CS 5201 Final: Solving a Dirichlet Problem via Cholesky Decomposition and Successive Over-Relaxation

Jackson Willbrand

May 2020

1 An Introduction to Poisson's Equation and the Dirichlet Problem

For the purposes of this final assignment and our related lectures, Poisson's equation, the equation we will be solving and the focus of this assignment, in two-dimensional Cartesian coordinates, takes the following form:

$$\frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2} = f(x, y). \tag{1}$$

For the purposes of this assignment, we are solving this equation in order to determine the values of black pixels on a white background. In our case, the set of all black pixels in the given image is defined as the set S, where the white boundaries are defined as the set ∂S. All white background pixels could be described as boundary pixels, which have a boundary value of zero. $U(x, y)$ is the boundary equation for this assignment, where $U(x, y)$ returns a value representing the distance from the closest boundary to the pixel located at (x, y). By creating this definition of $U(x, y)$, we are able to classify $U(x, y)$ recursively, in terms of the four average surround values, plus a constant value. This constant value is a representation of the time required to move to a neighbor, and will equal 1 for this assignment. Thus, at every point (x, y) that is not a boundary value, i.e. $(x, y) \notin \partial S$, the value $U(x, y)$ can be described as:

$$U(x, y) = 1 + \frac{1}{4}(U(x + h, y) + U(x - h, y) + U(x, y + h) + U(x, y - h)). \tag{2}$$

Since the U value of each black pixel is a function of the surrounding pixels, it would be wise to think of this problem in terms of a large, albeit sparse matrix, where each row (equation) in the matrix is reliant on a maximum of four other unknown U values, with the remaining entries in the row equation to zero. This system of equations forms the large matrix A. If we represent this system as a linear system of equations, we can accurately represent it under the equation $Ax = b$, where $x = \{(x, y) : U(x, y) \notin \partial S\}$, or the set of all unknown pixels, and $b = f(x, y)$, where $f(x, y)$ is a forcing function equal to the time required to move to an immediate neighbor. As stated above, this value is 1, and thus $b = \{1, 1, ..., 1, 1\}$, a set of all 1 values the same size as x.

2 Ax = b and the Unique Properties of A

The size of A grows to be an $n \times n$ matrix, where n is the number of unknown values in the image. This matrix grows extremely quickly, and thus conventional methods of matrix solving such as Gaussian Elimination and LU Decomposition, which run in $O(n^3)$ time, are not quick enough to efficiently solve these systems. Fortunately for us, there are properties of the matrix A that make other types of matrix solving much more efficient, and examining these methods is the ultimate goal of this project.

Since all unknowns are a function of the surrounding unknown values (and vice versa), our matrix is symmetric, meaning that $A = A^T$, or that switching the rows and columns of A with themselves results in the same matrix. On top of this, since each unknown contains itself in its equation, the diagonal of A is filled with 1 values, equivalent to an identity matrix's diagonal. As stated in the introduction, this matrix is also extremely sparse. Each row in the matrix can only have a maximum of 4 nonzero values (besides itself), and so an $n \times n$ matrix with $n = 250$ will have a maximum of 1000 '-0.25' values, and 250 1 values down the diagonal. The 61250 remaining matrix entries will be zeroes.

Finally, this matrix is banded. If a matrix A has a bandwidth p, then $A_{ij} = 0$, for all values of i and j within p of each other. In matrix form, this representation looks similar to a diagonal band of nonzero entries running from the top left to the bottom right of the matrix, with the remaining elements as zero. An example of a matrix with this banded property is shown below (and has a bandwidth value of $p = 1$):

$$A = \begin{bmatrix} a & b & c & 0 & 0 & 0 & 0 \\ 0 & d & e & f & 0 & 0 & 0 \\ 0 & 0 & g & h & i & 0 & 0 \\ 0 & 0 & 0 & j & k & l & 0 \\ 0 & 0 & 0 & 0 & m & n & o \end{bmatrix}. \tag{3}$$

Combining these properties, we are able to implement two more efficient methods to solve our matrix system $Ax = b$. These two methods are Cholesky Decomposition, which can reduce our time complexity to $O(\frac{n^3}{3})$, around 3 times more efficient than simple Gaussian elimination, and Successive Over-Relaxation, which has a time complexity of $O(n^2)$, an order of magnitude more efficient than both methods, and the predominant focus of the assignment.

3 Implementing Dirichlet Boundary Problem Solutions

3.1 Cholesky Decomposition

My first algorithm to solve the assignment's problem was Cholesky Decomposition. Cholesky Decomposition is available to all matrices that Hermitian and positive-definite. In this case, since our matrices were dealing with real numbers exclusively, a matrix that is symmetric and real is by definition a Hermitian matrix. Thus, Cholesky Decomposition is a viable option for solving our matrix equation $Ax = b$. I chose to perform Cholesky Decomposition as one of my solving algorithms because while it is not as efficient as many of the other available iterative methods, it is a direct method of solving $Ax = b$. This means that the solution to the matrix equation via Cholesky Decomposition would be perfectly correct, assuming no floating point error. This makes testing future methods much easier, as comparing their solutions to a solution found via Cholesky Decomposition would be a good comparison of accuracy.

Cholesky Decomposition, in addition to the plus that it is a direct method, is around half of the time cost of LU decomposition or Gaussian elimination, which has a time cost on the order of $O(\frac{2n^3}{3})$. Decomposing a typical matrix A into Cholesky form involves the following pseudocode, which converts a symmetric positive definite matrix A (whose elements are described as a_{ij}) into a lower triangular matrix L (whose elements are described by l_{ij}):

for $k = 1 : n$ **do**
  for $i = 0 : k - 1$ **do**
    $inner\_sum = \sum_{j=1}^{i-1} l_{ij} a_{kj}$
    $l_{ki} = (a_{ki} - inner\_sum)/l_{ii}$
  end for
  $outer\_sum = \sum_{j=1}^{k-1} l_{kj}^2$
end for

Implementing the following pseudocode will convert a positive-definite, symmetric matrix A into a lower triangular matrix L, such that $A = LL^T$. This is useful for a few reasons. First, having our upper triangular matrix be the transpose of our lower triangular matrix, we don't have to store the upper triangular in memory, as we can access the lower triangular matrix to get the values we need. This should cut the cost of storing A in half. Second,

similarly to LU decomposition, once we decompose A into its Cholesky form, we can solve the system

$$LL^T x = b \tag{4}$$

in $O(n^2)$ instead of $O(n^3)$ time, as L is in lower triangular form and thus is already reduced to row echelon form. As such, Rearranging equation (4) into the following two equations

$$Ly = b \tag{5}$$

$$L^T x = y \tag{6}$$

and solving them sequentially will allow us to solve the system without reducing to row echelon form, and will allow us to solve $Ax = b$ for an infinite number of b vectors.

3.2 Successive Over-Relaxation

While the Cholesky Decomposition is around 3 times faster than typical Gaussian elimination and allows us to solve linear systems $Ax = b$ for any b vector in $O(n^2)$ time after decomposition, more efficient methods exist, even when solving a system for the first time. For the purposes of this assignment and for this specific problem, I decided to implement Successive Over-Relaxation.

Successive Over-Relaxation (known as SOR henceforth) is an iterative technique that allows us to solve the linear system $Ax = b$ in $O(n^2)$ time (even the first time!), an order of magnitude better than Cholesky Decomposition. However, iterative techniques do have downsides. As they are not direct techniques, an exact solution to the system can not be perfectly found. However, the accuracy of these iterative methods after sufficient iterations is accurate enough for the purposes of this assignment (and almost all applications). To determine the error of these iterative techniques, we cannot simply compare the current x vector to the desired x vector, because we don't know the desired x vector (that's the whole point of the problem). Instead, we compare the change in magnitude of the x vector between iterations. The magnitude, or norm, of a vector x is calculated as follows:

$$||x|| = \sqrt{x_1^2 + x_2^2 + ... + x_n - 1^2 + x_n^2}, \forall x_i \in x. \tag{7}$$

If a vector is not changing much between iterations (ie it is close to converging to the correct solution), then its $||x||$ will be small. Thus, we can assign a variable e_s to be our error threshold. This value is the minimum amount of error we are content with. We will continue to iterate our SOR until the vector x is changing by a magnitude of e_s or less between iterations. Once this error is small enough, we stop iterating.

Now that we have a good explanation/understanding of our algorithm's stopping conditions, we will examine the algorithm itself in more detail. Successive Over-Relaxation (SOR) is a variant of the Gauss-Seidel method for solving linear systems of equations. This method has a faster runtime than Gauss0Seidel, and converges faster. Given a linear system $Ax = b$, we must have A such that it can be decomposed into its diagonal component D, and its strictly lower and upper components L and U, where:

$$D = \begin{bmatrix} a_{11} & 0 & ... & 0 \\ 0 & a_{22} & ... & 0 \\ ... & ... & ... & ... \\ 0 & 0 & ... & a_{nn} \end{bmatrix}, L = \begin{bmatrix} 0 & 0 & ... & 0 \\ a_{21} & 0 & ... & 0 \\ ... & ... & ... & ... \\ a_{n1} & a_{n2} & ... & 0 \end{bmatrix}, U = \begin{bmatrix} 0 & a_{12} & ... & a_{1n} \\ 0 & 0 & ... & a_{2n} \\ ... & ... & ... & ... \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Thus, the equation $Ax = b$ can be written with its component matrices D, L, and U as

$$(D + \omega L)x = \omega b - [\omega U + (\omega - 1)D]x, \tag{8}$$

where $\omega > 1$ is the relaxation factor, or simply a measure of the weight we should put on the previous iteration of the method, and is a function of the spectral radius of the matrix A. For the purposes of our equations, since direct computation of eignvalues are expensive, we want to choose a relaxation parameter suitable for all matrix cases (although not perfect). For spectral radius from 0.6 to 0.95, which handles the vast majority of cases of A, we will choose $\omega = 1.6$.

Rearranging equation (8) into an iterative technique, we arrive at the following method:

$$x_i^{k+1} = (1 - \omega)x_i^k + \frac{\omega}{a_{ii}}(b_i - \sum_{j<i} a_{ij}x_j^{k+1} - \sum_{j>i} a_{ij}x_j^k), \tag{9}$$

iterating from $i = 1 : n$. Since the iteration occurs n times (where n is the rows/columns of A), and each iteration involves a summation from 1 to

n, this algorithm successfully runs with a time complexity of $O(n^2)$, an order of magnitude faster than the Cholesky Decomposition, which contributes to a much faster runtime, especially for large matrices.

In addition to this, we can shorten the runtime of the algorithm by applying our approximate error measurements described above. This will stop the algorithm from unnecessarily running after convergence have been sufficiently found. The generalized algorithm for Successive Over-Relaxation, then, is below:

$iter = 0$
while $iter < n, e_a > e_s$ **do**
  $e_a = 0$
  for $i = 1 : n$ **do**
    $sum = 0$
    for $j = 0 : n$ **do**
      if $i = j$ **then**
        $sum+ = a_{ij}x_j$
      end if
    end for
    $x_{old} = x_i$
    $x_i+ = \omega * (((b_i - sum)/a_{ii}) - x_i)$
    $e_c = abs(x_{old} - x_i)$
    if $e_c > e_a$ **then**
      $e_a = e_c$
    end if
  end for
end while

,

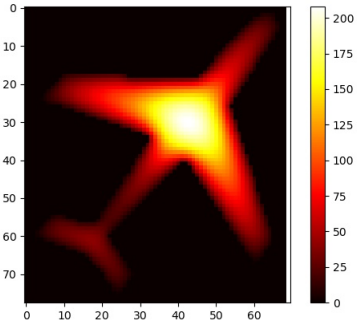
where e_s is the accepted error threshold, A is the coefficient matrix with elements a_{ij}, n is the amount of rows/columns in A, and x is the solution vector.

4 Implemented Solutions on Various Example Problems

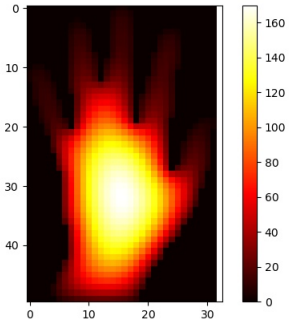
This section will be dedicated to providing solutions my program achieved when giving an example problem. The strcture for this assignment was as follows: (1) a program converted a black and white image to a text file filled with B and W, representing black and white pixels. Then, (2) my program, given this text file, created the associated coefficient matrix A and vector b to solve the linear system $Ax = b$, where x contains the values of all the black pixels. It then solved this system. Then, a separate function outputted this data into a csv file, mimicing the same shape as the original photo. Finally, a separate python program that I created converted this csv to a readable heatmap, succinctly demonstrating my solutions.

In each example below, I will include the original black and white picture, a variable n representing the size of the coefficient matrix A (where A is an $n \times n$ matrix), and the respective heatmap after all solving has been completed. Also, I have included the time that Cholesky Decomposition or Successive Over-Relaxation took to solve the problem.

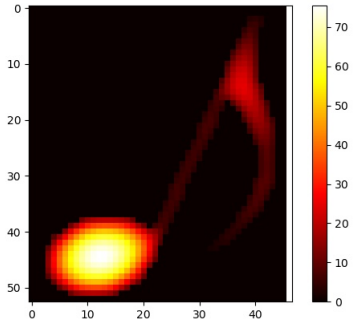
4.1 Solution 1 (n = 1679, SOR Solve Time = 14.2s)



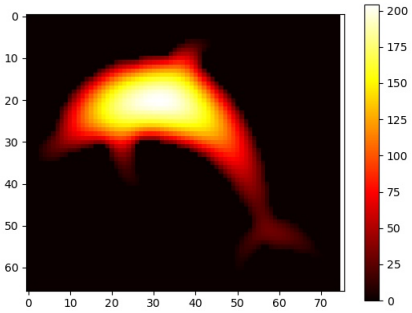
4.2 Solution 2 (n = 900, SOR Solve Time = 2.4s)



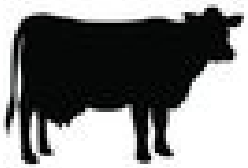
4.3 Solution 3 (n = 490, SOR Solve Time = .38s)

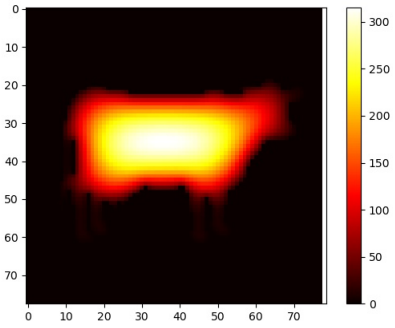


4.4 Solution 4 (n = 1207, SOR Solve Time = 5.98s)



4.5 Solution 5 (n = 1636, SOR Solve Time = 15.2s)





5 Program Reusability, Adaptability, and Modularity

An important aspect of object-oriented programming is that programs are reusable (meaning that can be run multiple times to solve problems many times), adaptable (they can easily be changed to accomodate other similar, though not exact problems) and modular (parts of the program can be taken out in sections and used in other programs). In this section, I will describe the ways in which this program is reusable, adaptable, and modular.

5.1 Design Reusability and Adaptability

Cholesky composition is reusable is the sense that it is implemented via a class. My Cholesky decomposition class stores the Cholesky-decomposed matrix in memory, allowing us to solve many systems of $Ax = b$ for many different b vectors, similarly to how LU decomposition can be used to solve many systems. Similarly, Successive Over-Relaxation functions in the same way, accepting any kind of b vector and solving the system $Ax = b$.

My program also takes command line arguments, namely the name of the image to read from and the name of the csv file to output afterwards. By Implementing a feature like this, my program is able to accept any type of image, and will convert it into a csv file with a name determined by the user. My program also allows for the user to choose whether they would like to use Cholesky Decomposition or Successive Over-Relaxation as their method,

letting them choose a slower, exact solution or a quick, convergent, accurate, yet approximate solution.

The custom matrix, vector, and iterator container classes can also be freely taken and used for other projects. They are templated on their type and can be used for many different purposes. For the purposes of this assignment, however, a double type is exclusively used, as using floats can lead to more error and I value the increased accuracy from doubles over the increased storage cost.

Finally, the forcing function $f(x, y)$ I utilized for this project to create the b vector, and the ω relaxation value for Successive Over-Relaxation is not hard coded. This allows the program to be modified to accept many different values of $f(x, y)$ and ω, providing even more adaptable functionality to the user of the program.

5.2 Design Modularity

The design of final project follows good object-oriented modular design principles. Important functions that are useful and repeated multiple times, but that serve one main functionality are built behind functor classes. The most relevant of these designs would be my "CholeskyDecomp" and "SuccessiveOR" classes. Both of these functor classes implement the functionality of Cholesky Decomposition and Successive Over-Relaxation, respectively. The member variables inside of them allow these operations to be used for many different types of A matrices (so long as the matrices fulfill the requirements of the respective method).

Another example of good usage of functor classes would be my "CSVOutputter" class. Similarly to the above classes, this functor class provides the ability to convert a solution vector x and its relevant B/W text file into a csv file that can be converted into a heatmap image. All of the 3 above classes are implemented in their own respective .h and .hpp files, allowing them to be taken out and reused for other projects when necessary (the same is the case for my "MyMatrix", "MyVector", and "MyIterator" container classes).

On top of this, the functions that help create the coefficient matrix A from a given text file are implemented in their own .h and .hpp files, and are similarly modular in their design approach.

Finally, the entire program contains extensive Doxygen-compatible documentation to accurately describe each function/class's functionality to a user.

6 Reflection

In retrospect, I am happy with how my implementation worked out. I chose methods that I learned very well by the end of the assignment, and my chosen methods solve the $U(x, y)$ values in the form of the linear system $Ax = b$ quickly and accurately. One of the things I wish I could do, given more time, and will probably do after the assignment is done, is implement solutions that run in $O(n)$, or similar time. Methods like this, such as the Conjugate Gradient and Multigrid methods, are extremely interesting and efficient, but also comparatively much harder to implement successfully, and I could not implement them efficiently or with a large enough understanding of the solution to be able to accurately describe it.

7 Conclusion

This program solved the Poisson equation and a Dirichlet Boundary problem by constructing a coefficient matrix A to simultaneously solve for all the unknown non-boundary values described by $U(x, y)$, the value of a given pixel at the location (x, y). Interesting properties arise from A, most importantly that A is sparse, positive-definite, symmetric, and banded. This allows us to apply a wide range of methods to solve the system $Ax = b$ in much faster time than the $O(n^3)$ produced by simple Gaussian elimination. To demonstrate this, I implemented Cholesky Decomposition and Successive Over-Relaxation. Cholesky Decomposition has a $O(n^3)$ runtime for the first system solved, by can solve any future system with the same coefficient matrix A in $O(n^2)$ time. Cholesky Decomposition is also an exact method. Successive Over-Relaxation, while not an exact method, converges to an accurate solution in $O(n^2)$ time, an order of magnitude faster than typical elimination methods.

Utilizing these methods, I displayed their runtimes and solutions to 5 example problems, providing a quick and accurate solution in all cases. The implementation of these algorithms followed good object-oriented principles, as the design was modular, reusable, and adaptable in many cases.