Project 2, Elliptic Problem

Miki Ivanovic

March 2020

1 Part 1

1.1 Preface

There are two approaches for implementing these iterative methods, the first is the **matrix** approach wherein we have some equation $Au = b$ that we iteratively solve for u. Generally for this problem this would require a penta-diagonal matrix A, and a column vector u mapping the grid points to our approximated solution for ϕ.

My initial hunch was that when we have a large grid size ($L \times N$) the matrix A will take up a lot of space in memory, and that it would also be a pain to construct matrix given its shape and size. Instead I opted for a more direct approach where the grid is stored directly in memory, and we perform iteration steps with a "**neighbours**" abstraction.

Note that since we are using neighbours, I use (n)orth, (e)ast, (s)outh, (w)est to refer to neighbours around the (c)urrent point. In this context north refers to the positive y-direction, and east to the positive x-direction

All of the iteration algorithms have a lot of similarities (or duplication), they only differ at the moment when U^{j+1} is calculated, therefore had my Gauss-Seidel algorithm inherit from my Jacobi scheme, and the SOR inherit from GS (since they both require dynamic programming).

1.2 Boundary Conditions

The problem has Neumann boundary conditions, at the north wall ($y \to \infty$) we have $d\phi/dy = 0$, on the east and west walls ($x \to \pm\infty$) we have $d\phi/dx = 0$, and on the south wall ($y = 0$) we have the same condition as the north wall except for the interval $0 \leq x \leq 1$ (at the aerofoil) where we have the condition $d\phi/dy = (1 + d\phi/dx)dy_b/dx$ (given by condition (6) in the paper as $\tau << 1$).

To actually apply these conditions, we create an imaginary point beyond the grid to apply a first order central difference (key since it has the same truncation error as the second order central difference). For example, say we are at a point $U^j_{m,n}$ where m is the x-axis index, n the y-axis index, and j the current *iteration* level. Suppose that this point is at the north wall. U clearly

has no index $n+1$ since it is only defined up to n, but we still define this point - namely as $U^j_{m,n+1} = U^j_{m,n-1}$ so that the first order central difference

$$\frac{U^j_{m,n+1} - U^j_{m,n-1}}{h_y} = 0$$

thereby satisfying the boundary condition. Then we use this new point in our Laplace equation to obtain a new approximation for $U^{j+1}_{m,n}$ with

$$U^{j+1}_{i,j} = \frac{h_x^2 h_y^2}{2} \times \left(\frac{U^j_{m+1,n} + U^j_{m-1,n}}{h_x^2} + \frac{U^j_{m,n+1} + U^j_{m,n-1}}{h_y^2} - f_{m,n} \right)$$

where here we have the Jacobi scheme, $f_{m,n}$ (always 0 for part 1) is the expected value, h_x is the change in x units, and h_y is the change in y units. Note that each U on RHS is a neighbour of our current point $U^j_{m,n}$:

- $U^j_{m,n+1}$ is the **north** neighbour
- $U^j_{m+1,n}$ is the **east** neighbour
- $U^j_{m,n-1}$ is the **south** neighbour
- $U^j_{m-1,n}$ is the **west** neighbour

This means that the above equation can be redefined in terms of the neighbours. In the given example at the north wall, the north neighbour (imaginary) is set to be equal to the south neighbour in order to meet the boundary condition, and subsequently be used in the Laplace equation. Similarly at the east wall, the east neighbour (imaginary) is set to the west neighbour to meet the boundary condition, and to be used in the Laplace equation - and this is extended to all four walls. By writing a **get_neighbours** function which returns the set of neighbouring points, and at the boundary applies the described assignments, almost all the Neumann boundary conditions are encapsulated naturally with this abstraction, even at the corners of the grid where two boundary conditions are applied. This also works in the G-S and SOR schemes where U^{j+1} is passed to get_neighbours rather than U^j (as it is initially an exact copy). The only one left to deal with is the boundary condition at the aerofoil.

1.2.1 Aerofoil Boundary

The aerofoil boundary condition is applied at $y = 0$, $0 \leq x \leq 1$ and is given by

$$\frac{U^j_{m,n+1} - U^j_{m,n-1}}{2h_y} \approx \frac{d\phi}{dy} = \left(1 + \frac{d\phi}{dx}\right) \frac{dy_b}{dx}$$

At the south wall we need to define the south neighbour in such a way that it satisfies this condition. We first approximate $d\phi/dx$ using a first order central

difference with the east and west neighbours. Then calculate dy_b/dx as it is a known derivative ($2\tau(1-2x)$) and the x value is always known (since we know where we are on the grid and how to map to the $x-y$ plane). With this the last equation can be rearranged as

$$U^j_{m,n-1} \approx U^j_{m,n+1} - 4\tau h_y(1-2x)\left(1 + \frac{U^j_{m+1,n} - U^j_{m-1,n}}{2h_x}\right)$$

to give the south neighbour for use in the Laplace equation.

1.3 Convergence Criteria

Grid independence occurs when the approximations U^j converge element-wise to the true U we are trying to find. This means that each $U^j_{m,n} \to U_{m,n}$ as $j \to \infty$.

In order for the algorithms to converge, the spectral radii of $B^{-1}C$ where $A = B+C$ must be less than 1. Since I don't use matrices in my implementation it would be too complex to manually compute the eigenvalues, however it is known that if the matrix A is diagonally dominant, then the algorithm will converge. Since the diagonals are $-2(1/h_x^2 + 1/h_y^2)$ and the rest of the matrix (in general) are multiples of either h_x or h_y, then A will be diagonally dominant so long as the *step size* in each direction is sufficiently small. This means in the context of my algorithms, we must have $L >> s+q$ and $N >> r$ in order to achieve convergence in the first place.

Now, given that good choices are made for these parameters, how can we actually tell that our solution is converging in a discrete system? We need to fix a tolerance **rtol** to compare each iteration to. We would also ideally get the error from each iteration and check it falls within our **rtol**, however the problem is that we don't have the true U in the first place - therefore I used the residue instead.

After each U^j has been computed, the residue $r^j = b - Au^j$ is calculated as well. Again, since matrices aren't used, the r^j (matrix in this case where each $r_{m,n}$ corresponds to the residue at $U_{m,n}$) is calculated with the neighbours abstraction described above. Then I take the euclidean norm of r^j and check that it is less than **rtol** (ϵ), and if it is, then we know that $r^k < \epsilon, k > j$ since each successive U^k gets closer to the true U.

1.4 Boundary

The boundary conditions in this discretized system apply at $y = r$, $x = -q$ and $x = s$ which are meant to represent ∞. There is a balance to be struck between having large (q, r, s) values, and the computational efficiency they result in.

Ideally these values would be some order of magnitude larger than the aerofoil boundary, however this would require testing on large grids which would take too long to run on my machine. Since the aerofoil is being approximated at $y = 0$, r could be fairly small. Moreover since the length of the aerofoil is 1 x

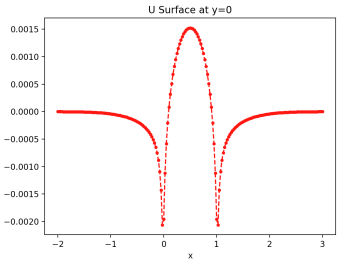

Figure 1: Jacobi U surface plot

unit, I figured that there should be at least 1 x unit of distance either side from the aerfoil to each wall. Therefore $(q, r, s) = (2, 2, 3)$ to keep the computations reasonable short, but to also maintain the integrity of the boundary conditions. I keep these values fixed for all the experiments detailed in the rest of this report, however these parameters can easily be set to any arbitrary natural number if desired (albeit requiring longer computation time).

1.5 Numerical Efficiency

The main indicator of numerical efficiency is the number of iterations required to produce a result. Moreover, since each of the initial schemes needs to update each value in the grid, each computation step at every iteration is $O(LN)$, therefore it naturally takes longer to run the program for larger grids. Since my machine isn't very powerful, I only conduct my tests at small grid sizes, but with enough variance in the L, N values that the results should extend to larger grid sizes. This means that for part 1-3, the main indicator of computer timing are the iteration counts.

In my program I also set a limiter on the number of iterations a convergence scheme is allowed to run in order to "cancel out noise" and to improve computation speeds, particularly when running multiple convergence tests in one go. By default it is set to 1000 iterations, but it can be changed to any limit. This feature also proved useful in my algorithm for the Multi-Grid method where we require a fixed number of G-S sweeps at each grid level.

1.6 Results

Each scheme was tested at the specified (q, r, s) values, at grid size $(201, 51)$ to obtain required U_s values. Tolerance tested at $\epsilon = 0.1$ for computer efficiency.

1.6.1 Jacobi

- $U\_s(x = 0.025) = -0.0011169807530781303$
- $U\_s(x = 0.25) = 0.0010567883773554372$

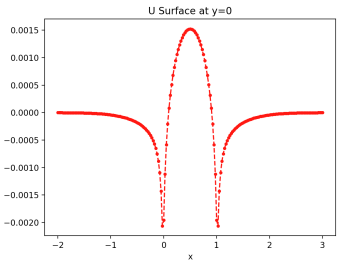

Figure 2: Jacobi V surface plot

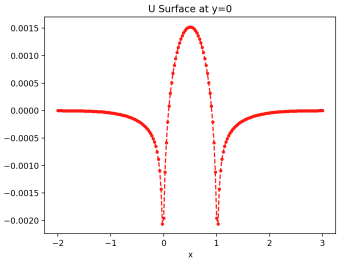

Figure 3: Jacobi vector field generated over (x,y) plane

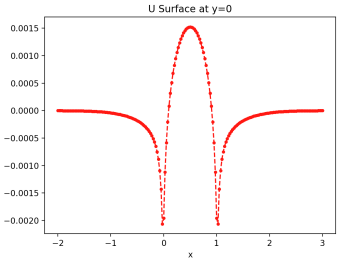

Figure 4: Jacobi ϕ surface over (x,y) plane

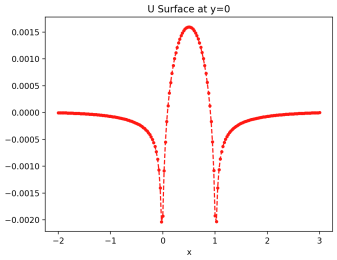

Figure 5: GS U surface plot

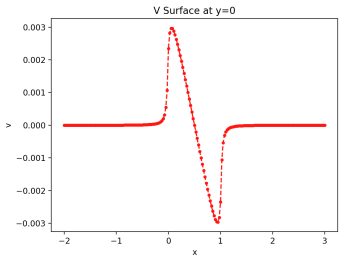

Figure 6: GS V surface plot

- $U\_s(x = 0.5) = 0.0015249477652748054$
- $U\_s(x = 0.75) = 0.0010567883773554354$
- $U\_s(x = 0.95) = -0.0005851249053919656$
- iterations required = 1001 (did not converge in time)

1.6.2 Gauss-Seidel (GS)

- $U\_s(x = 0.025) = -0.0010824348351403373$

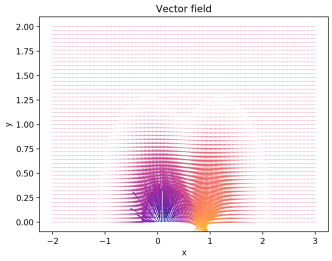

Figure 7: GS vector field generated over (x,y) plane

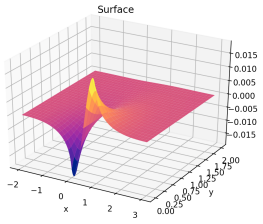

Figure 8: GS ϕ surface over (x,y) plane

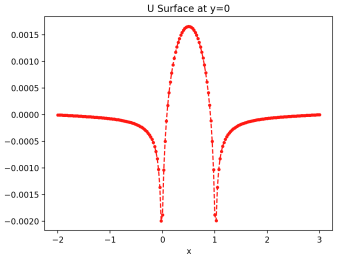

Figure 9: SOR U surface plot

- $U\_s(x = 0.25) = 0.001117856266412429$
- $U\_s(x = 0.5) = 0.0015985474396729283$
- $U\_s(x = 0.75) = 0.0011198998492811018$
- $U\_s(x = 0.95) = -0.0005439478027275384$
- iterations required = 1001 (did not converge in time)

1.6.3 Successive Over Relaxation (SOR) with $\omega = 1.7$

- $U\_s(x = 0.025) = -0.0010370581721185074$
- $U\_s(x = 0.25) = 0.0011758578078730972$
- $U\_s(x = 0.5) = 0.0016608779315560652$
- $U\_s(x = 0.75) = 0.001177164291044682$
- $U\_s(x = 0.95) = -0.0004973836063800682$
- iterations required = 643 (converged at given tolerance)

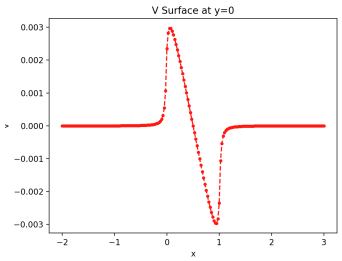

Figure 10: SOR V surface plot

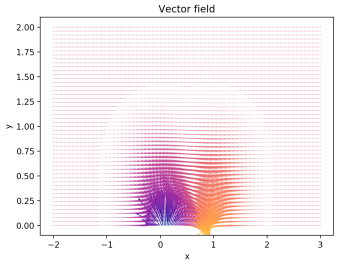

Figure 11: SOR vector field generated over (x,y) plane

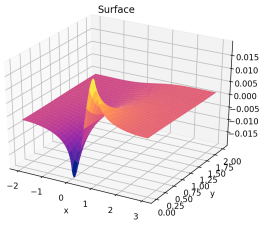

Figure 12: SOR ϕ surface over (x,y) plane

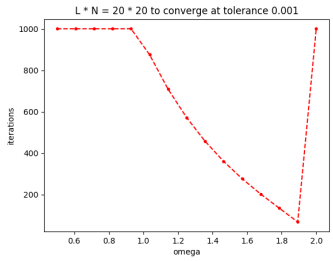

Figure 13: Convergence acceleration for grid (20, 20)

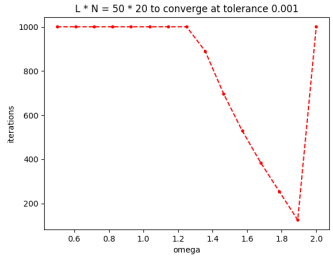

Figure 14: Convergence acceleration for grid (50, 20)

1.7 Analysis

Since the Jacobi and GS schemes did not converge in time, I ran another set of tests with $\epsilon = 0.01$ and grid size (51, 21). The Jacobi still required 1927 iterations to complete whereas the GS finished in 950, which is around half of that for Jacobi. SOR with $\omega = 1.7$ converged in 204 iterations at this tolerance, almost an order of magnitude faster than the Jacobi method.

2 Part 2

I tested my SOR implentation for $0.5 \leq \omega \leq 2$ over the grid spaces $(20, 20)$, $(50, 20)$ and $(50, 50)$ at $\epsilon = 0.001$. In these figures, I limited the number of iterations to 1000, however we still see the convergence acceleration around the optimal ω:

For each of my tests, the optimal ω was around 1.9 - which would suggest the spectral radii is close to 1 - and that it not necessarily linked with the grid size, or h_x and h_y values. Before this point, there is a parabolic decrease in the number of iterations as ω increases in the range $1 \leq \omega \leq \omega_o$. From $\omega_o \leq \omega \leq 2$ the iteration increases linearly.

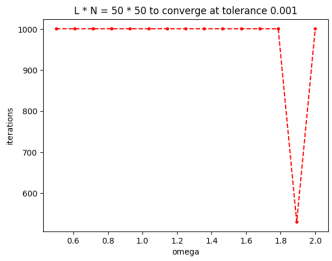

Figure 15: Convergence acceleration for grid (50, 50)

3 Part 3

My hypothesis is that in the new coordinate system, the boundary condition at $0 \leq x \leq 1$ disappears and the others stay the same. Furthermore the Laplace equation is transformed to

$$\frac{\partial^2 \phi}{\partial x^2}\left(1 - \frac{1}{4\tau^2(1-2x)^2}\right) + \frac{\partial^2 \phi}{\partial \eta^2} = 0$$

However I did not have time to compile any results or provide my derivation of this formula. The trick condition arises at $x = 0.5$ where we get a division by infinity, so at that point we just set to 0.

4 Part 4

For the Multi-Grid method, my implementation requires that each grid has form $2^p + 1$. This condition means that the interpolation and restriction functions map exactly onto each other. The implementation starts with a fine grid, and recursively smooths and restricts the residues at each step to the lowest level when at least one of the grid dimensions is 3. It then interpolates the coarsest grid back to the original fine grid giving our approximation for U.

The numerical efficiency gain over the convergence schemes in part 1 is astonishing. By smoothing with a fixed number of GS iterations at each level means that even though the GS computation scales with grid size, the low iterations means that there aren't many computations overall. Moreover at each level the size of the grid is roughly quartered, so the GS sweeps get exponentially faster.

At each step in the algorithm there are 2×10 GS sweeps, with a total of $\mathbf{min}(\log_2(L) - 1, \log_2(N) - 1)$ levels. This means that for a grid of size (257, 65) (which is slightly larger than those used in part 1) the number of iterations is $20 \times \log_2(65) - 1 = 100$ iterations, but even then not all of these iterations are equal. In practise, this means that the time taken for the computation to finish is orders of magnitude lower than that of any of the other schemes.

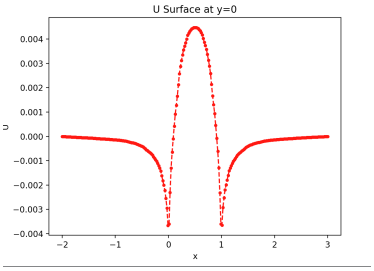

Figure 16: MG U surface plot

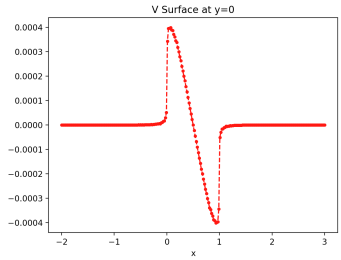

Figure 17: MG V surface plot

In order to demonstrate this computational gain, I ran the Multi-Grid at $\epsilon = 0.001$ with a grid size of (257, 257). The program finished in 61.82630205154419 seconds whereas the others each took several minutes (and being cut off at 1000 iterations).

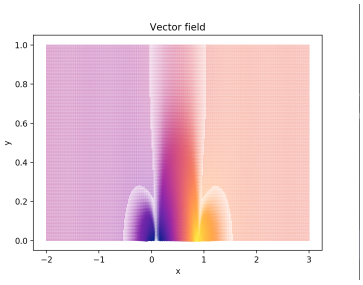

Figure 18: MG vector field generated over (x,y) plane

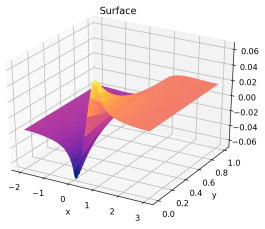

Figure 19: MG ϕ surface over (x,y) plane