# Final Project: Image Analysis with Poisson's Equation

Noah Klein and Jeffrey Strahm

May 10th 2020

## 1 Problem Statement

In the field of computer vision, silhouettes are incredibly valuable tools that can be used to determine many properties of an image. In this assignment, we were tasked with employing two different methods of solving Poisson's equation to assign each "pixel" in an image a value based on its average cost to randomly walk to the edge of a silhouette.

We create this system by partitioning a domain in the xy-plane that in our case would represent the border of the picture we are concerned with. For each point in the interior of this domain, we find that the value at any point is given by the average of all four of its neighbors minus the step size required to reach a neighboring point, given by a "forcing function". This creates the equation:

$$u(x_i, y_j) = \frac{1}{4}[u(x_{i+1}, y_j) + u(x_i, y_{j+1}) + u(x_{i-1}, y_j) + u(x_i, y_{j-1})] - \frac{h^2}{4} f(x_i, y_j)$$

With this equation, we need to be able to find the value for all of the interior points simultaneously. We set it up as a system of equations, Ax = b + f, where A represents the coefficients for each point, x is a vector that holds the values of all the interior points that we are solving for, b is a vector that holds all of our known information at all the boundary points, and f is also a vector that holds all the values that the known forcing function produces at all the points.

For this project, the silhouette is represented as a character matrix of 'B' and 'W' characters where the 'B' characters represented a pixel inside the silhouette and the 'W' characters represented a pixel outside the silhouette. The program reads in this matrix from a file. Once read, the program constructs a system of equations based off of the input matrix that is used to find the values of the interior pixels of the input matrix. The program performs the user specified method of solving the Poisson system. The user can either choose the Cholesky decomposition method or the Jacobi iteration method.

## 2 Mathematical Justification

In this section we look at how both methods we chose for solving this specific system of equations take advantage of the specific properties of the system in order to create a fast solution.

### 2.1 Cholesky

In the Poisson system, the the A matrix from the Ax = b system has many useful properties that a programmer can take advantage of when choosing an algorithm to solve the system. Some of these properties include being symmetrical, banded, sparse, and weakly positive definite. The base Cholesky algorithm takes advantage of the symmetric and weakly positive definite properties of the matrix to improve upon the LU decomposition algorithm for solving a system of equations. The Cholesky algorithm saves memory and time by decomposing the matrix $A = LL^T$[1]. The creation of the L matrix by decomposing the A matrix with the Cholesky algorithm also saves a lot of time compared to normal Gaussian elimination, since there are two formulas that are used depending on the position of the current point you are attempting to decompose. This allows the program to only have to store the L matrix instead of having a L and a U matrix, and simplifies the decomposition process into two formulas.

### 2.2 Jacobi

The Poisson equation is solved when each cell equals the average of it's surrounding cells plus the step size, not including the known 0 cells. Thus, if we perform this calculation repeatedly, we should eventually converge on the solution. While it would take near if not infinite iterations to find the true value, we can quickly find a close approximation.

## 3 Methodologies

### 3.1 Cholesky Techniques

While the base Cholesky algorithm takes advantage of many of the properties of the A matrix that we discussed in the last section, we can also optimize our code to take advantage of the sparse and banded properties of the A matrix as well. While each row of the matrix contains a coefficient for every interior point

[1]$L^T$ denotes the transpose of L

in the space we are observing, no individual point (represented by the column on the diagonal of the matrix) will ever reference a point other than the space immediately left or right on its row. It will also never refer to a point other than the ones immediately above or below itself in the character matrix. If the character matrix has N columns, these points are represented by the points ±N points columns away from the diagonal of the A matrix. We can see that this creates a banded matrix where all the other points will be 0.

With SIZE equaling the number of rows/columns in the A matrix, and 0 as the starting index for the array, the two base decomposition methods mentioned earlier can be written as

$$L(i,j) = \sqrt{A(i,j) - \sum_{k=0}^{SIZE-1} L(i,k)^2}$$

For every point on the diagonal of L, or when i = j, and

$$L(i,j) = \frac{1}{L(j,j)} \left( A(i,j) - \sum_{k=0}^{SIZE-1} (L(i,k) * L(j,k)) \right)$$

For every other point not on the diagonal. This constructs a lower triangular matrix, L, where $LL^T = A$. This is nice and simple since the algorithm iterates through every point in the L matrix underneath the diagonal, or j <= i, to construct it. However, since we know that our A matrix is sparse and banded, we can incorporate some bounds on the j and k loops to save a lot of unnecessary looping. Instead of having the j and k loops start at zero for every row, we can use the fact that no point will ever reference another if it is more than N (number of columns in the character matrix) spaces away. With this, we can allow the j and k loops to start at $i-N$ when $i >= N$ drastically reducing the number of iterations required for constructing the L matrix once i gets sufficiently large. We can assume that by default the spaces we skipped in deriving the values for the L matrix are set to 0 in the default construction of the L matrix.

Now that we have our L matrix constructed, we still have to solve the equations Ly = b and $L^T x = y$. We can simply use forward substitution to fill in the values for the y vector

$$y(i) = \frac{1}{L(i,i)} \left( b(i) - \sum_{j=0}^{i-1} (L(i,j) * y(j)) \right)$$

then use backward substitution to fill in the unknown values of the x vector

$$x(i) = \frac{1}{L(i,i)} \left( y(i) - \sum_{j=i}^{SIZE-1} (L(j,i) * x(j)) \right)$$

giving us our solution. This method can also be improved in a similar way to the construction of the L matrix if we notice that the L matrix itself is sparse and banded. This allows us to employ a similar time saving technique by bounding the j loop by the same bounds of i ±N to save ourselves from having to iterate through a bunch of values that we know would produce 0 anyways.

The final algorithm improvement comes from the fact that we already know many of the values of the x vector already, the 'W' or white points inside the boundary. We can skip solving for the values of y and x that we already know. Following this algorithm produces an exact solution to the system we originally started with where the values of x can be converted back into a matrix form based on the dimensions of our original character matrix.

## 3.2 Cholesky Parameters

For implementing this algorithm we created a cholesky class that had an operator() overload. This operator takes two parameters. One is a nTrix<char> object called *data* that represents the 'B' and 'W' version of the image silhouette. Either dimension of this matrix must be smaller than 32,767 to meet the size constraint of an array in c++. The other parameter is a float called *step* that represents the step distance from one point to one of its neighbors. This section lists out all the parameter's construction and usage in this function.

### 3.2.1 coefficient

The variable *coefficient* represents the A matrix in the system Ax = b. This matrix holds floating point values and is constructed as a (M*N) * (M*N) matrix where M and N are the number of rows and columns in the *data* parameter matrix respectively.

### 3.2.2 solution

The variable *solution* represents the matrix representation of the x vector in the Ax = b system. It is constructed as a M * N matrix.

### 3.2.3 lower

The variable *lower* stores all the values of the deconstructed A matrix from the equation A = LL$^{\mathrm{T}}$. Like *A*, *L* is constructed as a (M*N) * (M*N) floating point matrix.

### 3.2.4 final

The variable *final* stores all the final values of each interior point from the Poisson equation and the equation Ax = b. It is constructed as a (M*N) size vector of floating point values.

#### 3.2.5 intermediate

The variable *intermediate* stores the intermediate floating point values when solving Ly = b. This is constructed as a (M*N) size vector of floating point values.

#### 3.2.6 known

The variable *known* stores the known values on the boundary of our Poisson system. This is constructed as a (M*N) size vector of floating point values and is initialized with the value of the *step* parameter passed.

### 3.3 Jacobi Methodologies

This solution calculates the value for each cell using the Poisson equation for this problem. It does this by iterating over the matrix of information and calculating each cell until the change of all individual cells falls with a designated error. It determines if it is within the error by seeing if the number is greater than the new number plus the error or less than the new number minus the error. Cells with a value of 0 are skipped. We can show this by using the equation

$$u(x,y,m{+}1) = \frac{1}{4}\left(u(x_{i-1},y_j,m) + u(x_{i+1},y_j,m) + u(x_i,y_{j-1},m) + u(x_i,y_{j+1},m)\right) + \mathit{step} \quad (1)$$

Where $m$ is the m$^{\text{th}}$ iteration of the calculation. We can calculate the error between iterations by checking the change of the value at every point against the tolerated error to make sure it is within the bounds. If any point fails, another iteration is run.

During construction, the program checks for consecutive rows and columns it can ignore on the outside of the silhouette in order to reduce unneeded checking of known zeroes. Two additional rows and columns are added on the outsides of the matrix in order to be sure that the algorithm will never reach off of the matrix.

### 3.4 Jacobi Parameters

Since both of our solving methods derived from the same base class, the parameters used in the jacobi class operator() overload are identical. The character matrix can only be 32,765 in either dimension since 2 rows and columns get added as padding for the boundaries. The other variables in this method are what allows this method to use less memory than the Cholesky algorithm.

#### 3.4.1 input matrix

Should be a M * N char matrix with white cells designated as W and black cells designated as B. N and M must be smaller 32,765.

#### 3.4.2 step size

The cost to move to a neighboring cell. Must be greater than 0 as there must be a cost to perform the step.

#### 3.4.3 Error

The *error* member variable of the jacobi class is a floating point value that denotes the maximum accepted error between iterations. This is what determines the stopping condition of the entire algorithm. This parameter can be set by a user and has a base of .000001. When the values of each cell change fall within + or - of the error the algorithm terminates and the output matrix is returned.

#### 3.4.4 result

The variable *result* is a matrix of size (M+2) * (N+2) of floating point values that stores all the values from each iteration and ultimately serves as the storage of the solution calculated by the jacobi method.

#### 3.4.5 Iterations

There is no maximum number of iterations as the program will continue until the change of all cell's values falls within the designated error.

## 4 Solution

We have provided an example of a solution that would be produced from each of our methods. We chose to use the cow.jpg image at maximum size. This creates a character matrix of size 78 * 78, N = 78. For both images, some of the surrounding zeroes are cropped for readability.

## 5 Viability

### 5.1 Cholesky

The Cholesky method allows us to calculate an exact solution to the problem, while still being able to take advantage of the many unique properties given to use in certain aspects of the problem. Generating an exact solution comes at a cost however, as this method is noticeably slower than the approximate solution reached by the Jacobi method.

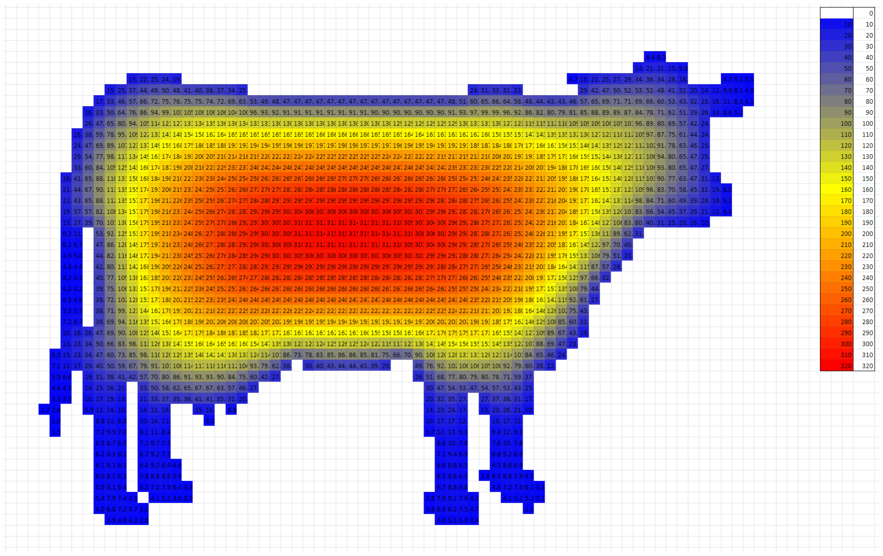

Figure 1: Cow created from Cholesky method

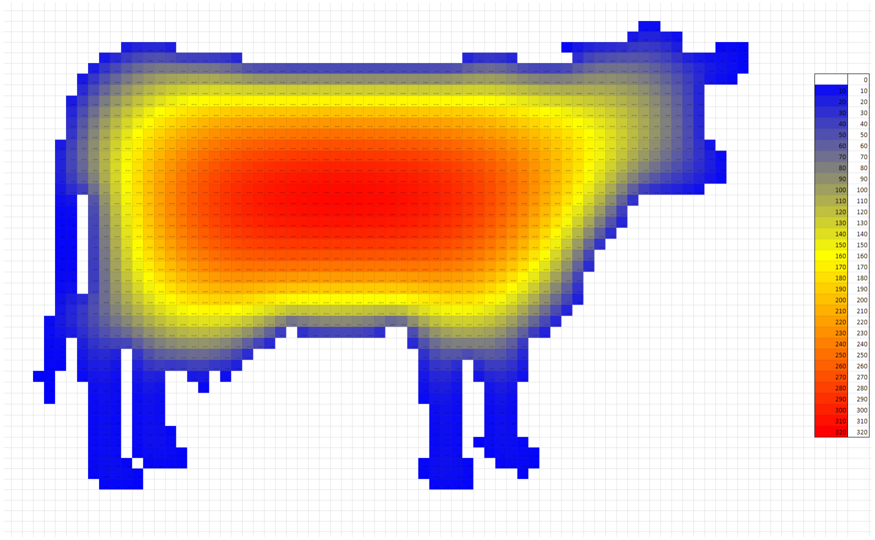

Figure 2: Cow created from Jacobi method

## 5.2 Jacobi

Jacobi is a simple yet powerful method for solving this problem as its simplicity allows for it to be massively optimized by the c++ compiler even after some added optimizations. This means that it can find a solution quickly. It also uses very little memory, meaning it can run on very large problems without demanding too large of a footprint. The cost of this is the solution matrix isn't exact and is prone to a degree of error.

## 6 Notes

Originally the Jacobi method was going to be the Jacobi overrelax method where you iterate over the space like a checkerboard where you go over the black cells first and then the red cells. This makes it so that every cell is only updated after its neighbors have been updated and even made room for a few other small optimizations like reducing the total needed checks for cells falling within the degree of error. This reduces the total number of iterations and does a decent job of speeding it up. It does not, however, do a better job than the c++ compiler using the -O3 flag and so it was later replaced with its simpler version, the Jacobi method. It was found that the simpler Jacobi method was almost one and a half times as fast as the overrelax version.