LU Decomposition Program

And C++ Algorithm Analysis

Philip Solimine

MAS3105-01, 4/17/15

(1) Introduction and Mathematical Background

This program finds and displays the LU decomposition of an nxn matrix. Initially, the user is asked to input n and their matrix of choice called **A**, and a vector **b**. Once the decomposition of the matrix has been found using the LU() function, the resulting triangular matrixes **L** and **U** are input into the FB() which will attempt to use forward and backward substitution to solve the equation **Ax**=**b** and return the vector **x**.

There are two algorithms central to this program. In the code, they can be found within the LU() and FB() function definitions. The first algorithm solves for the triangular matrixes **L** and **U**. This function contains a C++ translation of the LU decomposition MatLab code provided on BlackBoard. It is worth noting, before explaining this algorithm, that C++ does not support the : operator that can be used in MatLab to supplement nested loops. The other primary difference is that while in MatLab, arrays, vectors and matrixes are indexed 1:n, the indices in C++ represent elements of arrays as 0:n-1. The LU() function works as follows:

The function is called using LU(A,I), where **A** is the user’s input matrix, and **I** is an identity matrix of size n.

(a)

(b)

The goal of this algorithm is to solve equation (a). This is accomplished, initially, by using equation (b). Using these equations together lets the program begin by combining them, **AI**=**LU**. To find **L** and **U**, we begin by setting them equal to **I** and **A**, respectively. Because of the nature of multiplication, there are many matrixes **L** and **U** that could solve equation (a). Beginning with the identity matrix allows us to ensure that the **L** obtained by the algorithm will be a lower triangular matrix with 1’s in its diagonal positions. Because the 1’s will remain as the diagonal of matrix **L**, a lower left triangular matrix, the algorithm must only solve for all values underneath the diagonal of **L**. The algorithm will solve for **L** as follows:

**,**

Which is simplified to the final algorithm for **L**:

Now it is time for the algorithm to solve for **U:**

**,**

Simplifying to:

,

Once the program has successfully decomposed the matrix **A** into lower and upper triangular matrices **L** and **U**, it is time to use forward and backward substitution to solve equation (c).

(c)

(d)

(e)

Forward solving equation (d) is performed first by the algorithm. Using equation (d):

**, ,**

The equation for **y** is now solved in ascending order for k = 1:n. Once the algorithm finishes solving for **y**, it is now time to use equation (e) and backwards substitution to solve for **x:**

**, ,**

This resulting algorithm is solved backwards from k = n:1, finding the vector **x** which solves equation (c). This vector solution is returned by value to the main() routine and printed with the rest of the output. At this point, the program will have used the above algorithms to solve for **L**, **U**, and **x**. Currently there is an issue with the algorithms which stems from the lack of native C++ support for complex numbers. However, for matrixes that have real solutions toequation (c), the algorithm should hold. A possible solution to this problem is listed alongside the code below, in bolded typeface and surrounded by /\* \*/ comment indicators.

(2) Code Listing

The code below is written in C++, designed in Microsoft VisualStudio 2012:

#include <iostream>

#include <vector>

#include <complex>

using namespace std;

typedef complex<double> com; //define complex, now any com can be entered as (a,b) as in

//a+bi. Digit entries are automatically stored as (a,0)

//and can be entered without parenthesis

typedef vector<com> vec;

//define vec type as a vector filled with complex variables

typedef vector<vec> mat; //define mat type as a matrix aka a vector filled with vectors

void LU(mat&, mat&); //LU Decomp & forward-backward substitution function prototypes

vec FB(mat&, mat&, vec);

int main() {

int n;

cout << "LU DECOMPOSITION OF nxn MATRIX\n";

cout << "n = ";

cin >> n;

vec row(n); //new double vector with n entries

mat A, I; //matrixes to hold A & I

mat L, U; //these will hold LU after running the LU() function

vec b(n); //new vector for b with n entries

vec x(n); //this vector will hold the solution to Ax=b after FB()

for (int i = 0; i<n; i++) {

for (int i2=0; i2<n; i2++) //initialize nxn identity matrix

if (i2==i)

row[i2]=1;

else

row[i2]=0;

I.push\_back(row);

}

cout << "Enter A by row, with each digit followed by the enter key: \n";

for (int i = 0; i<(n); i++) {

for (int i2 = 0; i2<n; i2++)//user enters A

cin >> row[i2]; //row is a temp variable that holds rows before //they're entered in A

A.push\_back(row); //append A with this row vector

}

cout << "Enter b with each digit followed by the enter key: \n";

for (int i = 0; i<n; i++) {

cin >> b[i]; //user enters b

}

cout << "\n\nA =\n"; //print A & b to the screen

for (int i = 0; i<(n); i++) {

cout << "\t[\t";

for (int i2 = 0; i2<n; i2++)

cout << A[i][i2] << "\t";

cout << "\t]" << endl;

}

cout << "\nb =\n";

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

cout << "\t[\t" << b[i] << "\t]" << "\n";

L = I; //copy I into L, and A into U

U = A; //function can change these without changing I or L in main()

LU(U,L); //decompose A into L and U

cout << endl << endl;

cout << "RESULT:\nL = " << endl; //print results of LU decomposition

for (int i = 0; i<n; i++) {

cout << "\t[\t";

for (int i2=0;i2<n;i2++)

cout << L[i][i2] << "\t";

cout << "\t]\n";

}

cout << endl << endl;

cout << "U = " << endl;

for (int i = 0; i<n; i++) {

cout << "\t[\t";

for (int i2=0;i2<n;i2++)

cout << U[i][i2] << "\t";

cout << "\t]\n";

}

x = FB(L,U,b); //perform forward & backward substitution to find x: Ax=b

cout << "\nSOLUTION (Ax=b):\nx =\n";//print x

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

cout << "\t[\t" << x[i] << "\t]\n";

cout << endl;

return 0;

}

void LU(mat &U, mat &L) {

int n = U.size(); //set n based on # of columns in A

//array indexes in C++ are 0:n-1 rather than 1:n

//loop bounds were changed to reflect this

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

for (int k2 = k+1; k2<n; k2++) {

L[k2][k] = U[k2][k]/U[k][k];//translated from prof. online matlab //code for LU decomposition:

} //C++ has no : operator, so an addt'l loop is req'd for i2

for (int j = k+1; j<n; j++) {

for (int j2=k; j2<n; j2++) {

U[j][j2] = U[j][j2] - L[j][k]\*U[k][j2];

}

} //this function uses pass by address instead of returning

} //a new value, allowing main() to get both L and U from LU()

}

vec FB(mat &L, mat &U, vec b) {

int n = b.size();

vec x(n), y(n); //LUx=b, Ux=y, Ly=b

//begin solving for y:Ly=b

for (int i=0;i<n;i++) {

y[i] = b[i]; //using derived algorithm yk=(bk-sum[i=1,k-1](lki\*yi))/lkk

//runs "forward" from y1 to yn

for (int ii=0;ii<i;ii++) {

y[i] = y[i] - (L[i][ii] \* y[ii]); //y[i]=b[i]-sum(L[i][0:i-1]\*y[0:i-1])

}

y[i] = y[i] / L[i][i]; //y[i]=(b[i]-sum(L[i][0:i-1]\*y[0:i-1]))/L[i][i]

}

//begin solving for x:Ux=y

for (int i=n-1;i>=0;i--) {

x[i]=y[i]; //derived algorithm xk=(yk-sum[i=k+1,n](Uki\*xi))/Ukk

//runs "backwards" from xn to x1

for (int ii=i+1; ii<n; ii++) {

x[i] = x[i] - U[i][ii] \* x[ii]; //x[i]=(y[i]-sum(U[i][i+1:n- //1]\*x[i+1:n-1]))

}

x[i] = x[i] / U[i][i];//x[i]=(y[i]-sum(U[i][i+1:n-1]\*x[i+1:n-1]))/U[i][i]

}

return x; //return solution as vector

}

(3) Results

Some recorded input and output, results given in the form (a,b) are equal to a+bi:

From now on, real results will be shown with only the a value of their (a,b) format.

The algorithm holds for any integer value of n. An example of successful decomposition of a 4x4 matrix:

The algorithm also holds for complex values in **A** and **b**. All entries in all matrixes in this program are set to complex variables, so that real number 2 can be entered as 2, and complex numbers such as 1+2i can be entered as (1,2):

(4) Conclusion

Below are some examples of when the LU() or FB() algorithms fail:

(No solution to **Ax**=**b** can be found using this **L** and **U**, the function prints QNAN error in place of values)

In the previous case, the algorithm failed because **U** contains a row of all zeroes, which in this case, when solving for **x** would imply that 0+0+0=b3=12, which is, of course, impossible. If a failure such as this occurs, #QNAN (Not a Number) errors are returned as values of **x**. This means that the algorithm has failed, most likely as a result of an attempt to divide by 0. The program could be cleaned up if it searched for #QNAN errors and returned a “FAILED” message for **x**. Similar 3x3 matrixes, however, can be solved by the algorithm without issue.

Another #QNAN error occurs, for certain matrices, in **U** (as shown above). This indicates that there is no **LU** decomposition that can be found using the version of **L** with 1’s as its diagonal values. To solve this issue, the algorithm would need to revert to the LU() function and run it with a different starting value of **L**, which in the case of this algorithm is initiated as the nxn identity matrix.

Previously, there was an issue with matrixes containing complex values. C++ has no native support for complex numbers. Fortunately, a convenient solution to this problem was found using the <complex> library. The code change was as follows:

#include <iostream>

#include <vector>

using namespace std;

typedef vector<double> vec;

typedef vector<vec> mat;

Changed to:

#include <iostream>

#include <vector>

#include <complex>

using namespace std;

typedef complex<double> com;

typedef vector<com> vec;

typedef vector<vec> mat;

Rather than defining a matrix as a vector filled with vectors which are filled with doubles, a vec is defined as a vector filled with complex numbers, and a matrix is a vector filled with those vecs. Because entry of the matrix utilizes dynamic memory allocation, there is no need to change any other variables from double to complex, making the fix as simple as altering the header as shown above.