HW2 Assignment

Mike Sutherland

April 21, 2022

1 Problem 1

We can separate the equation into two directions, the x direction and the y direction. For any point, we have

$$\frac{T(x_0 + h, y_0) - 2T(x_0, y_0) - T(x_0 - h, y0)}{h^2}$$

We can represent our temperature with a 2-dimensional grid, with spacing in the x, y as h. Then, we can rewrite our equation:

$$\frac{T_{i+1,j} - 2T_{i,j} + T_{i-1,j}}{h^2}$$

And the y-direction becomes:

$$\frac{T_{i,j+1} - 2T_{i,j} + T_{i,j-1}}{h^2}$$

We can see that each equation has 4 components, corresponding to neighboring cells. Therefore, we expect each temperature to be related to, at a minimum, the temperature of two other cells (such as a cell on a corner boundary), and at most, the temperature of 4 other cells (such as an interior cell). We can simplify again and combine both directions to yield

$$1/h^2 \left(T_{i+1,j} + T_{i,j+1} - 4T_{i,j} + T_{i-1,j} + T_{i,j-1}\right) = 0$$

This equation gives us insight into what each row of our matrix will look like: the matrix encodes how the temperature of each cell is related to every other cell, including the boundaries. The matrix A we construct will be an $n \times n$ matrix, where n is the number of cells in the grid. In our case, the A matrix is $n = 9$ yields 9×9 matrix, for 9 cells. Carrying out each relation, we find the

following matrix:

$$A = \begin{bmatrix} 4 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 4 & -1 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 4 & -1 & 0 & -1 & 0 & 0 & 0 \\ -1 & 0 & -1 & 4 & -1 & 0 & -1 & 0 & 0 \\ 0 & -1 & 0 & -1 & 4 & -1 & 0 & -1 & 0 \\ 0 & 0 & -1 & 0 & -1 & 4 & -1 & 0 & -1 \\ 0 & 0 & 0 & -1 & 0 & -1 & 4 & -1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & -1 & 4 & -1 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & -1 & 4 \end{bmatrix} \tag{1}$$

This matrix is then multiplied by a 9×1 length vector T, which is the temperature array flattened[1] from 2D into 1D:

$$T = \begin{bmatrix} T_{1,1} \\ T_{1,2} \\ T_{1,3} \\ T_{2,1} \\ T_{2,2} \\ T_{2,3} \\ T_{3,1} \\ T_{3,2} \\ T_{3,3} \end{bmatrix} \tag{2}$$

Finally, we encode the boundary conditions with a 9×1 length vector b, which is the boundary array flattened from 2D into 1D. We construct this vector by subtracting right and top boundaries (those correspond to $+h$ in the x and y direction) and by adding the left and bottom boundaries (corresponding to $-h$ in the x and y direction). We populate this vector by looking at the corresponding cell of that row:

$$b = \begin{array}{rcll} T_{1,1} & \rightarrow & -\text{left} & -\text{bottom} \\ T_{1,2} & \rightarrow & -\text{left} & 0 \\ T_{1,3} & \rightarrow & -\text{left} & +\text{top} \\ T_{2,1} & \rightarrow & 0 & -\text{bottom} \\ T_{2,2} & \rightarrow & 0 & 0 \\ T_{2,3} & \rightarrow & 0 & +\text{-top} \\ T_{3,1} & \rightarrow & +\text{right} & -\text{bottom} \\ T_{3,2} & \rightarrow & +\text{right} & 0 \\ T_{3,3} & \rightarrow & +\text{right} & +\text{top} \end{array} \tag{3}$$

Because the values of boundaries are moved to the opposite side of the equation, in the b vector, they are negated. Thus, our system can be constructed, and solving the equation

$$[A]T = b \tag{4}$$

[1]Depending on how our temperature array is stored in memory, (row-major or column-major), it may be advantageous to flatten by columns instead of by rows. In that case, the next step changes, since it is reliant on the ordering of the temperature vector. The change we make to A is the transpose, so its structure is unaffected, because it is symmetric.

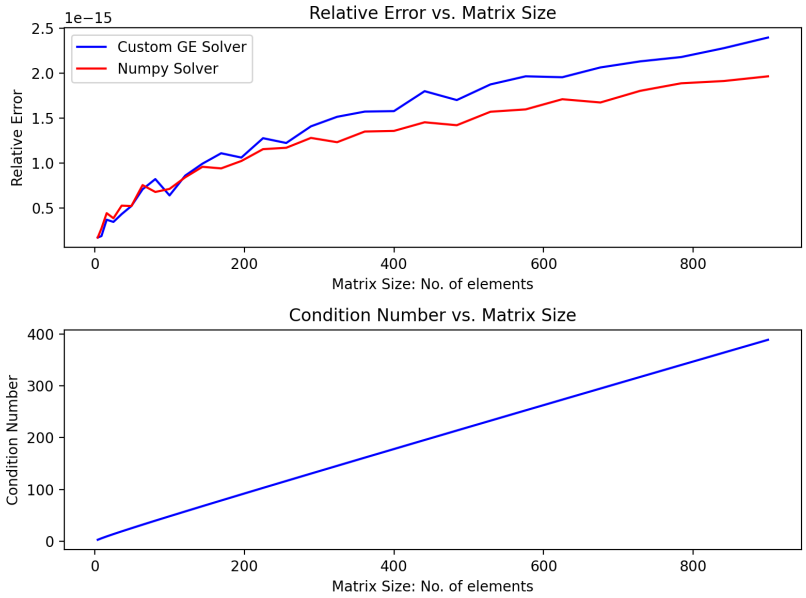

Figure 1: Relative Error of the solution (Top) and Condition Number of the A-Matrix (Bottom) as a function of the number of matrix elements. The relative error likely scales with the square root of the number of matrix elements (i.e. with n), while the condition number scales linearly with the number of matrix elements (i.e., with $n \times n$.)

With boundary conditions (b) and the system relationship (A) known, we solve the system for T and unpack it to arrive at the temperature at each grid points.

2 Problem 2 - 4

We notice that matrix A (eq. (1)) is diagonally-dominant: that for a given row, the absolute value of the diagonal is 4, while the sum of the absolute values of the off-diagonals is, at most, 4. To answer Problem 2 - 4, we solved the same problem for various matrix sizes, from $n = 2$ to $n = 31$. Because the custom Gauss-Elimination solution has $\mathcal{O}(n^3)$ time complexity, we did not exceed $n = 31$, because the solve time increased precipitously. The relationship between relative error and number of matrix elements appears to follow a square-root relationship: the relative error scales linearly with the number of equations being solved, n. Meanwhile, the condition number of the A matrix scales with the number of elements in the matrix. To verify our solver, we can build a heatmap of temperatures. First, we verify our solver by checking its solution against the example problem in [3, p. 871]. This is shown in fig. 2. Our solution scales well for larger cell counts, as we can see in fig. 3 – except runtime, of

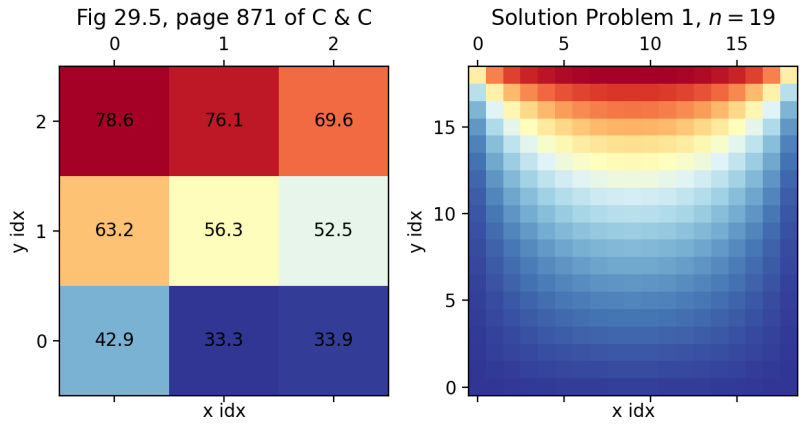

Figure 2: Left: Heatmap of the solution to the problem in [3, p. 871]. Right: Heatmap of the solution to problem 2 for $n = 19$.

course: solving for the 31×31 matrix takes about 3 minutes for a 2015 MacBook Pro.

2.1 Runtime

As we expect from a Gauss Elimination solver, runtime is $\mathcal{O}(n^3)$, where n is the number of matrix elements. This naive implementation is not particularly efficient. We can improve performance by exploiting the fact that the matrix is diagonally dominant and sparse, which would allow us to use an $\mathcal{O}(n^2)$ algorithm like Gauss-Seidel to find the solution. In fig. 4, we look at the solution time for our algorithm, vs. the solution time for the equivalent `numpy.linalg.solve` routine for the same matrix. With some digging [2], we discover that numpy uses the LAPACK solver, which is written in FORTRAN 77. Numpy uses a version of LAPACK maintained by Intel, which uses fast iterative methods on single precision, and only falls back to double precision if solutions cannot be computed. These routines are also optimized for the x86 architecture. This means that the solver is much faster than the naive implementation. [1]

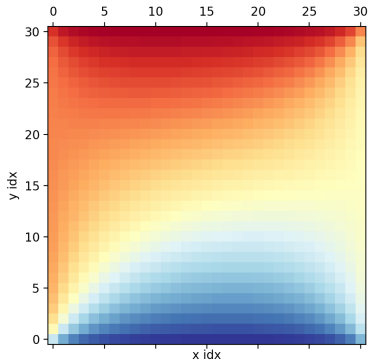

Figure 3: Heatmap of the solution to the problem in [3, p. 871], but scaled to $n = 31$.

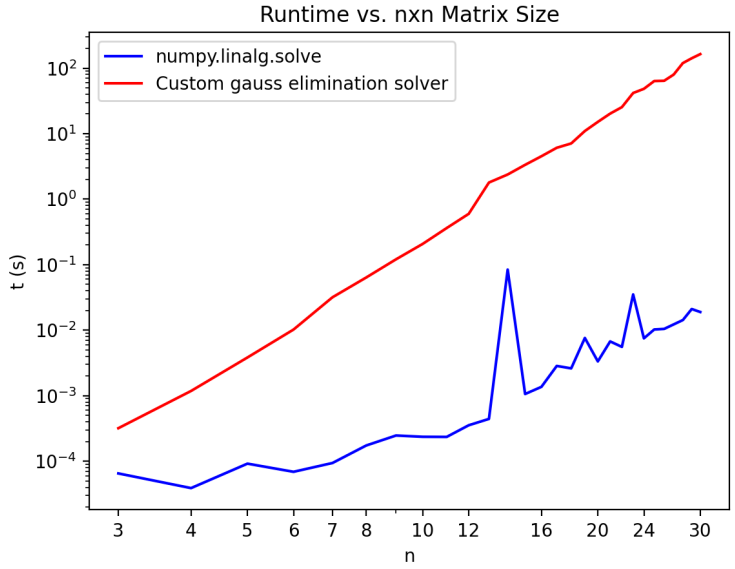

Figure 4: Runtime of the solver (y-axis) vs. the n of $n \times n$ matrix (x-axis). Both axes are log scale, which clearly shows the $\mathcal{O}(n^3)$ of the Gauss Elimination solver.

3 Source Code

The source used to generate plots in this report, as well as the LaTeX source, can be found at `https://github.com/rland93/MAE195` in the "HW2" folder.

References

[1] *?Gesv*. Intel. URL: `https://www.intel.com/content/www/us/en/develop/documentation/onemkl-developer-reference-c/top/lapack-routines/lapack-linear-equation-routines/lapack-linear-equation-driver-routines/gesv.html` (visited on 04/22/2022).

[2] ali_m. *Answer to "Why Does Numpy.Linalg.Solve() Offer More Precise Matrix Inversions than Numpy.Linalg.Inv()?"*. Stack Overflow. July 7, 2015. URL: `https://stackoverflow.com/a/31257909/9344822` (visited on 04/22/2022).

[3] Steven Chapra and Raymond Canale. *Numerical Methods for Engineers*. 8th edition. New York, NY: McGraw Hill, Mar. 3, 2020. 1008 pp. ISBN: 978-1-260-23207-3.