MS6021 Scientific Computation

Assignment1

Dara Corr - 22275193

September 22, 2022

1 Part I

The following is an outline of the tasks we are asked to do in part I:

- We consider the following boundary value problem

$$-u'' + 4u = exp(x), u(0) = u(1) = 0$$

- Solve the BVP analytically
- Introduce a numerical method to solve the BVP numerically in Matlab using a sparse matrix
- Use the spy function to visualise the sparse matrix that was created in the numerical method
- Solve the equation using a full matrix
- plot the analytical solution and numerical solutions for different values of N on the same plot and find the maximum nodal error for each of the numerical solutions

1.1 Analytical Solution

First step is to solve the BVP analytically. To do this we seek to find a solution u(x) which solves the differential equation $-u'' + 4u = exp(x)$ with boundary values $u(0) = u(1) = 0$. This differential equation is given in the form $-u''(x) + a(x) \cdot u(x) = f(x)$. To get the general solution to the BVP, we need to find the homogenous solution $u_h(x)$ and the particular solution $u_p(x)$. Our general solution will then be $u(x) = u_h(x) + u_p(x)$.

To find $u_h(x)$ we set $f(x) = 0$. We attempt to find a homogeneous solution of the form αe^{kx}. This solves our equation for $\alpha = 1$ and $k = \pm 2$. Our homogeneous solution is then:

$$u_h(x) = A \cdot e^{2x} + B \cdot e^{-2x}$$

By trying expontential functions as particular solutions to the particular form of the ODE, we find that:

$$u_p(x) = \frac{1}{3}e^x$$

When Boundary Conditions $u(0) = u(1) = 0$ are applied we are left with 2 simultaneous equations to solve for A and B

$$A + B = -\frac{1}{3}$$

$$A \cdot e^2 + B \cdot e^{-2} + \frac{1}{3}e^1 = 0$$

Solving these equations for A and B gives the full analytical solution to u(x):

$$A = -\frac{1}{3}\left[\frac{1 - exp(-1)}{e^{-4} - 1} + 1\right]$$

$$B = \frac{1}{3}\left[\frac{1 - exp(-1)}{e^{-4} - 1}\right]$$

$$u(x) = -\frac{1}{3}\left[\frac{1 - exp(-1)}{e^{-4} - 1} + 1\right] \cdot e^{2x} + \frac{1}{3}\left[\frac{1 - exp(-1)}{e^{-4} - 1}\right] \cdot e^{-2x} + \frac{1}{3}exp(x)$$

1.2 Creating Numerical Solutions

The next objective is to introduce a numerical method which discretizes the problem so that numerical solutions can be evaluated. To achieve this we divide the interval (0,1) into N equal subintervals and we introduce the centred finite distance method to discretize the derivative $-u''(x)$ in our numerical method.

The ODE is discretized using the following equation for the centred finite difference method:

$$u''(x) = \frac{u(x_i + h) - 2u(x_i) + u(x_i - h)}{h^2}$$

where x_i is indexed from x_1 to x_{N+1} and $h = \frac{1}{N}$

We can now write a system of equations for the system (keeping in mind that $u_1 = u_{N+1} = 0$):

$$-\frac{1}{h^2}\left[u_{i+1} - 2u_i + u_{i-1}\right] + 4u_i = f(x_i)$$

and we can write in terms of N:

$$-N^2\left[u_{i+1} - 2u_i + u_{i-1}\right] + 4u_i = f(x_i)$$

In order to get Matlab to solve our system so we can get values for U, we want to write our system as a matrix equation of the form $\mathbf{AU} = \mathbf{F}$.

Since we already know that $u_1 = 0$ and $u_{N+1} = 0$, we only need to have U ranging from $U_2, ..., U_N$ in our numerical method. So U is a column vector of length $N - 1$ from $U_2, ..., U_N$ and similarly A will be a $N - 1 \times N - 1$ matrix and F will be a column vector of length $N - 1$ with entries $f(x_2), ..., f(x_N)$.

Our matrix system for this problem should look like this (this example is for the N = 5 case to visualise the size of the matrices):

$$\begin{bmatrix} 2N^2+4 & -N^2 & 0 & 0 \\ -N^2 & 2N^2+4 & -N^2 & 0 \\ 0 & -N^2 & 2N^2+4 & -N^2 \\ 0 & 0 & -N^2 & 2N^2+4 \end{bmatrix} \cdot \begin{bmatrix} u_2 \\ \vdots \\ \vdots \\ u_N \end{bmatrix} = \begin{bmatrix} exp(x_2) \\ \vdots \\ \vdots \\ exp(x_N) \end{bmatrix}$$

We can use Spdiags() in Matlab to input the matrix A. we create an array x to hold the x values and another array f to hold the $F(x_i) = exp(x_i)$ values. Then U = $A\backslash F$ to find u_2, ... ,u_N. Then we include boundary conditions $U_1 = U_{N+1} = 0$ and we will have our numerical solution to the problem in Matlab.

The code that was used to implement this approximation into Matlab can be found below. The output is the Solution vector named U1. When this section of code is used in future it is called as a function named NUMsolve1(N) which solves this ODE numerically for a given value of N using this code.

```
%assignment 1: using finite difference method to numerically solve the ODE:
%-u(x)'' + 4u(x) = exp(x), u(0) = u(1) = 0

%define N number of subintervals between 0 and 1
N=100

x = linspace(0,1,N+1) %create vector of N+1 values of x between 0 and 1

%visualise ODE as matrix equation AU = F, where we are trying to solve for U

%create sparse Matrix A, which implements the 2nd order central finite difference
%method to our problem as a matrix equation. A is multiplied by U to give us the
%right hand side of our ODE

b = -(N^2)*ones(N-1,1); %the diagonals directly beneath and above A
c = (2*N^2 + 4)*ones(N-1,1); %the main diagonal of A
A = spdiags([b c b], [1 0 -1], N-1, N-1); %using spdiags to create A using the
    vectors b and c that we just created
A;

f = [exp(x)]'; %gives f for all x, we dont want f1 or fN+1 in calculating U
f(1) = []; f(end) = []; %removes f(x1) and f(xN+1) from f

U = A\f %finds U for U2....UN

U1 = [0, U', 0]'; %defines U(x_1=0) =0 and U(x_N+1 = 1) = 0
```

1.3 Visualising The Sparse Matrix A

We can use Matlab's spy function to get a visualization of what the Matrix looks like. Here we will use it for N = 8.

```
%visualise the spare matrix we created using the spy function
spy(A)
```

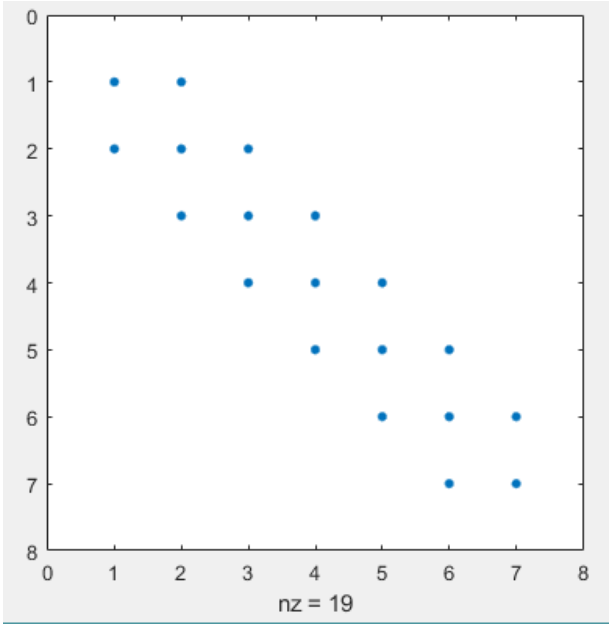

Figure 1: Visualisation of the sparse Matrix A found by using the spy function in Matlab.

This plot tells us there are 19 non-zero entries are saved in the sparse matrix and what locations they are saved in. This is a more efficient method of creating a matrix as it only has to save the non-zero entries, whereas for a full-matrix in Matlab, it has to save every entry in the matrix which can be very inefficient and uses much more of the computer's resources for large computations.

1.4 Creating the Numerical Solution using a Full Matrix

To demonstrate how sparse matrices are more efficient than full matrices in computations, I saved the Matrix A as a full matrix and ran the same computation to compute the numerical solution U.

```
%create full matrix from scratch

S = full(A)

U_ = S\f

U2 = [0, U_', 0]' %defines U(x1=0) =0 and U(xN+1 = 1) = 0
U2
```

From doing this computation I noticed that it took more time to solve for U using a full matrix for A instead of a sparse matrix.

To demonstrate that using sparse matrices is far more efficient than using full matrices, I kept increasing N for each method until my computer ran out of memory and could not finish the computation

For Sparse Matrices, Matlab allowed me to compute U for $N \lessapprox 10^7$. My computer ran out of memory and was unable to complete the computation for $N \approx 10^8$.

For Full Matrices, Matlab allowed me to compute U for $N \lessapprox 10,000$. Matlab returned the following error when trying to compute U for $N = 100,000$ (10^5):

```
Error using full
Requested 99999x99999 (74.5GB) array exceeds maximum array size preference. Creation of arrays greater than this limit may take a long time
and cause MATLAB to become unresponsive.
```

Figure 2: Matlab returned an error when trying to solve for U, using full matrices for A when $N \approx 100,000$

Thus Sparse Matrices are far more efficient than Full Matrices for computations in Matlab.

1.5 Comparing Numerical Solutions to the Analytical Solution

We can use Matlab to plot Numerical Solutions to the problem for different values of N, on the same plot as the analytical solution. This allows us to see whether the Numerical Solution provides a good approximation to the analytical solution.

The plot of the solutions and the associated code can be found in the following 2 pages in the report:

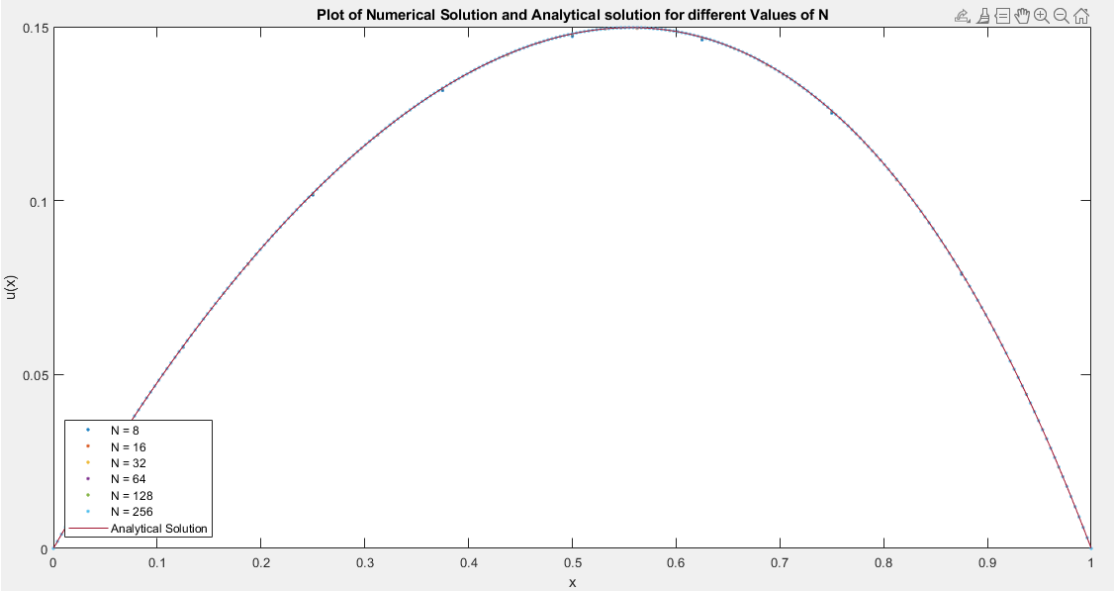

Figure 3: Plotting the analytical solution and numerical solutions for different N values against x.

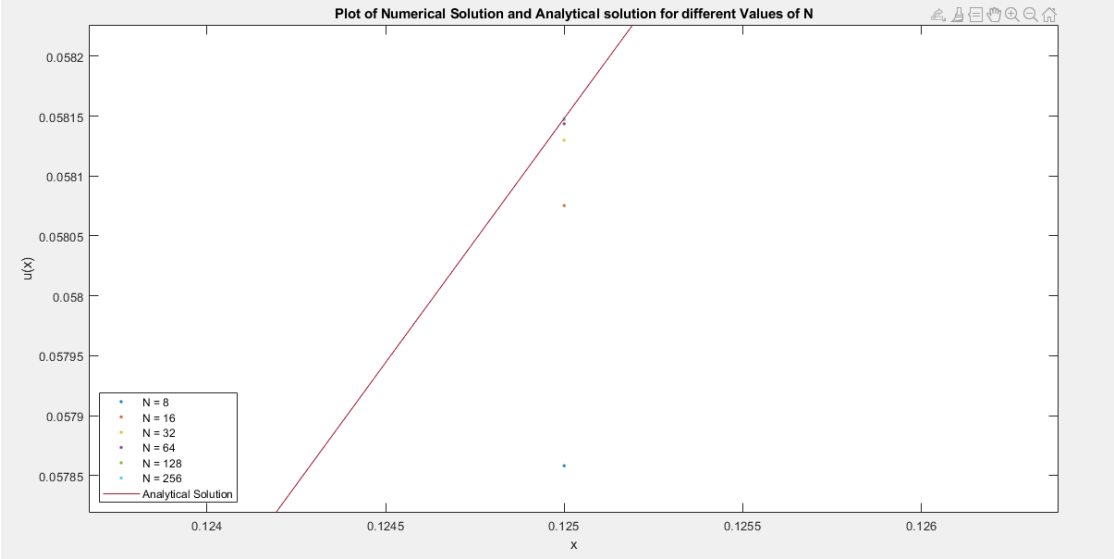

Figure 4: Zoomed in section of figure 3, which shows us how well the points from the Numerical method fit the analytical solution.

Matlab Code used for making the plot:

```
%plot analytic solution and numerical solution for different values of N on
%the same plot

%plot(x_analytic,u_analytic)
for N = [8,16,32,64,128,256] %plotting the numerical approximation of U(x) for
    different values of N
    x = linspace(0,1,N+1)
    U1 = NUMsolve1(N)
    plot(x,U1,'.')
    hold on
end
N = 1024
x = linspace(0,1,N+1)

%plot analytical function
u_analytic = -1/3 * (((1- exp(-1))/(exp(-4) -1)) +1)*exp(2*x) + 1/3 * ((1- exp(-1)
    )/(exp(-4) -1))*exp(-2*x) + 1/3 *(exp(x));


plot(x,u_analytic) %plotting the analytical function against x


hold off
title('Plot of Numerical Solution and Analytical solution for different Values of
    N')
legend({'N = 8','N = 16','N = 32','N = 64','N = 128','N = 256', 'Analytical
    Solution'},'Location','southwest')
xlabel('x')
ylabel('u(x)')
```

From the plot we can see that the numerical method makes a very good approximate solution to our ODE, especially for larger values of N. Next, We will look at the maximum nodal errors to see how they change when N is increased:

```
%calculating the maximum difference between the analytical function and the
%numerical approximation that we created
errors = [];

for N = [8,16,32,64,128,256,512]
    x = linspace(0,1,N+1)
    U2 = NUMsolve1(N)
    y2 = [-1/3 * (((1- exp(-1))/(exp(-4) -1)) +1)*exp(2*x) + 1/3 * ((1- exp(-1))/(
    exp(-4) -1))*exp(-2*x) + 1/3 *(exp(x))]'

    diff1 = U2 -y2
    abs(diff1)
    errors(end+1) = max(abs(diff1))
end

errors
%maximum nodal errors
```

```
errors =

   7.3340e-04   1.8614e-04   4.6596e-05   1.1657e-05   2.9146e-06   7.2867e-07   1.8217e-07
```

Figure 5: Maximum nodal errors between Numerical solution and the analytical solution for increasing values of N

Maximum Nodal Error							
N	8	16	32	64	128	256	512
Error	7.3340e-04	1.8614e-04	4.6596e-05	1.1657e-05	2.9146e-06	7.2867e-07	1.8217e-07

From the table we see that the errors are small for this numerical method so we can say that it is a good numerical approximation to this ODE. We notice as the value of N doubles, the maximum nodal error is divided by 4. Thus we can conclude that for increasing N, the Numerical Approximation becomes closer to the analytical solution to the differential equation.

2 Part II

The following is an outline of the tasks we are asked to do in part II:

- In Part II, we consider a similar boundary problem to the one we solved in Part I, Here we consider the following differential equation

$$-u'' + a(x) \cdot u = f(x), u(0) = u(1) = 0$$
$$a(x) = exp(x^3 - x)$$
$$f(x) = cos(x^2) + \frac{3}{x+1}$$

- Modify the numerical method used in Part I and implement it into Matlab
- Find the maximum double-mesh errors for the numerical method and describe how I implemented it into Matlab code.
- Complete the table containing the double-mesh errors

2.1 Numerical Method for part II

Since our equation is still of the form $-u'' + a(x) \cdot u = f(x)$, we can modify the numerical method we used in part I to create the numerical method for this part.
The only things that have changed in Part II are $a(x)$ and $f(x)$.
The Matrix representation of the problem in part I can be written as follows:

$$\begin{bmatrix} 2N^2 + a(x_2) & -N^2 & 0 & 0 \\ -N^2 & \ddots & -N^2 & 0 \\ 0 & -N^2 & \ddots & -N^2 \\ 0 & 0 & -N^2 & 2N^2 + a(x_N) \end{bmatrix} \cdot \begin{bmatrix} u_2 \\ \vdots \\ \vdots \\ u_N \end{bmatrix} = \begin{bmatrix} f(x_2) \\ \vdots \\ \vdots \\ f(x_N) \end{bmatrix}$$

In Part I $a(x)$ was a constant value of 4, here it depends on x, so we must factor that into our matrix equation. On the right hand side of the equation, $f(x)$ has also changed, we need to put the new function $f(x)$ here.
The Matrix representation of the numerical method for part II is as follows (N = 5 case as an example):

$$\begin{bmatrix} 2N^2 + exp(x_2^3 - x_2) & -N^2 & 0 & 0 \\ -N^2 & 2N^2 + exp(x_3^3 - x_3) & -N^2 & 0 \\ 0 & -N^2 & 2N^2 + exp(x_4^3 - x_4) & -N^2 \\ 0 & 0 & -N^2 & 2N^2 + exp(x_5^3 - x_5)) \end{bmatrix} \cdot \begin{bmatrix} u_2 \\ \vdots \\ \vdots \\ u_5 \end{bmatrix} = \begin{bmatrix} cos(x_2^2) + \frac{3}{x_2+1} \\ cos(x_3^2) + \frac{3}{x_3+1} \\ cos(x_4^2) + \frac{3}{x_4+1} \\ cos(x_5^2) + \frac{3}{x_5+1} \end{bmatrix}$$

Like part I, this is a matrix equation in the form

$$A \cdot U = F$$

and to solve for U in Matlab, we write $U = A\backslash F$ and apply the boundary conditions U(0) = U(1) = 0

```
%%%%%%%%%%%%%% Part 2



%solving the following ODE for given a(x) and given f(x)
%-u(x)'' + a(x)*u(x) = f(x), u(0) = u(1) = 0

%a(x) = exp(x^3 - x)

%f(x) = 5/(1+x+x^2)

%define N number of subintervals between 0 and 1
N=200

x = linspace(0,1,N+1) %create vector of N+1 values of x between 0 and 1

%visualise ODE as matrix equation AU = F, where we are trying to solve for
%U

%create sparse Matrix A, which implements the 2nd order central finite difference
%method to our problem as a matrix equation. A is multiplied by U to give us the
%right hand side of our ODE

h = -(N^2)*ones(N-1,1); %the diagonals directly beneath and above A
g = (2*N^2)*ones(N-1,1); %the main diagonal of A
B = spdiags([h g h], [1 0 -1], N-1, N-1); %using spdiags to create A using the
    vectors b and c that we just created
B

a = [exp(x.^3 - x)]'
a(1) = []; a(end) = [];

C = spdiags([a], [0], N-1, N-1)


D = B + C


f2 = [cos(x.^2) + 3./(x+1)]'; %gives f for all x, we dont want f1 or fN+1 in
    calculating U
f2(1) = []; f2(end) = []; %removes f(x1) and f(xN+1) from f

U4 = D\f2 %finds U for U2....UN

U3 = [0, U4', 0]'; %defines U(x1=0) =0 and U(xN+1 = 1) = 0

U3 %numerical solution vector for part II
```

Then we plotted the numerical solution to part II.

```
1
2 plot(x,U3)
3 title('Plot of Numerical Solution to $$-u^{\prime \prime}(x) + exp(x^3 -x)\cdot u(
      x) = cos(x^2) + \frac{3}{x+1}$$','interpreter','latex')
4 xlabel('x')
5 ylabel('u(x)')
```

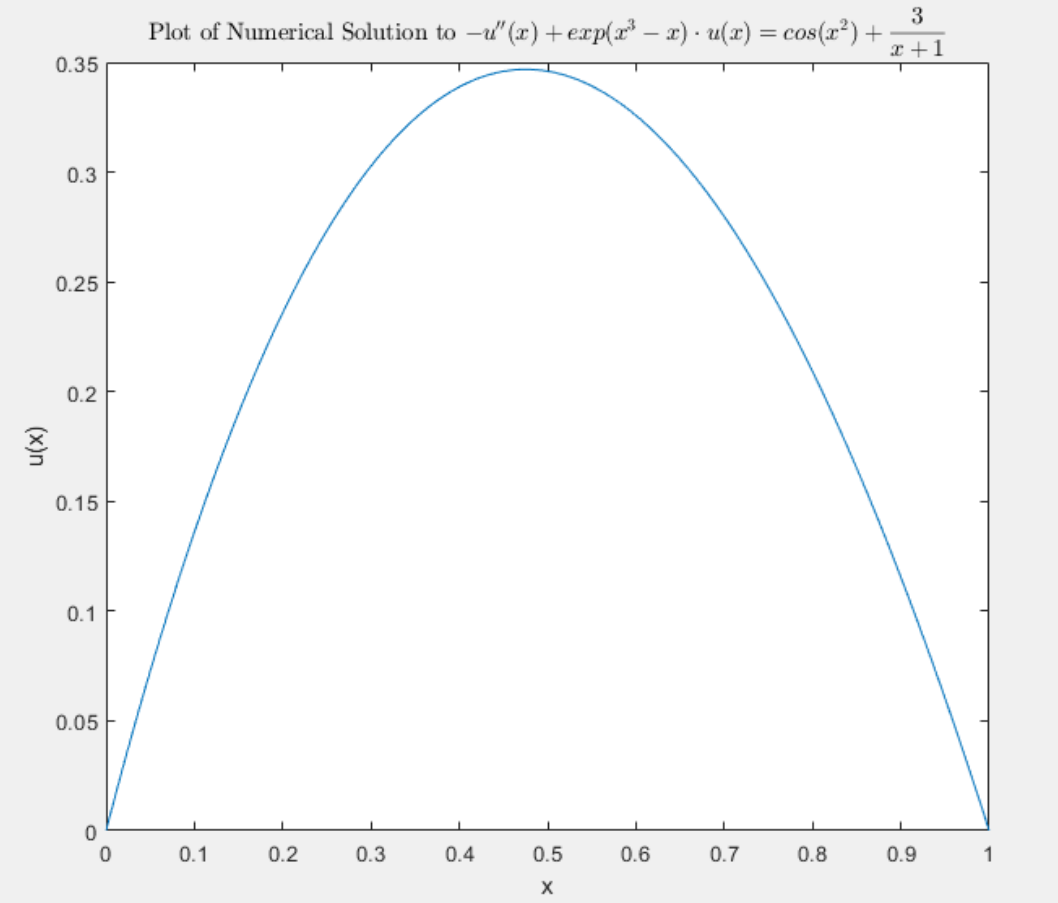

Figure 6: Plot of the numerical solution u(x) to the ODE in part II against x for N = 200

2.2 Finding Double-Mesh Errors

To find the Double Mesh Error for a numerical solution, we compute a new numerical solution where N is now twice the value of N used for the previous numerical solution. Like finding the error for part II, we find the difference between these two solutions, removing every second value of the second solution and then taking the difference of the two solutions and finding the maximum absolute difference between them.

The idea behind this is that when N is larger, the numerical solution is on a finer mesh of x values, so it should in theory be closer to the actual analytical solution to the problem. The double-mesh error is not the actual error of the numerical method but it can tell us if the errors are increasing/decreasing and at what rate do they increase or decrease depending on how N is changed.

For this part I created a function NUMsolve2(N) which computes the numerical solution U for a given N. Then I found the maximum absolute difference between $U(N)$ and $U(2N)$ which gives me the double-mesh error.

```
dmesh_errors = [];


for N = [16,32,64,128,256,512]
    x = linspace(0,1,N+1)
    U = NUMsolve2(N)
    U2 = NUMsolve2(2*N)

    U2(1:2:end-1) = [] %remove 2nd elements in U2 array so we can compare U wtih
    U2 on the same x points

    diff2 = U2 - U %difference between U(N) and U(2N)

    dmesh_errors(end+1) = max(abs(diff2))%compute double mesh error for each N
end

dmesh_errors %double mesh errors
```

```
dmesh_errors =

   4.6605e-02   2.3784e-02   1.2013e-02   6.0371e-03   3.0262e-03   1.5150e-03
```

Figure 7: Computed Double-Mesh Errors for part II

Double-Mesh Errors						
N	16	32	64	128	256	512
Error	4.6605e-02	2.3784e-02	1.2013e-02	6.0371e-03	3.0262e-03	1.5150e-03

We can see from the table that as N doubles, the double-mesh Error halves. From this we see that the error reduces as N increases, and thus we can deduce that the numerical solution becomes closer to the exact solution as N is increased.