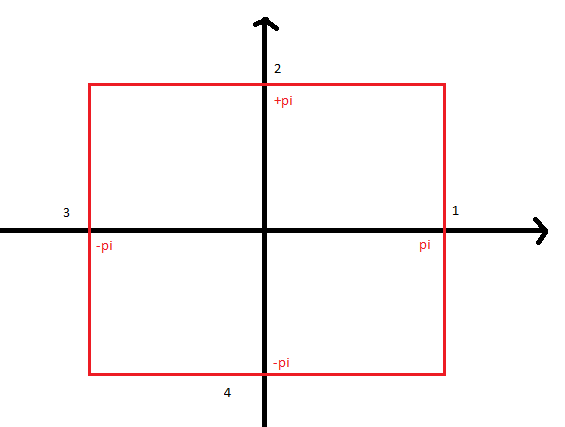
Abstract

This paper cover a finite difference solution to a given Poisson partial differential equation problem having both Neumann and Dirichelt boundary conditions. The Problem is explained and is discretized to be able to be solved by a computer. One explains how the problem can be solved numerically along with specific modifications needed due to the nature of the boundary conditions (e. g. ghost nodes). The report covers and explains two different types of methods that can be used to solve the given problem. The two are the Gauss-Seidel and Successive Over Relaxation (SOR). Once the numerical methodology of solving the problem is completed, a pseudo-code explanation of how the problem would be solved numerically by the computer is given. To make sure that the set-up of the problem is done correctly, a grid convergence study is done to ensure that the solution is independent of the how fine the mesh used is. To verify, the program the method of manufactured solutions is both described and the results of the process are shown. Once this is done, the solutions for the given Poisson equation and boundary conditions solved by the computer program are given and discussed.

For the semester project, the student was assigned to solve the Poisson equation. This equation is as follows:



Figure

This equations was to be solved on two dimensions in a square ‘plate’ domain of range from negative pi to positive pi in both the x and y axes. Moreover, each of the four sides of the ‘plate’ has a boundary condition to make to problem solvable. The conditions are as follows:

Conditions 2 & 4 are Neumann Conditions. This means that the ‘flux’ across y for either of these sides must be zero. In other words:

Conditions 1 & 3 are Ditchelt Conditions. Here, for each edge, the respective points along that edge must have an assigned value that depends on the equation for each one. For edge 1, the points of this edge as y changes follow the directive of:

This however was changed at the last minute by electronic mail to be instead the following [6]:

Thus here, on x= pi, the values will change and depend on where y is at along the edge.

Edge 3 follows a different directive, but the principle is the same. The equation here is:

Finally, the F is also given. This F will have assign a certain value to every location on the ‘plate’ that is not one of the boundary edges. This F is the following:

To first solve this problem numerically a discretization scheme for the Laplacian of the equation such that the computer is able to solve the problem must be introduced. The Laplacian is just the sum of the double derivative of u with respect to both of the two dimensions of x and y. In other words,

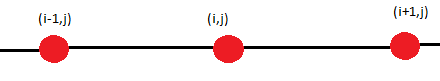
Each one of these double derivatives can be roughly approximated by the following:

This method is done so by applying a Taylor expansion of u(x,y) to further points of u(x-h,y) and u(x+h,y). Where h is a small increment. By carrying out and solving a system of linear equations, one ends with the formula above [1]. One must also notice the ‘P’ term. This term is the remaining error due to the additional series in the Taylor expansion. This error will be of magnitude of about h squared. The same procedure can be done for the ‘y’ term and one ends up with a similar result.

If the 2-d plate was broken up into discrete values rather than continuous values, where each one was a distance of ‘h’ in either the x or y direction from the next closest point. Doing this, one would end up with a finite number of defined points rather than the mathematically infinite quantity of points within the plate area.

Now the double derivatives can be approximated as the following:

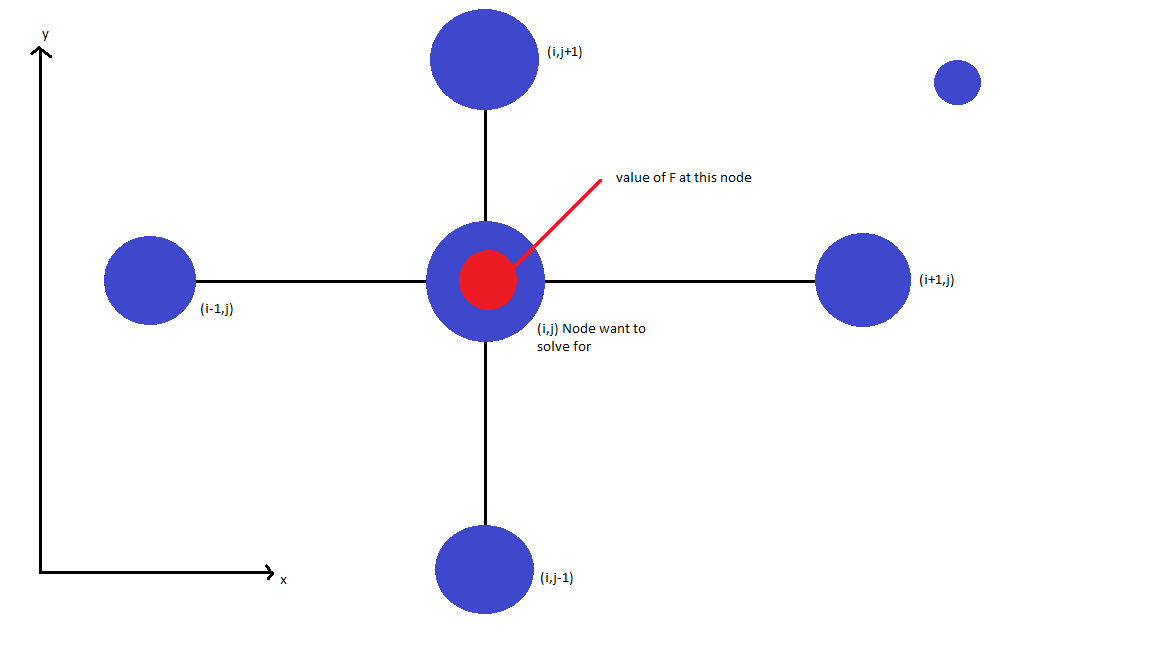
One can see that the error terms still remains, but the smaller the h is and thus the larger amount of discretizations that there will be will lead to this error to decrease- up to some limitations.



Figure

Figure 2 shows how this process works for the double derivative along x; one will need three discretized values to solve for the double derivative. A similar argument can be done for the double derivative along the y axis. Using both of these derived and discretized arguments, one can rewrite the initial Poisson equation as follows:

Here the error term has been dropped and all sides have been multiplies by ‘h’ squared. A closer and illustrated look at this gives one the following information: the solution for u at any discretized point can be solved. This u will depend of the four adjacent u values plus the known value of F at the discretized point. This formula can be applied for all discrete points within the plate.



Figure

If the unknown u’s for the plate are defined in (i,j) from 1 to N for either i or j to cover the whole plate at specific points, one will end up with a set of formulas. For example, to find the first u at (1,1). The equation would be as follows:

Here, for this case, the u(1,1), u(2,1), and u(1,2) would be unknown. However the rest of the values would be known. By looking at the figure, one can see that the known u’s for these cases will be the boundary conditions given at the start of the problem. Where u(0,1) would apply to edge three while v(1,0) would apply to edge 4 as defined on the figure. Since defined these conditions are known more over the following directive can be stated:

Any u(0,y) will be given be the boundary condition three for the given y, while any u(N+1,y) will be given boundary condition one.

The same can be done for v(x,0), giving them condition four, and for v(x,N+1), giving them condition two.

However a problem does arise. While condition three and one give a u value for their set of points, conditions two and four give a value that is the derivative of u- since they are Neumann. However, one is trying to solve for the value of u at every discrete point- not its derivative. The Neumann condition clearly states the u derivative for edges two and four, but does not give the u values for these two edges-all of the u’s in these two edges must be additionally solved. This can be done by using the method of ghost nodes.

This methods works as follows:

By using the Taylor expansion once again, one can end up with a close approximation of derivative at a specific point. For this example, this will be u(1,0) from the previous equation.

Here, the v(1,-1) would belong to a region outside the plate, never-less one can manipulate the above and end up with the following:

Thus, one has a value for a node outside of the ‘plate’. This and similar values will be used to solve for u on the plate, or in other words along the coordinates of u(x,0) from x=1 to N.

So to solve for the first non-corner element on edge four, one gets the following:

Here, the u(0,0) is known as the corner value is an overlap of edges three and four and u is known on three.

For the next node on the same line, one gets the following:

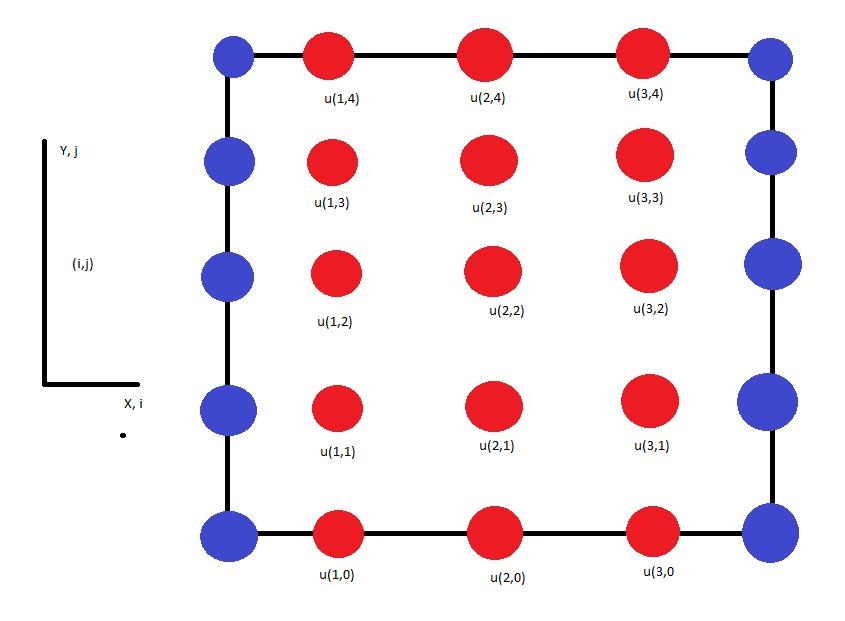


Figure simplified node example, red nodes are unknown and must be solved for

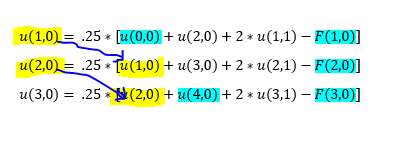
Therefore, one has a method to solve for every u on the ‘plate’ including for the Neumann nodes on edges 2 and 4. With all of these derived methods, one can solve for a very simple example with a node length of three to end up with 15 nodes to solve for (nine inner and six on the Neumann lines). This will result in 15 different equations, and a 15 of total unknowns. The equations are as follows:

Here, every value highlighted in light blue is already known, due to either being part of given F or from belonging to the edge values of edges one and three.

IF this whole system, was to be place In matrix form as follows:

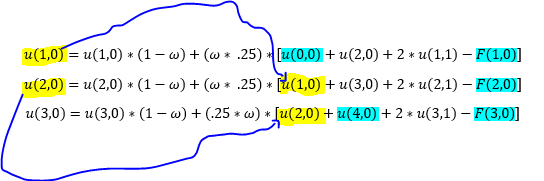
Where A would be the values of U for each corresponding row, u would be all of the u arranged where one would run the I’s for each j and repeat the process, and the F vector would represent the known (cyan) F and u values. In Matlab, it is possible to solve this by using the built in F/A command. However, this would only work for this very simple example and cannot be used to solve for the u values in problems with very large numbers of nodes. If put into matrix form as described above, one would see that the matrix is quite sparse, meaning that there are a lot of zeros present. While sparse one cannot use the tri-diagonal or Thomas algorithm to solve for this matrix as it will not be tri-diagonal. However, one can solve for the u’s in a more manageable way by using either the Gauss-Seidel method or successive over relaxation (SOR).

Using the Gauss-Seidel method, one solves for each u by doing the following steps. First one would guess the values of all the u’s for all of the unknown nodes. With these guessed values, one can find a new value for each unknown u as shown in the set up on the previous page. When the u at a specific solution row is found, this new value is used on the next row to help find a new value for the u on this row. The following equations show the example on how this would happen. Here the first value to solve is for is u(1,0). When this is solved for, one goes to the next row and solves for u(2,0). However, one will use the *updated* u(1,0) (highlighted in yellow) value figured out in the preceding row to help find u(2,0). A similar process happens in the next row using the just found u(2,0) value to solve for u(3,0). This process will go on down for all the rows until the last row is reached. Once the last row is reached one starts all over and uses the latest value to solve for each row-specific u. This process is repeated until some sort of convergence is reached.



Figure

Successive-Over-Relaxation or SOR works in a similar principle as the Gauss-Seidel method. However, now each rows has a so-called correction made up of the old value of the respective u that one is trying to solve along as shown in the following:



Figure

As one can, for each u index, the current u of the same index is used to help find an updated value of the u. Also notice that the omega factor now included. This omega is put here to ensure faster convergence than when compared to the Gauss-Seidel. The value of omega can be any number between one and two, with 1.5 being a decent choice[2]. Like before, SOR is iterated to the process is repeated various times down the rows. Yet, the addition of the omegas allow a faster convergence to a solution than the plain Gauss- Seidel.

While both the gauss-Seidel and the SOR methods will iterate to the solution eventually in theory, there is a problem. This problem is when to stop the iteration in a way such that the u solution for either method is close enough? A solution to this is to use the following for both methods: One compares the normalized ‘error’ for each u index by taking the difference between the just solved u index and the value of the same u index obtained from the last iteration. One would then divide this difference by the u index of the old iteration to obtain a normalized error. This would be done as follows:

Once the whole array error matrix has been found, the largest index is chosen. If the value of the largest error index is decently small enough, this means that the normalized difference between the previous and the new index value has not changed significantly. This will imply that the solution has iterated for long enough to converge towards a specific value.

Now that both the Gauss-Seidel and the SOR iterative solution methods have been explained, one can demonstrate how each one would be solve in the program by using pseudocode and explanations for each part in parenthesis. In the MATLAB program, both the Gauss-Seidel and SOR solve as follows:

*While condition==0, do:*

*For i =1 to max-1*

*U(i , 1) = (Formula for solving for U values at boundary #4 as equation will be different due added value Neumann condition)*

*End*

*For j=2 to max-1*

*For i=2 to max -1*

*U(i,j)= (Formula for solving U in INTERIOR nodes either in G-S or SOR. To see how the program solves this, please refer to figure 4 for G-S & figure 5 for SOR. Algorithm is set up in a way such that for a certain j value (y) every i value is run through first and the y value increases. This is done in the same way as in the example shown by figures 4 & 5.)*

*End*

*End*

*For = i=1 to max -1*

*U(i, Max)= (formula for solving U values at boundary #2, like for boundary #4, the equation will be different than the standard one with the interior nodes due to the Neumann boundary condition)*

*End*

*For j=1 to max-1*

*For i=1 to max-1*

*Comparer(i,j)= absolute value of (Here program will compared relative error with respect to current and the previously computed value on an iteration of either the G-S or SOR.)*

*End*

*End*

*If ( pick largest index value of comparer index) < wanted accuracy*

*Condition ==1 (Here condition is set to one and the program can escape the outermost ‘while’ loop and end the iterative process. This means that the change in values has become below a certain threshold and the solution has converged.)*

*Else*

*U\_old=U;*

*‘Counter’ adds one*

*(Here, if the Threshold is not reached, the just found U values are set aside as old values to be compared in the next iterative step. The program keeps track of the counter by adding one and the whole program loop begins again)*

*End*

*End while (End of the iterative solver part of the program when condition is met)*

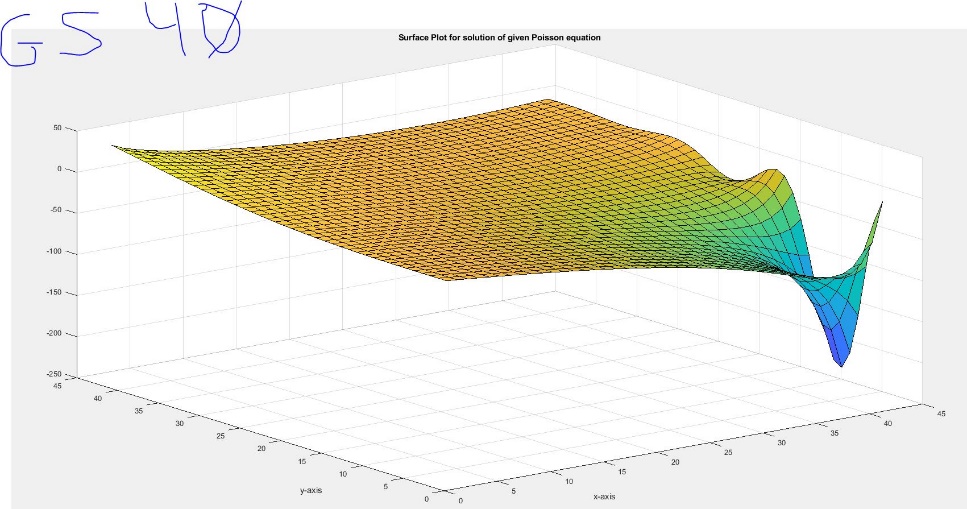
The first step to make sure the answer is the correct one, one must prove that the numerical solution to the given problem will be independent of the grid size used. In other words, the answer must not change if the program’s number of input nodes changes. This means that the solution must be solved for in a quantity range of nodes that give the lowest possible error. IF too few nodes are used, the solution will show discrepancies due to truncations. However, if one uses too many nodes, there will discrepancies due to so-called round off error. The acceptable node quantity range lies between these two regions and serves as a ‘Goldilocks zone’ that has neither too few nor too many nodes and will give the results with the least error[ucsb].

\begin{figure}\begin{center}
\leavevmode
\epsfbox{error.eps}\end{center}\end{figure}

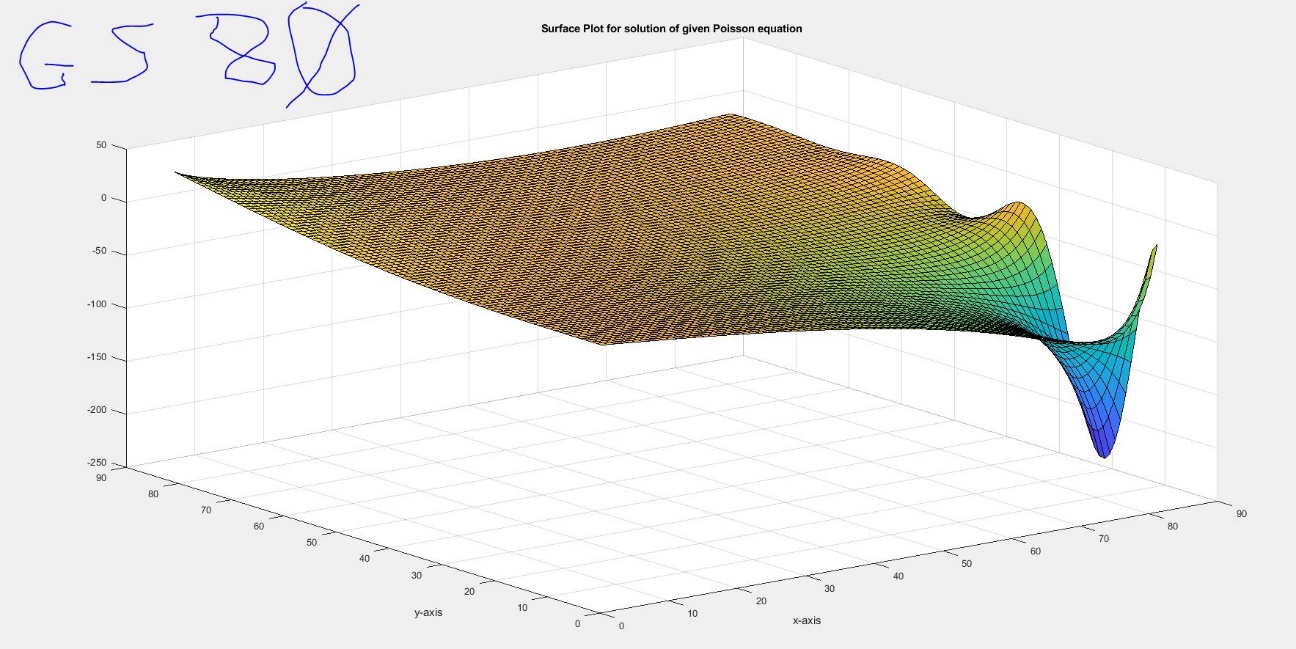
Figure Showing how error types vary with grid size. The dip in the total error is the area one aims for [ ]

Thus, one must first determine the acceptable zone where the solution surface for either method remains constant regardless of the quantity of nodes used. This can be done by running the program and solving for the equation for both iterative methods. The solution surface is looked into closely to see that the allowable node range where the solution surface does not change when putting more or less nodes in the set-up.

Before any results from running the computer program are shown, a brief discussion of the computer being used to run the MATLAB program is warranted. The computer used to run all of the simulations was a computer available to all students in the library’s computer lab. This computer runs a 64 bit Windows10 and is named ‘REF279’. According to the system information page, the computer consists of one processor: an Intel Core i5-6600 CPU with four cores. The current clock frequency is set at 3.30 Gigahertz. This frequency is significantly below the maximum frequency of 3.90 Gigahertz that the processor can have. The processor L3 memory size is of 6 megabytes with a bus speed of 8 GT/s. On average, the processor dissipates about 65 watts of power during usage [3].

Foremost, here the solution by method of GS for 40 by 40 interior nodes grid. This one and all of the following will have an ‘error’ condition between the difference of the normalized current and previous node value set as .001 for the solver to escape from the iteration

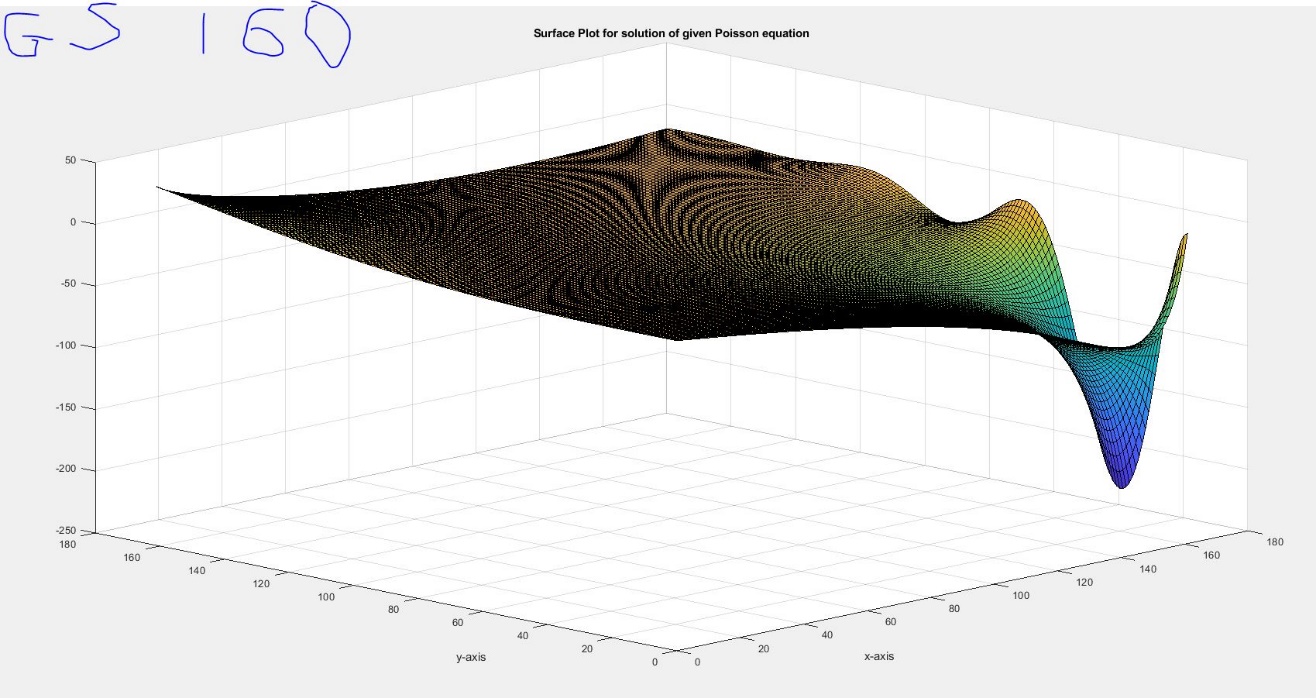
Figure



Figure

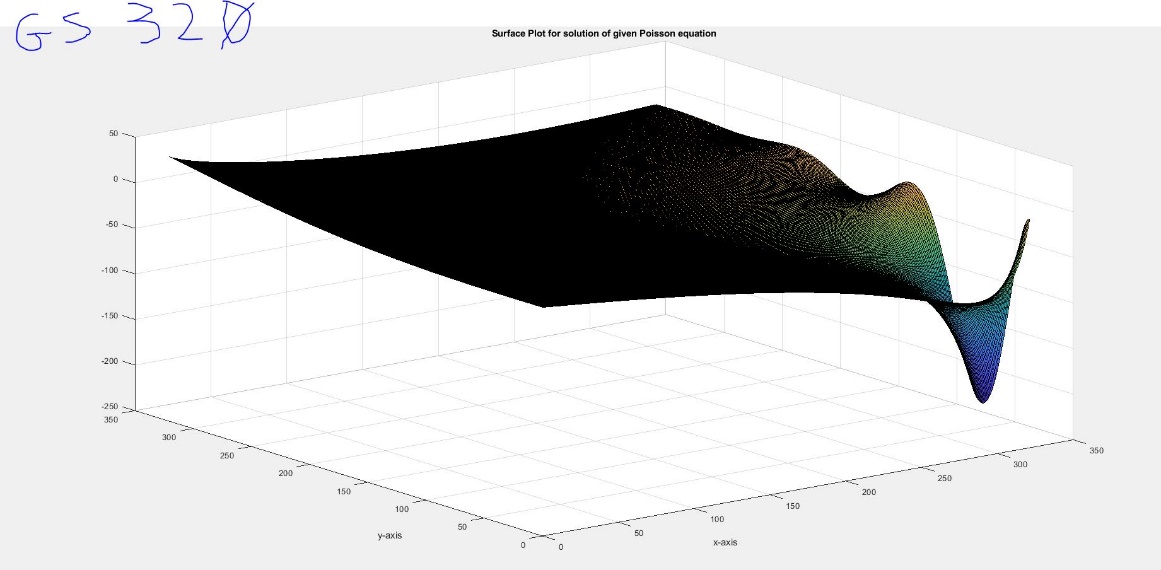
The figure above shows the GS solution for an 80 by 80 interior node size.

Meanwhile, the figure below shows the solution for a 160 by 160 interior node size



Figure

Lastly the figure below show the solution by having a 320 by 320 nodes mesh:



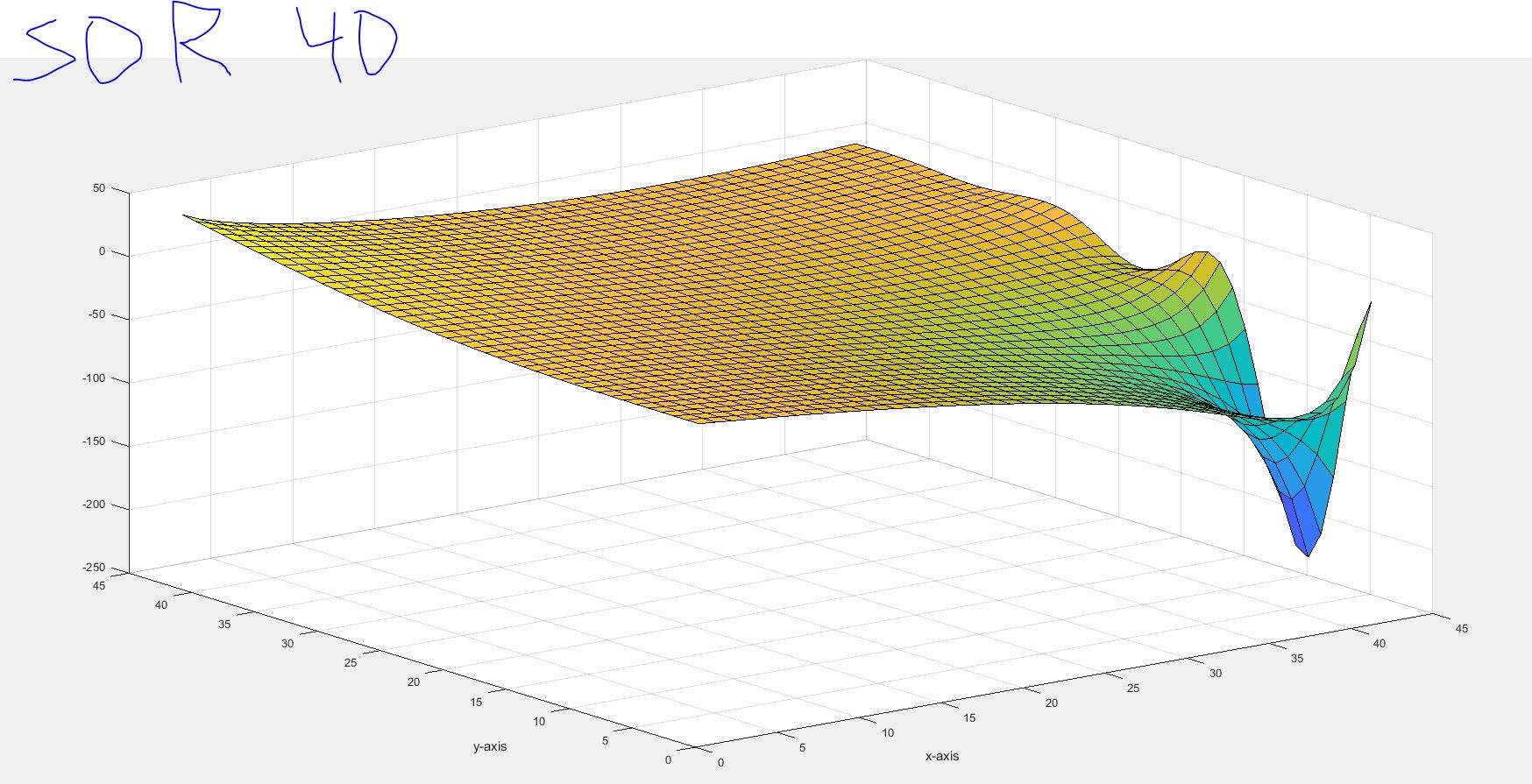
Figure

By looking at these results, the ones between 80 to 320 nodes seem to be the most similar and unaffected by the change in node quantity, from this the best mesh for the GS method appears to be 160-ish, As it gives both independence and a decent computing time as shown in the table below.



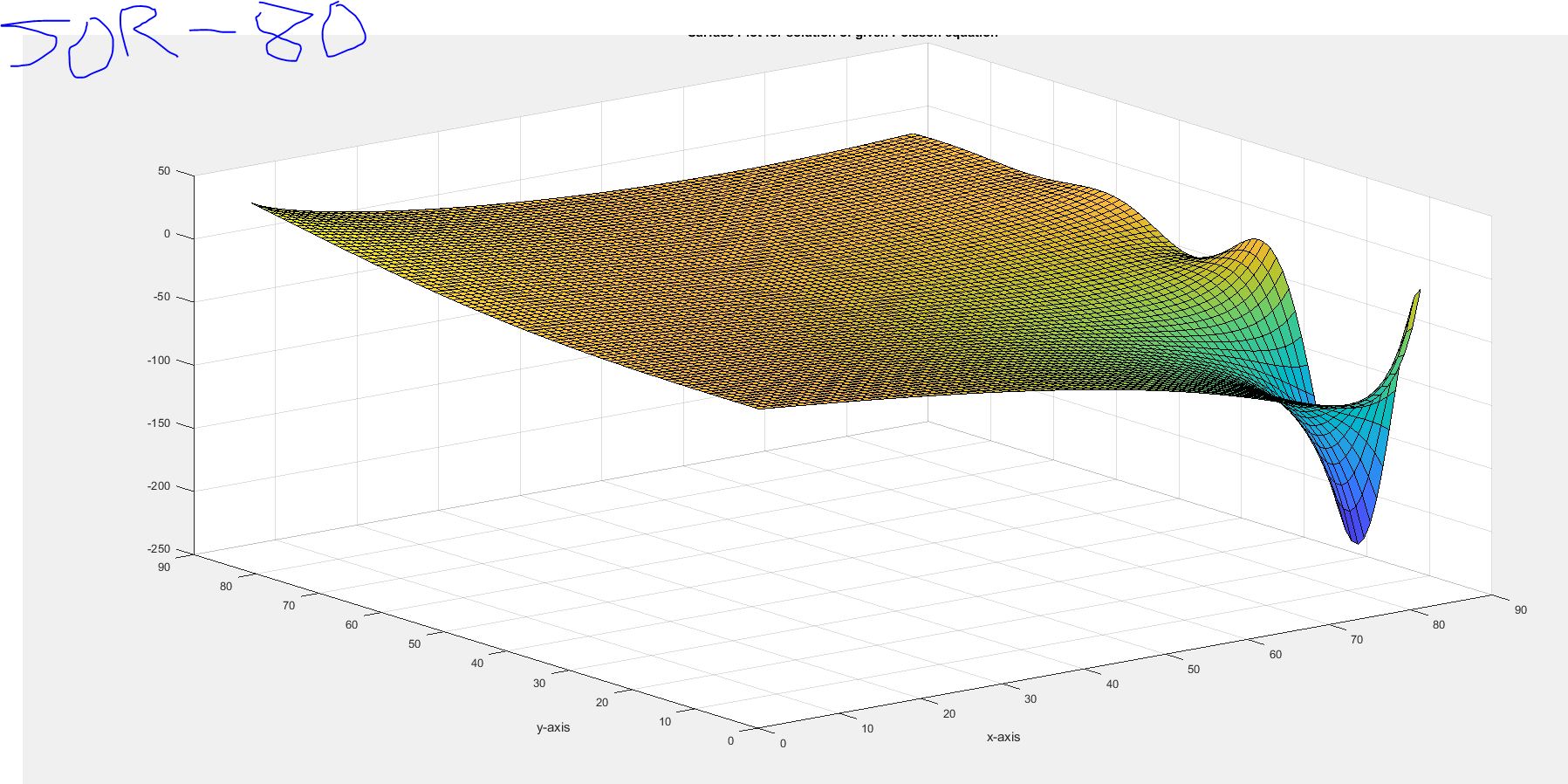
Doing the same procedure for the SOR one gets the following:

For a 40 by 40 interior node mesh, one gets:



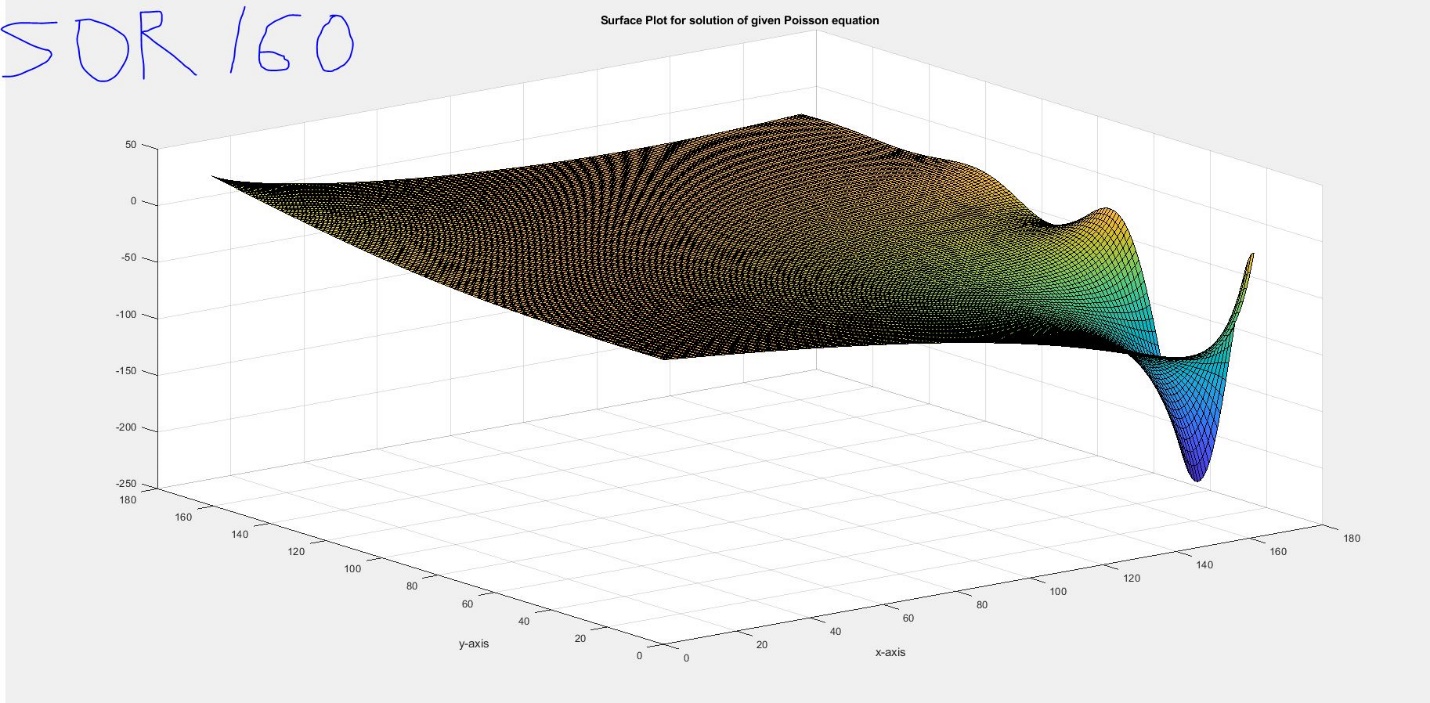
Figure

Below are the results for an 80 by 80 node mesh

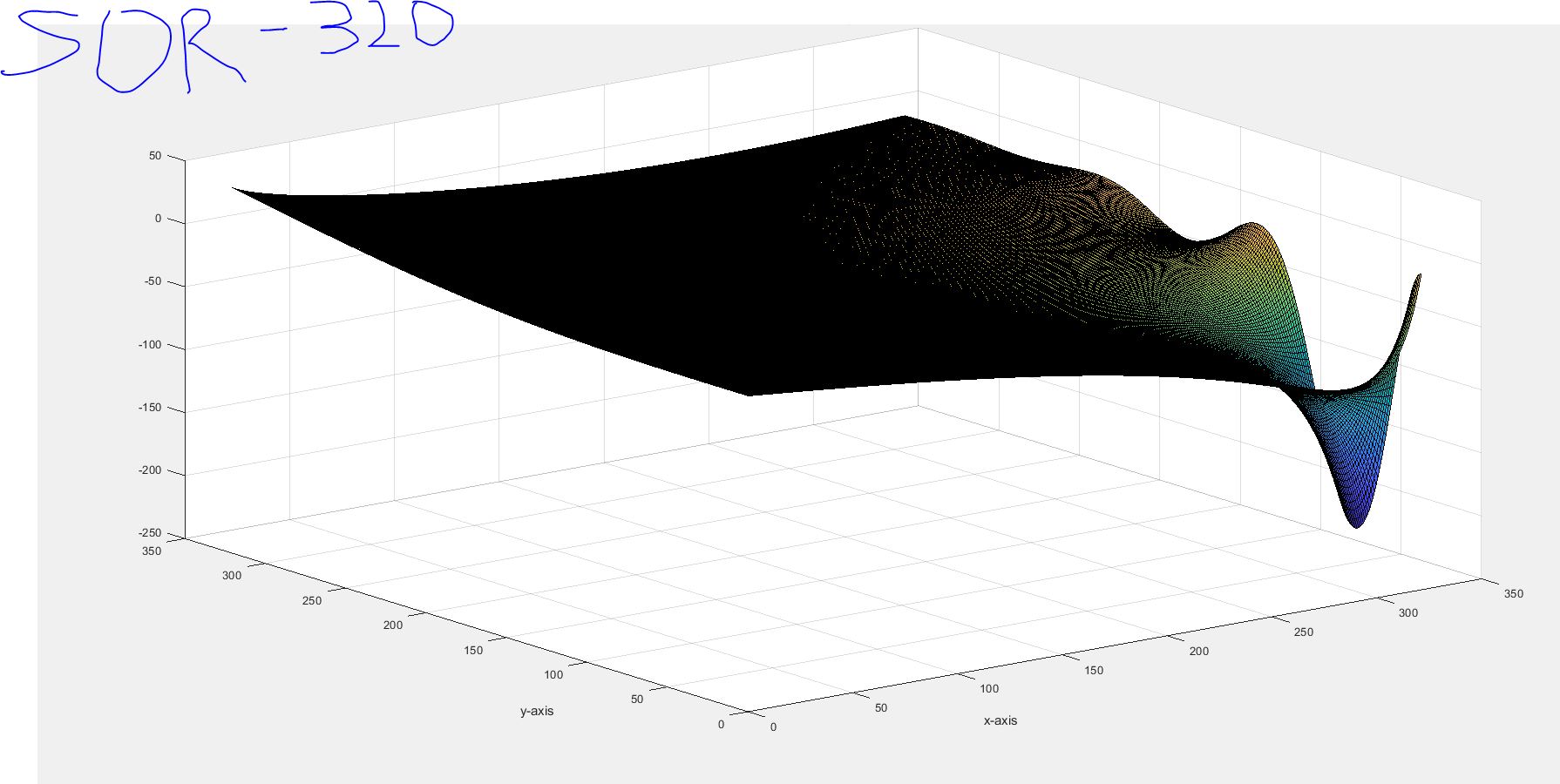


Figure

Lastly, the solutions from a 160 by 160 and a 320 by 320 nodes mesh are shown respectively



Figure



Figure

Based on the similar argument, the last two solutions seems to hold a constant geometry, despite the change in the size of node points. From this, the node quantity will be set to be 160 for the SOR, as it provides both good independence and computing time.



While no closed-form solution was given (and virtually all engineering problems to be solved numerically wont or else they wouldn’t needed to be solved so), there is a way for one to verify that the program is working properly and solves as it should. This method is called the method of manufactured solutions. This method consists of just inventing a solution for the problem and then working backwards from there. There are certain directives on which types of functions can be used [4]. However, for the Poisson type equation, the professor instructed the following [Verification]:

First, a solution is made up and the following made-up solution will be used:

This solution plugged into the Poisson equation in place of ‘u’. Taking the Laplacian of V results in the following equation:

The result of this Laplacian is plugged into instead of ‘F’ and one solves for the problem *numerically to* find a solution of V’. V’ prime can be compared to the manufactured to judge how the program works. A question does arise on what to do with respect to the boundary conditions to the problem. While changing boundary condition on a partial differential equation will give a different value answer, the same answer can be given by different sets of boundary condition types [4]. This allows one to obtain the any type of boundary condition from the manufactured V. Too keep things constant, one will use the same type of boundary conditions as given in the problem statement.

So for boundary one: for each point while y is allowed to vary

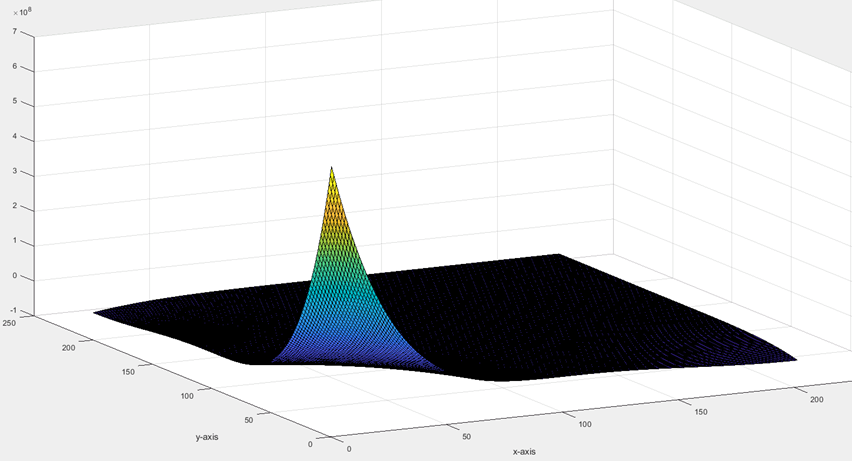
For boundary three: for each node point while y varies

Boundary two and four are Neumann types, so the derivative of V with respect to y is taken to get the ‘flux’ across at each respective point in the boundary. Differentiating V results in the following:

So for boundary two: while x can vary for each node

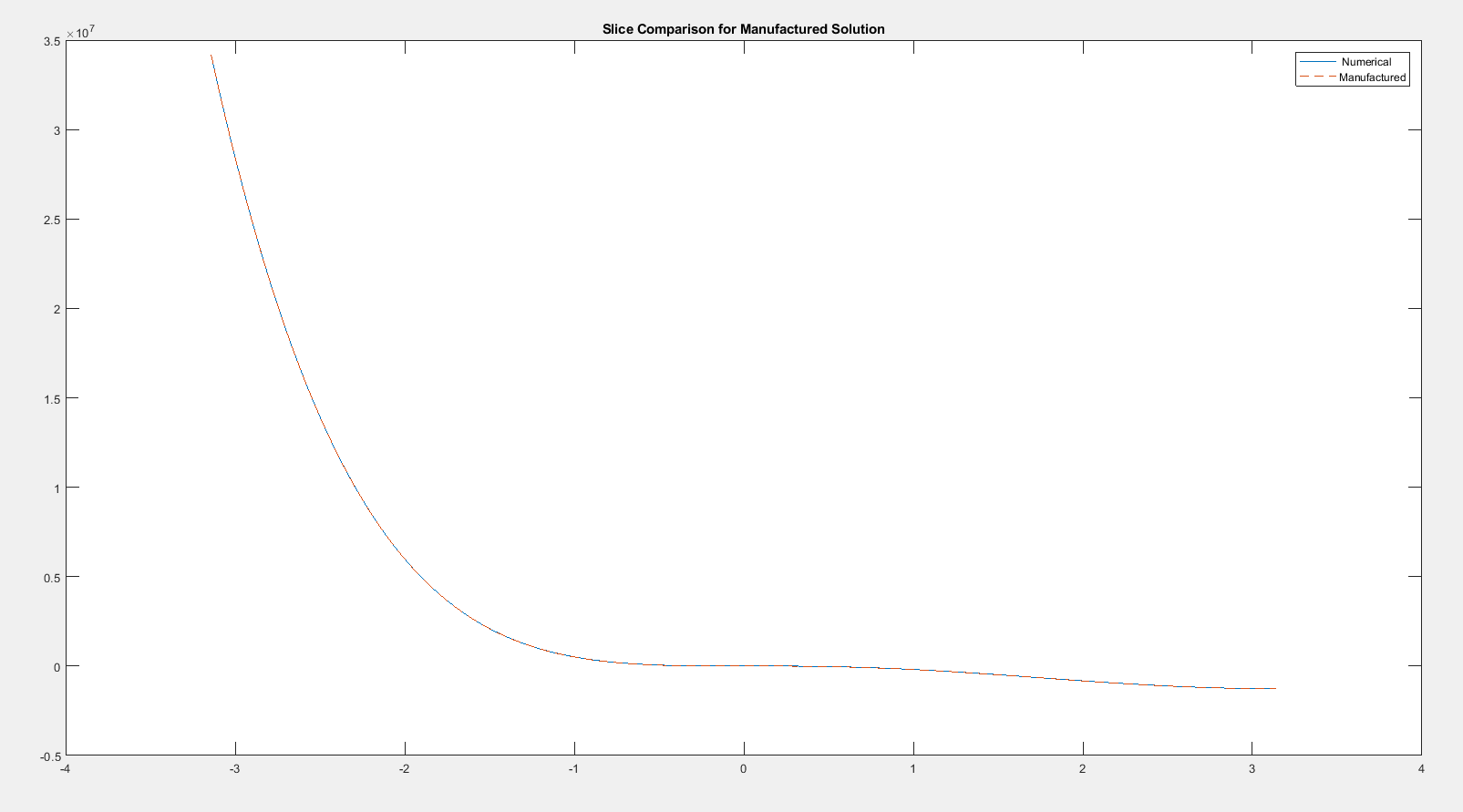
And finally boundary four:

Using these V derived boundary conditions along with, with previously discussed methods such as ghost nodes to accommodate the Neumann conditions, one can solve for V’ numerically by using the boundary conditions figured out by the manufactured V. This procedure would be done by using the same program type explained in the pseudo code part of the report, except that the F values for each node are replaced along with the boundary conditions when necessary. When done so, with a grid of 200 nodes, one gets the following:



Figure

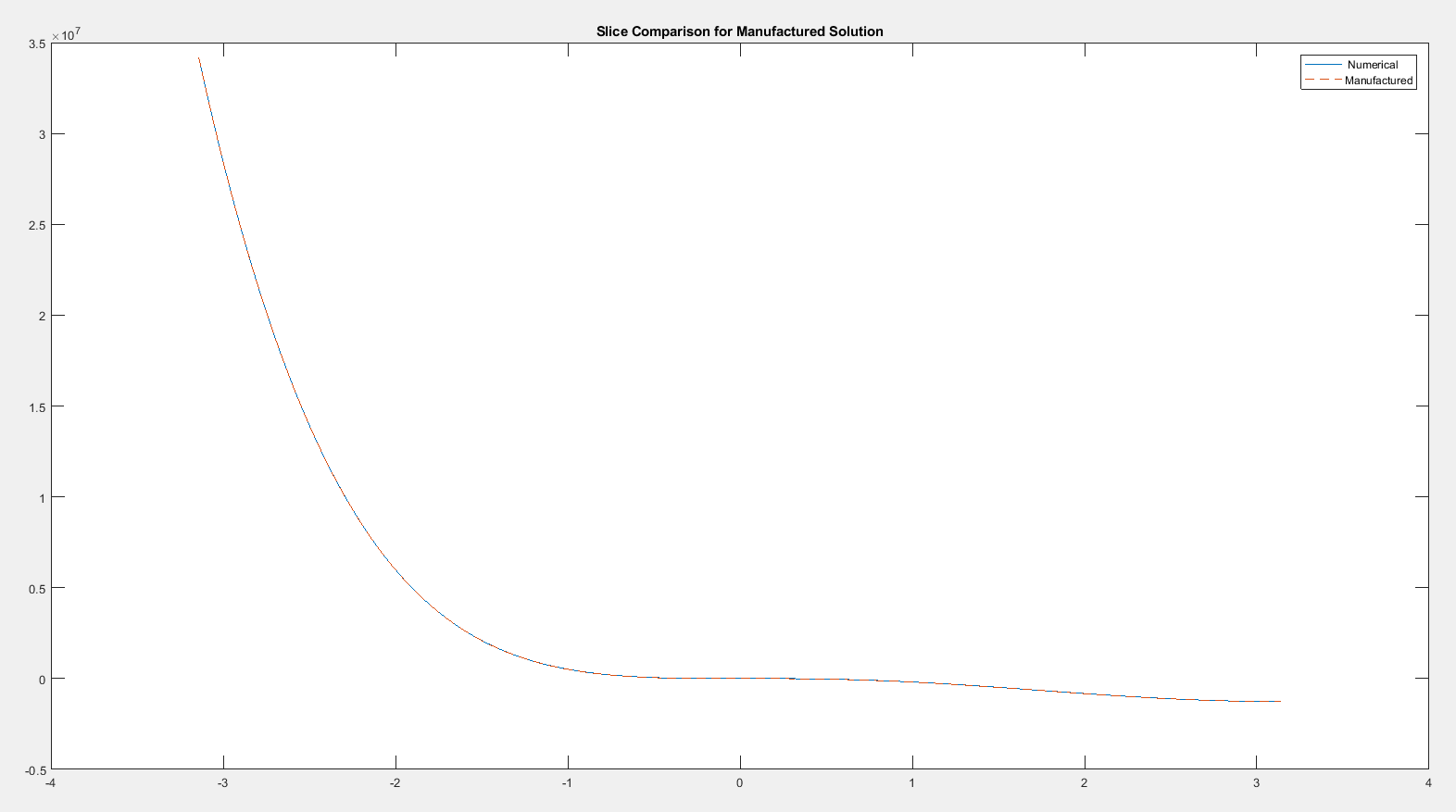
Figure 6 shows the graph of the two surfaces, the manufactured V and the numerically derived V’. Here the V’ was solved for by using the Gauss-Seidel method. This picture shows a fairly close overlap. For a better view, a ‘slice’ for each one is taken and they are compared. The following shows a slice comparing the numerical solution solved using the G-S method with the manufactured one. This slice is the line set about a quarter of the way on the x- axis and where the y values are allowed to vary throughout.



Figure

As one can see, both solutions overlap very well and closely.

The same procedure described just now was done again to compare the manufactured solution with the numerical solution of the iterative SOR solver. The results here are as follows on a ‘slice’. This slice is in the same location as the one for the GS solver, but now the results from the SOR solver are shown.



Figure

While both solvers eventually converge to become very close to the manufactured solution, each solver’s time taken and number of iterations required to reach it are significantly different. In tabular for they are as follow:

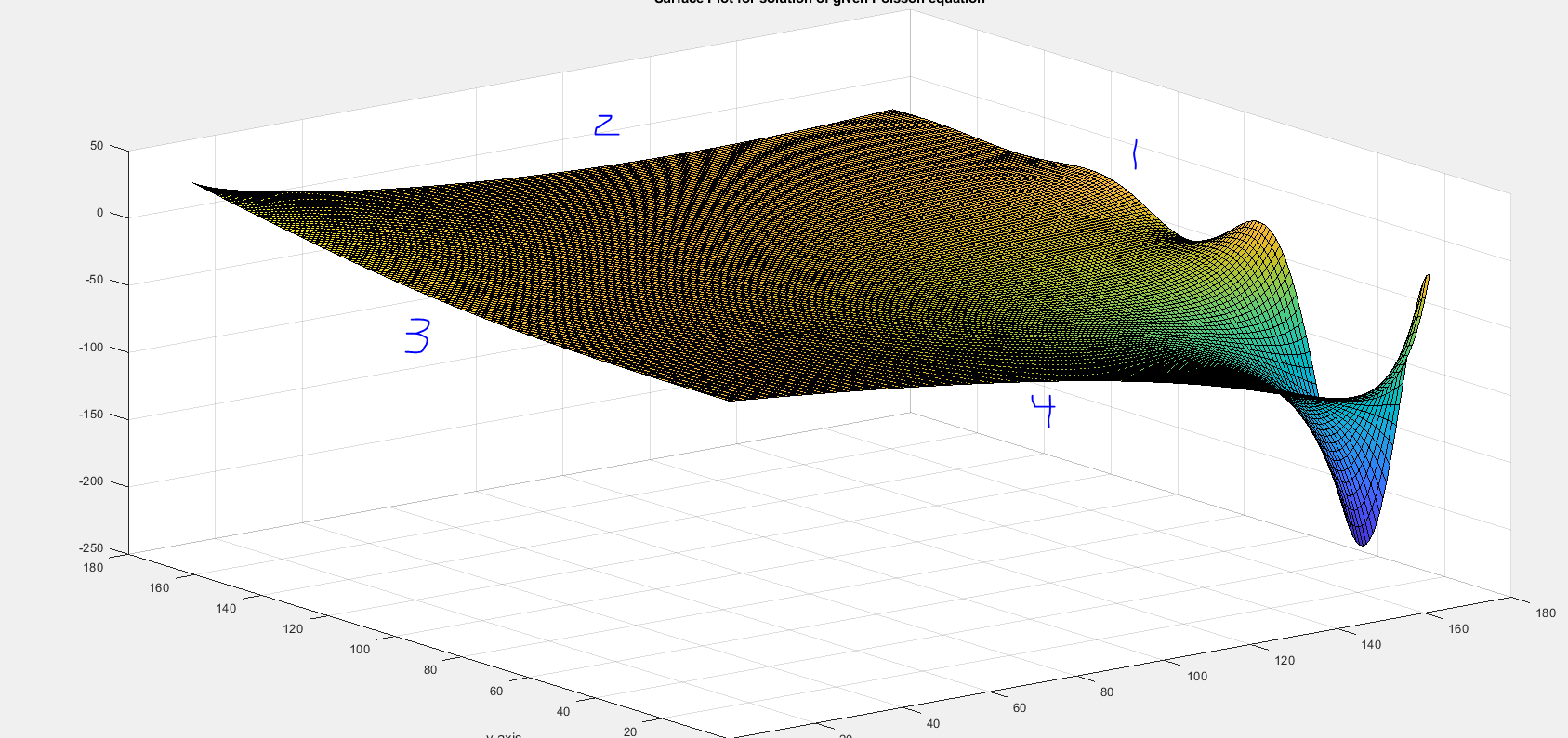


With the time being seconds. As one can see, the SOR method is significantly faster than the GS method. As learned in class, the addition of the old U being solved along with the omega factor give each step an additional correction that allows for a higher convergence. The reason why the elapsed time and number of iterations required for either step is still quite large is because of the ‘error’ term threshold. For both of these, this was chosen to be .00000001. Thus, the maximum normalized difference between the current value and the previous value for each node had to be such that.

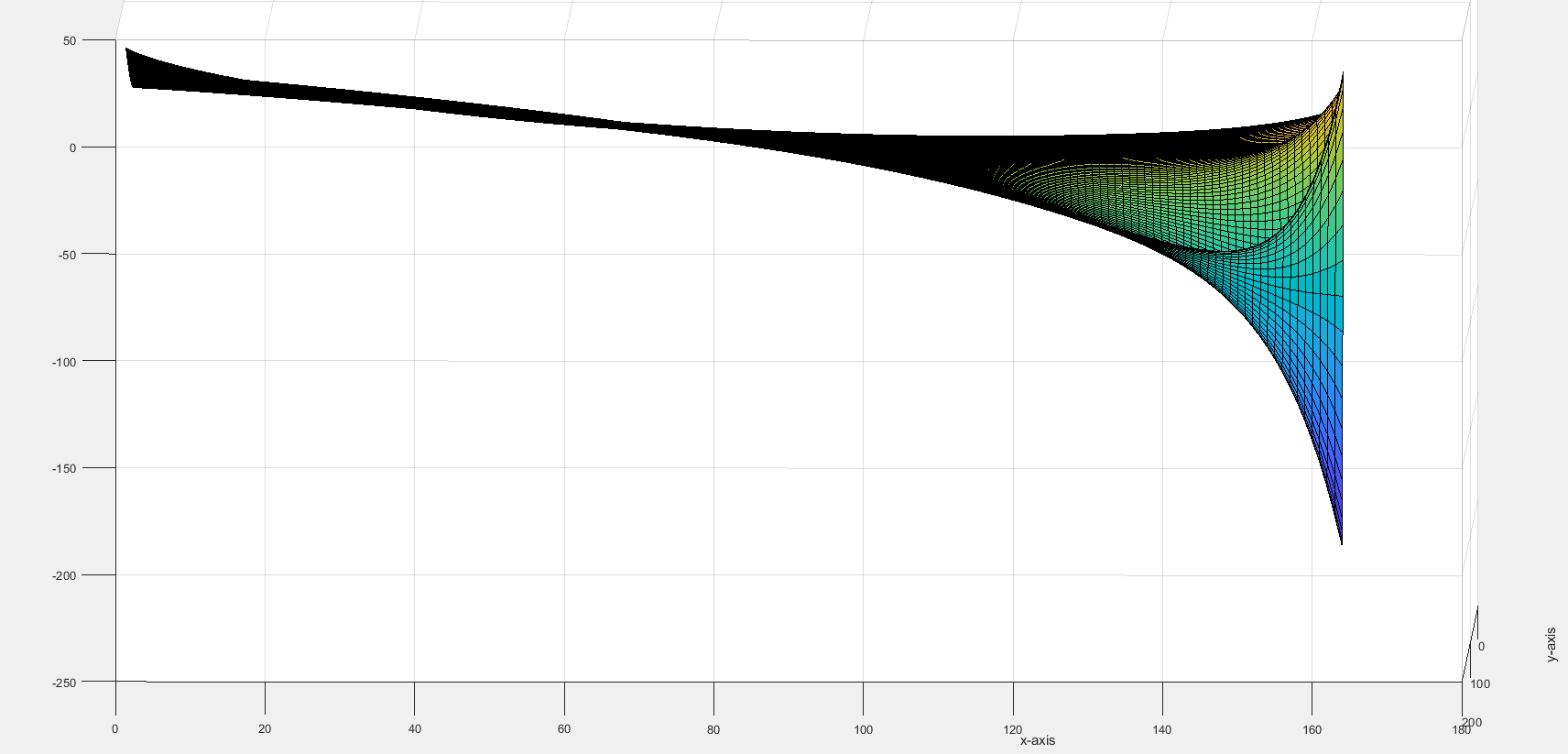
For the solver to stop working.

Now that all the verification and set up is complete, one can show the numerically derived solutions for the Poisson equation for the given function and boundary conditions.

The following four figures show the solution for the Gauss-Seidel method on a 160 by 160 node mesh from different view perspectives. The first image shows the solution with the four edges labeled appropriately as in the boundary ‘plate’ of image one. Here the PSI boundary condition is applied to one and the PHI boundary condition is applied to the edge labeled as three. The remaining two edges are Neumann and the program solves for them respectively.

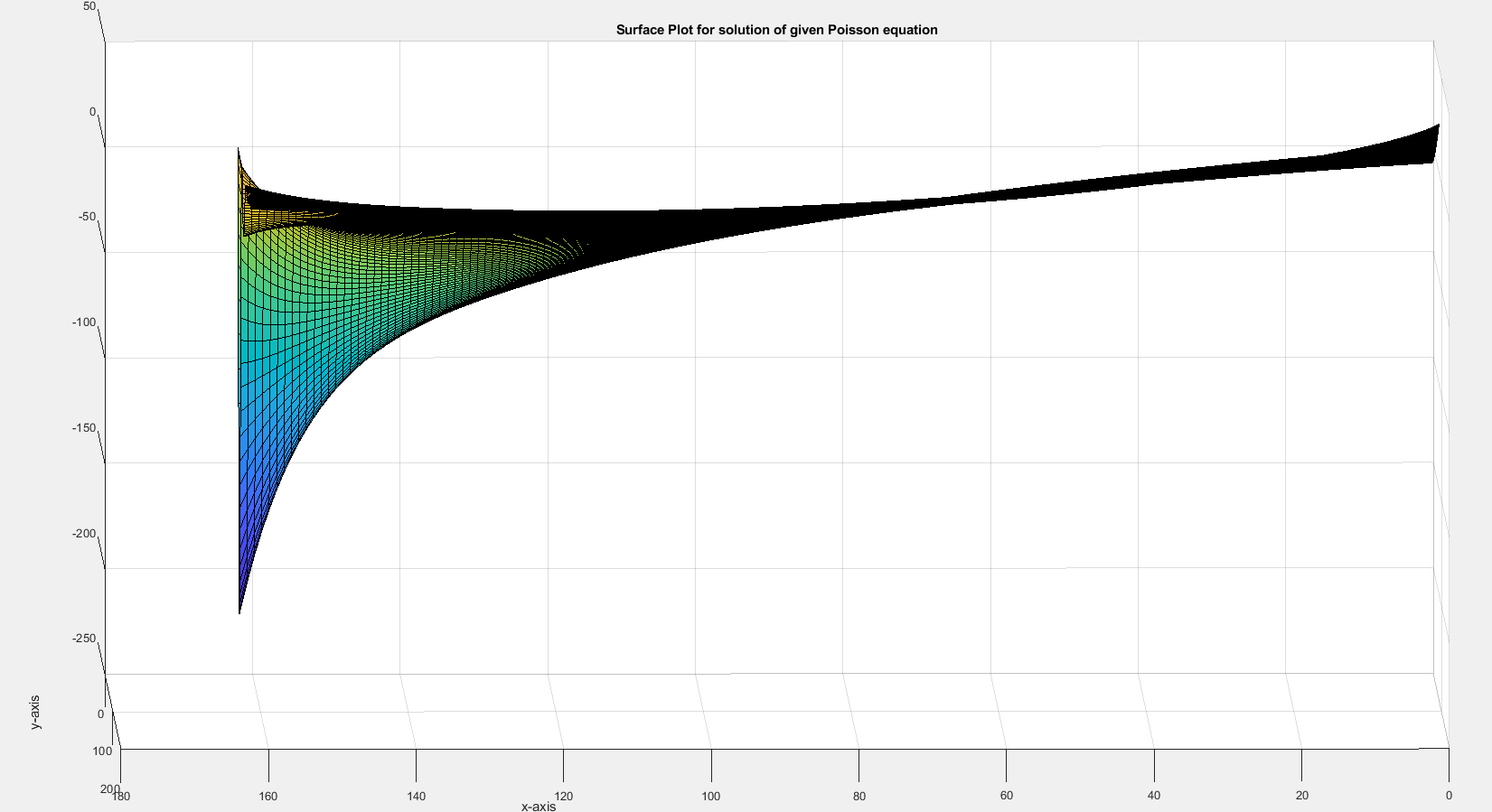


Figure



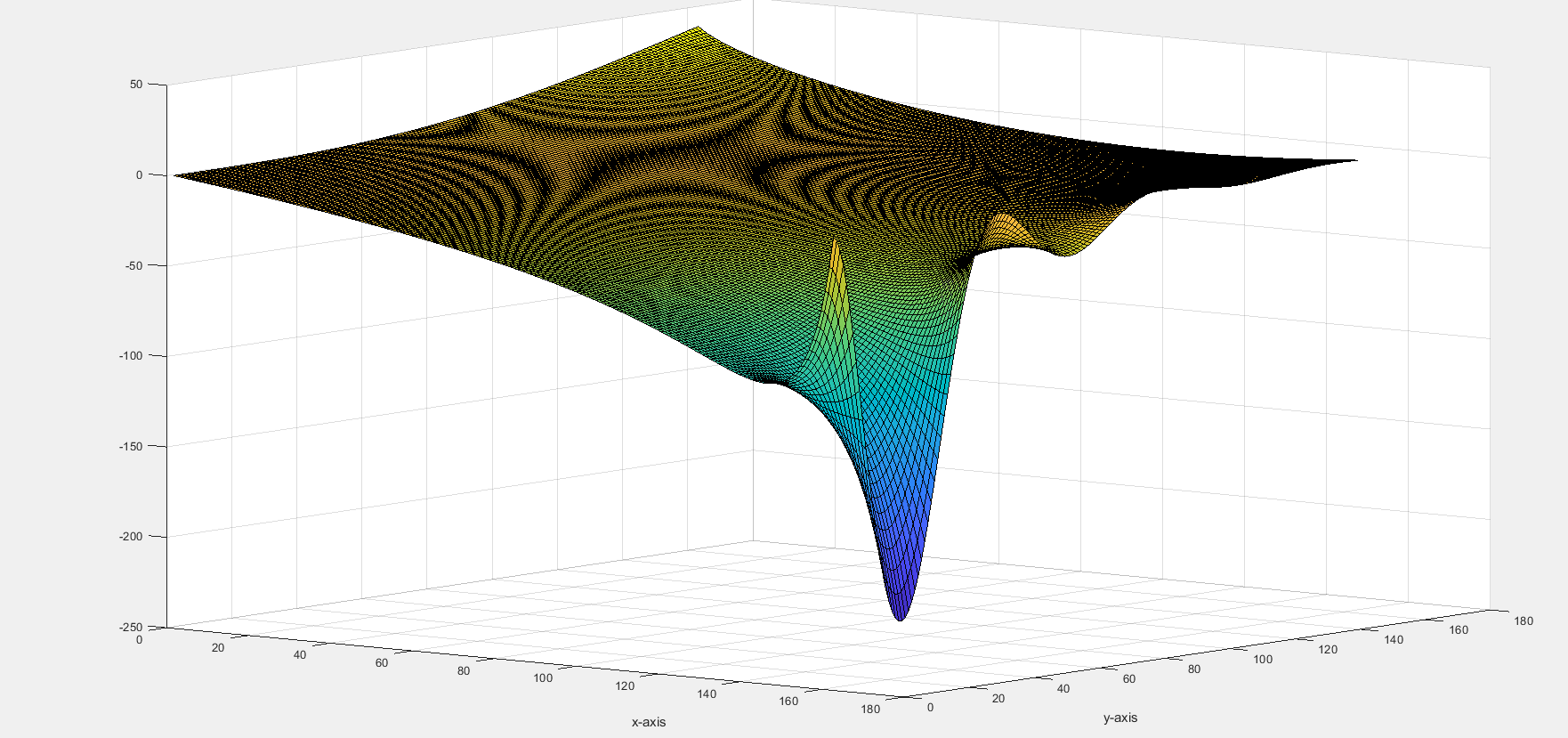
Figure

Figure 20 places an emphasis of what the figured out values for edge four are, and how they vary as one travels along this edge.



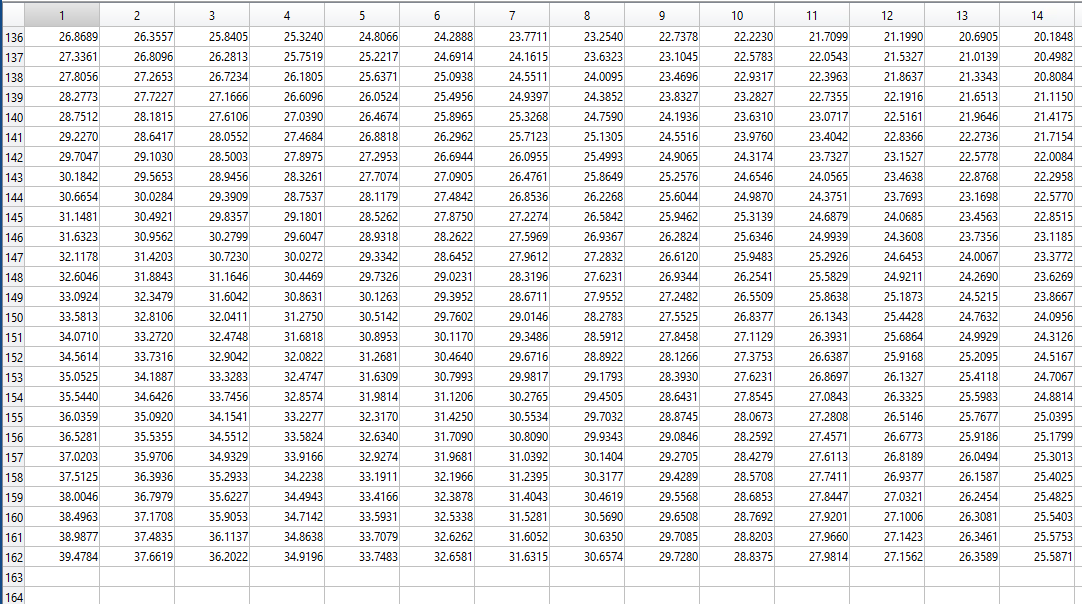
Figure

Figure 21 shows the figured out values along edge number 2.



Figure

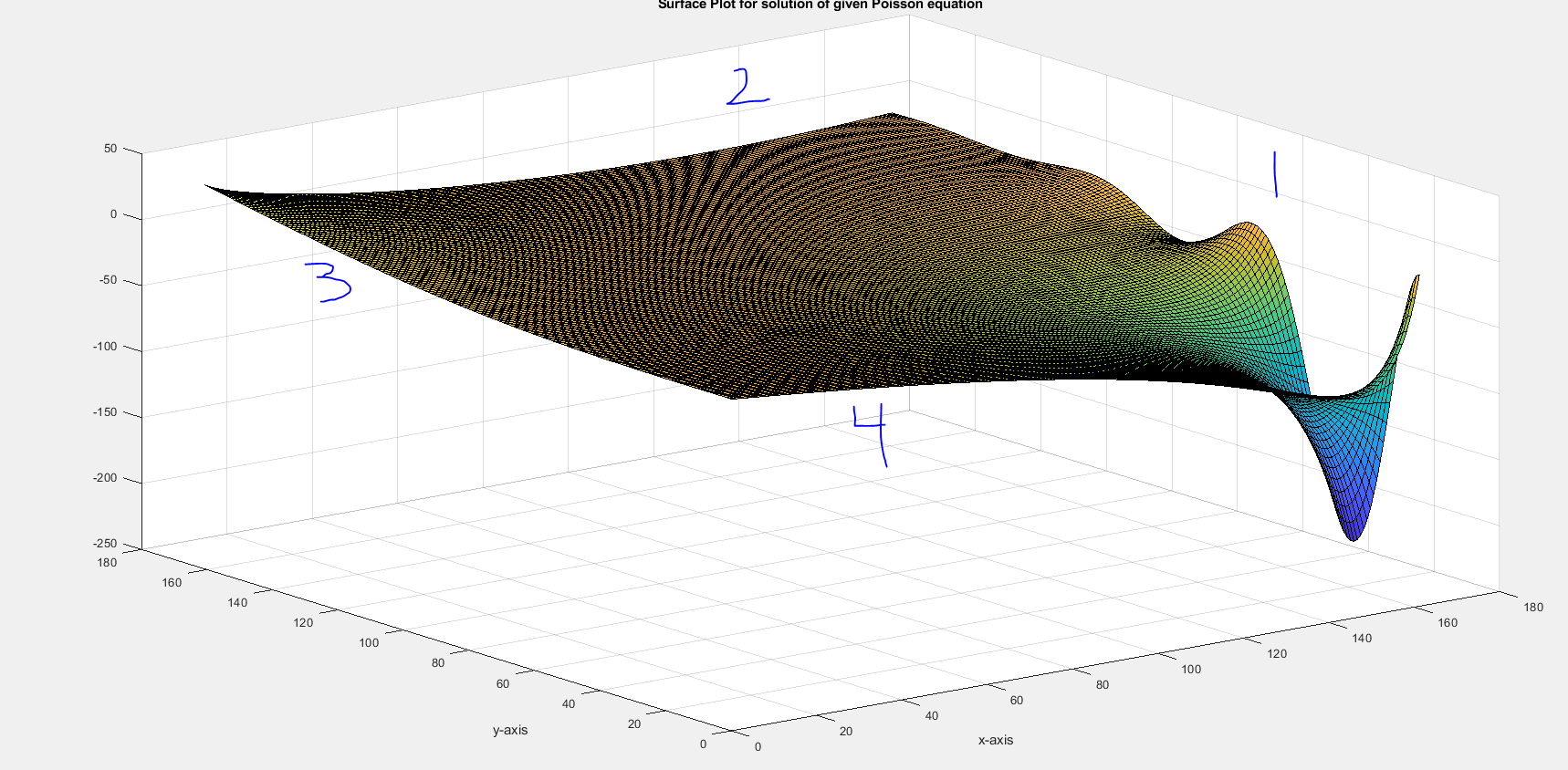
Figure 22 shows a view as seen as opposite of Figure 19. Based on these results, one can see that the PSI boundary conditions exerts the greatest influence on how the solution develops. This is because of its magnitude- especially in the corner where edges one and four meet. The PHI boundary condition of edge three also exerts an influence, but this is not as much due to its lesser magnitude. Values near the PSI condition are heavily influenced by it, but the further one gets, the less the influence is. The influence does appear to have a bigger range if the value at the edge is of greater magnitude. A closer inspection of edges two and four seemed to show that while they changed, they did not ‘slope’ along the y axis, satisfying the Neumann Boundary Conditions. The given F value seems to be the smallest, as it is just a two-dimensional sinusoidal function with a max of one and a minimum of minus one. Intuitively, the solution appears to be correct. The solution conforms to the given boundary conditions and develops smoothly throughout the ‘plate’. While the solution seems to be constant at some parts of the ‘plate’, this is not the case. The issue here is that the very large dip on one of the corners dwarfs everything else. The following figure shows some the node values of the solution on the corner diagonally across from this ‘dip’.



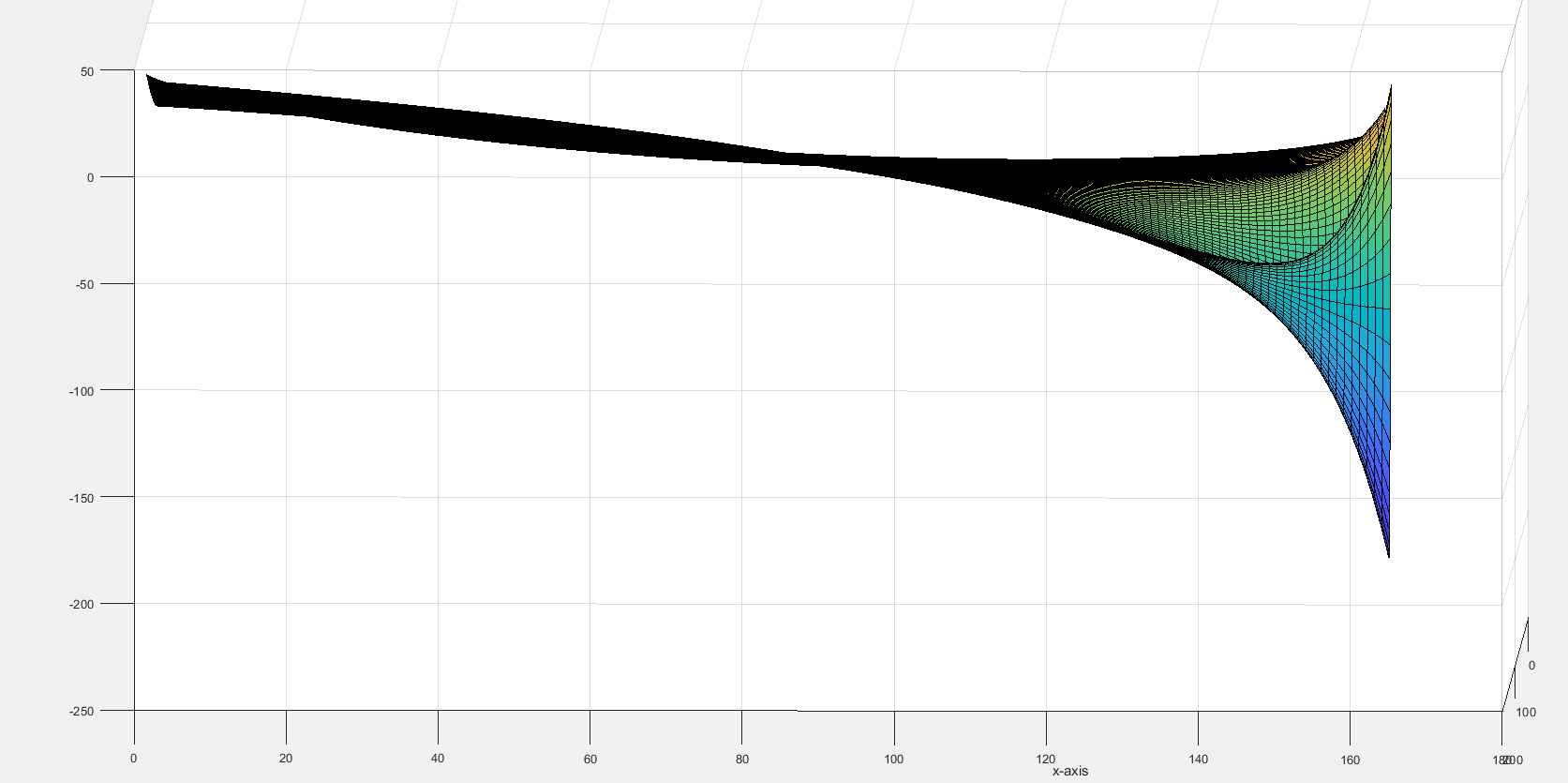
Figure

As one can see, the values do change. But because the ‘dip’ sets a bad resolution, these values are hard to observe to observe visually.

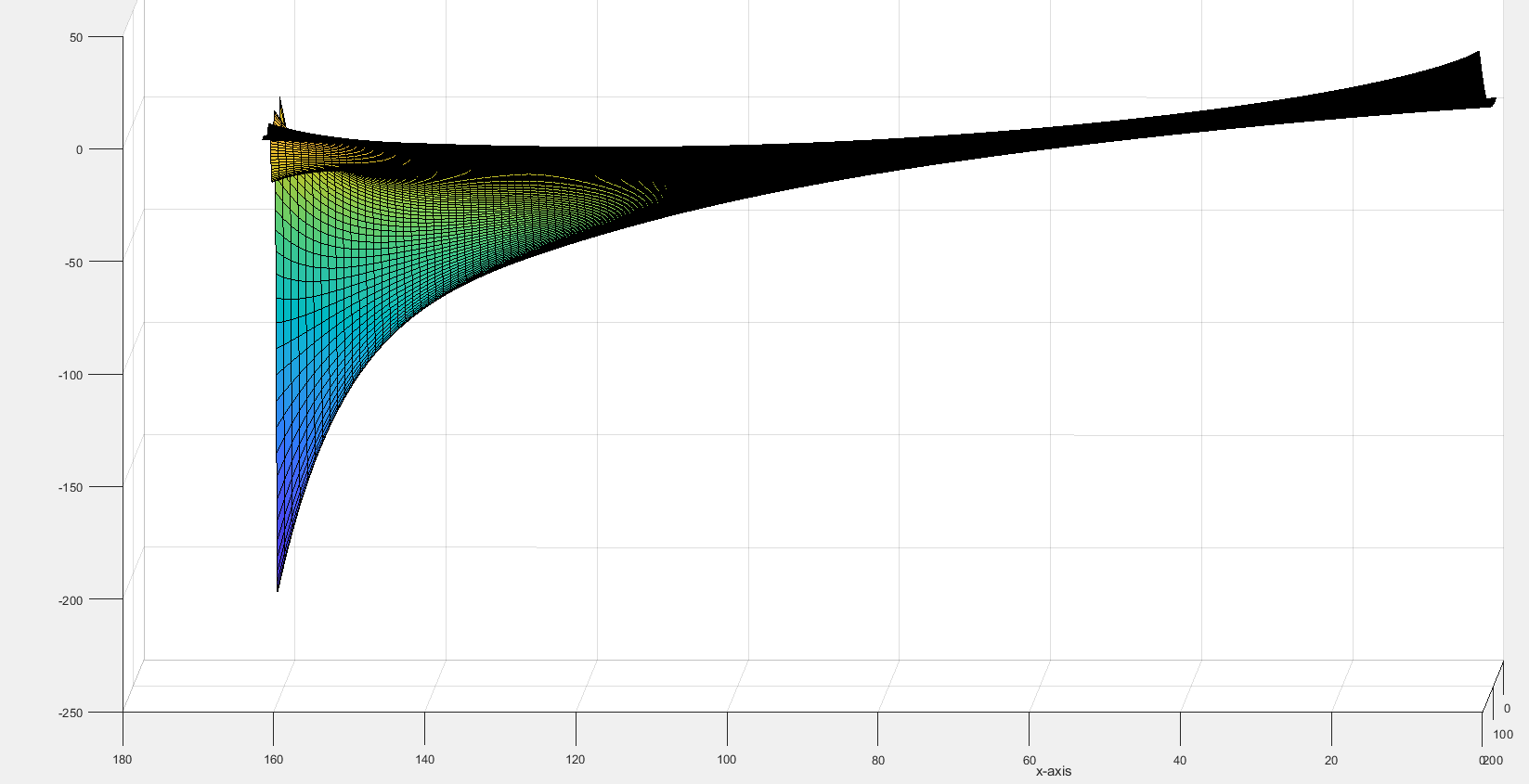
With the solution from the G-S method shown along with a limited discussion, one can now show the solution results from the other iterative method: the SOR.



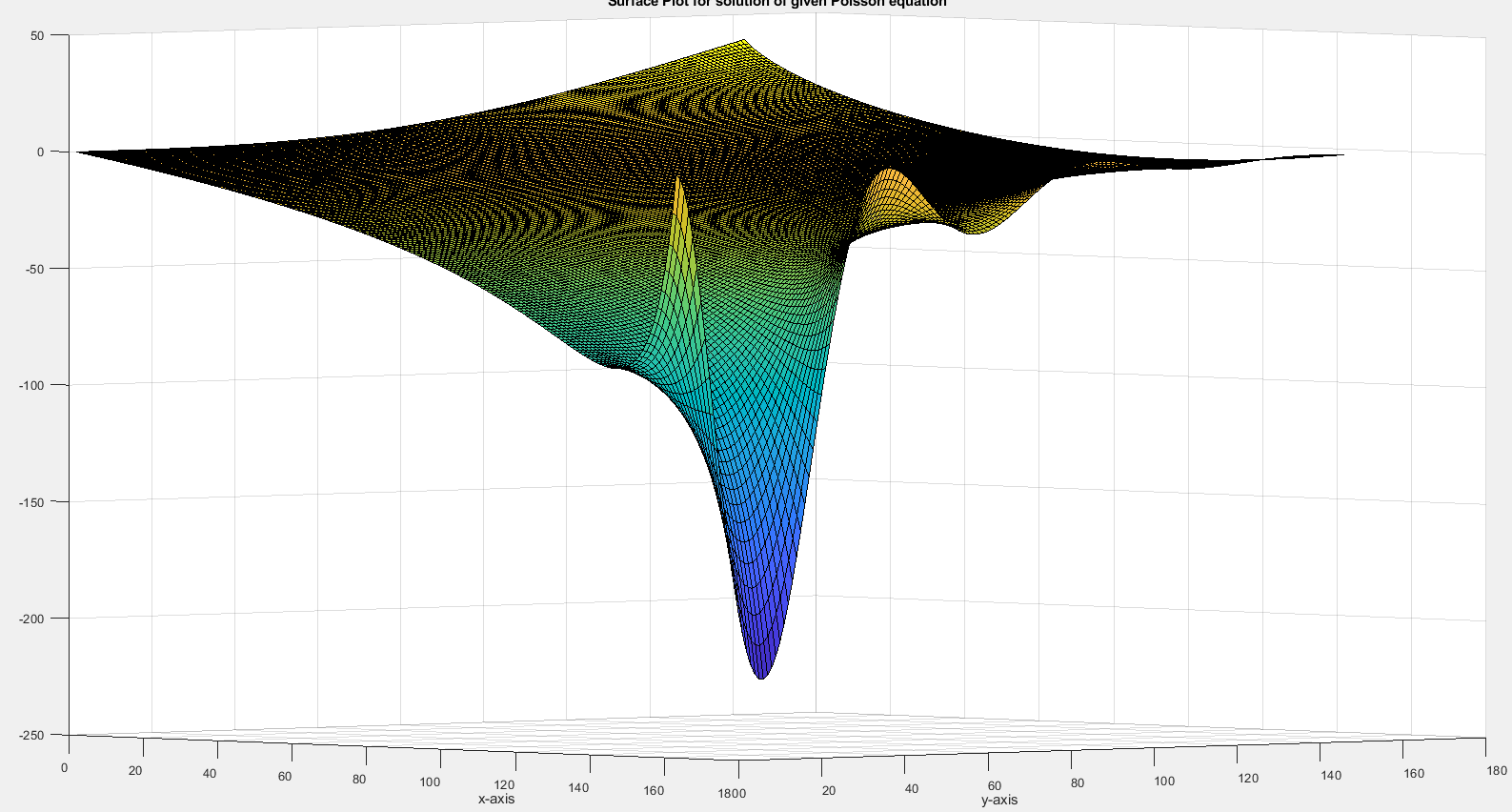
Figure



Figure



Figure



Figure

The preceding four images show the results of the SOR method, from different angles that try to imitate the same perspective of the G-S results. From this, it appears that both the G-S and the SOR give the same solution or very close. This is expected, as both methods are iterative and will converge towards to solution until the certain ‘error’ threshold is reached. However, there is one key difference between the G-S and the SOR methods when solving this Poisson problem. The SOR methods converges within the required tolerance significantly faster time-wise and undergoes a lesser amount of iterations to do so than the G-S method. A look into each method’s respective table during the grid independence study shows this. As stated previously, the reason for this is because the SOR has additional ‘correction factors’. These factors allow for the solutions at each node to converge to the specified tolerance significantly faster and use less computational resources (e.g. undergo less iterations). Because of this, one would recommend to attempt to use the SOR over the G-S method when solving these types of problems. An obstacle here is deciding on what value to use for omega. While going over the literature, the value of having omega as 1.5 was found. From these preceding results it appears to have worked well. But this should just be a decent start and more research to find what values of omega would be best should be undertaken to solve further problems even more efficiently.

Like all programs, it is very likely that this program used to solve the Poisson equation has bugs embedded in it. During the testing, the program sometimes crashed. From these crashes, bugs were identified and fixed, but it is possible that unknown bugs still remain. It is possible that one of these unknown bugs is making the program give a wrong solution to the stated problem. However, as of now, there are two main arguments against this. The first one is that the program was verified by the method of manufactured solutions. This means that the program can solve for and match a known solution to the Poisson problem. The other one is the grid independence, meaning that the grid size has no influence on the numerical solution to the problem.

Yet one can argue that perhaps in retrospect the method of manufactured solutions was not done properly enough. Maybe the made-up solution should have been manufactured in such a way to better imitate the boundary conditions. This way maybe program would have been better tested against the boundary conditions. Either way, bugs should not be seen as a total failure, but rather a necessary post processing step to make the program run better. Like all engineering matters, programs to solve equations should be seen as open-ended affairs that are capable of being both perpetually fixed and optimized to run better.

**References/Sources**

Prosperetti. *Numerical Approximation of Derivatives*. January 24, 2018.

https://www3.nd.edu/~gtryggva/CFD-Course/2011-Lecture-19.pdf

https://ark.intel.com/products/88188/Intel-Core-i5-6600-Processor-6M-Cache-up-to-3\_90-GHz

Roache, Patrick. *Code Verification by Method of Manufactured Solutions*. Transactions of the ASME, Volume 124, March 2002.

Prosperetti. *Verification and Validation*. April 30, 2018

<https://me.ucsb.edu/~moehlis/APC591/tutorials/tutorial5/node3.html>