PHY905 - Project 1

Mengzhi Chen[1]

[1]Department of Physics and Astronomy, NSCL/FRIB Laboratory
Michigan State University, East Lansing, Michigan 48824, USA

*To whom correspondence should be addressed; E-mail: chenme24@msu.edu.

Feburary 3, 2018

In this project, we implement three numerical solvers for one-dimensional Poisson's equation with Dirichlet boundary condition. The special solver has the highest efficiency but the narrowest application; the LU solver can be widely used but extremely slow. In addition, we show relative errors cannot decline all the way with the decreasing step size.

1 Introduction

Poisson's equation plays an important role in classical electrodynamics. It's a partial differential equation describing the potential field generated by given charge distributions. Under spherical symmetric situation, Poisson's equation has a form (*1*)

$$\frac{1}{r^2}\frac{d}{dr}(r^2\frac{d\Phi}{dr}) = -4\pi\rho(r).$$

With substitutions: $u(r) \rightarrow \Phi(r)r$, $-4\pi\rho(r)r \rightarrow f(r)$ and $x \rightarrow r$. It becomes a one-dimensional second order differential equation written as

$$-u''(x) = f(x).$$

In this project, we solve this equation in the region $x \in (0, 1)$ with the boundary conditions: $u(0) = u(1) = 0$. In order to solve equation numerically, we approximate $u(x)$ by n-1

discretized points and denote them as v_i with $i = 1, 2, ...n$. The boundary conditions are: $v_0 = v_n = 0$, the spacing between two points is $h = 1/n$, so $v_i = v_0 + ih$. Taylor expanding v_{i+1} and v_{i-1} around v_1 gives

$$v_{i+1} = v_i + v_i'h + \frac{v_i''}{2}h^2 + \frac{v^{(3)}}{3!}h^3 + O(h^4), \tag{1}$$

$$v_{i-1} = v_i - v_i'h + \frac{v_i''}{2}h^2 - \frac{v^{(3)}}{3!}h^3 + O(h^4). \tag{2}$$

Add equation (1) and (2) and reorder terms, we get the three point formula

$$\frac{-(v_{i+1} + v_{i-1} - 2v_i)}{h^2} = v_i'' + O(h^2) \approx f_i, \tag{3}$$

where $f_i = f(x_i)$. We can rewrite equation (3) as a group of linear equations with the form

$$\hat{\mathbf{A}}\mathbf{v} = \bar{\mathbf{f}},$$

where $\mathbf{v} = (v_1, v_2, ...v_n)^T$, $\bar{\mathbf{f}} = (f_1, f_2, ...f_n)^T h^2$ and $\hat{\mathbf{A}}$ is an $n \times n$ tridiagonal symmetric matrix of the form

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

Our mission is then to develop solvers solving these equations.

In section 2, the general, special, LU decomposition solvers and their mathematical backgrounds are introduced. Three algorithms' complexities are listed. In section 3, solvers' performances are tested and their errors are analyzed. In the end, we summarize our work and draw conclusions in section 4.

2 Method and algorithm

2.1 Gaussian elimination

For a broader usage of our solver, suppose $\hat{\mathbf{A}}$ is a $n \times n$ general tridiagonal matrix, then equations take the form

$$\begin{bmatrix} b_1 & c_1 & 0 & \cdots & \cdots & 0 \\ a_1 & b_2 & c_2 & 0 & \cdots & 0 \\ 0 & a_2 & b_3 & c_3 & \cdots & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & 0 \\ 0 & \cdots & \cdots & a_{n-2} & b_{n-1} & c_{n-1} \\ 0 & \cdots & \cdots & 0 & a_{n-1} & b_n \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ \cdots \\ \cdots \\ \cdots \\ v_n \end{bmatrix} = \begin{bmatrix} \bar{f}_1 \\ \bar{f}_2 \\ \cdots \\ \cdots \\ \cdots \\ \bar{f}_n \end{bmatrix}$$

We then do Gaussian elimination *(1, 2)* row by row. The first step is to subtract $\hat{\mathbf{A}}_{\mathbf{21}}$ by following operation:

$$\left[\begin{array}{cccccc|c} b_1 & c_1 & 0 & \cdots & \cdots & 0 & \bar{f}_1 \\ a_1 & b_2 & c_2 & 0 & \cdots & 0 & \bar{f}_2 \\ 0 & a_2 & b_3 & c_3 & \cdots & 0 & \cdots \\ \vdots & \vdots & \ddots & \ddots & \ddots & 0 & \cdots \\ 0 & \cdots & \cdots & a_{n-2} & b_{n-1} & c_{n-1} & \cdots \\ 0 & \cdots & \cdots & 0 & a_{n-1} & b_n & \bar{f}_n \end{array}\right] \rightarrow \left[\begin{array}{cccccc|c} b_1 & c_1 & 0 & \cdots & \cdots & 0 & \bar{f}_1 \\ 0 & b_2 - \frac{a_1 c_1}{b_1} & c_2 & 0 & \cdots & 0 & \bar{f}_2 - \frac{c_1 \bar{f}_1}{b} 1 \\ 0 & a_2 & b_3 & c_3 & \cdots & 0 & \cdots \\ \vdots & \vdots & \ddots & \ddots & \ddots & 0 & \cdots \\ 0 & \cdots & \cdots & a_{n-2} & b_{n-1} & c_{n-1} & \cdots \\ 0 & \cdots & \cdots & 0 & a_{n-1} & b_n & \bar{f}_n \end{array}\right]$$

Repeat it until all $\hat{\mathbf{A}}_{\mathbf{i+1,i}} (i = 1, 2...n-1)$ elements are eliminated. We can denote new diagonal elements as $\tilde{\mathbf{b}}$ and new $\bar{\mathbf{f}}$ as $\tilde{\mathbf{f}}$. Formulas for them can be generalized

$$\tilde{b}_i = b_i - \frac{a_{i-1} c_{i-1}}{b_{i-1}} \tag{4}$$

$$\tilde{f}_i = \bar{f}_i - \frac{\bar{f}_{i-1} c_{i-1}}{b_{i-1}}. \tag{5}$$

Above substitutions start from $\tilde{b}_1 = b_1$ and $\tilde{f}_1 = \bar{f}_1$ in sequence of $i = 2, 3, ...n$. After these forward substitutions, the equations become

$$\begin{bmatrix} \tilde{b}_1 & c_1 & 0 & \cdots & \cdots & 0 \\ 0 & \tilde{b}_2 & c_2 & 0 & \cdots & 0 \\ 0 & 0 & \tilde{b}_3 & c_3 & \cdots & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & 0 \\ 0 & \cdots & \cdots & 0 & \tilde{b}_{n-1} & c_{n-1} \\ 0 & \cdots & \cdots & 0 & 0 & \tilde{b}_n \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ \cdots \\ \cdots \\ \cdots \\ v_n \end{bmatrix} = \begin{bmatrix} \tilde{f}_1 \\ \tilde{f}_2 \\ \cdots \\ \cdots \\ \cdots \\ \tilde{f}_n \end{bmatrix}$$

We then apply backward substitution from the last row. Starting from $v_n = \tilde{f}_n/\tilde{b}_n$, we can generalize expressions

$$u_i = \frac{\tilde{f}_i - c_i u_{i+1}}{\tilde{b}_i} \tag{6}$$

where i in sequence of $i = n-1, n-2, ...1$. It stops until every v_i is solved out.

2.2 A general solver

As derived above, only non-zero elements of matrix matter in our calculation. Together with the fact that tridiagonal matrix is extremely sparse. It inspires us a great idea: instead of storing the whole matrix with a lot of useless zeros, why don't we only allocate space for three diagonal and off-diagonal vectors? By doing so, it saves us a lot of memory space and speeds up our algorithms by abandoning meaningless calculations. Based on these considerations, a general solver is implemented and the pseudocode is stated in Algorithm 1.

Input: float vectors: $\vec{a}$, $\vec{b}$, $\vec{c}$ and $\vec{\bar{f}}$ from exact solution with length n+1
Output: solution: $\vec{u}$

// Set Boundary conditions;
$u_0 = u_n = 0$;
// Forward Substitution;
for $i = 2; i \leq n-2; i++$ **do**
    $fac = c_i/b_i$;
    $b_i = \bar{f}_i - (a_{i-1} * fac)$;
    $\bar{f}_i = \bar{f}_i - (\bar{f}_{i-1} * fac)$;
end
// Backward Substitution;
$u_{n-1} = \bar{f}_{n-1}/b_{n-1}$;
for $i = n-2; i \geq 1; i--$ **do**
    $u_i = (\tilde{f}_i - c_i u_{i+1})/b_i$;
end
return $\vec{u}$;

Algorithm 1: The general solver for tridiagonal matrix

We can see from above algorithm that by pre-calculating c_i/b_i, this general solver takes 8 FLOPS for one u_i. Thus, the total number of operations should be

$$8(n-2) \sim \mathcal{O}(8n) \text{ FLOPS}.$$

2.3 A special solver

As shown in section 2.1, for one dimensional Poisson's equation, the matrix $\hat{\mathbf{A}}$ is symmetric and all diagonal elements are 2 and off-diagonal elements equal to -1. By substituting these numbers into equation (4), we can generalize a simpler expression for $\tilde{b}_i$ as

$$\tilde{b}_i = \frac{i+1}{i} \tag{7}$$

Also, equation (5) and (6) can be simplify to forms

$$\tilde{f}_i = \bar{f}_i + \frac{\bar{f}_{i-1}}{b_{i-1}}. \tag{8}$$

$$u_i = \frac{\tilde{f}_i + u_{i+1}}{\tilde{b}_i} \tag{9}$$

Bases on above equations (7), (8) and (9), a special solver is then implemented and its pseudocode is stated in Algorithm 2. $\tilde{b}_i$ can be quickly calculated through vectorization. We can

Input: float vectors: $\vec{b}$ and $\vec{\bar{f}}$ from exact solution with length n+1
Output: solution: $\vec{u}$
// Set Boundary conditions;
$u_0 = u_n = 0$;
// Initialize $\vec{b}$;
$b_i = (i+1)/i$;
// Forward Substitution;
for $i = 2; i \leq n-2; i++$ **do**
  $\bar{f}_i = \bar{f}_i + \bar{f}_{i-1}/b_{i-1}$;
end
// Backward Substitution;
$u_{n-1} = \bar{f}_{n-1}/b_{n-1}$;
for $i = n-2; i \geq 1; i--$ **do**
  $u_i = (\tilde{f}_i + u_{i+1})/b_i$;
end
return $\vec{u}$;

Algorithm 2: The special solver for a special tridiagonal matrix

conclude from equations (8) and (9), that this special solver costs only 4 FLOPS for one u_i. Thus, the total number of operations should be

$$4(n-2) \sim \mathcal{O}(4n) \text{ FLOPS}.$$

Theoretically, the special solver speeds up a factor of 2 in certain problem compared with the general solver.

2.4 LU decomposition and solver

In linear algebra, it can be proved that an inevitable matrix $\hat{\mathbf{A}}$ can be LU(LU stands for lower and upper triangular matrices) decomposed. (*2, 3*). One intuitive way is through Gaussian elimination. LU decomposition in matrix form can be written as

$$\hat{\mathbf{A}} = \hat{\mathbf{L}}\hat{\mathbf{U}} = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ l_{21} & 1 & 0 & \cdots & 0 \\ l_{31} & l_{32} & 1 & \cdots & 0 \\ \vdots & & \cdots & \ddots & 0 \\ l_{n1} & l_{n2} & l_{n3} & \cdots & 1 \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} & \cdots & u_{1n} \\ 0 & u_{22} & u_{23} & \cdots & u_{2n} \\ 0 & 0 & u_{33} & \cdots & u_{3n} \\ \vdots & & \cdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & u_{nn} \end{bmatrix}.$$

So linear equations $\hat{\mathbf{A}}\mathbf{v} = \bar{\mathbf{f}}$ can be expressed as $\hat{\mathbf{L}}(\hat{\mathbf{U}}\mathbf{v}) = \bar{\mathbf{f}}$. We then define an auxiliary vector $\mathbf{c}$ which satisfies $\mathbf{c} = \mathbf{U}\mathbf{v}$. Then, we can divide problems into two steps.

- Step1: Solve $\hat{\mathbf{L}}\mathbf{c} = \bar{\mathbf{f}}$ for $\mathbf{c}$
- Step2: Having c from last step, solve $\hat{\mathbf{U}}\mathbf{u} = \mathbf{c}$ to get $\mathbf{u}$

In the first step, L is a lower triangle matrix. So forward substitution is used in the first step. Starting from $c_1 = \bar{f}_1$, we get expressions for c_i

$$c_i = \bar{f}_i - \sum_{j=1}^{i-1} l_{ij}\bar{f}_j \tag{10}$$

where $i = 2, 3, ...n-1$. We then use $\mathbf{c}$ to proceed the second step. In that step, backward substitution is used for U is an upper triangle matrix. (For clarification, when u has one subscript, it stands for elements from vector $\mathbf{u}$. Two subscripts mean element in matrix $\hat{\mathbf{U}}$). Starting from $u_{n-1} = c_{n-1}/u_{n-1,n-1}$, we generalize formula for u_i

$$u_i = \frac{c_i - \sum_{j=i+1}^{n-1} u_{ij}u_j}{u_{ii}} \tag{11}$$

where $i = n-2, n-3, ...1$. A solver bases on LU decomposition (LU solver) is implemented and shows in Algorithm 3 (*4*).

The complexity of LU decomposition is $\mathcal{O}(2/3n^3)$, same as general Gaussian decomposition. When we are solving k linear equation sets with same $\hat{\mathbf{A}}$ but varying $\bar{\mathbf{f}}$, the total complexity of Gaussian elimination is $\mathcal{O}(2/3kn^3)$. At that time, LU decomposition shows superiority; because it only needs to deal with $\hat{\mathbf{A}}$ once and then does k times substitutions. Its complexity is $\mathcal{O}(2/3n^3 + kn^2)$ which has a large difference for large n value.

Input: float $(n-1) \times (n-1)$ matrix $\hat{\mathbf{A}}$ and $\vec{\bar{f}}$ from exact solution with length n+1
Output: solution: $\vec{u}$

// Set Boundary conditions;
$u_0 = u_n = 0$;
// LU decomposition(using armadillo lib);
$lu(L, U, A)$;
// Forward Substitution;
for $i = 2; i \leq n-2; i++$ **do**
    for $j = 1; j \leq i-1; j++$ **do**
        $c_i = \bar{f}_i + l_{ij}\bar{f}_j$;
    end
end
$u_{n-1} = c_{n-1}/u_{n-1,n-1}$ // Backward Substitution;
for $i = n-2; i \geq 1; i--$ **do**
    for $j = i; j \leq n-1; j++$ **do**
        $u_i = (c_i - u_{ij}u_j)$;
    end
    $u_i = u_i/u_{ii}$
end
return $\vec{u}$;

Algorithm 3: LU decomposition solver

3 Problems, results and discussions

- Clang compiler is used and codes are run on Macbook macOS High Sierra, 1.3 GHz Intel Core m7 processor. (see src repository for more information)
- All data are average of three repetitions.

3.1 Validity and efficiency

We try to solve a system with $f(x) = 100e^{-10x}$ and boundary conditions $u(x = 0) = u(x = 1) = 0$. It has an analytic solution $u(x) = 1 - (1 - e^{-10})x - e^{-10x}$ for verification.

The general solver is tested with different steps by setting $n = 10, 100$ and 1000. The analytic and numerical results are shown in figure 1. We can see that when $n = 10$, the numerical error is quite large. However, as n increases to 100 and 1000, we can hardly see differences between analytic and numerical solutions. For further analysis of errors, we need a more quantitative discussion.

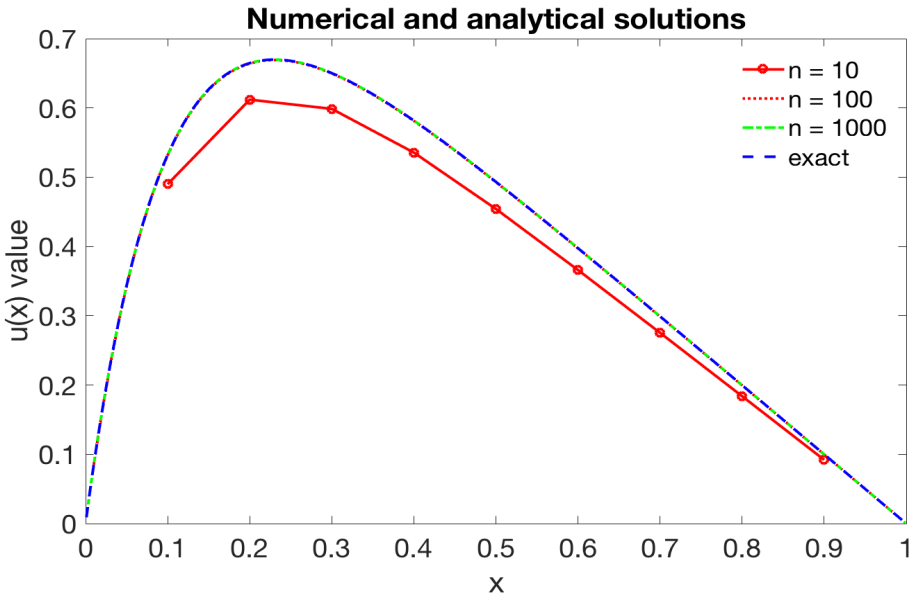

Fig. 1: Analytic and numerical solutions for n = 10, 100 and 1000 in interval (0,1).

As we discussed in section 2.2, the special solver has a higher efficiency in this problem. Therefore, we would like to verify this point. In table 1, execution times for two solvers with increasing n up to 10^7 are listed (only forward and backward substitution processes are included). As we can see, the special solver does have higher efficiency. However, it can't reduce half of the execution as predicted. We believe it's normal because we only cut down half of FLOPS. There are still overheads coming from loop initialization, data reading and writing, etc where we do nothing on. Luckily, as increasing n, programs become more compute bound which means we can save more time by using special solver.

Size n	Execution time of General Solver (ms)	Execution time of Special Solver (ms)
10^1	0.003	0.003
10^2	0.005	0.004
10^3	0.042	0.019
10^4	0.330	0.227
10^5	2.850	2.380
10^6	25.876	23.508
10^7	253.118	197.520

Tab. 1: Execution times for general and special solvers for several different size n.

3.2 Error analysis

Driven by the motivation of increasing numerical precision, we carry out an error analysis for our solvers.

As stated in reference (*5*), the total error comes from both truncation approximation and round-off error. It can be written as

$$\epsilon_{tot} = \epsilon_{app} + \epsilon_{ro}.$$

In our case of using three point formula and one FLOP for double precision numbers,

$$\epsilon_{app} \approx \frac{f_0^{(4)}}{12}h^2; \quad |\epsilon_M| \leq 10^{-15};$$

$$|\epsilon_{tot}| \leq \frac{f_0^{(4)}}{12}h^2 + \frac{2\epsilon_M}{h^2}. \tag{12}$$

In order to get suitable h, we calculate the derivative of $|\epsilon_{tot}|$ with regard to h in equation (12). When

$$h = (\frac{24\epsilon_M}{f_0^{(4)}})^{1/4}, \tag{13}$$

it gives the minimum total error. Using formula $f(x = 1)$ in our problem, we get $h \sim 10^{-5}$. In order to see how total error varies with spacing h ($h = 1/n$), we define logarithm relative error as

$$\epsilon_i = \log_{10}\left(\left|\frac{u_i - u_i(exact)}{u_i(exact)}\right|\right).$$

In figure 2, we show how $max(\epsilon_i)$ varies with $\log_{10} h$. At the same time, some of these values are listed in table 2. We can see both solvers have the lowest error at about $h \sim 10^{-5} - 10^{-6}$. It

Size n	Spacing h	Maximum ϵ_i of General Solver	Maximum ϵ_i of Special Solver
10^1	10^{-1}	-1.1006	-1.1006
10^2	10^{-2}	-3.0794	-3.0794
10^3	10^{-3}	-5.7092	-5.7092
10^4	10^{-4}	-7.0792	-7.0792
10^5	10^{-5}	-8.8430	-9.0790
10^6	10^{-6}	-6.0755	-10.1757
10^7	10^{-7}	-5.5252	-9.6534

Tab. 2: Relative errors for general and special solvers for several different size n.

agrees with our previous prediction. In the large h region, truncation error dominates, so we can see two solvers have identical errors because they use the same formula. After the lowest point, relative error grows with decreasing h because round-off error dominates now; it also explains why two solvers behave diversely at small h region. Their different calculation processes bring distinct round-off errors.

In a word, we can not achieve infinite small error; so we should better set the step size to gain high efficiency and accuracy.

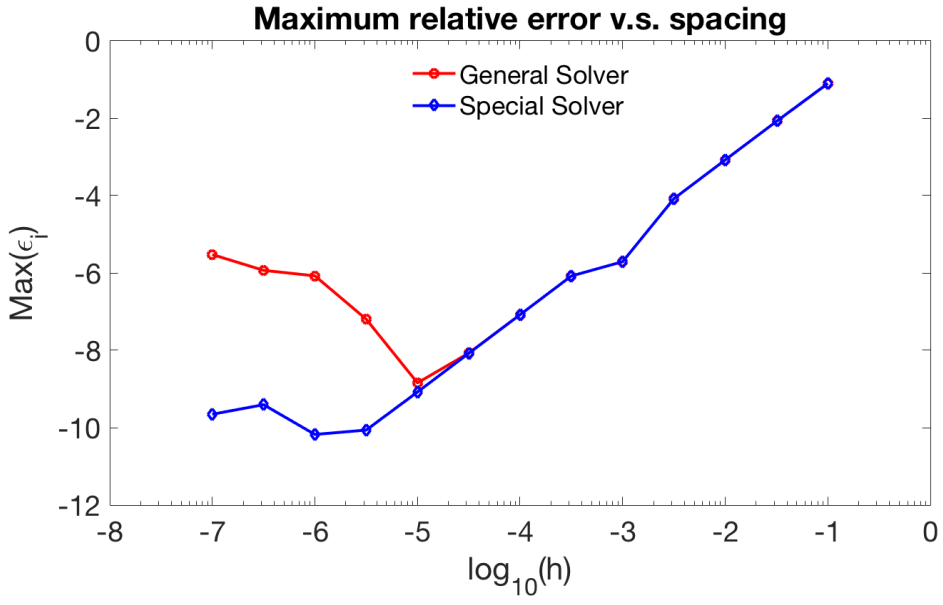

Fig. 2: Maximum logarithm relative errors for general and special solvers with different spacing

Size n	Execution time of General Solver (ms)	Execution time of Special Solver (ms)	Execution time of LU decomposit Solver (ms)
10^1	0.003	0.003	0.045
10^2	0.005	0.004	0.748
10^3	0.042	0.019	75.208
10^4	0.330	0.227	46590.267
10^5	2.850	2.380	Nan
10^6	25.876	23.508	Nan
10^7	253.118	197.520	Nan

Tab. 3: Execution times for general, special and LU solvers for several different size n.

3.3 Comparison with LU solver

To show the superiority of our solvers, I compared the execution times of general, special and LU solvers. The results are shown in table 3. As we can see, the execution time of LU solver increases extremely fast compared with others. It can be explained as we stated in section 2. The complexity of LU solver is $\mathcal{O}(2/3n^3)$ and others two are $\mathcal{O}(kn)$.

There is also one more thing worth noticing. We are not able to attain results for $n > 10^5$. A $10^6 \times 10^6$ double precision matrix occupies more than 900 GB space; nevertheless, there is not enough space to store it in a personal computer. Eventually, it leads a collapse of LU solver.

The Special solver is faster than the general one; they two are much faster than the LU solver. But we should keep in mind that the LU solver can be applied to any linear equations and the general solver can deal with general tridiagonal matrix; but special solver only works with our problem. In sum, the moment we get efficiency is also the minute we lose generality. Sometimes, we have to trade off carefully between these two.

4 Summary and conclusions

In this project, we develop the general and special solvers for numerically solving one-dimensional Poisson's equation. Their correctness is verified by solving $f(x) = 100 \exp^{-10x}$ whose analytically solution is known. Through measurement of their execution times, we see that the special solver has higher efficiency but the general solver can be broadly used to any tridiagonal matrix. Error analysis indicates only the appropriate n gives the smallest total relative error. It inspires

us to carefully choose step size to get both high efficiency and precision. In our case, $n = 10^6$ is the best choice for the special solver.

In addition, these two solvers are compared with the LU solver. The general and special solvers show superior speed advantage. However, they also have a narrower range of application (tridiagonal matrix only). On the contrary, the LU solver can deal with any matrix but may subject to limited memory space and low speed. We should carefully choose solvers for specific problems. As we are solving a one-dimensional Poisson's equation, the special solver should be our top option

5 Comments

Personally speaking, I think this project is very suitable as a beginning of our course. It combines widely seen Poisson's equation with basic linear equations solvers. Professor also reminds us some important points in the numerical calculation such as the origin of errors, algorithms selecting. I appreciate patience guidance from Professor Morten. I also look forward to learning more from future projects.

References and Notes

1. Hjort-Jensen,M., Spring 2018, Computational Physics, Matrices and linear algebra. Assessible in: https://compphysics.github.io/ComputationalPhysicsMSU/doc/web/course.

2. T. Sauer, *Numerical Analysis (2nd)* (Addison-Wesley, New Jersey, 2012).

3. G. H. Golub, C. F. Van Loan, *Matrix computations*, vol. 3 (JHU Press, 2012).

4. C. Sanderson, R. Curtin, *Journal of Open Source Software* (2016).

5. Hjort-Jensen,M., Spring 2018, Computational Physics, Introduction to programming. Assessible in: https://compphysics.github.io/ComputationalPhysicsMSU/doc/web/course.