Second-Order differential Equation

Jiahong Xue

Abstract

This project will discuss solving the second derivative with linear algebra and matrices application in computational language. Specifically, Gaussian elimination with Euler`s formula and LU decomposition method. Two methods will be compared with run time and the numerical results of those 2 methods will be compared to its analytic solutio

Introduction

In this Project, I am going to use C++ to perform a Gaussian Elimination and LU decomposition to solve the second order differential equations

$$-y'' = f(x)$$

The idea of solving this differential equation is to turn this equation into a matrix A times a vector to obtain my values of f(x). The matrix can be set up with the definition of the derivative back in Calc 1.

$$f'(x) = -lim_{h \to 0}(f(x+h) - f(x))/h$$

This is also called Euler`s 2 points expression, where h is called step length, so the derivative is also the slope at point x of f. To get the second derivative, we need 3 points:

$$f''(x) = -lim_{h \to 0}\{[f(x+h) - f(x)] - [f(x) - f(x-h)]\}/h * h$$

$$f''(x) = -lim_{h \to 0}\{[f(x+h)) + f(x-h) - 2f(x)]\}/h * h$$

Now we have the theory of our algorithm. We need to do now is transform this into a matrx A times a vector v, to get our new vector f. To do that, let's observe our equation, it has some unchanged constant such as 1, 1 and -2. That`s what we can do to set up our matrix. Considering the (x+h) as an index of function f. then the equation above becomes

$$f''(x) = [-f_{x-1} + 2f_x - f_{x+1}]/h * h$$

Here, we can drop the limit sign because h is the value we are going to initialize, set it to be small values can works as a limit goes to 0. The other factor we are going to initialize is function f"x, where is the initial function that we need to solve the second-order differential equation for. So let`s put them together on the same side of the equation. Then we can focus on the what is left on the other side of the equation. Since we make it into an indexing form, and the factor -1, 2, -1 does not change. And these factors is doing the operation in an order. As the index going forward, the -1, 2, -1 goes forward, too! So the function can be set up as:

$$\begin{bmatrix} 2 & -1 & 0 & & 0 \\ -1 & 2 & -1 & \cdots & 0 \\ 0 & -1 & 2 & & 0 \\ & \vdots & & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & -1 \quad 2 \end{bmatrix}$$

Let`s call this matrix A as before, the v automatically becomes the result for the function we need to find at x values. So the equation becomes to be A v = f, where A is already set up, f is the result of the

second-differential function, and v is the vector we need to solve. (give up on the first and last value in v, look at the matrix above, the first row and last row is incomplete, so the result won`t be true.) To solve it, we can use either Gaussian Elimination or LU decomposition.

Method

Gaussian Elimination:

Let`s say I have a matrix A with element a[r][c], where r is the index in rows and c is the index in columns. And v[n], f[x] as the elements in vector v and f respectively. Then I have:

$$f[x] = \sum a[r][c(k)] * r[n(k)], \quad (1)$$

with r=x, c(k) and n(k) is the same and iterate from k=1 to k=N, where N is the total number of elements in columns of matrix A and number of rows in vector v. The above equation is just simply matrix multiplication, which is for the value in x row of f, is equal to the sum of the elements of same row number in matrix A, multiplied by the elements in vector v, where the multiplication will only be performed with elements of the same column number (k) in x row of matrix A and elements with the same row number (k) in the vector v and take the sum of it is our f.

Based on this, we can do operation like addition and multiplications for each rows of our (1), what Gaussian elimination will do here is to multiply the first whole row by first element in the second row and divide its own first element,

$$R1 * a[2][1]/a[1][1]$$

Where R1 represents Row first. This makes our first element in first row is equal to the first element in second row. And then, we subtract the second row from first, so the first element in second row is canceled out. And the place our old Row back into first row.

Do the same thing for the rest of the rows, with Rn-R1*a[n][1]/a[1][1]. After this, we will get a new matrix with the first column are 0s but the first one is still a[1][1]. Let`s move on and take away the first row and for column in the matrix A. so the matrix A is now size (n-1)*(n-1). Do the same thing to make the first column all 0 except for our new a[1][1]. And we put our taken away row and columns back, we will see the matrix is rewritten as:

$$\begin{bmatrix} 2 & -1 & 0 & & 0 \\ 0 & 2 & -1 & \cdots & 0 \\ 0 & -1 & 2 & & 0 \\ & \vdots & & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & -1 \quad 2 \end{bmatrix}$$

As we can see the high-lighted element are now becomes 0. Now let`s go ahead and keep the same procedures for the rest of the matrix till we reduce the matrix to 2*2 matrix and finish it. And now let`s put everything back. Then the new matrix will have bottom left all 0s.

$$\begin{bmatrix} 2 & -1 & 0 & & 0 \\ 0 & 2 & -1 & \cdots & 0 \\ 0 & 0 & 2 & & 0 \\ & \vdots & & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 0 \quad 2 \end{bmatrix}$$

All of what have done above is called Forward Substitution, which is the first half of our Gaussian Elimination, what it does it to "organize" our matrix to become the form like above with all 0s on the bottom left, without changing our any elements in vector v, the vector we need to solve for. Now let`s start the second half of Gaussian Elimination, which is called Backward Substitution.

Start from bottom row of the matrix, and get our equation (1) back. Since everything else here is 0 but the last element. Plug this into our equation (1), we get

$$0 + \cdots + 0 + a[n][n] * v[n] = f[n]$$

Where a[n][n] here in our case is 2, and f[n] was the from the function given we need to solve. The our last element in vector v is

$$v[n] = f[n]/a[n][n]$$

Let`s look at the Row [n-1] now, it has one 0 less than the last row but it`s ok, we have our v[n], we plug everything in equation(1) again, we get

$$0 + 0 + \cdots + 0 + a[n-1][n-1] * v[n-1] + a[n][n] * v[n][n] = f[n-1]$$

And similar to what we did above, we can get v[n-1] to be

$$v[n-1] = \{f[n-1] - a[n][n] * v[n]\}/\, a[n-1][n-1]$$

So as we see here, when we keep going up through the rows, we will be able to get all the elements in v. which we can make a plot and a fit to find out our function for v to solve the second-order differential equation.

LU-decomposition

What LU-decomposition does is to break the matrix A into a product of 2 special matrix L and U, where L has all 0 on the upper right and 1 on diagonals and U has 0 on the lower left half. As shown below

$$\text{L} = \begin{bmatrix} 1 & 0 & 0 \\ l_{21} & 1 & 0 \\ l_{31} & l_{32} & 1 \end{bmatrix} \qquad \text{U} = \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}$$

And then I have $\mathrm{Ax=LUx=f(x)}$, where $\mathrm{f(x)}$ is as above, the result of second-order-differential equation.

And $\mathrm{LUx=f}$ can be written as 2 steps: $\mathrm{LY=f}$, $\mathrm{Ux=Y}$, which is now easier to find our x here. To do this, first we need to find our L and U matrix.

Since A=L*U and we have L and U set up as above format, that makes $\boldsymbol{u_{11}} = a_{11}$, and then, we can get l_{21} by $a_{21} = \boldsymbol{l_{21}} * u_{11}$, and keep going as a example of A is a 3*3 matrix:

$$a_{11} = \boldsymbol{u_{11}}$$

$$a_{12} = \boldsymbol{u_{12}}$$

$$a_{13} = \boldsymbol{u_{13}}$$

$$a_{21} = \boldsymbol{l_{21}} * u_{11}$$
$$a_{22} = l_{21} * u_{12} + 1 * \boldsymbol{u_{22}}$$
$$a_{23} = l_{21} * u_{13} + 1 * \boldsymbol{u_{23}}$$

$$a_{31} = \boldsymbol{l_{31}} * u_{11}$$
$$a_{32} = l_{31} * u_{12} + \boldsymbol{l_{32}} * u_{22}$$
$$a_{33} = l_{31} * u_{13} + l_{32} * u_{23} + 1 * \boldsymbol{u_{33}}$$

I put the operation of the matrix multiplication here in the order of iterating in rows in A. So I started with first row and iterating through and then start with the second row. So as we can see here, I get all u from first row and one l and 2 u in the second row and 2 l and 1 u in the third row. So the pattern here is if we consider to get the l and u here with the diagonal of the matrix A, when I pass or at my diagonals, I can get u out from multiplication, and before the diagonal, I get l, which I is exactly matches the elements that I need in matrix L and U. And also, that can make our algorithm clear to make. Then, we can conclude the equation for getting general L and U non zero-elements, that is:

$$u_{\boldsymbol{ij}} = a_{ij} - \sum_{k=1}^{i-1} l_{ik} * u_{kj}$$

$$l_{\boldsymbol{ij}} = \frac{1}{u_{(i-1)j}} * \left(a_{ij} - \sum_{k=1}^{j-1} l_{ik} * u_{kj} \right)$$

Where I is row number and j is column number.

So now we have Matrix L and Matrix U, we need to solve for x. As I mentioned before, LUx=f can be written as 2 steps: LY=f, Ux=Y. To do this, let me use an example for 3*3 matrix again:

$$\begin{bmatrix} 1 & 0 & 0 \\ l_{21} & 1 & 0 \\ l_{31} & l_{32} & 1 \end{bmatrix} * \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \\ f_3 \end{bmatrix}$$

Since f is just the result of the second order differential function, so f is all given. And L was solved in the last step, so y are very easy to solve here. And similar for the x, which is what we are solving for.

Results

The results from the program are shown below:

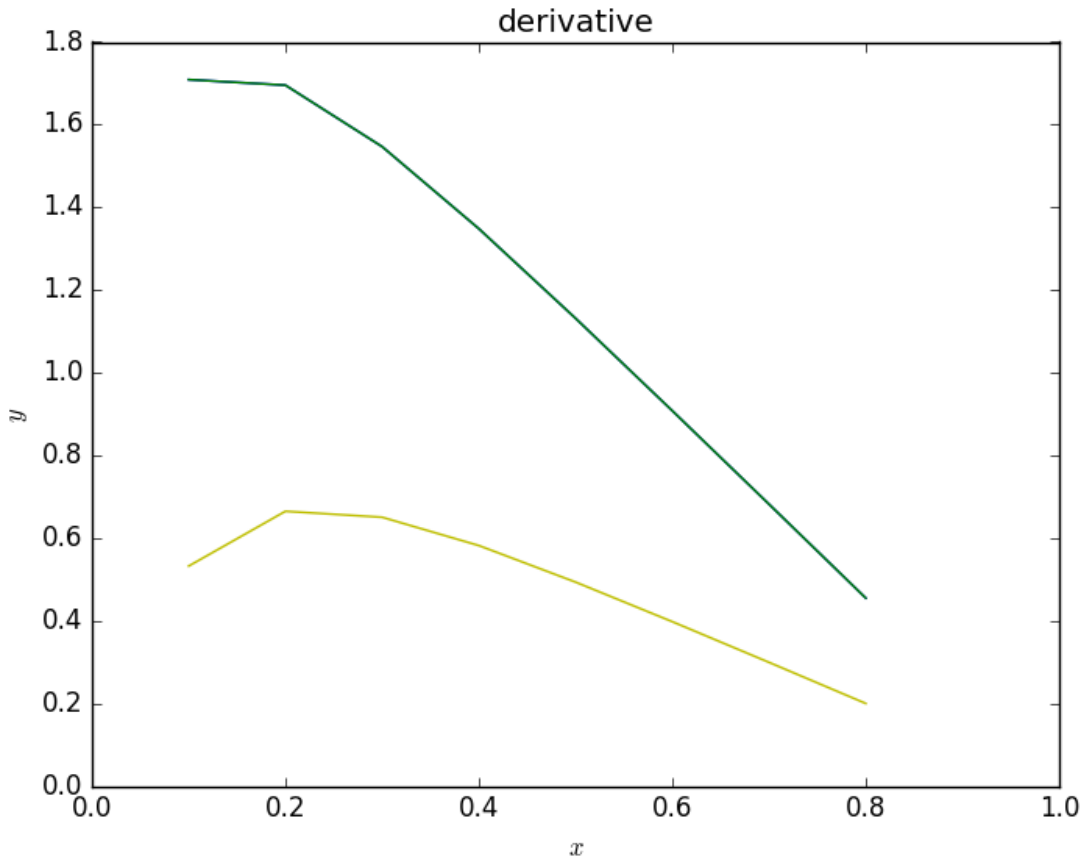


Figure 1. 10 points approximation plots.

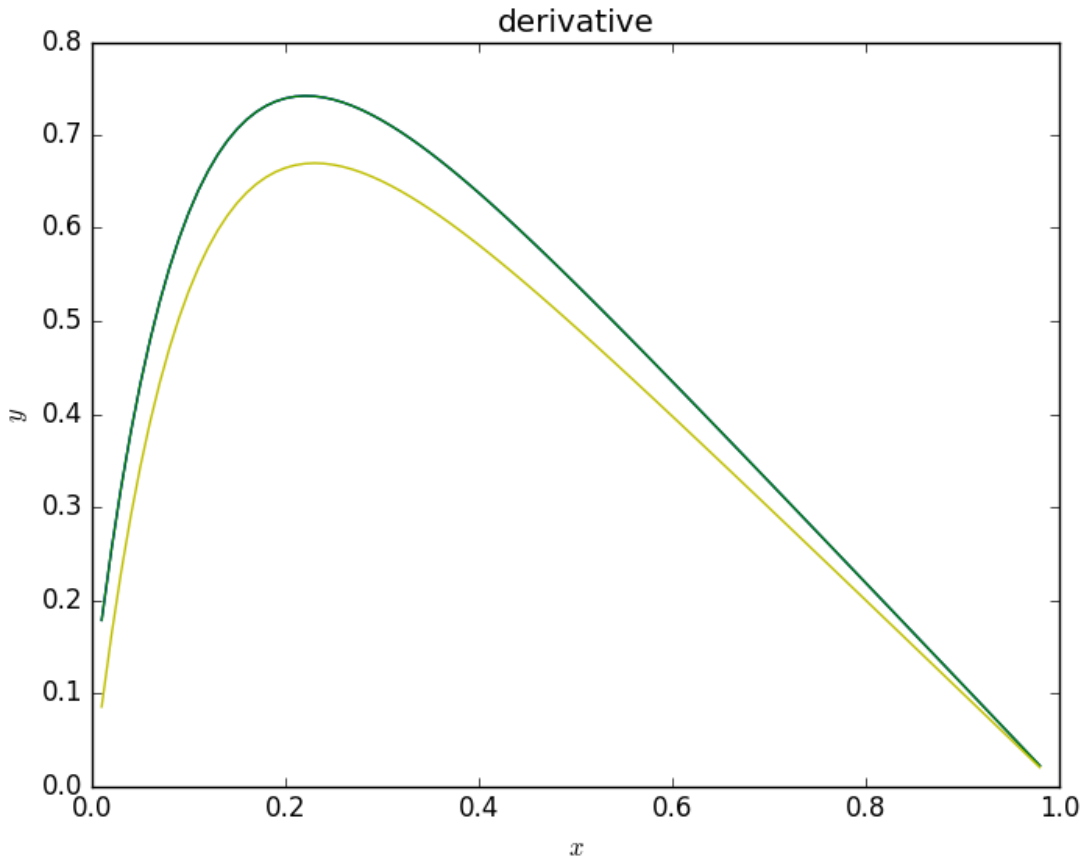


Figure 2. 100 points approximation plots.

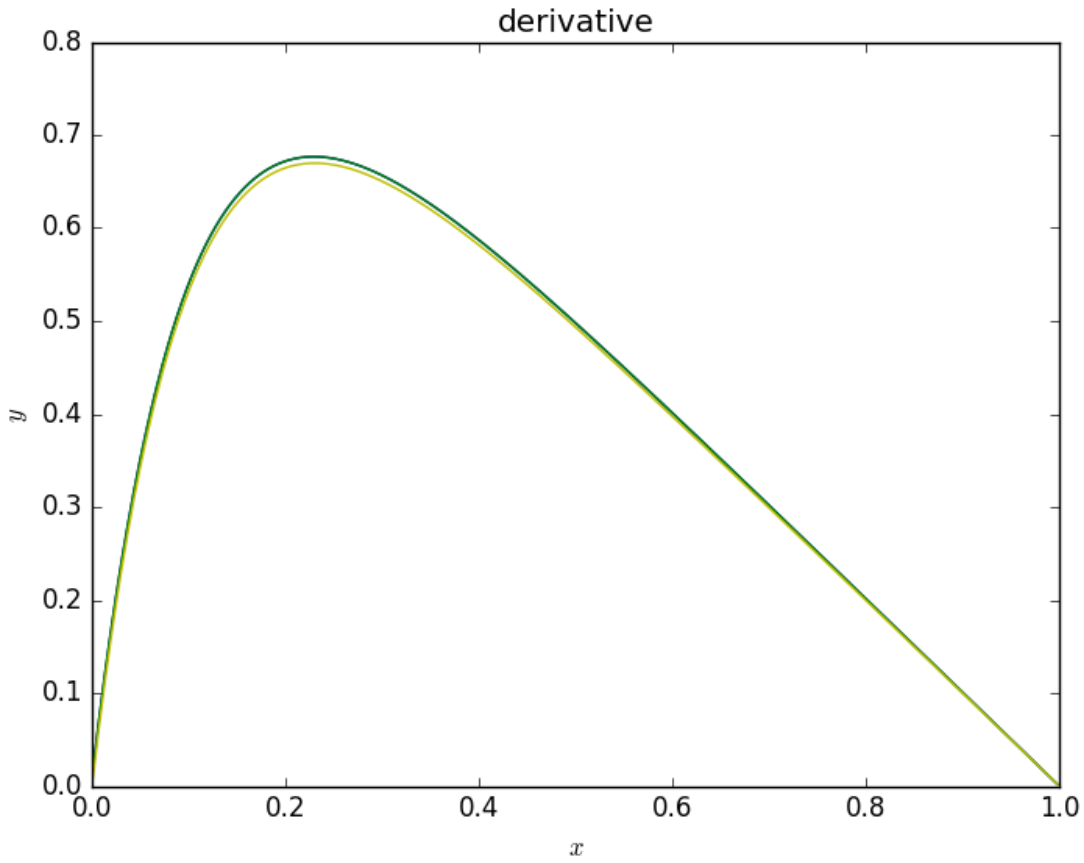


Figure 3. 1000 points approximation plots.

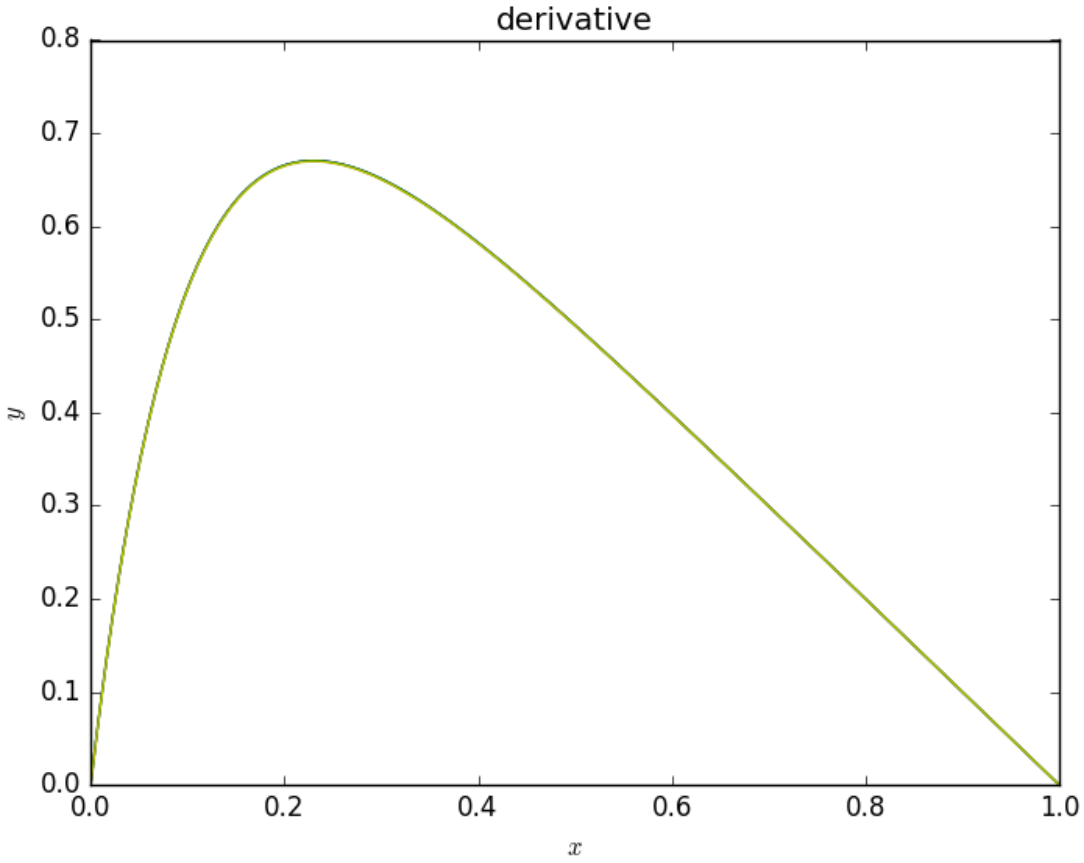


Figure 4. 10000 points approximation plots.

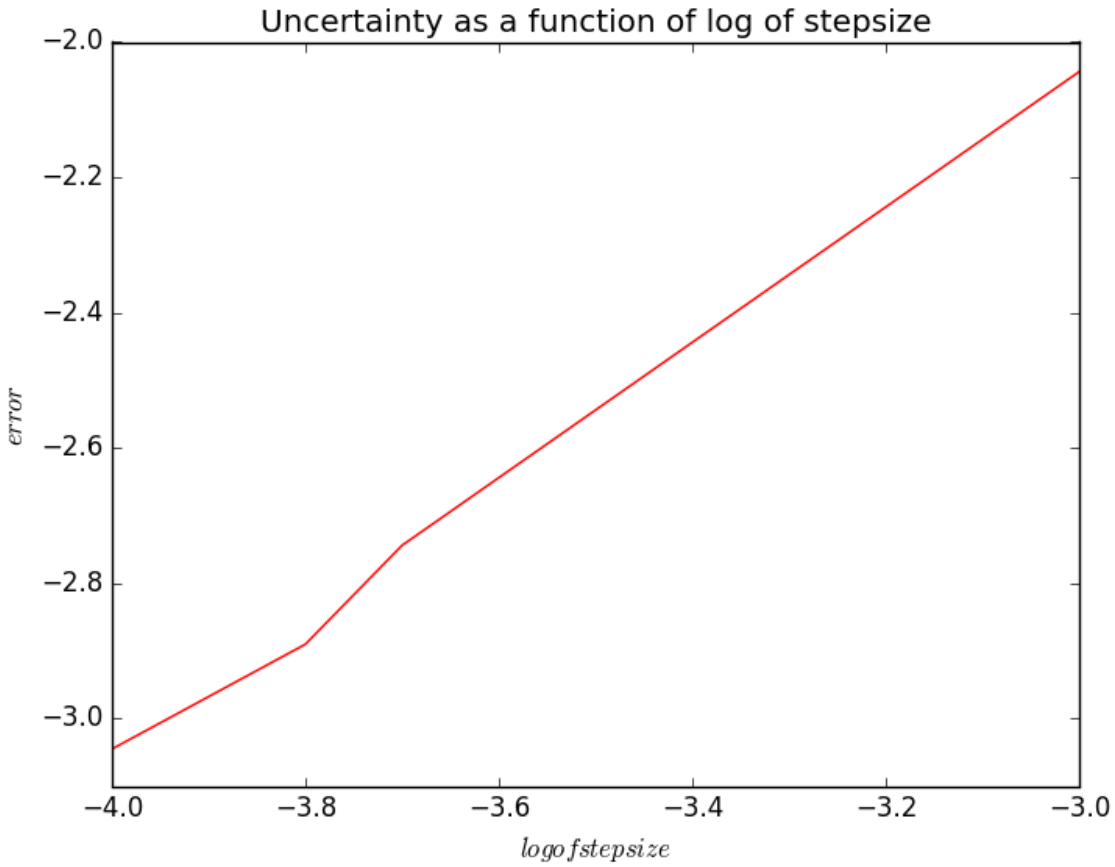
The result from the program of the Gaussian Elimination and LU decomposition was good. I made a log plot of the error which was defined as:

$$Error = \log_{10} \left| \frac{result - actual\ value}{actual\ value} \right|$$

where result is from the LU-decomposition and actual value was calculated from

$$f(x) = 1 - (1 - e^{-10}) * x - \mathrm{e}^{-10\mathrm{x}}$$

Which is the actual resulting function from second-order derivative from the function that was solving for. And the plot of the error function is given below:



From the curve, we can see that the error is quite small. Since the function is in log 10 base, the smaller the value is, the smaller the error are. Most of the value is getting close to -1. So the uncertainty is around 10%, which is quite good. The algorithm works well.

Data points taken	step length h in log	Error in logarithm
1000	-3.0	-2.04
5000	-3.7	-2.74
7000	-3.8	-2.89
10000	-4	-3.05

Table 1. Values of error in logarithm with its step length.

Data points taken	Running time (Gaussian)	Running time (LU)	
1000	0.04	3.5	
5000	0.44	731.9	
7000	0.81	1424	
10000	1.59	5407	

Table 2. Values of running time of each methods with data points

As we can see from the table, with larger data points which is the smaller step size, the error becomes smaller. But also, the running time goes up significantly, especially for LU decomposition method.

Conclusion

There are 2 main algorithm carried out in this project: LU-decomposition and Gaussian Elimination.

The result shows that the method of LU decomposition is a fairly more efficient way for solving second-order differential equation. The way LU decomposition doing was quite a mind-opening algorithm for me. What LU decomposition does is to break the original matrix into 2 easier and doable matrices in a form that can be calculated with only 1 unknown in each row. That apparently gives a good way for the machine (CPU) to solve the problem instead of solving for a couple unknowns from some equations that what original matrix will do. This is also a good way to study matrix and linear algebra, in which we can have a bigger picture of how matrix is operated in a multiplication function. For Gaussian elimination, what it does is to reduce the leading element of the matrix by each rows so that the last row will only have one leading element at the end of the row, which looks like the matrix "U" in LU decomposition, and the matrix becomes solvable. For both methods, they actually have the same idea, that is to reduce the matrix to an easier and doable for the CPU to work with. Always have just one unknown for each operation, and repeatedly doing that, it will give us the result that we are looking for.

Theoretically, we needs a large number of the data points, to get smaller step size to make it more accurate. And the program did show that it is more accurate with smaller step length from Figure 1 through 4.

Partial Codes Attachment

For easier reading, I put f"x as the second order differential equation, and A is the transformed matrix, f is the matrix we are solving for.

Method 1. Gaussian Elimination

Gaussian Elimination Forward Substitution

```
for (int j=1;j<n;j++){
        for (int i=0;i<n;i++){
            if(i==j){
                A[j*n+i]-=(A[(j-1)*n+i]*(A[(j-1)*n+i]))/(A[(j-1)*n+(i-1)]);
                f"x[j]-=(-f"x[j-1])/(A[(j-1)*n+(i-1)]);
            }
            else if(i==j-1)
                A[j*n+i]=0;
            else
                continue;
        }
  }
```

Gaussian Elimination Backward Substitution

```
f=new double[n];
  cout<<endl;
  f[n-1]=f"x[n-1]/A[n*n-1];
  for (int i=n-2;i>=0;i--){
      f[i]=(f"x[i]+f[i+1])/A[i*n+i];
  }
```

Method 2. LU decomposition

Setting up

```
for (int j=0;j<n;j++){
      for (int i=0;i<n;i++){
          //diagonal elements = 1
          if(i==j){
              L[j*n+i]=1;
          }
          //first column in l
          if(i==0){
              L[j*n]=row3[j*n]/U[0]*1.0;
          }
          if(j==0){
              U[i]=row3[i];
          }
          if(j>i)
              U[j*n+i]=0;
          if(i>j)
              L[j*n+i]=0;
      }
  }
  for (int j=1;j<n;j++){
      for (int i=1;i<n;i++){
          if(i>=j){
              for (int k=0;k<j;k++){
                  U[j*n+i]=row3[j*n+i]-L[j*n+k]*U[k*n+i];
              }
          }
          if(i<j){
              for (int k=0;k<i;k++){
                  if(U[(j-1)*n+i]==0)
                      U[(j-1)*n+i]=0.00000000000000001;
                  L[j*n+i]=1.0/U[(j-1)*n+i]*(row3[j*n+i]-
L[j*n+k]*U[k*n+i]);
              }
          }
      }
  }
```

Backward substitution

X is the vector that store the values in f. so:

$$L * (U * f) = L * Y = f"$$

$$Y = U * f = U * X$$

```
Y[0]=f"[0]/L[0];
  for (int i=1;i<=n-1;i++){
```

```
    Y[i]=(f”[i]-L[i*n+i-1]*Y[i-1])/L[i*n+i];
}
X[n-1]=Y[n-1]/U[n*n-1];
for (int i=n-2;i>=0;i--){
    X[i]=(Y[i]-U[i*n+i+1]*X[i+1])/U[i*n+i];
}
```