Tom Howie 25317679

FEEG6002 – Advanced Computational Methods Assignment

Partial Differential Equation (PDE) setup

The objective of this assignment was to solve PDE1 using various different numerical methods. This reports details four methods implemented to solve PDE1, with explanations of each one, and a comparison between them. PDE 1 and its corresponding boundary conditions are given below:

$$\nabla^2 u = \rho$$

Where $u = u(x, y)$, and $\rho(0.5,0.5) = 2$ (and zero elsewhere), where $0 \le x, y \le 1$ and $u = 0$ on all the boundaries.
The PDE 1 can be expressed in matrix form Ax = b where:
A = ∇^2 $(Laplacian\ Operator)$
x = u : This is the vector of unknown variables to be found.
b = $\rho\ (Rho)$: This is the vector of variables rho which we are given in the question.

Method 1 – Finite Central Difference(4P) & Numpy

This method uses the first central difference for second order equation and is given by:

$$f''(x) = \frac{f(x-h) - 2f(x) + f(x+h)}{h^2} + O(h^2),$$

$Eq1$ [1]

A mesh is created using this equation in both the x and y directions, with an interval size of h, which when assuming a uniform grid results in the same step size in both the x and y directions, i.e. $h_x = h_y = h$ Therefore:

$$Eq2 \qquad u''^{(x,y)} = \frac{u(x-h,y) + u(x,y-h) - 4u(x,y) + u(x+h,y) + u(x,y+h)}{h^2}$$

This gives a four point stencil as each node in the mesh depends on its four surrounding nodes, i.e. North, South, East, West of it, and can be found using the above equation(Eq2) to give a weighted average value of the four surrounding nodes. The code for this section was obtained and modified from S.Cox's lecture slides [2]. At the beginning of this code the size of the mesh, n, is defined. The function "A_mat" takes the argument n and applies Eq3 to each point in the mesh to map the equations. Note that due to the iteration between nodes h in the code, the interval, can be assumed to be equal to 1 and thus ignored. This produces the A matrix which is of size $n\ x\ n$ and provides n^2 simultaneous equations. The function is compatible with any odd value of n so as to allow for a greater mesh size to be created and greater accuracy if required. The value of n must be odd so that the centre point can be set to the given value of ρ, 2. A test is included to ensure that the system produces an error message and exits the process if n is even, but commented out for submission. The function "A_mat_test" is used to print the A matrix to the screen to check for any errors that may have occurred. A number of values of n were tested and visually inspected before finalising the code. The single column b vector is created using the "b_vec" function with n rows and all values set to zero except for the mid value which is set to $= 2$. Similar to the A matrix, the b vector is visually inspected using the "b_vec_test" function output to check for errors.

The matrix A and Vector b are then passed as arguments into the imported "numpy.linalg.solve" function, which solves the problem and outputs the solution "soln1", a single column vector, i.e 1D matrix of values. In order to obtain the 2D solution "soln1" is wrapped into the correct 2D form of a $n\ x\ n$ matrix before being embedded in the boundary conditions of zero at its edges. This is performed using the “embed” function and results in the full solution U1 in its correct form (n+2*n+2 matrix). The function “rho_check” is used to check that the values of U1 are correct, the finite central difference equation is applied to the centre point of the matrix. The result, if U1 values are correct, should be equal to $2 = \rho(0.5, 0.5)$. The graphs are created for various values of n using the “plot_graph” function.

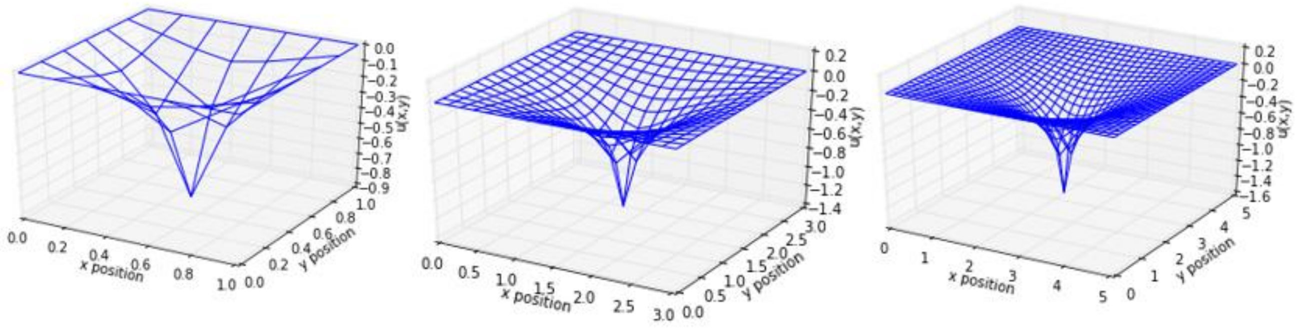

Figure 1 - Method 1 Graphs - Mesh Size n=5 (left), n=17(middle), n=29(right)

Method 2 – Over-Relaxation Gauss-Seidel

This method uses an iterative process to reach the solutions. The setup is the same as the first method, Matrix A and vector b are obtained in the same way. However instead of using the numpy function to solve it, a Gauss-Seidel solver is utilised. The code in this section for the “iter_eq” and “gauss-seidel” functions were adapted versions of those taken from Numerical Methods in Engineering [3]referenced in S.Cox’s notes [2].

$$x_i^{(k+1)} = \frac{\omega}{a_{ii}}\left[b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k)}\right] + (1-\omega)x_i^{(k)}, i = 1, 2, \ldots, n.$$

Eq3

ω is the relaxation parameter
if $0 < \omega < 1$ then under-relaxation , if $\omega > 1$ then over-relaxation.

The Gauss-Seidel method takes an initial guess for the solution of the equations and iterates over each element in the mesh, until it reaches a final solution that is of satisfactory tolerance. It is an improvement over the Jacobi method as it uses the latest estimate for solution as soon as it is created. Furthermore the relaxation parameter allows for interpolation/extrapolation between the old estimate and the newest using a weighting factor. This is known as a successive over-relaxation technique.

The “gauss_seidel” function in the code is passed 3 known arguments, the A matrix, the b vector and the specified tolerance for the solution. Within it omega is initially defined as 1, although this is further optimised by the function during operation(see below). The “iter_eq”, which represents Eq4, is called for each pass to produce a new estimate. The magnitude of change between the current estimate and the previous is compared at each pass until it is less than the specified tolerance. When this occurs the current solution is returned as the final solution along with omega and the number of iterations. If the “gauss_seidel” function fails to converge within 1000 passes it produces and error message and returns None. The tolerance was set to 1.0e-7 as this returned an accurate answer without excessive computational time.

The steps for ω optimisation are as follows [3]:

- *After k iterations with the initial value of ω, the magnitude of change during the kth iteration is recorded.*
- *Then p more iterations are carried out and the magnitude of change during the last iteration is recorded.*
- *The rest of the iterations are then carried out with ω_{opt}*

$$\omega_{\text{opt}} \approx \frac{2}{1+\sqrt{1-\left(\Delta x^{(k+p)}/\Delta x^{(k)}\right)^{1/p}}}$$

Eq4

By trial and error of the values of k and p, the Gauss-Seidel method can be refined to speed up convergence and decrease the computational run time. Reducing the tolerance can lead to a reduction in computational time and but also a reduction in accuracy. As this process is an iterative process is it generally slower, and does not provide as an exact answer as method 1 or 3. The solution is wrapped, embedded, checked and plotted in the same manner as in method 1.

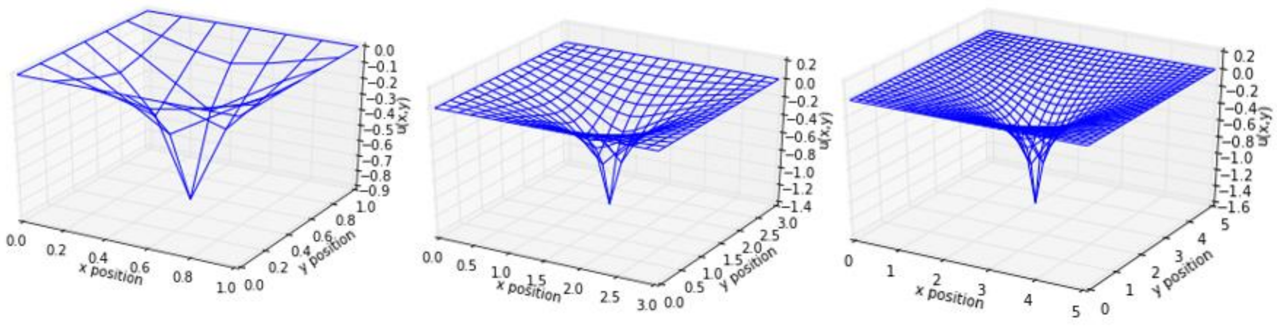

Figure 2 - Method 2 Graphs - Mesh Size n=5 (left), n=17(middle), n=29(right)

Method 3 – Eight Point Stencil and Numpy

This method is similar to method one however instead higher order derivative terms are utilised to produce a more detailed stencil:

$$Eq5 \qquad u''_{i,j} = \frac{-60u_{i,j} + 16\left(u_{i-1,j} + u_{i,j-1} + u_{i+1,j} + u_{i,j+1}\right) - \left(u_{i-2,j} + u_{i,j-2} + u_{i+2,j} + u_{i,j+2}\right)}{12h^2}$$

This gives an 8 point stencil as each node in the mesh now depends on its eight surrounding nodes, 2 in each direction, i.e. North(N1) & NorthNorth(N2), South(S1) & SouthSouth(S2), East(E1) & EastEast(E2), West & WestWest(W2).

The function “A_8p_mat” was written to create a new matrix A that utilises the new stencil and maps the equations to each node within the mesh. The “b_8p_ vec” function creates a b vector that accounts for the multiplication of ρ by the denominator 12, thus resulting in a mid value of vector b of 24. The A_8p matrix and b_8p vector is visually inspected for errors using “A_8p_mat_test” and b_8p_vec_test” functions. The mesh size must still be odd but now must also be a minimum of 5 due to the increased complexity and each node depending on 2 points in each direction. “numpy.linalg.solve” is still used to produce the solution, but the “embed_8p” function must be used as the u matrix requires a double boundary layer of zeros due to the stencil. The solution is checked using the “rho_check_8p” function with the new higher order finite central difference equation. The “plot_graph” function is called as before except passed the argument n+2 instead of n due to the double boundary layer.

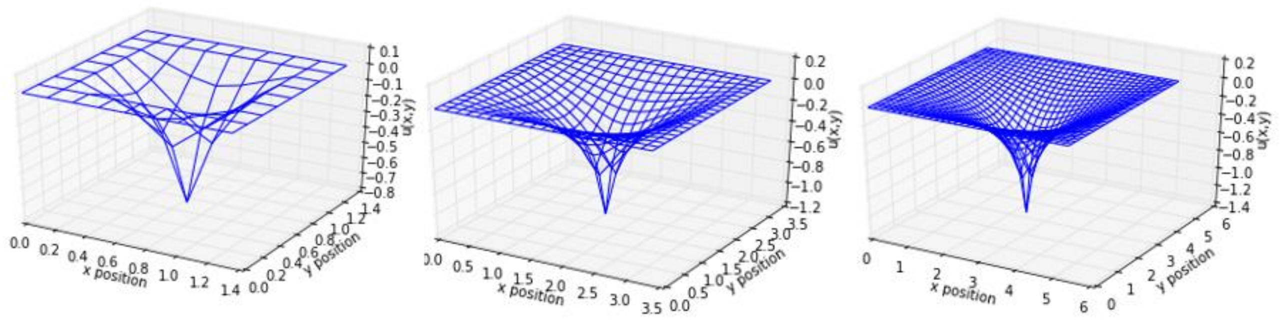

Figure 3- Method 3 Graphs - Mesh Size n=5 (left), n=17(middle), n=29(right

Method 4 Gauss-Seidel “Red-Black” Solver

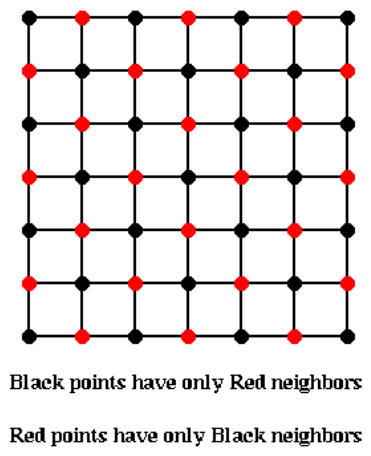
Black points have only Red neighbors
Red points have only Black neighbors

Figure 4 - Red Black Grid [5]

This method is similar to method 2 however it modifies the iteration of the points. In method 2, as is typical, combined for loops are used in order to iterate each element in the grid row by row and column by column, working its way out from one corner. The difference with the red-black solver is that the elements are alternating from red to black as shown in figure 4. The result is that black points are only connected to red points and vice versa. Therefore all the black points can be updated simultaneously from the most recently iterated red values, and then all the red points can be updated simultaneously using the most recently iterated black values. The advantage of this is if parallel computing is utilised, as groups of red points can be iterated simultaneously using multiple or multithreaded processors or computers and then groups of black points can also be iterated simultaneously. Technically each red point could be computed in parallel and then after, all the black points can be computed in parallel, providing the processor(s) are capable. Parallel Programming can be used to drastically speed up programs compared to running them serially.

The “red_black_guass-seidel” function is exactly the same as in method 2 however now it calls a different iteration function “red_black_iter_eq”. This contains two separate loops, instead of two combined together, the first one iterates over the odd nodes in the mesh, and the second iterates over the even nodes in the mesh. The “soln4” is then wrapped, embedded, checked and plotted using the same functions as in methods 1 & 2.

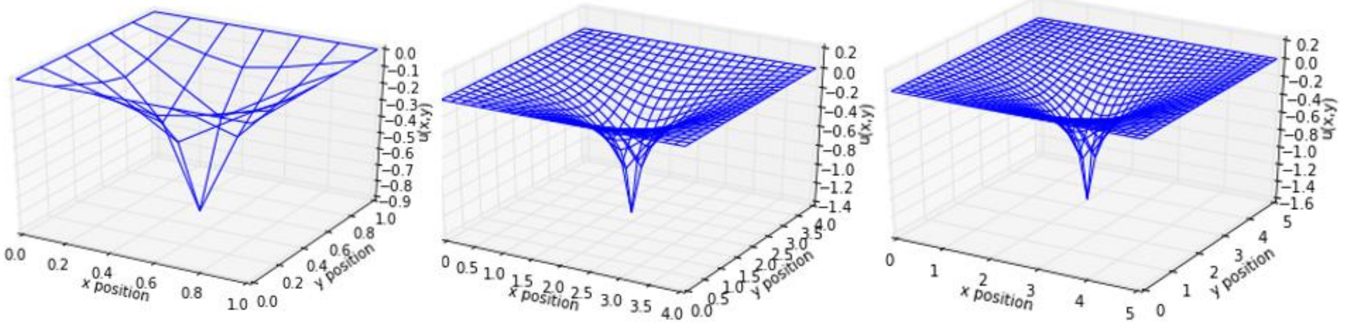

Figure 5 - Method 4 Graphs - Mesh Size n=5 (left), n=17(middle), n=29(right

Method Comparison and Methods to Decrease Computational Time

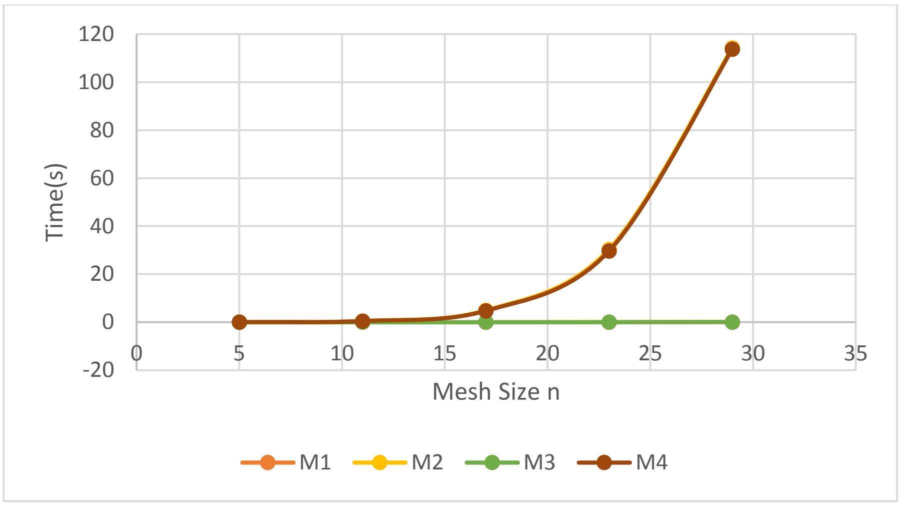

Figure 6 - Computational Time Comparison

- Methods 2 and 4 are much slower methods, especially at high mesh sizes(large n) as they are an iterative method and rely on an initial guess. They could be refined by trial and error of various k and p values within the Guass-Seidel formula to find the optimum relaxation parameter.
- Method 4 is marginally quicker (fractions of a second) than Method 2 thanks to the red-black solver, the difference would be larger for more powerful multi core / multi processor computers.
- Methods 1 and 3 are virtually the same, as they both use the already highly optimised linalg function of numpy. However the setup of matrix A could possibly be optimised and written in such a way to decrease the computational time, perhaps by utilising more efficient for loops.
- Methods 1, 2 and 4 produce very similar solutions as they use the same 4 point stencil, whereas method 3 an 8 point stencil, which is more accurate. This difference can be seen in the graphs.
- The graphs also show that for a larger mesh size the values become more accurate with the point (x,y) = (0.5,0.5) tending closer to 2 with a narrower neck.
- Increasing the mesh size, n, increases the computational time, exponentially for the iterative methods (2&4).

General Points for Decreasing Computational Time

There is always a trade-off between computational time and accuracy, with the higher the accuracy generally the higher the time to complete the task. However there are ways of reducing it without greatly affecting the accuracy:

- As python is an interpreted language it is relatively slow, especially when implementing for loops such as this code does. A drastic improvement would be seen if it was to be written in a compiled language such as C.
- Instead of completely rewriting the code, it may be easier to simply find which sections of the code are the slowest and use either the weave command to include C code or Cython to compile the corresponding code into C.
- Computational time could also be reduced by executing the code on high performance or supercomputers.

Other Ways to Decrease Computational Time for Iterative Methods(2&4)

- Effectively utilise parallel programming on high specification computers.
- Using multigrid methods

Multigrid Methods

Multigrid Methods can be used to speed up the relaxation method on a grid of specified fineness. This is the grid that the problem is defined on and the source terms evaluated on. [4] A coarser grid of the same size but with fewer points is used to find an estimate of the error between passes. For example if h is the interval of the fine grid, then 2h might be the interval of the coarse grid. A few iterations are carried out on the finer grid allowing the initial difference/error between iterations to be estimated and passed to the coarse grid. The coarse grid is easier to solve as it has fewer points and iterations are carried out it, from which the error of the pass can be found and then interpolated back onto the finer grid, before being iterated a few more times. Effectively this is a faster method of estimating the error between passes using a lower resolution grid and is known as a two grid method. Multi layers of increasingly coarse grids can be used in order to speed up iterative processes such as the Gauss-Seidel method and increase the efficiency further. The benefits are more apparent for high resolution meshes, i.e. a large value of n, where a successive over relaxation technique like the Gauss-Seidel method might take a long time to compute and converge.

References

[1] S. Dr Sinayoko, Partial Differential Equations II: 2D Laplace Lecture Slides, University of Southampton.

[2] D. S. S. Prof Simon J. Cox, “Advanced Computational Methods & Modelling Lecture Notes”.

[3] J. Kiusalaas, in *Numerical Methods in Engineering with Python*, Cambridge University Press, 2010, pp. 82-92.

[4] W. T. S. V. W. a. F. B. Press, “Multigrid Methods for Boundary Value Problems,” in *Numerical Recipes in C, Second Edition*, Cambridge University Press, 1992, pp. 871-889.

[5] University of California, Berkeley, “Solving the Discrete Poisson Equation using Jacobi, SOR and the FFT,” 11 04 1995. [Online]. Available: http://www.cs.berkeley.edu/~demmel/cs267-1995/lecture24/lecture24.html.