MS6021 Scientific Computation

Assignment 2

Dara Corr - 22275193

October 15, 2022

Contents

1 Introduction

In this assignment we investigate 3 different methods to compute numerical solutions to the semi-linear problem:

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

We consider 3 particular cases for this problem:

$$f(x,u) = \frac{u - cos(x)}{2-u} - exp(4x) \tag{1}$$

$$f(x,u) = 3(u^4 - 1) - x^2 e^{5x} \tag{2}$$

$$f(x,u) = 200(u^4 - 1) - x^2 e^{5x} \tag{3}$$

The three numerical methods we are going to investigate are:

- Fixed Point Iteration Method
- Newton's (Newton-Raphson) Method
- Damped Newton's Method

2 Fixed Point Iteration Method

We can start to understand the Fixed Point Iteration method by considering an equation $f(x) = 0$. We can then rewrite the equation in the form $x = f(x)$.
This method is applied by taking a value x_0 as an initial guess. Then we can generate the sequence to determine better approximations for the value of x which solves the equation:

$$x_{n+1} = g(x_n)$$

This scheme takes an initial guess x_0 and substitutes it into the right hand side of the equation. If the sequence converges, then with each iteration the value of x generated should be closer to the real value of x. If the sequence does not converge then either we could try a different initial guess to see if it will work or we may need to try a different method.
We will try to implement the fixed point iteration method scheme to get a numerical solution to our problem. First we consider the equation for our problem:

$$-u''(x) + f(x, u) = 0$$

we rewrite this equation so that we can apply the Fixed point iteration method:

$$u^{(n+1)\prime\prime} = f(x, u^{(n)})$$

where (n) in this case denotes the number of iterations of this method (note N denotes the size of the x and U matrices)
To apply this method in Matlab we discretise the left hand side of the equation using the centred finite difference method as we did in Assignment 1:

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

where $h = \frac{1}{N}$ and wrote our problem as a Matrix equation:

$$u^{(n+1)\prime\prime} = f(x, u^{(n)})$$

$$AU^{(n+1)} = F(x, U^{(n)})$$

$$\begin{bmatrix} -2N^2 & N^2 & 0 & 0 \\ N^2 & \ddots & N^2 & 0 \\ 0 & N^2 & \ddots & N^2 \\ 0 & 0 & N^2 & -2N^2 \end{bmatrix} \cdot \begin{bmatrix} u_2^{(n+1)} \\ \vdots \\ \vdots \\ u_N^{(n+1)} \end{bmatrix} = \begin{bmatrix} f(x_2, u_2^{(n)}) \\ \vdots \\ \vdots \\ f(x_N, u_N^{(n)}) \end{bmatrix}$$

To find the Next value of $U^{(n+1)}$, we run this as an algorithm in a for loop in Matlab.
We solve for the matrix $U^{(n+1)}$ by writing:

$$U^{(n+1)} = A \backslash F$$

This algorithm repeats itself, for a specified maximum number of iterations (denoted as k_MAX in my MATLAB code) or until it reaches a stopping criterion.

In my code, I utilised two different stopping criteria for all three approximation methods. The first criterion checks that the maximum absolute difference between $U^{(n+1)}$ and $U^{(n)}$ is below a specified tolerance value (TOL): $|U^{(n+1)} - U^{(n)}| < \text{TOL}$. The second criterion checks that the equation $-u'' + f(x,u) = 0$ is verified within a given tolerance value (TOL) $|-AU^{(n+1)} + f(x,U)| < \text{TOL}$. I.e. the left hand side of the equation is minimized. For this assignment I used a TOL value of 10^{-6}.

Code used for The Fixed Point Iteration Method implementation can be found below:

```
function [result,iterations_performed,flag] = FixedIterfunc(N,g,k_MAX,TOL) %
    outputs result, no. of iterations performed and flag
%input -> size of mesh N, input function g, k_max no. of max iterations and
%stopping criterion TOL

x = linspace(0,1,N+1);
x(1) = []; x(end) = [];

U = [zeros(N-1,1)]';

b = (N^2)*ones(N-1,1); %the diagonals directly beneath and above A
c = -(2*N^2)*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;

iterations_performed = 0

    for range = 1:k_MAX
            Un = g(x,U);%rhs u(n)_
            U_next = A\(Un); %find lhs -> calculates u(n+1)
            diff = [U_next - U'];
            iterations_performed = iterations_performed + 1;


            if  ( max(abs(diff)) < TOL ) & ( max(abs(-A*U_next + g(x,U))) <
    TOL );
                flag = 1;
                break

            else
                flag = 0;
                U = U_next';
            end
    end

    result = U_next
```

```
    iterations_performed = iterations_performed
    flag
end
```

```
%Implementation of fixed iteration method for our problem:
%-u(x)'' + f(x,u) = 0, u(0) = u(1) = 0

g1 = @(x,U)(U - [cos(x)]')/(2-U) - [exp(4.*x)]';%defining f(x,u) in the
    equation -u'' + f(x,u) = 0
g2 = @(x,U) 3.*(U'.^4 - 1) -[x.^2 .*exp(5*x)]';
g3 = @(x,U) 200.*(U'.^4 - 1) - [x.^2 .*exp(5.*x)]';

%define N
N = 100

x = linspace(0,1,N+1)%define x
x(1) = []; x(end) = [];

U = [zeros(N-1,1)]'; %define U

b = (N^2)*ones(N-1,1); %the diagonals directly beneath and above A
c = -(2*N^2)*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;

%[result,iterations_performed,flag] = FixedIterfunc(N,g1,20,10e-6) %works -
%5 iterations for N=100 and N = 1000


%[result,iterations_performed,flag] = FixedIterfunc(N,g2,3,1) %does not work
    as expected
%-> does not hit stopping criteria -> does not converge

[result,iterations_performed,flag] = FixedIterfunc(N,g3,4,10e-6)
% -> fixed iteration not effective for this function -> divergent

U_final = [0,result',0]

x = [0,x,1]

plot(x,U_final)
xlabel('x')
ylabel('u(x)')

fprintf('Iterations Performed = %d \n',iterations_performed);

fprintf('Flag = %d \n', flag);
```

Plots produced:

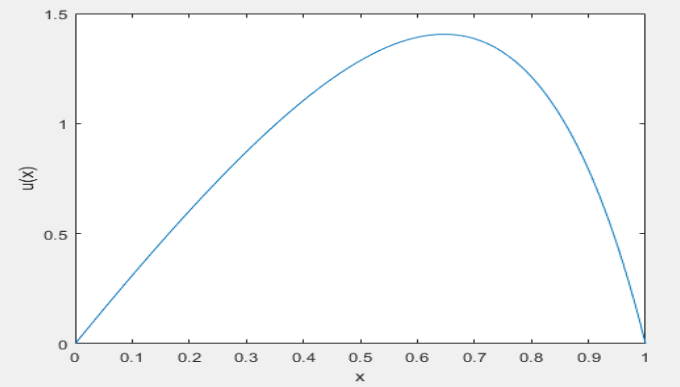

Figure 1: Plot of u(x) against x for the first case of the problem using fixed point iteration method

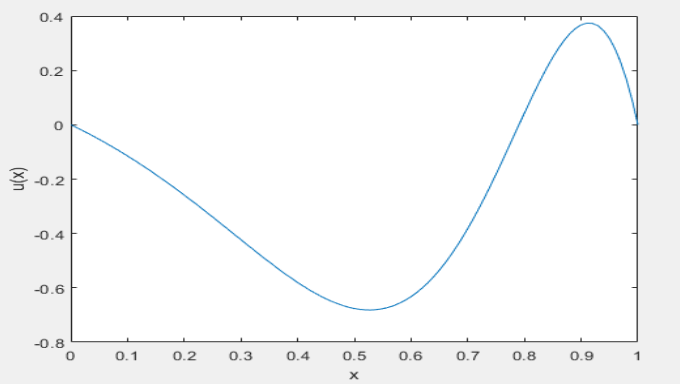

Figure 2: Plot of u(x) against x for the second case of the problem using fixed point iteration method for odd number of iterations

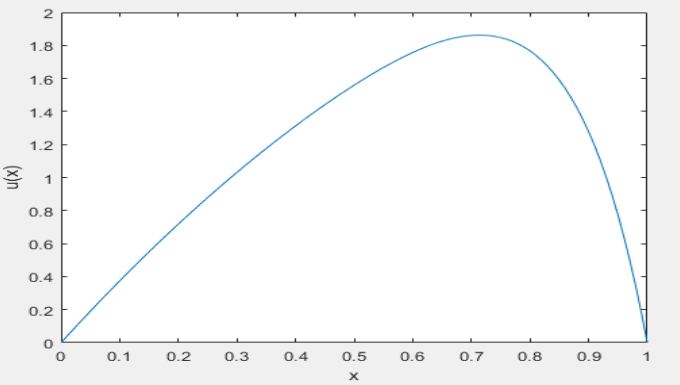

Figure 3: Plot of u(x) against x for the second case of the problem using fixed point iteration method for even number of iterations

Notes on Results:

- The fixed point iteration method works well for the first case, finding a soltution that reaches the stopping criteria after 5 iterations.
- This method does not work well for the second case of this problem, while it does produce solutions the solutions are inconsistent and oscillate between two different solutions after each iteration. Even number of iterations produce a different solution to a solution produced using an odd number of iterations of this method. This method does not converge to a solution for this problem (does not meet stopping criteria). Thus I conclude that the fixed point iteration method is not suited to case 2 of our problem.
- No plot is produced for the third case of the problem after several iterations, the solution grows to infinity after about 5 iterations and it does not converge. It is divergent and the fixed point iteration is not suited for this problem. Other methods should be investigated instead such as Newton's Method.

3 Newton's Method

Newton's Method (also known as Newton-Raphson method) is the next numerical method we will use to compute numerical solutions to our problem.
Newton's method uses the equation for tangent lines to the curve $y = f(x)$ at $x = x_0$

$$y = f(x_0) + f'(x_0)(x - x_0)$$

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

to find the root to an equation $f(x) = 0$. We can make an initial guess for x, implement the method and then take our next value of x as the value of x where the tangent line intercepts the x axis. as the process is repeated, x becomes closer with each iteration to the root r which solves $f(r) = 0$.
We will implement this method to our problem by rewriting the equation in terms of x,u and f(x,u).

$$-u^{(n+1)\prime\prime} + f(x, u^{(n)}) + f_u(x, u^{(n)})(u^{(n+1)} - u^{(n)}) = 0$$

$$-u^{(n+1)\prime\prime} + f_u(x, u^{(n)})u^{(n+1)} = -f(x, u^{(n)}) + f_u(x, u^{(n)})$$

The Matrix form for our scheme is then:

$$AU^{(n+1)} = -F(X, U^{(n)}) + F_u(X, U^{(n)})$$

$$\begin{bmatrix} 2N^2 + f(x_2, u_2^{(n)}) & -N^2 & 0 & 0 \\ -N^2 & 2N^2 + f(x_3, u_3^{(n)}) & -N^2 & 0 \\ 0 & -N^2 & \ddots & -N^2 \\ 0 & 0 & -N^2 & 2N^2 + f(x_N, u_N^{(n)}) \end{bmatrix} \cdot \begin{bmatrix} u_2^{(n+1)} \\ \vdots \\ \vdots \\ u_N^{(n+1)} \end{bmatrix} = \begin{bmatrix} -f(x_2, u_2^{(n)}) + f_u(x_2, u_2^{(n)}) \\ \vdots \\ \vdots \\ -f(x_N, u_N^{(n)}) + f_u(x_N, u_N^{(n)}) \end{bmatrix}$$

so then to get the solution vector U:

$$U^{(n+1)} = A \backslash \left[-F(X, U^{(n)}) + F_u(X, U^{(n)}) \right]$$

Note: since $f_u(x, U^{(n)})$ is a function of U it must be calculated in the for loop at each iteration of the method in MATLAB.

I calcuated the three derivatives with respect to u $f_u(x, u)$ on paper:

- (1) $f_u = \frac{2 - cos(x)}{(2-u)^2}$
- (2) $f_u = 12u^3$
- (3) $f_u = 800u^3$

```
function [result,iterations_performed,flag] = NewtonMethodfunc(N,g,g_u,k_MAX,
    TOL)%outputs result, no. of iterations performed and flag
%input -> size of mesh N, input function g,derivative of g wrt u:  g_u , k_max
     no. of max iterations and
%stopping criterion TOL
x = linspace(0,1,N+1)
x(1) = []; x(end) = [];

U = [zeros(N-1,1)];

iterations_performed = 0

    for range = 1:k_MAX
            Un = -[g(x,U)] + [g_u(x,U)].*U;%rhs u(n)_
            Un

            b = -(N^2)*ones(N-1,1); %the diagonals directly beneath and above
    A
            c = (2*N^2)*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 = [g_u(x,U)];
            B = spdiags([a], [0], N-1, N-1);

            C = A+B;


            U_next = C\(Un); %find lhs -> calculates u(n+1) %U_next represents
     u(n+1)
            diff = [U_next - U];


            iterations_performed = iterations_performed + 1;



            if  (max(abs(diff)) < TOL) & (max(abs(A*U_next + g(x,U_next)  )) <
     TOL  )
                flag = 1;
                break

            else
                flag = 0;
                U = U_next;
            end

    end

    result = U_next
    iterations_performed = iterations_performed
    flag

end
```

```
%-u(x)'' + f(x,u) = 0, u(0) = u(1) = 0


g1 = @(x,U)([U] - [cos(x)]')./(2-[U]) - [exp(4.*x)]';
g2 = @(x,U) 3.*([U].^4 - 1) -[x.^2 .*exp(5.*x)]';
g3 = @(x,U) 200.*([U].^4 - 1) - [x.^2 .*exp(5.*x)]';

g1_u = @(x,U)(2-[cos(x)]')./([2-[U]].^2); %derivatives wrt u
g2_u = @(x,U)12.*[U.^3]
g3_u = @(x,U)800.*[U.^3]

%define N
N = 100

x = linspace(0,1,N+1)%define x
x(1) = []; x(end) = [];

U = [zeros(N-1,1)]; %define U

%[result,iterations_performed,flag] = NewtonMethodfunc(N,g1,g1_u,20,10e-6) %
    works - 4 iterations for N = 100/1000

%[result,iterations_performed,flag] = NewtonMethodfunc(N,g2,g2_u,20,10e-6)
%works - 6 iterations for N = 100 and N = 1000

[result,iterations_performed,flag] = NewtonMethodfunc(N,g3,g3_u,20,10e-6)
%works - 16 iterations for N = 100 and N = 1000

U_final = [0,result',0]

x = [0,x,1]
plot(x,U_final)
xlabel('x')
ylabel('u(x)')

fprintf('Iterations Performed = %d \n',iterations_performed);

fprintf('Flag = %d \n', flag);
```

Plots produced:

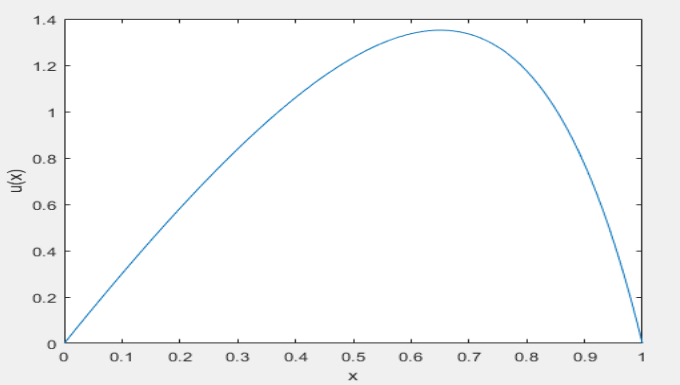

Figure 4: Plot of u(x) against x for the first case of the problem using the Newton-Raphson Method

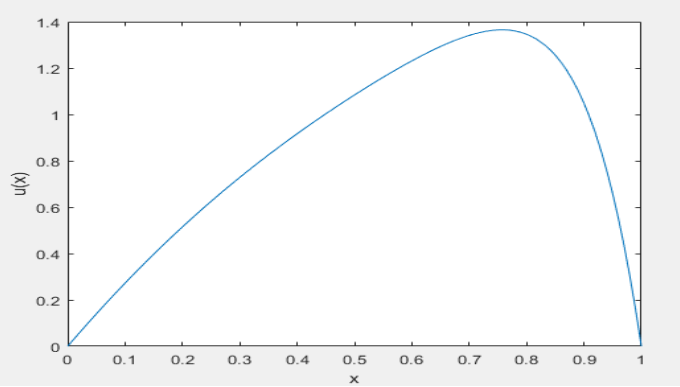

Figure 5: Plot of u(x) against x for the second case of the problem using the Newton-Raphson Method

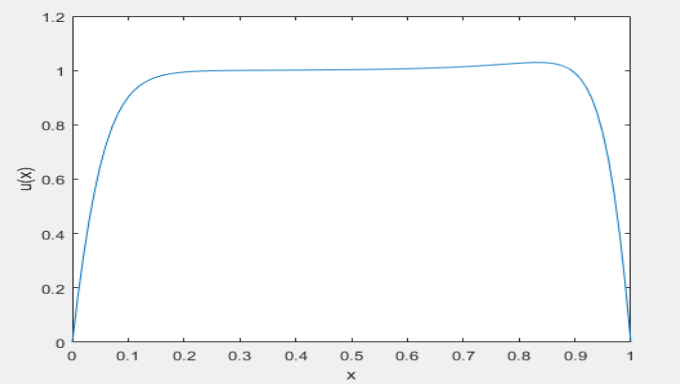

Figure 6: Plot of u(x) against x for the third case of the problem using the Newton-Raphson Method

Notes on Results:

- Newton's Method produces a result in 4 iterations. This method is more efficient than Fixed iteration for case one of this problem as it produces the solution in one less iteration. It obtains the numerical solution in one less iteration than the Fixed Point Iteration method does.
- Newton's Method produces a solution that satisfies the stopping criteria for case two of our problem in 6 iterations. This is a much better improvement than the Fixed Iteration Method for this case as the Fixed Point Iteration Method failed to produce a convergent solution for this case of the problem.
- Newton's Method produced a solution to case three of our problem in 16 iterations. This is a big improvement from using Fixed Point Iteration for this case of the problem, where the solutions diverged very quickly with each new iteration. 16 Iterations is still relatively large compared to 4 and 6 iterations that were achieved with Newton's method for the first two cases of this problem. Next we will look to the Damped Newton-Raphson Method to see if it can address this problem.

4 Damped Newton's Method

Damped Newton's Method is a version of Newton's Method where the increment (V) added to $U^{(n)}$ to calculate $U^{(n+1)}$ at each iteration is damped by a scalar value $0 < \alpha \leq 1$.

The idea behind this method is that with Newton's Method, the first few iterations of Newton's Method may overshoot and miss where the root or solution is and it will take more iterations to correct this or the solution may not even converge at all based on this initial guess.

Damped Newton's Method damps the increment (by multiplying by scalar α) added to the intial guess with each iteration such that the value of α used, makes the left hand side of the equation $-u'' + f(x, u) = 0$ closest to zero. It is much more computationally inexpensive to check for the optimal value of α to dampen by in this method than it is to evaluate $U^{(n+1)}$ using Newton's Method, which is why this method is effective.

Newton's increment $V^{(n+1)}$ is added to $U^{(n)}$ to calculate $U^{(n+1)}$ using Newton's Method:

$$U^{(n+1)} = U^{(n)} + V^{(n+1)}$$

Now we apply damping with damping coefficient α to calculate $U^{(n+1)}$ using Damped Newton's Method:

$$U^{(n+1)} = U^{(n)} + \alpha^{(n+1)} V^{(n+1)}$$

In my implementation of Damped Newton's Method I found Newton's increment by calculating $u^{(n+1)}$ using Newton's Method as previously shown and finding the difference between $U^{(n+1)}$ and $U^{(n)}$

$$V^{(n+1)} = U^{(n+1)} - U^{(n)}$$

And then I used an array of values for α in this method, where $[\alpha] = \frac{1}{2^{[0:5]}} = \left(1, \frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{16}, \frac{1}{32}\right)$

I found which value of α minimised the equation for that iteration and then checked our two stopping criteria:

$$|(U^{(n)} + \alpha \cdot V^{(n+1)}) - U^{(n)}| < TOL$$

$$|A \cdot (U^{(n)} + \alpha \cdot V^{(n+1)}) + f(x, U^{(n)} + \alpha \cdot V^{(n+1)})| < TOL)$$

and depending on whether the criertia were satisfied or not, the loop will finish with result $= U^{(n)} + \alpha_{optimum} \cdot V^{(n+1)}$ or the loop will begin again with initial value of U $= U^{(n)} + \alpha_{optimum} \cdot V^{(n+1)}$

```
function [result,iterations_performed,flag] = DampedNewtonfunc(N,g,g_u,k_MAX,
    TOL)%outputs result, no. of iterations performed and flag
%input -> size of mesh N, input function g,derivative of g wrt u:  g_u , k_max
     no. of max iterations and
%stopping criterion TOL
x = linspace(0,1,N+1)
x(1) = []; x(end) = [];

U = [zeros(N-1,1)];


k = [0:5]
alphas = [1./(2.^k)]


iterations_performed = 0

    for range = 1:k_MAX
            Un = -[g(x,U)] + [g_u(x,U)].*U;%rhs u(n)_


            b = -(N^2)*ones(N-1,1); %the diagonals directly beneath and above
    A
            c = (2*N^2)*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 = [g_u(x,U)];
            B = spdiags([a], [0], N-1, N-1);

            C = A+B;


            U_next = C\(Un); %find lhs -> calculates u(n+1) %U_next represents
     u(n+1)



            V= U_next - U

            test_vals = []
            for i = 1:length(alphas)
                %index of min value of f(x,Un+alpha*V) should correspond to
    the optimum alpha's
                %index

                test_vals(end+1) = max(abs(A*(U + alphas(i).*V ) + g(x,U +
    alphas(i).*V ) ) )
            end
            [M, I] = min(test_vals)
            alpha_optimum = alphas(I)

            diff = [(U+ alpha_optimum.*V) - U];

```

```
            iterations_performed = iterations_performed + 1;



            if  (max(abs(diff)) < TOL) & (max(abs(A*(U+ alpha_optimum.*V) + g(x,(U+ alpha_optimum.*V))  )) < TOL  )
                flag = 1;
                break

            else
                flag = 0;
                U = (U+ alpha_optimum.*V);
            end

    end

    result = (U+ alpha_optimum.*V)
    iterations_performed = iterations_performed
    flag

end
```

```
%%
%Implementation of Damped Newton-Raphson Method for our problem:
%-u(x)'' + f(x,u) = 0, u(0) = u(1) = 0


g1 = @(x,U)([U] - [cos(x)]')./(2-[U]) - [exp(4.*x)]';
g2 = @(x,U) 3.*([U].^4 - 1) -[x.^2 .*exp(5.*x)]';
g3 = @(x,U) 200.*([U].^4 - 1) - [x.^2 .*exp(5.*x)]';

g1_u = @(x,U)(2-[cos(x)]')./([2-[U]].^2); %derivatives wrt u
g2_u = @(x,U)12.*[U.^3]
g3_u = @(x,U)800.*[U.^3]

%define N
N = 100

x = linspace(0,1,N+1)%define x
x(1) = []; x(end) = [];

U = [zeros(N-1,1)]; %define U

%[result,iterations_performed,flag] = DampedNewtonfunc(N,g1,g1_u,20,10e-6) %works - 4 iterations for N = 100/1000

%[result,iterations_performed,flag] = DampedNewtonfunc(N,g2,g2_u,20,10e-6)
%works - 6 iterations for N = 100 and N = 1000

[result,iterations_performed,flag] = DampedNewtonfunc(N,g3,g3_u,20,10e-6)
%works - 7 iterations for N = 100 and N = 1000
```

```
29
30 U_final = [0,result',0]
31
32 x = [0,x,1]
33 plot(x,U_final)
34 xlabel('x')
35 ylabel('u(x)')
36
37 fprintf('Iterations Performed = %d \n',iterations_performed);
38
39 fprintf('Flag = %d \n', flag);
```

Plots produced:

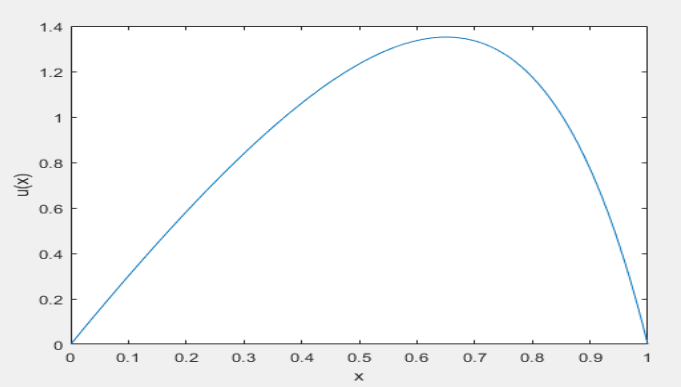

Figure 7: Plot of u(x) against x for the first case of the problem using the Damped Newton Method

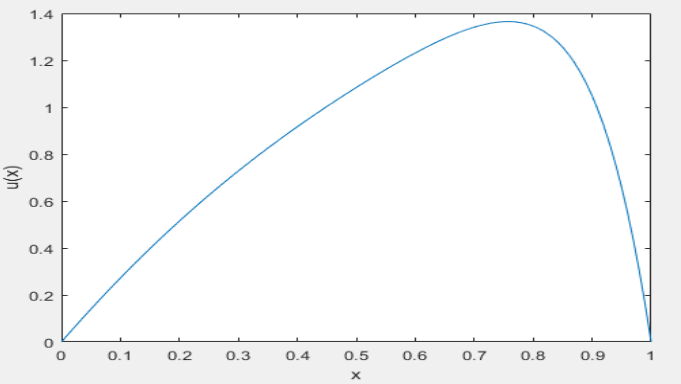

Figure 8: Plot of u(x) against x for the second case of the problem using the Damped Newton Method

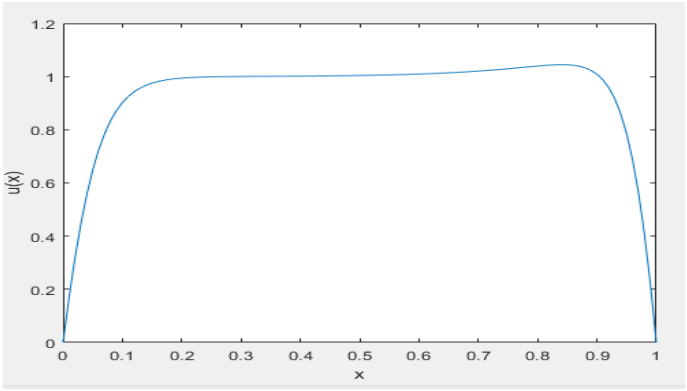

Figure 9: Plot of u(x) against x for the third case of the problem using the Damped Newton Method

Notes on results:

- For cases 1 and 2 of our problem, the plots of solutions remain the same and the number of iterations are the same (4 iterations and 6 iterations respectively) as they were when we used the undamped Newton Method. We can say that the undamped case ($\alpha = 1$), is the optimum for these cases and the damped method does not make computing these solutions anymore efficient.
- For the third case of our problem, The number of iterations performed drops from 16 iterations using the undamped method to 7 iterations using the damped method. This is a huge improvement in efficiency for this method and it shows that dampening the increments allowed the solution to be found faster in this case.

5 Conclusions

- I conclude that the Fixed Point Iteration method is not very suited to solving semi-linear problems of this type. While it manages to solve the first case of this problem, the other two cases fail to converge using this method. Although it is the most straightforward method to implement, it is also the slowest method of the three methods which I investigated.
- Newton's method is a good method for solving these semi-linear problems of this type. Newton's Method converged to a solution for all three cases of our problem and it is also a quicker method than the Fixed Point Iteration Method.
- Sometimes the initial guess with Newton's Method will make the next guess very far from the solution, and this will take time for Newton's method to correct this. The Damped Newton Method is introduced to correct this. Another downside to Newton's Method is that it needs a good initial guess located close to the minimum of the function we are trying to solve.
- Damped Newton's Method is the fastest method of the list. It can be further optimised by using damped newton's method for the first few iterations and then swapping to a different numerical method i.e. Undamped Newton's Method once the guesses are close enough to the minimum of the function.

5.1 Table of Results

Table of results denoting the number of iterations for each method with N = 100 and N = 1000 (DC denotes solutions which do not converge)

	Problem (1)	Problem (2)	Problem (3)	Problem (1)	Problem (2)	Problem (3)
Fixed Iteration	5	DC*	DC*	5	DC*	DC*
Newton	4	6	16	4	6	16
Damped Newton	4	6	7	4	6	7
	N= 100			N= 1000		