Solving Poisson's Equation Using the Finite Difference Method

Zachary Bahr* and Jacob LeGrand†
Missouri University of Science and Technology
CS 5201 - Introduction to Object-Oriented Numerical Modeling
(Dated: May 13, 2019)

Abstract: Poisson's equation is a second order partial differential equation applicable to electrostatics and theoretical physics that is difficult to solve analytically. In this report, we take a numerical approach via the finite difference method (written in C++) to approximate the solution to the equation. The finite difference method allows us to turn the problem into a linear system of equations. To solve these systems, we employ Gaussian elimination and Cholesky decomposition and compare the pros and cons of each approach.

I. The Problem

Partial differential equations contain a multivariate function in relation to one or more of its partial derivatives. When introducing functions of more than one variable, the complexity of the differential equation increases drastically, as we cannot use the same analytical approaches that are applied to equations containing functions of one variable, also known as ordinary differential equations.

Poisson's equation is a second order partial differential equation, meaning that it contains partial second derivatives of the multivariate function. It is defined in terms of the Laplace operator (Δ), a multivariate function (ϕ), and some forcing function (f) as follows:

$$\Delta\phi = f \tag{1}$$

In the two dimensional Cartesian system, the Laplace operator is defined as the sum of the second partial derivatives of the variables of its associated function. Simplifying (1) yields:

$$\frac{\partial^2\phi}{\partial x^2} + \frac{\partial^2\phi}{\partial y^2} = f \tag{2}$$

In order to solve this equation for ϕ, we define a known boundary around the region of ϕ that we are approximating. We do this to provide the finite difference method a base from which to work. Without such a boundary defined, the finite difference method would not be effective. The boundary function is defined for some $a < b$ as follows:

$$b(x,y) = \begin{cases} \phi(x,y) = b_1, & \text{if } y = a \text{ and } a < x < b \\ \phi(x,y) = b_2, & \text{if } x = a \text{ and } a < y < b \\ \phi(x,y) = b_3, & \text{if } y = b \text{ and } a < x < b \\ \phi(x,y) = b_4, & \text{if } x = b \text{ and } a < y < b \end{cases} \tag{3}$$

Utilizing a boundary function makes this a `Dirichlet problem` [1].

* zjb998@mst.edu
† jbl7d8@mst.edu

II. Mathematical Approach

The Finite Difference Method

In the field of numerical methods, it is possible to approximate the derivative of a function at a specific point by subtracting the function value at that point from the function value of a slightly larger point and dividing by the difference of the points, like so:

$$\frac{d\phi}{dx} \approx \frac{\phi(x+h) - \phi(x)}{h} \tag{4}$$

This is what is known as a `finite difference quotient`. This concept can be extended for higher order derivatives and multivariate functions as follows, where ϕ is a function in terms of x and y that is four times continuously differentiable on its domain and ϵ is the error term for some ξ in the domain:

$$\begin{aligned} \frac{\partial^2\phi}{\partial x^2} &= \frac{\phi(x+h,y) - 2\phi(x,y) + \phi(x-h,y)}{h^2} + \epsilon \\ \epsilon &= \frac{-h^2}{12} * \frac{\partial^4\phi}{\partial x^4}(\xi, y) \end{aligned} \tag{5}$$

Using (5), we can express (2) in terms of numerical approximations. For simplicity sake, we will discard the error terms and use subscripts to denote neighboring points.

$$\begin{aligned} f(x_j, y_k) = &\frac{\phi(x_{j+1}, y_k) - 2\phi(x_j, y_k) + \phi(x_{j-1}, y_k)}{h^2} \\ &+ \frac{\phi(x_j, y_{k+1}) - 2\phi(x_j, y_k) + \phi(x_j, y_{k-1})}{h^2} \end{aligned} \tag{6}$$

Simplifying (6) yields an equation for the value of the original function ϕ at points (x, y) in relation to the value of the function at surrounding points. ϕ will be defined in terms of the step size h to signify the closeness of points. A lower h means that points are closer together.

$$\begin{aligned} \phi_h(x_j, y_k) = &\frac{1}{4}\phi_h(x_{j-1}, y_k) + \frac{1}{4}\phi_h(x_{j+1}, y_k) \\ &+ \frac{1}{4}\phi_h(x_j, y_{k-1}) + \frac{1}{4}\phi_h(x_j, y_{k+1}) \\ &- \frac{h^2}{4} f(x_j, y_k) \end{aligned} \tag{7}$$

Essentially, we have created a "mesh" that overlays the region of the function ϕ inside the boundary. A point within the boundary is defined in terms of points directly neighboring it at a distance defined by the step size; the step size is defined as the upper bound value divided by the mesh length value N. Therefore, equation (7) represents a linear system of $(N-1)^2$ equations and $(N-1)^2$ unknowns. With this information, we have the ability to establish a matrix-vector pair in the form $Ax = b$ to find the values of ϕ evaluated at $(N-1)^2$ x-y pairs within the boundary. (Note: we are given that $u_{xx}(x,y)+u_{yy}) = 0$, therefore $f(x,y) = 0$, so the $-\frac{h^2}{4}f(x_j,y_k)$ from (7) is ignored when creating this linear system.) There are numerous approaches to solving such linear systems, which will be described in the next section [1].

III. C++ Implementation

In order to facilitate efficient linear system solving, a matrix-solver library was created to optimize matrix generation, operations, and storage. We implemented several linear system solving algorithms designed to take advantage of these performance enhancements. For this report we will be discussing the positives and negatives of `Gaussian elimination` and `Cholesky decomposition` when solving Poisson's equation using different mesh lengths (N). Further details about specific implementation decisions are provided below.

A. Matrix and Vector Generation

Our approach to coding this solution begins with the generation of a coefficient matrix A and a vector of constants b directly from the system of linear equations as specified in (7).

To generate the coefficient matrix, a pattern was found among all coefficient matrices for all values of N. Every coefficient matrix has 1's along the main diagonal, as well as $-\frac{1}{4}$ on two other diagonals, a distance of $(N-1)$ from the main diagonal, in either direction. Finally, the matrix is populated on either side of the main diagonal with an alternating pattern of $-\frac{1}{4}$ and 0, where a zero is present every $(N-1)$ rows. All other entries of the matrix are 0. An annotated example matrix for an N of 4 is shown in Fig. 1. This matrix is also symmetric, and is created using a simple C++ function that constructs a `Symmetric_Matrix` object and populates it according to the patterns stated above. To generate the constant vector b, we implemented a function using the "pointer to function template" method of callbacks. The four boundary functions (denoted as b_1, b_2, b_3, and b_4 in equation (3)) are defined in our driver file and then passed via function pointers to a function that then generates the b vector. This function iterates through all $(N-1)^2$ equations, and for each one calls a `callback` function. This `callback` function is templated on a `boundary_func`

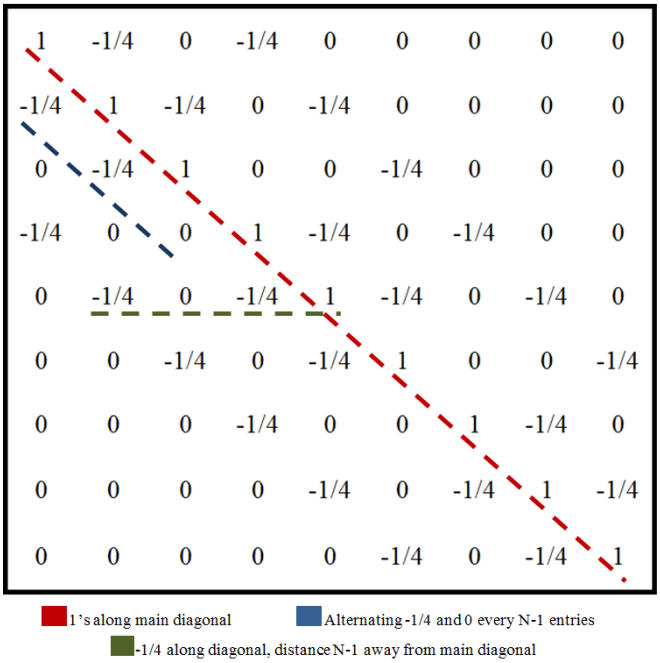

FIG. 1. A coefficient matrix for an N of 4.

function which it calls for each of the right-hand side ϕ's whose x or y value is equal to the lower or upper boundary and sums up the values. In the case of this problem, the upper and lower boundaries were given as π and 0, respectively, and the boundary equations were given as follows:

$$\phi(x,0) = sin(x) \qquad \phi(x,\pi) = 0 \qquad (0 < x < \pi)$$
$$\phi(0,y) = sin(y) \qquad \phi(\pi,y) = 0 \qquad (0 < y < \pi)$$

B. Solution Approximation

As previously stated, we used both Gaussian elimination and `Cholesky decomposition` to solve this problem. Our `Gaussian elimination` approach involved performing row operations on an augmented matrix of our coefficient matrix A and our constant vector b. This augmented matrix is represented by an instance of our `General_Matrix` class and an instance of our `Vector` class. The row operations consisted of iterating down the coefficient matrix along the main diagonal. In this method, the elements along the main diagonal are referred to as pivot elements. For each pivot element, a multiple of its row is added to all rows below it in order to make all elements below the pivot equal to 0 [2]. This was achieved by performing simple vector-vector addition and vector-constant multiplication, using the overloaded `+operator` and `*operator` defined in our `Vector` class. Once all these row operations are performed, the coefficient matrix is now in upper triangular form, and our solution vector x can be found through back substitution, using the same vector addition and multiplication

[2]. For this problem, we chose not to implement scaled partial pivoting; an explanation is provided in Section VI.

Our `Cholesky decomposition` approach involved decomposing our coefficient matrix A (initially stored as an instance of the `Symmetric_Matrix` class) into a lower triangular matrix L, represented by our `L_Triangle_Matrix` class. The equations for generating L are given below [3].

$$L_{ki} = \frac{A_{ki} - \sum_{j=1}^{i-1} L_{ij}L_{kj}}{L_{ii}}$$
$$L_{kk} = \sqrt{A_{kk} - \sum_{j=1}^{k-1} L_{kj}^2} \tag{8}$$

Once this lower triangular matrix is created, its transpose L^T (represented using our `U_Triangle_Matrix` class) can be used in the equation $L^T y = b$, where the solution vector y can be found by forward substitution. This vector y can then be used in the equation $Lx = y$, where the solution vector x can be found by back substitution. This vector x is the solution to our problem [3].

IV. Solution Comparison

Both `Gaussian elimination` and `Cholesky decomposition` are stable methods for solving a linear system with this type of coefficient matrix (details in Section VI); as such, they produced identical solutions for $N < 97$. Numerical approximations of Poisson's equation on the boundary $[0, \pi]$ are plotted in relation to the actual solution in Figures 2-4. As we can see,

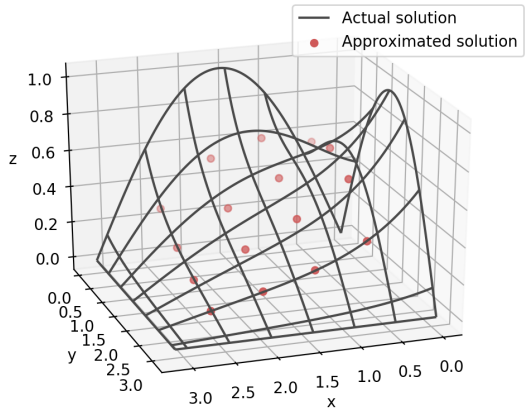

FIG. 2. Numerical approximation of the Poisson surface curve with $N = 5$

with higher mesh lengths we achieve a smaller step size, allowing an increased number of points to be generated within the boundary. More points in the boundary

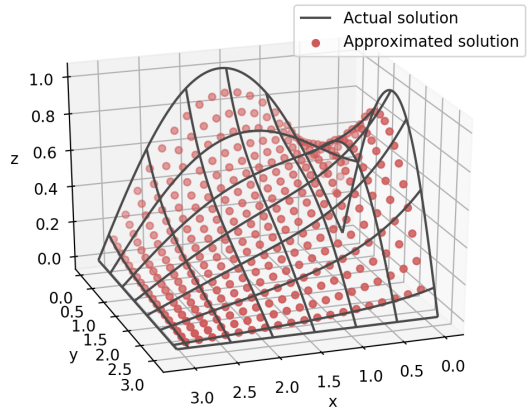

FIG. 3. Numerical approximation of the Poisson surface curve with $N = 20$

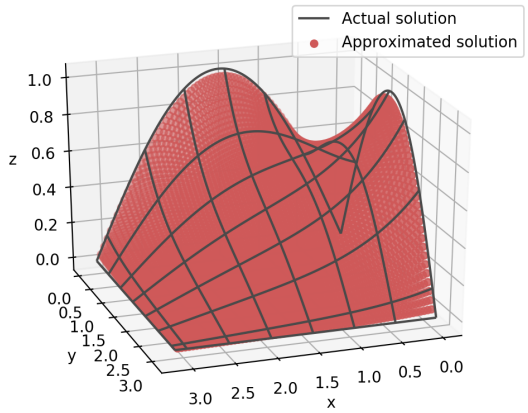

FIG. 4. Numerical approximation of the Poisson surface curve with $N = 70$

allow us to improve our approximate of the curve. The following sub-sections will delve further into performance comparisons of `Gaussian elimination` and `Cholesky decomposition`.

A. Gaussian vs. Cholesky: Time

Both algorithms are of $O(n^3)$ time complexity. When comparing the runtimes for a variety of mesh lengths, we can see (in Fig. 5) that our implementation of `Gaussian elimination` is running more efficiently than `Cholesky decomposition`. LU decomposition algorithms, of which `Cholesky decomposition` is a subset, require the creation of lower and upper triangular matrices followed by two rounds of substitution in order to solve a given linear system. In contrast, `Gaussian elimination` does not require auxiliary matrices to operate, but does tend to require more numerical operations than `Cholesky decomposition`. We propose that the creation of lower and upper triangular matrices could be optimized in our code to streamline the decompo-

sition process; an explanation is provided in Section VI. One advantage that `Cholesky decomposition` holds

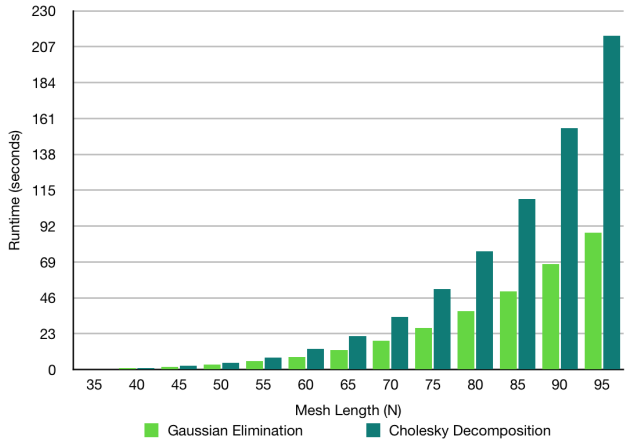

FIG. 5. The effect of mesh length on program runtime

over `Gaussian elimination` is its ability to efficiently recompute a solution when the forcing function or boundary function is changed. In these cases, only the b in $Ax = b$ is affected by such a change, so there is no need to decompose the matrix again. A two-step substitution would result in a solution to the new equation. In this regard, `Cholesky decomposition` vastly outperforms `Gaussian elimination`.

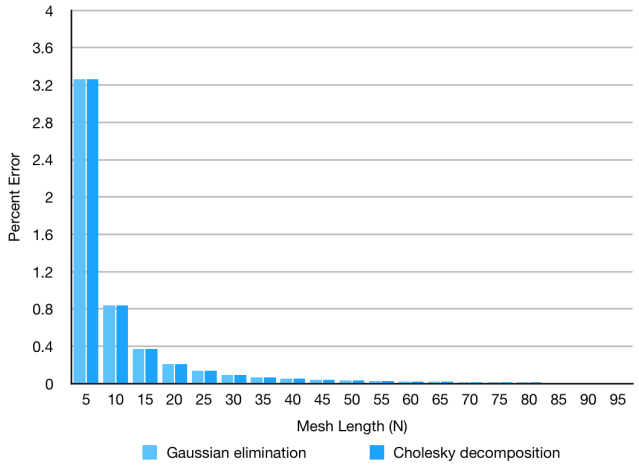

FIG. 6. The effect of mesh length on percent error

B. Gaussian vs. Cholesky: Error

As previously stated, both `Gaussian elimination` and `Cholesky decomposition` produced identical solutions for $N < 97$. As such, they produced identical error. In order to calculate the error between the approximation and the actual solution: we utilized the following norm:

$$\epsilon = \sqrt{h^2 * \sum(\phi_{approx}(i,j) - \phi_{exact}(i,j))^2} \qquad (9)$$

A graph detailing the effect of mesh length on relative error is shown in Fig. 6. For large mesh lengths, the percent error decreases and our approximate solution more closely resembles that of the actual solution.

For mesh lengths greater than 96, `Gaussian elimination` does not produce a valid solution. This is due to the amount of row operations needed for a matrix of size 96^2. At such a size, these row operations result in data that is so large that it exceeds the storage capacity of C++, more specifically of a `long double`.

V. Conclusion

Overall, `Gaussian elimination` and `Cholesky decomposition` are both useful tools for solving `Poisson's equation Dirichlet problems`. Using a custom matrix library written in C++, we have shown how to efficiently compute and solve a matrix-vector pair representing a "mesh" of equations that approximate the solution of `Poisson's equation` within a given boundary. While our implementation of `Gaussian elimination` is more efficient, `Cholesky decomposition` has an advantage at computing large mesh lengths and efficiently re-computing solutions. Further explanations regarding implementation details are noted in the following section.

VI. Authors' Thoughts

- As mentioned in section III, subsection B, we chose not to implement scaled partial pivoting when using `Gaussian elimination`. Scaled partial pivoting involves creating a vector of ratios by dividing each leading element of a matrix row by the maximum element in that row. The maximum value of these ratios is then selected and the row associated with that ratio is chosen as the pivot row. As our coefficient matrix solely consists of $-\frac{1}{4}$ and 1, the "scaling" effect is eliminated and rows with the highest "un-scaled" ratio are always selected. We observed that this led very quickly to large pivot numbers that exceeded the storage capacity of a `long double` for $N > 18$. It might be possible to negate this effect by recomputing the maximum elements for all rows still left to be simplified. Originally, the algorithm the authors followed only suggested computing the maximum values of each row at the beginning of the Gaussian process, rather than as described in the previous sentence.

- As addressed in section IV, both `Gaussian elimination` and `Cholesky decomposition` produced identical solutions. Both methods are stable for positive definite, diagonally dominant matrices, such as our coefficient matrix A, which leads them to produce the same solution (save for the edge case of `Gaussian elimination` with $N > 96$) [2] [3].
- The most costly aspect of our code, with respect to time, is the generation of lower and upper triangular matrices when using `Cholesky decomposition`. We observe that this process could potentially be sped up by implementing a modified form of the equations in (8), one in which certain zero elements are ignored rather than calculated. Since such a large majority of our coefficient matrix consists of 0's, we propose that a large number of unnecessary 0-multiplies can be avoided.

---

[1] Price, Clayton "The Almighty". Lecture notes. Written sometime in Spring 2019.

[2] Weisstein, Eric W. "Gaussian Elimination." From MathWorld–A Wolfram Web Resource. *http://mathworld.wolfram.com/GaussianElimination.html*

[3] Rosetta Code: Cholesky Decomposition *https://rosettacode.org/wiki/Cholesky_decomposition*