Project 1

Computational Physics I FYS3150/FYS4150

Hyejin Yun, Yisha Chen

Sep 28, 2017

Abstract

We solved the one-dimensional Poisson equation with Dirichlet boundary conditions by rewriting it as a set of linear equations. LU decomposition as a general algorithm and optimized algorithm for specific tri-diagonal matrix were applied to obtain solutions. We find that optimized algorithm is much more time efficient compared to general algorithm. The max relative errors between closed-form solution and numerical solution were calculated with different grid points *n* (corresponding to different step length *h*). The result shows that the maximum relative error decreases with the increase of *n* first and then increases. When n=10^5, we get the minimum value for maximum relative error.

1. Introduction

The aim of this project is to be skilled using dynamic memory handling of matrices and vectors when programing. Thus, dynamic memory allocation and Armadillo library were used for array handling here. Both general algorithm with LU decomposition and optimized algorithm were implemented to solve a one-dimensional Poisson equation with Dirichlet boundary conditions. In addition, CPU time cost and FLOPS (Floating-point operations per second) were compared to find out the more efficient method. The trend of maximum relative errors between closed-form solution and numerical solution was also discussed with the increasing of *n*. Detail methods and algorithms are described in the following section.

2. Methods

We first rewrote continuous functions into a set of linear equations. Then LU decomposition method and Tri-diagonal Matrix Algorithm were used to get a numerical solution.

2.1. Rewriting Continuous Functions into Linear Equations

The one-dimensional Poisson equation with Dirichlet boundary conditions can be rewritten as a set of linear equations. Therefore, linear second-order differential equation

$$-u''(x) = f(x) \tag{1}$$

where $x \in (0,1)$, $u(0) = u(1) = 0$ can be rewritten as

$$\mathbf{A}\mathbf{v} = \tilde{\mathbf{b}}, \tag{2}$$

$\mathbf{A}$ is a $n \times n$ matrix, and $\mathbf{v}$, $\tilde{\mathbf{b}}$ are $n \times 1$ vectors.

In our case, $f(x) = 100e^{-10x}$, and the closed solution is $u(x) = 1 - (1 - e^{-10})x - e^{-10x}$. The following shows details how to rewrite continuous functions into an equation of matrix.

Let *h* the step length,

$$h = \frac{1}{n} \tag{3}$$

The second derivative of *u* is approximated with

$$-\frac{v_{i+1}+v_{i-1}-2v_i}{h^2} = f_i \qquad \text{for } i = 1, \cdots, n \tag{4}$$

where v_i and f_i are the discrete expressions of $u(x_0 + ih)$ and $f(x_0 + ih)$ for $i = 1, \cdots, n$.

$$-(v_{i+1} + v_{i-1} - 2v_i) = f_i * h^2 \ , \ b_i = f_i * h^2$$
$$-(v_{i+1} + v_{i-1} - 2v_i) = b_i$$

for $i = 1, \cdots, n$ we can list down the equations as

$$\begin{aligned} -(\mathrm{v}_2 + \mathrm{v}_0 - 2\mathrm{v}_1) &= \mathrm{b}_1 \\ -(\mathrm{v}_3 + \mathrm{v}_1 - 2\mathrm{v}_2) &= \mathrm{b}_2 \\ -(\mathrm{v}_4 + \mathrm{v}_2 - 2\mathrm{v}_3) &= \mathrm{b}_3 \\ &\vdots \\ -(\mathrm{v}_{n+1} + \mathrm{v}_{n-1} - 2\mathrm{v}_n) &= \mathrm{b}_n \end{aligned}$$

Let's rearrange the elements

$$\begin{aligned} -(\mathrm{v}_0 - 2\mathrm{v}_1 + \mathrm{v}_2) &= \mathrm{b}_1 \\ -(\mathrm{v}_1 - 2\mathrm{v}_2 + \mathrm{v}_3) &= \mathrm{b}_2 \\ -(\mathrm{v}_2 - 2\mathrm{v}_3 + \mathrm{v}_4) &= \mathrm{b}_3 \\ &\vdots \\ -(\mathrm{v}_{n-1} - 2\mathrm{v}_n + \mathrm{v}_{n+1}) &= \mathrm{b}_n \end{aligned}$$

The variables of unknown function v can be gathered as a set of vectors, and their coefficient can be written as a matrix to be multiplied to v. Since all of the coefficients of each equation are -1, -2, and -1, the element of the matrix will only contain -1 and -2 along the diagonal. Therefore, the matrix would be,

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

When we have a closed solution, we can easily check the equation. For example,

$$u(x) = \ 1 - (1 - e^{-10})x - e^{-10x}$$

The first derivative of $u(x)$ is

$$u'(x) = \ (e^{-10} - 1) + 10e^{-10x}$$

The second derivative of $u(x)$ is

$$u''(x) = \ -100e^{-10x}$$

Therefore,

$$-u''(x) = 100e^{-10x} = f(x)$$

2.2.1 LU Decomposition

LU decomposition is conducted in order to relief the difficulty of calculations. This method has four basic steps:

(1) rewrite $\mathbf{Ax} = \mathbf{b}$ to $\mathbf{LUx} = \mathbf{b}$.
(2) define a new vector **y** by $\mathbf{Ux} = \mathbf{y}$, and write $\mathbf{LUx} = \mathbf{b}$ as $\mathbf{Ly} = \mathbf{b}$.
(3) find **y** with the general solution.
(4) insert **y** to $\mathbf{Ux} = \mathbf{y}$, and find **x** with the general solution.

LU decomposition increase the number of operations, but due to the property of matrix **L** and **U**, calculating each elements is simplified.

$$\begin{gathered} u_{11}x_1 + u_{12}x_1 + u_{13}x_1 + \cdots + u_{1n}x_1 = y_1 \\ u_{22}x_2 + u_{23}x_3 + \cdots + u_{2n}x_n = y_2 \\ u_{23}x_3 + \cdots + u_{2n}x_n = y_3 \\ \vdots \\ u_{n-1n-1}x_{n-1} + u_{n-1n}x_n = y_{n-1} \\ u_{nn}x_n = y_n \end{gathered} \tag{5}$$

Since every diagonal elements of the upper triangular matrix **U** are equal to 1, we get

$$\mathbf{x_n} = \mathbf{y_n}. \tag{6}$$

And after that, the values of $\mathbf{u_{n-1n}}$ are easily calculated. Similarly when calculating $\mathbf{Ly} = \mathbf{b}$, the equations remain in a simple form as (5). Although some type of matrices are not able to use LU decomposition, due to the simplicity of calculation, LU decomposition is a general method for matrix calculation.

In the program we use LU decomposition as the general algorithm and a comparison to the optimized algorithm.

2.2.2. Tri-diagonal Matrix Algorithm

Tri-diagonal Matrix Algorithm also known as the Thomas algorithm is specified to tri-diagonal matrix. This algorithm returns the solution of

$$\mathbf{Ax} = \mathbf{b},$$

where **A** is a $n \times n$ tri-diagonal matrix, **x** and **b** are $n \times 1$ vectors.

The tri-diagonal algorithm uses forward and backward substitution. The forward substitution iteratively eliminates the elements of the lower diagonal by row reduction. As a result of forward substitution, the original matrix **A** becomes a matrix which has '1' on the diagonal. Afterwards, by backward substitution, the matrix **A** has only one coefficient to solve on each row. Since the element of the last row is the only value, which is '1', $\mathbf{x_n}$ is the modified $\mathbf{b_n}$, and in a chain the other coefficient and solution can be solved.

2.3. Program structure

The program used for this project is designed to execute several different tasks. Each task is accessible by using flags. The main function initially receives parameters from the command line and checks whether it is a valid command line. Command line should always consist of at least two elements: there should be more than one flag and essentially need to have an integer, which specify the size of the matrix. If the command line is valid for the program, variables are declared and initialized. Finally, the flag checking section retrieves the flag and jumps to the function.

You can choose a flag for each solution. Flag '-t' gives you the duration of time when each solution are run. Flag '-s' executes the general solution, flag '-sc' executes the optimized solution, flag '-sLU' executes the LU decomposition, and flag '-err' executes the calculation of relative error. Each solution generates a text file that writes every value of the solution. In case the error is calculated, we will receive a text file that

contains every error of each solution. When you try to calculate the time or count the number of FLOPS, the results will be on the output window.

Figure 1 below shows the command line in Qt. In Figure 1, -sLU is mentioned as a flag and we receive 1000 as the number of mesh points.

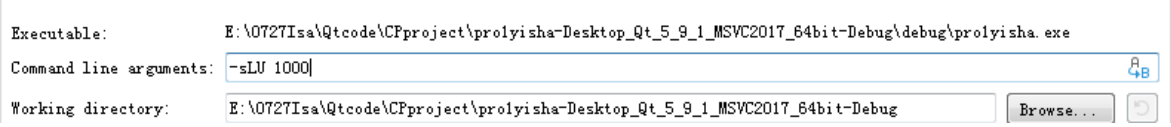


Figure 1. Command line (running LU decomposition with n=10^3).

After the sub-function of each flag are finished, the results are stored into a text file. Each text file has a distinct name depending on the properties. For example, a file output of the solution by LU decomposition of a matrix sized by 10×10 will have a name "solution_LUd_n10.txt".

The sub-functions for each task are based on the algorithms explained above in section 2.2.1 and 2.2.2. For further understanding reading the code uploaded on github will be helpful. You can find the page in the appendix.

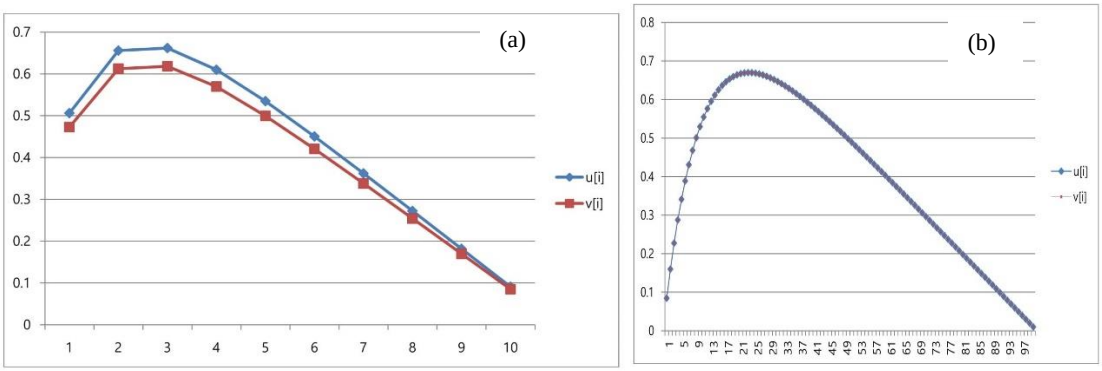
3. Results and Discussion

3.1 Closed-form solution and numerical solution comparing

The closed-form solution and numerical solution, using general algorithm is compared with different n, which can be seen in Figure 2, where matrix **A** is specified as

$$\mathbf{A} = \begin{bmatrix} 2 & -1 & 0 & \cdots \\ -1 & 2 & -1 & 0 \\ 0 & -1 & \ddots & \ddots \\ \vdots & 0 & \ddots & 2 \end{bmatrix} \tag{7}$$

When the size of matrix **A** is set as 10×10, the difference between closed-form solution and numerical solution is obvious. With the increase of matrix size, the difference can be reduced effectively.



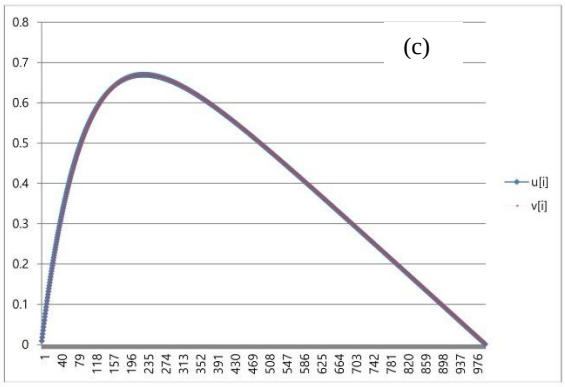


Figure 2. Comparison between closed-form solution $u(i)$ and numerical solution $v(i)$. The size of matrix is 10×10 (a), 100×100 (b), 1000×1000 (c).

3.2 FLOPS counting and CPU time comparison between optimized algorithm and general algorithm

As shown in Table 1, CPU time for optimized algorithm is much shorter than general algorithm (LU decomposition), which means Tri-diagonal Matrix Algorithm is a time saving method.

The number of FLOPS needed in optimized algorithm is given by $3\times (n-1)$, while for general algorithm is n^3. Out of memory for LU decomposition, when the matrix size is $10^5 \times 10^5$, shown in Figure 3. In the case larger memory is available to use, the standard LU decomposition could be run, but with limited hardware, due to lack of memory, the operating system will kill the program. With LU decomposition, we could only get results until $n = 10^5$. This differs according to the size of memory that is available for each computer.

Table 1. FLOPS counting and CPU time comparison between optimized algorithm and general algorithm for specific tri-diagonal matrix.

n		10	10^2	10^3	10^4	10^5	10^6
FLOPS for optimized algorithm (Tri-diagonal Matrix Algorithm)		27	297	2997	29997	..	...
FLOPS for general algorithm		10^3	10^6	10^9	…	…	…
Time (s)	Optimized algorithm	5.987e-06	1.3256e-05	7.2268e-05	0.00056138	/	/
	General algorithm	0.00103399	0.0021548	0.133163	13.4013	/	/

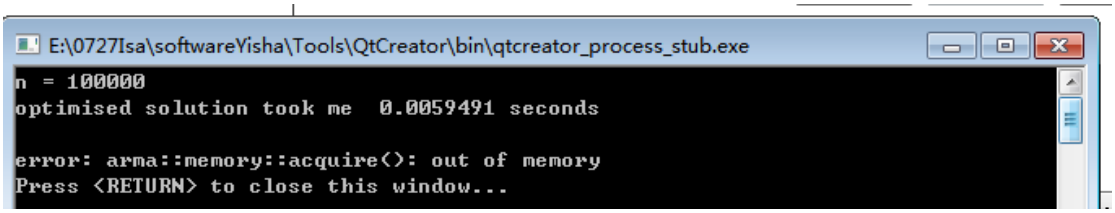


Figure 3. LU decomposition stops working due to lack of memory when n≥105.

3.3 Relative error computing

The relative error between closed-form solution and numerical solution is calculated as

$$\varepsilon_i = log_{10}\left(\left|\frac{v_i - u_i}{u_i}\right|\right). \quad (8)$$

For each step length, the max value of the relative error is extracted and displayed in Table 2. The max relative error decreases with the increasing of n until 10^5. When n is larger than 10^5, the relative error begins to increase (the larger the error, the ε_{max} becomes smaller because of the operation $\varepsilon_i = \log_{10}\left(\left|\frac{v_i - u_i}{u_i}\right|\right)$).

The program fails to make a file of relative errors when n is 1000 or larger due to assertion error (Figure 4).

Table 2. Max value of the relative error with different n

n	10	100	1000	10^4	10^5	10^6	10^7
ε_{max}	-1.1797	-3.08804	-5.08005	-7.07928	-8.84297	-6.07547	-5.52523

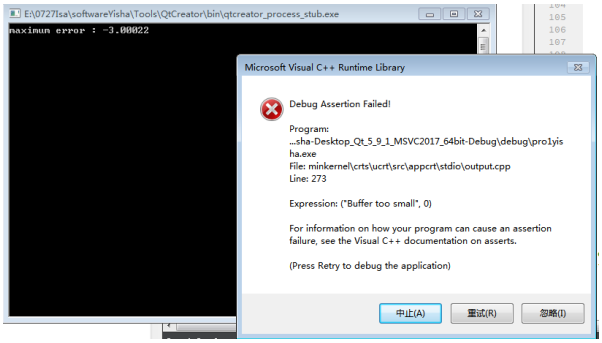


Figure 4. Error due to small buffer

4. Conclusion and Perspectives

By comparing optimized algorithm and general one (LU decomposition), we find that the LU decomposition uses more memory. Since all the elements of tri-diagonal matrix (optimized algorithm) are zero except for those on and immediately above and below the leading diagonal, instead of calculating every element in $n \times n$ arrays, we can only calculate the elements that matters and save memory. Moreover, when the size of the matrix increases, due to the number of FLOPS, the time consumed by each algorithm has a different state of increase. The optimized algorithm is rather linear, while the LU decomposition increases more steeply. Therefore, in case of solving tri-diagonal matrices using optimized algorithm is more reasonable for both perspectives of hardware and software.

Appendix with extra material

Github address for full code :
https://github.com/isabel2017/C.P.Projects-Yisha---Hyejin/blob/0929-Isabel/project10922.cpp

Bibliography

David Potter, *Computational Physics*, Imperial College, London, John Wiley & Sons, 1973: 82-87