MA228 NUMERICAL ANALYSIS - EXERCISE SHEET 3

Joe Tilley 1403899

February 11, 2016

Question 1

If we are given a function $f(t, x)$ such that $\frac{d}{dt}x(t) = f(t, x)$, a time range $[a, b]$ (assuming $a < b$), an initial value x_0 such that $x(a) = x_0$ and a step width N, then we can code Euler's Method as follows:

```
function eulermethod(a, b, func, x0, N)
    %a is initial time
    %b is end time (assuming a<b)
    %func is the function f(t,x) as a string in terms of x and t
    %x0 is initial value
    %N is number of steps

    clear t                              %Clears any saved values
    clear x
    format long
    func = strcat('@(t,x)',func);        %converts func into form of anonymous ...
        function
    f = str2func(func);                  %converts the string to an anonymous ...
        function
    t(1)=a;                              %saves initial time value
    x(1)=x0;                             %saves initial x value
    h=(b-a)/N;                           %defines time step width

    for n=1:N
        t(n+1)=t(n)+h;                   %adds new time entry
        x(n+1)=x(n)+h*f(t(n),x(n));      %uses Euler Method to add new x entry
    end

    plot(t,x)                            %plots function
end
```

Here, we have defined the step size h in terms of the step width N using the relationship $h = \frac{b-a}{N}$, so we have a constant step width. Now, let us choose the function $f : [0, 1.2] \times \mathbb{R} \to \mathbb{R}$ (so $a = 0$ and $b = 1.2$) given by $f(t, x) = \frac{\cos(t^2)}{x^2}$, and choose initial state $x(0) = 1 = x_0$, and a number of steps $N = 20$ (and so $h_0 = \frac{b-a}{N} = \frac{1.2-0}{20} = 0.06$). If we also wish to repeat the computation but with step size halved, then we can simply add the following code to the end of the current algorithm.

```
%Additional Code for halved step size
```

```
h=h/2;                                  %halves step size
N=(b-a)/h;                              %recalculates N
newt(1)=a;                              %saves initial time value
newx(1)=x0;                             %saves initial x value
for n=1:N
    newt(n+1)=newt(n)+h;                %adds new time entry
    newx(n+1)=newx(n)+h*f(newt(n),newx(n));  %uses Euler Method to add new x entry
end
hold on
plot(newt,newx)
```

Note that N is recalculated in such a way that the end time b is certainly reached. Then, to create a table comparing the distance between the values of the states $x(t)$ in absolute value at times that coincide, we should further add the following code:

```
%Making the table
skipnewx = [];                          %empty list
for i=1:N+1                             %iterates through length of newx
    if mod(i,2)==1                      %skips out every other value
        skipnewx = [skipnewx, newx(i)]; %stores values where coincide
    end
end
display(table(t',x',skipnewx',abs(x'-skipnewx'),'VariableNames',{'Time' 'OldX' ...
    'NewX' 'Difference'}))              %creates table
%Time = t
%OldX = x(t) using step size h_0
%NewX = x(t) using step size h_0/2
%Difference = |OldX(t)-NewX(t)|
hold off
```

The titles of each column are explained in the code notes (MATLAB annoyingly only allows one-word titles.) Since all of this additional code is included in the function ‘eulermethod.m’, we can plot the graph of the two computations and the difference table by simply running:

```
>>eulermethod(0, 1.2, 'cos(t^2)/(x^2)', 1, 20)
```

This produces the following graph (Figure 1).

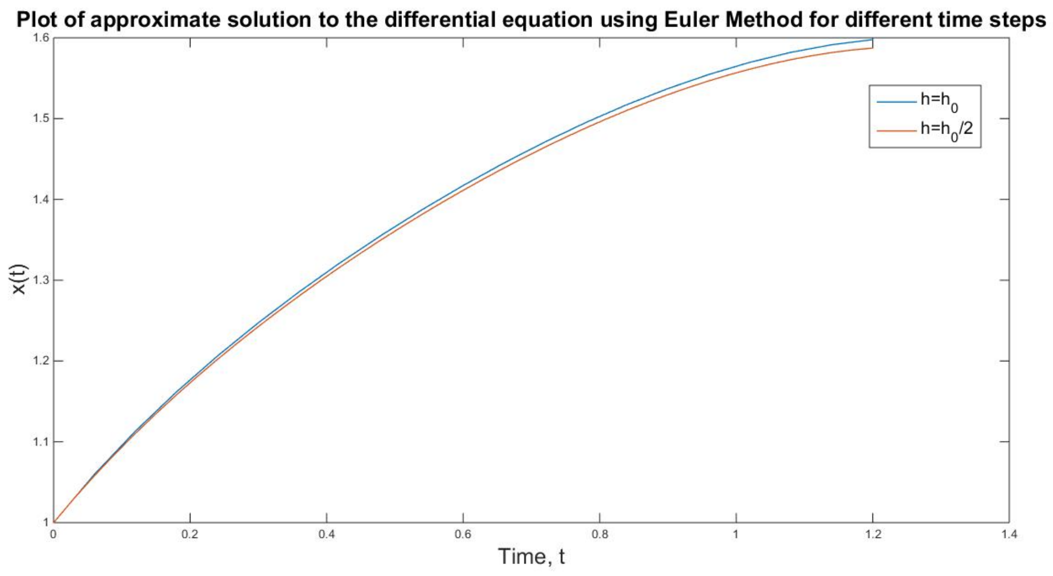

Figure 1: Graph displaying the approximate solutions using the a step size h_0 and $\frac{h_0}{2}$ in Euler method

Furthermore, the following table is outputted:

```
Time          OldX                NewX              Difference
    ----     ----------------    ----------------    --------------------

       0                    1                   1                       0
    0.06                 1.06    1.05827786582147     0.00172213417852696
    0.12     1.11339944037061    1.11054434416673     0.00285509620388402
    0.18     1.16179485597731    1.15815406688262     0.00364078909468235
    0.24     1.20622363308266    1.20201439365704     0.00420923942562101
     0.3     1.24739305253368    1.24275575345438     0.00463729907929955
    0.36     1.28579766051123    1.28082363272772     0.00497402778351375
    0.42     1.32178489653352    1.31653199882233     0.00525289771119075
    0.48     1.35559429018035    1.35009614022613     0.00549814995421904
    0.54     1.38738214772914     1.3816538307211     0.00572831700803733
     0.6     1.41723783924923     1.4112795667099     0.00595827253932635
    0.66      1.4451950438879    1.43899457318725     0.00620047070064578
    0.72     1.47123991408313    1.46477419695963     0.00646571712350164
    0.78     1.49531737111177    1.48855371409298     0.00676365701879034
    0.84     1.51733632682684    1.51023324234989     0.00710308447695085
     0.9     1.53717438730235    1.52968225492407     0.00749213237827595
    0.96     1.55468245651342    1.54674408035524     0.00793837615818815
    1.02     1.56968958178126    1.56124071379762     0.00844886798363942
    1.08     1.58200834577904    1.57297823992682     0.00903010585221842
    1.14     1.59144110064326    1.58175316876227     0.00968793188099615
     1.2     1.59778735185322    1.58736000816041      0.0104273436928071
```

Note that the difference between the two approximations increases as time increases. This is due to the convex yet strictly increasing nature of the solution on the given range, and that larger

h (step size) gives a worse approximation than the approximation given by the step size halved. Hence, the approximation given by h_0 will continue to deviate from the true solution and the better approximation (given by $\frac{h_0}{2}$), so their difference increases.

Question 2

The following is pseudocode for the Runge Kutta Method (RK4). Note I have used % to refer to a comment.

```
input $a, b, f, x_0, N$                          %time range ($a < b$), function, initial value, # of steps
$t_0 := a$                                       %sets initial time
$N := \frac{b-a}{h}$                             %calculates number of steps
for $n = 0, 1, 2, ..., N-1$                      %iterates
|$k_1 := f(t_n, x_n)$                            %calculates all required k's
|$k_2 := f(t_n + \frac{h}{2}, x_n + \frac{h}{2}k_1)$
|$k_3 := f(t_n + \frac{h}{2}, x_n + \frac{h}{2}k_2)$
|$k_4 := f(t_n + h, x_n + hk_3)$
|$t_{n+1} := t_n + h$                            %adds step to time
|$x_{n+1} := x_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)$   %determines new value
end for loop
ouput $x_0, x_1, x_2, ..., x_N$                  % outputs the x-values (i.e. the approximate solution)
```

Implementing this code into MATLAB gives the following code:

```matlab
function rungekutta(a, b, func, x0, N)
    %a is initial time
    %b is end time
    %func is the function f(t,x) as a string in terms of x and t
    %x0 is initial value
    %N is the number of steps

    clear t                             %Clears any saved values
    clear x
    format long
    func = strcat('@(t,x)',func);       %converts func into form of anonymous ...
        function
    f = str2func(func);                 %converts the string to an anonymous ...
        function
    t(1)=a;                             %saves initial time value
    x(1)=x0;                            %saves initial x value
    h=(b-a)/N;                          %calculates time step width

    for n=1:N
        k_1=f(t(n),x(n));                   %calculates k values
        k_2=f(t(n)+h/2,x(n)+(h/2)*k_1);
        k_3=f(t(n)+h/2,x(n)+(h/2)*k_2);
        k_4=f(t(n)+h,x(n)+h*k_3);
        t(n+1)=t(n)+h;                      %adds new time entry
        x(n+1)=x(n)+ (h/6)*(k_1+2*k_2+2*k_3+k_4);  %adds new x entry
    end
    plot(t,x)                               %plots function
end
```

Now, if we wish to solve the exact same ODE as before with the exact same step width, and then wish to repeat the process but with a step width halved, then we can implement the **exact** same code as the previous additional code of Euler method (from line 24-47) due to the similarities in style and variable names in both codes. Now we can plot the graph of the two computations (the estimate solutions using step size $h_0(=\frac{1.2-0}{20}=0.06)$ and halved) and the difference table by simply running:

```
>>rungekutta(0, 1.2, 'cos(t^2)/(x^2)', 1, 20)
```

This produces the following graph (Figure 2).

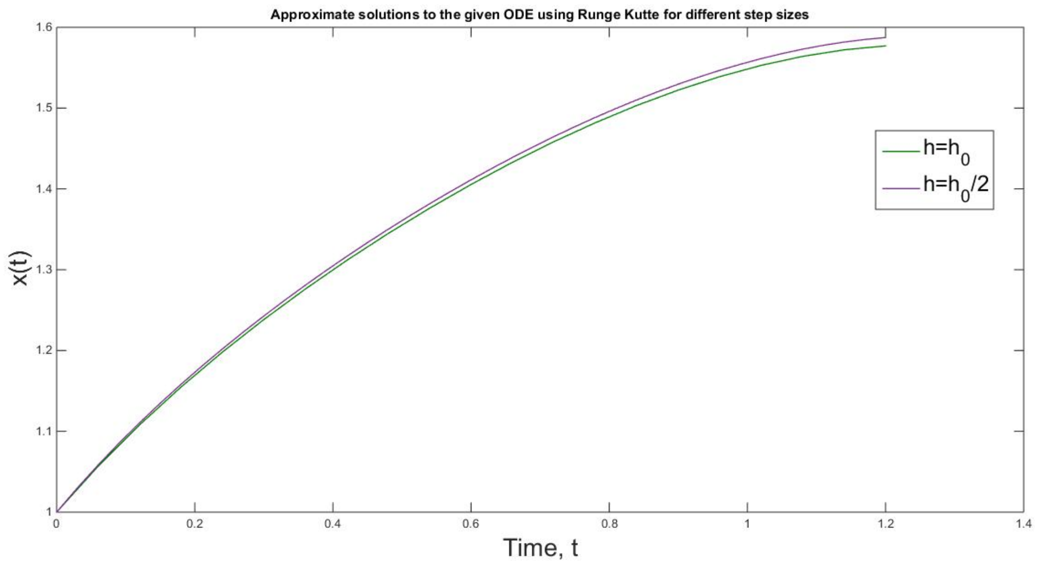

Figure 2: A graph displaying the approximate solutions using the a step size h_0 and $\frac{h_0}{2}$ in Runge Kutta

Furthermore, the following table is outputted:

```
Time            OldX                 NewX              Difference
     ----     ----------------    ----------------    -------------------

        0                    1                   1                        0
```

```
      0.06    1.05672183224986    1.05827786582147    0.00155603357161782
      0.12    1.10792975736782    1.11054434416673     0.0026145867989118
      0.18    1.15478632719103    1.15815406688262    0.00336773969159587
      0.24    1.19808996964225    1.20201439365704    0.00392442401478177
       0.3    1.23840410909113    1.24275575345438    0.00435164436324098
      0.36    1.27613010008734    1.28082363272772    0.00469353264037609
      0.42     1.3115510281933    1.31653199882233    0.00498097062903535
      0.48     1.3448593461911    1.35009614022613    0.00523679403502464
      0.54    1.37617506657223     1.3816538307211    0.00547876414887272
       0.6    1.40555822624463     1.4112795667099    0.00572134046527117
      0.66    1.43301779758267    1.43899457318725    0.00597677560457988
      0.72    1.45851838701375    1.46477419695963    0.00625580994587516
      0.78    1.48198559411303    1.48855371409298    0.00656811997995144
      0.84    1.50331063387402    1.51023324234989    0.00692260847587067
       0.9    1.52235466757351    1.52968225492407    0.00732758735055872
      0.96    1.53895319866401    1.54674408035524    0.00779088169122355
      1.02     1.5529208452885    1.56124071379762    0.00831986850911548
      1.08    1.56405678744884    1.57297823992682    0.00892145247798481
      1.14    1.57215119816412    1.58175316876227    0.00960197059814649
       1.2    1.57699300162073    1.58736000816041     0.0103670065396784
```

The exact same conclusions can be drawn by these results as we drew from the Euler method (the halved step size is a better approximation, the difference increases due to convexity and strictly increasing, etc.)

Question 3

We know that the solution to the ODE

$$\frac{d}{dt}x(t) = x(t) \qquad x(0) = 1$$

is given by $x(t) = e^t$. So, let $f = x$ in our code. Furthermore, let us remove the additional code (i.e. any of the code used to generate the approximate solutions with the step size halved). Then, we should insert code the generate the actual error and to plot it, given by:

```
23 error=abs(exp(t)-x);          %finds the actual error
24 plot(t, error)                %plots the error
```

Note that the line number will be 26 in the Runge Kutta m-file instead. We will use the same fixed step size $h_0 = \frac{b-a}{N}$ as before, which is determined by the number of steps N, which we shall choose to be the same for both algorithms in order to compare the error. We shall choose $a = 0$ and $b = 1$, and choose $N = 100$. Running the following code

```
>> eulermethod(0, 1, 'exp(t)', 1, 100)
>> rungekutta(0, 1, 'exp(t)', 1, 100)
```

produces the following two graphs which show the distances of the true solution with solutions of the Euler method and the Runge-Kutta method at given time points using a fixed step width (i.e. the exact error) (Figure 3 and Figure 4).

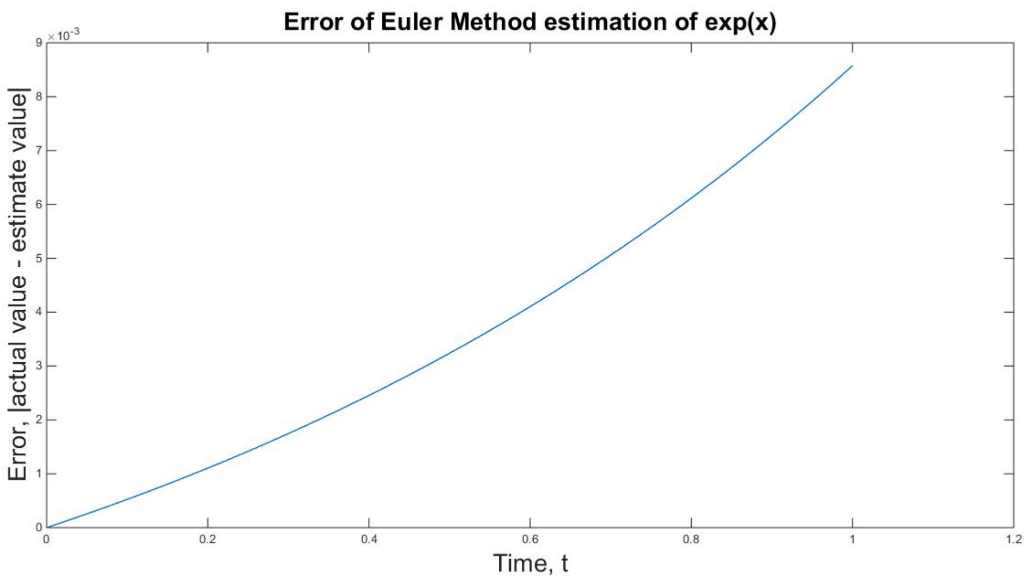

Figure 3: A graph displaying the exact error for the Euler Method ($N = 100$)

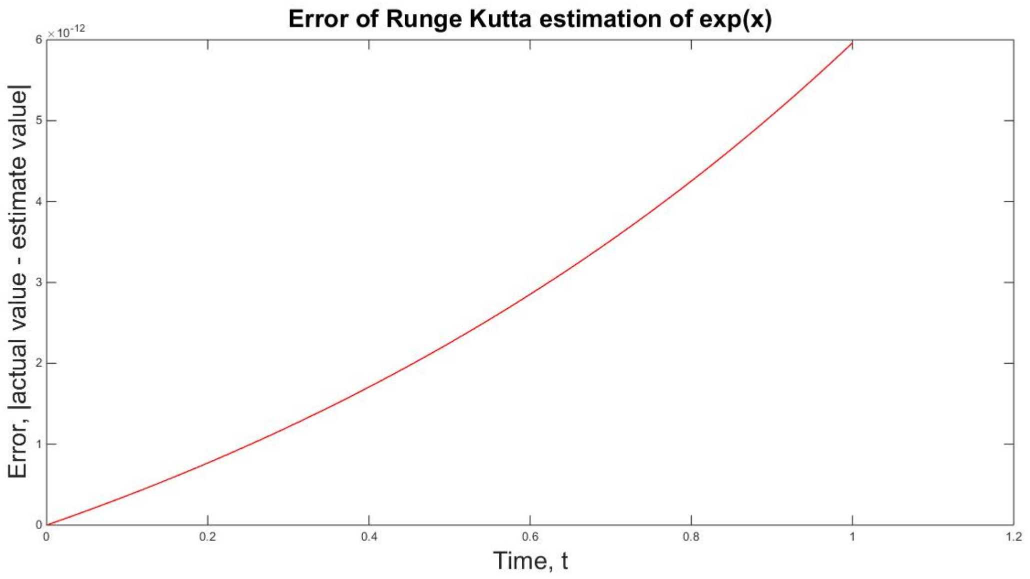

Figure 4: A graph displaying the exact error for Runge Kutta ($N = 100$)

We can note that the shape of both the errors is very similar. They both follow an exponential growth in error. This is due to the fact that they are approximating an exponential function, so once the approximation grows apart from the actual function, then it only continues to grow further apart (at an exponential rate). This pattern will occur for any strictly increasing, convex function (though the error may not be necessarily an exponentially increasing function, but certainly increasing).

We can also note that the error for Runge Kutta is *very* small compared to the error for Euler method. We see that Runge Kutta error is of the order 10^{-12}, whereas the Euler method has an error of 10^{-3}. This can be fully explained by considering 'the Runge Kutta methods', a class of ODE solution approximating methods. The Euler method is the most simple of these, the first order Runge Kutta method. Whereas the Runge Kutta (the method we have used, RK4) is a more sophisticated and better approximation to solutions, as it is a forth order method. Hence it has a smaller deviation (error) from the true solution.

Further note that in this example, $h = 0.01$. The theoretical global truncation error for Euler is of the order $h = 10^{-2}$, whereas Runge Kutta is of the order $h^4 = 10^{-12}$. This is somewhat seen in our plots, as the Euler method error is a little below 10^{-2} on average, and the Runge Kutta error is roughly 10^{-12}, and so the orders of the error are roughly seen in these plots, however this is a mere observation of very rough order and far from being explicitly shown or proof.

Question 4

Firstly, we must define grid points on the square $\Omega = (0,1) \times (0,1)$. We will let $N > 0$ be a natural number defining the number of lattice points in each direction (e.g. $N = 2$ gives a 2×2 grid) which is decided by the user. Now set $h = \frac{1}{N-1}$ (a fixed step width), and define grid points (x_i, y_j) where $x_i = (i-1)h$ and $y_j = (j-1)h$ for $i, j = 1, ..., N$ (so equal distance points are defined and the end values are achieved). We shall denote $u(x_i, y_j)$ by $u_{i,j}$. Now, if we wish to solve

$$\begin{aligned} -\triangle\, u(x,y) &= f(x,y) \qquad (x,y) \in \Omega \\ u(x,y) &= g(x,y) \qquad (x,y) \in \partial\Omega \end{aligned}$$

then we can use a finite difference approximation of the second partial derivatives at each point to estimate u at each points, based on the values surrounding it (and f at that point), as given by the expression

$$u_{i-1,j} + -u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - 4u_{i,j} = h^2 f(x_i, y_j)$$

This will form a system of equations for the non-boundary points of the lattice (the boundary points are known, so we need not estimate them), which can be encoded into matrix form as

$$A\mathbf{u} = \mathbf{b}.$$

where A is an $(N-2)(N-2) \times (N-2)(N-2)$ matrix, given by

$$A = \begin{bmatrix} D & -I & 0 & 0 & 0 & \dots & 0 \\ -I & D & -I & 0 & 0 & \dots & 0 \\ 0 & -I & D & -I & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & \dots & 0 & -I & D & -I & 0 \\ 0 & \dots & \dots & 0 & -I & D & -I \\ 0 & \dots & \dots & \dots & 0 & -I & D \end{bmatrix} \quad \text{where} \quad D = \begin{bmatrix} 4 & -1 & 0 & 0 & 0 & \dots & 0 \\ -1 & 4 & -1 & 0 & 0 & \dots & 0 \\ 0 & -1 & 4 & -1 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & \dots & 0 & -1 & 4 & -1 & 0 \\ 0 & \dots & \dots & 0 & -1 & 4 & -1 \\ 0 & \dots & \dots & \dots & 0 & -1 & 4 \end{bmatrix}$$

and I is the identity matrix. D and I have dimensions $(N-2) \times (N-2)$.

Furthermore, $\mathbf{u}$ and $\mathbf{b}$ are column vectors of length $(N-2)(N-2)$, where $\mathbf{u}$ is the solution to be found/estimated (form as lexicographical order given below), and $\mathbf{b}$ is given similarly as

$$\mathbf{u} = \begin{bmatrix} u_{2,2} \\ u_{3,2} \\ \vdots \\ u_{N-1,2} \\ u_{2,3} \\ u_{3,3} \\ \vdots \\ u_{N-2,N-2} \end{bmatrix} \quad \text{and} \quad \mathbf{b} = \begin{bmatrix} h^2 f(x_2, y_2) + u_{1,2} + u_{2,1} \\ h^2 f(x_3, y_2) + u_{3,1} \\ \vdots \\ h^2 f(x_{N-1}, y_2) + u_{N,2} + u_{N-1,1} \\ h^2 f(x_2, y_3) + u_{1,3} \\ h^2 f(x_3, y_3) \\ \vdots \\ h^2 f(x_{N-1}, y_{N-1}) + u_{N,N-1} + u_{N-1,N} \end{bmatrix}$$

Note that $\mathbf{b}$ has some additional terms added in some of the entries, varying each time. In general, for a term starting $h^2 f(x_i, y_j)$, then the $u(\cdot,\cdot)$ of all points adjacent to (x_i, y_j) in the lattice which *are* boundary points are added to that entry. Now we can perform the Gauss-Seidel iteration on this system given by the equation

$$u_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j<i} a_{ij} u_j^{(k+1)} - \sum_{j>i} a_{ij} u_j^{(k)} \right), \quad i, j = 1, 2, \ldots, N-2$$

where u_i is the i^{th} entry of the vector $\mathbf{u}$, a_{ij} is the i,j-entry of the matrix A, b_i is the i^{th} entry of the vector $\mathbf{b}$, and k is the number of the iteration. We also require an initial estimate, $\mathbf{u}^{(0)}$. I have coded this as a function as follows:

```
function [u] = gauss_seidel(A, b, u0, iters)
    n = length(A);
    u = u0;
    for k = 1:iters
        for i = 1:n
            u(i) = (1/A(i, i))*(b(i) - A(i, 1:n)*u + A(i, i)*u(i));
        end
    end
end
```

This function will be called upon in our code to solve the problem. Let us choose $f(x,y) = x^y - y^x$ on Ω, and let $g(x,y) = \sin(\pi x) + \sin(\pi y)$ on $\partial\Omega$. We shall also choose to have 5000 iterations, let $N = 101$ (so $h = 0.01$), and let $\mathbf{u}^{(0)} = (1, 1, ..., 1)^T$ be our initial estimate solution. The following code performs a finite difference discretisation with fixed step width, and solves the problem using a Gauss-Seidel iteration numerically.

```
%initial setup
clear
iterations = 50;                %Number of iterations
N = 31;                         %Number of lattice points
h = 1/(N-1);                    %step size
[X,Y] = meshgrid(0:h:1, 0:h:1); %creates omega
f = X.^Y - Y.^X;                %nabla(u) on omega
```

```
g = sin(pi*X)+sin(pi*Y);         %u on the boundary
u0 = eye(1,(N-2)*(N-2))';        %initial guess u0 (all 1's)

%Makes matrix D
D = 4.*eye(N-2);                 %Makes 4's along diagonal
for i=1:N-3                      %Makes -1's above and below diagonal
    D(i,i+1)=-1;
    D(i+1,i)=-1;
end

%Makes matrix A
A = kron(eye(N-2),D);        %Puts D's along diagonal of A
for i=1:((N-2)*(N-3))        %Puts in -(identity matrix) above and below D's
    A(i,i+(N-2))=-1;
    A(i+(N-2),i)=-1;
end
A = sparse(A);               %used to help store data efficiently

%Makes column vector b
b = zeros(1,(N-2)*(N-2));  %initially sets b as a zero array
index=1;                   %used to iterate through the index of b
for j=2:N-1
    for i=2:N-1
        %adds boundary conditions in b (if any) by considering points
        %adjacent to boundary points
        if j == 2
            b(index)= b(index) + g(i,j-1);
        end
        if i== 2
            b(index)= b(index) + g(i-1,j);
        end
        if j == N-1
            b(index)= b(index) + g(i,j+1);
        end
        if i== N-1
            b(index)= b(index) + g(i+1,j);
        end
        b(index) = b(index) + h^2*f(i,j);%adds non-boundary conditions in b
        index=index+1;
    end
end
b=b';                                         %makes b column vector

solution = gauss_seidel(A,b,u0,iterations); %uses gauss seidel method to give ...
    approximate solution as a column vector
solution_lattice = zeros(N);        %Initially sets zero matrix as solution

%Converts lexicographical order to matrix
index = 1;  %used to iterate through the index of solution array
for j=2:N-1
    for i=2:N-1
        solution_lattice(i,j) = solution(index);  %stores array value in ...
            appropriate matrix position
        index=index+1;
    end
end
```

```

%adds boundary conditions back in to lattice
for j=1:N
    solution_lattice(1,j) = g(1,j);
    solution_lattice(N,j) = g(N,j);
end

for i=1:N
    solution_lattice(i,1) = g(i,1);
    solution_lattice(i,N) = g(i,N);
end

%surface and contour plot of approximate solution
surfc(X,Y,solution_lattice)
```

The output of this is the approximate solution $u(x, y)$ for $(x, y) \in \Omega$, which I have plotted as a surface with contour plots (to help understand the shape better), as seen below (Figure 5).

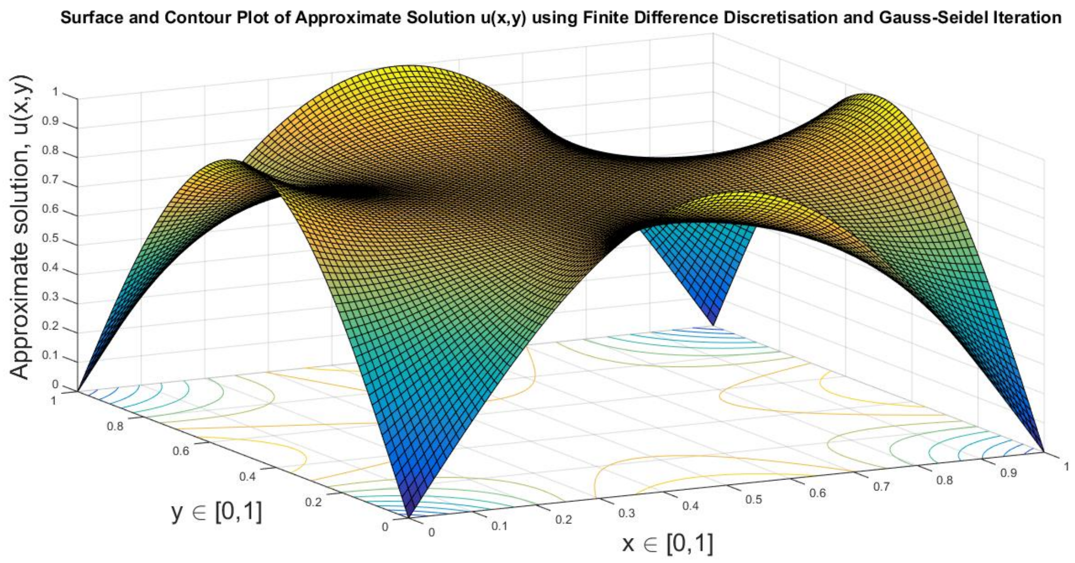

Figure 5: A graph displaying the approximate solution generated by the code