Abira Das & Patrick Sullivan

*Final Project Report*

CS5201

5/7/15

**Problem Statement**:

Evaluating mathematical functions or solving them through analytical means can be difficult.  We aim to avoid doing so by means of setting safe conditions to approximate the function with a small amount of error.  Our goal is to approximate a partial differential equation (PDE) that is comparable to the analytical solution of the PDE.

For this project, we are using Poisson’s equation, given by

u(x, y) = [sin(x)sinh(π-y)+sin(y)sinh(π -x)]

bounded on 0<x< and 0<y<. The values of Poisson’s equation on those boundaries are as follows:

u(x,0) = sin(x)           u(x,π) = 0      (0 < x < π),

u(0,y) = sin(y) u(π,y) = 0      (0 < y < π)

We constructed a comprehensive library of Matrix data structures and algorithms to solve a system of equations Ax = b, generated from an input n. The library comes with Gaussian Elimination with Partial Pivoting for regular matrices, and Cholesky decomposition for Symmetric positive definite matrices. We will use these two methods to solve the system Ax=b. We generate the A matrix based on the relations of the approximations to each other and the b vector based on the boundary values.  Solving Ax=b gives us the x vector, which is composed of our approximations.  The dimensions of A, and the lengths of x and b are all dependent on N, the mesh density of our approximations inside the boundary. To analyze this result, we compare the runtime of our chosen algorithm, measure the deviation of the approximation from the analytical solution values at the same coordinate, and observe how both of these metrics are affected by N.

**Project Design**

In our UML diagram, we define our matrix classes and solver classes through two interface bases. Cholesky Decomposition utilizes Symmetric, Upper, and Lower Triangular Matrices. These are represented as SymmMatrix, UpperTriMatrix, and LowerTriMatrix respectively. Gaussian Elimination works with a regular matrix with no special properties, implemented by FullMatrix. All four of these classes are templated on a given datatype, DT, and inherit basic matrix functions from MatrixBase. Binary operators for arithmetic, input/output, and relations are defined to utilize members defined in the matrix or vector class.  These functions take in one or two MatrixBases templated on <DT, MT> and passed by reference. They are defined in a way that when used, they will call the respective operator from whichever specific child class was passed in.

We represent our solver methods with three functors - Substitution Solver, Gaussian Solver, and Cholesky Solver. Substitution Solver is for the UpperTriMatrix and LowerTriMatrix classes, and they implement backward and forward substitution. CholeskySolver solves a SymmMatrix object with Cholesky Decomposition, and GaussianSolver does the same to a FullMatrix with Gaussian Elimination. These all inherit from the interface base SolverBase, which overloads the ( ) operator and takes in a MatrixBase<MT, DT>& and a Vector<DT>. For all solvers, the matrix represents A and the vector represents b for the Ax = b problem.

In addition to our matrix and solver library, we have a few more files. GlobalPDE defines the functions that were used to obtain the approximations and analytical solution. These are explained below in the *Mathematical Generation* section. All matrices are defined as a vector of vectors (of the same or varying sizes), so we also include a custom vector class. Finally, we have two error classes RangeError and SizeError for exception handling.

**Mathematical Justification**

Since solving a partial differential equation analytically can be a very complicated process, if not impossible, it is usually much easier to approximate the solution instead.

The Dirichlet problem shows how an approximation of a function can be made from within a constructed boundary of known values.  Using the Centered Difference Formula, we assume that each point within the boundary is the average value of the four cardinal points surrounding it.  This gives us the basis for constructing the A matrix, which shows the relation of each approximation within the boundary to any other approximation within the boundary.  The b vector is made up of the known values on the boundaries of our function.  We are solving to find the values in the x vector, which will be our final approximations.  We combine all of this knowledge into the equation Ax=b.  This means that when combining (through matrix multiplication) the interrelation of the approximations with the values of those approximations, the boundary values are all that remain.

We choose n to be the given mesh density of our approximations in the boundary.  This makes the A be a (n-1) by (n-1) matrix, as well as x and b vectors be of size (n-1).  Solving for the x vector of approximations uses the (n-1)2 equations of A.  The end result is the (n-1)2 approximations that are distributed within the boundary.  As n increases, the distance between interior points decreases and therefore the error also decreases. So for high values of n, the approximation will be extremely close to the analytical values. Thus, we use this method of solving a linear system of equations to estimate solutions for difficult partial differential equations.

In this analysis, we focus on applying Dirichlet problem to a PDE.  Since u(x, y) = f(x, y) on the boundary of the domain, by using the second derivative we can obtain an equation for the interior points based on the Centered Difference formula. Ignoring the error and assuming a large n (eliminating the (h2/4)g(xj, yk) term from the Centered Difference formula second derivative),

this equation is:

uh(xj, yk) = ¼[uh(xj+1, yk) + uh(xj, yk+1) + uh(xj-1, yk) + uh(xj, yk-1)]

**Mathematical Generation & Explanation**

To obtain our approximations, we wrote functions to generate the A matrix & b vector in the Ax = b problem for the given bounded Poisson equation. Our first step was to define the boundaries X\_MIN, X\_MAX, Y\_MIN, and Y\_MAX as zero for the minimums and pi for the maximums. The function poissonEdge takes in x and y parameters and returns their Poisson equation value if one of them is on a boundary. That function is called by genBpdevector to create the b vector used for solving the system. To make the vector, genBpdevector takes in n as its parameter and loops (n-1) \* (n-1) times, incrementing the x and y values by delta = (max - min)/n. To get the b vector value at a certain (x, y) point, it looks for adjacent boundaries and sums up their amounts. To fill the A matrix with the function genApdeMatrix, we use three loops to set the diagonals. Since our SymmMatrix class keeps the reflective property of Symmetric matrices, these diagonals will be created for both halves of the matrix. To calculate the approximations, we call the function pdeApproximate, which is templated on DT (data type), MT (matrix type), and ST (solver type). It generates A and b, solves the system of equations depending on the passed in solver, and returns a vector of approximations.

For the analytical answers, we only use two functions - one that contains the actual equation and one that repeatedly calculates on that equation for generated x and y values. The function poissonAnalytical takes two doubles, x and y and puts them into the formula to compute the answer at the point. Then, getErrorMatrix iterates on two nested loops to produce x and y points, the same way that genBpdevector does. However, instead of the analytical answer vector, the function subtracts corresponding analytical values from their respective approximation values and returns an error matrix. Since errors are represented as a matrix and not a vector, we must pass in the approximations as a const MatrixBase<MT, DT>&, which is made when getErrorMatrix is called in analyzeApproximation.

**Analysis of Results**

The first analysis of results involves the most basic component of approximating PDEs, solving a system of equations.  We compared the runtime duration of two different matrix-solving algorithms, Gaussian Elimination, and Cholesky Decomposition with Substitution.  The N value determines the size of the system to be solved, forming (N-1)2 equations and (N-1)2 unknowns.  Runtime was measured on the same Linux machine, under similar conditions.

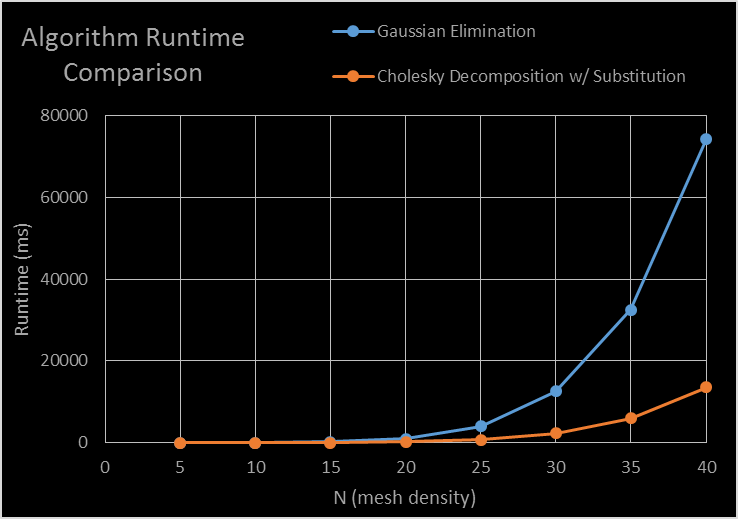
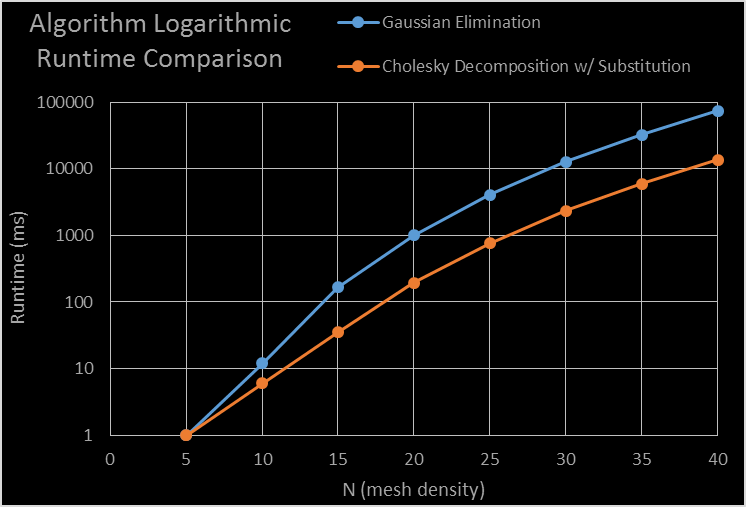


Figure : Algorithm Runtime Comparison



*Figure 2: Logarithmic Algorithm Runtime Comparison*

The results show that in this case, Cholesky’s Decomposition had a consistently shorter runtime than Gaussian Elimination.  This difference remains at roughly one magnitude of difference in runtime! It is also clear that increasing N exponentially increases the execution time of the algorithm for both algorithms.  This agrees with the previous observation that N forms a system of size (N-1)2.

These differences are consistent with the algorithm runtimes. Gaussian elimination and Cholesky Decomposition both take O(n3) time. However, a closer analysis reveals that Cholesky is about three times as fast because it does half the amount of computations. It takes three steps: decomposition, forward substitution, and back substitution. The decomposition takes (1/3)n3 flops, and forward and back substitution are both O(n2) flops. Thus, the runtime is O(n3/3) + O(n2) + O(n2) = O(n3/3) + 2O(n2) ⇒ O(n3)

Gaussian elimination takes two steps - forward elimination and back substitution. Again, back substitution is O(n2) but forward elimination is more costly. The cost is:

2n2 + 2(n-1)2 + ….. 2(22) +2(12)

=2 \* [[n(n+1)(2n+1)]/6]

=⅓(2n3+2n2+n) ⇒ O(n3)

Where 2n(n-1) is the cost of the first column (simplifying to 2(n2), 2(n-1)2 is for the second column, and etc...If we take the two largest values in Forward Elimination and Decomposition, one is approximately double the size of the other. The largest value in elimination is ⅓(2n3+2n2+n) is 2n3/3 and the largest in decomposition is n3/3, so forward elimination takes twice as much time. The total cost for Gaussian elimination is the elimination plus the back substitution, or ⅓(2n3+2n2+n) + O(n2). Therefore, since the Cholesky decomposition takes half as many operations as the Gaussian for the largest computational factor, it is more efficient.

We can also consider our storage means for SymmMatrix, UpperTriMatrix, and LowerTriMatrix compared to FullMatrix. When filling these three types of matrices, we are only concerned with half of the values. When using a FullMatrix, we must fill in all of the values even if one half contains zeros or the same reflected values of the other half. Therefore, we can potentially save some computation time when working with a large n and thus generating a large matrix. Since CholeskySolver utilizes these storage capabilities, it benefits from them upon matrix construction.

In the analysis of the results from approximating Poisson’s PDE, “Approx” refers to the set of z values calculated using boundary conditioning and the Centered Difference formula on Poisson’s Equation. The boundaries set for these approximations are 0<x< and 0<y<. Error is calculated as the absolute value of the difference between the approximation value and the value gained by the analytical solution.  This allows us to compare the resulting approximations based on their range of error as well as the average error per approximation. Error, instead of runtime, is the defining measurement in this analysis. This makes algorithm choice irrelevant, except in the case of information loss by truncation in calculations. However, no observable difference was found in the amount of error even when compared in all trials. It is safe to assume they behaved identically in these results.

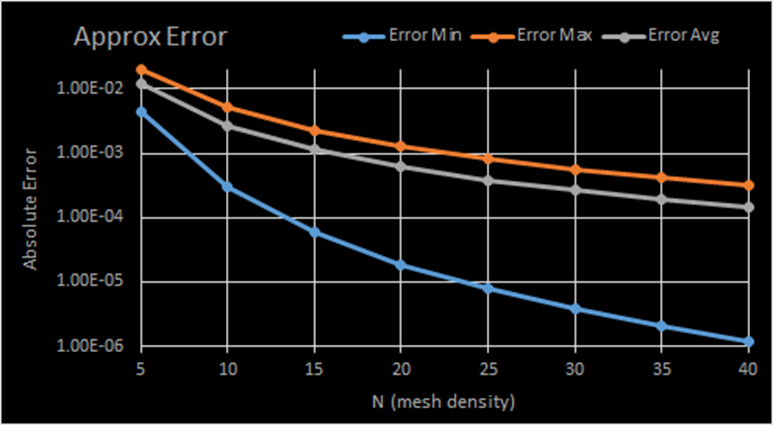


Figure : Relation of error in approximation to value of N

The overall analysis of error in Poisson’s Equation approximation shows the minimum, maximum, and average absolute error for the set of approximations constructed inside of the boundary.  Note the logarithmic scale of error, and the decrease of all error features in the approximations.  As N increases, the range of possible error values increases drastically. An important part of these results is the diminishing returns of increasing N to decrease error.

Results of the approximations were recorded and graphed for N values of 5, 10, and 20.  Changing the mesh density allows us to see the effects and clarity of increasing N to decrease error and see more features of the analytical solution in our approximation.

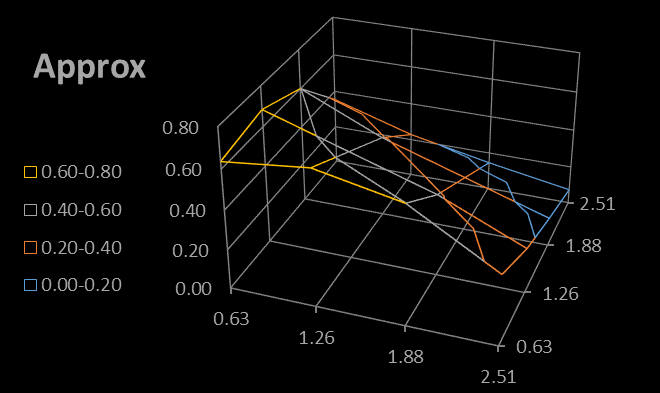


Figure : N = 5 Approximations

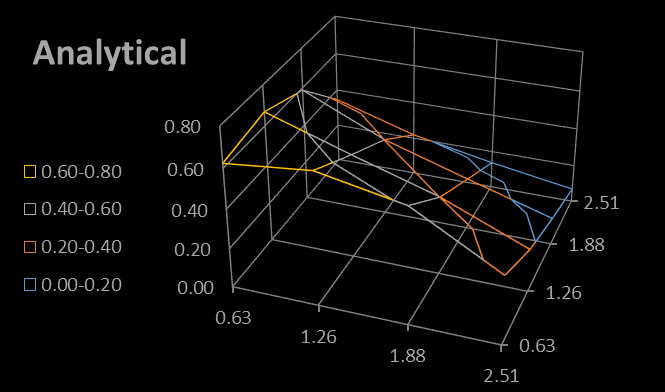


Figure : N = 5 Analytical Values

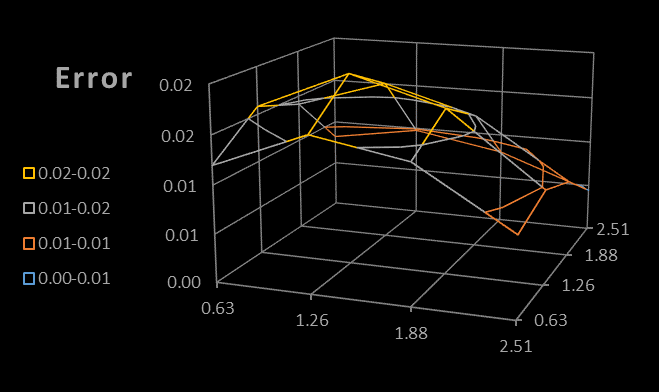


Figure : N = 5 Error in Approximations

N=5 results are shown to be very similar to the analytical values generated at the same x & y values.  All error in approximations is seen to be less than 0.02.

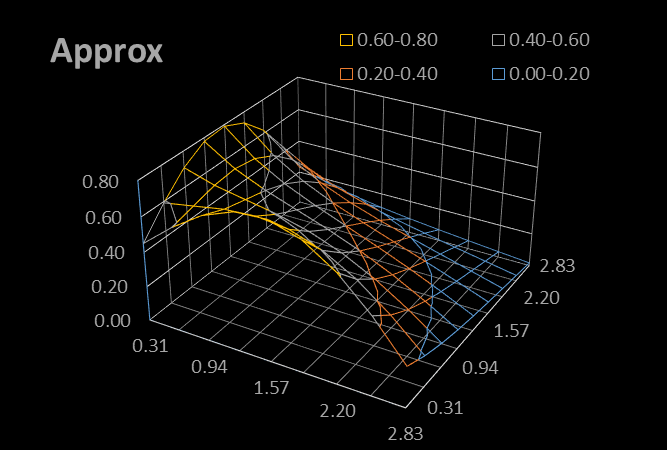


Figure : N = 10 Approximations

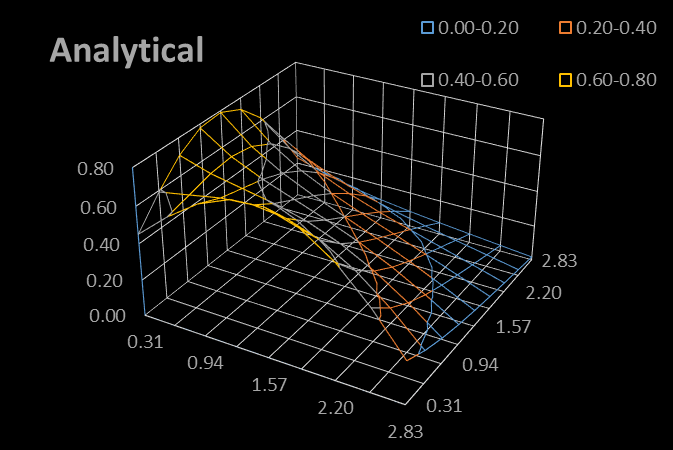


Figure : N = 10 Analytical Values

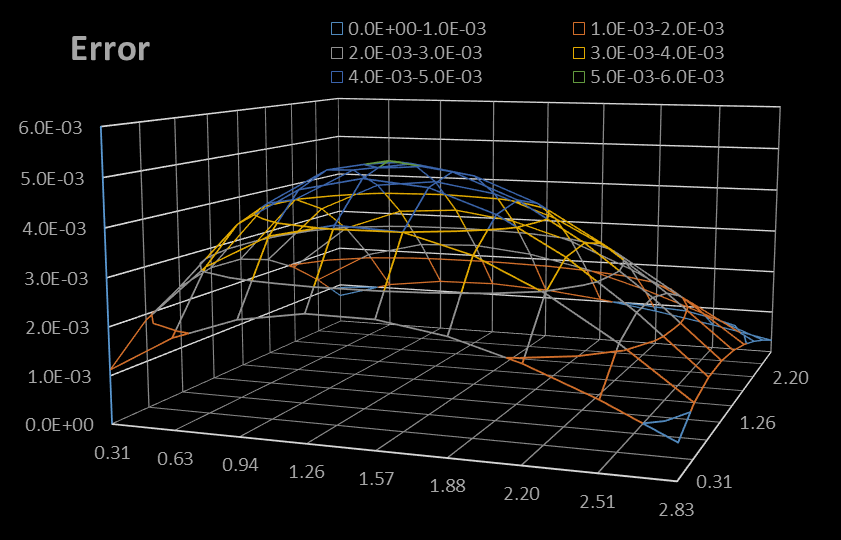


Figure : N = 10 Error in Approximations

N=10 results show a clearer and more continuous function.  The error observed in the approximation is relatively high in the center of the boundary.  This is expected, as they are the coordinates farthest from the errorless boundary values in the approximation.  Coordinates that are closer to the corner of the bounded area have less error in their approximations due to their proximity to two errorless boundary values.

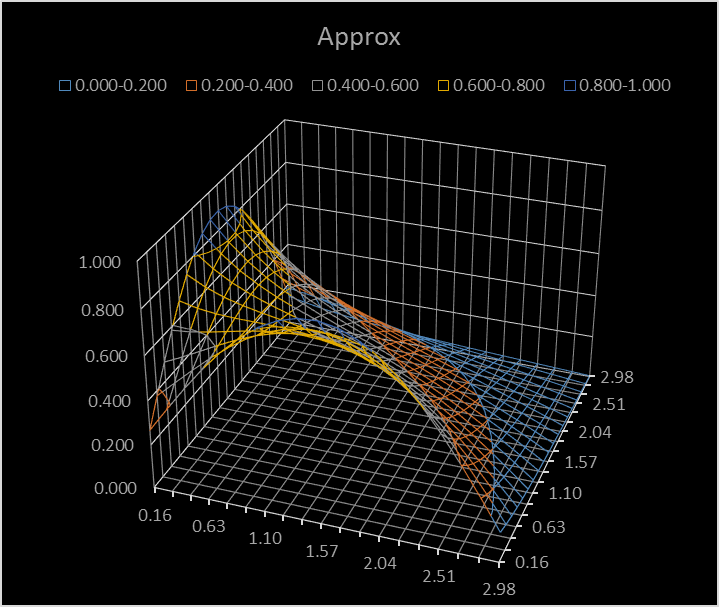


Figure : N = 20 Approximations

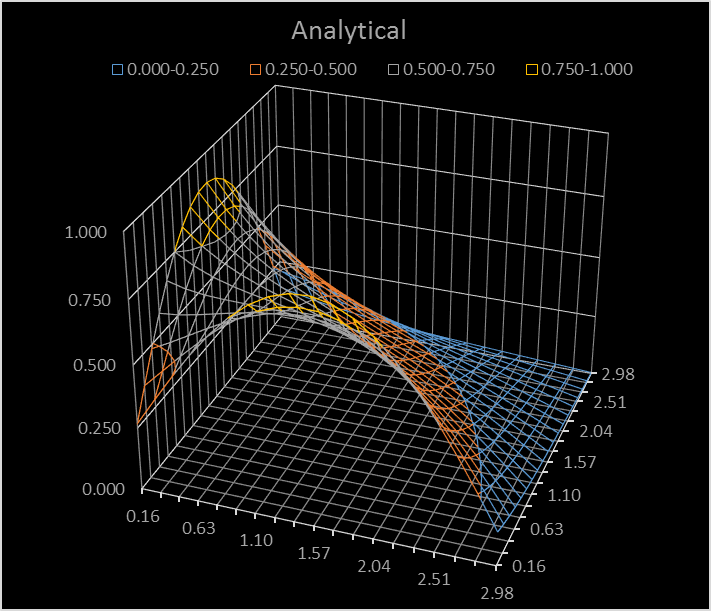


Figure : N = 20 Analytical Values

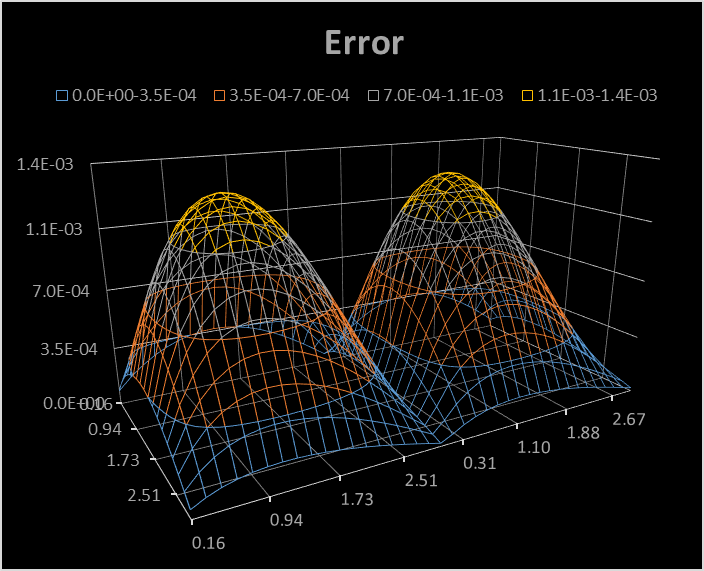


Figure : N = 20 Error in Approximations

With N=20, results are comparable to N=10.  However, notice the valley formation found in the error.  This would likely be caused by moving for repeated overestimation in approximation values to repeated underestimation in approximation values.  Error is now below 0.0015.

**Conclusion**

There is an infinitely large set of Partial Differential Equations. Solving these equations by analytical means can be difficult, time-consuming, or even impossible. Thus, calculating approximations for these equations is a valuable way to obtain extremely close estimations.

Approximating allows us to reduce the number of evaluations and avoid the analytical solution of the function altogether.  Approximating the values of PDEs through Dirichlet’s problem on a small boundary allows us to calculate close approximations of the function within the boundary while minimizing error.  We constructed a program to do this with Poisson's equation.

From our program, we made several conclusions about the impact of n’s value and the runtime of Cholesky Decomposition and Gaussian Elimination. We saw that increasing the mesh density led to non-exponentially less error, and consequently took more time to compute. This increase of n did improve our approximations of Poisson’s Equation.  As interior points got closer to the center, the magnitude of error increased. The error propagated from calculating points adjacent to boundaries and going inward. We saw that our CholeskySolver was significantly faster than our GaussianSolver, which was reinforced by Big-O complexity analysis of the two algorithms. Finally, we concluded that approximations are an efficient, useful way to solve difficult mathematical problems.