Course Assignment 3

Vijaya Krishna Bodla, Roman Tschentscher

5th December 2005

Chapter 1

Solutions for the tasks

1.1 Exercise 1

1) First a 'simp.m' is generated based on the Simpson's rule of integration, by dividing the interval [a,b] to m even subintervals:

$$\int_a^b F(x)dx = S(h) + R_T \tag{1.1}$$

with $h = \frac{b-a}{m}$ and

$$S(h) = \frac{h}{3}(f_0 + 4f_1 + 2f_2 + ... + 2f_{m-2} + 4f_{m-1} + f_m) \tag{1.2}$$

$$R_T = \frac{b-a}{180}h^4 f^4 \eta \tag{1.3}$$

with $a < \eta < b$. Matlab code of this function is as following:

```
function f=func(x)
f=sqrt(x)*sin(x);


the calling function using the simpson rule
function  S = simp(a,b,m)
% Approximate integral by simpson's rule

format long
x = linspace(a,b,m+1);                    % grid points
h=(b-a)/m;                                %defining the interval
S = (feval('func',a) + feval('func',b));  % end point contrib.
```

```
for  i = 1 : m -1
 if rem(i,2)==0
 S = S + 2*feval('func',x(i+1));           % interior point contrib.
 else
     S = S + 4*feval('func',x(i+1));
 end
 end
S = h/3 * S;                               % multiply by  h
```

2) The function $f(x) = \sqrt{x}sinx$ has a singularity at x=0. both Simpson's and Trapezoidal has problem in this point. So instead of 0 the matlab function **realmin** is used. The program is given below:

```
fprintf('\n solution for part 1.2')
m=[20 40 80 160];
for i=1:4,
   a(i)=feval('simp',realmin,0.5,m(i));
   b(i)=feval('trapezrule',realmin,0.5,m(i));
a(i)=abs(a(i)-0.06908795737);
b(i)=abs(b(i)-0.06908795737);
end
fprintf('simpson          trapezrule');
for i=1:3,
p(i) = -log2(a(i)/a(i+1));
q(i) = -log2(b(i)/b(i+1));
fprintf('\n %15.10f %15.10f',p(i),q(i));
end

% part 1.3
fprintf('\n solution for part 1.3')
format long
m=6;k=1;
a=realmin;b=0.5;
while k==1
 F=feval('simp',realmin,0.5,m);
m=m+2;
Err=abs(F-0.06908795737);
Derr=10^(-10);
  if Err < Derr
     k=0;
  end
end
```

```
m=m-2
T=F
```

Different values of intervals m=20, 40, 80 and 160 have been tested for the methods and the following values for the integral have been obtained:

```
m          Simpson            Trapezoidal
20      0.0690893432     0.0691354167
40      0.0690882029     0.0691000064
80      0.0690880008     0.0690910022
160    0.0690879651     0.0690887243
```

The higher the number of intervals, the more precise are the results. Simpson gives a better approximation. The general equation for finding discretization error for these two methods is $E(h) \leq Kh^P$. By deviding two errors for m=20 and 40 we obtain:

$$\frac{E_{20}}{E_{40}} = \frac{Km^P}{K(2m)^P} = \frac{1}{(2)^P} \tag{1.4}$$

So P is calculated by:

$$P = log_2 \frac{E_{40}}{E_{20}} \tag{1.5}$$

The following values of P for 20, 40, 80 and 160 interval points have been obtained:

```
m                    Simpson           Trapezoidal

20, 40      -2.4966001880   -1.9777782105
40, 80      -2.4987644833   -1.9844776633
80,160     -2.4994896613   -1.9891201509
```

So the truncation error of Simpson's rule is around $O(m)^{-2.5}$ and for the trapez rule $O(m)^{-2.0}$.

3) To find an approximation which has an error less than given in the assignment, first the initiate number of intervals is chosen. The integral is calculated for this number of m and compared to the given value in the assignment. In the case wherethe error is larger, the number of interval is increased by 2 and this procedure is continued till the error is less than the given. The matlab code is as following. It calls the program from part 1).

```
fprintf('\n \n solution for part 1.3')
format long
m=6;k=1;
a=realmin;b=0.5;
while k==1
```

```
 F=feval('simp',realmin,0.5,m);
m=m+2;
Err=abs(F-0.06908795737);
Derr=10^(-10);
  if Err < Derr
     k=0;
  end
end
m=m-2;fprintf('\n \n Number of intervals (m)= %g',m);
T=F;fprintf('\n \n Value of integral(I) for [0;0.5]= %15.10f',T);
```

Running the program results in a value of $m = 912$, for which the condition is satisfied.

Part 4) If we choose $x = \sqrt{t}$ and replace, we still have the singularity at $x = 0$. By using $x = t^2$ and inserting into the integral function we get:

$$I = \int_0^0 .70712t^2 sin(t^2)dt \tag{1.6}$$

which has no singularity at $x = 0$

5) Using the variable transformation in the program analog 2.2) we obtain the values for P:

```
m                    Simpson           Trapezoidal

20, 40       -4.0068970226   -1.9998687091
40, 80       -4.0005098827   -1.9999671907
80,160      -3.9807586982   -1.9999916877
```

So the error for simpson is closer to $O(m^{-4})$ than in 2.2). The number of intervals needed to reach the error is $m = 122$, which is much less than the one obtained in 2.3). The programs are given in the appendix.

1.2 Exercise 2

Question 2.1) The quad function in matlab uses an absolute error of 1e-6 by default. Using the function $I = l \cdot \int_0^T k(t)dt$ with t from 0 to 20.

$$k(t) = 1.2 \cdot 10^{-6} \cdot exp(-\frac{1}{6}(t-7)^2) + 0.8 \cdot 10^{-6} \cdot exp(-\frac{1}{6}(t-14)^2) \tag{1.7}$$

The relative error is $RelErr = |\frac{AbsErr}{I}| = 0.0931$. To obtain this relative error less than 10^{-3}, the tolerance for the quad function has to be decreased to 10^{-8}. The programs are shown below.

```
function k=Vfunc(t)
k=1.2*10^-6*exp(-(1/6)*(t-7).^2)+0.8*10^-6*exp((-1/6)*(t-14).^2);

The call file:
nTol=1e-8;
V = 1.2*quad(@Vfunc,0,20,nTol)

RelErr=abs(nTol/V)
```

The result is:

```
V =

  1.0412e-005


RelErr =

  9.6042e-004
```

Question 2.2)

The file **jobdata.mat** is loaded into matlab. Using a spline interpolation we get a polynomial which is used in the quad function, as can be seen in the program:

```
S=load('joddata.mat');

x1 = linspace(0,20,200);

cs=spline(S.tobs,S.kobs,x1);


cs1x=spline(S.tobs,S.kobs,x1);
cs1=spline(S.tobs,S.kobs);
cs2x=spline(S.tobs,[0 S.kobs 0],x1);
cs2=spline(S.tobs,[0 S.kobs 0]);

x=S.tobs;

nTol=1e-8;
V = 1.2*quad(@(x)ppval(cs2,x),0,20,nTol)
V1 = 1.2*quad(@(x)ppval(cs1,x),0,20,nTol)
```

```
figure(1),plot(S.tobs,S.kobs,'o',x1,cs2x);
figure(2),plot(S.tobs,S.kobs,'o',x1,cs1x);
```

The result is:

```
V =

  1.0408e-005


V1 =

  1.0354e-005
```

where V is the the result using rand, V1 is the result obtained without rand.

As can be seen in figure 1.1 the polynomial takes on negative values in the last part, which is not feasible due to the physical meaning of the function. So the rand condition has to be used and the results are acceptable. The errors calculated similar to 2.1 with rand are:

$$AbsError = 1.0408e - 005 \tag{1.8}$$

$$RelErr = 3.8417e - 004 \tag{1.9}$$

1.3 Exercise 3

Inserting the values for x and y in the expressions for r_1 and r_2 given in the assignment and solving gives:

$$r_1 = \sqrt{(\frac{1}{2} - \mu + \mu)^2 + \frac{3}{4}} = 1 \tag{1.10}$$

$$r_2 = \sqrt{((-\frac{1}{2})^2 + \frac{3}{4}} = 1 \tag{1.11}$$

Replacing these values in the second order differential equations and assuming that $x' = 0$, $y' = 0$ gives:

$$x'' = 2 \cdot 0 + (\frac{1}{2} - \mu) - \frac{1}{2} \cdot (1 - \mu) - \mu \cdot (-\frac{1}{2}) = 0 \tag{1.12}$$

$$y'' = 2 \cdot 0 + y - y \cdot (1 - \mu) - \mu \cdot y = 0 \tag{1.13}$$

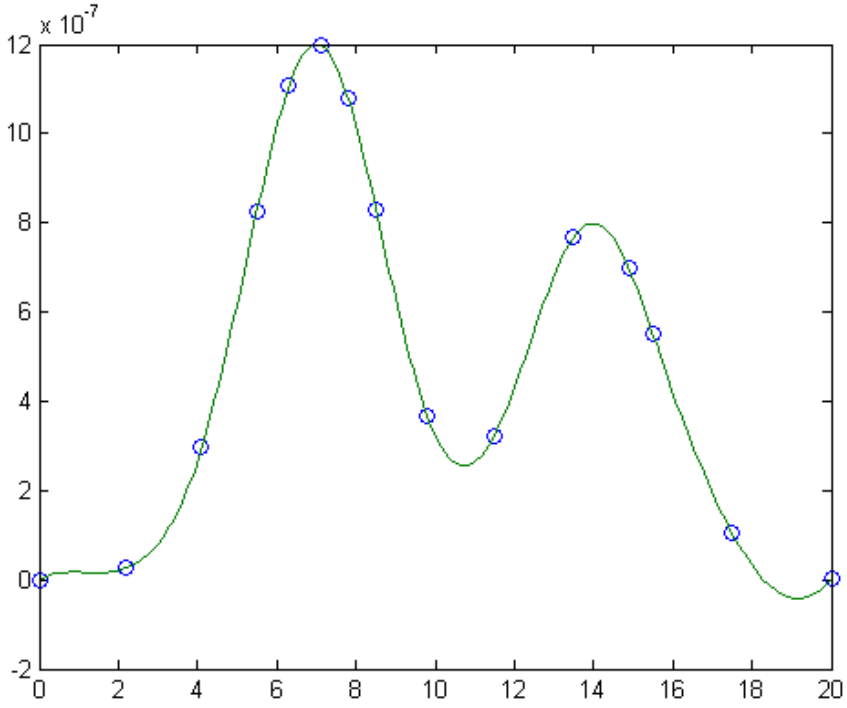

Figure 1.1: Datapoints and polynomial without rand.

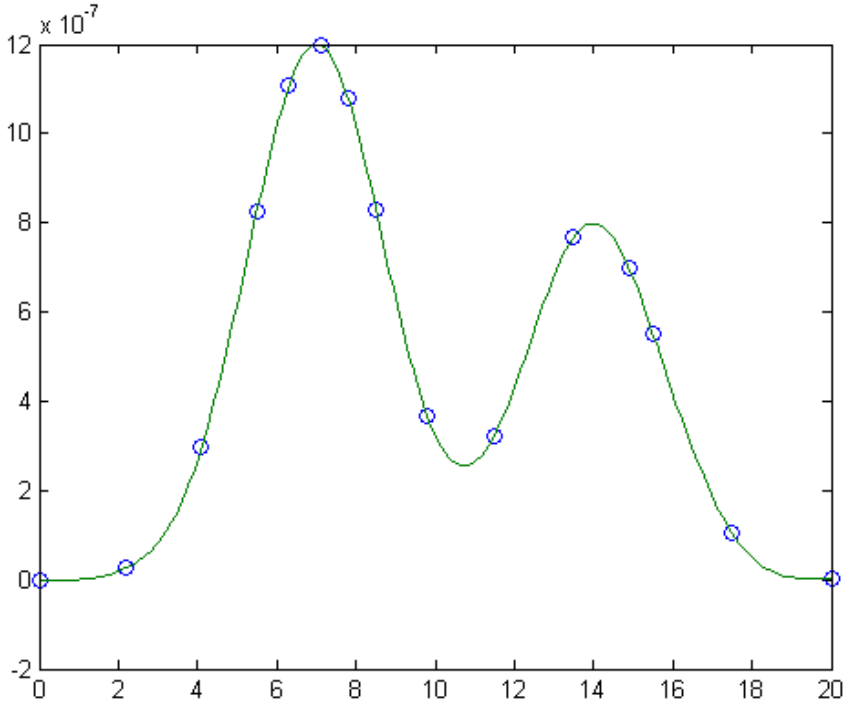

Figure 1.2: Datapoints and polynomial with rand.

3.2)
Second order equations cannot be solved by matlab. It is therefore needed to rearrange it in the following way:

$$x = x(1) \qquad x' = x(1)' = x(3) \qquad x'' = x(1)'' = x(3)' \tag{1.14}$$
$$y = x(2) \qquad y' = x(2)' = x(4) \qquad y'' = x(2)'' = x(4)' \tag{1.15}$$

The four variables and the differential equations are described as one matrix:

$$x = \begin{bmatrix} x \\ y \\ x' \\ y' \end{bmatrix} = \begin{bmatrix} x(1) \\ x(2) \\ x(3) \\ x(4) \end{bmatrix} \tag{1.16}$$

The system of differential equations is set up in the same way:

$$dx = \begin{bmatrix} x' \\ y' \\ x'' \\ y'' \end{bmatrix} \tag{1.17}$$

where the differential equations from (1.14) and the assignment are used. The four variables x, y, x', y' are expressed by $x(1) - x(4)$, as can be seen in the program **satellite.m**:

```
function dx=satellite(t,x)

dx=zeros(4,1);
%
% parameters
M1=5.976*10^24;
M2=7.349*10^22;
mu=M2/(M1+M2);
% dimensionless distances to the gravity point
r1=sqrt((x(1)+mu)^2+x(2)^2);
r2=sqrt((x(1)-1+mu)^2+x(2)^2);
% the differential system
dx(1)=x(3); % x_1'=x_2
dx(2)=x(4); % y_1'=y_2
dx(3)=2*x(4)+x(1)-(1-mu)*(x(1)+mu)/r1^3-mu*(x(1)-1+mu)/r2^3; % x''=....
dx(4)=-2*x(3)+x(2)-(1-mu)*x(2)/r1^3-mu*x(2)/r2^3; % y''=....
```

This system of four first order differential equations can now be solved numerically using an ode-solver.

Inserting the starting conditions as given in the assignment, where (x(1),x(2)) is the position of a Lagrange-point and for the velocity (x(3),x(4)) we insert (0,0), the following result is obtained:

```
>> satellitetest

x =

    0.4879    0.8660         0         0


dx =

  1.0e-015 *

         0         0   -0.1310   -0.2637


ans =

  1.0e-015 *

         0         0   -0.1310   -0.2637
```

showing that x' and y' are zero, x'' and y'' are close to zero, which means that the satellite remains at this position. The program is shown in the appendix.

3.3)

The matlab program **run_satellite.m**, that calls the program **satellite.m** given above and using the initial conditions as given in the assignment, is shown in the following

```
close all;
x0=[1.2 0 0 -1.0496];

% options=odeset('Stats','on');
options=odeset('AbsTol',1e-8,'RelTol',1e-8,'Stats','on');

[t,x]=ode45(@satellite,[0 10],[x0],options);
figure;
```

```
plot(t,x(:,1),t,x(:,3))
figure;
plot(t,x(:,2),t,x(:,4))
% parameters
M1=5.976*10^24;
M2=7.349*10^22;
mu=M2/(M1+M2);
xE=-mu;yE=0;
xM=1-mu;yM=0;
figure;
plot(x(:,1),x(:,2),xE,yE,xM,yM);
xlabel('x')
ylabel('y')
```

Without changing the odeset options the error tolerance properties are $RelTol = 10^{-3}$ and $AbsTol = 10^{-6}$. The results show a very strong change in the satellite positions due to the big stepsize of the solver:

```
>> run_satellite
81 successful steps
22 failed attempts
619 function evaluations
0 partial derivatives
0 LU decompositions
0 solutions of linear systems
```

Changing the odeset-options as given in the assignment gives a much smoother curve. The stepsize is much reduced, the calculation time increases but the result is much better:

```
>> run_satellite
567 successful steps
0 failed attempts
3403 function evaluations
0 partial derivatives
0 LU decompositions
0 solutions of linear systems
```

3.4)a)

The algorithm shown in the assignment is built in the following way: From the ode-solver results the position of the satellite relative to the moon is calculated for all points of the way the satellite takes. Then for every point of the curve the distance from the center of the moon is calculated. If then a distance is found that is smaller

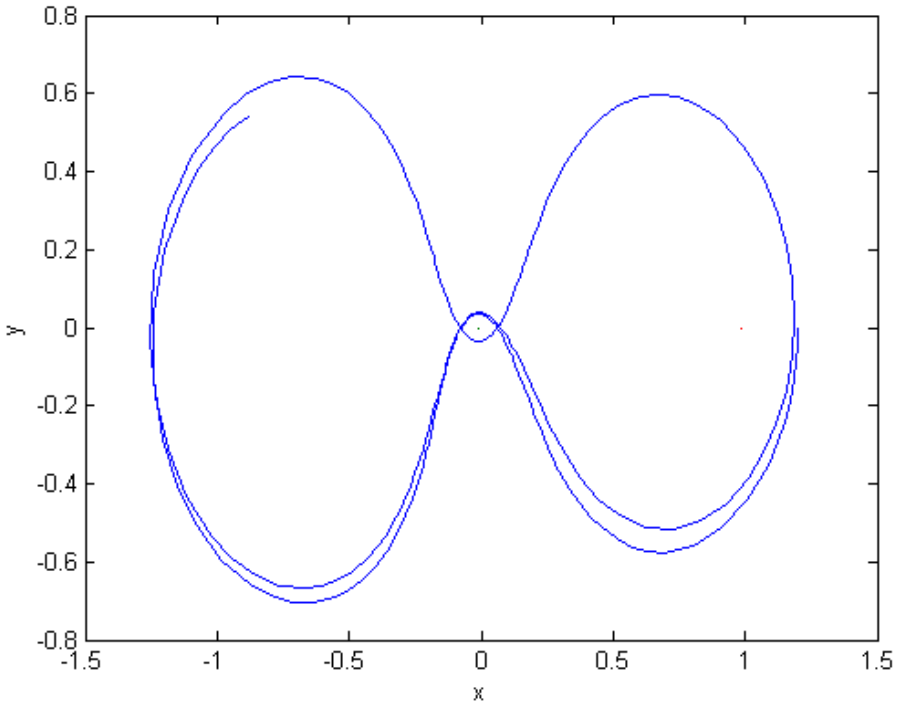

Figure 1.3: Circulation of the satellite for tolerance given in the ode-solver.

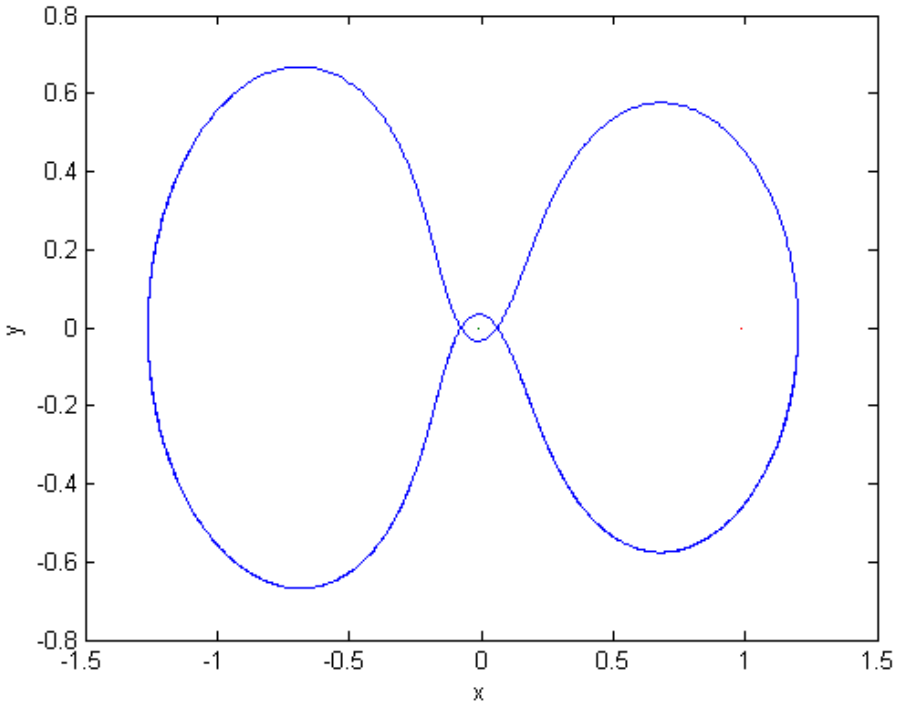

Figure 1.4: Circulation of the satellite for reduced tolerance in the ode-solver.

then the moon radius, it means that the satellite crashes the moon on this curve. If it doesnt crash, the algorith checks the smallest distance between satellite and moon. Depending on the sign and value of this value it checks if the velocity was too small. This is, if the x-value of the closest position is smaller then the moon position, the satellite surrounds the earth without reaching the moon. The other possibility is that the satellite passes the moon from downwards. So the satellite y-position at lowest distance is negative.

Problems can occur in this statement: If the satellite is to fast,so it surrounds the moon and just crossed the x-axis behind the moon, then the distance of the first point in the negative y-region could be the closest to the moon. So here the algorithm shows that the velocity is too small, but in fact it is too high. This algorithm also calculates the distances for the whole curve. It could b written in that way, that the x-position when crossing the x-axis is found and then checked, if this is before or after the moon or the moon is crashed. And then a different velocity is chosen depending on that.

b)
To find the velocity at which the moon is crashed by the satellite the bisectional methos is used. The nice thing of this method is that not just a velocity but a range of velocities is given as result, due to the fact, that the moon is not a point but a body of certain size. So a range of velocities leads to a crash at different positions of the surface of the moon. The velocity range is given from 0-10, so the initial velocity chosen is 5. The radius of moon and earth are made dimensionless. A loop was built, in which the algorithm given in the assignment is inserted. Depending on the value of f the starting velocity is changed and thereby the velocity range is decreased in every refining step. The program **solve_satellite.m** can be found in the appendix, page 21.

When a velocity is fould at which the satellite hits the moon, the loop is left and this velocity ans well as the two velocities from the refinement step before are given. A plot is made showing the crash.

```
>> solve_satellite

v_y =

    7.7539


v_y1 =

    7.7344
```

```
v_y2 =

    7.7734
```

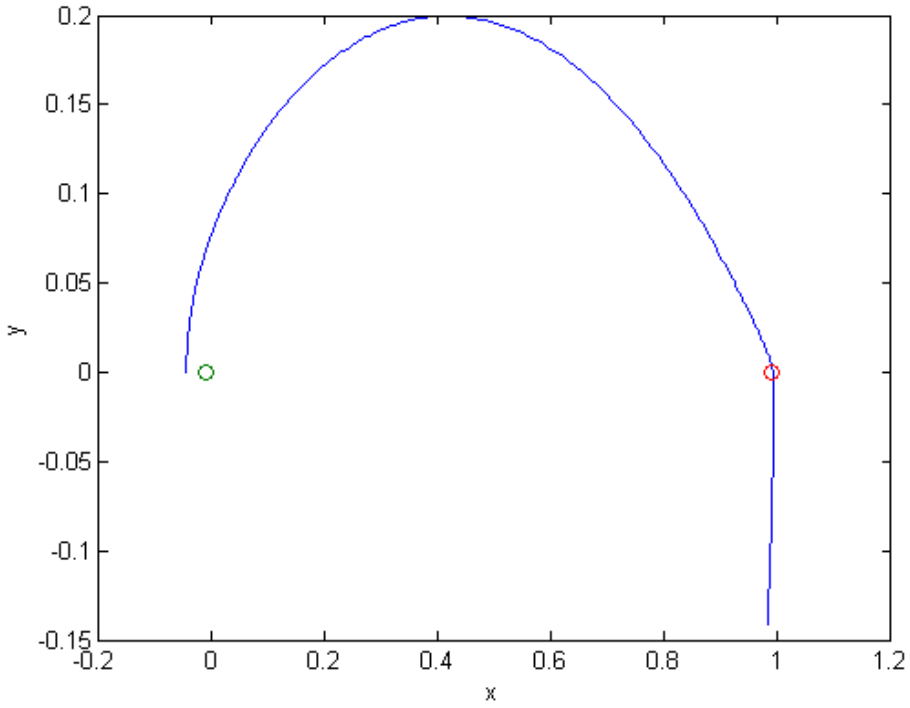

Figure 1.5: Crash of the satellite for reduced tolerance in the ode-solver.

Appendix A

Programs for exercise 1

```
function f=func14(x)
f=2*x^2*sin(x^2);

The call function using the simpson rule is
function  S = simp14(a,b,m)
% part 1.1 for 1.5
% Approximate integral by trapezoidal rule

% Version 11.12.2003.  INCBOX
format long
x = linspace(a,b,m+1);                    % grid points
h=(b-a)/m;          %defining the interval
S = (feval('func14',a) + feval('func14',b));   % end point contrib.
for  i = 1 : m -1
 if rem(i,2)==0
 S = S + 2*feval('func14',x(i+1));  % interior point contrib.
 else
     S = S + 4*feval('func14',x(i+1));
 end
 end
S = h/3 * S; % multiply by  h

The function to find the number of intervalls for given error is:
%part 1.5
fprintf('\n solution for part 1.5 part one')
m=[20 40 80 160];
for i=1:4,
   a(i)=feval('simp14',0,sqrt(0.5),m(i));
```

```
   b(i)=feval('trapezrule',0,sqrt(0.5),m(i));
a(i)=abs(a(i)-0.06908795737);
b(i)=abs(b(i)-0.06908795737);
end
fprintf('\n simpson          trapezrule');
for i=1:3,
p(i) = -log2(a(i)/a(i+1));
q(i) = -log2(b(i)/b(i+1));
fprintf('\n %15.10f %15.10f',p(i),q(i));
end
fprintf('\n part two')
format long
m=6;k=1;
a=0;b=sqrt(0.5);
while k==1
 F=feval('simp14',0,sqrt(0.5),m);
m=m+2;
Err=abs(F-0.06908795737);
Derr=10^(-10);
  if Err < Derr
     k=0;
  end
end
m=m-2
T=F
```

Appendix B

Program satellitetest.m

```
function dx=satellitetest(t,x)
dx=zeros(4,1); %

% parameters
M1=5.976*10^24;
M2=7.349*10^22;

mu=M2/(M1+M2);

% test parameters
% position is lagrange point
x(1)=0.5-mu;
x(2)=sqrt(3)/2;
% velocity is zero
x(3)=0;
x(4)=0;
x

% dimensionless distances to the gravity point
r1=sqrt((x(1)+mu)^2+x(2)^2);
r2=sqrt((x(1)-1+mu)^2+x(2)^2);


% the differential system
dx(1)=x(3); % x_1'=x_2

dx(2)=x(4); % y_1'=y_2
```

```
dx(3)=2*x(4)+x(1)-(1-mu)*(x(1)+mu)/r1^3-mu*(x(1)-1+mu)/r2^3; % x''=....

dx(4)=-2*x(3)+x(2)-(1-mu)*x(2)/r1^3-mu*x(2)/r2^3; % y''=....

dx=dx'

% the result must give for dx(1-2)=zero, cause we are at a lagrange point
% and have no velocity
```

Appendix C

Program solve_satellite.m

```
close all;
% parameters
M1=5.976*10^24;
M2=7.349*10^22;
mu=M2/(M1+M2)
RJ=6378.14;
RM=1737.4;
R=384400;
rJ=RJ/R
rM=RM/R;
xs=-mu-rJ-1.03*rJ

v_y1=0;
v_y2=10;
% loop
done=1 % statement, which is fullfilled or not
while (done==1)
v_y=(v_y1+v_y2)/2
x0=[xs 0 0 v_y];
options=odeset('AbsTol',1e-8,'RelTol',1e-8,'Stats','on');
[t,x]=ode45(@satellite,[0 0.5],[x0],options);
% checking if the distance of one of the points in the projectile
% is smaller than the moon radius --> then we have hit the moon
dxy = [x(:,1)-(1-mu) x(:,2)];
r2 = sqrt( sum(dxy.^2,2) );
i = find(r2 <= rM);
if ~isempty(i)
done=0;
```

```
f=0;
else
[f,i] = min(r2);
if x(i,2) <= 0 | max(x(:,1)) < 1-mu, f = -f; end
end
if f < 0
    v_y1=v_y;
elseif f>0
    v_y2=v_y;
end
end

% show the velocities
v_y
v_y1
v_y2

xE=-mu;yE=0;
xM=1-mu;yM=0;
figure;
plot(x(:,1),x(:,2),xE,yE,'o',xM,yM,'o');
xlabel('x')
ylabel('y')
```