A C++ Implementation of the Thomas Algorithm to Solve the One Dimensional Poisson Equation

Curtis Rau

February 16, 2016

Abstract

1 Motivation

The Poisson Equation is a partial differential equation of the form

$$-\frac{\partial^2 u}{\partial x^2} = f(x) \tag{1}$$

which comes up often in physics. One such application is Gauss's Law in differential form. To find a numerical solution to Poisson's Equation we discretize x

$$\begin{aligned} x \to & \quad x_i \\ u(x) \to & \quad u(x_i) = u_i \\ f(x) \to & \quad f(x_i) = f_i \end{aligned} \tag{2}$$

and we utilize the limit definition of the second derivative

$$\frac{\partial^2 u}{\partial x^2} = \lim_{N\to\infty} \frac{u_{i-1} - 2u_i + u_{i+1}}{h^2} \tag{3}$$

where $h = x_{i+1} - x_i$ = (length) / (number of divisions). The discretized Poisson's Equation reads

$$\begin{aligned} \frac{u_{i-1} - 2u_i + u_{i+1}}{h^2} &= -f_i \\ -u_{i-1} + 2u_i - u_{i+1} &= h^2 f_i \end{aligned} \tag{4}$$

As the number of divisions increases the numerical results will converge to the analytical result. We can now write this as a system of linear equations:

$$\begin{aligned} u_1 = & \quad uFirst \\ -u_1 + 2u_2 - u_3 = & \quad h^2 f_2 \\ -u_2 + 2u_3 - u_4 = & \quad h^2 f_3 \\ & \vdots \\ -u_{N-3} + 2u_{N-2} - u_{N-1} = & \quad h^2 f_{N-2} \\ -u_{N-2} + 2u_{N-1} - u_N = & \quad h^2 f_{N-1} \\ u_N = & \quad uLast \end{aligned} \tag{5}$$

The first and last equations are the boundary conditions. There are N equations and N unknowns, so this linear system has only one solution. The language of C++, in which the program was written, has the interesting (read frustrating and unintuitive) property that counting starts with 0. So before we proceed we should make the indicies of the arrays f and u start with 0 so

$$\begin{aligned} f_i &\to f_{i-2} \\ u_i &\to u_{i-1} \end{aligned} \tag{6}$$

The endpoints, u_0 and u_N, are already known (because of boundary conditions) so we can remove the first and last equation from the system. By rearranging the second and second to last equation we can arrive at a suggestive form for the system. In the following equation, blank spots within the matrix represent 0's, and were ommitted so as to highlight the structure of the sparse matrix.

$$\begin{pmatrix} 2 & -1 & & & \\ -1 & 2 & -1 & & \\ & & \ddots & & \\ & & -1 & 2 & -1 \\ & & & -1 & 2 \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_{N-3} \\ u_{N-2} \end{pmatrix} = \begin{pmatrix} h^2 f_0 + u_0 \\ h^2 f_1 \\ \vdots \\ h^2 f_{N-4} \\ h^2 f_{N-3} + u_{N-1} \end{pmatrix} \tag{7}$$

We can append the endpoints to the solution array after the program has run if we so desire.

2 The Thomas Algorithm

To solve equation 12 we use the Thomas Algorithm. A theorm of Linear Algebra states that any matrix can be decomposed into a product of an upper and lower triangular matrix. The Thomas Algorithm is a LU Decomposition algorithm that is equivalent to a Gaussian Elimination algorithm. First consider the general $m \times m$ tridiagonal matrix system.

$$\begin{pmatrix} b0 & c0 & & & \\ a0 & b1 & c1 & & \\ & & \ddots & & \\ & & a_{m-3} & b_{m-2} & c_{m-2} \\ & & & a_{m-2} & b_{m-1} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{m-2} \\ x_{m-1} \end{pmatrix} = \begin{pmatrix} g_0 \\ g_1 \\ \vdots \\ g_{m-2} \\ g_{m-1} \end{pmatrix} \tag{8}$$

The first step is to multiply the first row by a0/b0 and subtract it from the second row. This is known as the forward substitution step. By itterating this step the a's are eliminated, leaving an upper triangular matrix.

$$\begin{aligned} a_i &= 0 \\ b_i &= b_i - \frac{a_{i-1}}{b_{i-1}} c_{i-1} \\ g_i &= g_i - \frac{a_{i-1}}{b_{i-1}} g_{i-1} \end{aligned} \tag{9}$$

At this point the simplified equation, with only an upper triangular matrix takes the form

$$\begin{pmatrix} b0 & c0 & & & \\ & b1 & c1 & & \\ & & \ddots & & \\ & & & b_{m-2} & c_{m-2} \\ & & & & b_{m-1} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{m-2} \\ x_{m-1} \end{pmatrix} = \begin{pmatrix} g_0 \\ g_1 \\ \vdots \\ g_{m-2} \\ g_{m-1} \end{pmatrix} \tag{10}$$

Now for the backwards substitution step. The last row is multiplied by c_{N-3}/b_{N-2} and subtracted from the second to last row, thus eliminating c_{N-3} from the last row. The general backwards substitution procedure is

$$\begin{aligned} c_i &= 0 \\ g_{i-1} &= g_{i-1} - \frac{c_{i-1}}{b_i} g_i \end{aligned} \tag{11}$$

Iterating this process over the remaining c's leaves a diagonal matrix

$$\begin{pmatrix} b0 & & & & \\ & b1 & & & \\ & & \ddots & & \\ & & & b_{m-2} & \\ & & & & b_{m-1} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{m-2} \\ x_{m-1} \end{pmatrix} = \begin{pmatrix} g_0 \\ g_1 \\ \vdots \\ g_{m-2} \\ g_{m-1} \end{pmatrix} \tag{12}$$

All that ramains to be computed is the solution, which is the very straight forward process of

$$x_i = \frac{f_i}{b_i} \tag{13}$$

The function I wrote to perform this algorithim is displayed below. It takes pointers to a, b, c, u, and f – the vectors; and a integer N to tell the function how long the vectors are for the for loops. The indicies of the u vector are shifted up one so that the endpoints can be assigned.

```
void triDiagMatSolve(int N, double* a, double* b, double* c, double* g, double* x)
/* Solves a linear system of the form:
 | b0  c0                    | |  x0  |   |  g0  |
 | a0  b1  c1                | |  x1  |   |  g1  |
 |     a1  b2   c2           | |  x2  |   |  g2  |
 |         **   **   **      | |  **  | = |  **  |
 |            aN-3 bN-2 cN-2 | | xN-2 |   | gN-2 |
 |                 aN-2 bN-1 | | xN-1 |   | gN-1 |
 */
{

    // Forward substitution
    for (int i=1; i<N; i++) {
        b[i] = b[i] - (a[i-1]*c[i-1])/b[i-1];
        g[i] = g[i] - (a[i-1]*g[i-1])/b[i-1];
    }

    x[N-1] = g[N-1] / b[N-1];

    // Backwards substitution
    for (int i = N-2; i >= 0; i--) {
        g[i] = g[i] - c[i]*g[i+1]/b[i+1];
        x[i] = g[i]/b[i];
    }
}
```

Figure 1

3 Faster, Tuned Algorithms

The generality of the Thomas Algorithm is unnecessary for the problem at hand. Using it to solve the Poisson Equation means $a = c = 1$ and $b = -2$. Additionally we are using dirichlet boundary conditions so calculating the f vector can be somewhat simplified. Further simplification can be realized by recognizing the recursive relation for b given in equation (9) can be expressed in explicit form:

$$b_i = \frac{i+2}{i+1} \tag{14}$$

The implementation of this looks like:

```
void poisonSolverDirichletBC1D(int N, double L, double* f, double* v)
{
    // Boundary Conditions
    v[0]   = 0.0;
    v[N-1] = 0.0;

    double h2 = L*L / ((N-1)*(N-1));                          // The step size squared;

    // Forward substitution
    for (int i=1; i<N-2; i++) {
        f[i] = f[i] + f[i-1] * i / (i+1.0);
    }

    v[N-2] = f[N-3] * h2 * (N-2.0) / (N-1.0);

    // Backwards substitution
    for (int i = N-3; i > 0; i--) {
        f[i-1] = f[i-1] + f[i] * (i+1.0) / (i+2.0);
        v[i]   = f[i-1] * h2 * i / (i+1.0);
    }
}
```

Figure 2

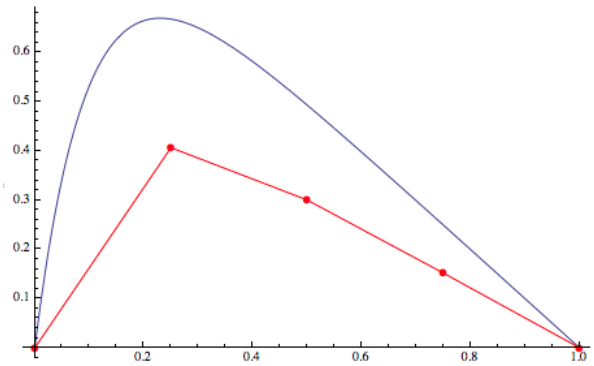
4 Results

The benchmarks for the three solving methods is given by the following table.

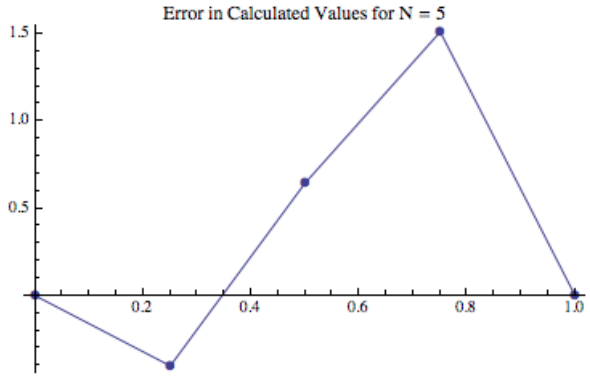
N	Thomas Algorithm Computation Time	Thomas Algorithm Computation Time (Adjusted)	Optimized Algorithm Computation Time
5	$4.7 \cdot 10^{-5}$	$3.5 \cdot 10^{-5}$	$3.3 \cdot 10^{-5}$
10	$3.5 \cdot 10^{-5}$	$3.5 \cdot 10^{-5}$	$3.3 \cdot 10^{-5}$
25	$3.5 \cdot 10^{-5}$	$3.5 \cdot 10^{-5}$	$3.5 \cdot 10^{-5}$
100	$7.3 \cdot 10^{-5}$	$4.1 \cdot 10^{-5}$	$3.7 \cdot 10^{-5}$
1,000	$7.0 \cdot 10^{-5}$	$9.4 \cdot 10^{-5}$	$6.9 \cdot 10^{-5}$
10,000	$4.06 \cdot 10^{-4}$	$6.05 \cdot 10^{-4}$	$3.79 \cdot 10^{-4}$

Table 1

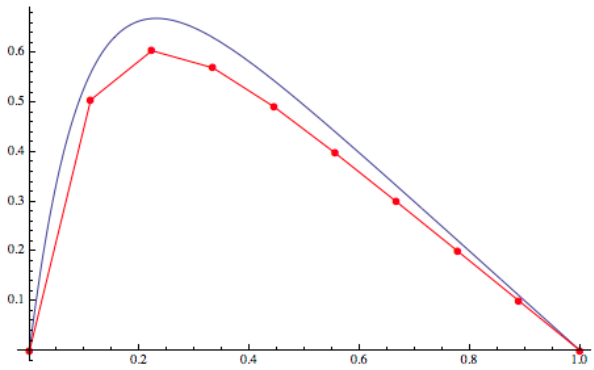
The code was benchmarked on a macbook pro. Apple's Grand Central Dispatch (GCD) makes accurately benchmarking almost impossible. The way this code is written it will execute on one thread. Modern operating systems break up thread execution because there are almost always more threads executing at any given time than there are cores in the processor. This means the execution time as shown above is probably a factor of 10 to 100 times longer than if the thread were to execute continuously. All that can be done on a computer running a modern operating system (without fancy code or dedicated cores) is set an upper limit on the computation time. This also makes it difficult to empirically verify the dependency of computation time on the number of points. Running identical code two times commonly yealds two distinct computation times. The ones shown in the table above represent the smallest time observed.



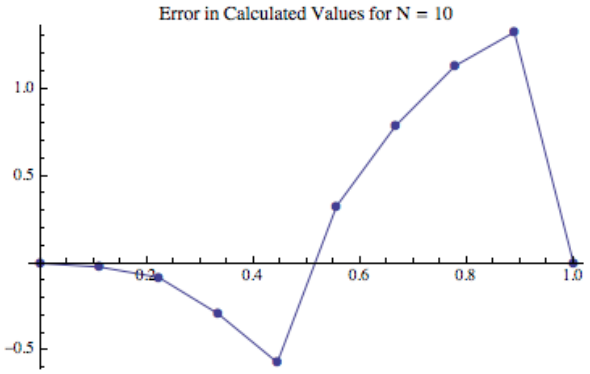
(a) Numerical solution with 5 interior points.



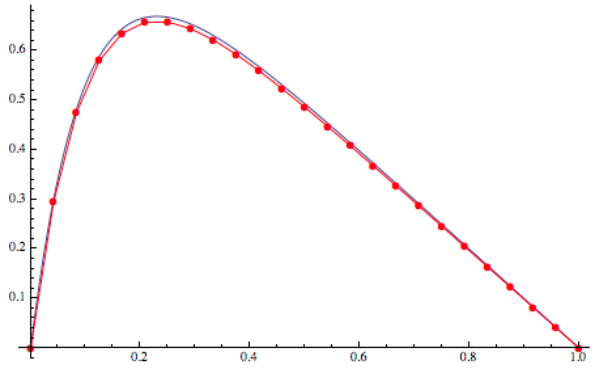
(b) Laguerre Gaussian Modes



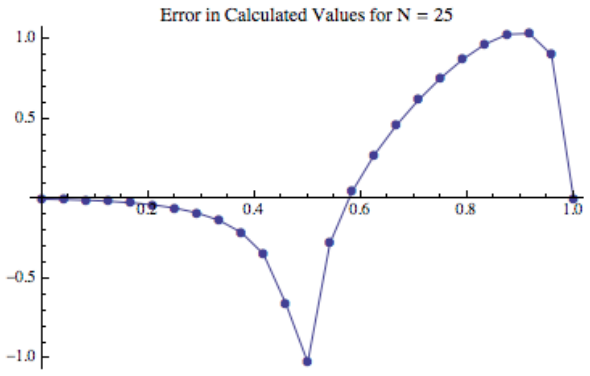
(a) Numerical solution with 10 interior points.



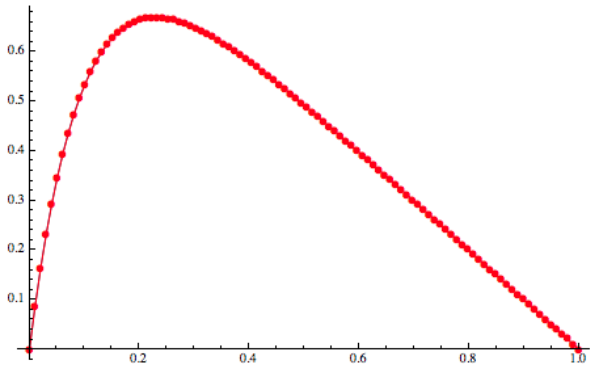
(b) Point by point error in numerical solution with 10 interior points



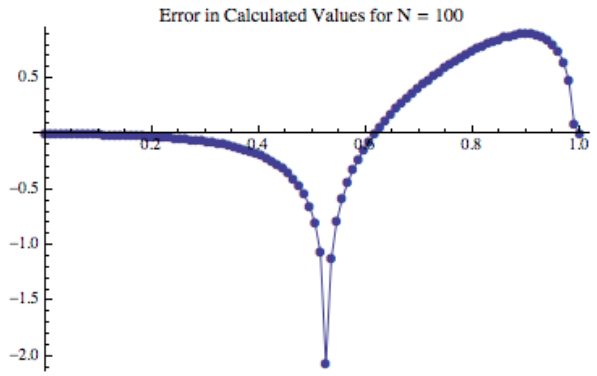
(a) Numerical solution with 25 interior points.



(b) Point by point error in numerical solution with 25 interior points



(a) Numerical solution with 100 interior points.



(b) Point by point error in numerical solution with 100 interior points

Figure 6: These calculated error graphs look pretty suspicious to me. I'm going to take a closer look at them. Namely ther error is clearly going to zero, but errors, indicated by the error graphs, are not going down, rather the shape is converging to something unexpected.