer's Rule
    mA2 = mFromStr("9,8;10,6"); //Values from question 8
    mb2 = mFromStr("138;136");  //Values from question 8
    m9 = solveByCramer(mA2, mb2);
    printMat("Q8 A values", mA2);
    printMat("Q8 b values", mb2);
    printMat("Solving Q8 using Cramer's Rule", m9);

    mA3 = mFromStr("6,7,1;10,6,8;7,10,2");  //Example values for 3x3
    mb3 = mFromStr("106;132;141");          //Example values for 3x3
    m10 = solveByCramer(mA3, mb3);
    printMat("3x3 example A values", mA3);
    printMat("3x3 example b values", mb3);
    printMat("Solving 3x3 example using Cramer's Rule", m10);

    mA4 = mFromStr("2,9,8,9;10,6,7,13;1,10,6,3;8,14,2,19"); //Example values for 4x4
    mb4 = mFromStr("188;226;122;279");                      //Example values for 4x4
    m11 = solveByCramer(mA4, mb4);
    printMat("4x4 example A values", mA4);
    printMat("4x4 example b values", mb4);
    printMat("Solving 4x4 example using Cramer's Rule", m11);

    system("pause");
    return 0;
}
```

Fibonacci Program

Fibonacci sequence is the sequence where the first two terms (not including 0) are 1 and each term after that is the sum of the previous two terms (e.g. third term is 1 + 1 = 2).

To implement the function recursively, the function should have a function call for itself inside the function. Starting values for the first two term should be present, and the next term should be set as the sum of the previous two, and should be passed as the parameters for this call so that the new call can calculate the sum of the previous two terms and call the next one until the term wanted is reached, which should be checked by a conditional statement before the call.

To implement the function iteratively, the function should have a loop where it calculates the new value from the previous two on each iteration. And when the desired term is reached, which is checked by the conditional statement of the loop, iteration stops.

Recursive method: Calculates the n'th term of the Fibonacci sequence. Uses recursive method to calculate the next term. 'limit' is the n'th term the function will finish at. 'ct' is the counter, starts at 3 as the terms before them are already in place. 'n1' is the the second last value where n1 + n2 equals to new value. 'n2' is the the last value where n1 + n2 equals to new value. Returns the n'th term as an integer.

```
int fibonacci_recursive(int limit, int ct = 3, int n1 = 1, int n2 = 1); ///Default values are set for the
counter to start at 3, n1 and n2 to be equal to 1 as thats the value for the first two term.

int fibonacci_recursive(int limit, int ct, int n1, int n2){
    if (limit <= 0) return 0; ///If the term wanted is 0th term or below that, return 0. (0th term of the
Fibonacci sequence is 0.)
    else if (limit <= 2) return 1; ///Otherwise, if the term wanted is the 1st or the 2nd term, return 1,
as it is their value.
    if (ct++ <= limit) return fibonacci_recursive(limit, ct, n2, n1 + n2); ///If the counter is below or
equals to the term wanted, recursively call and return the returned value of this function by passing the
limit value, the increased counter, n2 value for the new n1 value and sum of n1 and n2 as the new n2
value (shifting the values and making the next value the sum of the last two, as in Fibonacci series).
    else return n2; ///If the counter is above the the term wanted, return the last calculated value,
which is stored in n2.
}
```

Iterative method: Calculates the n'th term of the Fibonacci sequence. Uses iterative method to calculate the next term. 'limit' is the n'th term the function will finish at. 'ct' is the counter, starts at 3 as the terms before them are already in place. Returns the n'th term as an integer.

```
int fibonacci_iterative(int limit){
    if (limit <= 0) return 0;       ///If the term wanted is 0th term or below that, return 0. (0th term
of the Fibonacci sequence is 0.)
    else if (limit <= 2) return 1;  ///Otherwise, if the term wanted is the 1st or the 2nd term, return
1, as it is their value.
    int n1 = 1, n2 = 1, ct = 3, temp;   ///'n1' is the the second last value and 'n2' is the the last
value where n1 + n2 equals to new value. 'ct' is the counter, starts at 3 as the terms before them are
already in place. 'temp' is the temporary value to store the sum of the previous two values before
shifting.
    while (ct++ <= limit){  ///While the counter is below or equals to the term wanted.. (The counter is
increased)
        temp = n1 + n2;     ///The sum of the last two values are stored in temp.
        n1 = n2;            ///The value of n2 is shifted to n1.
        n2 = temp;          ///The sum value stored in temp is set as the new value of n2.
    }
    return n2;              ///Returns the last calculated value, which is stored in n2.
}
```

Main function that calculates the sum the first 30 terms in the series and reports how long it takes, for each version.

```
int main(){
    int sum = 0; ///Values for the sum and to calculate the time are set/declared.
    clock_t t1 = clock();    // return time now
    for(int i = 0; i < 30; i++) sum += fibonacci_recursive(i);      ///The actual loop to calculate the
sum of the first 30 terms in the fibonacci series by recursive method. Only one parameter is needed to
specify the wanted term as the other parameters have default values.
    clock_t t2 = clock();
    cout << "Sum: " << sum << "\nThe time it took for the recursive method (in milliseconds): " <<
(((float)(t2-t1))/CLOCKS_PER_SEC) * 1000 << "\n"; ///Reporst the sum and the time it took. Calculations
are made to convert the unit of the result to milliseconds.
    sum = 0; //Resetting the sum.
    ///The next is for the iterative method.
    t1 = clock();    // return time now
    for(int i = 0; i < 30; i++) sum += fibonacci_iterative(i);      ///The actual loop to calculate the
sum of the first 30 terms in the fibonacci series by iterative method.
    t2 = clock();
    cout << "Sum: " << sum << "\nThe time it took for the iterative method (in milliseconds): " <<
(((float)(t2-t1))/CLOCKS_PER_SEC) * 1000 << "\n";
    system("pause");
    return 0;
}
```

Output: The results show that it takes less time to execute the iterative method than it takes for the recursive method.

```
Sum: 1346268
The time it took for the recursive method (in milliseconds): 0.012
Sum: 1346268
The time it took for the iterative method (in milliseconds): 0.005
```

Reflection

Utilising matrix concepts and calculations made me think creatively to figure out how to implement them in coding. Tasks were designed to make us use our problem solving skills to figure out how to add a new feature using the existing ones. Using recursive and iterative methods for the same task and comparing them drew my attention to the performance aspect of my codes.