[Company name]

Numerical Analysis of ODEs using Matlab Coursework 2018

[Document subtitle]

Rallis, Vasilios
//TODO: ADD names and CID here

Contents

1. RL Circuit

//Write something here

Exercise 1

//Write something here

Something else you want to mention in exercise 1

//Write something here

3. Relaxation

The approaches outlined so far will only work for Initial Value Problem (IVP) in which all the initial conditions are given for the same value of the independent variable. In Boundary Valued Problems (BVP); however, the initial conditions are not given for the same value of the independent variable. In such cases, solutions might not even exist and even if they do, we cannot use the approaches for solving IVP (e.g. the RK method) since there is no guarantee that the desired final boundary value will be reached.

In BVP, the method of Finite Differences can be used to compute an estimation for the solution of the differential equation. Effectively, the method of finite differences breaks up the interval in question into equal segments and then uses the equal segments to develop a set of simultaneous equations. The solution of these equations will give an approximation to the solution of the differential equation.

Developing Tools

Before using the Finite Difference method, we must develop some tools. Specifically, we need to find some estimations for the first and second derivative of a function.

The derivative of a function f at x_0 is

$$f'(x_0) = \lim_{h\to\infty} \frac{f(x_0+h) - f(x_0)}{h}$$

Given that h is small, an obvious approximation of $f'(x_0)$ is

$$\frac{f(x_0+h) - f(x_0)}{h}$$

This formula is known as the forward difference if $h > 0$ and the backward difference if $h < 0$.

We can also develop another more accurate approximation of $f'(x_0)$ using the Taylor Series. Using the Taylor series for $f(x)$ around the point x_0 we can estimate $f(x_0+h)$ and $f(x_0-h)$ to be

$$f(x_0+h) = f(x_0) + f'(x_0)h + \frac{f^{(2)}(x_0)h^2}{2!} + \frac{f^{(3)}(x_0)h^3}{3!} + \cdots$$

$$f(x_0-h) = f(x_0) - f'(x_0)h + \frac{f^{(2)}(x_0)h^2}{2!} - \frac{f^{(3)}(x_0)h^3}{3!} - \cdots$$

Subtract the two equations above to get

$$f(x_0+h) - f(x_0-h) = 2f'(x_0) + \frac{2f^{(3)}(x_0)h^3}{3!} + \cdots$$

$$f'(x_0) = \frac{f(x_0+h) - f(x_0-h)}{2h} - \frac{f^{(3)}(x_0)h^2}{3!} + \cdots$$

$$f'(x_0) = \frac{f(x_0 + h) - f(x_0 - h)}{2h} + Error$$

Where the error is $O(h^2)$.

The same can be done for the second derivative; however, instead of subtracting equation (XX) from equation (XX), we add them to get.

$$f^{(2)}(x_0) = \frac{f(x_0 + h) - 2f(x_0) + f(x_0 - h)}{h^2} + Error$$

Where the error again is $O(h^2)$

These equations can also be extended to partial derivative. Thus, for a function $u(x, y)$

$$\frac{\partial u}{\partial x} \approx \frac{u(x + h, y) - u(x - h, y)}{2h}$$

$$\frac{\partial^2 u}{\partial x^2} = \frac{u(x + h, y) - 2u(x, y) + u(x - h, y)}{h^2}$$

Finite Differences

Given that we have the Laplace Equation

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0$$

We divide the x and y intervals into N segments of equal length h where

$$h = \frac{1}{N}$$

Let $x_i = ih$ and $y_j = jh$. Thus

$$x_0 = 0, x_1 = h, x_2 = 2h \ldots$$

$$y_0 = 0, y_1 = h, y_2 = 2h \ldots$$

Furthermore, let $U_i^j = u(x_i + y_j)$

Using (XX) we can rewrite (XX) as

$$\frac{U_{i+1}^j - 2U_i^j + U_{i-1}^j}{h^2} + \frac{\left(U_i^{j+1} - 2U_i^j + U_i^{j-1}\right)}{h^2} = 0$$

$$U_{i+1}^j + U_{i-1}^j + U_i^{j+1} + U_i^{j-1} - 4U_i^j = 0$$

In the simplest form, we are interest in the values of Laplace's equation on a rectangle taken as $0 \le x \le a$ and $0 \le y \le b$. We divide x and y according to equations (XX) and (XX). To solve the equation, the values of $u(x, y)$ must be known on the boundaries $u(x, b) = \Phi_1(x), u(a, y) = \Phi_2(y), u(x, 0) = \Phi_3(x), u(0, y) = \Phi_4(y)$

The boundary functions give us the values of U_i^j on the edges of the rectangle. For example, if we let $a = 1$, $b = 1$ and $N = 3$ for simplicity. Using equation (XX) we can generate a system of equations

$$E1: U_2^1 + U_1^2 - 4U_1^1 = -(U_0^1 + U_1^0)$$

$$E2: U_1^1 + U_2^2 - 4U_2^1 = -(U_2^0 + U_3^1)$$

$$E3: U_1^1 + U_2^2 - 4U_1^2 = -(U_0^2 + U_1^3)$$

$$E4: U_1^2 + U_2^1 - 4U_2^2 = -(U_2^3 + U_3^2)$$

This can also be written in matrix form $A\boldsymbol{x} = \boldsymbol{b}$

$$\begin{bmatrix} -4 & 1 & 1 & 0 \\ 1 & -4 & 0 & 1 \\ 1 & 0 & -4 & 1 \\ 0 & 1 & 1 & -4 \end{bmatrix} \begin{bmatrix} U_1^1 \\ U_2^1 \\ U_1^2 \\ U_2^2 \end{bmatrix} = - \begin{bmatrix} U_0^1 + U_1^0 \\ U_2^0 + U_3^1 \\ U_0^2 + U_1^3 \\ U_2^3 + U_3^2 \end{bmatrix}$$

This matrix can then be solved using matrix methods. However, for large matrices it gets computationally expensive.

Jacobi and Guass-Siedel Iterative methods

Iterative methods are used to matrix systems with a high number of zero entries since in these cases, they are often more efficient than matrix methods (e.g. Gaussian elimination).

An iterative method solves an $n \times n$ linear system in the form $A\boldsymbol{x} = \boldsymbol{b}$ by first estimating an initial solution $\boldsymbol{x}^0$ (basically a guess) and then generating a sequence of vectors which represent more accurate solutions until some maximum error tolerance is reached.

The Jacobi iterative method generates the component x_i^k of $\boldsymbol{x}^k$ from the components of $\boldsymbol{x}^{k-1}$ using

$$x_i^k = \frac{1}{a_{ii}} \left(-\sum_{\substack{j=1 \\ j \neq i}}^{n} \left(a_{ii} x_j^{k-1} \right) + b_i \right)$$

We continue to do this until we generate a solution where

$$\frac{\|\boldsymbol{x}^k - \boldsymbol{x}^{k-1}\|_\infty}{\|\boldsymbol{x}^k\|_\infty} < percetage\ error\ tolerance$$

Where $\|\boldsymbol{x}\|_\infty = \max(|x_1|, \dots, |x_n|)$ (i.e. the maximum norm of the vector)

The Guass-Siedel method is an improvement to the Jacobi. It comes from the observation that we only use the elements of $\boldsymbol{x}^{k-1}$ to computer x_i^k. However, for $i > 1$ we have already computer

some of the components of $\boldsymbol{x}^k$ which are supposed to be better approximation of the actual solutions. Thus, we would use them to compute x_i^k. The method now becomes

$$x_i^k = \frac{1}{a_{ii}}\left(-\sum_{j=1}^{i-1}\left(a_{ii}x_j^k\right) - \sum_{j=i+1}^{n}\left(a_{ii}x_j^{k-1}\right) + b_i\right)$$

The Guass-Siedel method is ideal for solving the matrix equation (XX) that we generated above since for each row we can only have a maximum of 4 non-zero columns. Effectively, the algorithm now becomes

$$\left(U_i^j\right)_{new} = \left(U_i^j\right)_{old} + r_i^j$$

Where the residual r_i^j is given by

$$r_i^j = \frac{U_{i+1}^i + U_{i-1}^j + U_i^{(j+1)} + U_i^{j-1} - 4U_i^j}{4}$$

In this case, r_i^j is similar to the $percetage\ error\ tolerance$ that we defined above be we didn't divide it.

Exercise 4

In this exercise, we effectively use the Guass-Siedel method to plot the solution to Laplace's equation given different initial condition.

The code for exercise 4 is shown below

```
%clean from previous run
close all;
clear;
clc;

%Choose gridsize and residual size tolerance
h = 0.05;
max_res = 1e-6;

%Choose the border functions
fi1 = @(x) 1;
fi2 = @(y) 0;
fi3 = @(x) heaviside(x-0.2)-heaviside(x-0.8);
fi4 = @(y) 0;

%Setting up
x(1) = 0;
xf = 1;
yf = 1;
n = round((xf - x(1))/h);
```

```
x = 0:h:n*h;
y = 0:h:n*h;
z = zeros(length(x),length(y));


sum = 0;

for i = 1:length(x)
    z(i,length(y)) = fi1(x(i));
    z(length(x),i) = fi2(y(i));
    z(i,1) = fi3(x(i));
    z(1,i) = fi4(y(i));
end

%Calculate the average value
total = 4*(n + 1) - 4;
avg = sum/total;

%Set all the inner points as this average value
for i = 2:length(x) - 1
    for j = 2:length(y) - 1
            z(i,j) = avg;
    end
end

while (true)
    %set the maximum residue found in this iteration to 0
    %it will be updated in the for loop
    r_max = 0;
    for i = 2:length(x) - 1
        for j = 2:length(y) -1
            %Calculate the residue
            r = (z(i+1,j) + z(i-1,j) + z(i,j+1) + z(i,j-1) -
4*z(i,j))/4;
            %Set the new value of the point
            z(i,j) = z(i,j) + r;
            if(abs(r) > r_max)
                r_max = abs(r);
            end
        end
    end
    %if the meximum residue value found in this iteration is smaller
    %than our maximum tolerence break
    if(r_max < max_res)
        break;
    end
end

%It is important not to forget to transpose because of the way matlab
%handles matrices
mesh(x,y,z');
```

```
xlabel('x-axis');
ylabel('y-axis');
zlabel('z-axis');
title('Solution to Laplace Equation');
```

For the boundary conditions given

$u(0,y) = u(1,y) = 0$, for $0 < y < 1$

$u(x,1) = 0$, for $0 < x < 1$

$u(x,0) = 0$, for $0 < x < 0.2$ and $0.8 < x < 1$

$u(x,0) = 1$, for $0.2 < x < 0.8$

With $h = 0.02$ and $residue = 10^{-4}$, the following plot was generated

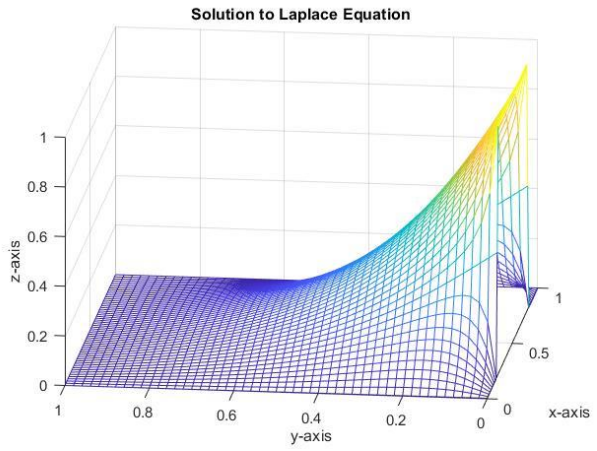


Figure 1

An interesting observation is how the $residue$ affects the overall final shape of the plot. We tried for 3 different residue values

With $h = 0.02$ and $residue = 10^{-2}$, the following plot was generated

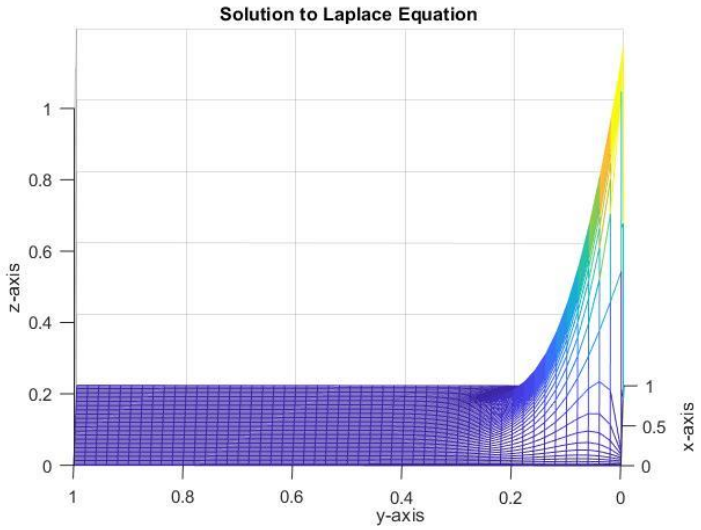


Figure 2

With $h = 0.02$ and $residue = 10^{-4}$, the following plot was generated

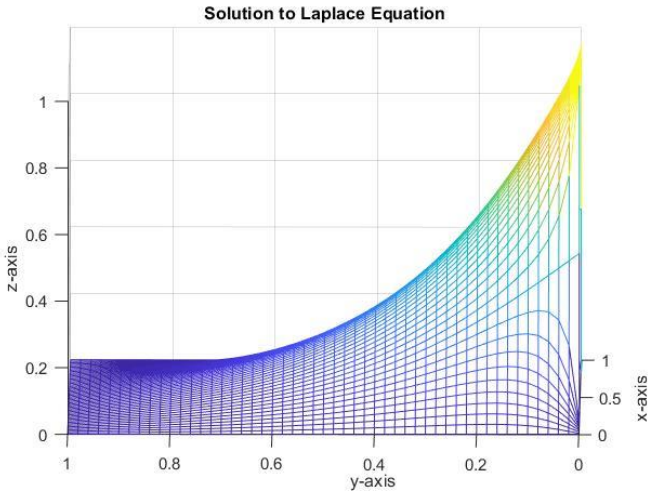


Figure 3

With $h = 0.02$ and $residue = 10^{-6}$, the following plot was generated

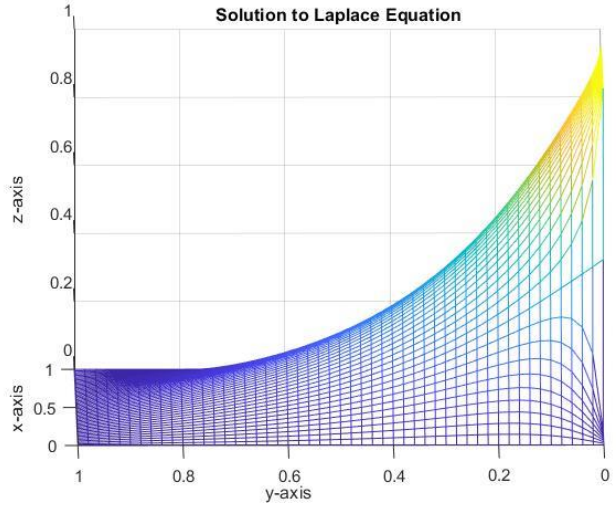


Figure 4

From the specify angle, we can see that as we decrease the $residue$, because the plot becomes more accurate, it takes longer for the graph to go to zero (i.e. the slope of the graph decreases since 2 adjacent points have values that are closer to each other).

We also tried to estimate the solution for more complex boundary conditions.

For the boundary conditions given by

$u(0, y) = u(1, y) = 0$, for $0 < y < 1$

$u(x, 1) = \sin(6\pi x)$, for $0 < x < 1$

$u(x, 0) = 0$, for $0 < x < 1$

With $h = 0.02$ and $residue = 10^{-6}$, the following plot was generated

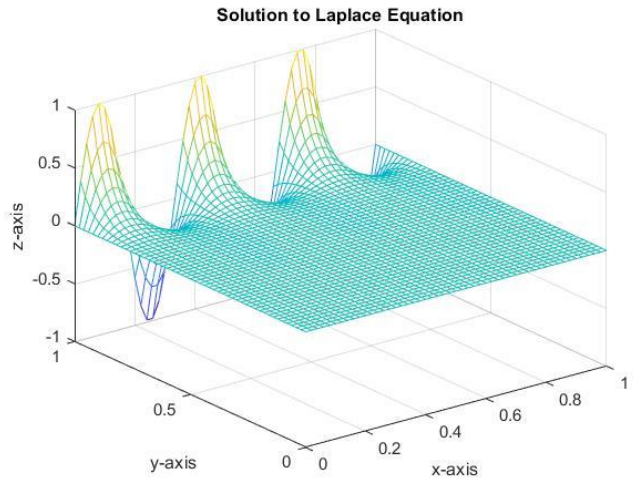


Figure 5

For the boundary conditions given by

$u(0, y) = u(1, y) = \sin(\pi y)$, for $0 < y < 1$

$u(x, 1) = u(x, 0) = \sin(\pi x)$, for $0 < x < 1$

With $h = 0.02$ and $residue = 10^{-6}$, the following plot was generated

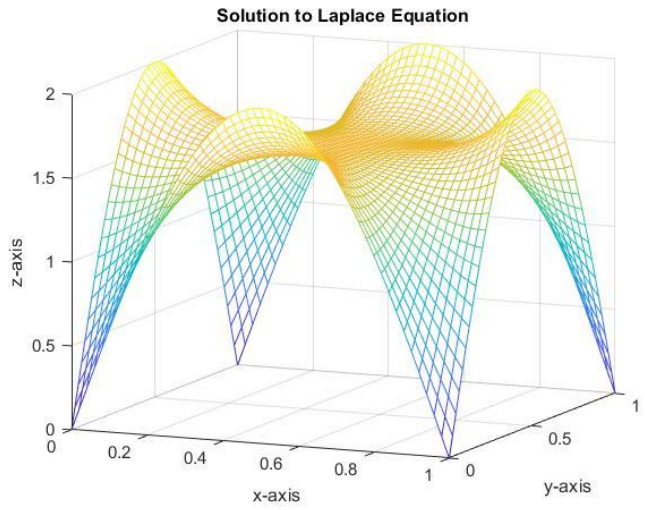


Figure 6

When the boundary conditions do not match at the corners, depending on which boundary it is, the code will select on of the two values provided. An example of this is shown below

For the boundary conditions given by

$u(0, y) = u(1, y) = 0,$ for $0 < y < 1$

$u(x, 1) = 1\,,$ for $0 < x < 1$

$u(x, 0) = 0,$ for $0 < x < 1$

With $h = 0.04$ and $residue = 10^{-6}$, the following plot was generated

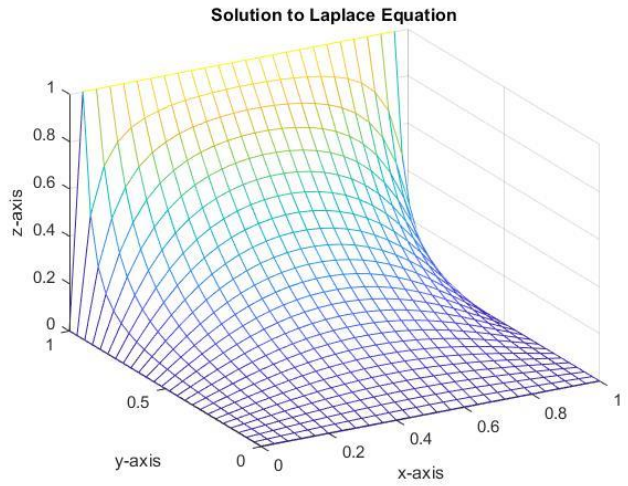


Figure 7

As we can see, the discontinuity in the boundary conditions doesn't really affect the plot since the corners are never considered in the estimation of the other points.

Exercise 5

This method can also be extended nicely to the Poison equation which has the form

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = g(x, y)$$

By simply setting the residual r_i^j to be

$$r_i^j = \frac{U_{i+1}^i + U_{i-1}^j + U_i^{(j+1)} + U_i^{j-1} - 4U_i^j - h^2 g_i^j}{4}$$

The code for exercise 5 (i.e. relaxation2.m) is shown below:

```
close all;
clear;
clc;

%Choose gridsize and residual size
h = 0.02;
max_res = 1e-3;

%Choose the border functions
fi1 = @(x) -2*x^2 + 3*x + 2;
```

```
fi2 = @(y) -y^2 + 3*y + 1;
fi3 = @(x) -2*x^2 + x + 2;
fi4 = @(y) -y^2 + y + 2;

%Choose forcing term and the actual function to compare
g = @(x,y) -6;

%Setting up
x(1) = 0;
xf = 1;
yf = 1;
n = round((xf - x(1))/h);
x = 0:h:n*h;
y = 0:h:n*h;
z = zeros(length(x),length(y));

%Calculating the actual values
[X, Y] = meshgrid(0:h:n*h);
% The U function represents the solution to the Poison Equation
U = -2*X.^2 - Y.^2 + X + Y + 2*X.*Y + 2;

sum = 0;

for i = 1:length(x)
    z(i,length(y)) = fi1(x(i));
    z(length(x),i) = fi2(y(i));
    z(i,1) = fi3(x(i));
    z(1,i) = fi4(y(i));
end

%Calculate the average value
total = 4*(n + 1) - 4;
avg = sum/total;

%Set all the inner points as this average value
for i = 2:length(x) - 1
    for j = 2:length(y) - 1
            z(i,j) = avg;
    end
end

while (true)
    %set the maximum residue found in this iteration to 0
    %it will be updated in the for loop
    r_max = 0;
    for i = 2:length(x) - 1
        for j = 2:length(y) -1
            %Calculate the residue
            r = (z(i+1,j) + z(i-1,j) + z(i,j+1) + z(i,j-1) -
h^2*g(x(i),y(j)) - 4*z(i,j))/4;
            %Set the new value of the point
```

```
            z(i,j) = z(i,j) + r;
            if(abs(r) > r_max)
                r_max = abs(r);
            end
        end
    end
    %if the meximum residue value found in this iteration is smaller
    %than our maximum tolerence break
    if(r_max < max_res)
        break;
    end
end

%It is important not to forget to transpose because of the way matlab
%handles matrices
%Uncomment the following line to get the estimation for the solution
%mesh(x,y,z');

%Plot error
er = abs(z' - U); mesh(X,Y,er);
xlabel('x-axis');
ylabel('y-axis');
zlabel('z-axis');
title('Solution to the Poison Equation');
```

We only must change a few lines to generate the solution for the Poison equation! Specifically, the highlighted lines indicate the changes that have been made to the code.

In this case, we also generate an error plot which shows the error made in calculating each point on the graph. We can choose between plotting the estimation and the error by just changing two lines.

With $h = 0.02$ and $residue = 10^{-3}$, the following estimation plot was generated

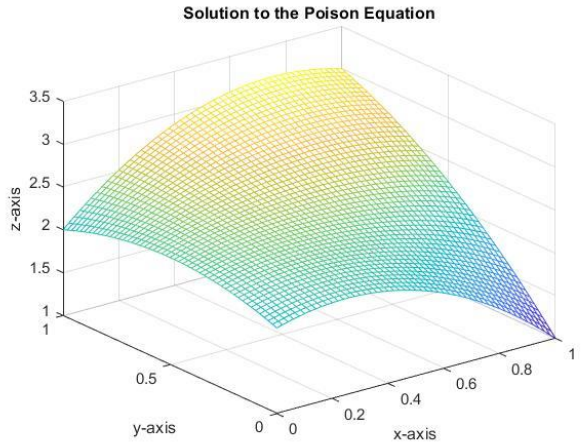


Figure 8

For the same inputs, the following error plot was generated

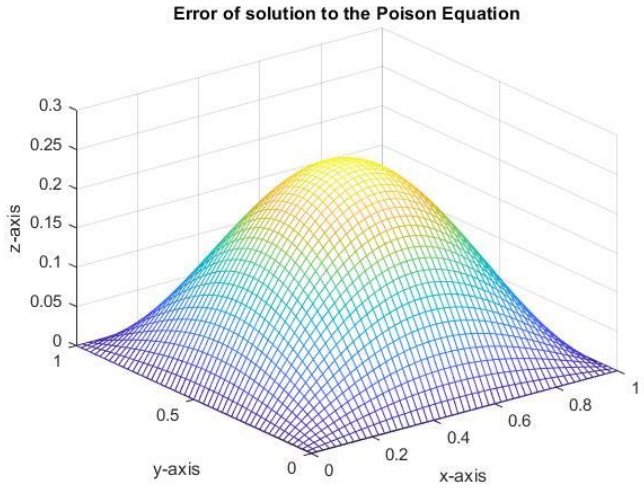


Figure 9

As we can see from figure 9, the error gets larger as we move closer to the center of the plot. Initially, this might seem sort of strange; however, we must realize that the estimations made for the points in the center are themselves based on points closer to the edges which are also just estimations in themselves. The closer to the center a point is, the more estimations the point was based on. Hence the points closer to the center will have the highest error.

Another observation is the maximum error is much larger than the $residue$ that we set. However, the residue assumes that the points nearby the point we are estimating are the correct values which isn't true in this case.

Commented [RV1]: This needs to be rephrased

Nevertheless, we can see how the $residue$ affects the error of the plot in the following figures

With $h = 0.02$ and $residue = 10^{-2}$, the following plot was generated

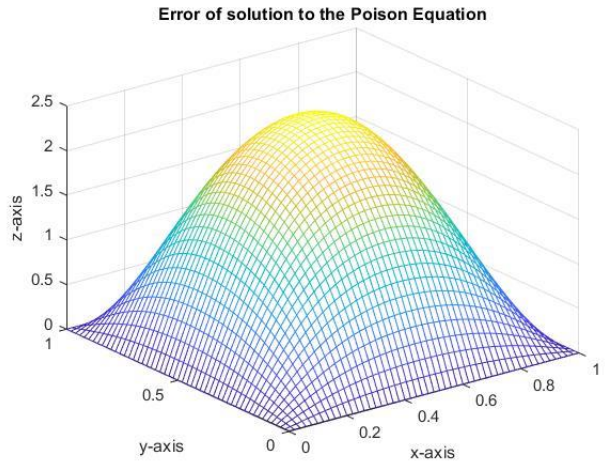
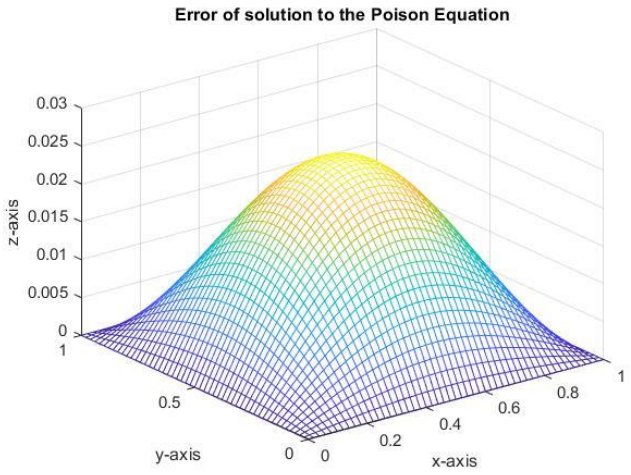
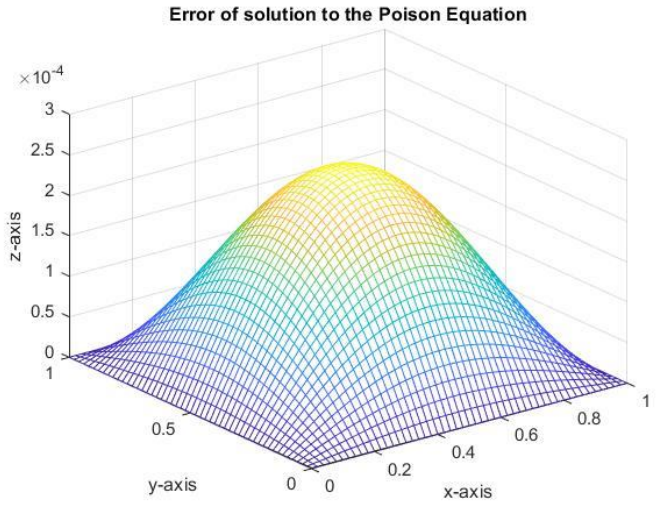


Figure 10

With $h = 0.02$ and $residue = 10^{-4}$, the following plot was generated



With $h = 0.02$ and $residue = 10^{-6}$, the following plot was generated



As we can see, it seems that the error at each point is proportional to the $residue$.

Bonus

So far, we have been assuming that the best way to get our approximations to converge to the actual solution is by setting the $residue$ (i.e. the difference between our estimation and the true solution) to be equation to zero.

For example, in Exercise 4, we set the $residual$ to be

$$r_i^{\,j} = \frac{U_{i+1}^{i} + U_{i-1}^{j} + U_i^{(j+1)} + U_i^{j-1} - 4U_i^{j}}{4}$$

We then set the new value to be

$$\left(U_i^j\right)_{new} = \left(U_i^j\right)_{old} + r_i^{\,j}$$

Thus, we are essentially choosing $\left(U_i^j\right)_{new}$ such that the $residual$ is zero. Choosing $\left(U_i^j\right)_{new}$ such that the new $residual$ is zero is not necessarily the most efficient way to converge the value of $\left(U_i^j\right)$ to the actual solution. We can modify equation (XX) to

$$\left(U_i^j\right)_{new} = \left(U_i^j\right)_{old} + \omega r_i^{\,j}$$

Where ω is positive. If $0 < \omega < 1$ then the procedure is called under relaxation method and if $1 < \omega$ then the procedure is called over relaxation method (SOR). The over relaxation methods are used to accelerate the convergence of the Guass-Siedel technique.

It can be proven that for SOR methods, can converge only if $0 < \omega < 2$[1]

Exercise 4 with SOR

The code for exercise 4 and 5 can be adjusted to implement SOR by simply changing a few lines. The new code for exercise 4 and 5 with SOR is shown below

```
close all;
clear;
clc;

%Choose gridsize and residual size
h = 0.02;
max_res = 1e-4;
omega = 1.0;

%Choose the border functions
fi1 = @(x) 0;
fi2 = @(y) 0;
fi3 = @(x) heaviside(x-0.2)-heaviside(x-0.8);
fi4 = @(y) 0;

%Choose forcing term and the actual function to compare
g = @(x,y) 0;

%Setting up
x(1) = 0;
xf = 1;
yf = 1;
n = round((xf - x(1))/h);
x = 0:h:n*h;
y = 0:h:n*h;
z = zeros(length(x),length(y));

%Calculating the actual values
[X, Y] = meshgrid(0:h:n*h);
% The U function represents the solution to the Poison Equation
%U = -2*X.^2 - Y.^2 + X + Y + 2*X.*Y + 2;

sum = 0;

for i = 1:length(x)
    z(i,length(y)) = fi1(x(i));
    z(length(x),i) = fi2(y(i));
    z(i,1) = fi3(x(i));
```

[1] Add reference

```
    z(1,i) = fi4(y(i));
end

%Calculate the average value
total = 4*(n + 1) - 4;
avg = sum/total;

%Set all the inner points as this average value
for i = 2:length(x) - 1
    for j = 2:length(y) - 1
            z(i,j) = avg;
    end
end

iteration = 0;
while (true)
    %set the maximum residue found in this iteration to 0
    %it will be updated in the for loop
    r_max = 0;
    for i = 2:length(x) - 1
        for j = 2:length(y) -1
            %Calculate the residue
            r = (z(i+1,j) + z(i-1,j) + z(i,j+1) + z(i,j-1) -
h^2*g(x(i),y(j)) - 4*z(i,j))/4;
            %Set the new value of the point
            z(i,j) = z(i,j) + omega*r;
            if(abs(r) > r_max)
                r_max = abs(r);
            end
        end
    end
    %if the meximum residue value found in this iteration is smaller
    %than our maximum tolerence break
    if(r_max < max_res)
        break;
    end
    iteration = iteration + 1;
end

%It is important not to forget to transpose because of the way matlab
%handles matrices
%Uncomment the following line to get the estimation for the solution
mesh(x,y,z');

%Plot error
%er = abs(z' - U); mesh(X,Y,er);
xlabel('x-axis');
ylabel('y-axis');
zlabel('z-axis');
title('Solution to the Laplace Equation with SOR');
```

The only thing that has changed is that we now added the omega variable and an iteration variable so that we can compare different values of omega so see which one converges faster.

For the comparisons, we kept the h and $residue$ values constant with $h = 0.02$ and $residue = 10^{-4}$

Finally, to make comparing the different values a bit easier, we ran the script shown below

```
myIterations = zeros(1,20);
myOmegas = zeros(1,20);

for myI = 0:19
    %Set the new value of Omega
    omega = 1+ 0.05*myI;
    %Run the sor script
    sor
    %Add iteration to the myIterations array and the Omegas to the
myOmegas
    %array
    myIterations(myI+1) = iteration;
    myOmegas(myI+1) = omega;
end

%Plot
plot(myOmegas,myIterations);
xlabel('Omega value');
ylabel('Number of iteration');
title('Omega vs Number of iterations');
```

Effectively, all it does is run the sor script for different values of ω it then records the numbers of iterations that were needed for the $residue$ to be satisfied and plots the results.

For exercise 4, we managed to plot the following

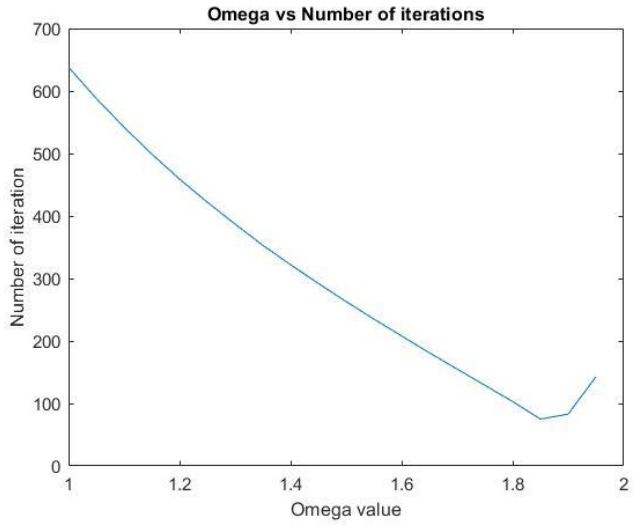


Figure 11

From the figure above, we can see that there is a certain range of ω values for which the number of iterations was significantly decreased (in this case it's around $\omega = 1.85$)

Exercise 5 with SOR

If we do the same process for exercise 5, we the following plot

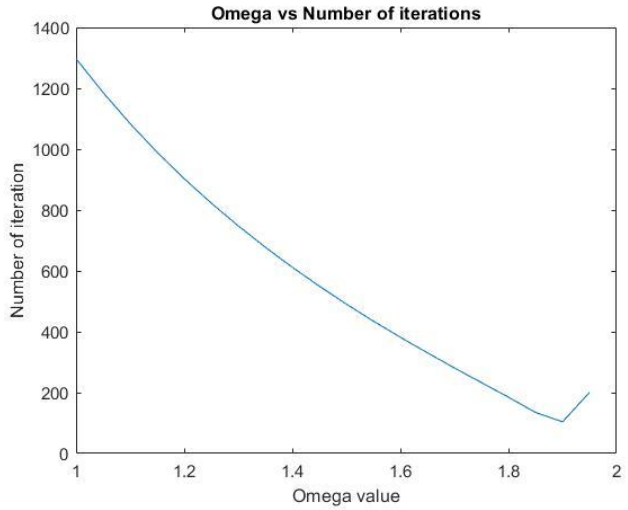


Figure 12

The two plots have a similar shape! This is because, in both cases, we are trying to derive an approximation to the linear system defined by $A\boldsymbol{x} = \boldsymbol{b}$ and by changing the boundary functions and the forcing term, we are only changing $\boldsymbol{b}$ while A remains the same. It seems reasonable that changes in the values of $\boldsymbol{b}$ shouldn't really change the shape of the graph.

We can try to confirm this by changing the value of h which will effectively change A

With $h = 0.1$ and $residue = 10^{-6}$, the following plot was generated

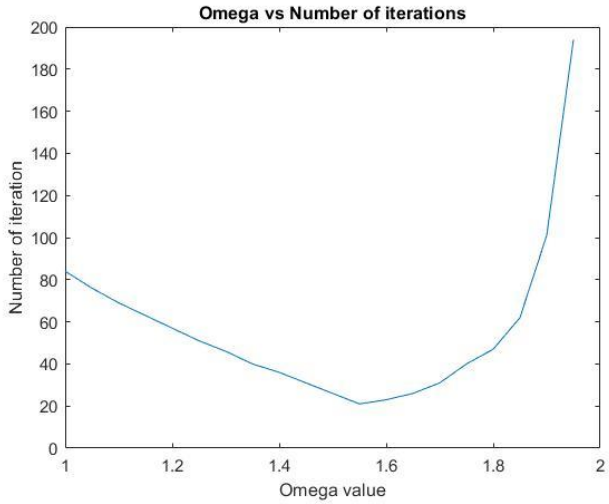


Figure 13

As we can see the shape of the graph has changed and now the minimum amount of iterations seems to have occurred at about $\omega = 1.55$

It is also interesting to observe how the choice of ω affects the error of the approximation. This is illustrated in the following graphs in which the h and $residue$ values were kept constant at $h = 0.02$ and $residue = 10^{-4}$.

With $\omega = 1$ following plot was generated

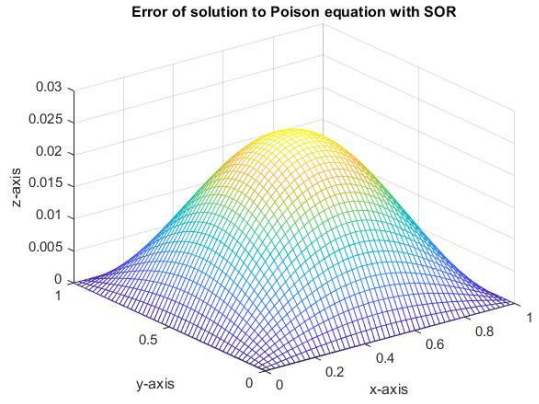


Figure 14

With $\omega = 1.2$ following plot was generated

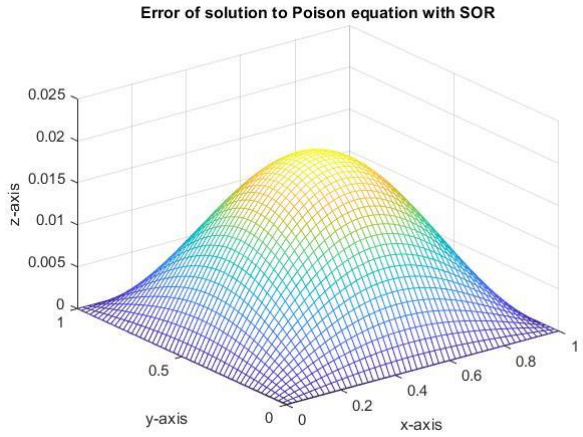


Figure 15

With $\omega = 1.4$ following plot was generated

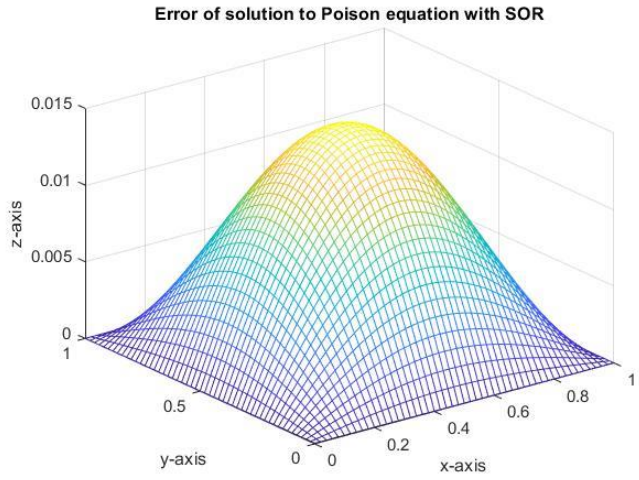


Figure 16

With $\omega = 1.6$ following plot was generated

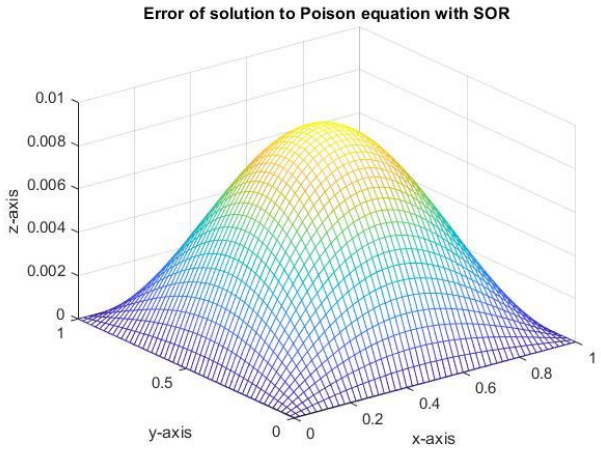


Figure 17

With $\omega = 1.87$ following plot was generated

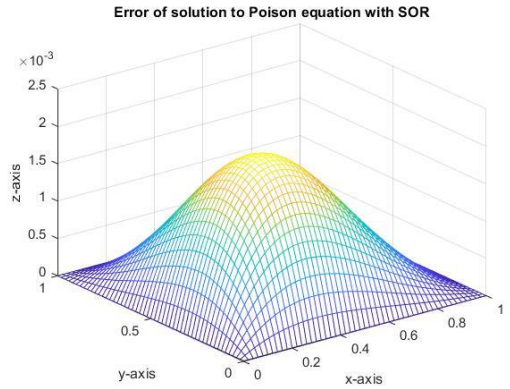


Figure 18

While with $\omega = 1.9$ which is only slightly above what we estimated to be our ω value for the minimum amount of iterations, this plot was generated

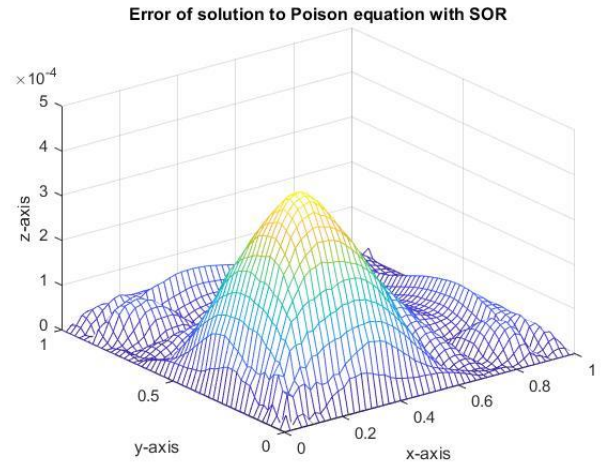


Figure 19

As we can see some sort of *instability* issue has occurred once exceed the best ω. This might be caused because of an overestimation of the point. // Ask about this

Finally, for with $\omega = 1.99$ (something near the max of 2) the following plot was generated

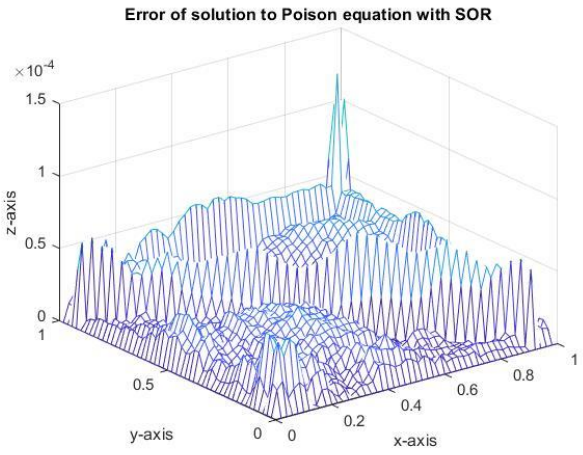


Figure 20

As we can see higher values of ω (up to a point) not only help find the solution faster but also make the solution more accurate given a constant $residue$.