```
% MATH 3036 COURSEWORK 1
% STUDENT NAME: JAKE DENTON
```

Question 1 File Checklist

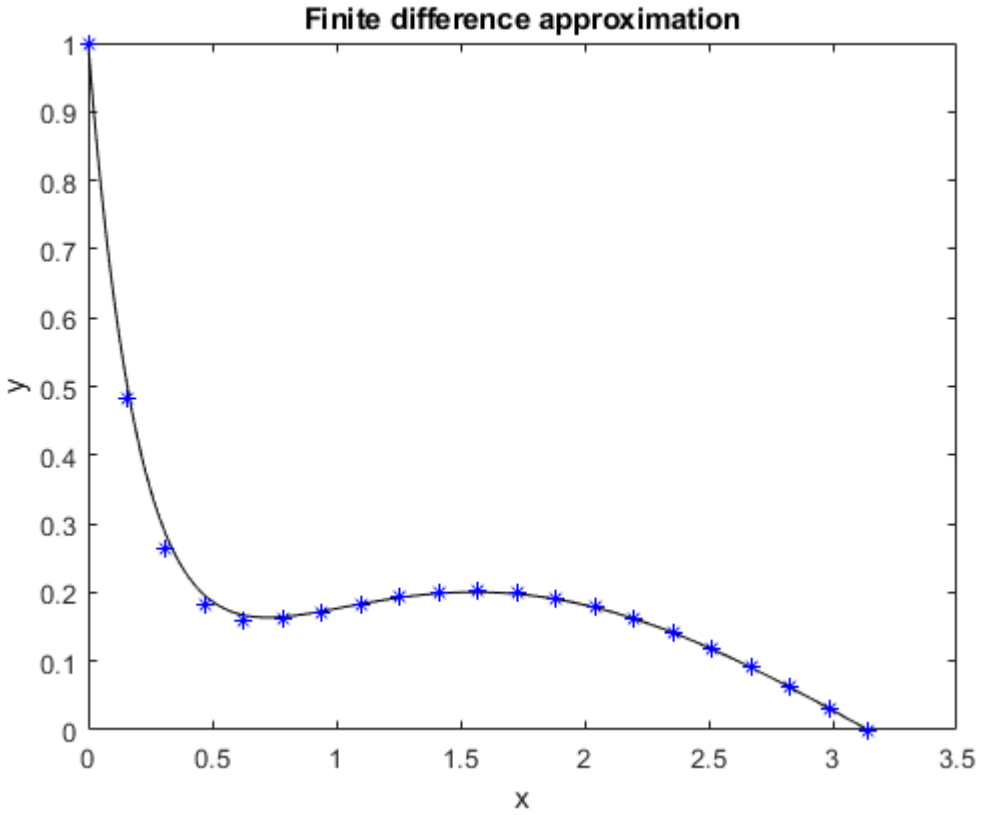
See the files finite_difference.m, shooting_forward_euler.m and shooting_backward_euler.m. Coursework1Script.m calls the functions to produce the required outputs

Question 1a)

This code section calls finite_difference.m to produce a vector of approximations y and the grid-function 2-norm e (which is displayed) along with a plot of the approximations with the exact solution.

```
[yFD,eFD]=finite_difference(0.2,20,1);
disp(['The value of the grid function 2-norm of the error is ',num2str(eFD)]); %display this as required in the question
```

```
The value of the grid function 2-norm of the error is 0.012453
```

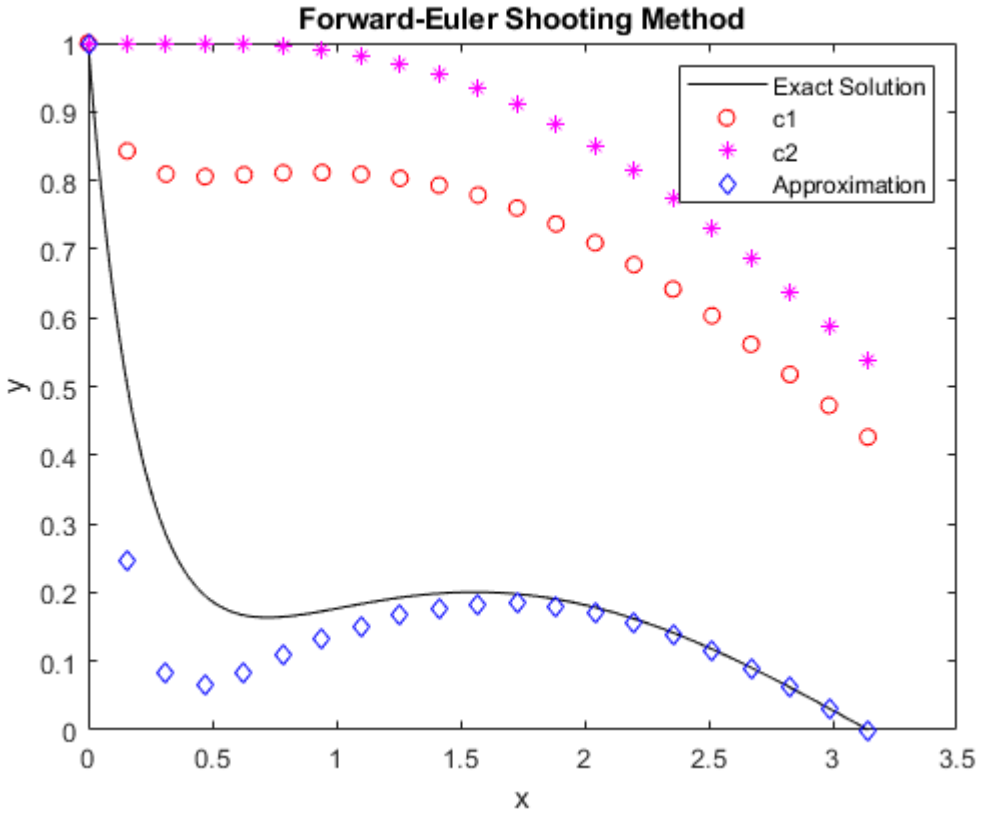


Question 1b)

This code section calls shooting_forward_euler.m to produce a vector of approximations y and the grid function 2-norm e. e is displayed along with the coefficients used in the linear superposition alpha1 and alpha2 and there is a plot showing the approximations with c1=-1,c2=0, the linear superposition approximation y and the exact solution.

```
[ySMF,eSMF]=shooting_forward_euler(0.2,20,-1,0,1);
disp(['The value of the grid function 2-norm of the error is ',num2str(eSMF)]); %display this as required in the question
```

```
The values of alpha1 and alpha2 respectively are 4.8061 and -3.8061
The value of the grid function 2-norm of the error is 0.14696
```

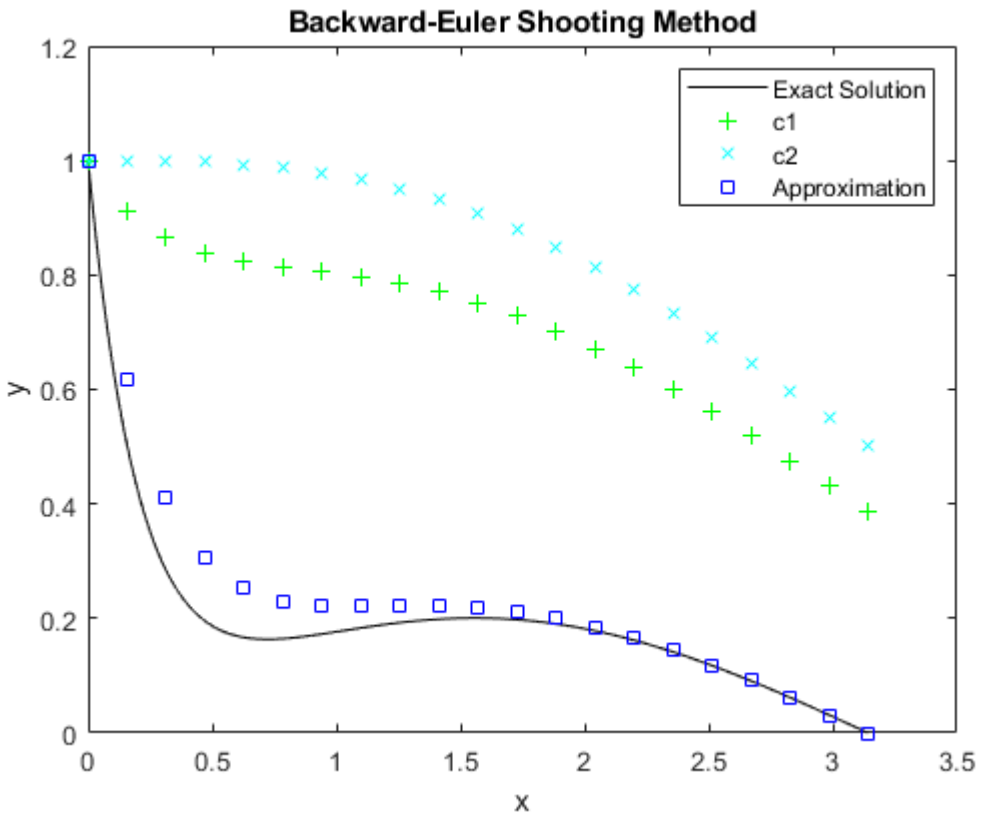


Question 1c)

This code section calls shooting_backward_euler.m to produce a vector of approximations y and the error e. The error is displayed along with the coefficients in the linear superposition and a plot is produced. Below is an image showing the equations used in a single step for this method, the first is a matrix equation used in the function to compute the approximations, and the other equations show the method explicitly.

```
[ySMB,eSMB]=shooting_backward_euler(0.2,20,-1,0,1);
disp(['The value of the grid function 2-norm of the error is ',num2str(eSMB)]); %display this as required in the question
```

```
The values of alpha1 and alpha2 respectively are 4.3904 and -3.3904
The value of the grid function 2-norm of the error is 0.097103
```




```
img=imread('Equation1c.png'); %This section needs to be removed before this script can be
checked
image(img);
axis off;
```

$$\begin{pmatrix} 1 & -h \\ h & 1+h/\varepsilon \end{pmatrix} \begin{pmatrix} y_{n+1} \\ v_{n+1} \end{pmatrix} = \begin{pmatrix} y_n \\ v_n + h\cos x_{n+1} \end{pmatrix}$$

$$y_{n+1} = y_n + h v_{n+1}$$

$$v_{n+1} = v_n + h\left(\cos x_{n+1} - y_{n+1} - \frac{v_{n+1}}{\varepsilon}\right)$$

Question 1d) Finite-Difference

The first table presents the error and error ratio for the finite difference method for step length h that is decreased by a factor of 2 each time. The error ratio shows that the error decreases by a factor of 4 with each halving of h. As a result, the finite difference method has second-order accuracy, as expected since the centred-difference methods used are second-order accurate.

```
h=[pi/20,pi/40,pi/80,pi/160]';
errorFD=zeros(4,1);
N=20;
for i=1:4
    [yNfd,eNfd]=finite_difference(0.2,N,0);
    N=2*N;
    errorFD(i)=eNfd;
end
ratioFD=zeros(4,1);
ratioFD(1)=1;
for j=2:4
    ratioFD(j)=errorFD(j-1)/errorFD(j);
end
Tfd=table(h,errorFD,ratioFD);
Tfd.Properties.VariableNames={'h','FiniteDiffError','FiniteDiffErrorRatio'};
disp(Tfd);
```

h	FiniteDiffError	FiniteDiffErrorRatio
0.15708	0.012453	1
0.07854	0.0030086	4.1391
0.03927	0.00074484	4.0392

```
    0.019635       0.00018574              4.0101
```

Question 1d) Forward-Shooting Method

The second table presents the error and error ratio for the shooting method where the forward-euler method is used to approximate points. The error ratio shows that the error decreases by a factor of 2 with each halving of h. As a result, this method is first-order accurate, as expected since the forward-euler method is first-order accurate.

```
errorFSM=zeros(4,1);
N=20;
for i=1:4
    [yNfsm,eNfsm]=shooting_forward_euler(0.2,N,-1,0,0);
    N=2*N;
    errorFSM(i)=eNfsm;
end
ratioFSM=zeros(4,1);
ratioFSM(1)=1;
for j=2:4
    ratioFSM(j)=errorFSM(j-1)/errorFSM(j);
end
Tfsm=table(h,errorFSM,ratioFSM);
Tfsm.Properties.VariableNames={'h','ForwardShootError','ForwardShootErrorRatio'};
disp(Tfsm);
```

h	ForwardShootError	ForwardShootErrorRatio
0.15708	0.14696	1
0.07854	0.063309	2.3212
0.03927	0.029876	2.1191
0.019635	0.014557	2.0523

Question 1d) Backward-Shooting Method

The third table presents the error and error ratio for the shooting method where the backward-euler method is used to approximate points. The error ratio approaches the value 2 as h is halved, so this method is first-order accurate. It's useful to note that the error is less than the forward-euler shooting method for all the step lengths in the table, but this difference gets smaller and smaller as the step length is halved.

```
errorBSM=zeros(4,1);
N=20;
for i=1:4
    [yNbsm,eNbsm]=shooting_backward_euler(0.2,N,-1,0,0);
    N=2*N;
    errorBSM(i)=eNbsm;
end
ratioBSM=zeros(4,1);
ratioBSM(1)=1;
for j=2:4
    ratioBSM(j)=errorBSM(j-1)/errorBSM(j);
end
Tbsm=table(h,errorBSM,ratioBSM);
```

```
Tbsm.Properties.VariableNames={'h','BackwardShootError','BackwardShootErrorRatio'};
disp(Tbsm);
```

h	BackwardShootError	BackwardShootErrorRatio
0.15708	0.097103	1
0.07854	0.052251	1.8584
0.03927	0.027194	1.9215
0.019635	0.013892	1.9576

Question 1d) Changing The Value Of Epsilon

When epsilon is set equal to 0.01, the coefficient of y' in the BVP is 100. Since this is so large, it leads to a massive variation in the true solution and consequently has an effect on some of the numerical methods. This is an example of a stiff system of equations. It is clear that the finite difference method is slightly less accurate than for the previous epsilon but the method gives stable approximations regardless. On the other hand, the forward-shooting method has enormous error for h=pi/40,pi/80 and so is very unstable. Finally, the backward-shooting method is very stable (as expected since the backward-Euler method is A-stable) and gives accurate approximtions for all the step lengths.

```
StiffErrorFD=zeros(4,1);
StiffErrorFSM=zeros(4,1);
StiffErrorBSM=zeros(4,1);
N=20;
for i=1:4
    [~,StiffeFD]=finite_difference(0.01,N,0);
    [~,StiffeFSM]=shooting_forward_euler(0.01,N,-1,0,0);
    [~,StiffeBSM]=shooting_backward_euler(0.01,N,-1,0,0);
    StiffErrorFD(i)=StiffeFD;
    StiffErrorFSM(i)=StiffeFSM;
    StiffErrorBSM(i)=StiffeBSM;
    N=2*N;
end
TStiff=table(h,StiffErrorFD,StiffErrorFSM,StiffErrorBSM);
TStiff.Properties.VariableNames={'h','FiniteDiffError','ForwardShootError','BackwardShootErro
r'};
disp(TStiff);
```

h	FiniteDiffError	ForwardShootError	BackwardShootError
0.15708	0.49068	1.7444	0.024437
0.07854	0.20731	4.8521e+09	0.031985
0.03927	0.071842	7.8955e+12	0.03733
0.019635	0.018591	0.50938	0.031101

Question 1d) Order of preference for different tolerances

(i) All the methods investigated achieve a tolerance of below 0.1 either with h=pi/20 or h=pi/40. Since this tolerance for the error is reasonably high, it makes sense to consider both the stability and the computational time required. The backward-Euler shooting method is the most stable, and achieves this tolerance with the largest step length in the table when epsilon is 0.2, so this would be

my preferred method. The forward-Euler shooting method needs a smaller step length to achieve this tolerance, but it is computationally quick so this comes next. Finally, the finite-difference method solves a large system of equations so it is the most computationally expensive. However, any of these methods would be useful for this tolerance. (ii) For a tolerance of 0.001, a more accurate method is required. It is for this reason that the finite-difference method should be used first, as it has the highest order of accuracy. This is followed by the backward-Euler method, which achieves this tolerance for epsilon equal to 0.2 when h=pi/180 and is the most stable method. The forward-Euler shooting method has lower accuracy (needs smaller step-length than pi/180 when epsilon=0.2)and is the most numerically unstable so this comes last.

Question 2 File Checklist

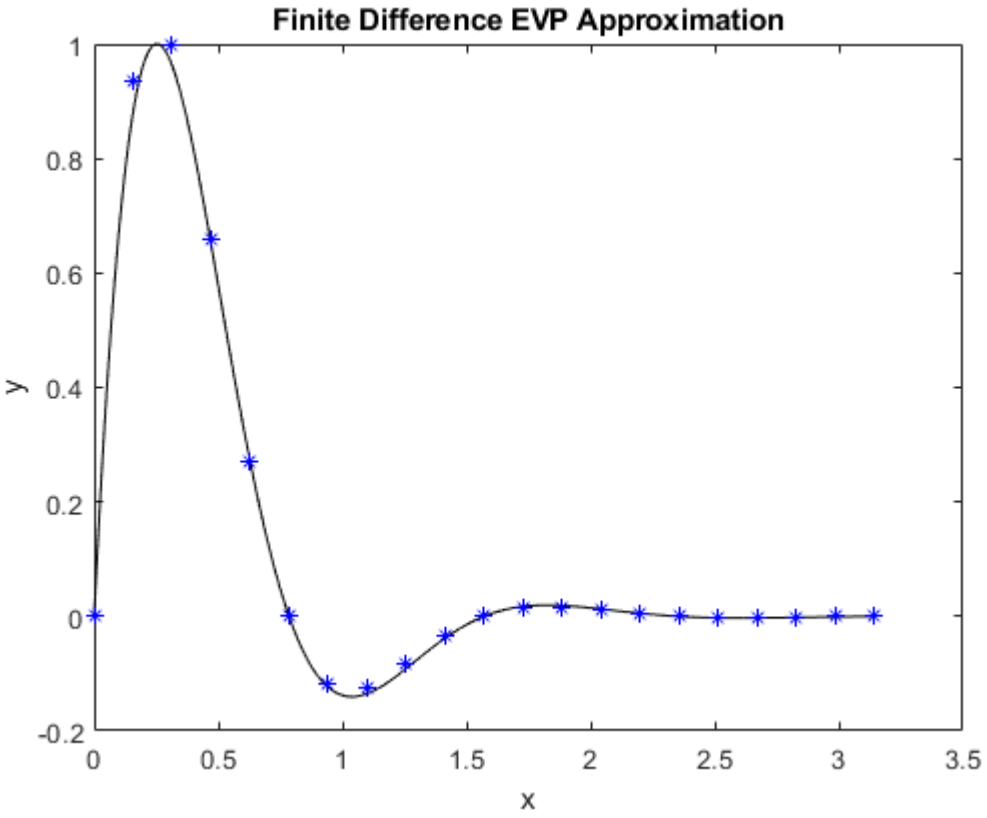
See the files finite_difference_evp.m, shooting_evp_newton.m and Coursework1Script.m (which calls the functions to produce the required outputs).

Question 2a)

This code calls the finite_difference_evp.m function. It produces a graph showing the approximations as blue asterisks along with the true solution, the approximate eigenvalue and its error are displayed along with the grid-function 2-norm of the error.

```
[eigApproxFD,eigErrorFD,~,eEvpFD]=finite_difference_evp(0.2,20,4,1);
disp(['The approximate eigenvalue is ',num2str(eigApproxFD),' with error
',num2str(eigErrorFD)]);
disp(['The grid function 2-norm error in the eigenfunction approximation is
',num2str(eEvpFD)]);
```

```
The approximate eigenvalue is -19.7485 with error 1.5015
The grid function 2-norm error in the eigenfunction approximation is 0.014553
```

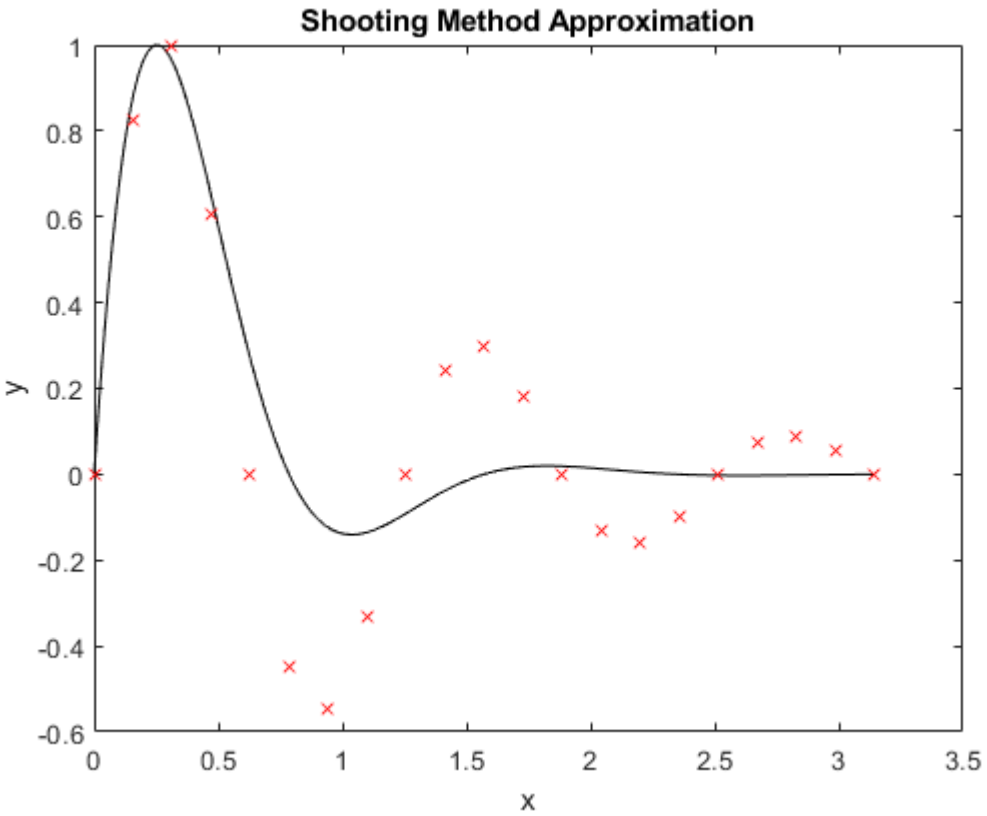


Question 2b)

This code calls the shooting_evp_newton.m function. It produces a graph showing the approximations as red crosses along with the true solution. The approximate eigenvalue is displayed for each step of the iteration, and the error after convergence is displayed with the error in the eigenfunction approximation. The approximations plotted oscillate widely around the true eigenfunction, this is especially clear near the local minimum at x approximately 1. The oscillations settle with decreasing amplitude nearer the endpoint of the interval.

```
[e,eigError,ylamb]=shooting_evp_newton(0.2,20,4,-20,1);
disp(['The error in the eigenvalue after convergence is ',num2str(eigError)]);
disp(['The grid-function 2-norm of the error for the final approximation is ',num2str(e)]);
```

```
Iteration:1
Approximate Eigenvalue:-20.20889556
Iteration:2
Approximate Eigenvalue:-20.19751951
Iteration:3
Approximate Eigenvalue:-20.19748484
Iteration:4
Approximate Eigenvalue:-20.19748484
The error in the eigenvalue after convergence is 1.0525
The grid-function 2-norm of the error for the final approximation is 0.35017
```



Question 2c) Finite Difference Order Of Accuracy

The table below shows that when the number of sub-intervals is doubled, the error in the approximation of the eigenfunction decreases by a factor of 4. Considering the error in the

eigenvalue approximations, it can be seen that error also decreases by a factor of 4 for every halving of the step length. Therefore, this numerical method is second-order accurate.

```
FdEigError=zeros(5,1);
FdYError=zeros(5,1);
h=zeros(5,1);
N=20;
for i=1:5
    [~,eigErrorFD,~,eEVPFD]=finite_difference_evp(0.2,N,4,0);
    h(i)=pi/N;
    N=2*N;
    FdEigError(i)=eigErrorFD;
    FdYError(i)=eEVPFD;
end
ratioFdEvp=zeros(5,1);
ratioFdEvp(1)=1; %this is just so the vector is the same length as the above for the table
for j=2:5
    ratioFdEvp(j)=FdYError(j-1)/FdYError(j);
end
TFdEvp=table(h,FdEigError,FdYError,ratioFdEvp);
TFdEvp.Properties.VariableNames={'h','ErrorInEigenFD','ErrorInYFD','FDYErrorRatio'};
disp(TFdEvp);
```

h	ErrorInEigenFD	ErrorInYFD	FDYErrorRatio
0.15708	1.5015	0.014553	1
0.07854	0.37864	0.0036962	3.9374
0.03927	0.094873	0.00091425	4.0429
0.019635	0.023732	0.00021872	4.18
0.0098175	0.0059338	5.4641e-05	4.0028

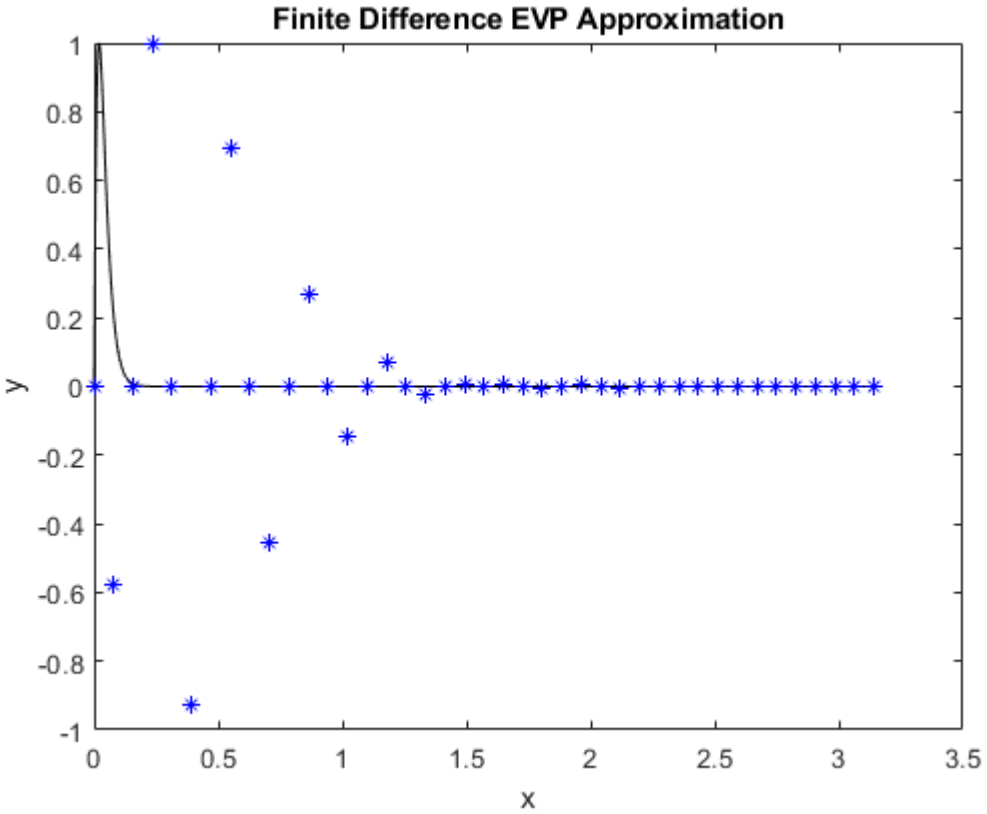
Question 2c) Reducing Epsilon For Finite Difference

Reducing epsilon to 0.01 increases the magnitude of the exact eigenvalues by a large amount. It also increases the magnitude of the off-diagonal elements b and c in the tridiagonal matrix A. I produced a table first which showed that the eigenvectors had imaginary parts, so within the function I restricted the calculation of the error and the plot to the real part, which you can see in the table below. The true eigenfunction under this value of epsilon goes to zero very quickly as shown alongside the N=40 approximations below. The approximation has instability for low x values which eventually evens out around zero for x>pi/2. When higher numbers of sub-intervals are used, the approximations reach zero much more quickly, but at the start of the interval there is massive oscillatory behaviour which consequently stops any convergence in the error to zero. The eigenvalue approximations diverge from the true eigenvalue, as shown in the table the error seems to decrease to begin with but then get very far from the true value, showing that the approximations using this method on a stiff system (low epsilon) are unstable and not accurate.

```
FdEigError2=zeros(8,1);
FdYError2=zeros(8,1);
h2=zeros(8,1);
N2=20;
for i=1:8
    [~,eigErrorFD,~,eEVPFD2]=finite_difference_evp(0.01,N2,4,0);
```

```
    h2(i)=pi/N2;
    N2=2*N2;
    FdEigError2(i)=eigErrorFD;
    FdYError2(i)=eEVPFD2;
end

TFdEvp2=table(h2,FdEigError2,FdYError2);
TFdEvp2.Properties.VariableNames={'h','EigenError','ErrorInY'};
disp(TFdEvp2);
[~,~,~,~]=finite_difference_evp(0.01,40,4,1);
```

h	EigenError	ErrorInY
0.15708	2434.9	0.74762
0.07854	2191.8	0.63536
0.03927	1219.1	0.28477
0.019635	3726.1	0.14769
0.0098175	36293	0.27189
0.0049087	1.6097e+05	0.27475
0.0024544	6.5905e+05	0.27202
0.0012272	2.6515e+06	0.27611



Question 2c) Shooting Method Order Of Accuracy

The table below shows that when the number of sub-intervals is doubled, the error in the eigenvalue approximation and the 2-norm error in the eigenfunction approximation halves. As a result, this method is first-order accurate.

```
SmEigError=zeros(5,1);
SmYError=zeros(5,1);
h3=zeros(5,1);
N3=80;
for i=1:5
    [eEVPSM,eigErrorSM,~]=shooting_evp_newton(0.2,N3,4,-20,0);
    h3(i)=pi/N3;
    N3=2*N3;
    SmEigError(i)=eigErrorSM;
    SmYError(i)=eEVPSM;
end
ratioSmEvp=zeros(5,1);
ratioSmEvp(1)=1; %this first value can be ignored
for j=2:5
    ratioSmEvp(j)=SmYError(j-1)/SmYError(j);
end
TSmEvp=table(h3,SmEigError,SmYError,ratioSmEvp);
TSmEvp.Properties.VariableNames={'h','SMErrorInEigen','SMErrorInY','SMYErrorRatio'};
disp(TSmEvp);
```

h	SMErrorInEigen	SMErrorInY	SMYErrorRatio
0.03927	2.7703	0.020707	1
0.019635	1.4725	0.010454	1.9807
0.0098175	0.76009	0.0052167	2.004
0.0049087	0.38628	0.0026061	2.0017
0.0024544	0.19473	0.0013082	1.9922

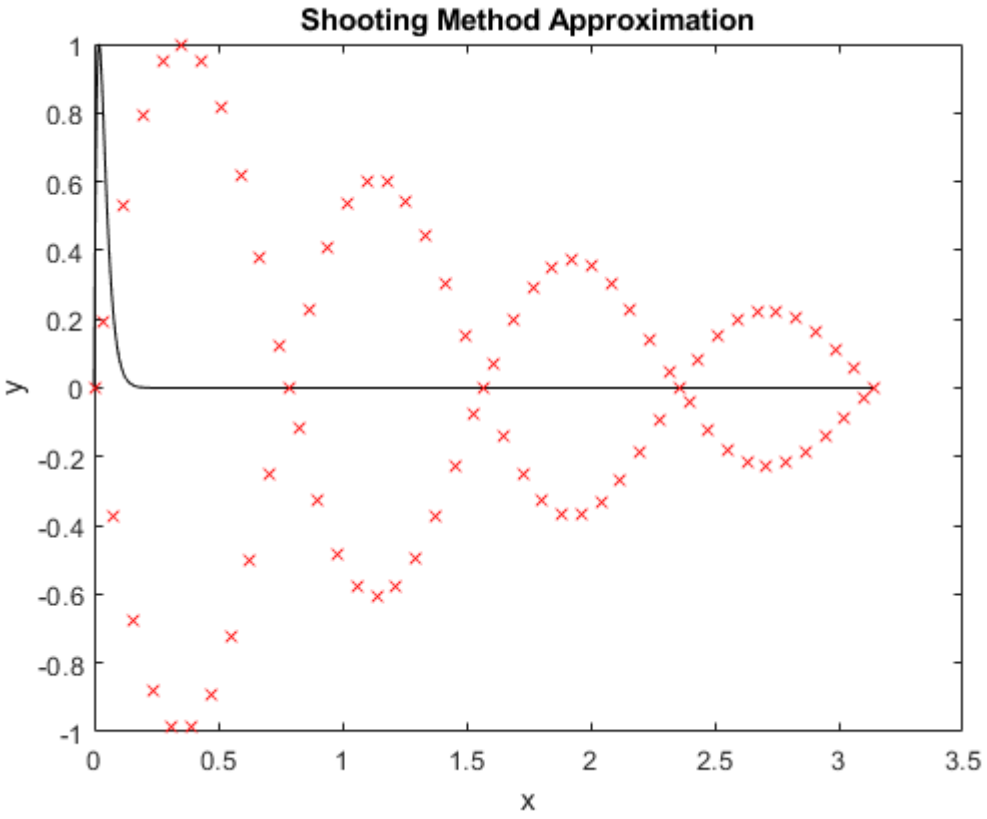
Question 2c) Reducing Epsilon For Shooting Method

Behaviour differs as epsilon is reduced as the reciprocal of epsilon grows larger. This in turn affects the system of equations in the forward Euler method leading to more sub-interval-sensitive approximations of the eigenfunction, as shown by the graph below with N=80. Another consequence of the change in epsilon is that the magnitude of the eigenvalues increases. For the lower n eigenvalues, the magnitude of 1/4*eps^2 far exceeds that of n^2, so to obtain decent approximations we are forced to try and be very accurate with our initial guess. The first table shows the effect that a change in the initial guess from -2517 to -2518 has on the error in the fourth eigenvalue approximation, and also shows that the first order accuracy is lost as including more points doesn't improve the estimate. This relatively small change in the initial guess leads to different eigenvalues being approximated altogether. On the other hand, for guesses greater than -2500, the n=1 eigenvalue is stable and the order of accuracy is preserved as shown in the second table which takes initial guesses of -2000 and -2450, so we can approximate this specific eigenvalue reliably. The final table considers approximating the 200th eigenvalue which has the exact value -42499. An initial approximation of -42498.5 (only 0.5 away) still leads to quite large errors even when many points are used. This shows that only the first eigenvalue can be approximated accurately, and even initial guesses very near to other eigenvalues tend to lead to lead to estimates with a lot of error which cannot be improved upon by considering more points. One possible explanation for the instability is that the eigenvalues are not in the stability region of the forward-Euler method, so a possible modification could be replacing this method with the backward-Euler method, which is A-stable (it's stability region is the left half of the complex plane).

```
[~,~,~]=shooting_evp_newton(0.01,80,4,-2516,1);
SubIntervals=[320,640,1280,2560]';
[~,eigErrorSm1,~]=shooting_evp_newton(0.01,320,4,-2518,0);
[~,eigErrorSm2,~]=shooting_evp_newton(0.01,320,4,-2517,0);
[~,eigErrorSm3,~]=shooting_evp_newton(0.01,640,4,-2518,0);
[~,eigErrorSm4,~]=shooting_evp_newton(0.01,640,4,-2517,0);
[~,eigErrorSm5,~]=shooting_evp_newton(0.01,1280,4,-2518,0);
[~,eigErrorSm6,~]=shooting_evp_newton(0.01,1280,4,-2517,0);
[~,eigErrorSm7,~]=shooting_evp_newton(0.01,2560,4,-2518,0);
[~,eigErrorSm8,~]=shooting_evp_newton(0.01,2560,4,-2517,0);
lambzero1=[eigErrorSm1,eigErrorSm3,eigErrorSm5,eigErrorSm7]';
lambzero2=[eigErrorSm2,eigErrorSm4,eigErrorSm6,eigErrorSm8]';
Tlambzero=table(SubIntervals,lambzero1,lambzero2);
Tlambzero.Properties.VariableNames={'N','LambdaGuess2518','LambdaGuess2517'};
disp(Tlambzero);

SubIntervals2=[1280,2560,5120]';
[~,eigErrorSm11,~]=shooting_evp_newton(0.01,1280,1,-2000,0);
[~,eigErrorSm12,~]=shooting_evp_newton(0.01,2560,1,-2000,0);
[~,eigErrorSm13,~]=shooting_evp_newton(0.01,5120,1,-2000,0);
[~,eigErrorSm14,~]=shooting_evp_newton(0.01,1280,1,-2450,0);
[~,eigErrorSm15,~]=shooting_evp_newton(0.01,2560,1,-2450,0);
[~,eigErrorSm16,~]=shooting_evp_newton(0.01,5120,1,-2450,0);
lambzero11=[eigErrorSm11,eigErrorSm12,eigErrorSm13]';
lambzero12=[eigErrorSm14,eigErrorSm15,eigErrorSm16]';
Tlambzero2=table(SubIntervals2,lambzero11,lambzero12);
Tlambzero2.Properties.VariableNames={'N','LambdaGuess2000','LambdaGuess2450'};
disp(Tlambzero2);

SubIntervals3=[1280,2560,5120]';
[~,eigErrorSm21,~]=shooting_evp_newton(0.01,1280,200,-42498.5,0);
[~,eigErrorSm22,~]=shooting_evp_newton(0.01,2560,200,-42498.5,0);
[~,eigErrorSm23,~]=shooting_evp_newton(0.01,5120,200,-42498.5,0);
lambzero200=[eigErrorSm21,eigErrorSm22,eigErrorSm23]';
Tlambzero200=table(SubIntervals3,lambzero200);
Tlambzero200.Properties.VariableNames={'N','Lambda200Guess'};
disp(Tlambzero200);
```

```
Iteration:1
Approximate Eigenvalue:-2513.755185
Iteration:2
Approximate Eigenvalue:-2514.104951
Iteration:3
Approximate Eigenvalue:-2514.10096
Iteration:4
Approximate Eigenvalue:-2514.10096
     N      LambdaGuess2518    LambdaGuess2517
    ____    _______________    _______________

     320        21.674             0.65786
     640         4.509               4.509
    1280        3.2425              3.2425
    2560        6.0267              40.391

     N      LambdaGuess2000    LambdaGuess2450
    ____    _______________    _______________
```

```
1280         0.23037          0.23037
2560         0.11895          0.11895
5120        0.060418         0.060418

 N       Lambda200Guess
____    _______________

1280         69.688
2560         163.43
5120         93.797
```



Published with MATLAB® R2019a