Numerical Linear Algebra methods for the 1D Poisson equation

Timofey Golubev, Hao Lin, Xingze Mao

February 6, 2017

Abstract

In this project, we transform a Dirichlet boundary value problem of Poisson equation into the problem of solving a matrix equation. The matrix of concern is tridiagonal, which enables us to apply three different methods: a general Gaussian elimination (GE) method for tridiagonal matrices, a optimized Gaussian elimination method (Optimized GE) for tridiagonal matrices with identical diagonal entries and identical off-diagonal entries, and a generic LU decomposition approach. We benchmark and discuss the results, accuracy, and algorithm efficiency across all methods.

1 Introduction

Differential equations are playing an increasingly important role in scientific research, as they provide relatively accurate descriptions and models of our physical world. Only a very small subset of them, however, are blessed to have analytic solutions. As a result, the search for stable and accurate numerical solvers of differential equations is of tremendous interest. Runge-Kutta methods and Numerov's method are among the popular choices for tackling *initial value problems*, but they are not well-tuned for *boundary value problems*.

In this project, we would like to look at the one-dimensional Poisson equation,

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

with *Dirichlet* boundary conditions $u(0) = 0$ and $u(1) = 0$ on the interval $[0\,,\,1]$. $f(x)$ is referred to as the *source* function.

2 Reduction of Poisson equations to linear systems

2.1 Discretization

While functions in physical models are predominantly smooth functions, such nice behaviors cannot be expected from our computing tools and structures. The

discrete manners of computers prompt us to discretize the differential equation.

A grid consisting of $n+2$ equally spaced grid points (aka mesh points), x_i, is applied to the interval $[0\,,\,1]$. The spacing between immediately adjacent grid points is $h = 1/(n+1)$. We then have

$$x_i = ih,$$

for all $i = 0, 1, ..., n+1$. In particular, $x_0 = 0$ and $x_{n+1} = 1$. The discretized versions of functions u and f follow naturally from the mesh above, i.e.,

$$\begin{aligned} u_i &= u(x_i), \\ f_i &= f(x_i), \end{aligned}$$

for all x_i. In the three-point stencil scheme, the second derivative of u is approximated as follows,

$$u_i'' = \frac{u_{i+1} + u_{i-1} - 2u_i}{h^2} + O(h^2),$$

for $i = 1, ..., n$. This formula can be proved by assuming $u \in C^4$ and invoking its Taylor expansion up to the fourth order.

Putting everything together, we have a discretized version of Eqn (1)

$$-\frac{u_{i+1} + u_{i-1} - 2u_i}{h^2} = f_i,$$

or

$$2u_i - u_{i+1} - u_{i-1} = h^2 f_i. \tag{2}$$

2.2 Equivalence to a linear system

Eqn (2) holds for every integer i from 1 to n. Grouping them altogether yields a system of linear equations of order n,

$$\begin{aligned} 2u_1 - u_2 &= h^2 f_1, \\ 2u_2 - u_3 - u_1 &= h^2 f_2, \\ &... \\ 2u_i - u_{i+1} - u_{i-1} &= h^2 f_i, \\ &... \\ 2u_n - u_{n-1} &= h^2 f_n. \end{aligned}$$

This linear system can be casted conveniently into a matrix form,

$$\mathbf{Au} = \mathbf{b} \tag{3}$$

by defining a tridiagonal matrix $\mathbf{A}$,

$$\mathbf{A} = \begin{bmatrix} 2 & -1 & 0 & \dots & \dots & 0 \\ -1 & 2 & -1 & 0 & \dots & \dots \\ 0 & -1 & 2 & -1 & 0 & \dots \\ & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & & -1 & 2 & -1 \\ 0 & \dots & & 0 & -1 & 2 \end{bmatrix},$$

and $b_i = h^2 f_i$.

The i-th equation in the linear system corresponds to the i-th row of $\mathbf{A}$ times $\mathbf{u}$ resulting in b_i.

3 Methods

3.1 Gaussian elimination method for tridiagonal matrix

For a general tridiagonal matrix, three arrays are used to save all the elements within the tridiagonal region. In our code, array `diagonal[ ]` saves the diagonal elements and `a[ ]`, `c[ ]` save the elements below and above diagonal line respectively. Array `f[ ]` saves function f from the right side of equation. A forward substitution performs the Gaussian Elimination and a backward substitution calculates the solution which is saved in the array `v[ ]`. For example, to eliminate element a_{21} of matrix A, we subtract the 2nd row of matrix A by the 1st row of A times the ratio $\frac{a_{21}}{a_{11}}$.

```
//Initialization of matrix elements
      for (int i = 1; i<n; i++) {
          diagonal[i] = 2.0; a[i] = -1;  c[i] = -1; }
//Forward substition
      for (int i = 2; i <n; i++) {
          adiag_ratio = a[i-1]/diagonal[i-1];
          diagonal[i] = diagonal[i] - adiag_ratio*c[i-1];
          f[i] = f[i] - adiag_ratio*f[i-1];}
//Backward substitution
      v[n-1] = f[n-1]/diagonal[n-1];
      for (int i = n-2; i >0; i--){
            v[i] = (f[i]-v[i+1]*c[i])/diagonal[i];}
```

With pre-calculation of the ratio `adiag_ratio`, it only contributes one floating point operation (FLOP) for each iteration of the forward substitution loop. For each iteration we also need to update the diagonal elements which uses 2 FLOPs. Finally we update the elements from the right hand side of the matrix equation which uses another 2 FLOPs. The forward substitution loop is repeated $n-2$ times (the equation is discretized into n points but the two endpoints are known from the boundary conditions) so it requires $5(n-2)$ FLOPs total. The backward substitution will also be repeated $n-2$ times and

3 FLOPs are used in each iteration for a total of $3(n-2)$ FLOPs. One additional operation is needed to solve the first linear equation for `v[n-1]`. This gives a total of $8n - 15$ FLOPs.

3.2 Optimized Gaussian elimination method for special tridiagonal matrix

If we look closer at the matrix we generate from the differential equation, we find that besides being a tridiagonal matrix, the matrix has identical off-diagonal elements. Then if we do a few more steps of the Gaussian elimination by hand, we notice that to annihilate the elements below the diagonal line, we will add $-\frac{1}{2}$ of the 1st row to the 2nd row, then add $-\frac{2}{3}$ of the 2nd row to the 3rd row, $-\frac{3}{4}$ of 3rd row to the 4th row, $-\frac{4}{5}$ of 4th row to 5th row... After all the calculations, the diagonal elements will be $\frac{2}{1}, \frac{3}{2}, \frac{4}{3}, \frac{5}{4}, \frac{6}{5} ... \frac{i+1}{i} ...$ Therefore in the new algorithm, diagonal elements are initialized directly by pre-calculation. The use of these pre-calculated elements in the substitutions significantly reduces the number of floating point operations.

```
//initialization of diagonal elements by pre-calculation
for (int i = 1; i<n; i++){
        d[i] = (i+1.0)/((double) i);}
//Forward substition, with only array f to calculate
for (int i = 2; i <n; i++){
        f[i] += f[i-1]/d[i-1];}
//Backward substitution
u[n-1] = f[n-1]/d[n-1];
for (int i = n-2; i >0; i--){
        u[i] = (f[i]+u[i+1])/d[i];}
```

Here, the pre-calculation of diagonal elements uses $2(n-1)$ FLOPs. The forward substitution uses $2(n-2)$ FLOPs and backward substitution uses $2(n-2)+1$ FLOPs. This gives $6n-9$ FLOPs in total which (for large n) is about 25% less than in the general Gaussian Elimination algorithm.

3.3 LU decomposition method to solve linear equations

If a square matrix A and all of its principal leading minors have nonzero determinants, then it can be factorized as the product of a lower triangular matrix L and an upper triangular matrix U : $A = LU$ [2]. The diagonal elements of L are identically 1. With $det|A| = det|LU| = det|L| \times det|U|$, the determinant of A will be the product of all U's diagonal elements.

```
//Define matrix for LU decomp
      for(int i = 0; i < n; i++) {
        for(int j = 0; j < n; j++){
          if (i==j){A[i][j] = 2.0;}
          else if (abs(i-j)==1){A[i][j] = -1.0;}
          else {A[i][j] = 0.0;}
        }
      }
// Perform the LU decomposition and backward substitution.
// Returns soln. in array f.
      LUDecomposition(A, n, indx);
      LUBackwardSubstitution(A, n, indx, f);
```

To solve the linear equation $A\vec{x} = f(x)$, we can do an LU decomposition first and then set $\vec{y} = U\vec{x} = L^{-1}f(x)$. The LU decomposition method can be called from the C++ library Armadillo or written as a function outside of the main function.

```
// Pseudocode for LU decomposition
L = I  // I is the identity matrix
U = A
for i = 1 to n-1
    for j = i+1 to n
        L[i,j] = U[i,j]/U[i,i]
        U[j,i:n] = U[j,i:n] - L[i,j]*U[i,i:n]
```

There are two nested loops adding up to $O(n^2)$ iterations. Within each innermost iteration, 2(n-i+1)+1 operations are performed. Thus, the total operation count is of the order of $O(n^3)$. A careful counting would reveal the total FLOP count of this LU decomposition algorithm scales like $\frac{2}{3}n^3$ [2]. Backward substitution generally requires $O(n^2)$ FLOPs for an $n \times n$ matrix so the LU decomposition operations dominate the FLOP count. In summary, this method uses $\sim \frac{2}{3}n^3$ FLOPs.

4 Results and discussion

4.1 Convergence to the analytic solution

Figure 1 shows both the analytical and numerical results for the Poisson equation $-u''(x) = 100e^{-10x}$ with Dirichlet boundary conditions. When only 10 mesh points (n) are used for discretization, the numerical result has a significant error. However, when we discretize with $n = 100$, the numerical and exact solutions are already nearly identical. Using $n = 1000$ mesh points yields a result even closer to the exact one. We verify that the numerical results from all three algorithms are consistent.

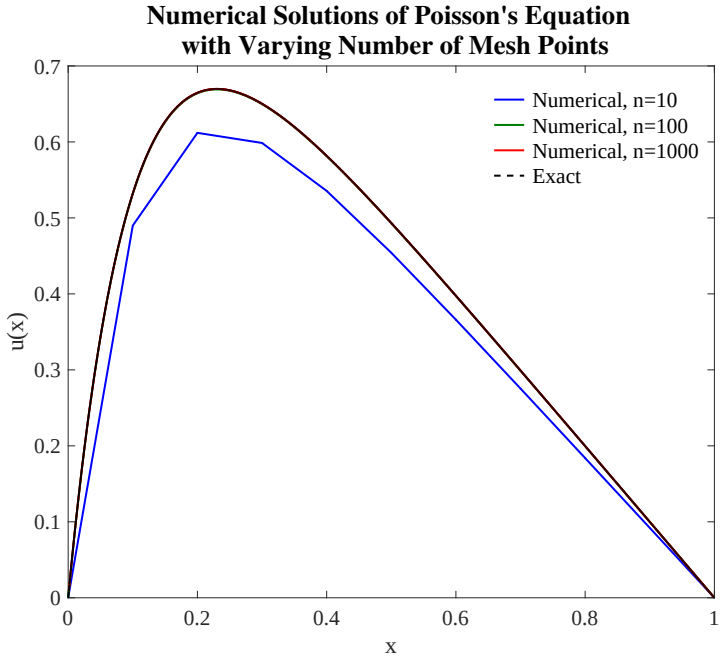

Figure 1: Numerical and analytic results for Poisson equation problem with source function $f(x) = 100e^{-10x}$. Three numerical solutions with varying number of mesh points (n) used are compared with the exact analytic solution.

4.2 Computation time

Table 1 shows the computation time for the three methods we investigated. With smaller step sizes we are discretizing the range $[0, 1]$ into a larger number of mesh points which is expected to give better results. However, the size of the matrix which needs to be processed scales as n^2 where n is the number of mesh points. Therefore, it is expected that more time is needed to carry out the calculations for larger matrices. After running the codes for different step sizes, indeed we found that the computation time is longer for the smaller step sizes. Comparing the results from the three methods using the same number of mesh points, we see that the Optimized GE method is the fastest. This was expected since the method uses the least FLOPs. Clearly, the LU decomposition method takes more time than both Gaussian elimination methods. We find that computation time for LU decomposition algorithm increases drastically with decreasing step size and we were unable to run the LU decomposition code for a step size of $< h^{-4}$ (or equivalently $n > 10^4$). The drastically increasing computation time for LU is due the algorithm requiring on the order of n^3 FLOPs.

step size h	GE-method (s)	Optimized GE-method (s)	LU decomposition (s)
10^{-1}	2.266e-06	2.265e-06	1.057e-05
10^{-2}	8.307e-06	8.684e-06	1.728e-03
10^{-3}	7.552e-05	7.325e-05	1.414
10^{-4}	8.363e-04	7.510e-04	2809
10^{-5}	9.895e-03	6.072e-03	N/A
10^{-6}	8.638e-02	5.745e-02	N/A
10^{-7}	6.354e-01	5.894e-01	N/A

Table 1: Computation time for different step sizes for three different algorithms

4.3 Error Analysis

The numerical results of the two Gaussian elimination methods are compared with the analytic result by calculating the error (Figure 2). We define the relative error for each mesh point i as

$$\epsilon_i = log_{10}\big(|\frac{v_i - u_i}{u_i}|\big)\,.$$

The relative error is calculated for all mesh points. Next we look at the relationship between relative error and step size h. The codes are run for step sizes from 10^{-1} to 10^{-7}. In order to simplify the analysis, we take only the maximum error for each step size used. We can see that for the Optimized GE method, the relative error decreases linearly with step size until step size reaches 10^{-6}. At $h < 10^{-6}$, rounding errors due to limitations of storing numbers in a computer, result in a larger error. Therefore, $h = 10^{-6}$ will give the most accurate numerical result for this particular problem when using these algorithms. In the general Gaussian elimination algorithm the error increases at $h < 10^{-5}$. We believe that the pre-calculations done in the optimized GE code reduce the rounding error and thus achieve a lower relative error. Therefore, optimizing a code by pre-calculating quantities which are used repetitively not only increases computation speed, but also makes the numerical results more accurate.

We find that the LU decomposition method has significantly higher relative error than the Gaussian elimination methods. The relationship between error and step size is linear. Extending this linear relationship, in order for LU decomposition calculations to achieve low errors comparable with the Gaussian elimination methods, one would probably have to use step size of $< 10^{-5}$. At those small step sizes, there is a good chance that rounding errors would cause the error to increase. Calculations with time step size beyond 10^{-4} are not carried out, due to the intensive amount of CPU time and memory required.

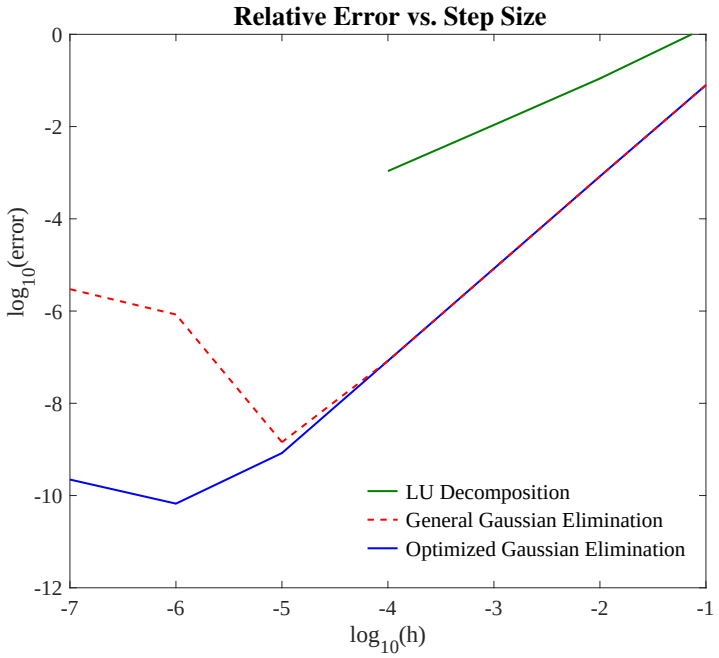

Figure 2: Dependence of relative error between the numerical and analytic result on chosen step size for the three algorithms which were tested

5 Conclusion

We have shown that the Poisson equation can be discretized and converted into a tridiagonal matrix equation. Gaussian elimination and LU-decomposition methods are widely employed to solve problems of this type. The error in the numerical result and computation time can differ significantly depending on which algorithm is chosen and how it is implemented. In order to improve the efficiency of the algorithm, we wrote an optimized Gaussian elimination code, making use of the fact that discretization of the Poisson equation gives a tridiagonal matrix with identical off-diagonal elements and identical diagonal elements. Comparison of the numerical results with the analytic solution showed that there is a smallest step size beyond which the error suddenly increases due to rounding errors, which occurs due to the way computers store numbers. A comparison of the computation times showed that the LU-decomposition method is slower because of the additional calculations that are needed. The fastest algorithm was the optimized Gaussian elimination, requiring the least FLOPs. An optimized code which makes use of special properties of a given problem can drastically decrease the computation time and allow one to solve larger systems of linear equations with the same computational power and memory resources.

6 Supplementary Material

All programs and benchmarks calculations can be found in the GIT repository: `https://github.com/tgolubev/PHY905MSU/tree/master/Project1`

References

[1] Morten Hjorth-Jensen: Project1-2016 Computational Physics Spring 2016, `https://github.com/CompPhysics/ComputationalPhysicsMSU/tree/master/doc/Projects/2017`

[2] Trefethen, Lloyd N., and David Bau III. Numerical linear algebra. Vol. 50. Siam, 1997.