A COMPARISON OF DIFFERENT LAPLACE EQUATION SOLVERS

SARATHCHANDRAN J | ECS 310 | November 2, 2019

THE LAPLACE EQUATION

The Laplace equation is a 2nd order differential equation and a special case of elliptical differential equations. Generally written as

$$\nabla^2 V = 0$$

is expanded as

$$\frac{\partial^2 V}{\partial x^2} + \frac{\partial^2 V}{\partial y^2} + \frac{\partial^2 V}{\partial z^2} = 0 \quad \text{, in 3D}$$

This equation finds numerous applications in fluid dynamics, thermal physics and electrodynamics. So it is advantageous to have a method of solving this equation in a given region in space. My project looks into various methods of solving the Laplace equation given the Dirichlet boundary conditions and provide an analysis on the advantages and disadvantages of each method.

THE LAPLACE EQUATION AS A SYSTEM OF LINEAR EQUATIONS

First consider a discrete space of an $n_x n$ grid on which we have to solve the Laplace equation. We are given the values of the function at the boundary of this grid, this leaves us with $(n-2)^2$ points on the grid at which we have to find the value of the function. Let the distance between any 2 points be h and $V_{i,j}$ represent the value of the function at a general point (i,j). Now we have,

$$\frac{\partial^2 V}{\partial x^2} \approx \frac{V_{i+1,j} - 2V_{i,j} + V_{i-1,j}}{h^2} \quad \text{eq1} \qquad \frac{\partial^2 V}{\partial y^2} \approx \frac{V_{i,j+1} - 2V_{i,j} + V_{i,j-1}}{h^2} \quad \text{eq2}$$

Which when combined gives us

$$V_{i,j} \approx \frac{V_{i+1,j} + V_{i,j+1} + V_{i-1,j} + V_{i,j-1}}{4} \quad \text{eq3}$$

This forms the system of $(n-2)^2$ linear equations. Since there are $(n-2)^2$ equations and $(n-2)^2$ variables we can use Gaussian Elimination to solve and get the answer.

WRITING THE SYSTEM OF LINEAR EQUATIONS AS A MATRIX – AN EXAMPLE

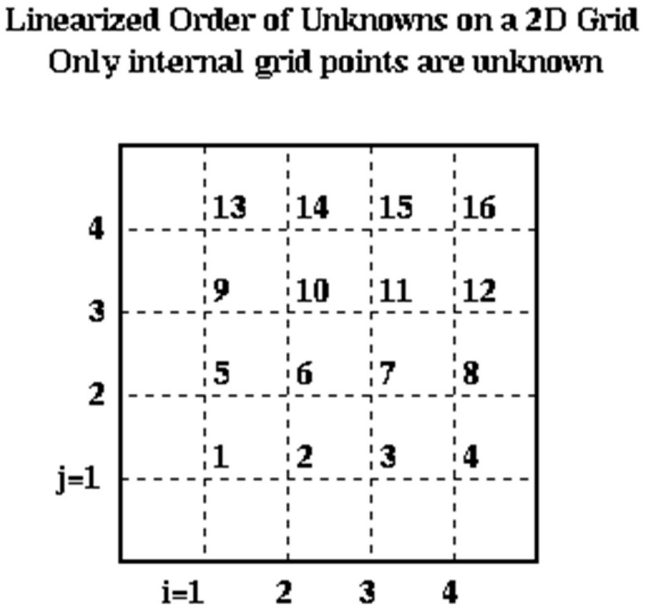

Fig 1[1]

Consider the above 4_x4 grid of which we have to solve the Laplace equation. The equation derived in the previous section may be represented as $A*U_{i,j} = b_{i,j}$, with the A matrix written as below if we follow the numbering as shown above.

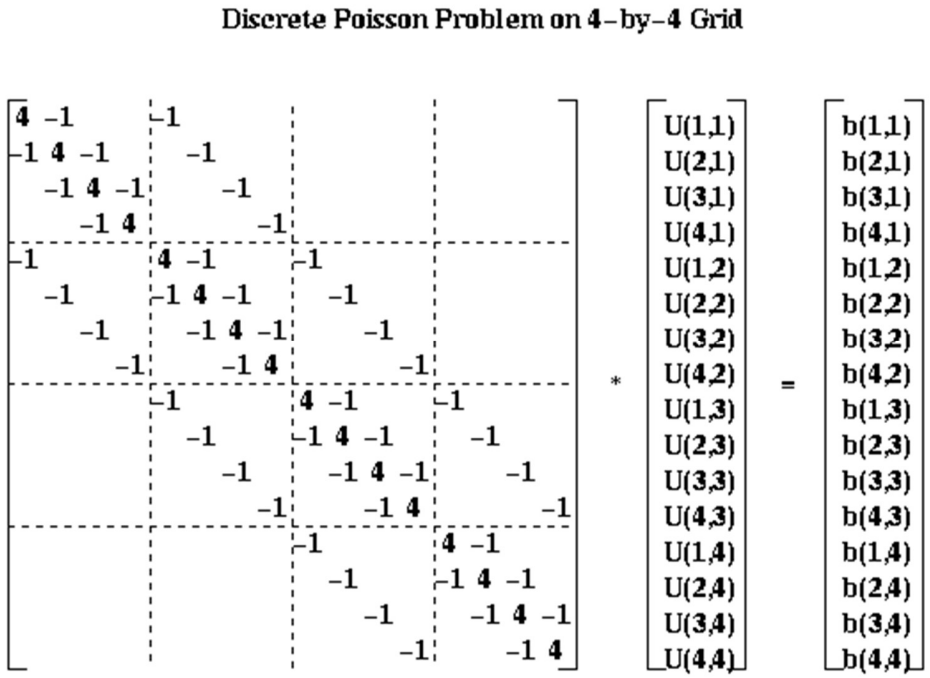

Fig 2

This method is extended for larger grids.

THE SETUP FOR THE TEST

To compare, all algorithms were used to solve the same boundary condition Laplace equation given below,

x_start = 1 x_stop = 100

y_start = 1 y_stop = 100

h = 1

BOUNDARY CONDITION:

(x, 1) = 0 (x, 100) = 0

(1, y) = 0 (100, y) = 10

SOLUTION GRAPH:

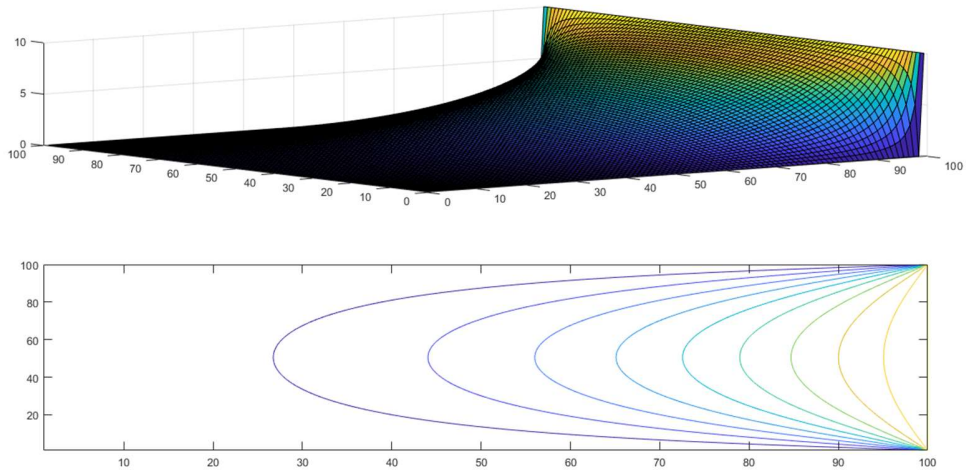

Fig3

GAUSSIAN ELIMINATION

Gaussian elimination also called row reduction algorithm involves performing row transformations on the matrix A and the corresponding terms of the vector b so as to transform A into an upper triangular matrix. This new matrix can now be solved by back substitution.

When sparse matrices are encountered to avoid the error that could be caused by division by zero and the round off errors when near singular matrices are encountered pivoting/partial pivoting is usually implemented. In this project partial pivoting was implemented which involves determining the element of the column with the maximum absolute value and performing row transformations to bring those elements to the diagonal positions. One major drawback to this is that the number of operations it has to do drastically increases with the size of the grid. Thus to reduce the load, rather than swapping the rows themselves a vector was made to keep track of the row numbers and these vector elements underwent the swapping. So in the

implementation for Gaussian Elimination with partial pivoting done, all rows are indexed using this indexing vector.

Despite these improvements the Gaussian Elimination proved to be a very slow algorithm requiring more than 1 minute to solve the system of linear equation. It is also a memory intensive algorithm as the entire matrix has to be stored which in this case was a $100^2 x 100^2$ matrix.

METHOD OF RELAXATION

Method of relaxation is an iterative process. It starts with an initial guess for the solution in this case the zero vector. Then it proceeds to update each element of the grid using eq3. This process is repeated until the entire grid "relaxes" into a distribution which will be the solution.

In a 1D case the loop would start from 1 end updating each successive term until it reaches the end at which point the entire process is repeated until the changes made during each update is lesser than a predefined threshold value (10^{-6}). The graph so obtained is shown below for the boundary condition f(0) = 0, f(10) = 1.

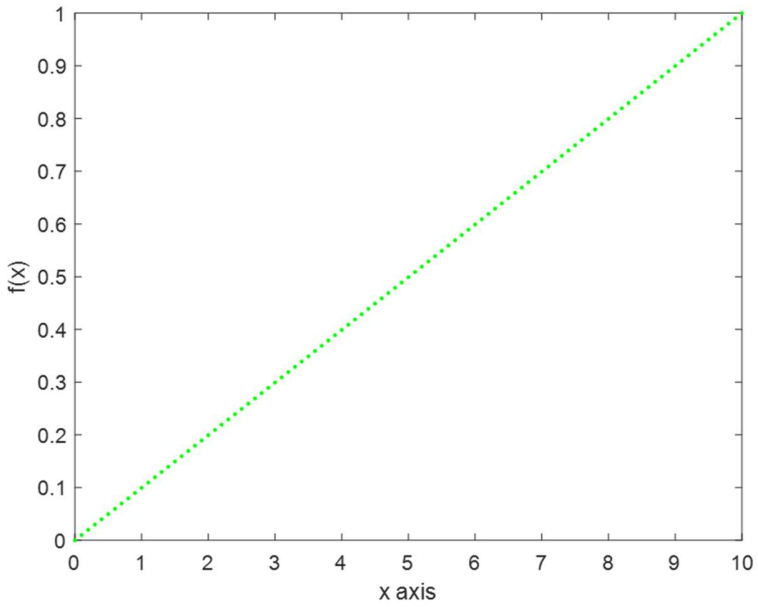

Fig4

This method is a great improvement over Gaussian Elimination and completed the 2D test in an average of 7.7s (including plotting). It took 8280 iterations to attain the threshold set. Compared to Gaussian Elimination this process is also less memory intensive, requiring only a 100_x100 grid.

IMPROVING THE RELAXATION METHOD: CHESS BOARD ALGORITHM

An improvement to the relaxation method could be made by changing the order in which the elements were updated. If we were to imagine the entire grid space to be like a chess board, (alternate black and white) consider updating all the white elements first and then the black elements. The advantage is that all the white elements will depend only on the black elements to calculate the average and vice versa. This means once the white elements are updated now the black elements could be updated using the new white values (Rather than using the old white values as in the previous algorithm). This results in the grid relaxing in lesser number of iterations.

For the test condition the convergence was achieved in 8255 steps, with an average runtime of 7.56s. Another advantage with this algorithm is that both the white update and black update are completely independent processes. This opens up the opportunity to parallelize the process.

SUCCESSIVE OVER RELAXATION

Successive Over Relaxation (SOR) is similar to the relaxation methods above. Consider eq3

$$V_{i,j} \approx \frac{V_{i+1,j} + V_{i,j+1} + V_{i-1,j} + V_{i,j-1}}{4} \quad \text{eq4}$$

This can be rewritten as

$$V_{i,j} \approx V_{i,j} + \frac{V_{i+1,j} + V_{i,j+1} + V_{i-1,j} + V_{i,j-1}}{4} - V_{i,j} \text{ eq5}$$

Or

$$V_{i,j} \approx V_{i,j} + correction\ factor \text{ eq6}$$

So multiplying the correction factor by ω will accelerate the rate of convergence. But determining a proper ω becomes a challenge as underestimating it will result in slow convergence while overestimating will result in us overshooting the convergence point again resulting in a slower convergence. Reference [3] provides a compact way to calculate the over relaxation factor ω.

$$\omega = \frac{2}{1 + sincos^{-1}\left(\frac{\cos\left(\frac{\pi}{m}\right) + \cos\left(\frac{\pi}{n}\right)}{k}\right)} \text{ eq7}$$

K = dimension of space, m = length along x axis, n = length along y axis

This algorithm ran the test in 290 iterations, with an average run time of 0.55s. This is by far a great improvement over the previously mentioned methods.

CONJUGATE GRADIENT METHOD

Conjugate gradient method(CG) is suitable for solving any linear system where the coefficient matrix A is both symmetric i.e. A = A' and positive definite. Two equivalent definitions of positive definite are given below

- All of A's Eigen values are positive
- x' * A * x >0 for all non-zero x

The algorithm goes as follows (referenced from[5])

```
x = initial guess for inv(A)*b
r = b - A*x   ... residual, to be made small
```

```
p = r          ... initial "search direction"
repeat
   v = A*p     ... matrix-vector multiply
   a = ( r'*r ) / ( p'*v )    ... dot product
   x = x + a*p                ... compute updated solution
   new_r = new_r - a*v        ... compute updated residual
   g = ( new_r'*new_r ) / ( r'*r )         ... dot product
   p = new_r + g*p    ... compute updated search direction
   r = new_r
until ( new_r'*new_r small enough )
```

CG maintains 3 vectors at all time the approximate solution x, its residual r and search direction p. At each step x is improved by searching for a better solution in the direction p yielding an improved solution x+a*p. The basic idea is to perform a gradient dissent on a measure of error given by sqrt(r'*inv(A)*r) , hence p is called the gradient in this algorithm.

This algorithm completed the test in an average time of 0.32s. It stored the matrix as a sparse matrix thus requiring the least data storage among the algorithms discussed until now.

CONCLUSION

Through this project I aimed to try some of the diverse approaches developed in solving the Laplace equation. I started with Gaussian Elimination but soon found that this general method is highly inefficient. Much more refined algorithms taking advantage of the sparse nature of the A matrix of the system of linear equations made by the Laplace equation was required. Granted some of these specific algorithms will not work on other matrices, but the frequency with which we encounter the Laplace equation makes it imperative to develop a quick and stable algorithm to solve the Laplace equation.

Through a comparison of all the algorithms that I did go through I found the CG method and SOR method to be the fastest* with the CG method also being able to store the data as a sparse matrix. Just a direct

comparison of the time taken for calculation for Gaussian Elimination (over a min) and CG shows that often there is an enormous payoff for using an algorithm specially tuned for the system at hand.

* There are faster algorithms like Fast Fourier Transform (FFT) and Multigrid Cycle which could perform better which haven't tried

REFERENCES

[1] https://people.eecs.berkeley.edu/~demmel/cs267/lecture17/lecture17.html

[2] *Numerical Recipes in C++.*

[3] https://pdfs.semanticscholar.org/6955/de9b4c6554a0d8a1ce8c7c06bfad7b4d1918.pdf

[4] http://www.damtp.cam.ac.uk/lab/people/sd/lectures/nummeth98/pdes.htm

[5] https://people.eecs.berkeley.edu/~demmel/cs267/