3.
(a)
For

$$-y''(x) + \mu y(x)y'(x) = f(x), x\epsilon(0,1)$$

$$y(0) = \alpha, y(1) = \beta$$

We need to consider the finite difference form. We will be using the three point midpoint finite difference formula for approximating the first derivative, and its second derivative analog. These formulas are as follows:

$$y''(x) \approx \frac{y(_{i+1} - 2y_i + y_{i-1})}{h^2}$$

$$y'(x) \approx \frac{y_{i+1} - y_{i-1}}{2h}$$

where h is the step size that separates the consecutive y_i. The initial and final boundary values are given, so that $y(0) = y(1) = \alpha$ and $y(1) = y(N) = \beta$ and $i = 2, ..., N-1$. Since our method is only an approximation to $y(x_i)$, we replace $y(x_i)$ with $u(x_i)$, where $u(x_i)$ is an approximation. We can now re-write the differential equation in terms of $u(x_i)$ and finite difference:

$$-\frac{y(_{i+1} - 2y_i + y_{i-1})}{h^2} + \mu(u_i)\frac{y_{i+1} - y_{i-1}}{2h} = f(x_i)$$

Considering the number of nodes N for which we are going to solve, we can write all of the finite difference equations in the form $\vec{F}(\vec{u}) = \vec{0}$, composed of the following equations and $i = 2, ...N-1$:

$$u_1 - \alpha = 0$$

$$-\frac{y(_{i+1} - 2y_i + y_{i-1})}{h^2} + \mu(u_i)\frac{y_{i+1} - y_{i-1}}{2h} - f(x_i) = 0, \quad i = 2, 3, ..., N-1$$

$$u_N - \beta = 0$$

which is,

$$\vec{F}(\vec{u}) = \begin{pmatrix} f_1(\vec{u}) \\ f_2(\vec{u}) \\ f_3(\vec{u}) \\ \vdots \\ f_{N-1}(\vec{u}) \\ f_N(\vec{u}) \end{pmatrix} = \begin{pmatrix} u_1 - \alpha \\ -u_3 + 2u_2 - u_1 + \frac{1}{2}h\mu u_2 u_3 - \frac{1}{2}h\mu u_2 u_1 - h^2 f(x_2) \\ -u_4 + 2u_3 - u_2 + \frac{1}{2}h\mu u_3 u_4 - \frac{1}{2}h\mu u_3 u_2 - h^2 f(x_3) \\ \vdots \\ -u_N + 2u_{N-1} - u_{N-2} + \frac{1}{2}h\mu u_{N-1} u_N - \frac{1}{2}h\mu u_{N-1} u_{N-2} - h^2 f(x_{N-1}) \\ u_N - \beta \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 0 \\ 0 \end{pmatrix}$$

The following code was used to program $\vec{F}(\vec{u})$.

```
function F = proj2_3_nonlinearF(u, x, q_fun, f_fun, alpha, beta)
N = length(u);
F = zeros(size(u));
h = x(2) - x(1);
F(1) = u(1) - alpha;
F(N) = u(N) - beta;

%For this specific project problem, q_fun is mu
for i = 2:N-1
    F(i) = -u(i-1) + 2*u(i) - u(i+1) + (1/2)*h*q_fun*u(i)*u(i+1) -...
        (1/2)*h*q_fun*u(i)*u(i-1) - (h^2)*f_fun(x(i));
end
end
```

(b)

To the find Jacobian of the above system of equations, we need to take the partial derivative of each $f_i(\vec{u})$ with respect to every u_i. To simplify the daunting task of finding the Jacobian of a potentially extremely large matrix, we can note that f_1 is only a function of u_i. Therefore, no matter how large N is, the first row of the Jacobian will be composed by the row vector

$$\left(\frac{\partial f_1(\vec{u})}{\partial u_1}, \frac{\partial f_1(\vec{u})}{\partial u_2}, ..., \frac{\partial f_1(\vec{u})}{\partial u_N}\right)$$

which is zero for all $u_i \neq u_1$. Thus, the entire first row is zero except for the first entry, $J_{11} = \frac{\partial f_1(\vec{u})}{\partial u_1} = \frac{\partial}{\partial u_1}(u_1 - \alpha) = 1$

A similar argument can be shown for the N^{th} row of the Jacobian, such that the last row is populated by the row vector

$$\left(0, 0, 0, ..., 0, 1\right)$$

since f_N is only a function of u_N. For every row in the Jacobian between the first and last row, we should take a look to see if a pattern emerges while evaluating the partial derivatives.

We begin by examining the second row of the vector, $f_2(\vec{u})$. A method of brute force will be employed to evaluate each partial derivative. We begin with the first partial derivative with respect to u_1:

$$\frac{\partial}{\partial(u_1)} f_2(\vec{u}) = \frac{\partial}{\partial(u_1)}(-u_3 + 2u_2 - u_1 + \frac{1}{2}h\mu u_2 u_3 - \frac{1}{2}h\mu u_2 u_1 - h^2 f(x_2))$$

which yields

$$-1 - \frac{1}{2}h\mu u_2$$

We next evaluate the same function with the partial with respect to u_2:

$$\frac{\partial}{\partial(u_2)} f_2(\vec{u}) = \frac{\partial}{\partial(u_2)}(-u_3 + 2u_2 - u_1 + \frac{1}{2}h\mu u_2 u_3 - \frac{1}{2}h\mu u_2 u_1 - h^2 f(x_2))$$

which yields

$$2 + \frac{1}{2}h\mu u_3 - \frac{1}{2}h\mu u_1$$

And lastly for this equation,

$$\frac{\partial}{\partial(u_3)} f_2(\vec{u}) = \frac{\partial}{\partial(u_1)}(-u_3 + 2u_2 - u_1 + \frac{1}{2}h\mu u_2 u_3 - \frac{1}{2}h\mu u_2 u_1 - h^2 f(x_2))$$

which gives

$$\frac{\partial}{\partial(u_3)} f_2(\vec{u}) = -1 + \frac{1}{2}h\mu u_2$$

At this point, one can note that $f_2(\vec{u})$ is only a function of u_1, u_2, and u_3. This then permits us to only take the first partial derivatives of $f_2(\vec{u})$ with respect to u_1, u_2, and u_3. Since the $f_2(\vec{u})$ contains no terms from $\{u_4, u_5, ...u_N\}$, all of those partial derivatives are zero. Hence we are done with $f_2(\vec{u})$.

We now move onto the third function of $\vec{F}(\vec{u})$, which is$f_3(\vec{u})$. The same process of partial derivatives is repeated. One might note that $f_3(\vec{u})$ is only a function of $\{u_2, u_3, u_4\}$. Then any partial derivative not with respect to those u_i is equivalent to zero. This, again, greatly reduces the mathematical workload for partial derivative evaluations. With the observation

$$\frac{\partial}{\partial(u_1)} f_3(\vec{u}) = 0$$

we continue with the mathematical intricacies omitted for brevity,

$$\frac{\partial}{\partial(u_2)} f_3(\vec{u}) = -1 - \frac{1}{2}h\mu u_3$$

and

$$\frac{\partial}{\partial(u_3)} f_3(\vec{u}) = 2 + \frac{1}{2}h\mu u_4 - \frac{1}{2}h\mu u_2$$

and lastly,

$$\frac{\partial}{\partial(u_4)} f_3(\vec{u}) = -1 + \frac{1}{2}h\mu u_3$$

At this point, the reader may notice that a pattern has indeed surfaced. For each $f_i(\vec{u})$, with $i = 2, ..., N-1$, every column of that row will be zero except for the $i-1, i$, and $i+1$ columns. With the knowledge of the first row, last row, and every row in between, we can now construct the Jacobian. The following notation is used for the generalized form: J(a,b) denotes the a^{th} row and b^{th} column of the Jacobian matrix. Every element in the matrix is assumed to be zero except for the following entries specified.

$$\begin{pmatrix} J(1,1) = 1 \\ J(N,N) = 1 \\ J(i,i-1) = -1 - \frac{1}{2}h\mu u_i \\ J(i,i) = 2 + \frac{1}{2}h\mu u_{i+1} - \frac{1}{2}h\mu u_{i-1} \\ J(i,i+1) = -1 + \frac{1}{2}h\mu u_i \end{pmatrix}$$

for every $i = 2, ..., N-1$ This can then be expanded to

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & ... & 0 \\ 1 - \frac{1}{2}h\mu u_2 & 2 + \frac{1}{2}h\mu u_3 - \frac{1}{2}h\mu u_1 & -1 + \frac{1}{2}h\mu u_2 & 0 & 0 & ... & 0 \\ 0 & 1 - \frac{1}{2}h\mu u_3 & 2 + \frac{1}{2}h\mu u_4 - \frac{1}{2}h\mu u_2 & -1 + \frac{1}{2}h\mu u_3 & 0 & ... & 0 \\ 0 & 0 & 1 - \frac{1}{2}h\mu u_4 & \frac{1}{2}h\mu u_5 - \frac{1}{2}h\mu u_3 & -1 + \frac{1}{2}h\mu u_4 & ... & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & ... & \vdots \\ 0 & 0 & 0 & -1 - \frac{1}{2}h\mu u_{N-1} & 2 + \frac{1}{2}h\mu u_N - \frac{1}{2}h\mu u_{N-2} & -1 + \frac{1}{2}h\mu u_{N-1} & 0 \\ 0 & 0 & 0 & 0 & ... & 0 & 1 \end{pmatrix}$$

Which is the Jacobian of $\vec{F}(\vec{u})$. The following MATLAB code was used to program the Jacobian

```
function J = proj2_3_Jacobian(u,N,h,q_fun)
J = zeros(N,N)
J(1,1) = 1; J(N,N) = 1;

for i = 2:N-1
    a = i-1; b = i+1;
    J(i,a) = (-1 - (1/2)*h*q_fun*u(i));
    J(i,i) = (2 + (1/2)*h*q_fun*u(i+1) - (1/2)*h*q_fun*u(i-1));
    J(i,b) = (-1 + (1/2)*h*q_fun*u(i));
end
end
```

(c)

For this part we used Newton's method and Broyden's method to solve the nonlinear system of the form

$$A\vec{u} = F$$

. The Newton's method was the code provided by the textbook, and the Broyden's was the following code:

```
function [zero,res,niter,B] = Broyden_quasiNewton(fun,B,x0,tol,nmax, varargin)
n = length(x0);
niter = 0; x = x0;
err = tol+1;
F = fun(x, varargin{:});
while (err >= tol && niter < nmax)
    delta = -B\F;
    x = x + delta;
    F = fun(x, varargin{:});
    B = B + F*delta'/(delta'*delta);
    err = norm(delta);
    niter = niter + 1;
end
res = norm(F);
zero = x;
end
```

Method	Number of Iterations	$y(0.5)$
Newton	4	6.420249594878583e-01
Broyden	7	6.420249594878588e-01

The following code was used to determine the value of $y(0.5)$.

```
z = find(abs(x - 0.5) < eps)
u_newt(z)
u_broyd(z)
```

The best initial guess u_0 was determined from a selection of three initial guesses. The first initial guess for $\vec{u}$ was a column vector of length $N+1$ entirely composed of zeros. The next initial guess was a column vector of length $N+1$ with every entry being 1. The third initial guess was constructed by:

$$u_0 = \frac{\beta - \alpha}{b - a}\vec{x}^T + \alpha - \frac{\beta - \alpha}{b - a} \cdot a$$

where $\vec{x}$ is a row vector of 101 linearly spaced points between the two endpoints, a and b, and $y(a) = \alpha$, $y(b) = \beta$ One might note that the same values were obtained for $y(0.5)$ from both methods. The Jacobian was then evaluated with each one of the initial guesses for u. At this point, Newton's method and Broyden's method were then used to solve the system. For all three cases, Broyden's method produced a smaller residual. Thus, Broyden's method is considered to be marginally more accurate than Newton's in this specific problem. The third initial guess produced the smallest residual from Broyden's. Hence the third choice for an initial guess was chosen as such for this problem. The code which shows this process is as follows:

```
a = 0; b = 1;
mu = 1; alpha = 0; beta = 1;
h = 0.01; N = (b-a)/h + 1;
x = linspace(a,b,N);
tol = 10^(-12); nmax = 100;
q_fun = mu;
f_fun = @(x) -exp(x-2*x.^2).*(x.*exp(x).*(-1 - x + 2*x.^2) +...
    exp(x.^2).*(2 - 5*x - 4*x.^2 + 4*x.^3));

u0_1 = zeros(N,1);
u0_2 = ones(N,1);
c1 = (beta - alpha)/(b-a); c2 = alpha - c1*a;
u0_3 = c1*x' + c2;

J1 = proj2_3_Jacobian(u0_1,N,h,q_fun);
```

```
J2 = proj2_3_Jacobian(u0_2,N,h,q_fun);
J3 = proj2_3_Jacobian(u0_3,N,h,q_fun);

[u1, res1, niter1] = Broyden_quasiNewton(@proj2_3_nonlinearF, J1, u0_1, tol, ...
nmax, x, q_fun, f_fun, alpha, beta);
[u2, res2, niter2] = Broyden_quasiNewton(@proj2_3_nonlinearF, J2, u0_2, tol, ...
nmax, x, q_fun, f_fun, alpha, beta);
[u3, res3, niter3] = Broyden_quasiNewton(@proj2_3_nonlinearF, J3, u0_3, tol, ...
nmax, x, q_fun, f_fun, alpha, beta);

u_tot = [u1,u2,u3];
diff1 = max(abs(u3-u1)); diff2 = max(abs(u3-u2));
diff3 = max(abs(u2-u1));
diff_tot = [diff1,diff2,diff3];
res_tot = [res1,res2,res3];
min(res_tot)
```

The minimum of the residual vector was the third intial guess, $u0_3$, for both Newton and Broyden's.

(d)

We would like to see what the approximation to $y(x)$ is where $x = 2/\pi$. However, the function $y(x)$ has only been approximated at the points that are spanned by multiples of h between $x = 0$ and $x = 1$ where $h = 0.010000000000000$. Moreover, since $x = 2/\pi$ is irrational, it can never be exactly realized by a multiple of any value for h. Thus, we must use a method of interpolation to extrapolate what we think the value should be based on the behavior of the surrounding points. To do this, we must construct an interpolating polynomial. This polynomial is a function that describes $y(x)$ in the neighborhood of $y(2/\pi)$. Let $x = 2/\pi$ be x_α. We then construct an interpolating function in the neighborhood of x_α with the points around x_α.

We will be using Horner's method, which is an adaptation to Newton's method to reduce multiplicative operations to addition. We have chosen to use this method since the number of nodes needed is minimal over a small interval, and obtaining the derivatives of $y(x)$ for cubic spline interpolation requires additional steps which are ultimately unnecessary. For a general good approximation without wild oscillations at the endpoints, we will consider the two points preceding x_α and two points succeeding x_α. This further reinforces that any discrepancy from minute oscillations towards the endpoints will not drastically effect our approximation for $y(x_\alpha)$, for it is in the middle of the points.

The procedure is to iterate through all of the points x_i until a value is reached that exceeds our desired x-value, $x = 2/\pi$. The two points before this condition and the two points after this condition will comprise the 4 interpolation points (nodes). The formula for this is as follows

$$p(x) = c_0(x - x_0) + c_1(x - x_0)(x - x_1) + c_2(x - x_0)(x - x_1)(x - x_2) + c_3(x - x_0)(x - x_1)(x - x_2)(x - x_3)$$

where x_0, x_1, x_2, x_3 are our four nodes surrounding $x = 2/\pi$ and c_i are constant coefficients determined by

$$c_k = \frac{y_k - p_{k-1}(x_k)}{\prod_{i=0}^{k-1}(x_k - x_i)}$$

We then obtain that

$$c_1 = 0.784728487247701, c_2 = 1.066580654465465, c_3 = -1.096778633837835, c_4 = -1.001305301377238$$

Which now gives us all the information we need to construct the polynomial $p(x)$ given above. After evaluating this polynomial with Horner's method, we obtain that

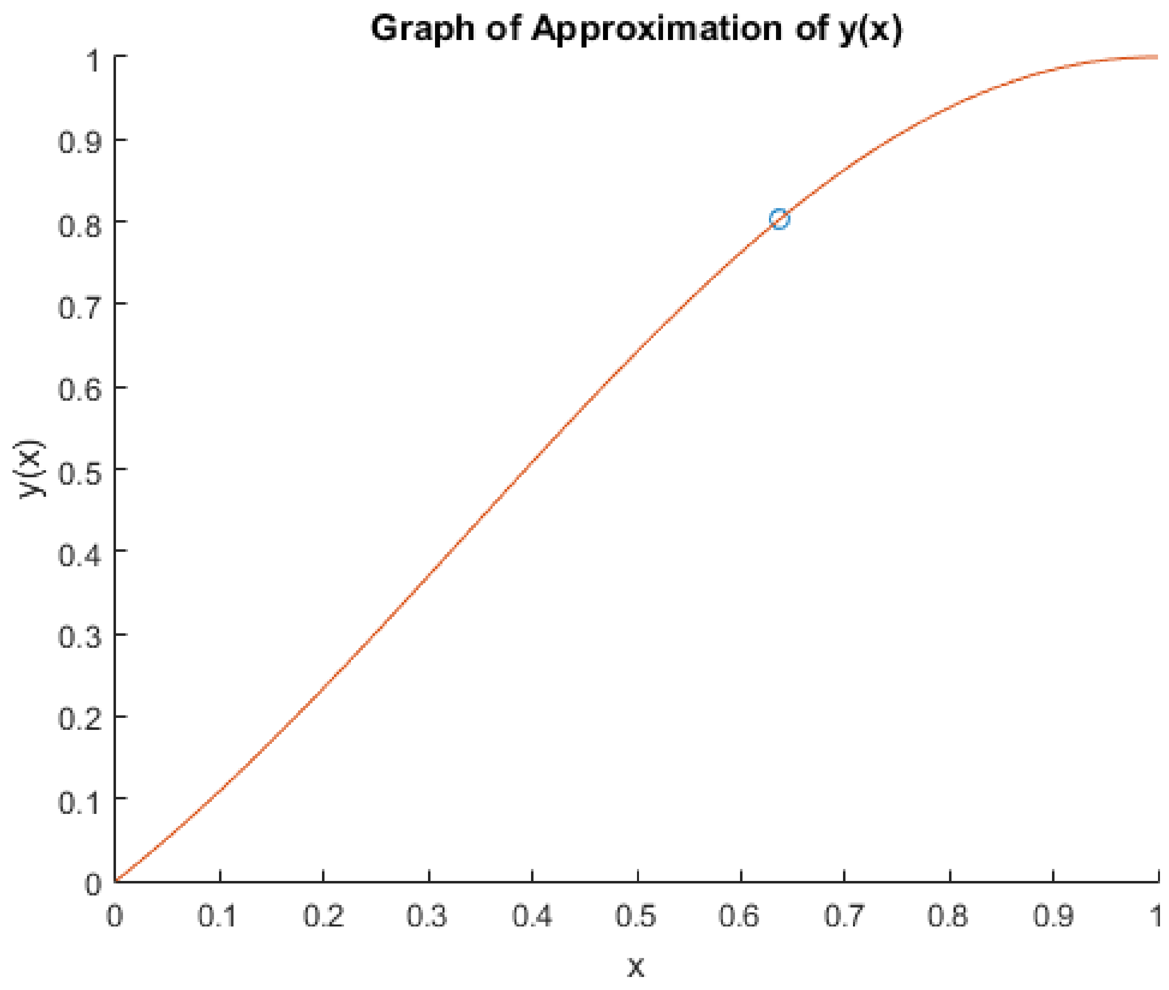
$$\boxed{y(\tfrac{2}{\pi}) \approx 8.023345207026221e - 01}$$

The following code was used to prepare the interpolation:

```
xx = 2/pi;
for i = 1:length(x) - 1
```

```
    if (xx >= x(i)) && (xx <= x(i+1))
    I = i; break;
end
end
x_int = x(I-1:I+2); y_int = u_broyd(I-1:I+2, 1);
coef_x = int_ntcoef(x_int, y_int);
y_2_pi = int_poly_eval_horner(xx, x_int, coef_x)
```

The following is a plot of the approximate solution, $u(x)$, and the specific point $\left(\frac{2}{\pi}, u(\frac{2}{\pi})\right)$.



4. a. Derive the ODE system based on using the finite difference to approximate the x partial derivatives for this IBVP on a set of equispaced nodes:

$$a = x_0 < x_1 < ... < x_{N_x} < x_{N_{x+1}}$$

$$h = x_i - x_{i-1}, i = 1, 2, ..., N_x + 1$$

Then put these ODEs together in the form

$$\vec{u}'(t) = \vec{F}(t, \vec{u})$$

Solution:

This problem will be solved using the finite difference scheme. We will use the following for our finite difference scheme:

$$\frac{\partial^2 y}{\partial x^2} \approx \frac{u(i-1) - 2u(i) + u(i+1))}{h^2}$$
$$\frac{\partial y}{\partial x} \approx \frac{u(i+1) - u(i-1)}{2h}$$
$$\frac{dy}{dt} = \frac{d^2y}{dx^2} - q(x, y(t))\frac{dy}{dx} + f(, t, y(x,t))$$

Then, we can conclude that:

$$\frac{dy}{dt} \approx \partial x^2 \approx \frac{u(i-1) - 2u(i) + u(i+1))}{h^2} - q(x, y(t))\frac{u(i+1) - u(i-1)}{2h} + f(x, t, y(x,t))$$

At the end points, we must impose the boundary conditions. That is, when $t = 0$:

$$\frac{dy}{dt} \approx \partial x^2 \approx \frac{\alpha(t) - 2u(i) + u(i+1))}{h^2} - q(x, y(t))\frac{u(i+1) - \alpha(t)}{2h} + f(x, t, y(x,t))$$

and when $t = 5$:

$$\frac{dy}{dt} \approx \partial x^2 \approx \frac{u(i-1) - 2u(i) + \beta(t))}{h^2} - q(x, y(t))\frac{\beta(t) - u(i-1)}{2h} + f(x, t, y(x,t))$$

b. Follow the Matlab funtion

```
heat_eq_nonlinear_ode_sys
```

in lecture slides to implement a MATLAB function for the ODE system above such that it has the following interface:

```
 function F = heat_eq_nonlinear_ode_sys_Burgers(t, u, ... x, mu, q_fun, f_fun, alpha_fun, beta_fun)
```

Solution:
This problem tests our knowledge of implementing systems of partial differential equations in MATLAB. The syntax is quite easy. Our MATLAB function will take as inputs: t, the time interval, u, the variable of interest, x, the x interval of interest, and the necessary

```
 mu, q_fun, f_fun
```

in order to implement the system, as well as

```
alpha_fun and beta_fun
```

the respective boundary conditions for our system of differential equations. Following the lecture slides, the first step was to create the xx variable, the NN variable, and the N variable. These values will be used in the forloop to create the system. After this, space is allocated for F, the system of differential equations. F(1) will represent the boundary conditions that we showed above, and will be programmed as such. After this, a forloop is used from 2 to N-1 in order to create the representative discretized differential equations at each point of our x discretization. Lastly, F(N) will represent the boundary conditions related to the

```
 beta_fun
```

input.
Once we are finished, the code is as follows:

```
function F = heat_eq_nonlinear_ode_sys_Burgers(t, u, ...
x, mu, q_fun, f_fun, alpha_fun, beta_fun)
NN = length(x); N = NN - 2; h = x(2) - x(1);
xx = x(2:end-1); F = zeros(N, 1);
F(1)=mu*(alpha_fun(t)-2*u(1)+u(2))/(h^2)-q_fun(xx(1),u(1))*((-alpha_fun(t)+u(2))/(2*h)) ...
    +f_fun(xx(1),t,u(1));
for i=2:N-1
```

```
    F(i)=mu*(u(i-1)-2*u(i)+u(i+1))/(h^2)-q_fun(xx(i),u(i))*((-u(i-1)+u(i+1))/(2*h)) ...
        +f_fun(xx(i),t,u(i));
end
F(N)=mu*(u(N-1)-2*u(N)+beta_fun(t))/(h^2)-q_fun(xx(N),u(N))*((-u(N-1)+beta_fun(t))/(2*h))
    +f_fun(xx(N),t,u(N));
return;
```

c. Use the Crank Nicolson method together with the function we just wrote in order to implement the IBVP with h=.01. Present your results in the table.

Solution: The Crank-Nicolson method gave the following results:

$y(2.51, 0.75)$	$\frac{\partial y(2.51,0.75)}{\partial t}$
0.429632447364680	-0.902994411072595

In order to generate these results, a timestep of 1.e-4 was used. We had to mess with the step size a bit in order to find a step size that did not give poorly scaled matrices in the Crank-Nicolson method. Once we found this timestep, we were able to trust our numerical results, as the output no longer contained mostly zeroes or NaNs.
Note that the order of the problem will now become

$$O(0.01^2) + O(0.0001^2) = O(h^2) + O(\gamma^2)$$

d.
Use the numerical solution generated above to find an approximation to $y(3/\pi, \pi)$. The Horner method of interpolation was chosen again this application due to the lack of a need for more post processing for derivative values. This is noticeably more work since we are now working with a matrix of points since our solution has been fully discretized. The solution obtained was:

$$\boxed{y(3/\pi, \pi) \approx -4.317987263564369e-02}$$

The following code was used:

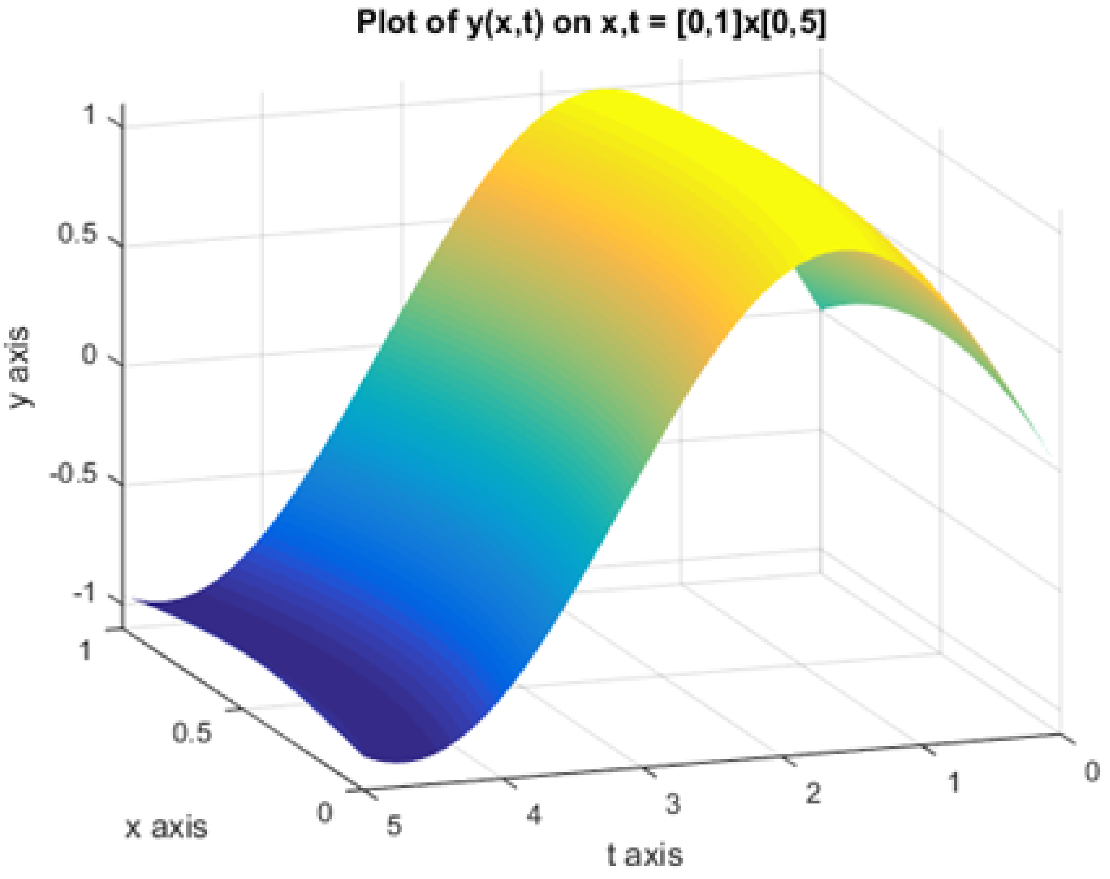
```
xx = 3/pi;
for i = 1:length(x) - 1
    if (xx >= x(i)) && (xx <= x(i+1))
    I_x = i; break;
end
end
tt = pi;
for i = 1:length(t) - 1
    if (tt >= t(i)) && (tt <= t(i+1))
    I_t = i; break;
end
end

x_int = x(I_x-1:I_x+2); y_int = u(I_t-1:I_t+2, I_x-1:I_x+2);
coef_x1 = int_ntcoef(x_int, y_int(:,1));
coef_x2 = int_ntcoef(x_int, y_int(:,2));
coef_x3 = int_ntcoef(x_int, y_int(:,3));
coef_x4 = int_ntcoef(x_int, y_int(:,4));
y1 = int_poly_eval_horner(xx, x_int, coef_x1)
y2 = int_poly_eval_horner(xx, x_int, coef_x2)
y3 = int_poly_eval_horner(xx, x_int, coef_x3)
y4 = int_poly_eval_horner(xx, x_int, coef_x4)
ytot = [y1 y2 y3 y4]
coef_tot = int_ntcoef(x_int,ytot)
y_spec = int_poly_eval_horner(xx, x_int, coef_tot)
```

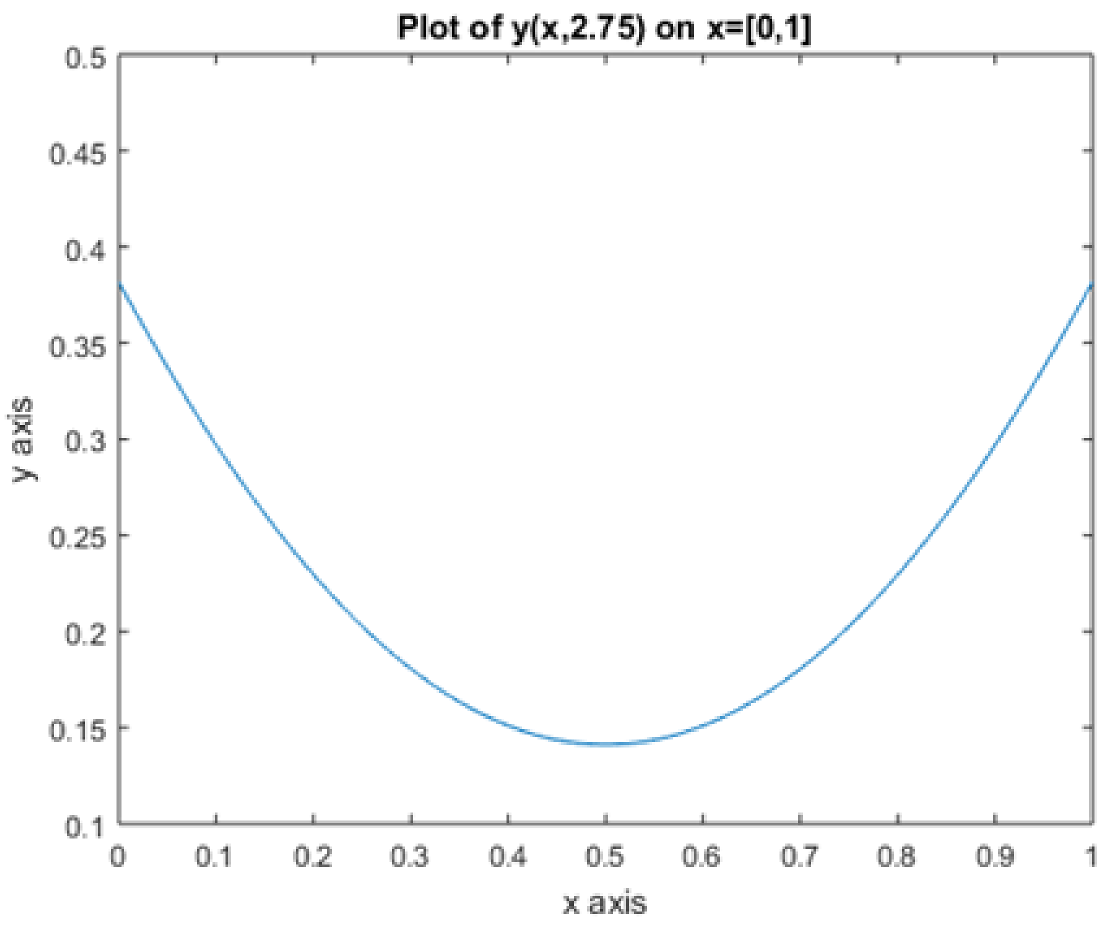
e. Make a plot of $y(x,t)$ for $(x,t)\epsilon[0,1]x[0,5]$. Scale your plot by the following MATLAB commands:

axis([0,1,0,5,-1.1,1.1])view(-109,18)

Solution:
Our plot (scaled accordingly) is as follows:



f. Make a plot of $y(x,2.75)$ for $x\epsilon[0,1]$. Scale your plot by axis([0,1,0.2,0.5])
Solution:
Our plot (scaled accordingly) is as follows:



The following code was used: (i)

`ODE_CN_Broyden`

(given)

```
function [t, u] = ode_CN_Broyden(odefun, tspan, y0, Nh, tol, nmax, varargin)
bm1 = 1/2; b0 = 1/2;
t = linspace(tspan(1),tspan(2),Nh+1); tau = t(2) - t(1);
u = zeros(Nh + 1, length(y0)); u(1,:) = y0;
B0 = eye(length(y0), length(y0));
for n = 1:Nh
vn = u(n,:)' + tau*b0*odefun(t(n), u(n,:)', varargin{:});
G = @(z) z - tau*bm1*odefun(t(n+1), z, varargin{:}) - vn;
z0 = u(n,:)' + tau*odefun(t(n), u(n,:)', varargin{:}); % better initial guess?
[z, res, niter, B0] = Broyden_TL(G, B0, z0, tol, nmax);
u(n+1, :) = z';
end
t = t'; % to make t a column vector
return;
```

(ii) BroydenTL

```
function [zero,res,niter,B] = Broyden_TL(fun,B,x0,tol,nmax, varargin)
n = length(x0);
niter = 0; x = x0;
err = tol+1;
F = fun(x, varargin{:});
while (err >= tol & niter < nmax)
    delta = -B\F;
    x = x + delta;
    F = fun(x, varargin{:});
    %if ~any(F)
     %   break
    %end
    B = B + F*delta'/(delta'*delta);
    err = norm(delta);
    niter = niter + 1;
end
res = norm(fun(x,varargin{:}));
zero = x;
end
```

(iii) Overall script (for plots, calculation)

```
clear
format long
a = 0; b=1; domain=[a,b];
mu=1;
y0_fun = @(x) sin(x-(x.^2));
alpha_fun = @(t) sin(t);
beta_fun = @(t) sin(t);
q_fun = @(x,y) y;
f_fun = @(x,t,y) 3*cos(t+(x.*(1-x)))-(((2*x)-1).*(1-(2*x)+cos(t+(x.*(1-x)))).*y);
h = 0.01;
Nh = (b-a)/h - 1; x = linspace(a, b, Nh+2);
u0 = y0_fun(x(2:end-1)');
tspan = [0,5];
Nt = 10000; tol = 10^(-7); nmax = 500;
[t,u] = ode_CN_Broyden(@heat_eq_nonlinear_ode_sys_Burgers, tspan, u0, Nt, ...
tol, nmax, x, mu, q_fun, f_fun, alpha_fun, beta_fun);
u = [alpha_fun(t), u, beta_fun(t)];

n_skip = 1; t_p = t(1:n_skip:end); u_p = u(1:n_skip:end, :);
surf(x, t_p, u_p); shading INTERP
xlabel('x axis');
ylabel('t axis');
zlabel('y axis');
title('Plot of y(x,t) on x,t = [0,1]x[0,5]')
axis([0, 1, 0, 5, -1.1, 1.1])
view(-109, 18)

yp=u(5021,76);
yp1=u(5021,75);
yp2=u(5021,77);
dy=(yp1-2*yp+yp2)/(h^2) - yp*(yp2-yp1)/(2*h) + f_fun(.75,2.51,yp)
```

```
e=find(t==2.75);
figure;
plot(x,u(e,:))
title('Plot of y(x,2.75) on x=[0,1]');
xlabel('x axis');
ylabel('y axis');
axis([0, 1, 0.1, 0.5]);
```