Assignment 1 for ME4233

Author: Giovani Hidalgo Ceotto

Professor: Mengqi Zhang

Date: Nov. 5th, 2018

Introduction

Consider the following Poisson equation:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = g(x,y), \qquad g(x,y) = -8\pi^2 \sin[2\pi(x+y)] \tag{1}$$

$$x, y \in \Omega = [0,1]^2, \qquad \partial\Omega = 0$$

The objective of this assignment is to use a second order of accuracy with five point central finite difference scheme to discretize the above equation and use direct and iterative methods to solve the problem.

The direct linear algebraic system solvers considered are LU decomposition and QR factorization, while the iterative solvers employed are Jacobi, Gauss-Seidel and Successive Over Relaxation (SOR) based on Gauss-Seidel. The five methods were implemented with the Python language, in combination with Numpy and Scipy, both used to manage matrices and vectors, in dense and sparse format respectively. The Numba library was also used for just in time compilation with the objective of speeding up the implementation. Furthermore, corresponding implementations from standard libraries were also briefly tested to compare results and evaluation speed. To execute the code, a Jupyter Notebook was used, keeping everything organized.

The report starts by briefly discussing the code structure implemented. Subsequently, the resultant algebraic system obtained after the finite difference discretization is presented. Then, the solution output given by the direct solvers is shown followed by the solution output given by iterative methods. Finally, the effect of initial guesses on number of iterations needed for convergence of each iterative method is explored, along with the validation of the order of accuracy of the five point central difference scheme employed.

Code Structure

The code used to assemble the discretized equation into a linear algebraic system, solve the system and post process the solution was completely written in Python. Following Python conventions, the code is organized in modules, equivalent to libraries, which can be imported and executed.

The two modules created are:

- *poisson_equation_prepost*
- *linear_system_solver*

The first one takes care of pre and post-processing, by assembling the algebraic system of equation which results from the finite difference discretization of the Poisson equation (1) while also being in charge of reshaping the solution vector, combining it with the boundary conditions and plotting the overall solution.

This modules has the following functions:

- *poisson_equation_prepost*
    - *assemble_algebraic_system*
    - *assemble_algebraic_system_slow*
    - *post_process_solution*

Their names are self-explanatory, but detailed documentation about parameters and outputs is available by means of docstrings embedded in the code.

The second module is used to solve a generic linear system. It employs the following functions:

- *linear_system_solver*
    - Main wrapper
        - *solve_algebraic_system*
    - Auxiliary functions for direct methods
        - *solve_upper_triangular_system*
        - *solve_lower_triangular_system*
    - Direct methods
        - *LU_factorization*
        - *LU_factorization_sparse*
        - *LU_factorization_compiled*

        - *QR_factorization*
        - *QR_factorization_compiled*
    - Iterative methods
        - *jacobi*
        - *jacobi_compiled*
        - *jacobi_sparse*
        - *gauss_seidel*
        - *gauss_seidel_compiled*
        - *gauss_seidel_sparse*
        - *sor*
        - *sor_compiled*
        - *sor_sparse*

Once again, the names are self-explanatory but detailed documentation is available in the code as docstrings. It is important to note the suffixes used, such as *_compiled* and *_sparse*. All solvers, with the exception of QR, have three different implementations. The original one uses mainly dense matrices and for loops. The *_compiled* implementation is a slightly modified version of the original implementations to allow for its just in time compilation. The *_sparse* implementation makes extensive use of the sparse properties of the input matrix to speed up evaluation. As predicted, *_compiled* and *_sparse* methods are faster, even though all three implementations of the same method give the same result, up to floating point errors.

Both modules are then imported and used in a Jupyter Notebook, named '*Assignment 1 Notebook.ipynb*'. The functions described above are used extensively throughout the notebook with the intent of generating the results which will follow.

The notebook, and the source code for the modules, can be accessed via GitHub. Furthermore, the notebook can be viewed using Nbviewer or safely executed through a web browser, without any extra installations, by accessing the Jupyter Hub Binder. This is the encouraged method to run the code for this report, as its execution has been verified.

Finite Difference Discretization

The finite difference discretization scheme used is the five point central difference method, represented by:

$$\frac{\partial^2 u_{i,j}}{\partial x^2} = \frac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{(\Delta x)^2}, \qquad \frac{\partial^2 u_{i,j}}{\partial y^2} = \frac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{(\Delta y)^2} \tag{2}$$

Applying the scheme to the Poisson equation (1) results in:

$$\frac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{(\Delta x)^2} + \frac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{(\Delta y)^2} = g_{i,j} \tag{3}$$

For the inner interior points, which are surrounded by other interior points, it is useful to express the difference equation (3) as:

$$\left(\frac{1}{\Delta x^2}\right) u_{i-1,j} + \left(\frac{1}{\Delta x^2}\right) u_{i+1,j} + \left(\frac{-2}{\Delta x^2} + \frac{-2}{\Delta y^2}\right) u_{i,j} + \left(\frac{1}{\Delta y^2}\right) u_{i,j-1} + \left(\frac{1}{\Delta y^2}\right) u_{i,j+1} = g_{i,j} \tag{4}$$

For the edge interior points, Dirichlet boundary conditions are applied. Combining all grid points in a column vector results in the following linear algebraic equation, presented as an example for a grid discretised with 6×6 grid points with $\Delta x = \Delta y = h$:

$$A \cdot u = b \tag{5}$$

$$\frac{1}{h^2}\begin{bmatrix}
-4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & -4 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -4 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4
\end{bmatrix}
\begin{bmatrix} u_{1,1} \\ u_{1,2} \\ u_{1,3} \\ u_{1,4} \\ u_{2,1} \\ u_{2,2} \\ u_{2,3} \\ u_{2,4} \\ u_{3,1} \\ u_{3,2} \\ u_{3,3} \\ u_{3,4} \\ u_{4,1} \\ u_{4,2} \\ u_{4,3} \\ u_{4,4} \end{bmatrix}
=
\begin{bmatrix} g_{1,1} - u_{1,0} - u_{0,1} \\ g_{1,2} - u_{0,2} \\ g_{1,3} - u_{0,3} \\ g_{1,4} - u_{0,4} - u_{1,5} \\ g_{2,1} - u_{2,0} \\ g_{2,2} \\ g_{2,3} \\ g_{2,4} - u_{2,5} \\ g_{3,1} - u_{3,0} \\ g_{3,2} \\ g_{3,3} \\ g_{3,4} - u_{3,5} \\ g_{4,1} - u_{4,0} - u_{5,1} \\ g_{4,2} - u_{5,2} \\ g_{4,3} - u_{5,3} \\ g_{4,4} - u_{4,5} - u_{5,4} \end{bmatrix}$$

The main blocks of the matrix are colour coded to facilitate visualization.

Two approaches to construct the linear system in Python were implemented. The first one, performed by the *assemble_algebraic_system*, works by generating the 5 non-zero diagonals as vectors and then constructing a diagonal based sparse matrix. The second approach, performed by the *assemble_algebraic_system_slow* function, works by looping over all inner interior grid points and populating the appropriate spaces in each line of the matrix and then looping over the edge interior points to finish populating the matrix taking into account the boundary conditions. Both functions create the b vector equally by looping over the boundary

points. As the naming suggests, the first implementation is up to ten times faster for the considered grid resolutions in this report.

The output of both functions is the same and is presented in Figure 1.

```
Matrix A:
[[-14400.   3600.      0. ...      0.      0.      0.]
 [  3600. -14400.   3600. ...      0.      0.      0.]
 [     0.   3600. -14400. ...      0.      0.      0.]
 ...
 [     0.      0.      0. ... -14400.   3600.      0.]
 [     0.      0.      0. ...   3600. -14400.   3600.]
 [     0.      0.      0. ...      0.   3600. -14400.]]

Vector b:
[-16.41604911 -24.3990039  -32.1146381  ...  32.1146381   24.3990039
  16.41604911]
```

Figure 1 - Output of the 'assemble_algebraic_system' function.

Direct Solvers

The linear algebraic system assembled in the previous section was initially solved using two direct methods based on LU and QR factorization.

The compiled implementation of both methods were used since they presented the fastest evaluation speed when compared to the other implementations, namely sparse and non-compiled.

After the factorization algorithms were executed, the *solve_upper_triangular_system* and *solve_lower_triangular_system* functions were used to find the solution vector for the LU method, while the QR method required only the use of '*solve_upper_triangular_system*' function.

The calls to all of these functions were wrapped by the general function *solve_algebraic_system* of the *linear_system_solver* module.

As checked, the solution output of both methods were equal to absolute tolerance of 10^{-13}. Figure 2 and Figure 3 present the solution obtained by using the LU method and the QR method, respectively.

As can be observed, the solution respects the boundary conditions imposed, zero in this case, while presenting an oscillatory behaviour aligned to the $y = x$ line creating crests and troughs.

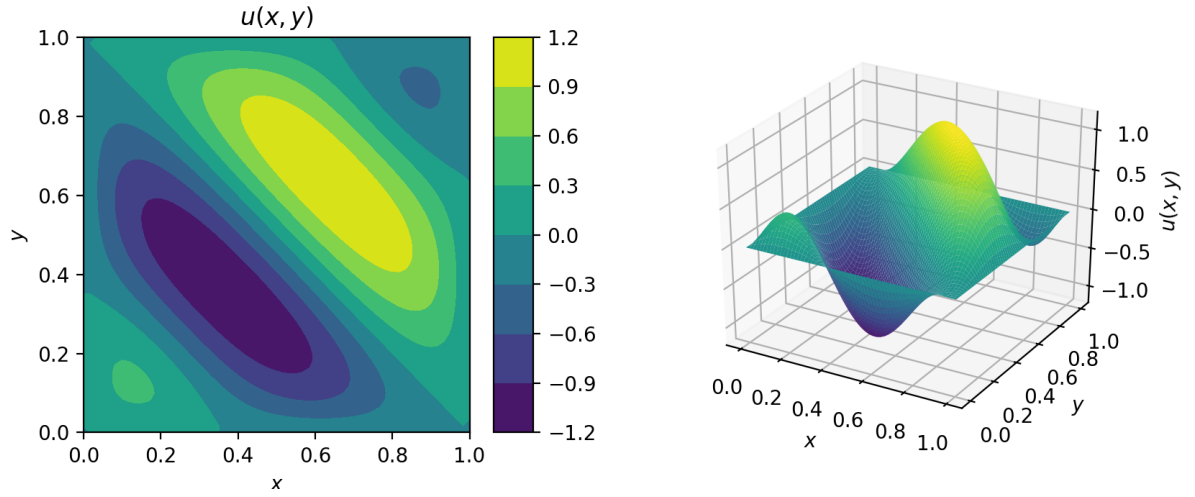

Figure 2 - Solution to the Poisson equation (1) obtained by LU factorization.

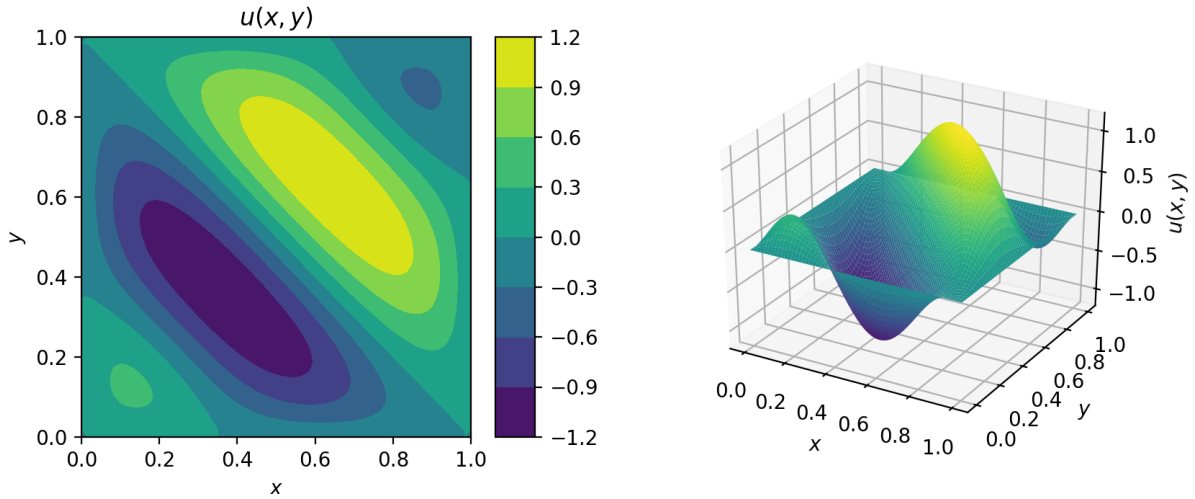

Figure 3 - Solution to the Poisson equation (1) obtained by QR factorization.

Indirect Solvers

The linear algebraic system which results from the finite difference discretization of the Poisson equation (1) can also be solved by making use of iterative solvers. In this work, three iterative solvers were implemented and used: Jacobi, Gauss-Seidel and Successive Over Relaxation based on Gauss-Seidel.

Each of the three methods were implemented in two formats, one making use of matrix operations while the other two looped over each item of the solution vector iteratively, with one of them being compiled. The matrix implementations were the fastest in terms of evaluation speed, specially due to the sparsity of the matrix considered. Therefore, the *jacobi_sparse*, *gauss_seidel_sparse* and *sor_sparse* methods were the ones mainly used.

The convergence criteria used for these iterative solvers is presented in equation (6):

$$r = \left|\left|u - u_{true}\right|\right| < 10^{-7} \quad (6)$$

Where u represents the vector which is being solved for, while u_{true} represents the solution obtained by the LU method.

The initial guess used was the same for all three methods and comprised of a random vector with values ranging from 0 to 1. The comparison of the number of iterations is shown in Figure 4.

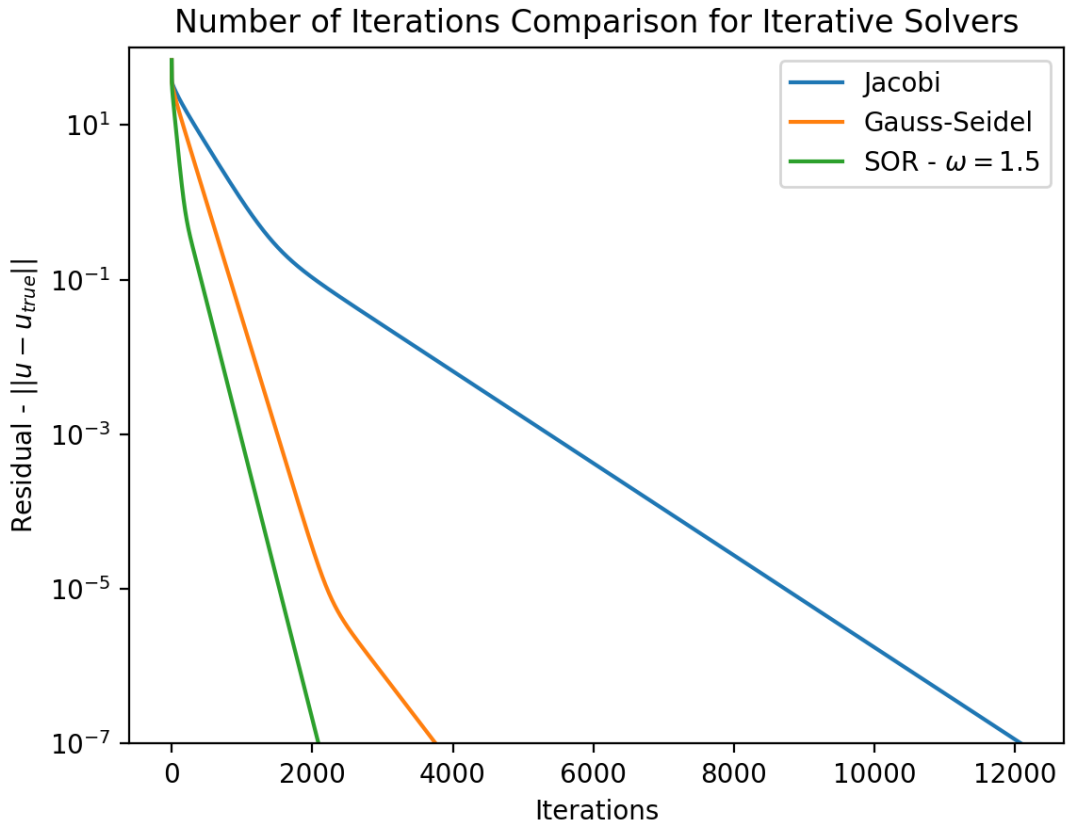

Figure 4 - Comparison of number of iterations for the convergence of Jacobi, Gauss-Seidel and Successive Over Relaxation method (using $\omega = 1.5$).

The graph shows that the Gauss-Seidel converges in significantly less iterations than the Jacobi method, in fact, using 66% less iterations. Furthermore, the Successive Over Relaxation method converges in even less iterations, just a little over 50% of the number o iterations needed for Gauss-Seidel.

Optimum ω for Successive Over Relaxation

It is widely known that Successive Over Relaxation converges for $0 < \omega < 2$ as long as the A matrix which defines the linear system is symmetric and positive definite, as is the case. Furthermore, it is also known that the optimum value of ω for the finite difference discretized

Poisson equation (1) in a $N \times N$ mesh with uniform grid spacing $h = \frac{1}{N+1}$ is given by equation (7):

$$\omega_{opt} = \frac{2}{1 + \sin(\pi h)} \tag{7}$$

Considering the case of $N = 61$, the optimum value of the relaxation parameter ω evaluates to:

$$\omega_{opt} = \frac{2}{1 + \sin\left[\pi\left(\frac{1}{1+N}\right)\right]} = 1.9005 \tag{8}$$

This value can be verified by solving the linear system with different values of ω and finding the point of minimum iterations. This was carried out and the results are presented in Figure 5 and Figure 6.

Figure 5 reveals that the number of iterations needed for convergence decreases significantly as ω increases from 0.4 to 1.9. For $\omega = 1.999$, the number of iterations raises once again.

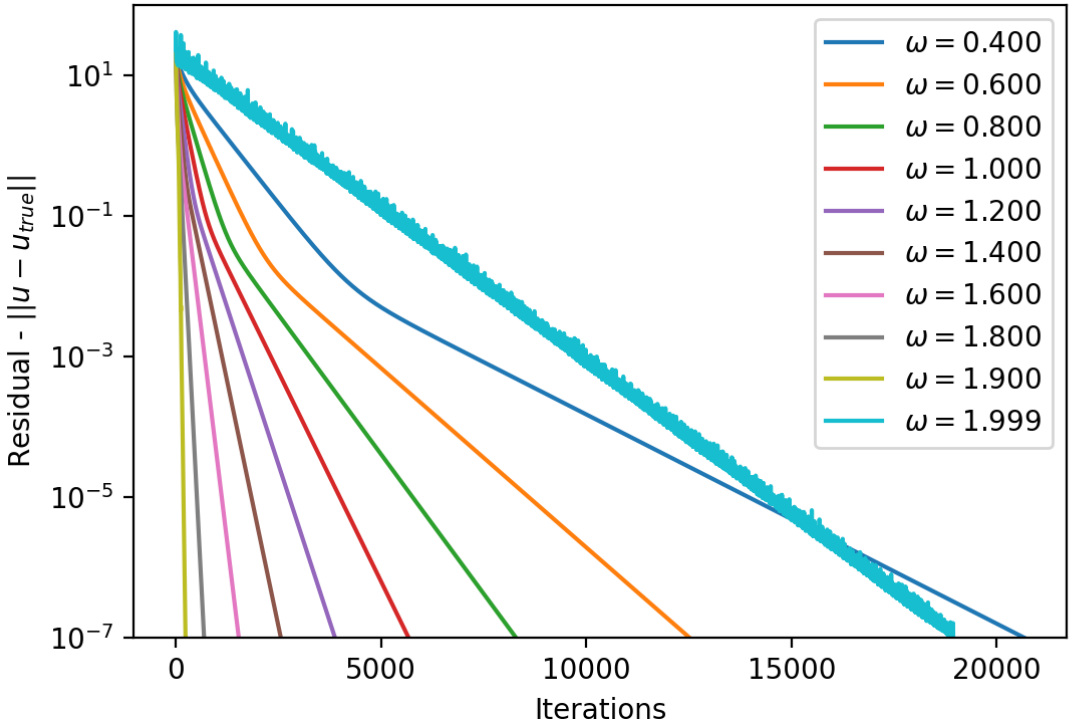

Figure 5 - Comparison of number of iterations for the convergence of Successive Over Relaxation method using different values of the relaxation parameter ω.

Figure 6 shows this effect more clearly by plotting the number of iterations needed for convergence as a function of ω with a log scale in the vertical axis. It confirms the optimum value of $\omega \approx 1.90$, in which case the number of iterations needed for convergence is just 235.

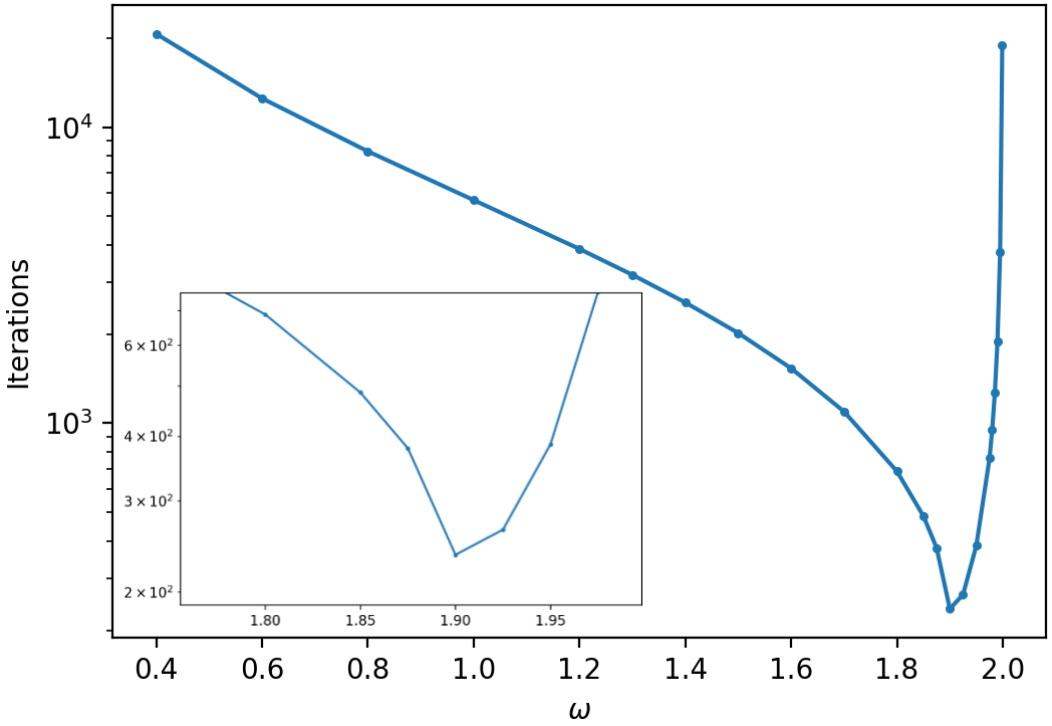

Figure 6 - Comparison of number of iterations for the convergence of Successive Over Relaxation method using different values of the relaxation parameter ω.

Initial Guess Effects

Another important consideration when using iterative methods such as the ones presented in the previous sections is the effect of the initial guess used on the number of iterations needed until convergence.

To analyse this, the Jacobi, Gauss-Seidel and Successive Over Relaxation methods were initialized using 3 random guesses, one guess which corresponds to an analytical solution to the differential equation, but does not satisfy the boundary conditions, and one final guess where all grid values equal to zero, to satisfy the boundary condition but not the equation.

Figures 7, 8 and 9 shows the convergence for the Jacobi, Gauss-Seidel and Successive Over Relaxation methods, respectively. As can be observe in Figure 7, in the case of the Jacobi solver, random initial guesses take many more iterations than using a zero initial guess. Using the analytical solution, even though it does not satisfy the boundary conditions, speeds up convergence even further. In fact, the analytical solution helped decrease the number of iterations needed for convergence by about 50%.

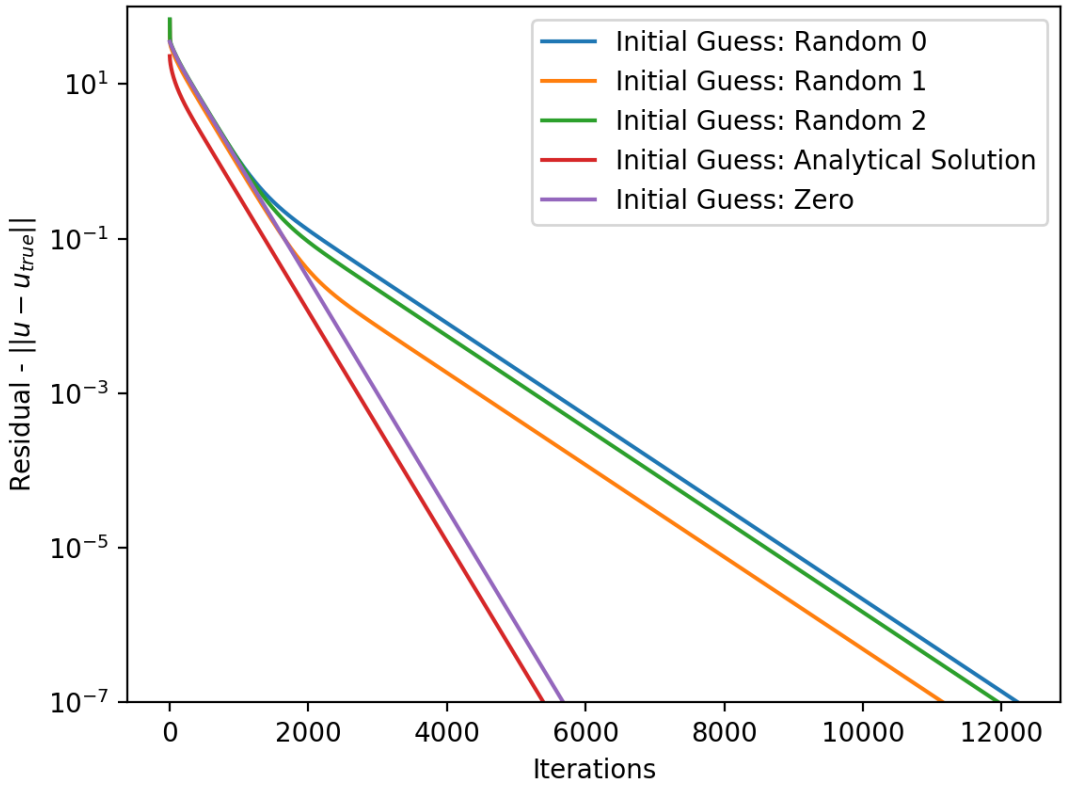

Figure 7 - Iterations Comparison for Different Initial Guesses using Jacobi.

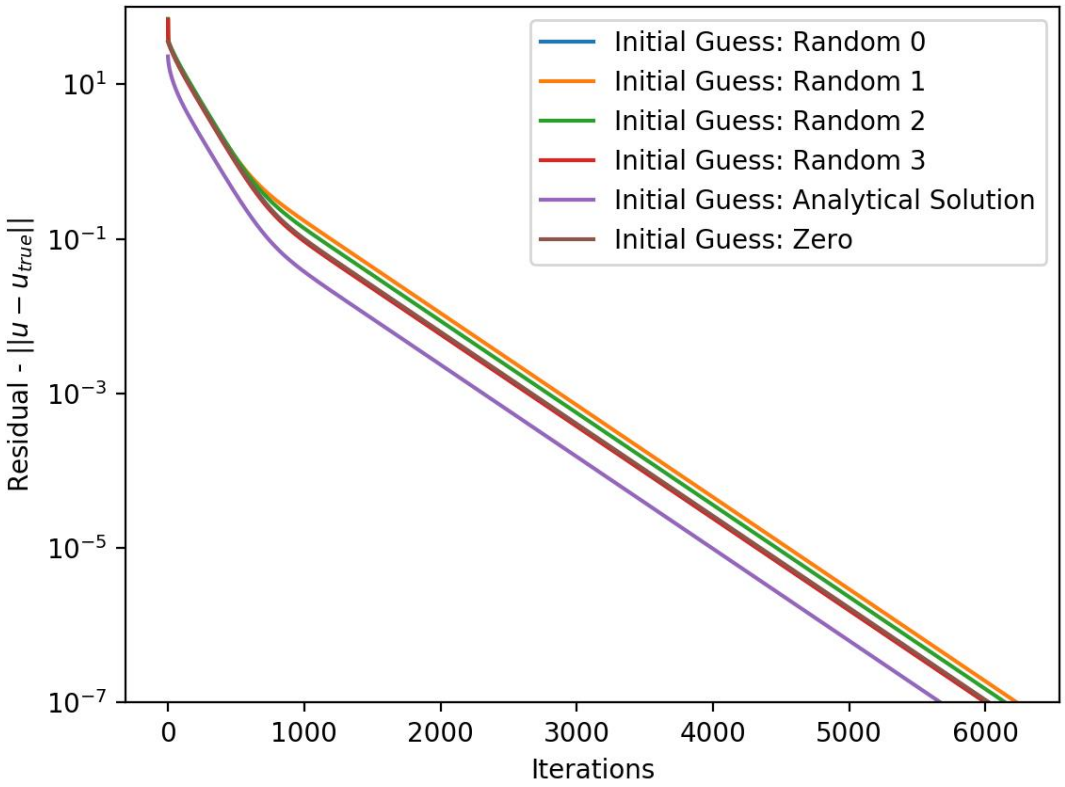

Figure 8 - Iterations Comparison for Different Initial Guesses using Gauss-Seidel.

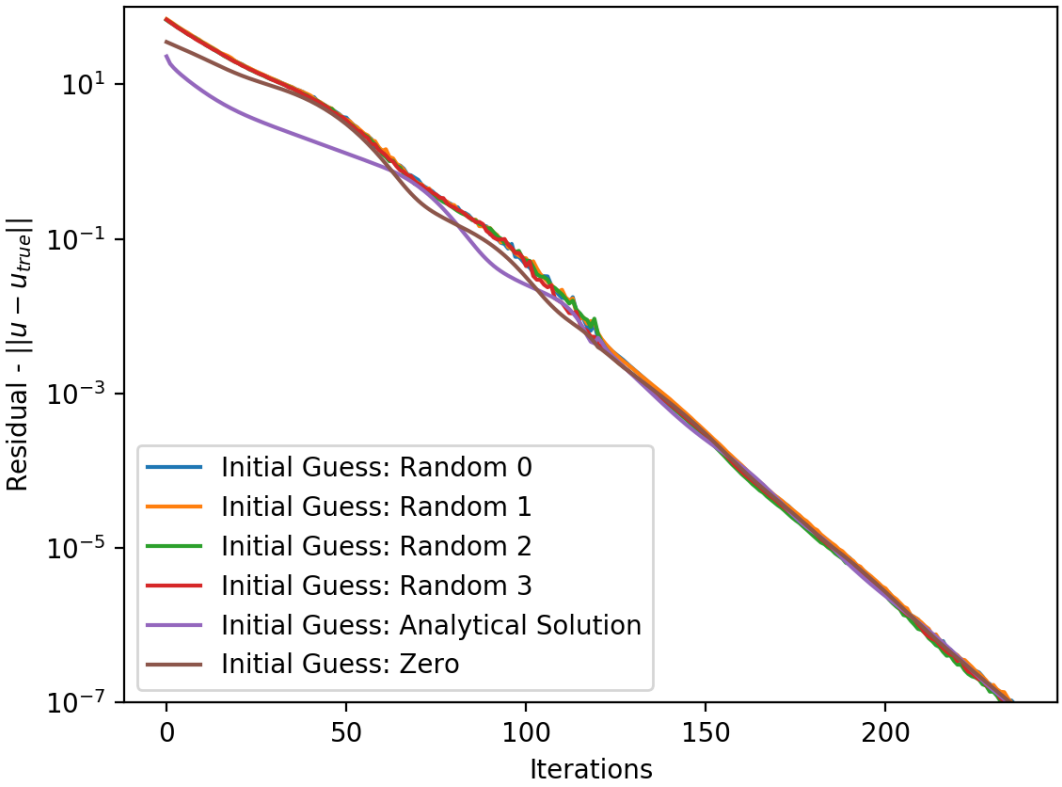

Figure 9 - Iterations Comparison for Different Initial Guesses using Successive Over Relaxation with $\omega = 1.9 \approx \omega_{opt}$.

Figure 8 shows that this effect is also present in the case of Gauss-Seidel, however, it is not as significant. Using the analytical solution does help it converge in less iterations, but only by about 10%. Figure 9, however, shows that this is not the case for the optimum Successive Over Relaxation. In fact, all initial guesses took about the same number of iterations to converge, only deviating by 1 iteration between each other.

Order of Accuracy of the Finite Difference Scheme Used

In order to verify the order of accuracy of the five point central finite difference scheme used for the second spatial derivatives, consider equation (9):

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = g(x,y), \qquad g(x,y) = -8\pi^2 \sin[2\pi(x+y)] \tag{9}$$

$$x, y \in \Omega = [0,1]^2, \qquad u(x,y) \in \partial\Omega = \sin[2\pi(x+y)]$$

Which has analytical solution given by (10):

$$u(x,y) = \sin[2\pi(x+y)] \tag{10}$$

The linear system which results from discretization of (9) was assembled and solved for different grid resolutions of $N \times N$ grid points using the LU method. Then, the error norm

$R_{L^2} = \sqrt{\frac{\sum(u_{numercal} - u_{analitical})^2}{N^2}}$ was plotted as a function of grid spacing $h = \frac{1}{N+1}$. In the case of a 61×61 grid discretization, the solution output is shown in Figure 10.

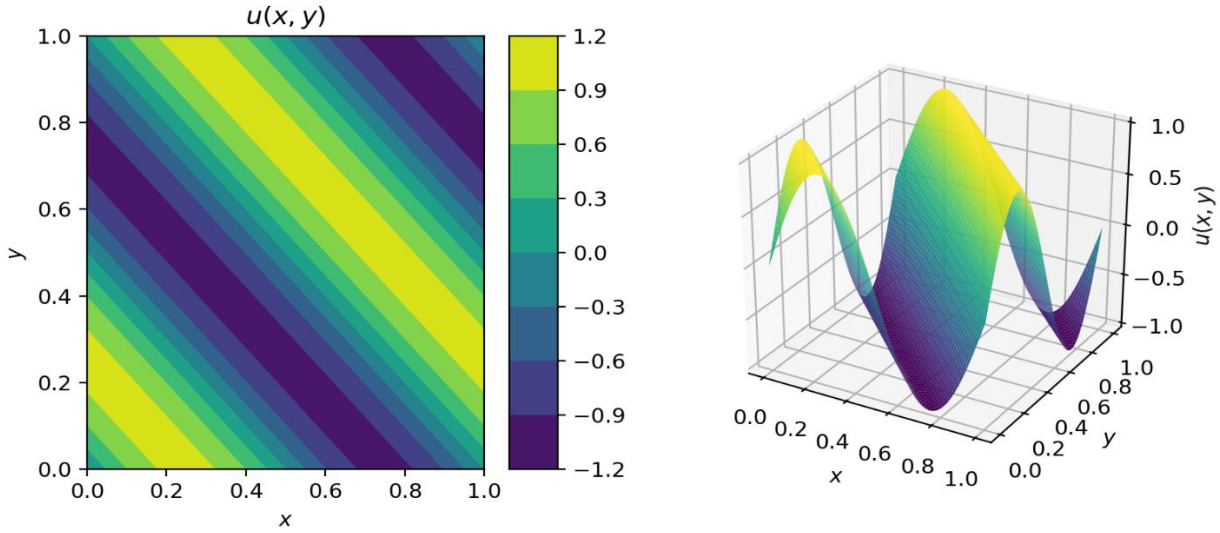

Figure 10 – Solution of equation (9) *by the LU method using a grid resolution of 61x61.*

The result of R_{L^2} as a function of h is presented in Figure 11, which compares the behaviour of the R_{L^2} norm obtained from the numerical scheme with the $O(h^2)$ behaviour. It leads to the conclusion that the five point central difference scheme used is indeed of second order of accuracy.

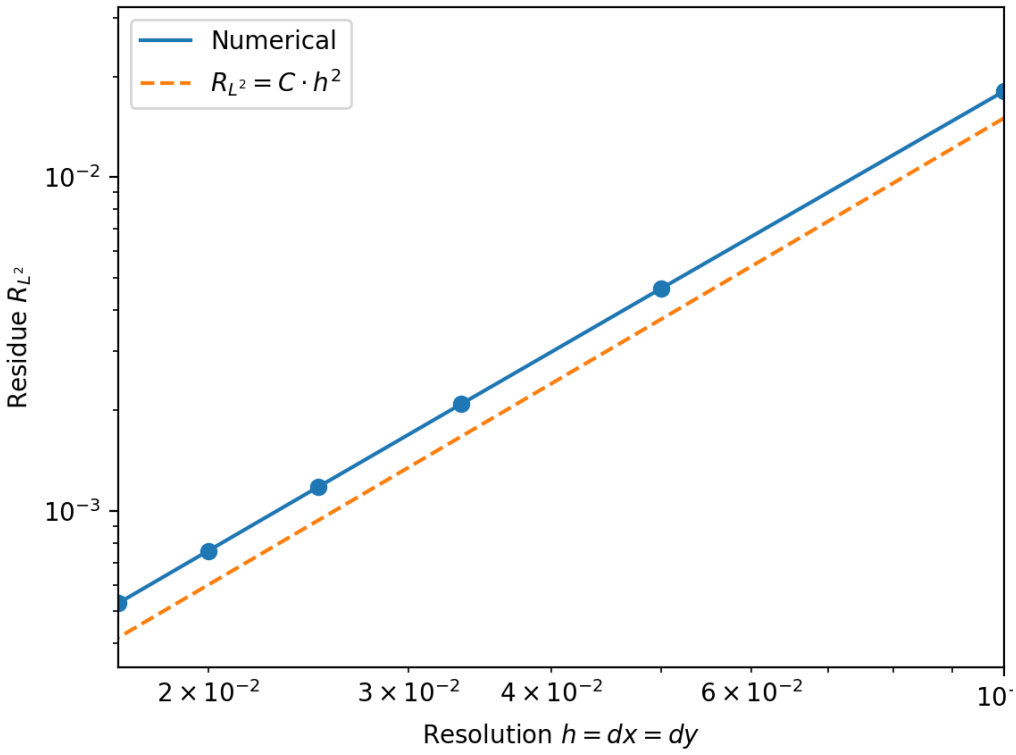

Figure 11 - R_{L^2} norm as a function of grid spacing h for the five point central difference scheme used.

Conclusion

This report has reviewed the numerical solution of the Poisson equation using direct and iterative solvers implemented in Python. It has been shown that Successive Over Relaxation can converge in just 235 iterations when the optimum relaxation parameter is used, in comparison to the 12000 needed for Jacobi, depending on initial guess. The impact of the initial guess has also been studied, showing that Jacobi exhibits up to 50% faster convergence when a good initial guess is used, while Gauss-Seidel and Successive Over Relaxation are not as sensitive to this effect. Finally, the implementation was successfully used to verify the second order of accuracy of the five point central difference scheme employed in the discretization.