University of Waterloo

Faculty of Engineering

Department of Electrical and Computer Engineering

Hello.

NE 216

Laboratory 4

FALL 2011

Prepared by

Sharda, Vid

UW Student ID Number: 20833192

UW User ID: v2sharda@uwaterloo.ca

Prepared by

Chockalingam Shanmugam, Anish

UW Student ID Number: 20814528

UW User ID: a2chocka @uwaterloo.ca

2A Nanotechnology Engineering

November 3, 2020

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

**4.1-7** **Your Solution**

In Laboratory 2, you were required to document all aspects of your algorithm. You were also required to check all parameters for their types, *etc*.

You must do the same for this laboratory; however, you will only need to provide the final solution for the Dormand-Prince algorithm; however, you will still be required to implement and test an implementation of the 4th-order Runge Kutta algorithm with the signature

function [t\_out, y\_out] = rk4( f, t\_rng, y0, n )

Copy and paste your **entire Matlab** function for dp45 with comments and error checking here:

function [t\_out, y\_out] = dp45( f, t\_rng, y0, h, eps\_abs )

function [t\_out, y\_out] = dp45( f, t\_rng, y0, h, eps\_abs )

%argument checking

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

throw( MException( 'MATLAB:invalid\_argument', 'the argument f is not a function handle' ) );

end

if ~all( size(t\_rng) == [1, 2] )

throw( MException( 'MATLAB:invalid\_argument','the argument t\_rng is not a 2-dimensional row vector' ) );

end

if ~isscalar(y0)

throw( MException( 'MATLAB:invalid\_argument','the argument y0 is not a scalar' ) );

end

if ~isscalar(h) || (h <= 0)

throw( MException( 'MATLAB:invalid\_argument','the argument h is not an valid' ) );

end

if ~isscalar(eps\_abs) || (eps\_abs <= 0)

throw( MException( 'MATLAB:invalid\_argument','the argument eps\_abs is not an valid' ) );

end

A = [ 0 0 0 0 0 0 0;

1 0 0 0 0 0 0;

1/4 3/4 0 0 0 0 0;

11/9 -14/3 40/9 0 0 0 0;

4843/1458 -3170/243 8056/729 -53/162 0 0 0;

9017/3168 -355/33 46732/5247 49/176 -5103/18656 0 0;

35/384 0 500/1113 125/192 -2187/6784 11/84 0]';

by = [5179/57600 0 7571/16695 393/640 -92097/339200 187/2100 1/40]';

bz = [ 35/384 0 500/1113 125/192 -2187/6784 11/84 0]';

c = [0 1/5 3/10 4/5 8/9 1 1]';

% Initialize t\_out and y\_outlen(

n\_K = 7;

K = zeros( 1, n\_K );

t0 = t\_rng(1);

tf = t\_rng(2);

t\_out(1) = t0;

y\_out(1) = y0;

% Initialize our location to k = 1

k = 1;

while t\_out(k) < tf

% Use Dormand Prince to find two approximations

% y\_tmp and z\_tmp to approximate y(t) at

% t = t\_out(k) + h for the current value of h

for m = 1:n\_K

K(m) = f( t\_out(k) + h\*c(m), y\_out(k) + h\*c(m)\*K\*A(:,m) );

end

y\_tmp = y\_out(k) + h\*K\*by;

z\_tmp = y\_out(k) + h\*K\*bz;

% Calculate the scaling factor 's'

s = ((h\*eps\_abs)/(2\*(tf-t0)\*abs(y\_tmp - z\_tmp)))^(0.25);

if s >= 2

% We use z\_tmp to approximate y\_out(k + 1)

% t\_out(k + 1) is the previous t-value plus h

% Increment k and double the value of h for the

% next iteration.

y\_out(k+1) = z\_tmp;

t\_out(k+1) = t\_out(k) + h;

h = h\*2;

k = k+1;

elseif s >= 1

% We use z\_tmp to approximate y\_out(k + 1)

% t\_out(k + 1) is the previous t-value plus h

% In this case, h is neither too large or too

% small, so only increment k

y\_out(k+1) = z\_tmp;

t\_out(k+1) = t\_out(k) + h;

k = k+1;

else s < 1

% Divide h by two and try again with the smaller

% value of h (just go through the loop again

% without updating t\_out, y\_out, or k)

h = h/2;

end

% We must make one final check before we end the loop:

% if t\_out(k) + h > tf, we must reduce the

% size of h so that t\_out(k) + h == tf

if t\_out(k) + h > tf

h = tf - t\_out(k);

end

end

end

**4.8** **Testing your Implementation**

**Warning: All the questions you will do in parts *a*, *b*, and *c* will be repeated with the Dormand-Prince method in parts *d*, *e*, and *f*, respectively. Use scripts to reduce your workload! Consider using a for loop for parts *c* and *f*.**

We are now ready to test your code. For all questions, use format long

**4.8*a*** With examples from the presentations, we had:

function [dy] = f4a(t, y)  
 dy = (y - 1).^2 .\* (t - 1).^2;  
end  
  
function [y] = y4a\_soln( t )  
 y = (t.^3 - 3\*t.^2 + 3\*t)./(t.^3 - 3\*t.^2 + 3\*t + 3);  
end

and we approximated the solution with:

[t2a, y2a] = rk4( @f4a, [0, 1], 0, 11 )   
plot( t2a, y2a, 'or' )  
hold on  
[t2a, y2a] = ode45( @f4a, [0, 1], 0 );

plot( t2a, y2a, 'b' )

It’s up to you to make sure your solution matches the output shown in the presentation.

Save the right-hand side of the differential equation



as the function *f*4*b* and execute the following:

[t2b, y2b] = rk4( @f4b, [0, 1], 0, 11 )

plot( t4b, y4b, 'or' )

[t2b, y4b] = ode45( @f4b, [0, 1], 0 );  
hold on

plot( t4b, y4b, 'b' )

title( 'uwuserid and uwuserid' ) % or title( 'uwuserid' )

Copy the output of the second plot (with the title) into Figure 1.

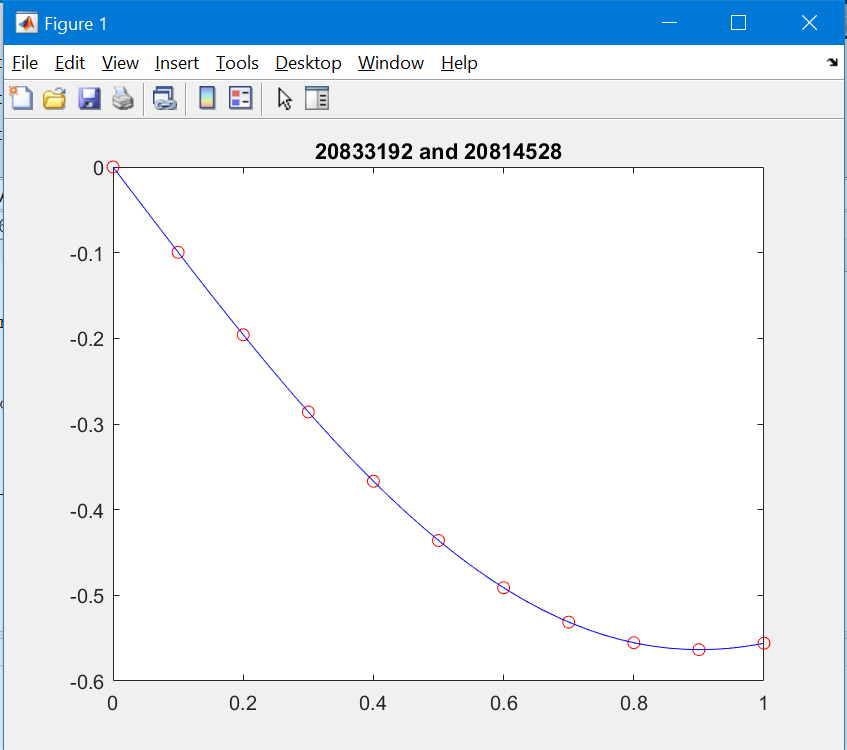


Figure . The approximation of the IVP *y*(1)(*t*) = *f*2*b*(*t*, *y*(*t*)) with *y*(0) = 0.

**4.8*b*** Approximate the solution of the IVP



by first defining the function f4c and copying it here:

function [dy] = f4c(t, y)  
 dy = (4-t)/(y + t);

end

Next, approximate *y*(1) using 11 points and print out the vectors of eleven *t-* and *y-*values.

Copy and paste your Matlab commands and the output here.

t\_out =

Columns 1 through 6

0 0.100000000000000 0.200000000000000 0.300000000000000 0.400000000000000 0.500000000000000

Columns 7 through 11

0.600000000000000 0.700000000000000 0.800000000000000 0.900000000000000 1.000000000000000

y\_out =

Columns 1 through 6

1.000000000000000 1.326546112892431 1.567902820687289 1.763795286762154 1.930031937789864 2.074849006941079

Columns 7 through 11

2.203193919137289 2.318318560608788 2.422498401065690 2.517400931142394 2.604292321666371

**4.8*c*** We will demonstrate that the error is O(*h*4) by taking the differential equation in Question 4.8*b* and using 11, 21, 41, 81, 161, 321, and 641 points to approximate *y*(1). The actual value, to 20 decimal digits, is *y*(1) = 2.604215099096980. Copy the approximation and the absolute error of the approximation into Table 4.8c.

Table 4.8*c*. Approximations of *y*(1) for the given IVP.

|  |  |  |  |
| --- | --- | --- | --- |
| *n* | *h* | Approximation | Absolute Error |
| 11 | 0.1 | 2.604292321666371 | 7.7223e-005 |
| 21 | 0.05 | 2.604219980033546 | 4.8809e-06 |
| 41 | 0.025 | 2.604215391440248 | 2.9234e-07 |
| 81 | 0.0125 | 2.604215116686282 | 1.7589e-08 |
| 161 | 0.00625 | 2.604215100170741 | 1.0738e-09 |
| 321 | 0.003125 | 2.604215099163233 | 6.6253e-11 |
| 641 | 0.0015625 | 2.604215099101094 | 4.1140e-12 |

Argue why this demonstrates that the error is O(*h*4).

It is because when h is divided by 2, the abs error is divided by 2^4. So the error is of order 4, O(*h*4).

What value of *n* must be used with Euler and Heun, respectively, in order to achieve approximately the same absolute error of the 4th-order Runge Kutta algorithm with  
*n* = 11? Compare the number of evaluations of the argument *f*.

For Euler: n between 10000 and 100000

For Heun: n=56

The relation between n trials and the methods are:

Runge Kutta: 4(n-1) = 40

Euler: (n-1) = approx. 100000

Heun: 2(n-1) = 110

This is due to the number of slopes that are being calculated in each case.

You will now implement the Dormand-Prince method and repeat these exercises.

**4.8*d*** As before, confirm your solution with:

[t4a, y4a] = dp45( @f4a, [0, 1], 0, 0.01, 1e-5 )   
plot( t4a, y4a, 'or' )  
hold on  
[t4a, y4a] = ode45( @f4a, [0, 1], 0 );

plot( t4a, y4a, 'b' )

and

[t4b, y4b] = dp45( @f4b, [0, 1], 0, 0.01, 1e-5 )

plot( t4b, y4b, 'or' )

[t4b, y4b] = ode45( @f4b, [0, 1], 0 );  
hold on

plot( t4b, y4b, 'b' )

title( 'uwuserid and uwuserid' ) % or title( 'uwuserid' )

It’s up to you to make sure your solution matches the output shown in the presentation.

Copy the output of the second plot (with the title) into Figure 2.

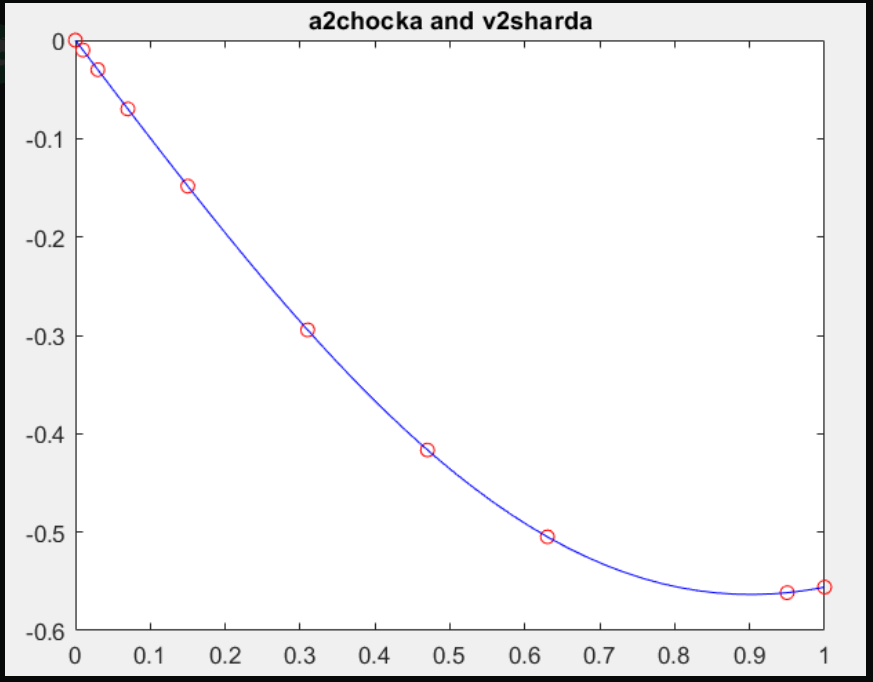


Figure 2. The approximation of the IVP *y*(1)(*t*) = *f*4*b*(*t*, *y*(*t*)) with *y*(0) = 0.

**4.8*e*** Approximate the solution of the IVP



at *y*(1) using *h* = 0.1 and **abs = 10–6and print out the vectors of eleven *t-* and *y-*values.

t4c =

Columns 1 through 5

0 0.025000000000000 0.050000000000000 0.075000000000000 0.100000000000000

Columns 6 through 10

0.125000000000000 0.175000000000000 0.225000000000000 0.275000000000000 0.325000000000000

Columns 11 through 15

0.425000000000000 0.525000000000000 0.625000000000000 0.725000000000000 0.925000000000000

Column 16

1.000000000000000

y4c =

Columns 1 through 5

1.000000000000000 1.094106431485348 1.178515263467405 1.255461102465744 1.326427862406392

Columns 6 through 10

1.392460856552590 1.512619873032635 1.620208401635104 1.717931151487049 1.807630627329486

Columns 11 through 15

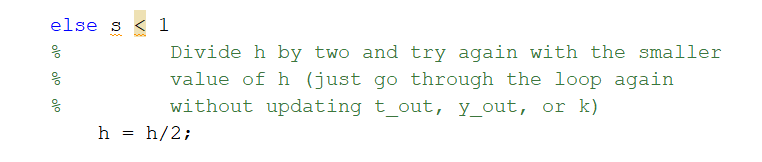
1.967916456613455 2.108240377788033 2.233025985710020 2.345232651687315 2.539753553421280

Column 16

2.604215105161850

What appears to be happening to the value of *h* as we progress? What happened with the first step where we should have approximated the solution at *t* = 0.1?

The h value is clearly not constant as we go through the loop, because of s. The first step we had h = 0.1, however it showed up as 0.025. This is because when the program calculated for s, it was less than 1. It was one of the conditions that if s <1, h = h/2. After executing this, it runs it again without modifying the values of t\_out, y\_out or k.



And since the value of h observed is ¼ of the initial, this condition must have been met 2 times. The same thing happens to h between 0.175 and 0.225, where s met the condition of being more than 2, which made h = h\*2.

**4.8*f*** We will demonstrate that the error is approximately O(*h*4) by taking the differential equation in Question 4.8*e* and using **abs = 0.0001/16*k* for *k* = 0, 1, …, 6 to approximate *y*(1). The actual value, to 20 decimal digits, is *y*(1) = 2.604215099096980. Copy the approximation and the absolute error of the approximation into Table 4.8*f*. Use *h* = 0.1.

Table 4.8*f*. Approximations of *y*(1) for the given IVP.

|  |  |  |  |
| --- | --- | --- | --- |
| *k* | **abs | length( t\_out ) | Absolute Error of Approximation of *y*(1) |
| 0 | 0.0001 | 7 | 4.6233e-006 |
| 1 | 0.0001/16 | 12 | 1.600764321096904e-08 |
| 2 | 0.0001/162 | 20 | 1.602362242891786e-09 |
| 3 | 0.0001/163 | 35 | 1.245501479729683e-10 |
| 4 | 0.0001/164 | 67 | 4.708233802830364e-12 |
| 5 | 0.0001/165 | 131 | 1.709743457922741e-13 |
| 6 | 0.0001/166 | 256 | 7.993605777301127e-15 |

Argue why this demonstrates that the error is approximately O(*h*4). Recall that if we have twice as many points, the average step size is dropping by half.

It is because when h is divided by 2, the abs error is divided by 2^4. This is what causes the error to be approximately O(*h*4).

What value of *n* must be used with the 4th-order Runge Kutta algorithm to achieve an accuracy of less than **abs = 10–6. In Question 4.8*e*, Dormand-Prince used *n* = 18 points and therefore 7(*n* – 1) = 119 function evaluations. Compare this with the number of evaluations of the argument *f* in rk4.

For the Runge Kutta method it was observed that:

|  |  |  |  |
| --- | --- | --- | --- |
| n | h | Approximation | Error |
| 21 | 0.05 | 2.604219980033546 | 4.8809e-06 |
| 41 | 0.025 | 2.604215391440248 | 2.9234e-07 |

So the n is in between 21 and 41.

for n = 21:41

[t4c, y4c] = rk4( @f4c, [0, 1], 1, n);

if abs(y4c(end)-2.604215099096980)<1e-6

n

break

end

end

>> n = 31

So, in order to get an **abs error less than 1e-6, the n value was 31. This results in 4(31 – 1) = 120 function evaluations. The function evaluations are very close when compared to dp45’s 119. However, there might be differences in other cases.

**4.9** Did you remember to copy your entire functions dp45 into Question 4.7? That is, all comments and all code?

Yes