**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*=105, we get the minimum value for maximum relative error.

## 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 change of *n*. Detail methods and algorithms are described in the following section.

## 2. Methods

**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,

(1)

where , can be rewritten as,

**Av** = . (2)

In equation (2) **A** is a matrix, and **v**, are vectors.

In our case, when , the closed solution is . The following shows how to rewrite continuous functions into a equation of matrix.

Let *h* the step length,

and the discrete expression of and .

The equation

(3)

, where

we can list down the equations as

Let’s rearrange the elements

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,

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

The first derivative of is

The second derivative of is

Therefore,

**2.2.1 LU Decomposition**

LU decompositions is a method to separate a matrix to two lower triangular matrix and upper triangular matrix in order to relief the difficulty of calculations. This method has four basic steps.

1. Rewrite to .

2. Define a new vector **y** by , and write to .

3. Find **y** with the general solution.

4. Insert **y** to , and find **x** with the general solution.

It might seem increasing the number of operations, but due to the shape of matrix **L** and **U**, calculating each elements become simpler.

(1)

In (1), every diagonal elements of the upper triangular matrix **U** are 1. Therefore, we get

.(2)

And after that, the values of areeasily calculated. Similarly when calculating , the equations remain in a simple form as (1). 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**

You can tell it from the name of the algorithm that it is an algorithm specifically for a tri-diagonal matrix. Also known as the Thomas algorithm, this algorithm returns the solution of

,

where **A** is a tri-diagonal matrix, **x** and **b** are 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’, **xn** is the modified **bn**, and in a chain the other coefficient and solution can be solved.

**2.3. The Program**

The program used in this report to get results is structured to be able to execute several different tasks. Each task will be able to access by using flags. Information about flags will be described in this section latter. 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 a integer which specify the size of the matrix. If the command line is valid for the program, variables are announced 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 for the value of the number of mesh points.

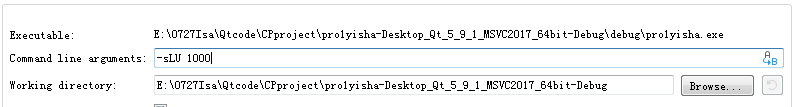


Figure 1. Running LU decomposition when *n*=103

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 1010 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

**(b) 1.** Set up the general algorithm (assuming different values for the matrix elements) for solving this set of linear equations.

**2.** Find also the precise number of floating point operations needed to solve the above equations.

**3.** Then you should code the above algorithm and solve the problem for matrices of the size 10×10, 100×100 and 1000×1000.

**4.** Compare your results (make plots) with the closed-form solution for the different number of grid points in the interval x ∈(0,1).

**(c) 1.** Specialize your algorithm to the special case and find the number of floating point operations for this specific tri-diagonal matrix.

**2.** Compare the CPU time with the general algorithm from the previous point for matrices up to n =106 grid points.

Table 1. Numbers of FLOPS for specific tri-diagonal matrix and CPU time comparison between tri-diagonal matrix and the general algorithm

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| n | | 10 | 102 | 103 | 104 | 105 | 106 |
| FLOPS for optimized algorithm | | 27 | 297 | 2997 | 29997 | .. | ... |
| FLOPS for general algorithm | | 103 | 106 | 109 | … | … | … |
| Time  (s) | Optimized algorithm  (tri-diagonal matrix) | 5.987e-06 | 1.3256e-05 | 7.2268e-05 | 0.00056138 | / | / |
| general algorithm | 0.00103399 | 0.0021548 | 0.133163 | 13.4013 | / | / |

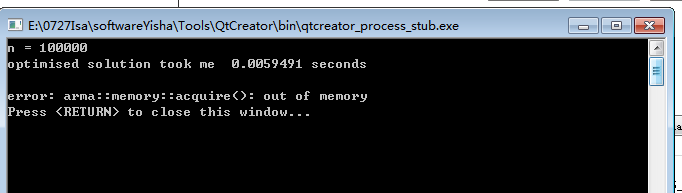


Figure 1. LU decomposition stops working due to lack of memory when *n*≥105

As shown in Table 1, CPU time for optimized algorithm (specific tri-diagonal matrix) is much shorter than general algorithm (LU decomposition).

Due to lack of memory, the program could not do LU decomposition when n was set larger than 105, which can be seen in Figure 1.

The number of FLOPS needed in opt

imized algorithm is given by 3\*(*n*-1).

**(d) 1.** Compute the relative error

as function of for the function values *ui* and *vi*. For each step length, extract the max value of the relative error. Try to increase n to *n*=107. Make a table of the results and comment your results. You can use either the algorithm from b) or c).

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

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| n | 10 | 100 | 1000 | 104 | 105 | 106 | 107 |
|  | -1.1797 | -3.08804 | -5.08005 | -7.07928 | -8.84297 | -6.07547 | -5.52523 |

The max relative error decreases with the increasing of n until 105. However, when n is larger than 105, the relative error begins to increase, which can be seen from Table 2 (the larger the error, the becomes smaller because of the operation ).

The program did not make a file of errors when n is 1000 or larger due to assertion error.

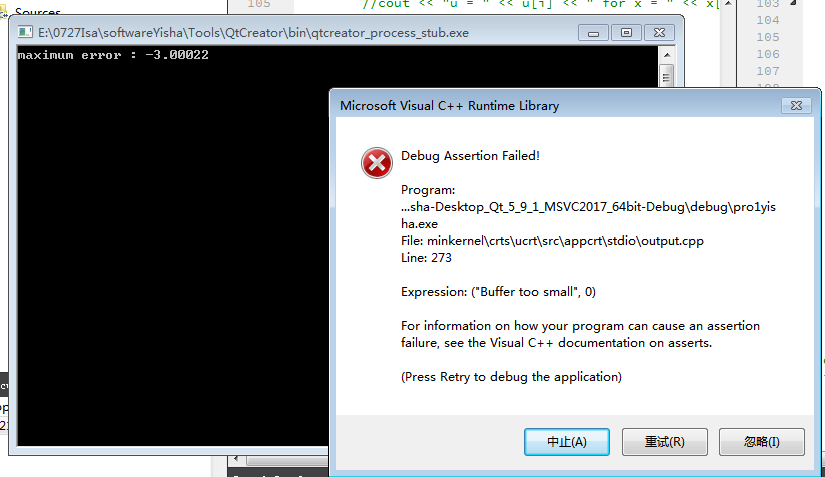


Figure 2. Error due to small buffer

**2.** Make a table of the results and comment the differences in execution time How many floating point operations does the LU decomposition use to solve the set of linear equations? Can you run the standard LU decomposition for a matrix of the size 105×105? Comment your results.

When you have n linear equations, you need n3 floating point operations. Out of memory for LU decomposition when the matrix size is . 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 is smaller than 1000. This differs to the size of memory that is able to use for each computer.

## 4. Conclusion and Perspectives

We found that the LU decomposition uses more memory compared to the optimized algorithm. 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 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, pg 82-87