University of Waterloo

Faculty of Engineering

Department of Electrical and Computer Engineering

ECE 206

Laboratory 1

FALL 2011

Prepared by

Chen, Elijah

UW Student ID Number: 20757687

UW User ID: ey3chen @uwaterloo.ca

Prepared by

Moin, Muneeb

UW Student ID Number: 20771958

UW User ID: mbmoin @uwaterloo.ca

2B Nanotechnolgy Engineering

25 May 2020

Having watched the presentation, we will go through the following exercise:

**1.1** **Stating the Problem**

Summarize in a sentence or two what you are trying to approximate in a boundary-value problem (BVP) with a linear ordinary differential equation (LODE) and a forcing function *g*(*x*).

We are attempting to approximate the solution function to the LODE when given boundary values. The forcing function is the function on the right after moving all the derivative terms to the left. This will be done using finite difference methods.

**1.2** **Determining the Formal Parameters**

Suppose we want to solve a boundary value problem where the ordinary differential equation is known to be linear ODE with constant coefficients together with a forcing function *g*(*x*); that is, we wish to approximate *u*(*x*) when



What are the values that would have to be passed by the user to a function? Answer this question by filling in Tables 1.1b.i, 1.1b.ii, and 1.1b.iii. Note the difference between a parameter that is used to describe the BVP (Tables 1.1b.i and 1.1b.ii) and a parameter that is used to control the numerical solver (Table1.1b.iii). The parameter(s) describing the problem should come before the parameter(s) used in the numerical solution when calling the function.

Table 1.1b.i Parameters describing the problem.

|  |  |
| --- | --- |
| **Given Values** | **Description** |
| [*c*­1, *c*­2, *c*­3] | The coefficients to the derivatives in the 2nd order LODE |
| [*a*, *b*] | Input values to the boundary values |
| [*ua*, *ub*] | Output values to the boundary values |

Table 1.1b.ii Functions describing the problem.

|  |  |  |
| --- | --- | --- |
| **Given Functions** | **Arguments** | **Description** |
| *g* | *x* | The function isolated on the right side without derivative terms related to the function *u(x)*. |

Table 1.1b.iii Parameters that control the numerical solver.

|  |  |
| --- | --- |
| **Method Parameters** | **Description** |
| *n* | The number of that points that will evenly divide the interval. |

Comments:

1. You can add more rows to the tables.
2. The descriptions should be independent of any programming language.
3. Recall that in Matlab, when a function is passed as an argument, you must pass it with @f which creates a *function handle*.

**1.3** **Determining the Return Values**

The return values should included two vectors:

1. A vector of *x* values, and
2. A vector of *u* values.

The order of the output should be the same as the output for the Matlab function ode45 (see help ode45). Give a brief description of these *n*-dimensional vectors.

|  |  |
| --- | --- |
| **Symbol** | **Description** |
| **x**out | The vector **x**out is made up of n elements that correspond to the x value where the approximation for the solution u(x) is evaluated. |
| **u**out | The vector **u**out is made up of n elements that correspond to the evaluation of u(x) at xout |

**1.4** **The Signature and Description of the Matlab Function**

Given the responses in 1.1, 1.2 and 1.3, fill in the description of the function here.

% bvp

% Copy the description from 1.1 here

%

% Parameters

% ==========

% c A vector made up of the coefficients to the derivatives in the 2nd order

% LODE

% x\_int Input values to the boundary values

% u\_int Output values to the boundary values

%

% g The function isolated on the right side without derivative terms related % to the function *u(x)*.

%

% n The number of that points that will evenly divide the interval.

%

% Return Values

% =============

% x The vector **x**out is made up of n elements that correspond to the x value

% where the approximation for the solution u(x) is evaluated.

% u The vector **u**out is made up of n elements that correspond to the

% evaluation of u(x) at xout

function [x, u] = bvp( c, x\_int, u\_int, g, n )

Save this function in Matlab and remember that it must be called bvp.m . To create a new function in Matlab, use *File*→*New*→*Function*. Make sure that the directory that you are saving the function in is the *Current Folder*.

**1.5** **Argument Checking**

The next step is to check the dimensions of the arguments. The function size( w ) returns a vector containing the row and column dimensions. This is then checked against a vector of the expected dimension in order to ensure that w is a 3D column vector (a  
3 × 1 matrix). The function isscalar( x ) returns true if x is a 1 × 1 matrix (Matlab’s version of a scalar value). A scalar is an integer if it equals itself when rounded. Finally, the isa command can be used to determine other properties. Replace the code below with your argument-checking conditional statements. Some examples of argument checking are provided.

if ~all( size( w ) == [3, 1] )

throw( MException( 'MATLAB:invalid\_argument', ...

'the argument w is not a 3-dimensional column vector' ) );

end

if ~isscalar( x )

throw( MException( 'MATLAB:invalid\_argument', ...

'the argument x is not a scalar' ) );

end

if ~isscalar( y ) || ( y ~= round( y ) )

throw( MException( 'MATLAB:invalid\_argument', ...

'the argument y is not an integer' ) );

end

if ~isa( z, 'function\_handle' )

throw( MException( 'MATLAB:invalid\_argument', ...

'the argument z is not a function handle' ) );

end

function [x, u] = bvp( c, x\_int, u\_int, g, n )

% Argument Checking

if ~all( size( c ) == [1, 3] )

throw( MException( 'MATLAB:invalid\_argument', ...

'the argument c is not a 3-dimensional row vector' ) );

end

if ~all( size( x\_int ) == [2, 1] )

throw( MException( 'MATLAB:invalid\_argument', ...

'the argument x\_int is not a 2-dimensional column vector' ) );

end

if ~all( size( u\_int ) == [2, 1] )

throw( MException( 'MATLAB:invalid\_argument', ...

'the argument u\_int is not a 2-dimensional column vector' ) );

end

if ~isa( g, 'function\_handle' )

throw( MException( 'MATLAB:invalid\_argument', ...

'the argument g is not a function handle' ) );

end

if ~isscalar( n ) || ( n ~= round( n ) )

throw( MException( 'MATLAB:invalid\_argument', ...

'the argument n is not an integer' ) );

end

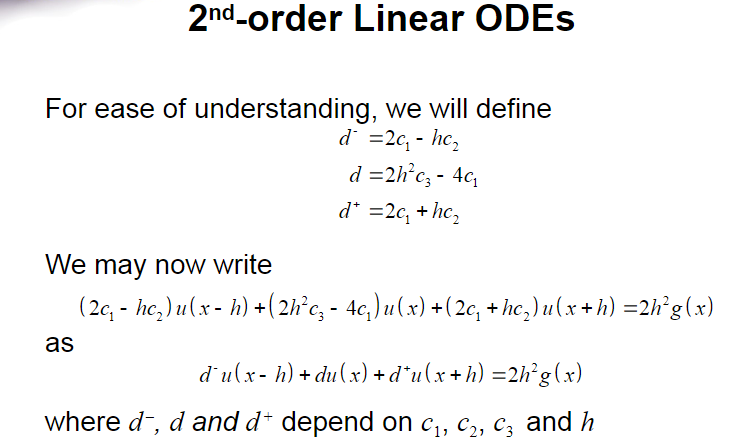
**1.6** **Describing the Solution**

We will now continue with the development of this function that you have defined. Write down, in English, the steps that you would perform in order to solve a boundary-value problem. Points to remember:

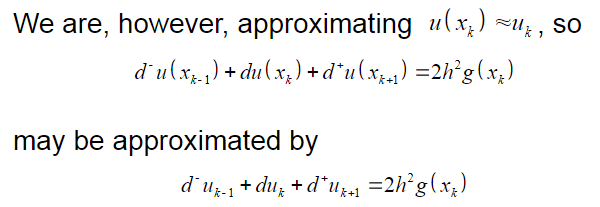
1. The output vector of *x* values will require *n* entries; however, the system of equations we are solving will be (*n* – 2) × (*n* – 2). Which *n* – 2 variables will be found by solving this system of equation?
2. What additional values are required in creating the matrix and the right-hand vector?
3. Once you have solved for the unknown values, can we simply pass these values back to the user?
4. Local variables need to be defined in order to make the operations less complex and easier to read.
5. Step size h needs to be defined. This is the interval distance between n points. This can be done with the linspace function or with this definition:



1. Variables d1, d2, d3 need to be defined. The are the coefficients in the finite difference equation



1. A vector of x values will need to be created which corresponds to each x value that will be approximated in the solution
2. There will be n number of x values. n-2 of which will be unknown as the boundary conditions give the start and end of the interval. This is why a (n-2)x(n-2) matrix needs to be solved.
3. When all the x values are subbed into the finite difference equation, an approximated u value will become an unknown in the system of linear equations.



1. A tridiagonal matrix needs to be created to solve the internal x2 – xn-1 points.
2. The right-hand side vector can be calculated with the forcing function evaluating each x value.
3. The first and last elements of the right-hand side can be expressed with the known boundary values. This allows the matrix to be solved.
4. The vector u which contains the approximations for the solutions at each x value can be solved now. It will need to be outputted as a n-dimensional column vector. A vector x should also be outputted which contains the points where u was evaluated.

Remember to describe your steps in English. It would make sense to step through the slides to determine the steps. Do not combine too many operations into a single step.

**1.7*a*** **Implementing the Solution**

The next step is to convert your code to Matlab code. Each of the steps you indicated in Question 1.2*a* will be translated into one or more Matlab commands. For each step, indicate the appropriate Matlab commands. If you define a local variable in any step, document the purpose of the variable.

% Step 1: give a brief description

a = ...; % purpose

Your commands here

% Step 2: give a brief description

b = ...; % purpose

Your commands here, ...

Copy and paste your entire Matlab function with comments here:

% bvp

% Copy the description from 1.1 here

%

% Parameters

% ==========

% c A vector made up of the coefficients to the derivatives in the 2nd order

% LODE

% x\_int Input values to the boundary values

% u\_int Output values to the boundary values

%

% g The function isolated on the right side without derivative terms related

% to the function u(x).

%

% n The number of that points that will evenly divide the interval.

%

% Return Values

% =============

% x The vector xout is made up of n elements that correspond to the x value

% where the approximation for the solution u(x) is evaluated.

% u The vector uout is made up of n elements that correspond to the

% evaluation of u(x) at xout

function [x, u] = bvp( c, x\_int, u\_int, @g, n )

% Argument Checking

if ~all( size( c ) == [1, 3] )

throw( MException( 'MATLAB:invalid\_argument', ...

'the argument c is not a 3-dimensional row vector' ) );

end

if ~all( size( x\_int ) == [2, 1] )

throw( MException( 'MATLAB:invalid\_argument', ...

'the argument x\_int is not a 2-dimensional column vector' ) );

end

if ~all( size( u\_int ) == [2, 1] )

throw( MException( 'MATLAB:invalid\_argument', ...

'the argument u\_int is not a 2-dimensional column vector' ) );

end

if ~isa( g, 'function\_handle' )

throw( MException( 'MATLAB:invalid\_argument', ...

'the argument g is not a function handle' ) );

end

if ~isscalar( n ) || ( n ~= round( n ) )

throw( MException( 'MATLAB:invalid\_argument', ...

'the argument n is not an integer' ) );

end

% define the step size h. h is needed in calculating the finite

% difference formula

h = (x\_int(2) - x\_int(1)) / (n-1);

% create variables d1, d2, d3 to simplify the coefficients

d1 = (2 \* c(1)) - (h \* c(2));

d2 = (2 \* (h ^ 2) \* c(3)) - (4 \* c(1));

d3 = ((2 \* c(1)) + (h \* c(2)));

% a vector of x values needs to be created which will be evaluated in the

% approximation

x\_values = linspace(x\_int(1), x\_int(2), n)';

% create the tri diagonal matrix of d coefficients

d\_matrix = (diag( d2 \* ones((n-2), 1)) + diag( d3 \* ones((n-3), 1), 1) + diag( d1 \* ones((n-3), 1), -1));

% create a vector of x\_values that will be evaluated at solution function

mysterious\_x\_values = x\_values(2: end -1);

% create target vector with forcing function involved

target\_vector = (2 .\* (h ^ 2) .\* (g(mysterious\_x\_values)));

intermediate\_zeroes\_vector = zeros((n-2), 1);

intermediate\_zeroes\_vector(1) = (-d1 \* u\_int(1));

intermediate\_zeroes\_vector(end) = (-d3 \* u\_int(2));

final\_target\_vector = target\_vector + intermediate\_zeroes\_vector;

% solve for the final vector u

u = d\_matrix \ final\_target\_vector;

**1.8** **Testing your Implementation**

We are now ready to test your code.

**1.8*a*** In the Laboratory slides, two examples are provided. The first example is homogeneous: that is, the forcing function is zero. You will therefore have to write a forcing function that takes a vector x as an argument and return a vector of zeros of the same dimension as x (try y = 0\*x). The second has a non-zero forcing function that you will also have to implement. Denote these two forcing functions by g1 and g2 respectively. With these functions, you will determine whether or not your code is working. Plot the points. Your plots do not have to include the actual solutions (the functions indicated in the slides in red). Instead, you will only have to plot the points as circles. For each of the outputs, include the title with your UW User ID(s).

title( 'uwuserid' );

title( 'uwuserid and uwuserid' );

Save these two images and replace them in Figures 1 and 2. List the commands you used to:

1. Call your function and approximate the solution, and
2. Plot the two approximations.

Your code here.

% bvp

% Copy the description from 1.1 here

%

% Parameters

% ==========

% c A vector made up of the coefficients to the derivatives in the 2nd order

% LODE

% x\_int Input values to the boundary values

% u\_int Output values to the boundary values

%

% g The function isolated on the right side without derivative terms related % to the function u(x).

%

% n The number of that points that will evenly divide the interval.

%

% Return Values

% =============

% x The vector xout is made up of n elements that correspond to the x value

% where the approximation for the solution u(x) is evaluated.

% u The vector uout is made up of n elements that correspond to the

% evaluation of u(x) at xout

function [x\_values, u\_values] = bvp(c, x\_int, u\_int, g, n)

% Argument Checking

if ~all( size( c ) == [1, 3] )

throw( MException( 'MATLAB:invalid\_argument', ...

'the argument c is not a 3-dimensional row vector' ) );

end

if ~all( size( x\_int ) == [1, 2] )

throw( MException( 'MATLAB:invalid\_argument', ...

'the argument x\_int is not a 2-dimensional row vector' ) );

end

if ~all( size( u\_int ) == [1, 2] )

throw( MException( 'MATLAB:invalid\_argument', ...

'the argument u\_int is not a 2-dimensional row vector' ) );

end

if ~isa( g, 'function\_handle' )

throw( MException( 'MATLAB:invalid\_argument', ...

'the argument g is not a function handle' ) );

end

if ~isscalar( n ) || ( n ~= round( n ) )

throw( MException( 'MATLAB:invalid\_argument', ...

'the argument n is not an integer' ) );

end

% define the step size h. h is needed in calculating the finite

% difference formula

h = (x\_int(2) - x\_int(1)) / (n-1);

% create variables d1, d2, d3 to simplify the coefficients

d1 = (2 \* c(1)) - (h \* c(2));

d2 = (2 \* (h ^ 2) \* c(3)) - (4 \* c(1));

d3 = ((2 \* c(1)) + (h \* c(2)));

% a vector of x values needs to be created which will be evaluated in the

% approximation

x\_values = (linspace(x\_int(1), x\_int(2), n)');

% create the tri diagonal matrix of d coefficients

d\_matrix = (diag( d2 \* ones((n-2), 1)) + diag( d3 \* ones((n-3), 1), 1) + diag( d1 \* ones((n-3), 1), -1));

% create a vector of x\_values that will be evaluated at solution function

mysterious\_x\_values = x\_values(2: end -1);

% create target vector with forcing function involved

target\_vector = (2 .\* (h ^ 2) .\* (g(mysterious\_x\_values)));

intermediate\_zeroes\_vector = zeros((n-2), 1);

intermediate\_zeroes\_vector(1) = (-d1 \* u\_int(1));

intermediate\_zeroes\_vector(end) = (-d3 \* u\_int(2));

final\_target\_vector = target\_vector + intermediate\_zeroes\_vector;

% solve for the final vector u

u = d\_matrix \ final\_target\_vector;

% add the known u terms to the final u vector

u\_values = zeros(n, 1);

u\_values(1) = u\_int(1);

u\_values(end) = u\_int(2);

u\_values(2:end - 1) = u;

% plot the solution

plot(x\_values, u\_values, 'ro');

title( 'ey3chen and mbmoin' );

>> bvp( [1 3 2], [0, 1], [4, 5], @g1, 9 )

>> bvp( [1 3 2], [0, 1], [4, 5], @g2, 9 )

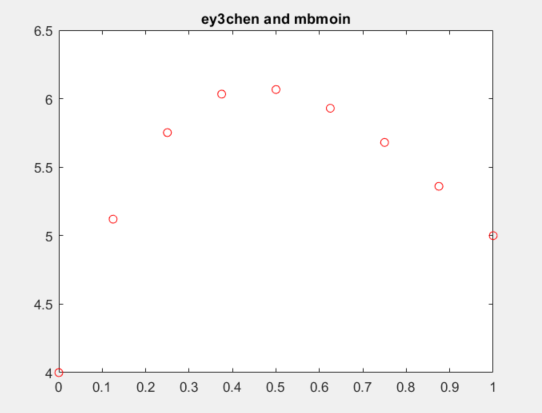


Figure 1. The first solution.

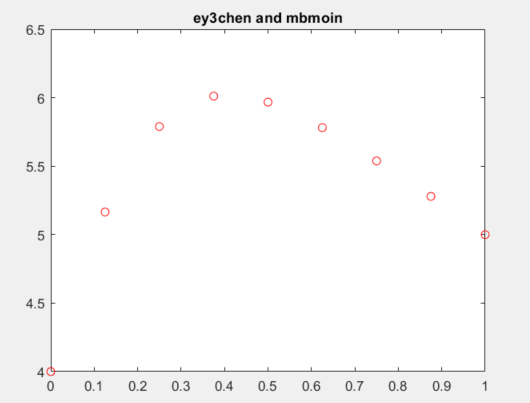


Figure 2. The second solution.

**1.8*b*** Repeat Problem 1.8*a*, but use a larger number of points. Use a value of *n* equal to 50 plus the last two digits of one of your UW Student ID numbers. Do not use circles, as you did in the previous question; instead, use red points in your plot. Use help plot to see how to plot points instead of circles. Save these two images and replace them in Figures 3 and 4. Be sure to include your UW User ID(s) in the title.

Save these two images and replace them in Figures 1 and 2. List the commands you used to solve the problem and plot the two solutions.

Your code here.

UW ID number sum 87 + 58 = 145

n = 195

>> bvp( [1 3 2], [0, 1], [4, 5], @g1, 195 )

>> bvp( [1 3 2], [0, 1], [4, 5], @g2, 195 )

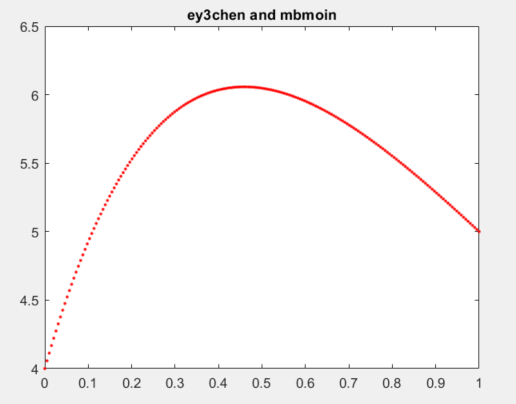


Figure 3. The first solution.

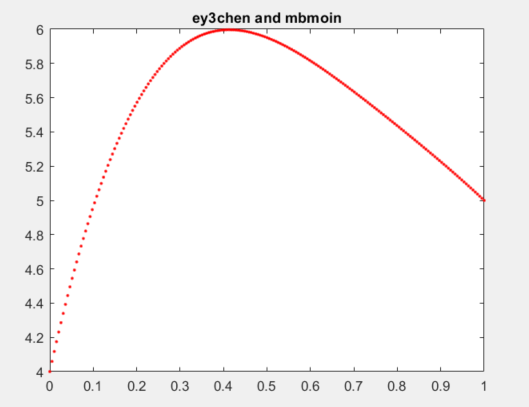


Figure 4. The second solution.

**1.8*c*** Now is your opportunity to create and solve your own BVP. Select values where the coefficients of the ODE are non-zero, pick two boundary points *a* and *b* where *a* < *b* and pick two boundary values. Use Maple to generate the solution by using the two commands:

dsolve( {c1\*(D@@2)(u)(x) + c2\*D(u)(x) + c3\*u(x) = 0, u(a) = u\_a, u(b) = u\_b}, u(x) );

plot( rhs( % ), x = a..b ); # use the same a and b...

Launch Maple and select *File*→*New*→*Worksheet Mode* and cut-and-paste the above Maple command.

Make sure you replace anything in red in the Maple code with your numbers!

Indicate the variables you have chosen here:

*c*1 = 9

*c*2 = 6

*c*3 = 5

*a* = 2

*b* = 8

*ua* = 28

*ub* = 3

The ODE will be homogenous. Forcing function will return 0

Copy the plot from Maple into Figure 5. WE DON’T HAVE MAPLE

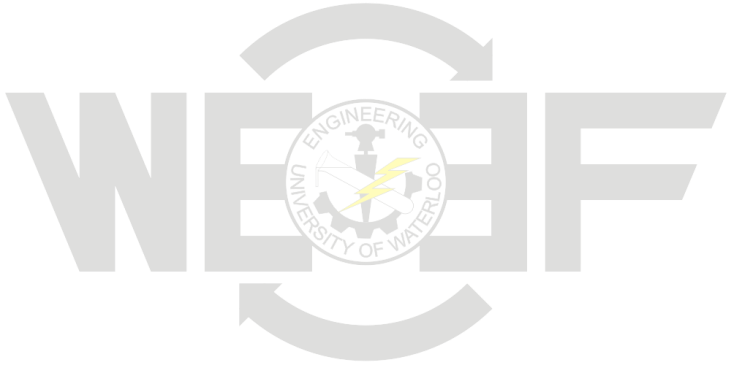


Figure 5. The Maple solution.

Finally, use your function with *n* = 10 points (use red circles) and *n* = 100 points (use red points) and copy the Matlab plots into Figure 6 and 7. Be sure to include your UW User ID(s) in the title.

List the commands you used to solve the problem and plot the two solutions.

Your code here.

% bvp

% Copy the description from 1.1 here

%

% Parameters

% ==========

% c A vector made up of the coefficients to the derivatives in the 2nd order

% LODE

% x\_int Input values to the boundary values

% u\_int Output values to the boundary values

%

% g The function isolated on the right side without derivative terms related % to the function u(x).

%

% n The number of that points that will evenly divide the interval.

%

% Return Values

% =============

% x The vector xout is made up of n elements that correspond to the x value

% where the approximation for the solution u(x) is evaluated.

% u The vector uout is made up of n elements that correspond to the

% evaluation of u(x) at xout

function [x\_values, u\_values] = bvp(c, x\_int, u\_int, g, n)

% Argument Checking

if ~all( size( c ) == [1, 3] )

throw( MException( 'MATLAB:invalid\_argument', ...

'the argument c is not a 3-dimensional row vector' ) );

end

if ~all( size( x\_int ) == [1, 2] )

throw( MException( 'MATLAB:invalid\_argument', ...

'the argument x\_int is not a 2-dimensional row vector' ) );

end

if ~all( size( u\_int ) == [1, 2] )

throw( MException( 'MATLAB:invalid\_argument', ...

'the argument u\_int is not a 2-dimensional row vector' ) );

end

if ~isa( g, 'function\_handle' )

throw( MException( 'MATLAB:invalid\_argument', ...

'the argument g is not a function handle' ) );

end

if ~isscalar( n ) || ( n ~= round( n ) )

throw( MException( 'MATLAB:invalid\_argument', ...

'the argument n is not an integer' ) );

end

% define the step size h. h is needed in calculating the finite

% difference formula

h = (x\_int(2) - x\_int(1)) / (n-1);

% create variables d1, d2, d3 to simplify the coefficients

d1 = (2 \* c(1)) - (h \* c(2));

d2 = (2 \* (h ^ 2) \* c(3)) - (4 \* c(1));

d3 = ((2 \* c(1)) + (h \* c(2)));

% a vector of x values needs to be created which will be evaluated in the

% approximation

x\_values = (linspace(x\_int(1), x\_int(2), n)');

% create the tri diagonal matrix of d coefficients

d\_matrix = (diag( d2 \* ones((n-2), 1)) + diag( d3 \* ones((n-3), 1), 1) + diag( d1 \* ones((n-3), 1), -1));

% create a vector of x\_values that will be evaluated at solution function

mysterious\_x\_values = x\_values(2: end -1);

% create target vector with forcing function involved

target\_vector = (2 .\* (h ^ 2) .\* (g(mysterious\_x\_values)));

intermediate\_zeroes\_vector = zeros((n-2), 1);

intermediate\_zeroes\_vector(1) = (-d1 \* u\_int(1));

intermediate\_zeroes\_vector(end) = (-d3 \* u\_int(2));

final\_target\_vector = target\_vector + intermediate\_zeroes\_vector;

% solve for the final vector u

u = d\_matrix \ final\_target\_vector;

% add the known u terms to the final u vector

u\_values = zeros(n, 1);

u\_values(1) = u\_int(1);

u\_values(end) = u\_int(2);

u\_values(2:end - 1) = u;

% plot the solution

plot(x\_values, u\_values, 'r.');

title( 'ey3chen and mbmoin' );

>> bvp( [9 6 5], [2, 8], [28, 3], @g1, 10 )

>> bvp( [9 6 5], [2, 8], [28, 3], @g1, 100 )

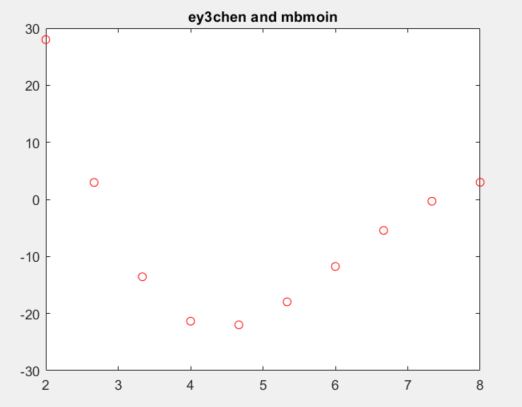


Figure 6. The approximation with 10 points.

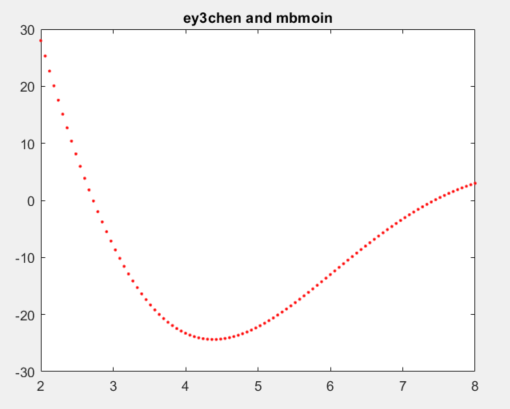


Figure 7. The approximation with 100 points.