Module Code: CS1FC16

Assignment report Title: Matrix and Fibonacci algorithms

Student Number: 30002734

Date (when the work completed: 13th Feb 2022

Actual hrs spent for the assignment: 15

Assignment evaluation (3 key points): This coursework was actually enjoyable. The specification was clear and easy to understand. I felt like I knew what I was supposed to do the whole time. This has strengthened my knowledge and understanding of C++ and matrices

Introduction

We were provided a factorial program and a matrix program with the goal of creating our own Fibonacci program and extending the features of the matrix program. The aim is to create 2 Fibonacci programs, one that generates the sequence using recursion and one using an iterative loop.

Matrix Program

A structure called myMat is used to store matrices. There is a 1d array called data that is used to store the data inside the matrix. It also has 2 int variables called numCols and numRows which define how many rows and columns each matrix has. With these attributes, vectors and matrices can be stored.

Function to create a matrix being a column vector from given matrix

This function takes in a matrix m and which column (col) you would like to extract from the matrix. It creates a new result column vector with however many rows the matrix m has. It then loops through the rows of the inputted column and puts it inside the result matrix which is then returned.

myMat mGetCol(myMat m, int col)

{

// create a matrix from m, having one col

myMat res = zeroMat(m.numRows, 1); // create a matrix of 1 col

for (int row = 0; row < m.numRows; row++) // for each element in col

res.data[row] = getElem(m, row, col); // copy row element to res

return res;

}

Function to calculate the dot product of two vectors

This function takes inputs of 2 vectors and returns an integer which is the dot product of the 2 vectors. A for loop goes through each row of the vectors and calculates the product. Then each product is added to the previous one to find the total of the products. The product is then returned.

int dotProd(myMat v1, myMat v2)

{

int product = 0;

for (int i = 0; i < v1.numRows; i++) // loops through the rows

product = product + v1.data[i] \* v2.data[i]; // product of v1 and v1 row found then added to total

return product; // returns total

}

Function to create a matrix which is the transpose of given matrix

This function has an input argument of a matrix m and returns a resultant matrix. It simply flips the row number with the column number of each element in matrix m and puts the new elements into matrix res.

myMat mTranspose(myMat m)

{

// return a matrix which is m transposed

myMat res;

res.numRows = m.numCols;

res.numCols = m.numRows; // rows and columns flip in transpose

for (int r = 0; r < m.numRows; r++) // do each row

for (int c = 0; c < m.numCols; c++) // do each column

setElem(res, c, r, getElem(m, r, c)); // flips each element in m's rows and columns

return res;

}

Function to add two matrices

This function takes two matrices as inputs. Each corresponding element in the two arrays are added together by looping through each element with a for loop and entered into a resultant matrix called res. Res is then returned.

myMat mAdd(myMat m1, myMat m2)

{

myMat res;

res.numRows = m1.numRows;

res.numCols = m1.numCols; // creating result matrix

for (int r = 0; r < m1.numRows; r++) // do each row

for (int c = 0; c < m1.numCols; c++) // do each column

setElem(res, r, c, (getElem(m1, r, c) + getElem(m2, r, c)));

/\*

get each element from m1 and m2

adds them together

then puts it in the corresponding place in result matrix

\*/

return res;

}

Function to multiply two matrices

This function takes two matrices m1 and m2 as inputs and multiplies them together. The output of this function is the resultant matrix. The resultant matrix size is determined by the two inputs. The number of columns in the first matrix must match the number of rows in the second matrix. There are two loops to go through each element in the matrix. To find each element in the matrix, m1 is transposed so that both inputs are the same shape. Then the dot product of each row and column of m1 and m2 is found and inputted into each element of the resultant matrix. This is because matrix multiplication is just a series of lots of dot product calculations.

myMat mMult(myMat m1, myMat m2)

{

myMat res; // result matrix

// the number of columns in the first matrix must match the number of rows in the second matrix

res.numRows = m1.numRows; // the result matrix has to have matrix m1's amount of rows

res.numCols = m2.numCols; // the result matrix has to have matrix m2's amount of columns

res = zeroMat(res.numRows, res.numCols); // empty result matrix

for (int i = 0; i < m1.numRows; i++) // loops through m1's rows

{

for (int j = 0; j < m2.numCols; j++) // loops through m2's columns

{

setElem(res, i, j, dotProd(mTranspose(mGetRow(m1, i)), mGetCol(m2, j)));

/\*

each row of m1 is transposed

then the dot product of this new vector and m2 is found

then set as the element in position (i, j) in the new matrix

\*/

}

}

return res;

}

One to multiply each element in a matrix by a scalar

This function has input arguments of a matrix and a scalar value that is stored as a double. I chose to use a double mainly because when finding the inverse of a matrix, often there is a fraction value you have to multiply your matrix by and I can use this scale function for that. There are 2 for loops that loop through each element in m1 and multiply the value by the scalar. Then put this new value into the corresponding place in the resultant matrix. Matrix res is then returned.

myMat mScal(myMat m1, double s)

{

myMat res;

res.numRows = m1.numRows;

res.numCols = m1.numCols; // creating result matrix

for (int r = 0; r < m1.numRows; r++) // do each row

for (int c = 0; c < m1.numCols; c++) // do each column

setElem(res, r, c, (getElem(m1, r, c) \* s)); // takes each element and multiplies it by scalar value

return res;

}

One to solve a two variable matrix equation using a ‘magic’ matrix

This function takes in a 2x2 matrix and a column vector. mA is the 2x2 matrix and mB is the vector. Ax=B is an equation of these matrices, and this function finds x. The inverse of mA has to be found to calculate mx. The inverse of mA is put into matrix iA and multiplied with mB to find mx. Then mx is returned.

myMat mEquat(myMat mA, myMat mB)

{

myMat mx, iA; // creating result and inverse matrix

double s; // scalar var

mx = zeroMat(1, 2);

mx.numCols = 1;

mx.numRows = 2;

iA = zeroMat(2, 2);

iA.numCols = 2;

iA.numRows = 2; // setting attributes for matrices

float a, b, c, d;

a = getElem(mA, 0, 0);

b = getElem(mA, 0, 1);

c = getElem(mA, 1, 0);

d = getElem(mA, 1, 1); // getting values in mA

setElem(iA, 0, 0, d);

setElem(iA, 0, 1, -b);

setElem(iA, 1, 0, -c);

setElem(iA, 1, 1, a); // creating inverse matrix

s = 1 / (a \* d - b \* c); // finding scalar

mx = mMult(iA, mB); // multiplying inverse A and B

mx = mScal(mx, s); // applying scalar

return mx;

}

One to calculate the determinant of a 2D matrix and solve an equation using Cramer’s Rule.

This function takes 2 matrices as inputs. One is a column vector m2 and one is a 2x2 matrix m1. I use 2 additional matrices in order to calculate the result. This is because you need to find the determinant of the two matrices and divide the determinants of those with the determinant of matrix m1. That is what Dx and Dy are.

myMat mCramer(myMat m1, myMat m2)

{

// Ax=B

// m1 = A

// m2 = B

// res = x

int D, Dx, Dy; // integers for determinants

myMat mDx, mDy, res; // matrices for calculations

res.numCols = 1;

res.numRows = 2;

mDx.numCols = 2;

mDx.numRows = 2;

mDy.numCols = 2;

mDy.numRows = 2; // setting matrix values

setElem(mDx, 0, 0, m2.data[0]); // giving values to matrix x to calculate determinant

setElem(mDx, 0, 1, m1.data[1]);

setElem(mDx, 1, 0, m2.data[1]);

setElem(mDx, 1, 1, m1.data[3]);

setElem(mDy, 0, 0, m1.data[0]); // giving values to matrix y to calculate determinant

setElem(mDy, 0, 1, m2.data[0]);

setElem(mDy, 1, 0, m1.data[2]);

setElem(mDy, 1, 1, m2.data[1]);

Dx = mDet(mDx); // determinant of mDx

Dy = mDet(mDy); // determinant of mDy

D = mDet(m1); // determinant of input matrix

res.data[0] = Dx / D; // dividing both deteminants to get result

res.data[1] = Dy / D; // dividing both deteminants to get result

return res; // returns result

}

Main Function

This is the main function of my program. For the magic matrix and Cramer function I used my student ID generated numbers.

int main()

{

myMat A, C, A2, B2, A3, B3;

A = mFromStr("7,6,7;8,3,6");

C = mFromStr("10,7;10,10;7,8");

printMat("A", A);

printMat("C", C);

printMat("A + C’", mAdd(A, mTranspose(C)));

printMat("A \* C", mMult(A, C));

printMat("C \* A", mMult(C, A));

printMat("5A – 4C’", mAdd(mScal(A, 5), mScal(mTranspose(C), -4)));

// two variable matrix equation using a ‘magic’ matrix

A2 = mFromStr("3,6;3,10");

B2 = mFromStr("84; 124");

printMat("Ax=B\nx", mEquat(A2, B2));

// Cramer function, question 8.

A3 = mFromStr("8,10;10,6");

B3 = mFromStr("140;136");

printMat("Cramers rule\nx", mCramer(A3, B3));

return 0;

}

Matrix Results

Text

Description automatically generated

Fibonacci Program

The Fibonacci series is a series of numbers that is generated by starting with 0 and 1, then adding them together to get a result. Then you use the result and add that number to the previous highest number. You end up having a previous number, current number, and next number. You add the current number and previous number to find the next number. Then you repeatedly use the next and current number to continue the sequence.

Iterative Fibonacci Program

#include <iostream>

using namespace std;

int main()

{

int a, b, c; // define 3 variables to store previous, current and next values for fib sequence

b = 0; // b starts at 0

c = 1; // c starts at 1 to start the sequence

for (int i = 0; i <= 30; i++) //iterative loop from 0 to 30

{

a = b; // previous number = current number

b = c; // current number = next number

c = a + b; // new next number is found by adding the 2 previous highest numbers

cout << "fib " << i << " is " << a << "\n"; // count and fib value is outputted

}

}

Graphical user interface

Description automatically generated

These are the results of the iterative Fibonacci algorithm.

Recursive Fibonacci Program

#include <iostream>

using namespace std;

int fib(int n) // fib function

{

if (n > 1) // fib of 1 returns 1, otherwise the fib algorithm is executed

return fib(n - 1) + fib(n - 2); // the previous number plus the one before it

else

return 1; // return 1 when n is less than 1

}

int main()

{

for (int i = 0; i <= 29; i++) // for loop to do fib function on numbers 0 to 30

cout << "fib " << i + 1 << " is " << fib(i) << "\n"; // console output of fib value

}

Graphical user interface, text, application

Description automatically generated

These are the results of the recursive Fibonacci algorithm. As you can see both algorithms produce the same results.

The iterative Fibonacci algorithm is faster than the recursive algorithm because there is only one for loop. The iterative algorithm has time complexity of O(N) and the recursive algorithm has a time complexity of O(2^n).

Reflection

Overall, I enjoyed this coursework. After finishing each function there was great satisfaction. Both Fibonacci programs work well, and I think the code is concise. I completed all the matrix functions and they all worked as expected. I changed the main function of the matrix program to output everything I needed to show. I didn’t show the inputs to the 2x2 magic matrix and Cramer’s rule to the console because you already have access to the numbers.