Programming a Linear Algebra Solution to the Poisson Equation

Elizabeth Drueke

February 11, 2016

Abstract

Poisson's equation is vital in the study of the physical world. In every subfield of physics, from mechanics and quantum mechanics to electromagnetism, we are able to model physical systems subject to boundary conditions with this versatile equation. This means that developing computer programs which can quickly and accurately solve this system is of the utmost importance. Here, we develop a code using C++ compiled within the ROOT framework which can create a numerical approximation to the solution of the Poisson equation, specifically subject to the Dirichlet boundary conditions. In the process, we also explore dynamic memory allocation in C++ and the use of classes.

1 Introduction

It is important in physics to develop ways to deal with large quantities of data and to use that data to make close approximations of physical conditions. In particular, we wish to develop computer programs which can both automate the process of numerical approximation and which can complete them in a timely manner. One example of an equation which we will often need to solve when investigating physical situations is the Poisson equation, given by

$$-u''(x) = f(x), \tag{1.1}$$

subject to the Dirichlet boundary conditions,

$$u(0) = u(1) = 0. \tag{1.2}$$

In this report, we will study various numerical approximations of solutions to the Poisson equation in great detail, particularly focusing on using linear algebra and the solution of linear systems to determine a close approximation to the solution $u(x)$. We pay special attention to the number of floating point operations (FLOPS) and the time required for various approximation methods.

2 Theory

The mathematics behind the solution to the Poisson equation presented here is mathematically rich in approximations. From Eq. 1.1, we can see that we are required to compute the function $u\left(x\right)$ from its second derivative. To do this, we note that we can always approximate the first derivative as

$$f'\left(x\right) \approx \frac{f\left(x+h\right)-f\left(x-h\right)}{2h}, \tag{2.1}$$

for $h << 1$ because the derivative is the slope of the line tangent to f at that point. This then implies that

$$\begin{array}{rcc} f''\left(x\right) & \approx & \frac{f'(x+h)-f'(x-h)}{2h} \\ & \approx & \frac{\frac{f(x+h+h)-f(x+h-h)}{2h}-\frac{f(x-h+h)-f(x-h-h)}{2h}}{2h} \\ & \approx & \frac{\frac{f(x+2h)-f(x)}{2h}-\frac{f(x)-f(x-2h)}{2h}}{2h} \\ & \approx & \frac{f(x+2h)-2f(x)+f(x-2h)}{(2h)^2}. \end{array}$$

Letting $2h \rightarrow h$, we then have

$$f''\left(x\right) \approx \frac{f\left(x+h\right)-2f\left(x\right)+f\left(x-h\right)}{h^2}. \tag{2.2}$$

Eq. 2.2 will serve as our approximation to the second derivative throughout the remainder of this discussion.

Now, we have an expression which lends itself to the use of vectors. Letting

$$h = \frac{1}{n+1}$$

be the step size for some number of steps $n \in \mathbb{N}$, we see that we can treat this as a partition of the interval $[0,1]$, on which the Dirichlet conditions are valid. Thus, we define each x_i in the partition as

$$x_i = ih, i = 0, \ldots, n+1.$$

From these x_i we can create a vector $\mathbf{x}$[1]

$$\mathbf{x} = \left(0, \frac{1}{n+1}, \frac{2}{n+1}, \ldots, \frac{n}{n+1}, 1\right)^T.$$

Then, let $\mathbf{b}$ be a vector of the function $f\left(x\right)$ evaluated at the points in $\mathbf{x}$. That is,

$$\begin{array}{rcc} \mathbf{b} & = & \left(f\left(0\right), f\left(\frac{1}{n+1}\right), f\left(\frac{2}{n+1}\right), \ldots, f\left(\frac{n}{n+1}\right), f\left(1\right)\right)^T \\ & = & \left(0, f\left(\frac{1}{n+1}\right), f\left(\frac{2}{n+1}\right), \ldots, f\left(\frac{n}{n+1}\right), 0\right)^T. \end{array}$$

Now, we see that Eq. 2.2 lends itself nicely to the introduction of a linear algebra problem. In particular, we can define an $(n+1) \times (n+1)$ matrix A such that

[1] Note that throughout we indicate vectors as bold, lowercase letters (eg. $\mathbf{x}$), and matrices as uppercase letters (eg. A).

$$A=\frac{1}{h^2}\begin{pmatrix} 2 & -1 & 0 & \cdots & \cdots & 0 \\ -1 & 2 & -1 & 0 & \cdots & 0 \\ 0 & -1 & 2 & -1 & 0 & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \vdots & \vdots & \ddots & -1 & 2 \end{pmatrix} \tag{2.3}$$

Then, we have that the Poisson Equation given by Eq. 1.1 can be approximated as

$$A\mathbf{v}=\mathbf{b} \tag{2.4}$$

for some $\mathbf{v}$ which represents $u(x)$ at various values of x. And so we have a system of $n+1$ equations in $n+1$ unknowns. Once we have solved for the vector $\mathbf{v}$, we can use a plotting tool to plot the points and extrapolate a fit to the function.

In general, there are two main ways in which we might solve Eq. 2.4, known as Gaussian elimination and LU Decomposition. We will discuss these in general in Section 2.1 and Section 2.2, respectively, before discussing how we developed the computer algorithms for these methods specifically for our matrix A given by Eq. 2.3 in Section 3.

2.1 Gaussian Elimination

Gaussian elimination is the method of solving linear systems typically taught in an introductory linear algebra class. This method involves adding multiples of rows of the matrix to other rows in order to eliminate (or set to zero) off-diagonal elements. In order to determine a solution, any row operation performed on the matrix A must be performed on our solution vector $\mathbf{b}$[2] in parallel. In most situations, it is necessary to employ both a forward and backward Gaussian elimination method in order to solve a full system. To demonstrate this method, we look at a 3 × 3 example. The algorithm used within the code will be more explicitly discussed in Section 3.

Suppose we have a matrix A of the form

$$A=\begin{pmatrix} a_{11} & a_{12} & a_{12} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \tag{2.5}$$

and we wish to reduce it to row echelon form. We might do this using Gaussian elimination. To begin, we would use forward elimination to set the a_{i1} components to zero for $i \neq 1$. Letting R_i denote the i^{th} row, we notice that letting

$$\begin{aligned} R_2 &= R_2-\frac{a_{21}}{a_{11}}R_1 \\ R_3 &= R_3-\frac{a_{31}}{a_{11}}R_1 \end{aligned} \tag{2.6}$$

should yield the desired outcome. That is, we have

[2] Following the notation in Eq. 2.4.

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \rightarrow \begin{pmatrix} a_{11} & a_{12} & a_{12} \\ 0 & a_{22} - \frac{a_{21}a_{12}}{a_{11}} & a_{23} - \frac{a_{21}a_{13}}{a_{11}} \\ 0 & a_{32} - \frac{a_{31}a_{12}}{a_{11}} & a_{33} - \frac{a_{31}a_{13}}{a_{11}} \end{pmatrix}.$$

Continuing in this manner, we will eventually have a matrix of the form

$$A\prime = \begin{pmatrix} a_{11}\prime & a_{12}\prime & a_{13}\prime \\ 0 & a_{22}\prime & a_{23}\prime \\ 0 & 0 & a_{33}\prime \end{pmatrix}, \tag{2.7}$$

where the $\prime$ indicates that the element has been modified from its original definition in Eq. 2.5.

At this point, we may begin the backward Gaussian elimination process. In the forward process, we were able to set all of the $a_{ij} = 0$ for $i > j$. In the backward process, we wish to do the same for the a_{ij} with $i < j$. In the end, we should have a pure diagonal matrix.

To begin the backward process, we note that we can set the a_{i3} elements to zero by similar calculations as those performed in Eq. 2.6. In particular, we can let

$$\begin{aligned} R_2 &= R_2 - \frac{a_{23}}{a_{33}} R_3 \\ R_1 &= R_1 - \frac{a_{13}}{a_{33}} R_3 \end{aligned} \tag{2.8}$$

Doing this, we see

$$\begin{pmatrix} a_{11}\prime & a_{12}\prime & a_{13}\prime \\ 0 & a_{22}\prime & a_{23}\prime \\ 0 & 0 & a_{33}\prime \end{pmatrix} \rightarrow \begin{pmatrix} a_{11}\prime & a_{12}\prime & 0 \\ 0 & a_{22}\prime & 0 \\ 0 & 0 & a_{33}\prime \end{pmatrix}.$$

Continuing in this manner, we can see that we will eventually come across a pure diagonal matrix. From here, having performed the row operations on the solution vector $\mathbf{b}$ as well as the matrix A, we can find our solution $\mathbf{v}$ by noting that

$$v_i = \frac{\widetilde{b_i}}{\widetilde{a_{ii}}}, \tag{2.9}$$

where the $\sim$ indicates that this is the component of the fully Gaussian eliminated matrix/vector. This procedure is easily generalized to work for any $n \times n$ matrix.

One downside to this method of solving the system of linear equations is that whatever row operations are performed on A in Eq. 2.4 must also be performed on $\mathbf{b}$. This means that the process must be repeated every time the solution vector is changed. This restriction can be a serious time constraint on any program written to implement Gaussian elimination in order to solve systems of linear equations. The time limitations of such an algorithm are discussed in Section 3.

2.2 LU Decomposition

The second major linear system solving method is what is known as LU decomposition. In this procedure, we decompose our A matrix into a lower-triangular L and an upper-triangular U of the form

$$\begin{array}{cccc} \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} & = & \begin{pmatrix} 1 & 0 & 0 \\ l_{21} & 1 & 0 \\ l_{31} & l_{32} & 1 \end{pmatrix} & \begin{pmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{pmatrix} \\ A & = & L & U \end{array} \tag{2.10}$$

In contrast with the Gaussian elimination method, this method does not need to be repeated for every choice of solution vector $\mathbf{b}$. Instead, once L and U have been computed, they can be used to determine the solution $\mathbf{v}$ for any solution vector $\mathbf{b}$.

As with the Gaussian elimination method, we present an example as an illustration of how the LU decomposition method works. Assume we have some

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix}. \tag{2.11}$$

Based on the multiplication as shown in Eq. 2.10, we can see that we can directly solve for the l_{ij} and u_{ij}. In particular, in order, we see[3]

$$\begin{aligned}
a_{11} &= \mathbf{u_{11}} && \Rightarrow \\
a_{21} &= \mathbf{l_{21}} u_{11} && \Rightarrow \\
a_{31} &= \mathbf{l_{31}} u_{11} && \Rightarrow \\
a_{41} &= \mathbf{l_{41}} u_{11} && \Rightarrow \\
a_{12} &= \mathbf{u_{12}} && \Rightarrow \\
a_{22} &= \mathbf{l_{21}} u_{12} + u_{22} && \Rightarrow \\
a_{32} &= l_{31} u_{12} + \mathbf{l_{32}} u_{22} && \Rightarrow \\
a_{42} &= l_{41} u_{12} + \mathbf{l_{42}} u_{22} && \Rightarrow \\
a_{13} &= \mathbf{u_{13}} && \Rightarrow \\
a_{23} &= l_{21} u_{13} + \mathbf{u_{23}} && \Rightarrow \\
a_{33} &= l_{31} u_{13} + l_{32} u_{23} + \mathbf{u_{33}} && \Rightarrow \\
a_{43} &= l_{42} u_{23} + \mathbf{l_{43}} u_{33} + l_{41} u_{13} && \Rightarrow \\
a_{14} &= \mathbf{u_{14}} && \Rightarrow \\
a_{24} &= l_{21} u_{14} + \mathbf{u_{24}} && \Rightarrow \\
a_{34} &= l_{31} u_{14} + l_{32} u_{24} + \mathbf{u_{34}} && \Rightarrow \\
a_{44} &= l_{41} u_{14} + l_{42} u_{24} + l_{43} u_{34} + \mathbf{u_{44}}.
\end{aligned} \tag{2.12}$$

This result generalizes to any $n \times n$ matrix. In particular, an algorithm can be developed which will always give the L and U matrix from known values. This algorithm, known as Doolittle's Algorithm **need citation**, proceeds as follows:

- Starting with column $j = 1$, compute the first element by

[3] Here, the boldface text indicates the unknown variable in each equation.

$$u_{1j} = a_{1j} \tag{2.13}$$

- For $i = 2, \ldots, j-1$, compute

$$u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj} \tag{2.14}$$

- Calculate the diagonal element as

$$u_{jj} = a_{jj} - \sum_{k-1}^{j-1} l_{jk} u_{kj} \tag{2.15}$$

- Calculate the l_{ij} for $i > j$ as

$$l_{ij} = \frac{1}{u_{jj}} \left(a_{ij} - \sum_{k=1}^{j-1} l_{jk} u_{kj} \right) \tag{2.16}$$

- Repeat with columns $j = 2, 3, \ldots, n$.

Of course, this algorithm does not actually compute the solution to the linear system presented in Eq. 2.4. To compute the solution $\mathbf{v}$, we must invoke another algorithm. To begin, we note that

$$A\mathbf{v} = \mathbf{b} \Rightarrow LU\mathbf{v} = \mathbf{b}. \tag{2.17}$$

This implies that we should be able to define an intermediate vector $\mathbf{y} = U\mathbf{v}$ which can be easily computed. In the four-dimensional example, we would have the set of equations

$$\begin{aligned} u_{44}v_4 &= y_4 \\ u_{33}v_3 + u_{34}v_4 &= y_3 \\ u_{22}v_2 + u_{23}v_3 + u_{24}v_4 &= y_2 \\ u_{11}v_1 + u_{12}v_2 + u_{13}v_3 + u_{14}v_4 &= y_1 \end{aligned}$$

and

$$\begin{aligned} y_1 &= b_1 \\ l_{21}y_1 + y_2 &= b_2 \\ l_{31}y_1 + l_{32}y_2 + y_3 &= b_3 \\ l_{41}y_1 + l_{42}y_2 + l_{43}y_3 + y_4 &= b_4. \end{aligned}$$

Thus, there must be both an algorithm to decompose the matrix A into an upper-triangular U and a lower-triangular L and another algorithm to actually compute the solution to Eq. 2.4 from that L and U.

One benefit to using LU decomposition over Gaussian elimination, aside from the fact that it will take less time than the standard Gaussian elimination method,[4] is that it is very natural to compute the determinant of the matrix once it has been LU-decomposed. This is because we have

$$A = LU \Rightarrow \det(A) = \det(LU) = \det(L)\det(U),$$

and the determinant of either an upper-triangular or lower-triangular matrix is simply the product of its diagonal elements. Because L has 1's along its diagonal, we can see that

$$\det(A) = \prod_{i=1}^{n} u_{ii}. \tag{2.18}$$

3 The Algorithm

The meat of this project is the development of an algorithm which can compute a fair approximation for $u(x)$ in Eq. 1.1. In order to check our solutions, we take our function f to be given by $f(x) = 100e^{-10x}$. We work with the linear algebra solutions described in Section 2. To do this, we first define our matrix A as in Eq. 2.3 and a vector of solutions $\mathbf{b}$ as in Eq. 2.4. Noting that, in fact, we know the solution for $u(x)$ at $x = 0$ and $x = 1$[5], we do not include either the x_0 or x_{n+1}[6] terms. However, when we perform the fit to our plot at the end, these boundary conditions will be included for completeness. To declare these objects, we create classes `themat` and `thevec` for matrices and vectors handled with dynamic memory.

The classes and function definitions come in documents `classes.h` and `classes.C`. Every object of the `thevec` class has associated with it an integer `sz` which is the number of rows of the vector, and a pointer to a dynamic memory array `point` in which we can place the values of the components of the vector. This class comes equipped with several constructors, a destructor (which de-allocates any memory in use by the object), several overloaded operators (addition of two vectors, subtraction of two vectors, multiplication of two vectors (in the form of a dot product), etc.), a component retrieval function (`operator[]`) which returns the value of the vector at a particular index, and a printing member function.

The setup of the `themat` class is very similar to that of the `thevec` class. The `sz` component here is the n which defines the matrix (ie. creates an $n \times n$ matrix) and the `point` is a double pointer. This class also comes with multiple constructors, several overloaded operators (including matrix-matrix and matrix-vector multiplication), a component-retrieval function, and a print function. For ease of use, `thevec` was declared as a friend function of `themat`.

Once the vector and matrix objects are defined, it is just a matter of solving Eq. 2.4 using the matrix and vector specific to the problem. However, because we are working with a specific kind of matrix, there are several things we can do to reduce the computing power needed to solve the problem. The first is noting that we do not have to proceed with the full LU-decomposition in order to decompose our special matrix. Looking at a 4×4 example, we see a pattern forming.

[4] The specific argument to this statement is presented in Section 3.1.

[5] These values are given by the Dirichlet boundary conditions.

[6] In the notation of $x_i = ih, i = 0, \ldots, n+1$ where $h = \frac{1}{n+1}$.

$$\begin{array}{cccc}
\begin{pmatrix} 1 & 0 & 0 & 0 \\ -\frac{1}{2} & 1 & 0 & 0 \\ 0 & -\frac{2}{3} & 1 & 0 \\ 0 & 0 & -\frac{3}{4} & 1 \end{pmatrix} &
\begin{pmatrix} 2 & -1 & 0 & 0 \\ 0 & \frac{3}{2} & -1 & 0 \\ 0 & 0 & \frac{4}{3} & -1 \\ 0 & 0 & 0 & \frac{5}{4} \end{pmatrix} & = &
\begin{pmatrix} 2 & -1 & 0 & 0 \\ -1 & 2 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 2 \end{pmatrix} \\
L & U & = & A.
\end{array} \tag{3.1}$$

In general, then, we might suppose that the matrix elements l_{ij} and u_{ij} of L and U will take the form

$$\begin{array}{c} l_{ii} = 1, \quad l_{i,i-1} = -\frac{i-1}{i-2}, \\ u_{ii} = \frac{i+1}{i}, \quad u_{i,i+1} = -1 \end{array} \tag{3.2}$$

and all other l_{ij} and u_{ij} are 0. Noticing this means that we are required to perform far fewer float operations in order to decompose the matrix.

From there, it becomes simpler to solve the linear system as well. Referring back to the 4×4 example, we can now solve for our intermediate $\mathbf{y}$ and then our solution $\mathbf{v}$ with the following equations:

$$\begin{array}{rcr}
y_1 = b_1 & & 2v_1 - v_2 = y_1 \\
-\frac{1}{2}y_1 + y_2 = b_2 & & \frac{3}{2}v_2 - v_3 = y_2 \\
-\frac{2}{3}y_2 + y_3 = b_3 & \rightarrow & \frac{4}{3}v_3 - v_4 = y_3 \\
-\frac{3}{4}y_3 + y_4 = b_4 & & \frac{5}{4}v_4 - v_5 = y_4 \\
-\frac{4}{5}y_4 + y_5 = b_5 & & \frac{6}{5}v_5 = y_5
\end{array} \tag{3.3}$$

In fact, with the matrix as specific as it is, it isn't necessary even to pass the L and U matrices to the solving algorithm in order to solve the system. Instead, one possible version of the algorithm is:

```
y[0]=b[0];
for(int i=1;i<n;i++){
   y[i]=b[i]+(i/(i+1))*y[i-1];
}
v[n-1] = n*y[n-1]/(n+1);
for(int i=n-2;i>-1;i--){
   v[i]=((i+1)/(i+2))*(y[i]+v[i+1]);
}
```

However, the main goal in rewriting the algorithm specific to the matrix required to solve the Poisson equation is to reduce the number of FLOPs, and writing the program in this way will actually force the computer to perform unnecessary floating point operations because the fractional factors will have already been computed in the LU decomposition step. It is for this reason that we create an essentially shorter version of the LU decomposition solving function, which works with the already-computed elements of the L and U matrices. In particular, our algorithm first solves for the intermediate vector $\mathbf{y}$ as

```
y[0]=b[0];
for(int i=0;i<n;i++){
   y[i]=b[i]-L[i][i-1]*y[i-1];
}
```

and then solves for the solution to the problem $\mathbf{v}$ as

```
v[n-1]=y[n-1]/U[n-1][n-1];
for(int i=n-2;i>-1;i--){
   v[i]=(y[i]+v[i+1])/U[i][i];
}
```

In this way, we are able to reasonably minimize the number of FLOPs required for the calculation. A full analysis of the expected and observed time dependences of these algorithms is presented in Section 3.1 and Section 4.

3.1 Time Dependences

As mentioned in Section 3, we wish to develop an algorithm which is not only able to solve numerically the Poisson equation, but which can also do so within a reasonable time frame and with limited computer memory usage. **need more here - what are the expected time dependences?**.

4 Results and Benchmarks

Before entering a discussion of the full results of the program developed, it is important to validate the code works properly. In particular, it is reasonable to assume that, if all three algorithms written predict the same solution vector, then the solution is correct. Such an analysis is performed in the `benchmarks.C` file. In this file, a matrix of the special form is created with varying dimensions between 1 × 1 and 50 × 50 and the resulting solution vector $\mathbf{v}$ is computed using the full Gaussian elimination method (`gauss_elim()`), the full LU decomposition solver (`LU_decomp()` together with `LU_decomp_solver()`), and the special LU decomposition solver which will work only on matrices of our specific form (`LU_decomp_special()` together with `LU_decomp_solver_special()`). The results of these tests are found in `benchmarks.txt`. It is shown here that, for a specific A, all three algorithms return precisely the same solution vector $\mathbf{v}$. In addition, the L and U matrices created by the different LU Decomposition algorithms are also identical and, when multiplied together, yield our original matrix A.

We now discuss the results of the analysis. Because of the simplicity of our test function, $f(x) = 100e^{-10x}$, we can directly compute $u(x)$ either by hand or with any computing software. We chose to use WolframAlpha **need citation**. We find that, subject to the Dirichlet boundary conditions, the solution is given by

$$u(x) = \frac{x}{e^{10}} - x - e^{-10x} + 1 \tag{4.1}$$

The results for various numbers of steps are plotted in **need figure**. From the image, it appears that increasing the number of steps improves the validity of the result. In particular, we are able to compute the relative error ϵ_i between the expected value u_{exp} of the function at point x_i and the result computed by our special LU decomposition algorithm u_{comp} as

$$\epsilon_i = \log_{10}\left(|\frac{u_{comp} - u_{exp}}{u_{exp}}|\right). \tag{4.2}$$

The maximum value of ϵ_i for each step value investigated is listed in Table **need table (and maybe plot?)**. This confirms our suspicions that the error decreases with increasing number of steps.
time required

5 Conclusions

Something about how the class isn't complete actually. Errors.