Computational Physics using MATLAB®

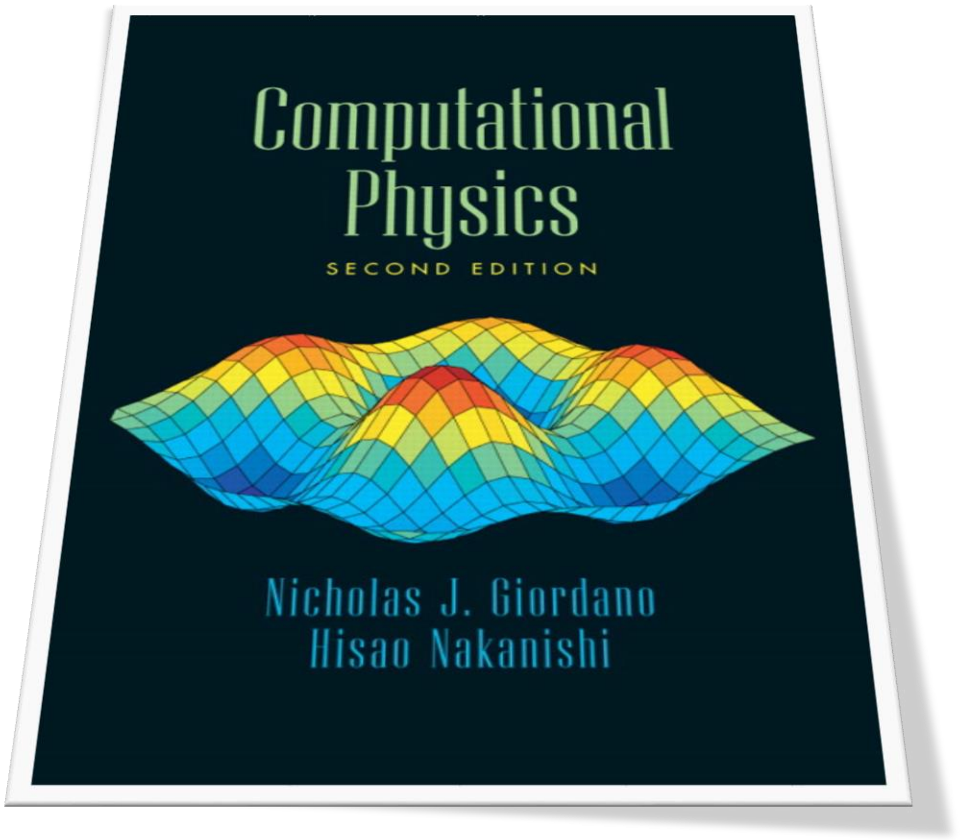


Table of Contents

Table of Figures

Preface

I came across the book, *'Computational Physics'*, in the library here in the Dublin Institute of Technology in early 2012. Although I was only looking for one, quite specific piece of information, I had a quick look at the Contents page and decided it was worth a more detailed examination. I hadn't looked at using numerical methods since leaving College almost a quarter century ago. I cannot remember much attention being paid to the fact that this stuff was meant to be done on a computer, presumably since desktop computers were still a bit of a novelty back then. And while all the usual methods, Euler, Runge-Kutta and others were covered, we didn't cover applications in much depth at all.

It is very difficult to anticipate what will trigger an individual's intellectual curiosity but this book certainly gripped me. The applications were particularly well chosen and interesting. Since then, I have been working through the exercises intermittently for my own interest and have documented my efforts in this book, still a work in progress.

Coincidentally, I had started to use MATLAB® for teaching several other subjects around this time. MATLAB® allows you to develop mathematical models quickly, using powerful language constructs, and is used in almost every Engineering School on Earth. MATLAB® has a particular strength in data visualisation, making it ideal for use for implementing the algorithms in this book.

The Dublin Institute of Technology has existing links with Purdue University since, together with UPC Barcelona, it offers a joint Master's Degree with Purdue in Sustainability, Technology and Innovation via the Atlantis Programme. I travelled to Purdue for two weeks in Autumn 2012 to accelerate the completion of this personal project.

I would like to thank a number of people who assisted in the production of this book. The authors of *'Computational Physics',* Nick Giordano and Hisao Nakanishi from the Department of Physics at Purdue must be first on the list. I would like to thank both of them sincerely for their interest, hospitality and many useful discussions while I was at Purdue. They provided lot of useful advice on the physics, and their enthusiasm for the project when initially proposed was very encouraging.

I would like to thank the School of Electronics and Communications Engineering at the Dublin Institute of Technology for affording me the opportunity to write this book. I would also like to thank the U.S. Department of Education and the European Commission's Directorate General for Education and Culture for funding the Atlantis Programme, and a particular thanks to Gareth O' Donnell from the DIT for cultivating this link.

Suggestions for improvements, error reports and additions to the book are always welcome and can be sent to me at kevin.berwick@dit.ie. Any errors are, of course, my fault entirely.

Finally, I would like to thank my family, who tolerated my absence when, largely self imposed, deadlines loomed.

Kevin Berwick
West Lafayette, Indiana,
USA,
September 2012

1. Uranium Decay

```
%
% 1D radioactive decay
% by Kevin Berwick,
% based on 'Computational Physics' book by N Giordano and H Nakanishi
% Section 1.2 p2

%  Solve the  Equation      dN/dt = -N/tau



N_uranium_initial = 1000;    %initial number of uranium atoms
npoints = 100;              %Discretize time into 100 intervals
dt = 1e7;                  % time step in years
tau=4.4e9;                   % mean lifetime of 238 U

N_uranium = zeros(npoints,1);   % initializes N_uranium, a vector of dimension npoints X 1,to being
all zeros
time = zeros(npoints,1);          % this initializes the vector time to being all zeros

N_uranium(1) = N_uranium_initial; % the initial condition, first entry in the vector N_uranium is
N_uranium_initial
time(1) =  0; %Initialise time

for step=1:npoints-1 % loop over the timesteps and calculate the numerical solution
N_uranium(step+1) = N_uranium(step) - (N_uranium(step)/tau)*dt;
time(step+1) = time(step) + dt;
end
% For comparison , calculate analytical solution
t=0:1e8:10e9;
N_analytical=N_uranium_initial*exp(-t/tau);
% Plot both numerical and analytical solution
plot(time,N_uranium,'r',t,N_analytical,'b'); %plots the numerical solution in red and the analytical
solution in blue
xlabel('Time in years')
ylabel('Number of atoms')
```

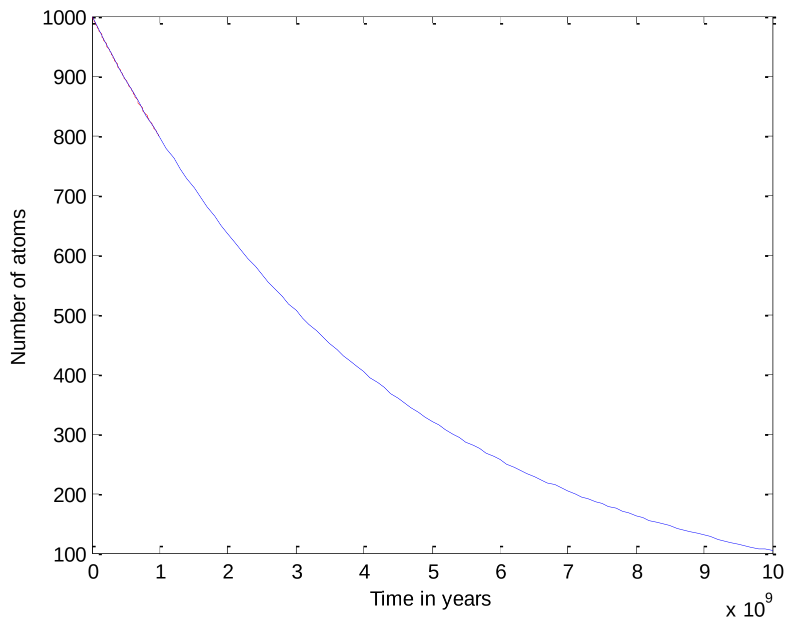


Figure 1. Uranium decay as a function of time

Note that the analytical and numerical solution are coincident in this diagram. It uses real data on Uranium and so the scales are slightly different than those used in the book.

3. The Pendulum

3.1 Solution using the Euler method

Here is the code for the numerical solution of the equations of motion for a simple pendulum using the Euler method. Note the oscillations grow with time. !!

```
%
%  Euler calculation of motion of simple pendulum
%  by Kevin Berwick,
%  based on 'Computational Physics' book by N Giordano and H Nakanishi,
%  section 3.1
%

clear;
length= 1;                              %pendulum length in metres
g=9.8                                   % acceleration due to gravity
npoints = 250;              %Discretize time into 250 intervals
dt = 0.04;                 % time step in seconds
omega = zeros(npoints,1);   % initializes omega, a vector of dimension npoints X 1,to being all zeros
theta = zeros(npoints,1);   % initializes theta, a vector of dimension npoints X 1,to being all zeros
time = zeros(npoints,1);    % this initializes the vector time to being all zeros
theta(1)=0.2;                  % you need to have some initial displacement, otherwise the pendulum will
not swing

for step = 1:npoints-1 % loop over the timesteps
omega(step+1) = omega(step) - (g/length)*theta(step)*dt;
theta(step+1) = theta(step)+omega(step)*dt
time(step+1) = time(step) + dt;
end

plot(time,theta,'r' ); %plots the numerical solution in red
xlabel('time (seconds) ');
ylabel('theta (radians)');
```

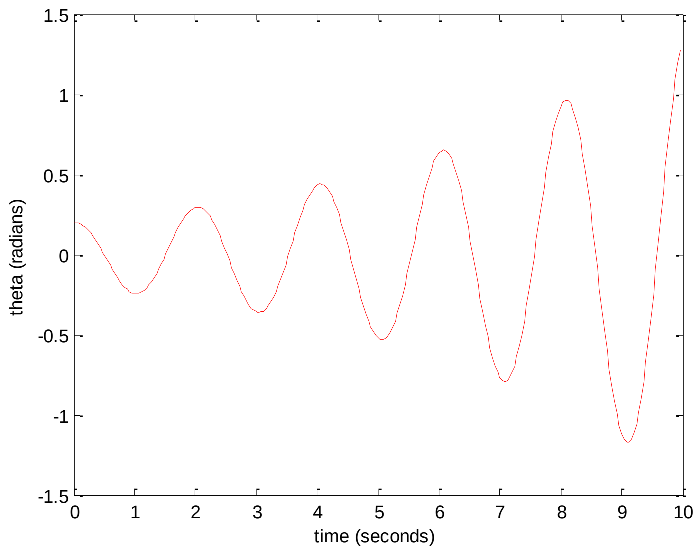


Figure 2. Simple Pendulum - Euler Method

3.1.1 Solution using the Euler-Cromer method.

This problem with growing oscillations is addressed by performing the solution using the Euler - Cromer method. The code is below

```
%
%  Euler_cromer calculation of motion of simple pendulum
%  by Kevin Berwick,
%  based on 'Computational Physics' book by N Giordano and H Nakanishi,
%  section 3.1
%

clear;
length= 1;                          %pendulum length in metres
g=9.8;                              % acceleration due to gravity
npoints = 250;             %Discretize time into 250 intervals
dt = 0.04;                 % time step in seconds
omega = zeros(npoints,1);  % initializes omega, a vector of dimension npoints X 1,to being all zeros
theta = zeros(npoints,1);  % initializes theta, a vector of dimension npoints X 1,to being all zeros
time = zeros(npoints,1);   % this initializes the vector time to being all zeros
theta(1)=0.2;              % you need to have some initial displacement, otherwise the pendulum will
not swing

for step = 1:npoints-1 % loop over the timesteps
omega(step+1) = omega(step) - (g/length)*theta(step)*dt;
theta(step+1) = theta(step)+omega(step+1)*dt;    %note that
% this line is the only change between
% this program and the standard Euler method
time(step+1) = time(step) + dt;
end;

plot(time,theta,'r' ); %plots the numerical solution in red
xlabel('time (seconds) ');
ylabel('theta (radians)');
```

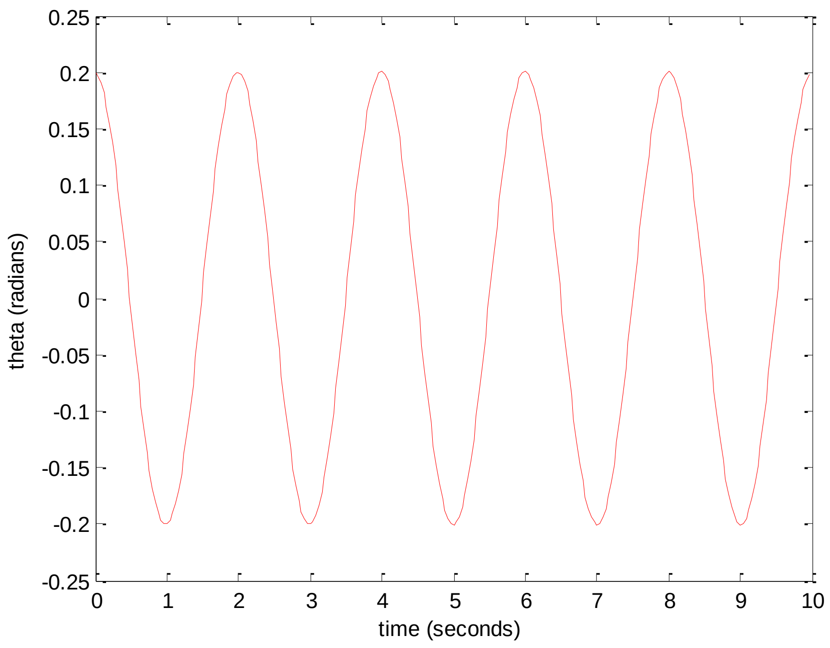


Figure 3. Simple Pendulum: Euler - Cromer method

3.1.2 Simple Harmonic motion example using a variety of numerical approaches

In this example I use a variety of approaches in order to solve the following, very simple, equation of motion. It is based on Equation 3.9, with k and α =1.

$$\frac{d^2y}{dt^2} = -y$$

I take 4 approaches to solving the equation, illustrating the use of the Euler, Euler Cromer, Second order Runge-Kutta and finally the built in MATLAB® solver ODE23.

The solution using the built in MATLAB® solver ODE23 is somewhat less straightforward than those using the other techniques. A discussion of the technique follows.

The first step is to take the second order ODE equation and split it into 2 first order ODE equations.
These are $\frac{dy}{dt} = v$ $\frac{dv}{dt} = -y$

Next you create a MATLAB® function that describes your system of differential equations. You get back a vector of times, T, and a matrix Y that has the values of each variable in your system of equations over the times in the time vector. Each column of Y is a different variable.
MATLAB® has a very specific way to define a differential equation, as a function that takes one vector of variables in the differential equation, plus a time vector, as an argument and returns the derivative of that vector. The only way that MATLAB® keeps track of which variable is which inside the vector is the order you choose to use the variables in. You define your differential equations based on that ordering of variables in the vector, you define your initial conditions in the same order, and the columns of your answer are also in that order.

In order to do this, you create a state vector y. Let element 1 be the vertical displacement, y1, and element 2 is the velocity,v. Next, we write down the state equations, dy1 and dy2. These are

dy1=v;
dy2=-y1

Next, we create a vector dy, with 2 elements, dy1 and dy2. Finally we call the MATLAB® ODE solver ODE23. We take the output of the function called my_shm. We perform the calculation for time values range from 0 to 100. The initial velocity is 0, the initial displacement is 10. The code to do this is here

```
[t,y]=ode45(@my_shm,[0,100],[0,10]);
```

Finally, we need to plot the second column of the y matrix, containing the displacement against time. The code to do this is

```
plot(t,y(:,2),'r');
```

Here is the top level code to do the comparison

```
%
%  Simple harmonic motion - comparison of Euler, Euler Cromer
%  and 2nd order Runge Kutta and built in MATLAB Runge Kutta
%  function ODE45to solve ODEs.
%  by Kevin Berwick,
%  based on 'Computational Physics' book by N Giordano and H Nakanishi,
%  section 3.1
% Equation is d2y/dt2 = -y

% Calculate the numerical solution using Euler method in red
[time,y] = SHM_Euler (10);

subplot(2,2,1);
plot(time,y,'r' );
axis([0 100 -100 100]);
xlabel('Time');
ylabel('Displacement');
legend ('Euler method');


% Calculate the numerical solution using Euler Cromer method in blue
[time,y] = SHM_Euler_Cromer (10);

subplot(2,2,2);
plot(time,y,'b' );
axis([0 100 -20 20]);
xlabel('Time');
ylabel('Displacement');
legend ('Euler Cromer method');

% Calculate the numerical solution using Second order Runge-Kutta method in green
[time,y] = SHM_Runge_Kutta (10);

subplot(2,2,3);
plot(time,y,'g' );
axis([0 100 -20 20]);
xlabel('Time');
ylabel('Displacement');
legend ('Runge-Kutta method');

% Use the built in MATLAB ODE45 solver to solve the ODE
% The function describing the SHM equations is called my_shm
% The time values range from 0 to 100
% The initial velocity is 0, the initial displacement is 10

[t,y]=ode23(@SHM_ODE45_function,[0,100],[0,10]);

% We need to plot the second column of the y matrix, containing the
% displacement against time in black

subplot(2,2,4);
plot(t,y(:,2),'k');
axis([0 100 -20 20]);
xlabel('Time');
ylabel('Displacement');
legend ('ODE45 Solver');
```

Here are the functions to do the individual calculations

```
%
%  Simple harmonic motion -  Euler method
%  by Kevin Berwick,
%  based on 'Computational Physics' book by N Giordano and H Nakanishi,
%  section 3.1
% Equation is d2y/dt2 = -y

function [time,y] = SHM_Euler (initial_displacement);

npoints = 2500;            %Discretize time into 250 intervals
dt = 0.04;                % time step in seconds

v = zeros(npoints,1);   % initializes v, a vector of dimension npoints X 1,to being all zeros
y = zeros(npoints,1);   % initializes y, a vector of dimension npoints X 1,to being all zeros
time = zeros(npoints,1); % this initializes the vector time to being all zeros
y(1)=initial_displacement;                   % need some initial displacement

% Euler solution
for step = 1:npoints-1 % loop over the timesteps
v(step+1) = v(step) - y(step)*dt;
y(step+1) = y(step)+v(step)*dt;
time(step+1) = time(step) + dt;
end;
```

```
%
%  Simple harmonic motion -  Euler Cromer method
%  by Kevin Berwick,
%  based on 'Computational Physics' book by N Giordano and H Nakanishi,
%  section 3.1
% Equation is d2y/dt2 = -y

function [time,y] = SHM_Euler_Cromer (initial_displacement);

npoints = 2500;            %Discretize time into 250 intervals
dt = 0.04;                % time step in seconds

v = zeros(npoints,1);   % initializes v, a vector of dimension npoints X 1,to being all zeros
y = zeros(npoints,1);   % initializes y, a vector of dimension npoints X 1,to being all zeros
time = zeros(npoints,1); % this initializes the vector time to being all zeros
y(1)=initial_displacement;                   % need some initial displacement

% Euler Cromer solution
for step = 1:npoints-1 % loop over the timesteps
v(step+1) = v(step) - y(step)*dt;
y(step+1) = y(step)+v(step+1)*dt;
time(step+1) = time(step) + dt;
end;
```

```
%
%  Simple harmonic motion -  Second order Runge Kutta method
%  by Kevin Berwick,
%  based on 'Computational Physics' book by N Giordano and H Nakanishi,
%  section 3.1
% Equation is d2y/dt2 = -y
function [time,y] = SHM_Runge_Kutta(initial_displacement);

% 2nd order Runge Kutta solution

npoints = 2500;            %Discretize time into 250 intervals
dt = 0.04;                 % time step in seconds

v = zeros(npoints,1);   % initializes v, a vector of dimension npoints X 1,to being all zeros
y = zeros(npoints,1);   % initializes y, a vector of dimension npoints X 1,to being all zeros
time = zeros(npoints,1); % this initializes the vector time to being all zeros
y(1)=initial_displacement;                  % need some initial displacement


v = zeros(npoints,1);   % initializes v, a vector of dimension npoints X 1,to being all zeros
y = zeros(npoints,1);   % initializes y, a vector of dimension npoints X 1,to being all zeros
v_dash = zeros(npoints,1);   % initializes y, a vector of dimension npoints X 1,to being all zeros
y_dash = zeros(npoints,1);   % initializes y, a vector of dimension npoints X 1,to being all zeros
time = zeros(npoints,1); % this initializes the vector time to being all zeros
y(1)=10;                   % need some initial displacement

for step = 1:npoints-1           % loop over the timesteps

   v_dash=v(step)-0.5*y(step)*dt;
   y_dash=y(step)+0.5*v(step)*dt;

  v(step+1) = v(step)-y_dash*dt;
  y(step+1) = y(step)+v_dash*dt;
   time(step+1) = time(step)+dt;
end;
```

```
%
%  Simple harmonic motion -  Built in MATLAB ODE45 method
%  by Kevin Berwick,
%  based on 'Computational Physics' book by N Giordano and H Nakanishi,
%  section 3.1
% Equation is d2y/dt2 = -y

function dy = SHM_ODE45_function(t,y);

% y is the state vector

y1 = y(1);      % y1 is displacement
v =  y(2);     % y2 is velocity

% write down the state equations

dy1=v;
dy2=-y1;

% collect the equations into a column vector, velocity in column 1,
% displacement in column 2
```

```
dy = [dy1;dy2];
```

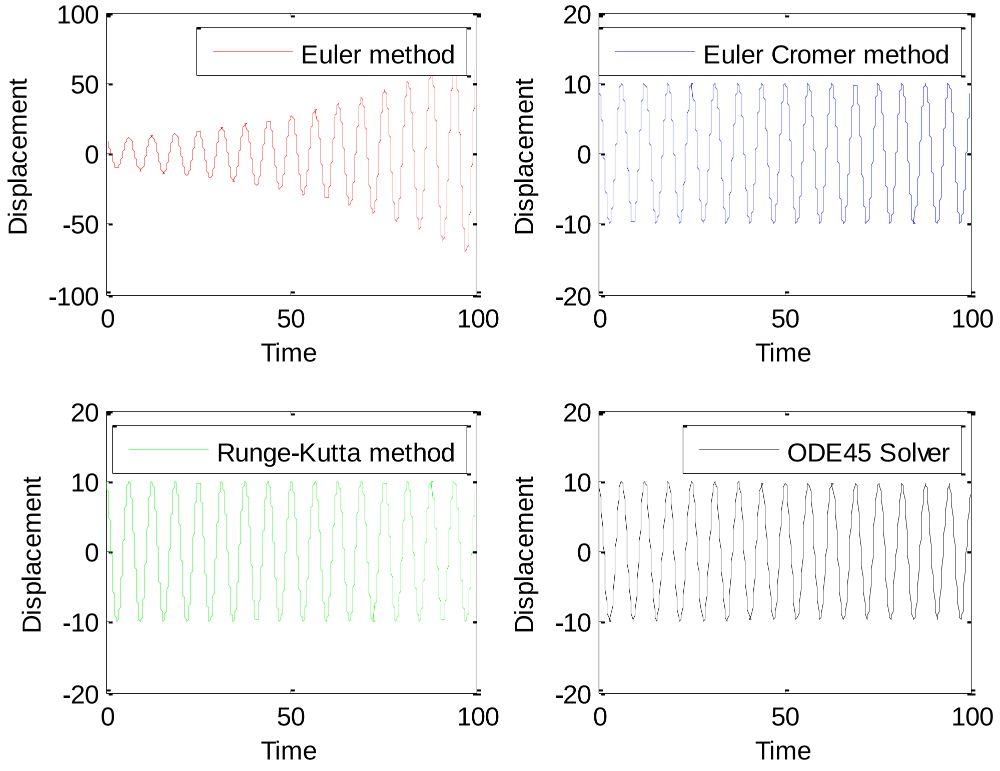


Figure 4. Simple pendulum solution using Euler, Euler Cromer, Runge Kutta and Matlab ODE45 solver.

3.2 Solution for a damped pendulum using the Euler-Cromer method.

This solution uses q=1

```
%
%  Euler_cromer calculation of motion of simple pendulum with damping
%  by Kevin Berwick,
%  based on 'Computational Physics' book by N Giordano and H Nakanishi,
%  section 3.2
%

clear;
length= 1;                              %pendulum length in metres
g=9.8;                                  % acceleration due to gravity
q=1;                                 % damping strength

npoints = 250;            %Discretize time into 250 intervals
dt = 0.04;               % time step in seconds

omega = zeros(npoints,1);   % initializes omega, a vector of dimension npoints X 1,to being all zeros
theta = zeros(npoints,1);   % initializes theta, a vector of dimension npoints X 1,to being all zeros
time = zeros(npoints,1);    % this initializes the vector time to being all zeros

theta(1)=0.2;                 % you need to have some initial displacement, otherwise the pendulum will
not swing

for step = 1:npoints-1 % loop over the timesteps
omega(step+1) = omega(step) - (g/length)*theta(step)*dt-q*omega(step)*dt;
theta(step+1) = theta(step)+omega(step+1)*dt;

% In the Euler method, , the previous value of omega
% and the previous value of theta are used to calculate the  new values of omega and theta.
% In the Euler Cromer method, the previous value of omega
% and the previous value of theta are used to calculate the the new value
% of omega. However, the NEW value of omega is used to calculate the new
% theta
%
time(step+1) = time(step) + dt;
end;

plot(time,theta,'r' ); %plots the numerical solution in red
xlabel('time (seconds) ');
ylabel('theta (radians)');
```

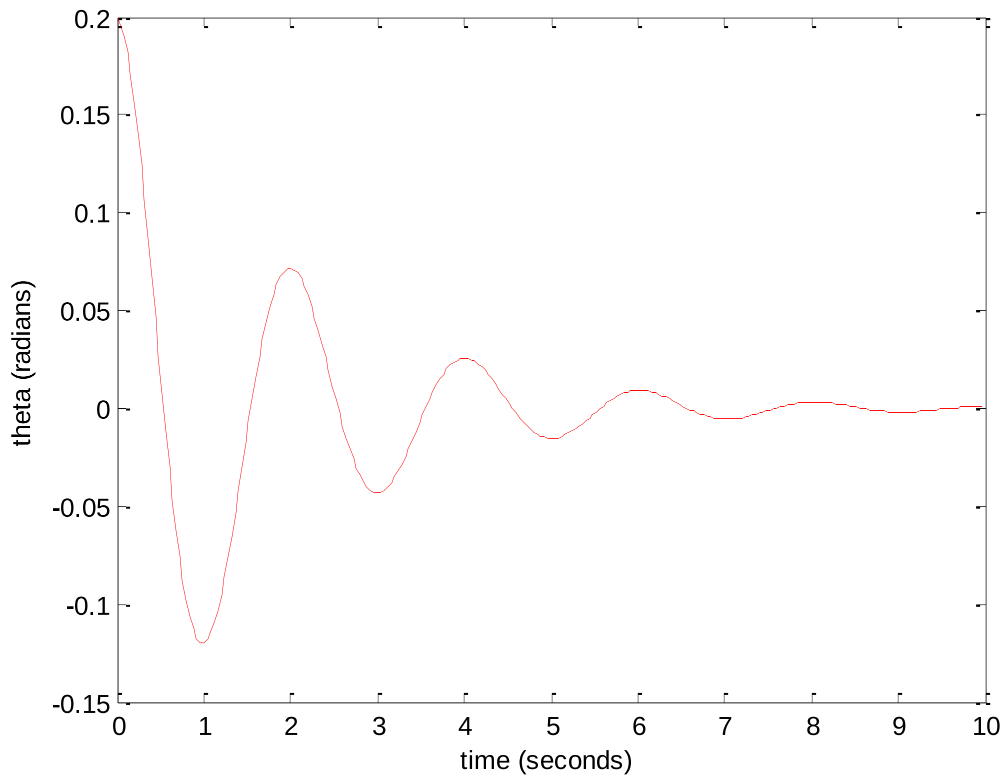


Figure 5. The damped pendulum using the Euler-Cromer method

3.3 Solution for a non-linear, damped, driven pendulum :- the Physical pendulum, using the Euler-Cromer method.

All of the next five plots were produced using the code below with slight modifications in either the input parameters or the plots.

```
%  Euler Cromer Solution for non-linear, damped, driven  pendulum
%  by Kevin Berwick,
%  based on 'Computational Physics' book by N Giordano and H Nakanishi,
%  section 3.3
%
clear;
length= 9.8;                        %pendulum length in metres
g=9.8;                               % acceleration due to gravity
q=0.5;

F_Drive=1.2;                        % damping strength
Omega_D=2/3;
npoints =15000;             %Discretize time
dt = 0.04;             % time step in seconds
 omega = zeros(npoints,1);   % initializes omega, a vector of dimension npoints X 1,to being all zeros
theta = zeros(npoints,1);   % initializes theta, a vector of dimension npoints X 1,to being all zeros
time = zeros(npoints,1);    % this initializes the vector time to being all zeros

theta(1)=0.2;               % you need to have some initial displacement, otherwise the pendulum will
not swing
omega(1)=0;

for step = 1:npoints-1;

% loop over the timesteps
% Note error in book in Equation for Example  3.3
omega(step+1)=omega(step)+(-(g/length)*sin(theta(step))-
q*omega(step)+F_Drive*sin(Omega_D*time(step)))*dt;
temporary_theta_step_plus_1 = theta(step)+omega(step+1)*dt;

% We need to adjust theta after each iteration so as to keep it between +/-pi
% The pendulum can now swing right around the pivot, corresponding to theta>2*pi.
% Theta is an angular variable so values of theta that differ by 2*pi correspond to the same position.
% For plotting purposes it is nice to keep (-pi<theta<pi).
% So, if theta is <-pi, add 2*pi.If theta is > pi, subtract 2*pi
% If the lines below between the ****** are commented out you get 3.6 (b)% bottom

 %*************************************************************************************
  if (temporary_theta_step_plus_1 < -pi)
      temporary_theta_step_plus_1= temporary_theta_step_plus_1+2*pi;
  elseif (temporary_theta_step_plus_1 > pi)
    temporary_theta_step_plus_1= temporary_theta_step_plus_1-2*pi;
 end;

%*************************************************************************************
 % Update theta array
      theta(step+1)=temporary_theta_step_plus_1;


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%


% In the Euler method, , the previous value of omega
% and the previous value of theta are used to calculate the  new values of omega and theta.
% In the Euler Cromer method, the previous value of omega
% and the previous value of theta are used to calculate the the new value
% of omega. However, the NEW value of omega is used to calculate the new
% theta
%
time(step+1) = time(step) + dt;
end;

plot (theta,omega,'r' ); %plots the numerical solution

xlabel('theta (radians)');
ylabel('omega (seconds)');
```

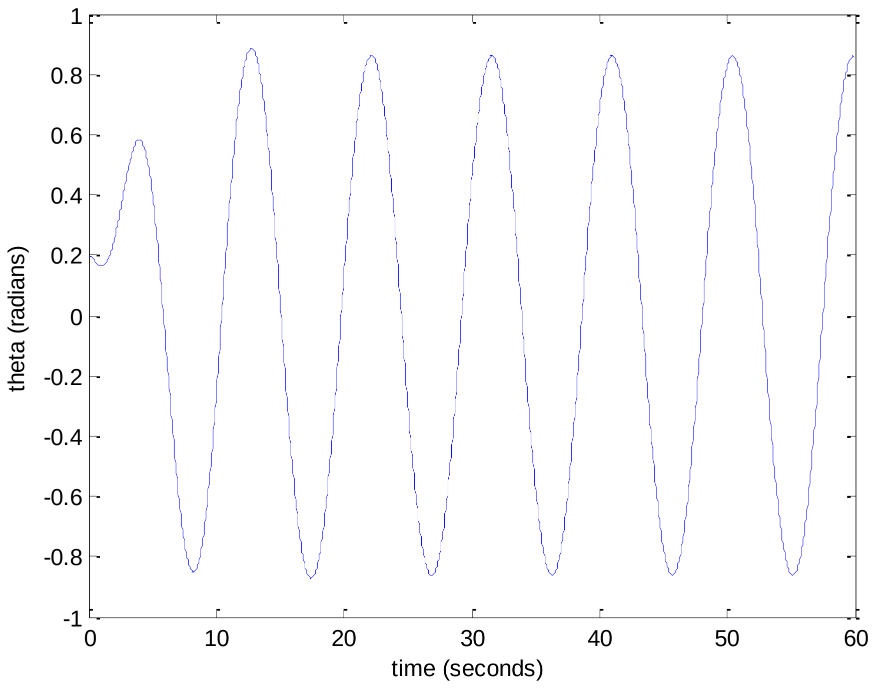


Figure 6. Results from Physical pendulum, using the Euler-Cromer method, F_drive =0.5

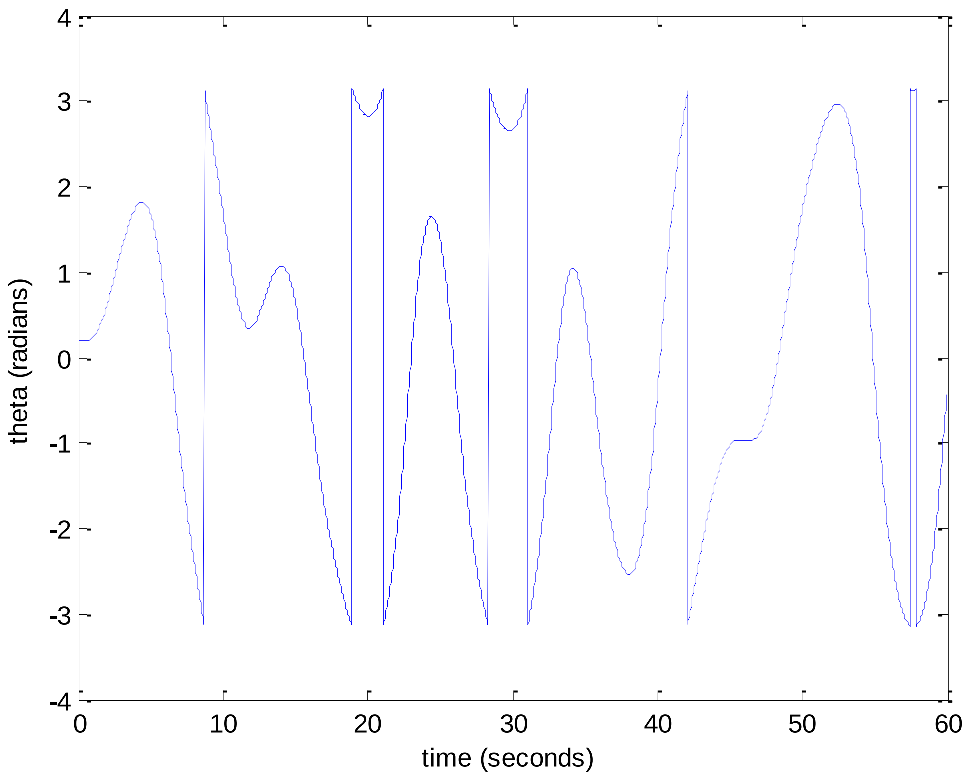


Figure 7.Results from Physical pendulum, using the Euler-Cromer method, F_drive =1.2

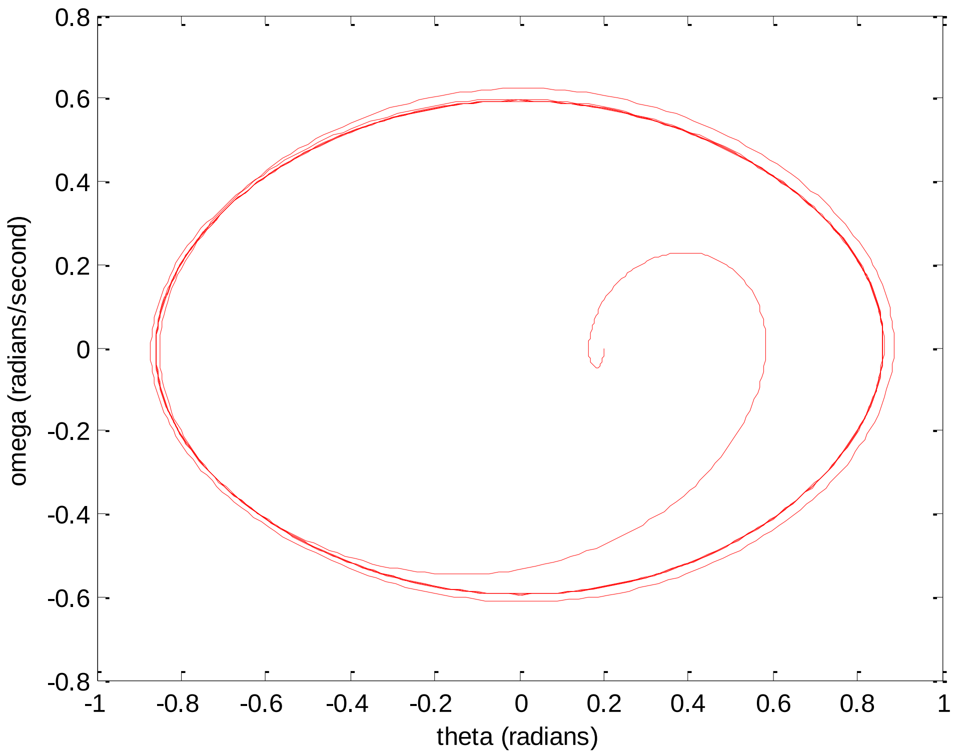


Figure 8. Results from Physical pendulum, using the Euler-Cromer method, F_drive =0.5

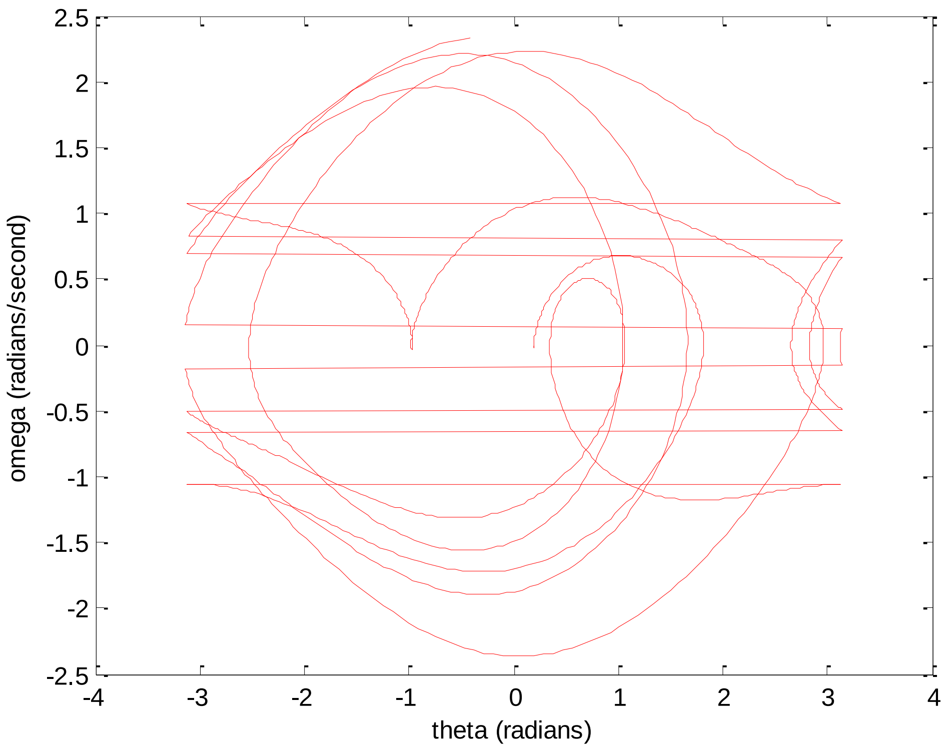


Figure 9. Results from Physical pendulum, using the Euler-Cromer method, F_Drive=1.2

 If you want higher resolution, simply increase the resolution by changing npoints. Note that this figure was produced using npoints = 15000. F_Drive =1.2.

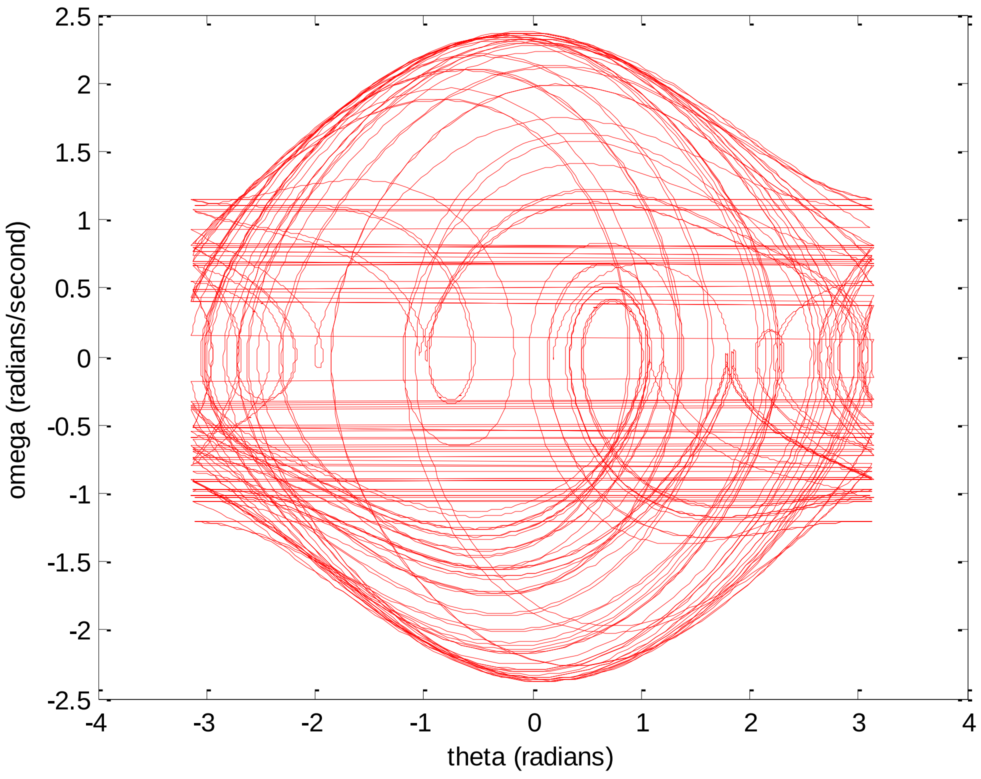


Figure 10. Increase resolution with npoints=15000.Results from Physical pendulum, using the Euler-Cromer method, F_Drive=1.2

```
%  Euler Cromer Solution for non-linear, damped, driven  pendulum
%  by Kevin Berwick,
%  based on 'Computational Physics' book by N Giordano and H Nakanishi,
%  section 3.3
%I modified the code in order to produce the Poincare section shown in Fig 3.9.
% It uses a little MATLAB trick in order to prevent plotting of any points that were not in
% phase with the driving force.

clear;
length= 9.8;                           %pendulum length in metres
g=9.8;                                  % acceleration due to gravity
q=0.5;
F_Drive=1.2;                          % damping strength
Omega_D=2/3;
npoints =1500000;               %Discretize time
dt = 0.04;                   % time step in seconds
 omega = zeros(npoints,1);   % initializes omega, a vector of dimension npoints X 1,to being all zeros
theta = zeros(npoints,1);   % initializes theta, a vector of dimension npoints X 1,to being all zeros
time = zeros(npoints,1);   % this initializes the vector time to being all zeros
 theta(1)=0.2;                  % you need to have some initial displacement, otherwise the pendulum will
not swing
omega(1)=0;

for step = 1:npoints-1;
```

```
% loop over the timesteps
% Note error in book in Equation for Example  3.3
omega(step+1)=omega(step)+(-(g/length)*sin(theta(step))-
q*omega(step)+F_Drive*sin(Omega_D*time(step)))*dt;
temporary_theta_step_plus_1 = theta(step)+omega(step+1)*dt;

  if (temporary_theta_step_plus_1 < -pi)
      temporary_theta_step_plus_1= temporary_theta_step_plus_1+2*pi;
  elseif (temporary_theta_step_plus_1 > pi)
    temporary_theta_step_plus_1= temporary_theta_step_plus_1-2*pi;
 end;

 % Update theta array
theta(step+1)=temporary_theta_step_plus_1;
time(step+1) = time(step) + dt;
end;

% Only plot omega and theta point when omega is in phase with the driving force Omega_D
I=find(abs(rem(time, 2*pi/Omega_D)) > 0.02);
omega(I)=NaN;
theta(I)=NaN;
scatter (theta,omega,2 ); %plots the numerical solution

plot (theta,omega,'k' ); %plots the numerical solution
xlabel('theta (radians)');
ylabel('omega (radians/second)');
```

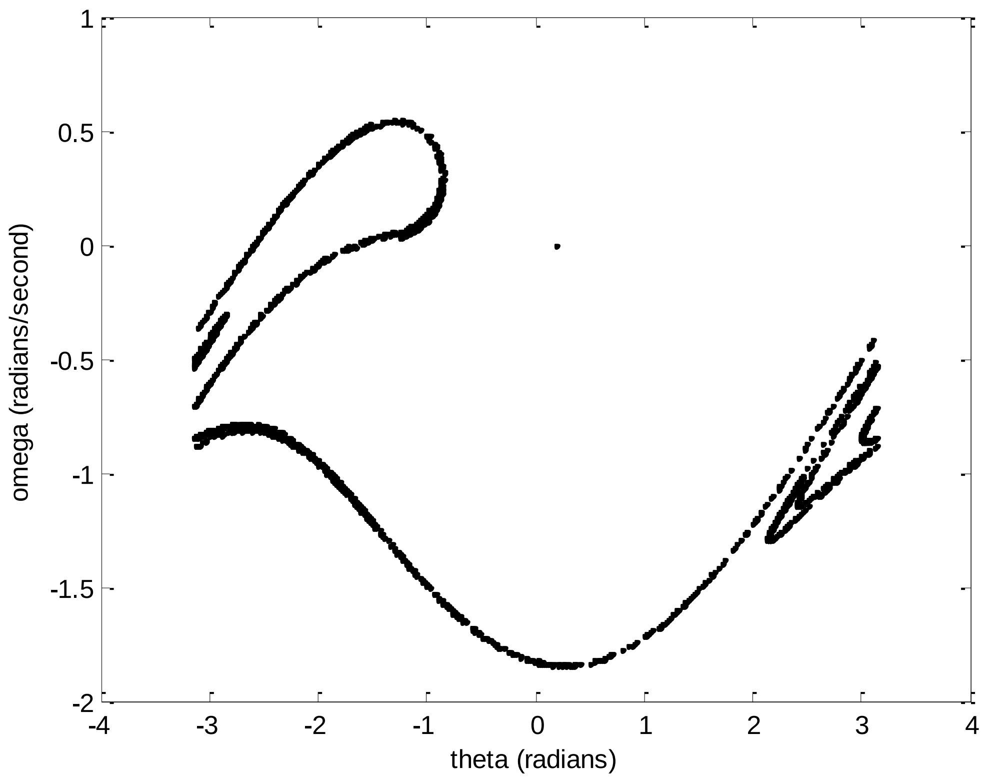


Figure 11. Poincare section (Strange attractor) Omega as a function of theta. F_Drive =1.2

3.4 Bifurcation diagram for the pendulum

```
% Program to perform Euler_cromer calculation of motion of physical pendulum
% by Kevin Berwick, and calculate the bifurcation diagram. You need to
% have the function 'pendulum_function' available in order to run this.
% based on 'Computational Physics' book by N Giordano and H Nakanishi,
% section 3.4.

Omega_D=2/3;
for F_Drive_step=1:0.1:13;
 F_Drive=1.35+F_Drive_step/100;
% Calculate the plot of theta as a function of time for the current drive step
% using the function :- pendulum_function
[time,theta]= pendulum_function(F_Drive, Omega_D);

%Filter the results to exclude initial transient of 300 periods, note
% that the period is 3*pi.

I=find (time< 3*pi*300);
time(I)=NaN;
theta(I)=NaN;

%Further filter the results so that only results in phase with the driving force
% F_Drive are displayed.
% Replace all those values NOT in phase with NaN

Z=find(abs(rem(time, 2*pi/Omega_D)) > 0.01);
time(Z)=NaN;
theta(Z)=NaN;

% Remove all NaN values from the array to reduce dataset size

time(isnan(time)) = [];
theta(isnan(theta)) = [];

% Now plot the results

plot(F_Drive,theta,'k');
hold on;
axis([1.35 1.5 1 3]);
xlabel('F Drive');
ylabel('theta (radians)');
end;


% Euler_cromer calculation of motion of physical pendulum
% by Kevin Berwick,
% based on 'Computational Physics' book by N Giordano and H Nakanishi,
% section 3.3

function [time,theta] = pendulum_function(F_Drive,Omega_D);

length= 9.8;                     %pendulum length in metres
g=9.8;                           % acceleration due to gravity
q=0.5;                          % damping strength
npoints =100000;             %Discretize time
dt = 0.04;                   % time step in seconds

omega = zeros(npoints,1);   % initializes omega, a vector of dimension npoints X 1,to being all zeros
theta = zeros(npoints,1);   % initializes theta, a vector of dimension npoints X 1,to being all zeros
```

```
time = zeros(npoints,1);    % this initializes the vector time to being all zeros

theta(1)=0.2;                 % you need to have some initial displacement, otherwise the pendulum will
not swing
omega(1)=0;

for step = 1:npoints-1;
% loop over the timesteps
omega(step+1)=omega(step)+(-(g/length)*sin(theta(step))-
q*omega(step)+F_Drive*sin(Omega_D*time(step)))*dt;
temporary_theta_step_plus_1 = theta(step)+omega(step+1)*dt;
 % Make corrections to keep theta between pi and -pi
if (temporary_theta_step_plus_1 < -pi)
        temporary_theta_step_plus_1= temporary_theta_step_plus_1+2*pi;
elseif (temporary_theta_step_plus_1 > pi)
     temporary_theta_step_plus_1= temporary_theta_step_plus_1-2*pi;
 end;
% Update theta array
theta(step+1)=temporary_theta_step_plus_1;
% Increment time
time(step+1) = time(step) + dt;
end;
```

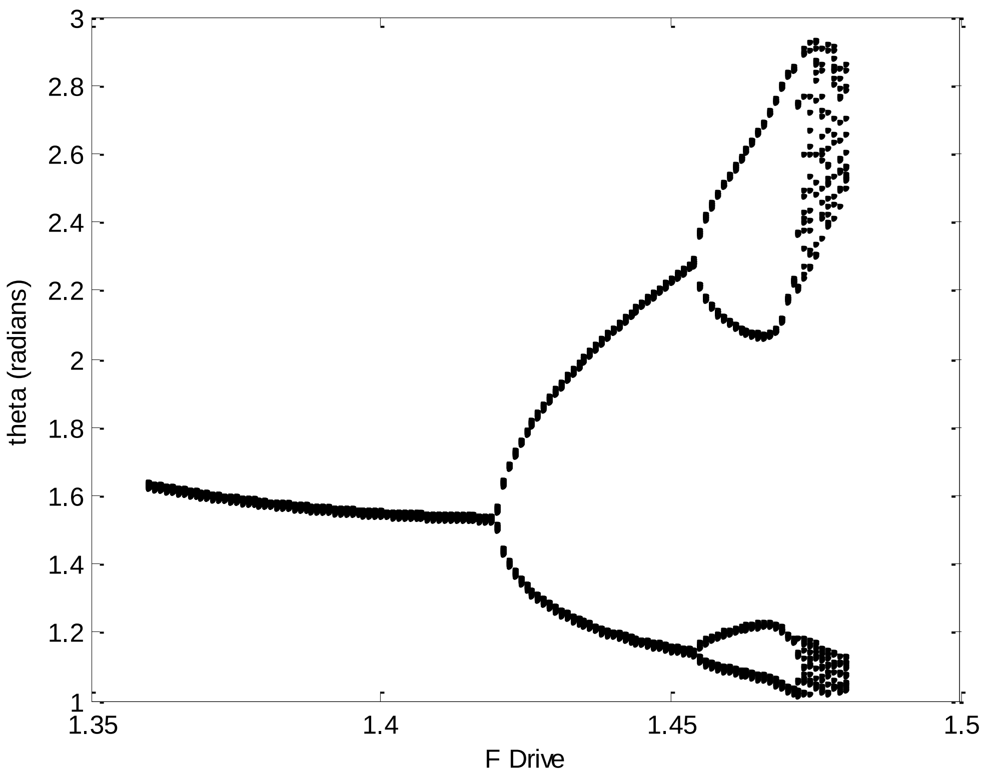


Figure 12. Bifurcation diagram for the pendulum

3.6 The Lorenz Model

The equations are the same as those as in 3.29

$$\frac{dx}{dt} = \sigma(y - x)$$

$$\frac{dy}{dt} = -xz + rx - y$$

$$\frac{dz}{dt} = xy - bz$$

The equations I used in the numerical solution are

$$x_{i+1} = x_i + \sigma(y_i - x_i)\Delta t$$

$$y_{i+1} = y_i + (-x_i\, z_i + rx_i - y_i)\Delta t$$

$$z_{i+1} = z_i + (\,x_i\, y_i - bz_i)\Delta t$$

```
%
%  Euler calculation of Lorenz equations
%  by Kevin Berwick,
%  based on 'Computational Physics' book by N Giordano and H Nakanishi,
%  section 3.6
%
clear
a=10;
b=8/3;
r=25;
sigma=10;
npoints =500000;            %Discretize time
dt = 0.0001;             % time step in seconds
x = zeros(npoints,1);   % initializes x, a vector of dimension npoints X 1,to being all zeros
y = zeros(npoints,1);   % initializes y, a vector of dimension npoints X 1,to being all zeros
z = zeros(npoints,1);   % initializes z, a vector of dimension npoints X 1,to being all zeros
time = zeros(npoints,1);   % this initializes the vector time to being all zeros
x(1)=1;

for step = 1:npoints-1

% loop over the timesteps and solve the difference equations

x(step+1)=x(step)+sigma*(y(step)-x(step))*dt;
y(step+1)=y(step)+(-x(step)*z(step)+r*x(step)-y(step))*dt;
z(step+1)=z(step)+(x(step)*y(step)-b*z(step))*dt;

  % Update time array

time(step+1) = time(step) + dt;
end;

subplot (2,1,1);
```

```
plot(time,z,'b' );
xlabel('time');
ylabel('z');
subplot (2,1,2);
plot (x,z,'g' );
xlabel('x');
ylabel('z')
```

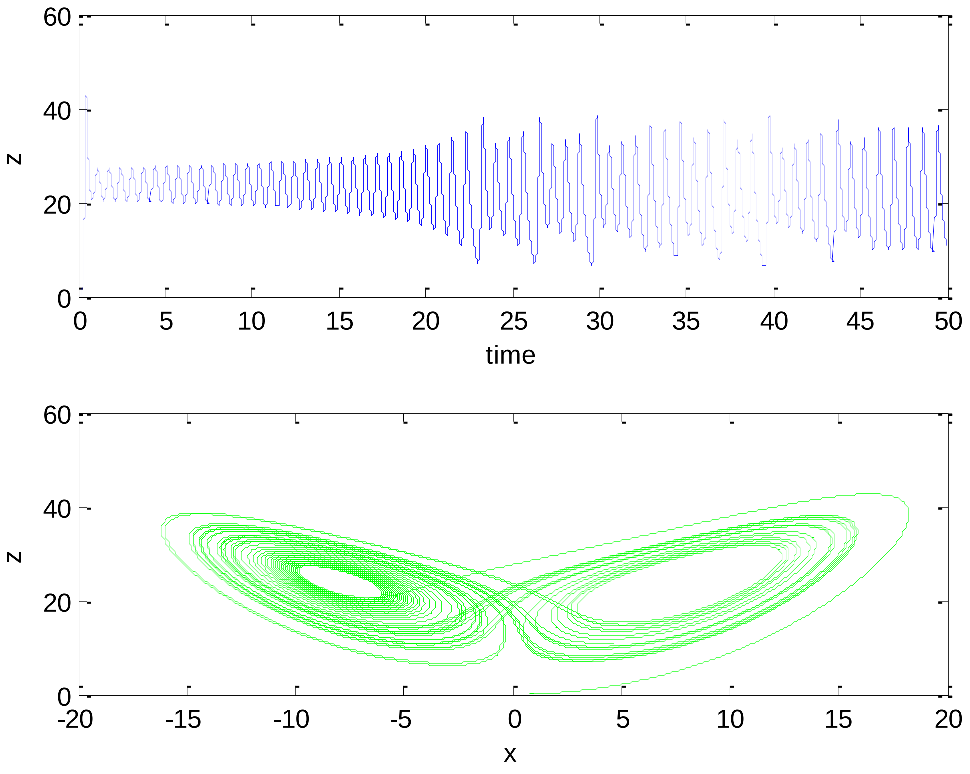


Figure 13. Variation of z as a function of time and corresponding strange attractor

4. The Solar System

4.1 Kepler's Laws

```
%
%  Planetary orbit using Euler Cromer methods.
%  by Kevin Berwick,
%  based on 'Computational Physics' book by N Giordano and H Nakanishi
%  Section 4.1
%

npoints=500;
dt = 0.002;                  % time step in years

x=1;                 %  initialise position of planet in AU
y=0;
v_x=0;                 % initialise velocity of planet in AU/yr
v_y=2*pi;

% Plot the Sun at the origin
plot(0,0,'oy','MarkerSize',30, 'MarkerFaceColor','yellow');
axis([-1 1 -1 1]);
xlabel('x(AU)');
ylabel('y(AU)');
hold on;

for step = 1:npoints-1;
% loop over the timesteps
radius=sqrt(x^2+y^2);
% Compute new velocities in the x and y directions
v_x_new=v_x - (4*pi^2*x*dt)/(radius^3);
v_y_new=v_y - (4*pi^2*y*dt)/(radius^3);

% Euler Cromer Step - update positions using newly calculated velocities

x_new=x+v_x_new*dt;
y_new=y+v_y_new*dt;

% Plot planet position immediately
 plot(x_new,y_new,'-k');
 drawnow;

 % Update x and y velocities  with new velocities
v_x=v_x_new;
v_y=v_y_new;
% Update x and y with new positions
x=x_new;
y=y_new;

end;
```

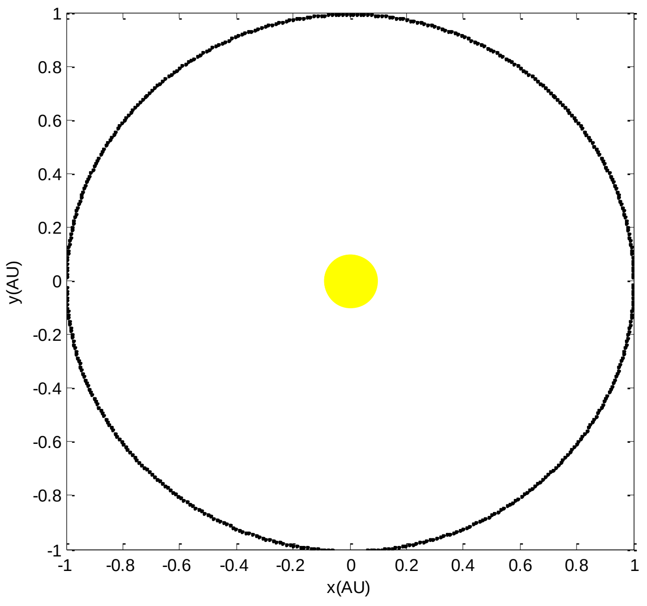


Figure 14. Simulation of Earth orbit around the Sun

Here is the code using a second order Runge Kutta method giving the same results.

```
%
%
%  Planetary orbit using second order Runge-Kutta method.
%  by Kevin Berwick,
%  based on 'Computational Physics' book by N Giordano and H Nakanishi
%  Section 4.1
%
%
npoints=500;
dt = 0.002;                   % time step in years
t=0;
x=1;                 %  initialise position of planet in AU
y=0;
v_x=0;                 % initialise x velocity of planet in AU/yr
v_y=2*pi;              % initialise y velocity of planet in AU/yr

% Plot the Sun at the origin
plot(0,0,'oy','MarkerSize',30, 'MarkerFaceColor','yellow');
axis([-1 1 -1 1]);
xlabel('x(AU)');
ylabel('y(AU)');
hold on;

for step = 1:npoints-1;

 % loop over the timesteps
radius=sqrt(x^2+y^2);

% Compute Runge Kutta values for the y equations

y_dash=y+0.5*v_y*dt;
v_y_dash=v_y - 0.5*(4*pi^2*y*dt)/(radius^3);

% update positions and new  y velocity
```

```
y_new=y+v_y_dash*dt;
v_y_new=v_y-(4*pi^2*y_dash*dt)/(radius^3);

% Compute Runge Kutta values for the x equations
x_dash=x+0.5*v_x*dt;
v_x_dash=v_x - 0.5*(4*pi^2*x*dt)/(radius^3);

% update positions using newly calculated velocity

x_new=x+v_x_dash*dt;
v_x_new=v_x-(4*pi^2*x_dash*dt)/(radius^3);
% Plot planet position immediately
 plot(x_new,y_new,'-k');
 drawnow;
 % Update x and y velocities  with new velocities
v_x=v_x_new;
v_y=v_y_new;
% Update x and y with new positions
x=x_new;
y=y_new;

end;
```

4.1.1 Ex 4.1 Planetary motion results using different time steps

In Exercise 4.1 we are asked to change the time step to show that for dt > 0.01 years, you get an unsatisfactory result. I chose dt=0.05 and got the Figure below. Clearly the orbit is unstable. This is in accordance with the rule of thumb that the time step should be less than 1% of the characteristic time scale of the problem.

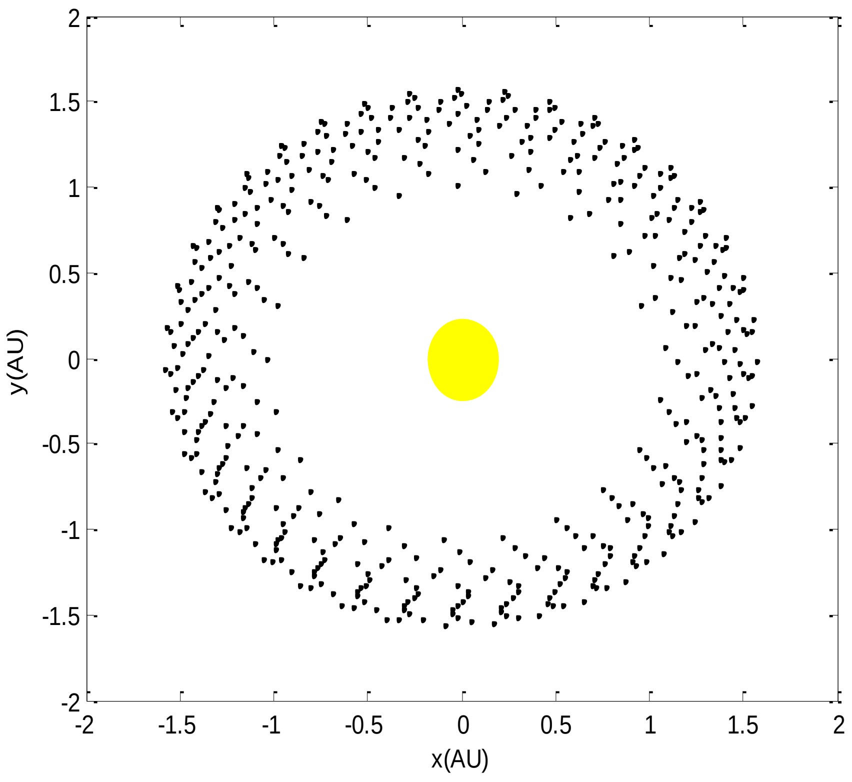


Figure 15. Simulation of Earth orbit with time step of 0.05

I also looked at changing the velocity to look at the effect of increasing the value of the initial velocity, while returning the time step to 0.002. Here is the plot, below, with an initial y velocity of 4, dt is 0.002.

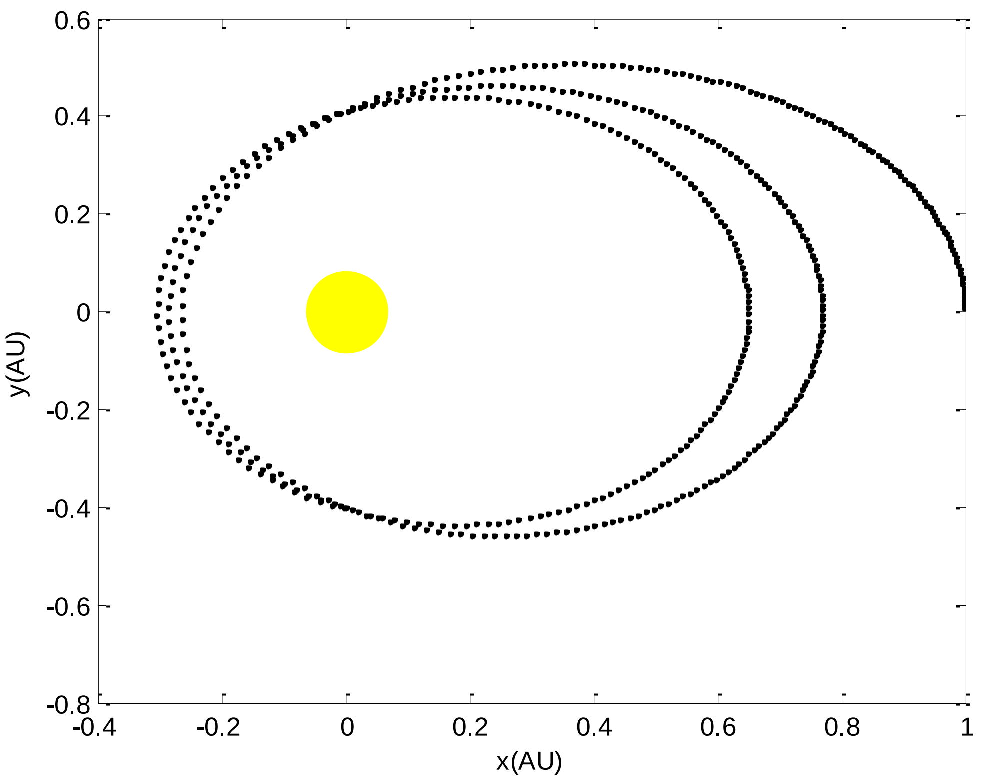


Figure 16. Simulation of Earth orbit, initial y velocity of 4, time step is 0.002.

Here is the plot with the same initial y velocity of 4, but dt is increased to 0.05. Clearly, the instability is apparent.

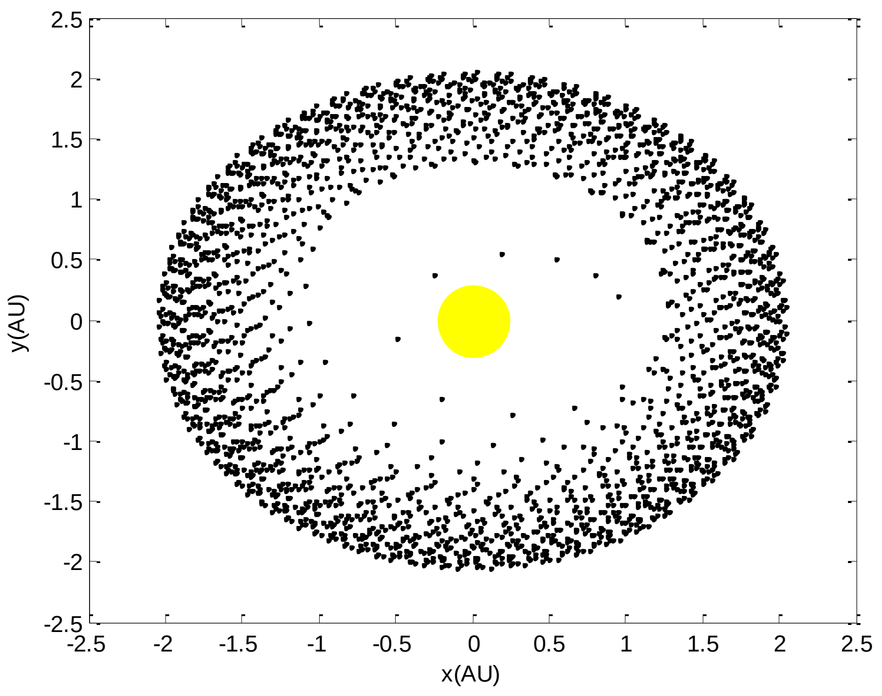


Figure 17.Simulation of Earth orbit, initial y velocity of 4, time step is 0.05

Here is the result for an initial y velocity of 8, dt is 0.002., npoints=2500. The Runge Kutta Method is used here. Note the relative stability of the orbit.

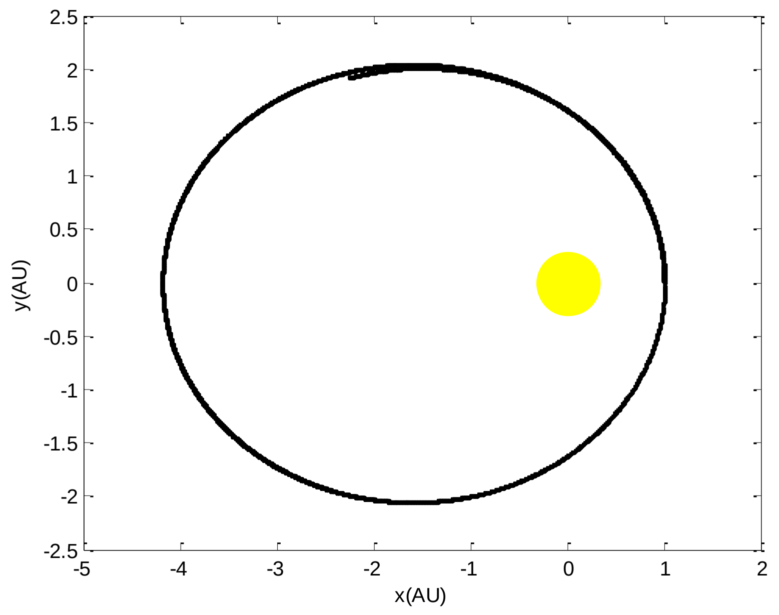


Figure 18. Simulation of Earth orbit, initial y velocity of 8, time step is 0.002. 2500 points and Runge Kutta method

Here is the code and Plot for an initial y velocity of 8, dt is 0.05, npoints=2500. The Runge Kutta Method is used here.

```
%
%
%  Planetary orbit using second order Runge-Kutta method.
%  by Kevin Berwick,
%  based on 'Computational Physics' book by N Giordano and H Nakanishi
%  Section 4.1
%
%
% npoints=500;
npoints=2500;
dt = 0.05;                  % time step in years

t=0;
x=1;                  %  initialise position of planet in AU
y=0;
v_x=0;                  % initialise x velocity of planet in AU/yr
% v_y=2*pi;             % initialise y velocity of planet in AU/yr
v_y=8;           % initialise y velocity of planet in AU/yr

% Plot the Sun at the origin
plot(0,0,'oy','MarkerSize',30, 'MarkerFaceColor','yellow');
% axis([-1 1 -1 1]);    remove in order to see effect of changing time step
xlabel('x(AU)');
ylabel('y(AU)');
hold on;

for step = 1:npoints-1;

 % loop over the timesteps
radius=sqrt(x^2+y^2);

% Compute Runge Kutta values for the y equations

y_dash=y+0.5*v_y*dt;
v_y_dash=v_y - 0.5*(4*pi^2*y*dt)/(radius^3);

% update positions and new  y velocity

y_new=y+v_y_dash*dt;
v_y_new=v_y-(4*pi^2*y_dash*dt)/(radius^3);

% Compute Runge Kutta values for the x equations
x_dash=x+0.5*v_x*dt;
v_x_dash=v_x - 0.5*(4*pi^2*x*dt)/(radius^3);

% update positions using newly calculated velocity

x_new=x+v_x_dash*dt;
v_x_new=v_x-(4*pi^2*x_dash*dt)/(radius^3);
% Plot planet position immediately
 plot(x_new,y_new,'-k');
 drawnow;
 % Update x and y velocities  with new velocities
v_x=v_x_new;
v_y=v_y_new;
% Update x and y with new positions
x=x_new;
y=y_new;

end;
```

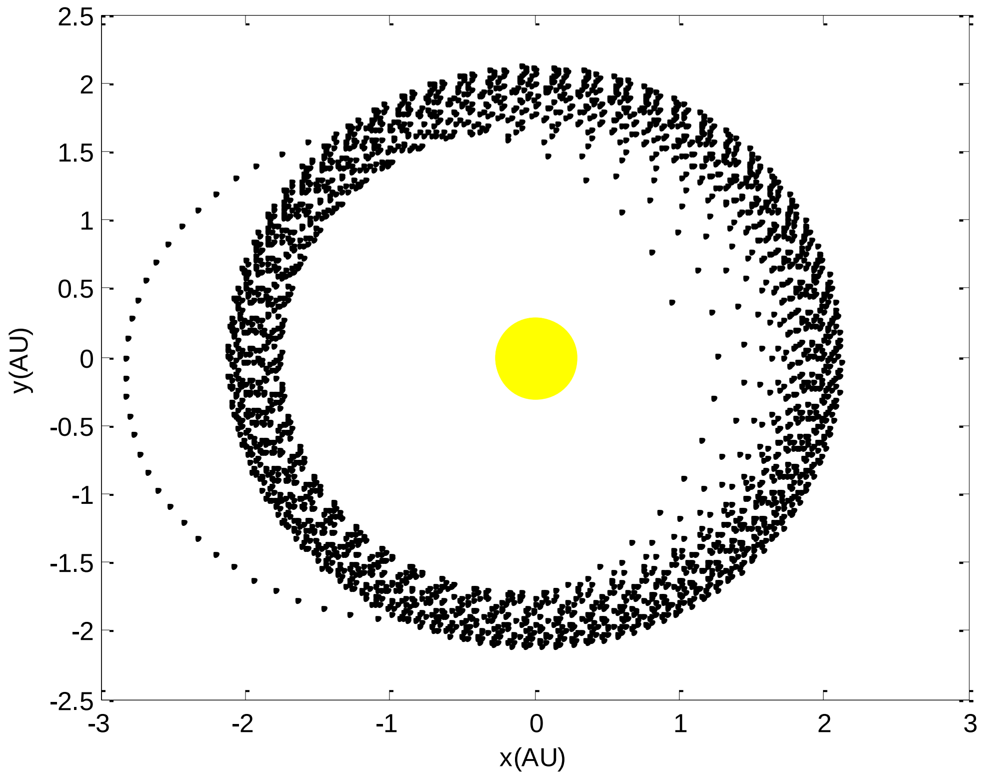


Figure 19.Plot for an initial y velocity of 8, dt is 0.05, npoints=2500. The Runge Kutta Method is used here

4.2 Orbits using different force laws

Here is the code to calculate the elliptical orbit for a force law with β=2. The time step is 0.001 years.

```
%
%
% Planetary orbit using second order Runge-Kutta method.
% by Kevin Berwick,
% based on 'Computational Physics' book by N Giordano and H Nakanishi
% Section 4.2
%
%

npoints=1000;
dt = 0.001;                    % time step in years

t=0;
x=1;                  % initialise position of planet in AU
y=0;
v_x=0;                  % initialise x velocity of planet in AU/yr
% v_y=2*pi;             % initialise y velocity of planet in AU/yr
v_y=5;              % initialise y velocity of planet in AU/yr

% Plot the Sun at the origin
```

```
plot(0,0,'oy','MarkerSize',30, 'MarkerFaceColor','yellow');
title('Beta = 2')
 axis([-1 1 -1 1]);
xlabel('x(AU)');
ylabel('y(AU)');
hold on;

for step = 1:npoints-1;

 % loop over the timesteps
radius=sqrt(x^2+y^2);

% Compute Runge Kutta values for the y equations

y_dash=y+0.5*v_y*dt;
v_y_dash=v_y - 0.5*(4*pi^2*y*dt)/(radius^3);

% update positions and new  y velocity

y_new=y+v_y_dash*dt;
v_y_new=v_y-(4*pi^2*y_dash*dt)/(radius^3);

% Compute Runge Kutta values for the x equations
x_dash=x+0.5*v_x*dt;
v_x_dash=v_x - 0.5*(4*pi^2*x*dt)/(radius^3);

% update positions using newly calculated velocity

x_new=x+v_x_dash*dt;
v_x_new=v_x-(4*pi^2*x_dash*dt)/(radius^3);
% Plot planet position immediately
 plot(x_new,y_new,'-k');
 drawnow;
 % Update x and y velocities  with new velocities
v_x=v_x_new;
v_y=v_y_new;
% Update x and y with new positions
x=x_new;
y=y_new;

end;
```

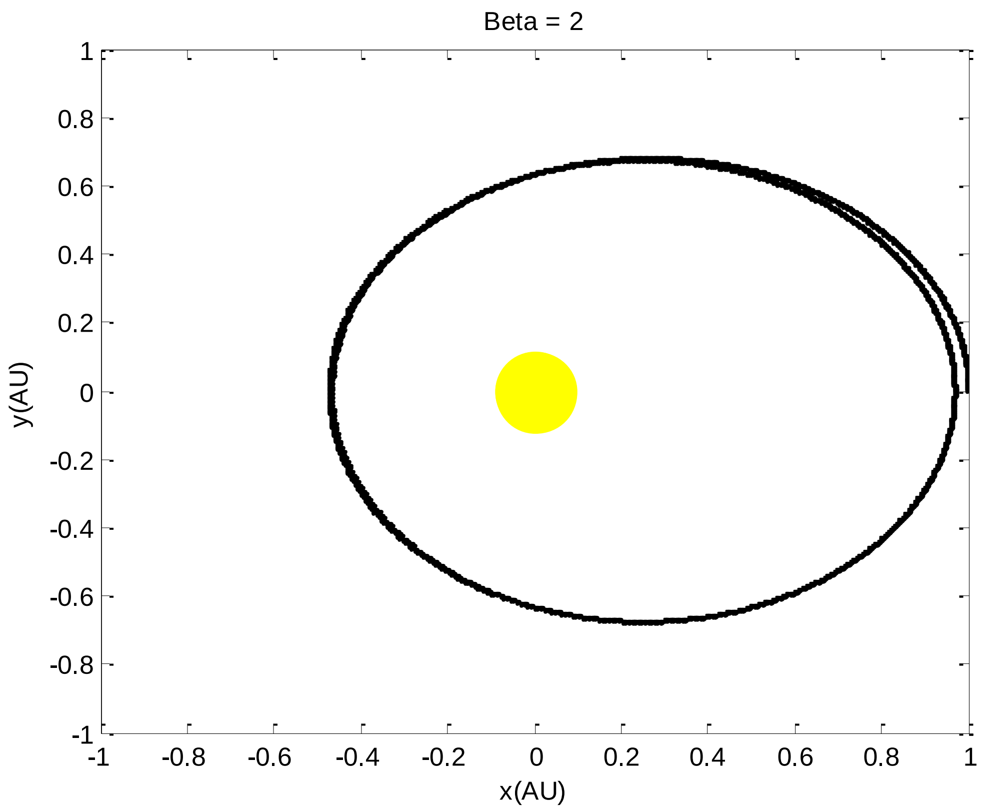


Figure 20. Orbit for a force law with β=2. The time step is 0.001 years.

Here is the code to calculate the elliptical orbit for a force law with β=2.5. The time step is 0.001 years.

```
% Planetary orbit using second order Runge-Kutta method.
% by Kevin Berwick,
% based on 'Computational Physics' book by N Giordano and H Nakanishi
% Section 4.2
%
%

npoints=1000;
dt = 0.001;                    % time step in years

t=0;
x=1;                    % initialise position of planet in AU
y=0;
v_x=0;                  % initialise x velocity of planet in AU/yr
% v_y=2*pi;             % initialise y velocity of planet in AU/yr
v_y=5;          % initialise y velocity of planet in AU/yr

% Plot the Sun at the origin
plot(0,0,'oy','MarkerSize',30, 'MarkerFaceColor','yellow');
title('Beta = 2.5')
 axis([-1 1 -1 1]);
xlabel('x(AU)');
ylabel('y(AU)');
hold on;

for step = 1:npoints-1;
```

```
 % loop over the timesteps
radius=sqrt(x^2+y^2);

% Compute Runge Kutta values for the y equations

y_dash=y+0.5*v_y*dt;
v_y_dash=v_y - 0.5*(4*pi^2*y*dt)/(radius^3.5);

% update positions and new  y velocity

y_new=y+v_y_dash*dt;
v_y_new=v_y-(4*pi^2*y_dash*dt)/(radius^3.5);

% Compute Runge Kutta values for the x equations
x_dash=x+0.5*v_x*dt;
v_x_dash=v_x - 0.5*(4*pi^2*x*dt)/(radius^3.5);

% update positions using newly calculated velocity

x_new=x+v_x_dash*dt;
v_x_new=v_x-(4*pi^2*x_dash*dt)/(radius^3.5);
% Plot planet position immediately
 plot(x_new,y_new,'-k');
 drawnow;
 % Update x and y velocities  with new velocities
v_x=v_x_new;
v_y=v_y_new;
% Update x and y with new positions
x=x_new;
y=y_new;

end;
```

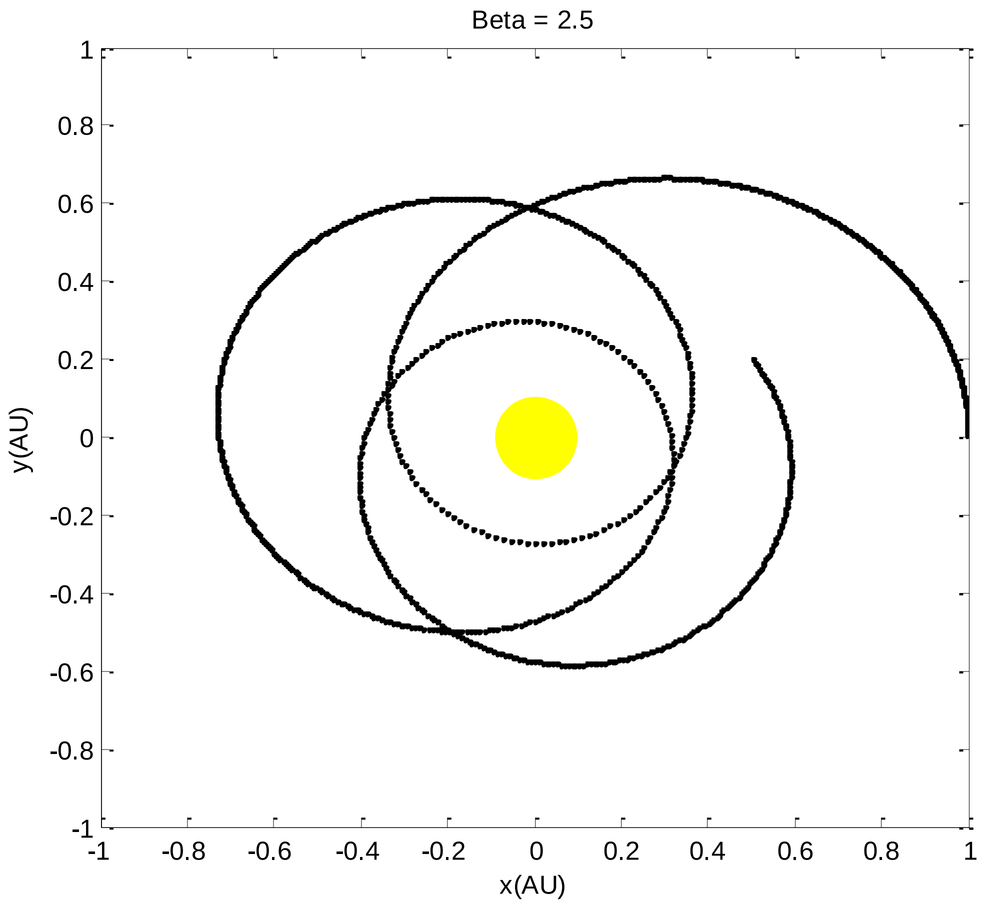


Figure 21. Orbit for a force law with β=2.5. The time step is 0.001 years.

Here is the Figure for β=3. Check out the planet being ejected from the solar system!! The Sun is at the origin

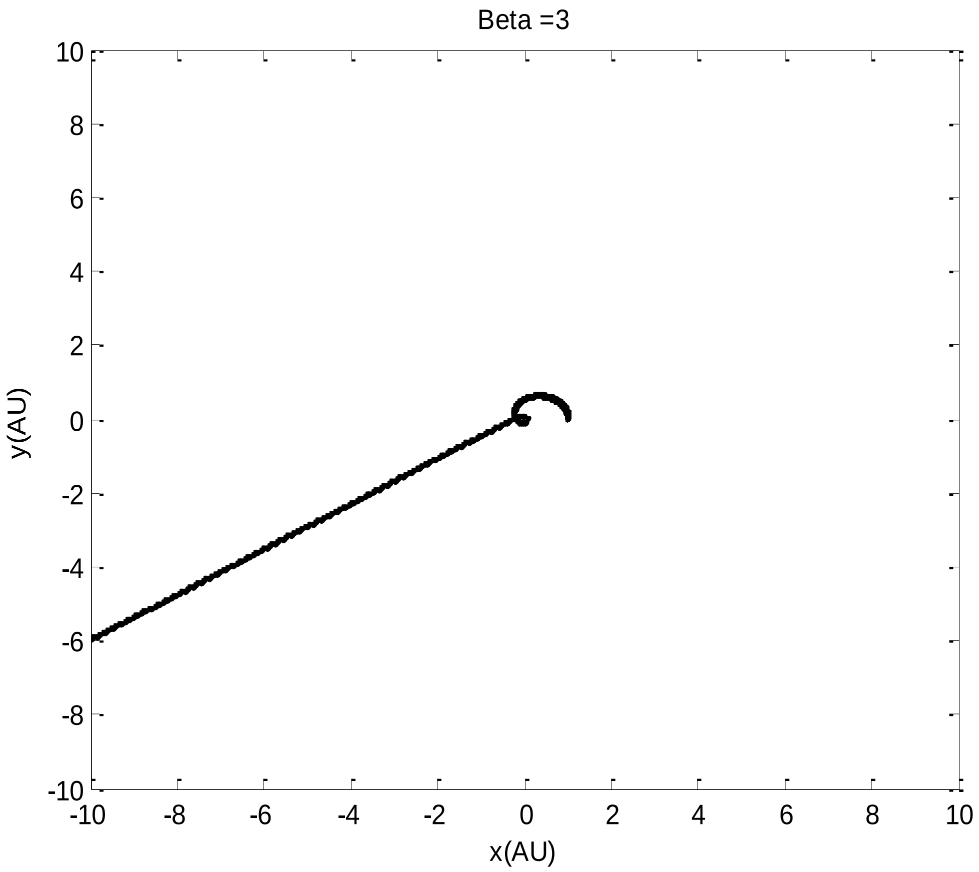


Figure 22. Orbit for a force law with β=3.

4.3 Precession of the perihelion of Mercury.

Let’s do the Maths here.

$$F_G = \frac{GM_sM_e}{r^2}\left(1+\frac{\alpha}{r^2}\right) \qquad so \qquad \frac{d^2x}{dt^2} = \frac{F_{G,x}}{M_E} \qquad \frac{d^2y}{dt^2} = \frac{F_{G,y}}{M_E}$$

$$F_{G,x} = -\frac{GM_sM_e}{r^2}\left(1+\frac{\alpha}{r^2}\right)cos\theta \qquad From\ the\ diagram\ \ cos\theta = \frac{x}{r}\ \ so$$

$$F_{G,x} = -\frac{GM_sM_ex}{r^3}\left(1+\frac{\alpha}{r^2}\right)$$

$$F_{G,y} = -\frac{GM_sM_e}{r^2}\left(1+\frac{\alpha}{r^2}\right)\ sin\theta \qquad From\ the\ diagram\ \ sin\theta = \frac{y}{r}\ \ so$$

$$F_{G,y} = -\frac{GM_sM_ey}{r^3}\left(1+\frac{\alpha}{r^2}\right)$$

Now, write each 2nd order differential equations as two, first order, differential equations.

$$\frac{dv_x}{dt} = -\frac{GM_s\,x}{r^3}\left(1+\frac{\alpha}{r^2}\right)$$

$$\frac{dx}{dt} = v_x$$

$$\frac{dv_y}{dt} = -\frac{GM_s\,y}{r^3}\left(1+\frac{\alpha}{r^2}\right)$$

$$\frac{dy}{dt} = v_y$$

So, the difference equation set using the Euler Cromer method is

$$v_{x,i+1} = v_{x,i} - \frac{4\pi^2 x_i}{r_i^3}\Delta t\left(1+\frac{\alpha}{r_i^2}\right)$$

$$x_{i+1} = x_i + v_{x,i+1}\,\Delta t$$

$$v_{y,i+1} = v_{y,i} - \frac{4\pi^2 y_i}{r_i^3}\Delta t\left(1+\frac{\alpha}{r_i^2}\right)$$

$$y_{i+1} = y_i + v_{y,i+1}\,\Delta t$$

We could go ahead and code this, but what about if we chose to attack the problem using the Runge Kutta method. The relevant equations are

$$y' = y_i + v_{y,i}\,\frac{\Delta t}{2}$$

$$v_y' = v_{y,i} - \frac{4\pi^2 y_i}{r_i^3}\frac{\Delta t}{2}\left(1+\frac{\alpha}{r_i^2}\right)$$

$$y_{i+1} = y_i + v_y'\,\Delta t$$

$$v_{y,i+1} = v_{y,i} - \frac{4\pi^2 y_i}{r_i^3}\Delta t\left(1+\frac{\alpha}{r_i^2}\right)$$

$$x' = x_i + v_{x,i}\,\frac{\Delta t}{2}$$

$$v_x' = v_{x,i} - \frac{4\pi^2 x_i}{r_i^3}\frac{\Delta t}{2}\left(1+\frac{\alpha}{r_i^2}\right)$$

$$x_{i+1} = x_i + v_x'\,\Delta t$$

$$v_{x,i+1} = v_{x,i} - \frac{4\pi^2 x_i}{r_i^3}\Delta t\left(1+\frac{\alpha}{r_i^2}\right)$$

In the case of the y equations for example, y' and v' is evaluated by the Euler method at $\frac{\Delta t}{2}$. Then to get the new values of y and v, we simply use the Euler method but using y' and v' in the equations.

So, here is the code for an alpha value of 0.0008

```
%
%
%  Precession of mercury using second order Runge-Kutta method.
%  by Kevin Berwick,
%  based on 'Computational Physics' book by N Giordano and H Nakanishi
%  Section 4.3
%
%

npoints=30000;
dt = 0.0001;                % time step in years
time = zeros(npoints,1);  % initializes time, a vector of dimension npoints X 1,to being all zeros
angleInDegrees = zeros(npoints,1);  % initializes angleInDegrees, a vector of dimension npoints X 1,to
being all zeros
x=0.47;                 %  initialise x position of planet in AU
y=0;                    %  initialise x position of planet in AU
v_x=0;                  % initialise x velocity of planet in AU/yr
v_y=8.2;                 % initialise y velocity of planet in AU/yr

alpha=0.0008;

for step = 1:npoints-1;       % loop over the timesteps

   time(step+1) = time(step) + dt;             % Increment total elapsed time

   radius=sqrt(x^2+y^2);      % Calculate radius

   relativity_factor=1+alpha/radius^2;

   % Compute Runge Kutta values for the y equations

   y_dash=y+0.5*v_y*dt;
   v_y_dash=v_y - 0.5*(4*pi^2*y*dt)*relativity_factor/(radius^3);

   % Update positions and new y velocity

   y_new=y+v_y_dash*dt;
   v_y_new=v_y-(4*pi^2*y_dash*dt)*relativity_factor/(radius^3);

   % Compute Runge Kutta values for the x equations

   x_dash=x+0.5*v_x*dt;
   v_x_dash=v_x - 0.5*(4*pi^2*x*dt)*relativity_factor/(radius^3);

   % Update positions using newly calculated velocity

   x_new=x+v_x_dash*dt;
   v_x_new=v_x-(4*pi^2*x_dash*dt)*relativity_factor/(radius^3);

   % Update x and y velocities  with new velocities
   v_x=v_x_new;
   v_y=v_y_new;

   % Identify semi-major axes in the planetary orbit and draw them on the
   % plot. I need to monitor the time derivative of the radius and identify when it
   % changes from positive to negative.  Then calculate the angle made
   % by the vector joining the origin and this point with the x axis.

   new_radius=sqrt(x_new^2+y_new^2);
   time_derivative=(new_radius-radius)/dt;
   %  Update x and y with new positions
   x=x_new;
   y=y_new;


   if abs(time_derivative) <0.0025;    % This is a way of identifying the long axis of the orbit. Note that
if this is not the case,the value
                                         % angle_In_Degrees will remain zero

        [theta,rho] = cart2pol(x_new,y_new);      %Convert Cartesian coordinates to polar,  noting that
the result is in radians
        angleInDegrees(step)= 180*(theta/pi);  % convert to degrees

   end;

  end;

% Plot Orbit orientation versus time. Remove data with angles = zero or
% less,  this means we only plot the angles of the long axes of the orbit

I=find(angleInDegrees < 0.01);
time(I)=NaN;
angleInDegrees(I)=NaN;

% Remove all NaN values from the array to reduce dataset size

time(isnan(time)) = [];
angleInDegrees(isnan(angleInDegrees)) = [];

  axis([0 3 0 20]);
  xlabel('time(year)');
  ylabel('theta(degrees)');
  hold on;
  scatter (time, angleInDegrees, 'or');

 % Perform a linear fit to the data, degree N=1,
 % returning the coefficient, or slope, to the variable slope

poly_matrix = polyfit(time,angleInDegrees,1) ;
slope=poly_matrix(1);
title(['Orbit orientation versus time for alpha=',num2str(alpha), ' and slope = ', num2str(slope)]);

% Plot the fit

time_for_fit=[0:0.1:3];
Polynomial_values = polyval(poly_matrix,time_for_fit);      % Evaluate the polynomial at times from
0 to 2.5
plot(time_for_fit,Polynomial_values, 'g', 'LineWidth',2);
```

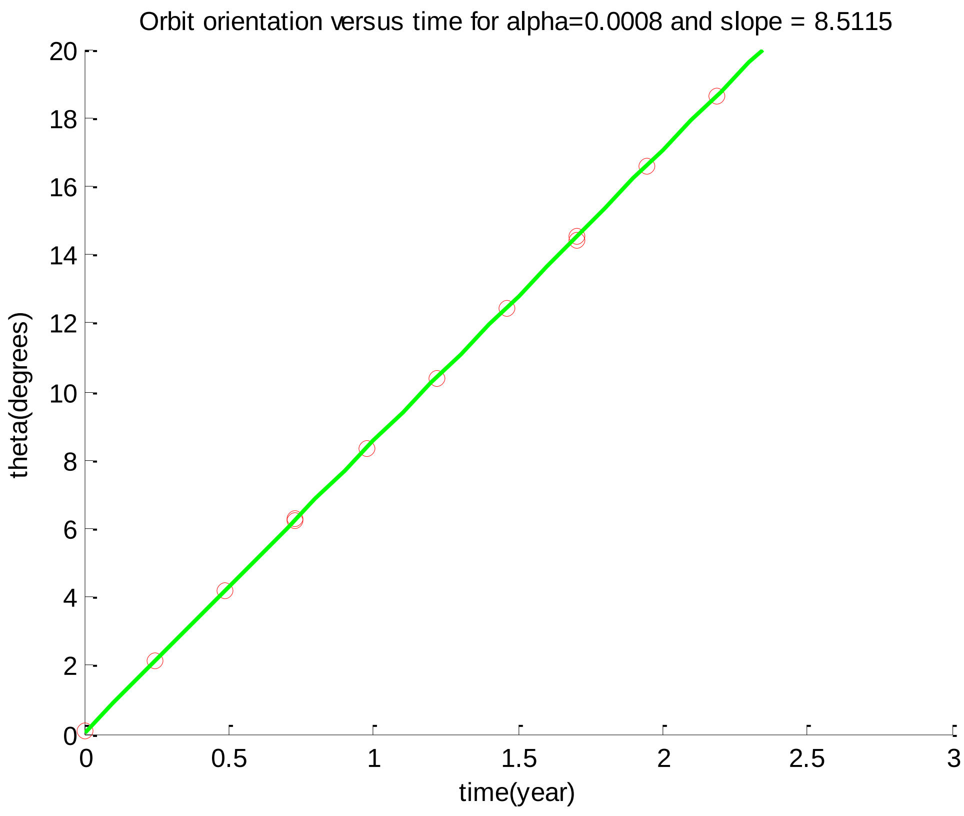


Figure 23. Orbit orientation as a function of time

If we rerun the code for various values of alpha. All we do is change one line in the code above, circled red. We note the value of the slope each time. The slope is just the time derivative of theta. We get the following values.

alpha	**Time derivative of theta (precession rate)**
0.0005	5.3
0.0007	7.4
0.001	10.7
0.002	21.9
0.003	33.6
0.004	45.9

We can use this next script to plot this data and then do a fit to finally calculate the precession rate of Mercury.

```
%
%
%  Precession of mercury using second order Runge-Kutta method.
%  Data plotting and fitting routines
%  by Kevin Berwick,
%  based on 'Computational Physics' book by N Giordano and H Nakanishi
%  Section 4.3
%
%
alphа_relativity=1.1e-8;       % predicted alpha value from General Relativity
                                                                                    % Load up data
alpha=[0.0005 0.0007 0.001 0.002 0.003 0.004];
precession_rate=[5.3 7.4 10.7 21.9 33.6 45.9];
                                                                                    % Format graph
axis([0 0.004 0 40]);
xlabel('alpha');
ylabel('Precession rate (degrees/year)');
hold on;
                                                                                     % Plot graph
scatter(alpha, precession_rate,  'ko')


%  Perform a linear fit to the data, degree N=1,
%  returning the coefficient, or slope. Note you can't use the MATLAB function polyval as the
%  intercept value of the fitted line would dominate the precession rate.
%
poly_matrix = polyfit(alpha, precession_rate, 1); % Perform the fit
                                                  % Plot the fit on the data
alpha_for_fit=[0:0.0001:0.004];
Polynomial_values = polyval(poly_matrix,alpha_for_fit);      % Evaluate the polynomial at points in
the vector
plot(alpha_for_fit,Polynomial_values, 'g', 'LineWidth',2);
;
Mercury_rate = poly_matrix(1)*alpha_relativity; % Extract the slope from the fit and multiply it by
the predicted alpha
                                                  % value from General Relativity. Answer is in degrees
per year

 Mercury_rate_arc_sec_century =  Mercury_rate*100 *3600;    % Convert to arc/s per century

title(['Calculated precession rate of Mercury for alpha = ', num2str(alpha_relativity),' AU^2 is
',num2str(Mercury_rate_arc_sec_century,'%.1f'),' arc/s per century']);
```

Finally, here are the results

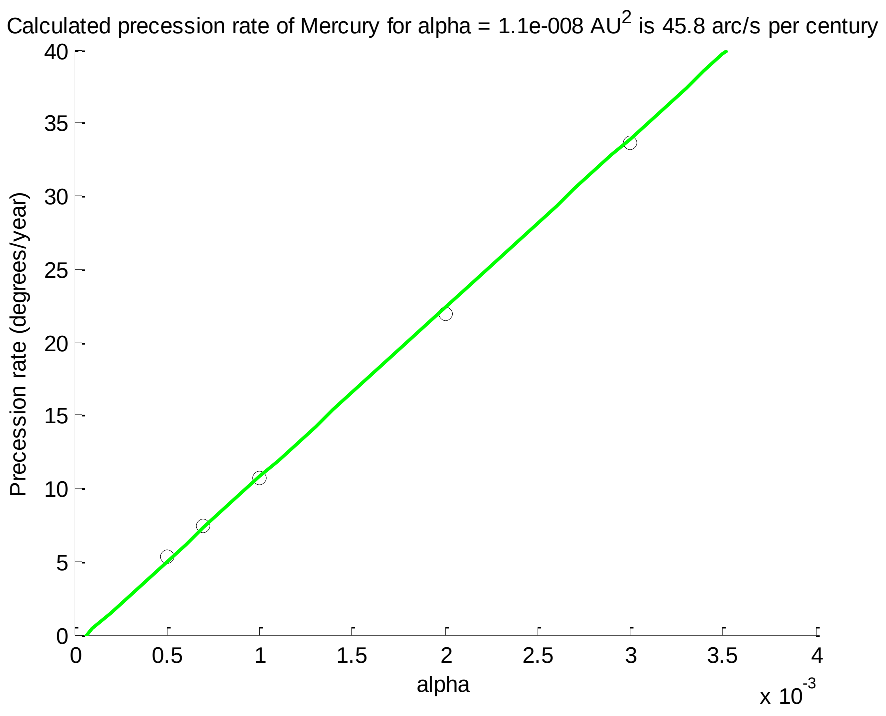


Figure 24. Calculated precession rate of Mercury

4.4 The three body problem and the effect of Jupiter on Earth

A couple of points on this problem. Firstly, the direction of the various forces between the 3 bodies is elegantly captured in the pseudocode given in Ex 4.2. Do not be tempted to start taking absolute values of the subtracted positions in a naïve bid to 'correct' the equations.
Here is the code

```
%
% 3 body simulation of Jupiter, Earth and Sun. Use Euler Cromer method
% based on 'Computational Physics' book by N Giordano and H Nakanishi
% Section 4.4
% by Kevin Berwick
%
%
npoints=1000000;
dt = 0.0001; % time step in years.
M_s=2e30; % Mass of the Sun in kg
M_e=6e24; % Mass of the Earth in kg
Mj_actual=1.9e27; % Actual mass of Jupiter
M_j=1500*Mj_actual              % Mass of Jupiter in kg this allows you to vary the value of Jupiter's mass in
                                %order to explore the effect of this on the simulation
x_e_initial=1; % Initial position of Earth in AU
y_e_initial=0;
v_e_x_initial=0; % Initial velocity of Earth in AU/yr
v_e_y_initial=2*pi;
x_j_initial=5.2; % Initial position of Jupiter in AU, assume at opposition initially
y_j_initial=0;
v_j_x_initial=0; % Initial velocity of Jupiter in AU/yr
v_j_y_initial= 2.7549; % This is 2*pi*5.2 AU/11.85 years = 2.75 AU/year
% Create arrays to store position and velocity of Earth
x_e=zeros(npoints,1);
y_e=zeros(npoints,1);
v_e_x=zeros(npoints,1);
v_e_y=zeros(npoints,1);
% Create arrays to store position and velocity of Jupiter
x_j=zeros(npoints,1);
y_j=zeros(npoints,1);
v_j_x=zeros(npoints,1);
v_j_y=zeros(npoints,1);
r_e=zeros(npoints,1);
r_j=zeros(npoints,1);
r_e_j=zeros(npoints,1);
% Initialise positions and velocities of Earth and Jupiter
x_e(1)=x_e_initial;
y_e(1)=y_e_initial;
v_e_x(1)=v_e_x_initial;
v_e_y(1)=v_e_y_initial;
x_j(1)=x_j_initial;
y_j(1)=y_j_initial;
v_j_x(1)=v_j_x_initial;
v_j_y(1)=v_j_y_initial;
for i = 1:npoints-1; % loop over the timesteps
% Calculate distances to Earth from Sun, Jupiter from Sun and Jupiter
% to Earth for current value of i
r_e(i)=sqrt(x_e(i)^2+y_e(i)^2);
r_j(i)=sqrt(x_j(i)^2+y_j(i)^2);
r_e_j(i)=sqrt((x_e(i)-x_j(i))^2 +(y_e(i)-y_j(i))^2);
% Compute x and y components for new velocity of Earth
```

```
v_e_x(i+1)=v_e_x(i)-4*pi^2*x_e(i)*dt/r_e(i)^3-4*pi^2*(M_j/M_s)*(x_e(i)-x_j(i))*dt/r_e_j(i)^3;
v_e_y(i+1)=v_e_y(i)-4*pi^2*y_e(i)*dt/r_e(i)^3-4*pi^2*(M_j/M_s)*(y_e(i)-y_j(i))*dt/r_e_j(i)^3;
% Compute x and y components for new velocity of Jupiter
v_j_x(i+1)=v_j_x(i)-4*pi^2*x_j(i)*dt/r_j(i)^3-4*pi^2*(M_e/M_s)*(x_j(i)-x_e(i))*dt/r_e_j(i)^3;
v_j_y(i+1)=v_j_y(i)-4*pi^2*y_j(i)*dt/r_j(i)^3-4*pi^2*(M_e/M_s)*(y_j(i)-y_e(i))*dt/r_e_j(i)^3;
%
% Use Euler Cromer technique to calculate the new positions of Earth and
% Jupiter. Note the use of the NEW vlaue of velocity in both equations
x_e(i+1)=x_e(i)+v_e_x(i+1)*dt;
y_e(i+1)=y_e(i)+v_e_y(i+1)*dt;
x_j(i+1)=x_j(i)+v_j_x(i+1)*dt;
y_j(i+1)=y_j(i)+v_j_y(i+1)*dt;
end;
plot(x_e,y_e, 'r', x_j,y_j, 'k');
axis([-7 7 -7 7]);
xlabel('x(AU)');
ylabel('y(AU)');
title('3 body simulation - Jupiter Earth');
```

Here are the results using the actual mass of Jupiter

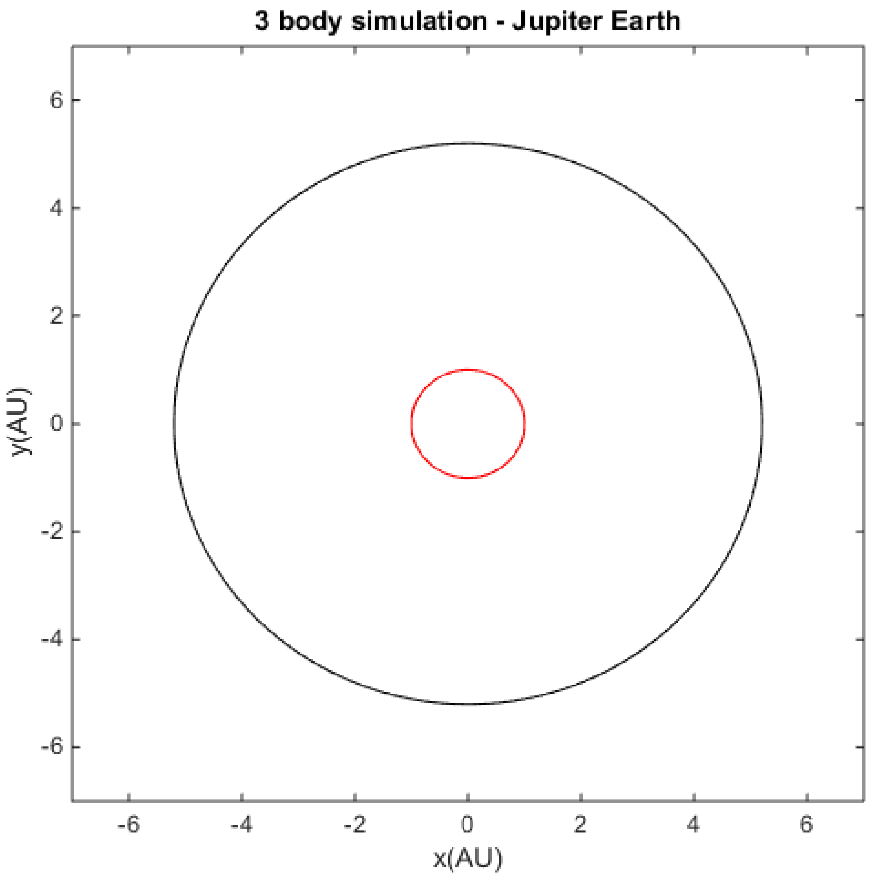


Figure 25. Simulation of solar system containing Jupiter and Earth. Actual mass of Jupiter used.

By changing the value of M_j, circled in red, you get the plot below for a mass of Jupiter = 10*M_j, that is, 10 times it's actual value.

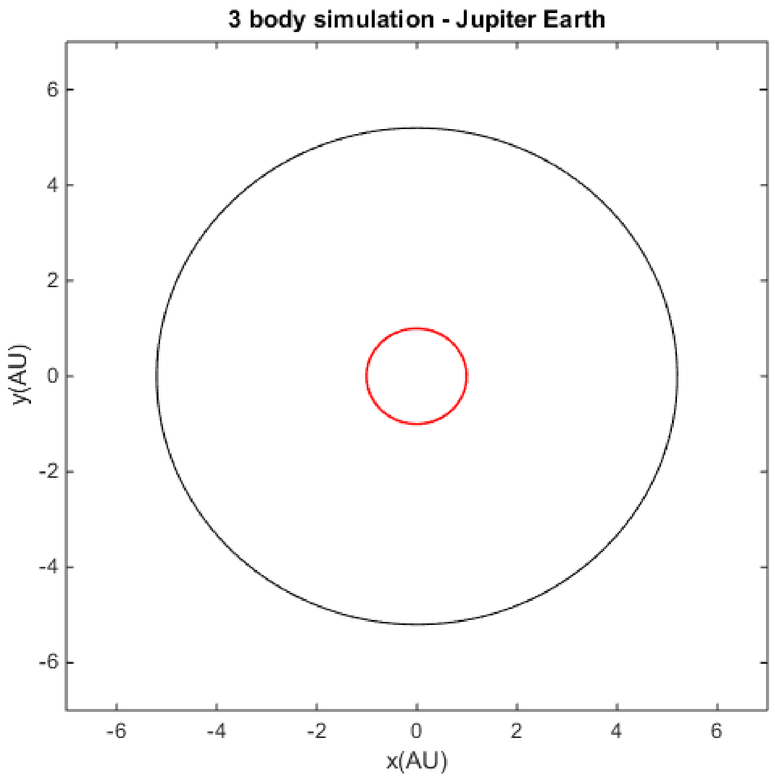


Figure 26. Simulation of solar system containing Jupiter and Earth. Jupiter mass is 10 X actual value.

If the mass of Jupiter is increased to 1000 times the actual value and the perturbation of Jupiter on the Sun is ignored then we get the plot below,

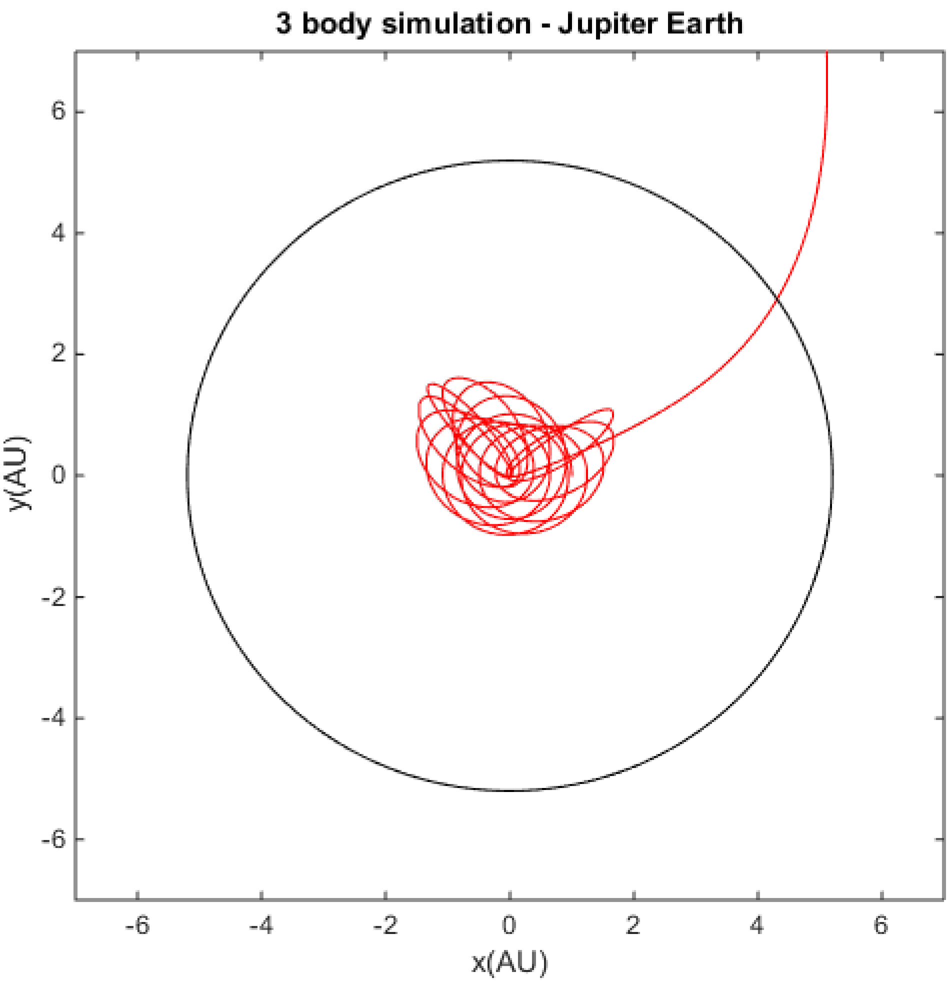


Figure 27.Simulation of solar system containing Jupiter and Earth. Jupiter mass is 1000 X actual value, ignoring perturbation of the Sun.

4.6 Chaotic tumbling of Hyperion

The model of the moon consists of two particles, m_1 and m_2 joined by a massless rod. This orbits around a massive object, Saturn, at the origin. We need to extend our original planetary motion program to include the rotation of the object. First we need to recall the maths in Section 4.1, we used in order to calculate the motion of the Earth around the Sun.

$$\frac{d^2x}{dt^2} = \frac{F_{G,x}}{M_E} \qquad \frac{d^2y}{dt^2} = \frac{F_{G,y}}{M_E}$$

$$F_{G,x} = -\frac{GM_sM_e}{r^2}\,cos\theta \qquad From\ the\ diagram\ \ cos\theta = \frac{x}{r}\ \ so$$

$$F_{G,x} = -\frac{GM_sM_ex}{r^3}$$

$$F_{G,y} = -\frac{GM_sM_e}{r^2}\,sin\theta \qquad From\ the\ diagram\ \ sin\theta = \frac{y}{r}\ \ so$$

$$F_{G,y} = -\frac{GM_sM_ey}{r^3}$$

Now, write each 2nd order differential equations as two, first order, differential equations.

$$\frac{dv_x}{dt} = -\frac{GM_s\,x}{r^3}$$

$$\frac{dx}{dt} = v_x$$

$$\frac{dv_y}{dt} = -\frac{GM_s\,y}{r^3}$$

$$\frac{dy}{dt} = v_y$$

We need suitable units of mass. Not that the Earth's orbit is circular. For circular motion we know that the centripetal force is given by $\frac{M_E\,v^2}{r}$, where v is the velocity of the Earth.

$$\frac{M_E\,v^2}{r} = F_G = \frac{GM_s\,M_E}{r^2}$$

$$GM_s = v^2r = 4\pi^2AU^3/yr^2$$

Since the velocity of Earth is $2\pi r/yr = 2\pi 1AU/yr$

So, the difference equation set using the Euler Cromer method is

$$v_{x,i+1} = v_{x,i} - \frac{4\pi^2 x_i}{r_i^3}\Delta t$$

$$x_{i+1} = x_i + v_{x,i+1}\,\Delta t$$

$$v_{y,i+1} = v_{y,i} - \frac{4\pi^2 y_i}{r_i^3}\Delta t$$

$$y_{i+1} = y_i + v_{y,i+1}\,\Delta t$$

We can use this equation set to model the motion of the centre of mass of Hyperion. Now, from the analysis of the motion of Hyperion.

$$\omega = \frac{d\theta}{dt}$$

$$\frac{d\omega}{dt} \approx -\frac{3GM_{sat}}{r_c^5}(x_c sin\theta - y_c cos\theta)(x_c cos\theta + y_c sin\theta)$$

So we need to add two more difference equations to our program, and noting that

$GM_{sat} = 4\pi^2$ as noted in the book,

$$\omega_{i+1} = \omega_i - \frac{3(4\pi^2)}{r_c^5}(x_i sin\theta_i - y_i cos\theta_i)(x_i cos\theta_i + y_i sin\theta_i)\Delta t$$

$$\theta_{i+1} = \theta_i + \omega_{i+1}\Delta t$$

Here is the code for the motion of Hyperion. The initial velocity in the y direction was 1 HU/Hyperion year as explained in the book. This gave a circular orbit. Note from the results that the tumbling is not chaotic under these conditions.

```
%
%  Simulation of chaotic tumbing of Hyperion, the moon of Saturn . Use Euler Cromer method
%  based on 'Computational Physics' book by N Giordano and H Nakanishi
%  Section 4.6
%  by Kevin Berwick
%
%


npoints=100000;
dt = 0.0001;                   % time step in years

time=zeros(npoints,1);
r_c=zeros(npoints,1);
                                   % Create arrays to store position, velocity and angle and angular velocity of
                                   % centre of mass
x=zeros(npoints,1);
y=zeros(npoints,1);

v_x=zeros(npoints,1);
v_y=zeros(npoints,1);

theta=zeros(npoints,1);
omega=zeros(npoints,1);

x(1)=1;                  %  initialise position of centre of mass of Hyperion  in HU
y(1)=0;
v_x(1)=0;                  % initialise velocity of centre of mass of Hyperion
v_y(1)=2*pi;

% initialise  theta and omega of Hyperion

for i= 1:npoints-1;

    % loop over the timesteps

    r_c(i)=sqrt(x(i)^2+y(i)^2);

    % Compute new velocities in the x and y directions

    v_x(i+1)=v_x(i) - (4*pi^2*x(i)*dt)/(r_c(i)^3);
    v_y(i+1)=v_y(i) - (4*pi^2*y(i)*dt)/(r_c(i)^3);

    % Euler Cromer Step - update positions of centre of mass of Hyperion using NEWLY calculated
velocities

    x(i+1)=x(i)+v_x(i+1)*dt;
    y(i+1)=y(i)+v_y(i+1)*dt;

% Calculate new angular velocity omega and angle theta. Note that GMsaturn=4*pi^2, see book for
details


Term1=3*4*pi^2/(r_c(i)^5);
Term2=x(i)*sin(theta(i))- y(i)*cos(theta(i));
Term3=x(i)*cos(theta(i)) +y(i)*sin(theta(i));

omega(i+1)=omega(i) -Term1*Term2*Term3*dt;

%Theta is an angular variable so values of theta that differ by 2*pi correspond to the same position.
%We need to adjust theta after each iteration so as to keep it between
```

```
%+/-pi for plotting purposes. We do that here

temporary_theta_i_plus_1= theta(i)+omega(i+1)*dt;
  if (temporary_theta_i_plus_1 < -pi)
      temporary_theta_i_plus_1= temporary_theta_i_plus_1+2*pi;
  elseif (temporary_theta_i_plus_1 > pi)
     temporary_theta_i_plus_1= temporary_theta_i_plus_1-2*pi;
 end;

 % Update theta array
      theta(i+1)=temporary_theta_i_plus_1;

time(i+1)=time(i)+dt;

end;

subplot(2,1,1);
plot(time, theta,'-g');
 axis([0 8 -4 4]);
xlabel('time(year)');
ylabel('theta(radians)');
title('theta versus time for Hyperion');

subplot(2,1,2);
plot(time, omega,'-k');
 axis([0 8 0 15]);
xlabel('time(year)');
ylabel('omega(radians/yr)');
title('omega versus time for Hyperion');
```

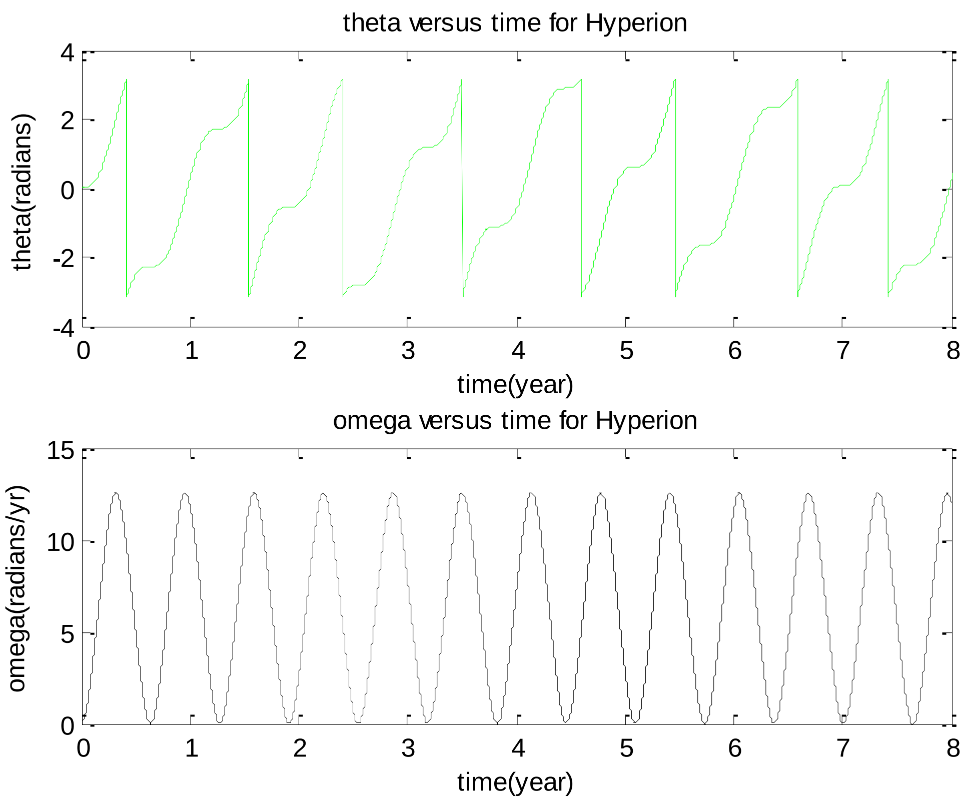


Figure 28.Motion of Hyperion. The initial velocity in the y direction was 1 HU/Hyperion year. This gave a circular orbit. Note from the results that the tumbling is not chaotic under these conditions.

If we change the initial velocity in the y direction to 5 HU/Hyperion year as explained in the book. This gave an elliptical orbit. Here is the new code and below this code are the results from running this code. Note that now the motion is chaotic.

```
%
%  Simulation of chaotic tumbing of Hyperion, the moon of Saturn . Use Euler Cromer method
%  based on 'Computational Physics' book by N Giordano and H Nakanishi
%  Section 4.6
%  by Kevin Berwick
%
%


npoints=100000;
dt = 0.0001;                % time step in years

time=zeros(npoints,1);
r_c=zeros(npoints,1);
                                 % Create arrays to store position, velocity and angle and angular velocity of
                                 % centre of mass
x=zeros(npoints,1);
y=zeros(npoints,1);

v_x=zeros(npoints,1);
v_y=zeros(npoints,1);

theta=zeros(npoints,1);
omega=zeros(npoints,1);

x(1)=1;              %  initialise position of centre of mass of Hyperion  in HU
y(1)=0;
v_x(1)=0;              % initialise velocity of centre of mass of Hyperion
v_y(1)=5;

% initialise  theta and omega of Hyperion

for i= 1:npoints-1;

    % loop over the timesteps

    r_c(i)=sqrt(x(i)^2+y(i)^2);

    % Compute new velocities in the x and y directions

    v_x(i+1)=v_x(i) - (4*pi^2*x(i)*dt)/(r_c(i)^3);
    v_y(i+1)=v_y(i) - (4*pi^2*y(i)*dt)/(r_c(i)^3);

    % Euler Cromer Step - update positions of centre of mass of Hyperion using NEWLY calculated
velocities

    x(i+1)=x(i)+v_x(i+1)*dt;
    y(i+1)=y(i)+v_y(i+1)*dt;

% Calculate new angular velocity omega and angle theta. Note that GMsaturn=4*pi^2, see book for
details


Term1=3*4*pi^2/(r_c(i)^5);
Term2=x(i)*sin(theta(i))- y(i)*cos(theta(i));
Term3=x(i)*cos(theta(i)) +y(i)*sin(theta(i));
```

```
omega(i+1)=omega(i) -Term1*Term2*Term3*dt;

%Theta is an angular variable so values of theta that differ by 2*pi correspond to the same position.
%We need to adjust theta after each iteration so as to keep it between
%+/-pi for plotting purposes. We do that here

temporary_theta_i_plus_1= theta(i)+omega(i+1)*dt;
  if (temporary_theta_i_plus_1 < -pi)
        temporary_theta_i_plus_1= temporary_theta_i_plus_1+2*pi;
  elseif (temporary_theta_i_plus_1 > pi)
      temporary_theta_i_plus_1= temporary_theta_i_plus_1-2*pi;
 end;

 % Update theta array
        theta(i+1)=temporary_theta_i_plus_1;

time(i+1)=time(i)+dt;

end;

subplot(2,1,1);
plot(time, theta,'-g');
 axis([0 10 -4 4]);
xlabel('time(year)');
ylabel('theta(radians)');
title('theta versus time for Hyperion');

subplot(2,1,2);
plot(time, omega,'-k');
 axis([0 10 -20 60]);
xlabel('time(year)');
ylabel('omega(radians/yr)');
title('omega versus time for Hyperion');
```

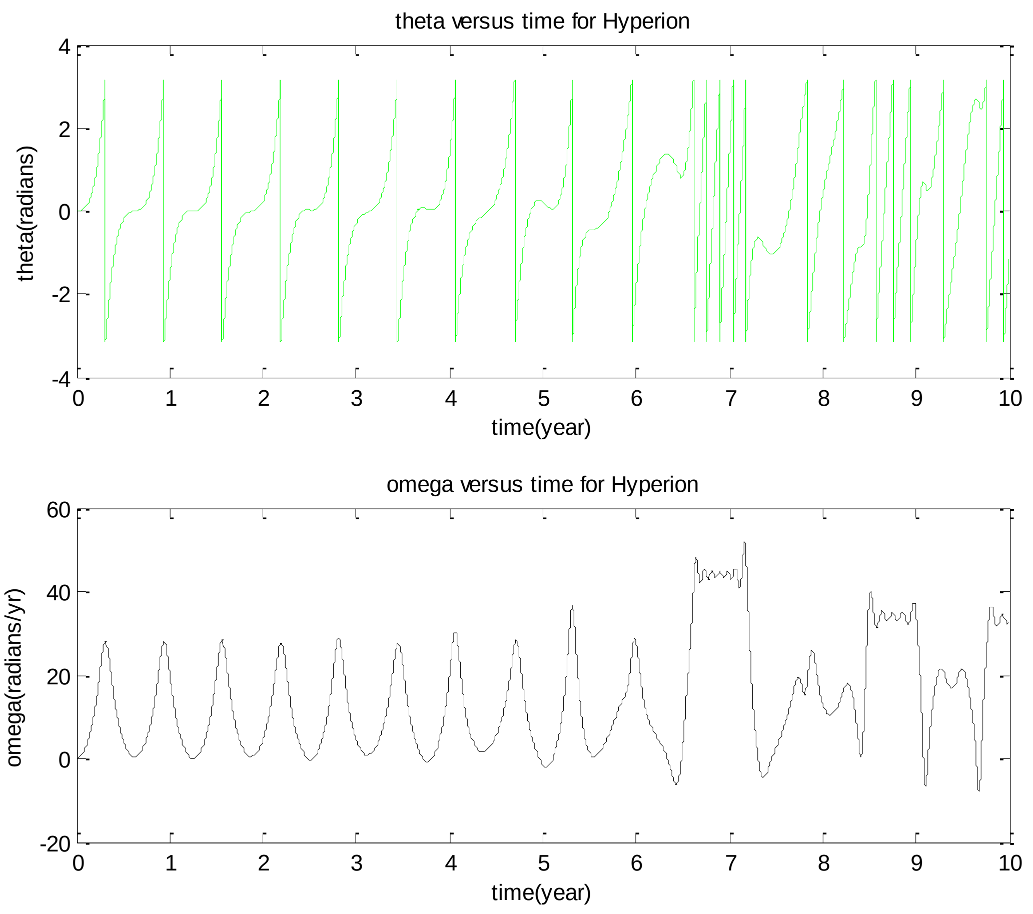


Figure 29.Motion of Hyperion. The initial velocity in the y direction was 5 HU/Hyperion year. This gave a circular orbit. Note from the results that the tumbling is chaotic under these conditions.

5. Potentials and Fields

5.1 Solution of Laplace's equation using the Jacobi relaxation method.

There are 3 files required here

1. Laplace_calculate_Jacobi_metal_box
2. Initialise_V_Jacobi_metal_box;
3. Update_V_Jacobi_Metal_box

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Jacobi method to solve Laplace equation
% based on 'Computational Physics' book by N Giordano and H Nakanishi
% Section 5.1
% by Kevin Berwick
%
% Load array into V

 [V] =Initialise_V_Jacobi_metal_box;

% run update routine and estimate convergence

%Initialise loop counter
 loops=1;
[V_new, delta_V_new]=Update_V_Jacobi_Metal_box(V);

% While we have not met the convergence criterion and the number of loops is <10 so that we
give the relaxation
% algorithm time to converge

 while (delta_V_new > 49e-5 & loops < 10);
   loops=loops+1;
   [V_new, delta_V_new]=Update_V_Jacobi_Metal_box(V_new);
   % draw the  surface using the mesh function
   mesh (V_new);
   title('Potential Surface');
   drawnow;
   % insert a pause here so we see the evolution of the potential
   % surface
   pause(1);
  end;
```

```
       %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%  Jacobi method to solve Laplace equation
%  based on 'Computational Physics' book by N Giordano and H Nakanishi
%  Section 5.1
%  by Kevin Berwick
%

% This function creates the intial voltage array V

function[V] =Initialise_V_Jacobi_metal_box;
% clear variables
clear;
V = [-1 -0.67 -0.33 0 0.33 0.67 1;
        -1     0      0    0    0      0    1;
        -1     0      0    0    0      0    1;
        -1     0      0    0    0      0    1;
        -1     0      0    0    0      0    1;
        -1     0      0    0    0      0    1;
        -1 -0.67 -0.33   0   0.33   0.67 1];


       %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [ V_new, delta_V_new] = Update_V_Jacobi_Metal_box(V);

% This function takes a matrix V and applies Eq 5.10 to it. Only the values inside the boundaries are
changed. It returns the
% processed matrix to the  calling function, together with the value of delta_V, the total accumulated
amount by which the elements
% of the matrix have changed

row_size = size(V,1);
column_size=size(V,2);

% preallocate memory for speed

V_new=V;
delta_V_new=0;

% Move along the matrix, element by element  computing  Eq 5.10, ignoring
% boundaries

   for j =2:column_size-1;
      for i=2:row_size -1;

        V_new(i,j) = (V(i-1,j)+V(i+1,j)+V(i,j-1)+V(i,j+1))/4;

        % Calculate delta_V_new value, the cumulative change in V during this update call,  to test for
convergence

        delta_V_new=delta_V_new+abs(V_new(i,j)-V(i,j));
      end;
   end;


        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

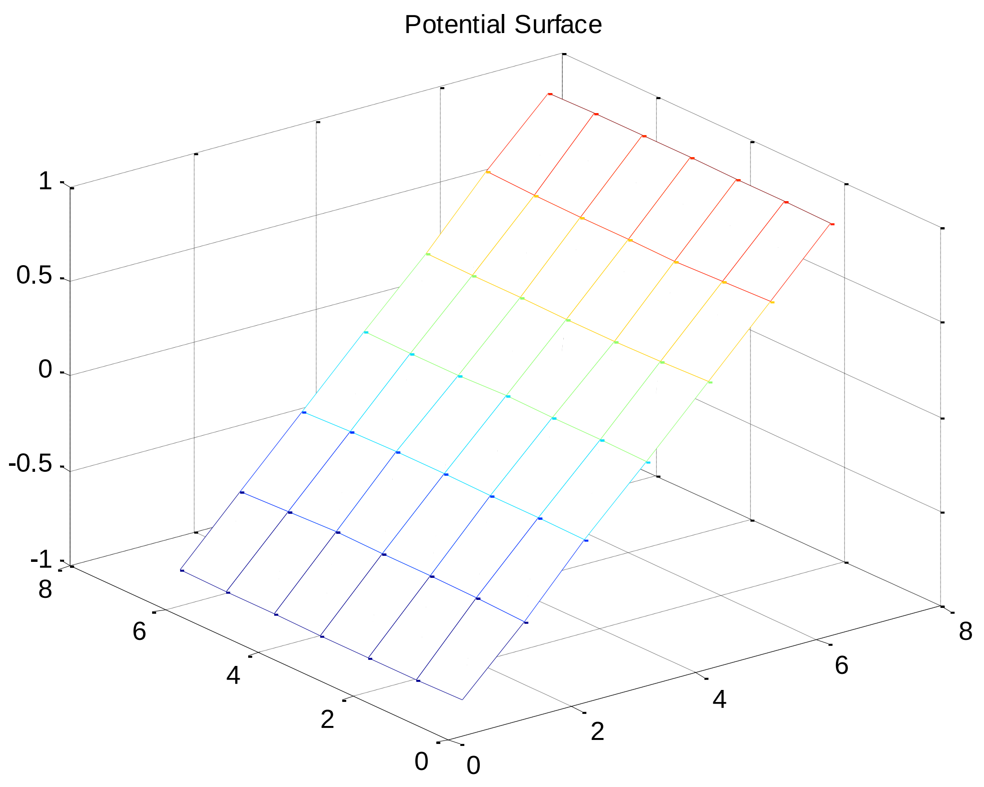


Figure 30. Equipotential surface for geometry depicted in Figure 5.2 in the book

5.1.1 Solution of Laplace's equation for a hollow metallic prism with a solid, metallic inner conductor.

This is the solution for the situation shown in Figure 5.4. There are 3 files required here and the code is listed in order below, together with the output.

1. Laplace_prism
2. Initialise_prism
3. Update_prism

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%
%  Jacobi method to solve Laplace equation
%  based on 'Computational Physics' book by N Giordano and H Nakanishi
%  Section 5.1
%  by Kevin Berwick
%
% Load array into V
 [V] =Initialise_prism;

% run update routine and estimate convergence

[V_new, delta_V_new]=Update_prism(V);

%Initialise loop counter
 loops=0;
%  While we have not met the convergence criterion and the number of loops is <10 so that we give the relaxation
% algorithm time to converge
% while (delta_V_new > & loops < 30);
   while (delta_V_new>4e-5  | loops < 20);
     loops=loops+1;
     [V_new, delta_V_new]=Update_prism(V_new);
   % draw the  surface using the mesh function
%      mesh (V_new,'FaceColor','interp','EdgeColor','none','FaceLighting','phong');
        mesh (V_new,'FaceColor','interp');

    title('Potential Surface');
     axis([0 20 0 20 0 1]);
   drawnow;
   % insert a pause here so we see the evolution of the potential
   % surface
   pause(0.5);
 end;


    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%  Jacobi method to solve Laplace equation
%  based on 'Computational Physics' book by N Giordano and H Nakanishi
%  Section 5.1
%  by Kevin Berwick
%

% This function creates the intial voltage array V

function[V] =Initialise_prism;
```

```
% clear variables
clear;

V = [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
     0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
     0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
     0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
     0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
     0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
     0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
     0 0 0 0 0 0  0 1  1  1  1  1  1 0 0 0 0 0 0 0
     0 0 0 0 0 0  0 1  1  1  1  1  1 0 0 0 0 0 0 0
     0 0 0 0 0 0 0 1  1  1  1  1  1 0 0 0 0 0 0 0
     0 0 0 0 0 0  0 1  1  1  1  1  1 0 0 0 0 0 0 0
     0 0 0 0 0 0  0 1  1  1  1  1  1 0 0 0 0 0 0 0
     0 0 0 0 0 0  0 1  1  1  1  1  1 0 0 0 0 0 0 0
     0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
     0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
     0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
     0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
     0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
     0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
     0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    ];
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [ V_new, delta_V_new] = Update_prism(V);

% This function takes a matrix V and applies Eq 5.10 to it. Only the values inside the boundaries are
changed. It returns the
% processed matrix to the  calling function, together with the value of delta_V, the total accumulated
amount by which the elements
% of the matrix have changed

row_size = size(V,1);
column_size=size(V,2);

% preallocate memeory for speed

V_new=V;
delta_V_new=0;

% Move along the matrix, element by element  computing  Eq 5.10, ignoring
% boundaries

  for j =2:column_size-1;
    for i=2:row_size -1;

      % Do not update V in metal bar
      if  V(i,j) ~=1;
        % If the value of V is not =1, calculate V_new and
        % delta_V_new to test for convergence
            V_new(i,j) = (V(i-1,j)+V(i+1,j)+V(i,j-1)+V(i,j+1))/4;
            delta_V_new=delta_V_new+abs(V_new(i,j)-V(i,j))
      else
        % otherwise, leave value unchanged
              V_new(i,j)=V(i,j);
      end;
    end;
  end;
```

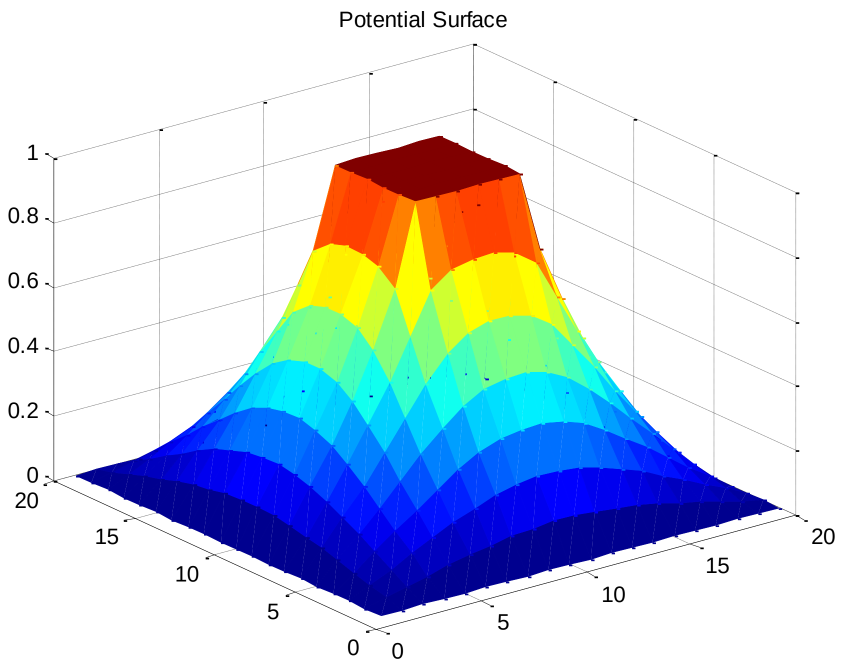


Figure 31.Equipotential surface for hollow metallic prism with a solid metallic inner conductor held at V=1.

5.1.2 Solution of Laplace's equation for a finite sized capacitor

This is the solution for the situation shown in Figure 5.6 and 5.7. There are 3 files required here and the code is listed in order below, together with the output.

1. capacitor_laplace
2. capacitor_initialise
3. capacitor_update

```
%
%  Jacobi method to solve Laplace equation
%  based on 'Computational Physics' book by N Giordano and H Nakanishi
%  Section 5.1
%  by Kevin Berwick
%
% Load array into V

 [V] =capacitor_initialise;

% run update routine and estimate convergence

[V_new, delta_V_new]=capacitor_update(V);

%Initialise loop counter
 loops=1;
%  While we have not met the convergence criterion and the number of loops is <20 so that we give
the relaxation
% algorithm time to converge

   while (delta_V_new>400e-5  | loops < 20);
       loops=loops+1;
       [V_new, delta_V_new]=capacitor_update(V_new);
```

```
        % draw the  surface using the mesh function
         mesh (V_new,'Facecolor','interp');
         title('Potential Surface');
         axis([0 20 0 20 -1 1]);
         drawnow;
           % insert a pause here so we see the evolution of the potential
             % surface
          pause(0.5);
  end;

 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%
%  Jacobi method to solve Laplace equation
%  based on 'Computational Physics' book by N Giordano and H Nakanishi
%  Section 5.1
%  by Kevin Berwick
%

% This function creates the intial voltage array V

function[V] =capacitor_initialise;
% clear variables
clear;

V = [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
       0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
       0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
       0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
       0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
       0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
       0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
       0 0 0 0 0 0 1 0 0 0 0 0 -1 0 0 0 0 0 0 0
       0 0 0 0 0 0 1 0 0 0 0 0 -1 0 0 0 0 0 0 0
       0 0 0 0 0 0 1 0 0 0 0 0 -1 0 0 0 0 0 0 0
       0 0 0 0 0 0 1 0 0 0 0 0 -1 0 0 0 0 0 0 0
       0 0 0 0 0 0 1 0 0 0 0 0 -1 0 0 0 0 0 0 0
       0 0 0 0 0 0 1 0 0 0 0 0 -1 0 0 0 0 0 0 0
       0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
       0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
       0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
       0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
       0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
       0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
       0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
      ];
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [ V_new, delta_V_new] = capacitor_update(V);

% This function takes a matrix V and applies Eq 5.10 to it. Only the values inside the boundaries are
changed. It returns the
% processed matrix to the  calling function, together with the value of delta_V, the total accumulated
amount by which the elements
% of the matrix have changed

row_size = size(V,1);
column_size=size(V,2);

% preallocate memory for speed

V_new=zeros(row_size, column_size);
delta_V_new=0;

% Move along the matrix, element by element  computing  Eq 5.10, ignoring
% boundaries

  for j =2:column_size-1;
    for i=2:row_size -1;

      % Do not update V on the plates
      if  V(i,j)~=1 & V(i,j) ~=-1;
        % If the value of V is not =1 or -1, calculate V_new and
        % cumulative delta_V_new to test for convergence
            V_new(i,j) = (V(i-1,j)+V(i+1,j)+V(i,j-1)+V(i,j+1))/4;
            delta_V_new=delta_V_new+abs(V_new(i,j)-V(i,j))
      else
        % otherwise, leave value unchanged
              V_new(i,j)=V(i,j);
      end;
    end;
  end;



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

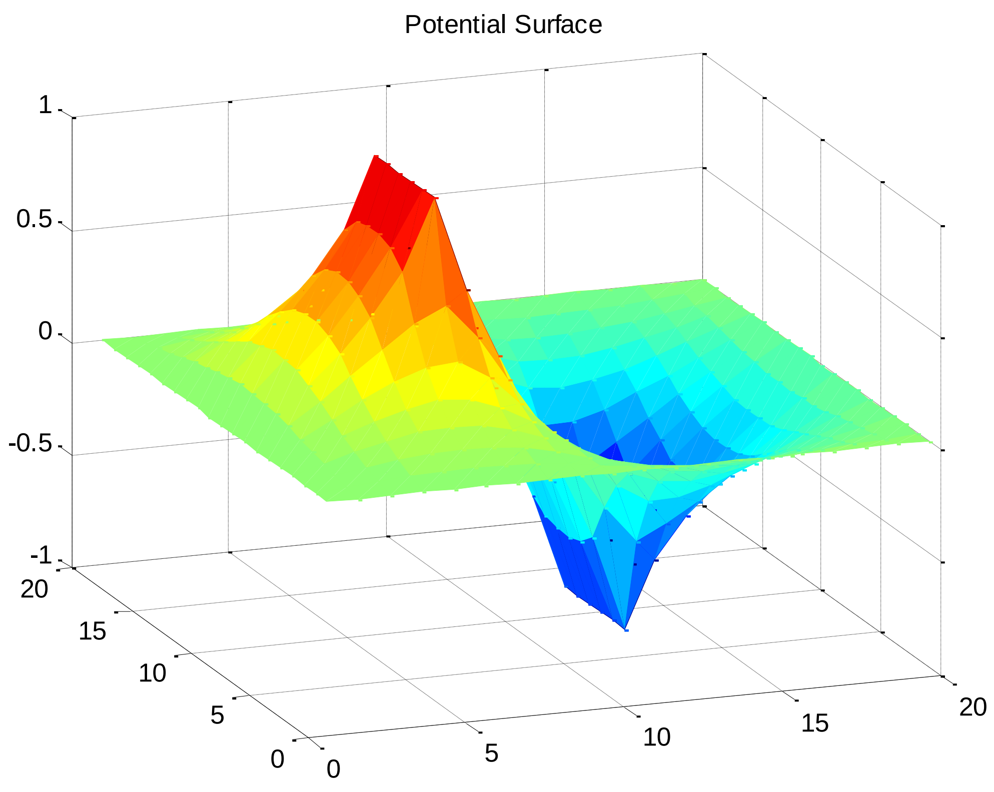


Figure 32.Equipotential surface for a finite sized capacitor.

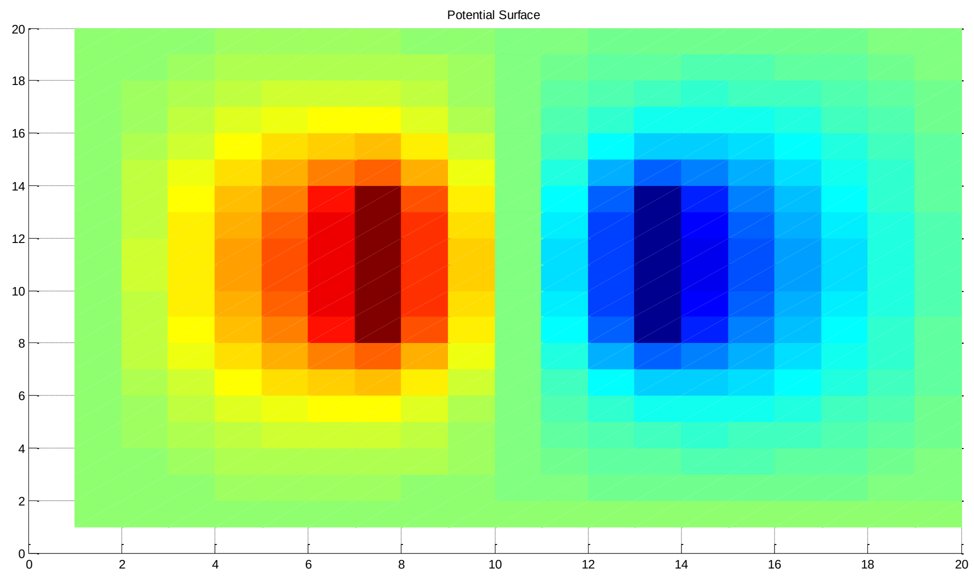


Figure 33. Equipotential contours near a finite sized capacitor.

5.1.3 Exercise 5.7 and the Successive Over Relaxation Algorithm

There are 3 files required here and the code for the SOR algorithm used in order to sole Laplaces equation for the capacitor is listed in order below, together with the output.

1. capacitor_laplace_SOR
2. capacitor_initialise_SOR
3. capacitor_update_SOR

```
%
%  SOR method to solve Laplace equation
%  based on 'Computational Physics' book by N Giordano and H Nakanishi
%  Section 5.1
%  by Kevin Berwick
%
% Load array into V

 [V] =capacitor_initialise_SOR;

% run update routine and estimate convergence

[V, delta_V_total]=capacitor_update_SOR(V);

%Initialise loop counter
 loops=1;
%  While we have not met the convergence criterion and the number of loops is <20 so that we give
the relaxation
% algorithm time to converge.Note convergence for 1e-5 * No of sites

   while (delta_V_total>1e-5*size(V,2)^2  | loops < 20);
        loops=loops+1
        [V, delta_V_total]=capacitor_update_SOR(V);

        % draw the  surface using the mesh function
         mesh(V,'Facecolor','interp');
         title('Potential Surface');
         axis([0 60 0 60 -1 1]);
         drawnow;
            % insert a pause here so we see the evolution of the potential
              % surface
          pause(0.5);
  end;
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  SOR method to solve Laplace equation
%  based on 'Computational Physics' book by N Giordano and H Nakanishi
%  Section 5.1
%  by Kevin Berwick
%

% This function creates the intial voltage array V

function[V] =capacitor_initialise_SOR;
% clear variables
clear;

V = [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
     0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
     0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
     0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
     0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
     0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
     0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
     0 0 0 0 0 0 1 0 0 0 0 0 -1 0 0 0 0 0 0 0
     0 0 0 0 0 0 1 0 0 0 0 0 -1 0 0 0 0 0 0 0
     0 0 0 0 0 0 1 0 0 0 0 0 -1 0 0 0 0 0 0 0
     0 0 0 0 0 0 1 0 0 0 0 0 -1 0 0 0 0 0 0 0
     0 0 0 0 0 0 1 0 0 0 0 0 -1 0 0 0 0 0 0 0
     0 0 0 0 0 0 1 0 0 0 0 0 -1 0 0 0 0 0 0 0
     0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
     0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
     0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
     0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
     0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
     0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
     0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    ];
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [V, delta_V_total] = capacitor_update_SOR(V);
% This function takes a matrix V and applies Eq 5.14 to it. Only the values inside the boundaries are
changed. It returns the
% processed matrix to the  calling function, together with the value of delta_V, the total accumulated
amount by which the elements
% of the matrix have changed

row_size = size(V,1);
column_size=size(V,2);
L=column_size;       % grid size, for a square grid 20 X 20 , L=20
alpha = 2/(1+pi/L); % use recommended value for book for alpha
          % intialise convergence metric
delta_V_total=0;

% Move along the matrix, in a raster scan, element by element  computing  Eq 5.14, ignoring
% boundaries

  for i=2:row_size -1;
    for j =2:column_size-1;
          % Do not update V on the plates
            if  V(i,j)~=1 & V(i,j) ~=-1;

              % If the value of V is not =1 or -1, calculate the new value of the cell and
              % delta_V_new to test for convergence

                     V_star = (V(i-1,j)+V(i+1,j)+V(i,j-1)+V(i,j+1))/4;    % This is the Gauss Siedel updated
value for the cell
                     delta_V =V_star-V(i,j);% delta_V is the difference between the Gauss Siedel updated
value for the cell
                     % and the original value of the cell
                    % Update Matrix V , in place, so latest values will be used
                    % for SOR
                    V(i,j)=alpha*delta_V+V(i,j);  % add a multiple of delta_V to the original value in the cell,
that is, 'over-relax'
                     % Update convergence metric for this update
                   delta_V_total= delta_V_total+abs(delta_V);
             end;
        end;
    end;
```

Note that when you run the SOR code, it creates a movie of the potential surface. You can see waves moving across the potential surface as the surface 'over relaxes' and then corrects itself.

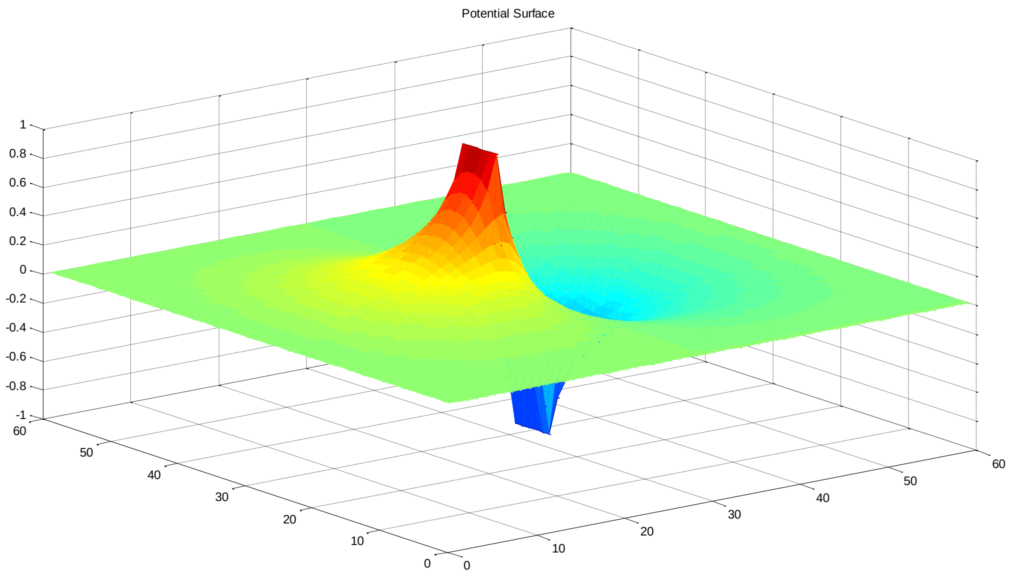


Figure 34.Equipotential surface in region of a simple capacitor as calculated using the SOR code for a 60 X 60 grid. The convergence criterion was that the simulation was halted when the difference in successively calculated surfaces was less than 10^{-5} per site.

The code was ran for both the Jacobi method and the SOR method, for the L X L grids shown. The convergence criterion was that the simulation was halted when the difference in successively calculated surfaces was less than 10^{-5} per site.
The results are summarised in the Table below.

L	**20 (400 elements)**	**40 (1600 elements)**	**60 (3600 elements)**
N Jacobi	128	492	884
N_SOR	33	60	84

Here are plots comparing the number of iterations required to give the same accuracy for each algorithm.

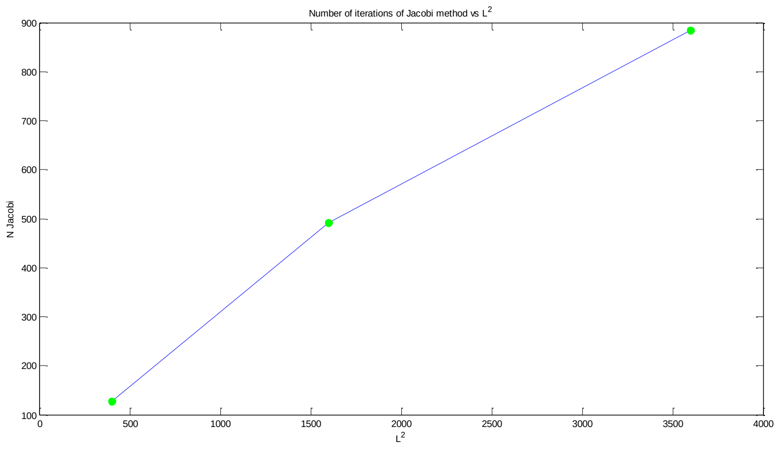


Figure 35.Number of iterations required for Jacobi method vs L for a simple capacitor. The convergence criterion was that the simulation was halted when the difference in successively calculated surfaces was less than 10^{-5} per site.

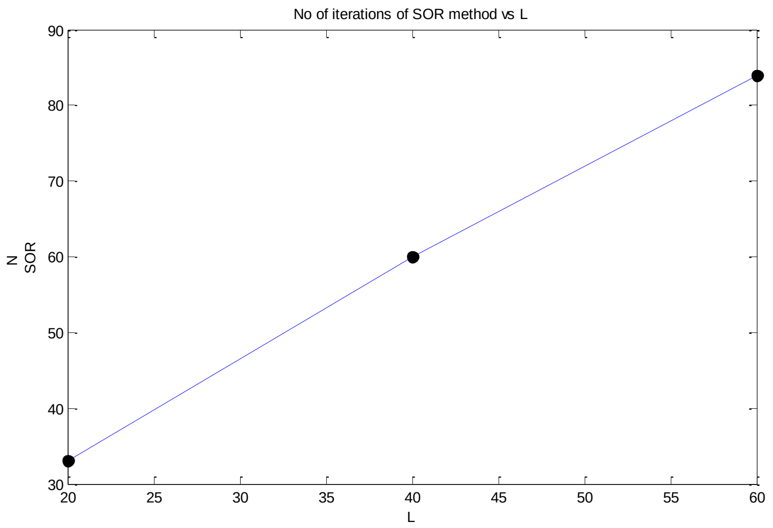


Figure 36.Number of iterations required for SOR method vs L for a simple capacitor. The convergence criterion was that the simulation was halted when the difference in successively calculated surfaces was less than 10^{-5} per site.

5.2 Potentials and fields near Electric charges, Poisson's Equation

Here we use the standard Jacobi method to calculate the potential surface near a point charge in the centre of a box, as outlined in Section 5.2

There are 3 files here

1. point_charge_Jacobi
2. point_charge_update
3. point_charge_coulomb

```
%  Jacobi method to solve Poisson's  equation
%  based on 'Computational Physics' book by N Giordano and H Nakanishi


% This code creates the intial potential and charge 3D matrices and
% definitions. The matrix  is 3D of length 20
clear;

delta_x=0.2;    % spatial step size
convergence_per_site=1e-6;
 rho=zeros(20,20,20);     % Create box to contain charge
rho(10,10,10)=(1/delta_x^3);  % place charge of 1 at centre of box shaped volume
% Create Potential matrix
V_matrix=zeros(20,20,20);

% run Update routine for the first time and calculate convergence metric

[V_new, delta_V_new]=point_charge_update(V_matrix, rho, delta_x);

%Initialise loop counter
 loops=1;
%  While we have not met the convergence criterion and the number of loops is <20 so that we give
the
% algorithm time to converge

   while (delta_V_new>convergence_per_site*size(V_matrix,2)^3 || loops < 20);
          loops=loops+1;
         [V_new, delta_V_new]=point_charge_update(V_new,rho, delta_x);
     end;
% Run the routine to plot the potential as calculated analytically using Coulomb's Law
     [coulomb, r]=point_charge_coulomb;
% Visualise result by taking a slice half way up the cube.
     slice = V_new(:, :, 10);
     subplot(1,2,1);
     surf(slice);
     title('Potential');
     view(3);
      axis on;
      grid on;
      light;
      lighting phong;
      camlight('left');
      shading interp;

  % take a cutline across the slice surface and plot

     subplot(1,2,2);
```

```
    cut_slice=slice(10, 10:20);
    rescale=[0:0.2:2];

    plot(rescale,cut_slice,'og');
    xlabel('x');
    ylabel('V');
    axis([0 2 0 0.8]);
    title('Numerical result and Coulombs Law')
    hold on;

    % Plot Coulomb's law result for comparison

    plot(r,coulomb,'k');
    legend('Numerical results','Coulomb');

 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


function [V_new, delta_V_new] = point_charge_update(V, rho, delta_x);

% This function takes a matrix V and solves Poisson's equation. It needs rho and delta_x
%the charge distribution and spatial step size also. It returns the
% processed matrix to the  calling function, together with the value of delta_V,
% the total accumulated amount by which the elements
% of the matrix have changed

x_size = size(V,1);
y_size = size(V,2);
z_size = size(V,3);

% preallocate memory for speed

V_new=zeros(x_size, y_size, z_size);
delta_V_new=0;

% Move along the matrix, element by element  computing  Eq 5.20, ignoring
% boundaries. Note the use of a,b,c instead of i,j,k since i and j are
% already defined in MATLAB

for c=2:z_size-1;
  for b =2:y_size-1;
     for a=2:x_size -1;

        % calculate V_new and cumulative delta_V_new to test for convergence
           V_new(a,b,c) = (V(a-1,b,c)+V(a+1,b,c)+V(a,b+1,c)+V(a,b-1,c)+V(a,b,c+1)+V(a,b,c-1))/6 +
rho(a,b,c)*delta_x^2/6 ;
           delta_V_new=delta_V_new+abs(V_new(a,b,c)-V(a,b,c));

        end;
     end;
  end;

 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [V,x] = point_charge_coulomb
% This function takes a matrix V and solves Poisson's equation. It needs rho and delta_x
%the charge distribution and spatial step size also. It returns the
% processed matrix to the  calling function, together with the value of delta_V,
% the total accumulated amount by which the elements
% of the matrix have changed
```

```
x=[0:0.05:2];
% Since q/epsillon_zero=1
V=1./(4*pi*x);
```

Here is the result of running the code

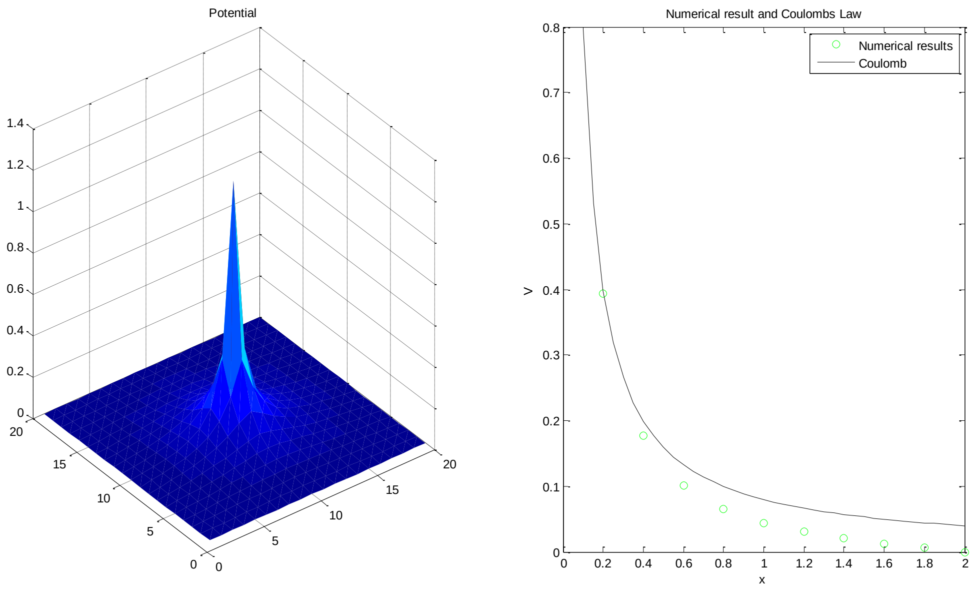


Figure 37. Equipotential surface near a point charge at the center of a 20X20 metal box. The Jacobi relaxation method was used . The plot on the right compares the numerical and analytical (as obtained from Coulomb's Law).

6. Waves

6.1 Waves on a string

Here we simulate the motion of a string using the wave equation.

There are 2 files here

1. waves
2. propagate

```
% Solution of wave equation for string
%  based on 'Computational Physics' book by N Giordano and H Nakanishi
clear;
string_dimension=100;
% Preallocate matrices for speed;
x=1/string_dimension:1/string_dimension:1;
x_scale=1:1:string_dimension;
y_next =zeros(1,string_dimension);
% Initialise string position
k=1000;
x_0=0.3;
initial_position=exp(-k.*(x-x_0).^2);
y_current =initial_position;
y_previous = initial_position;

for time_step = 1:500;
        [y_next]=propagate(y_current, y_previous);
        y_previous=y_current;
        y_current=y_next;
%           pause(0.1);
        clf;
        plot(x_scale/string_dimension, y_current,'r');
        title('Waves on a string - fixed ends');
        xlabel('distance');
        ylabel('Displacement');
        axis([0 1 -1 1]);
        hold on;
        drawnow;
end;

 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [y_next] = propagate(y_current, y_previous)

r=1;   % since r=c*delta_t/delta_x and for the Figure 6.2 example delta_t=delta_x/c, giving r=1
M=size(y_current,2);       % Vector size = number of columns

% there are 3 vectors containing positional information for the whole
% string,;    y_new, y_current and y_old
% preallocate memory for speed
% This function calculates the new shape of the string after one time step

% HERE IS THE ORIGINAL CODE
% for i=2:M-1;                                    %This loop index takes care of the fact that the boundaries are
fixed
%       y_next(i) = 2*(1-r^2)*y_current(i)....
%                    -y_previous(i)....
%                    +r^2*(y_current(i+1)+y_current(i-1));
%     end;
% HERE IS THE VECTORISED CODE, WHICH IS MORE EFFICIENT
```

```
y_next=zeros(1,M);
 i=2:1:M-1;                          %This loop index takes care of the fact that the boundaries are
fixed
     y_next(i) = 2*(1-r^2)*y_current(i)....
                    -y_previous(i)....
                    +r^2*(y_current(i+1)+y_current(i-1));
```

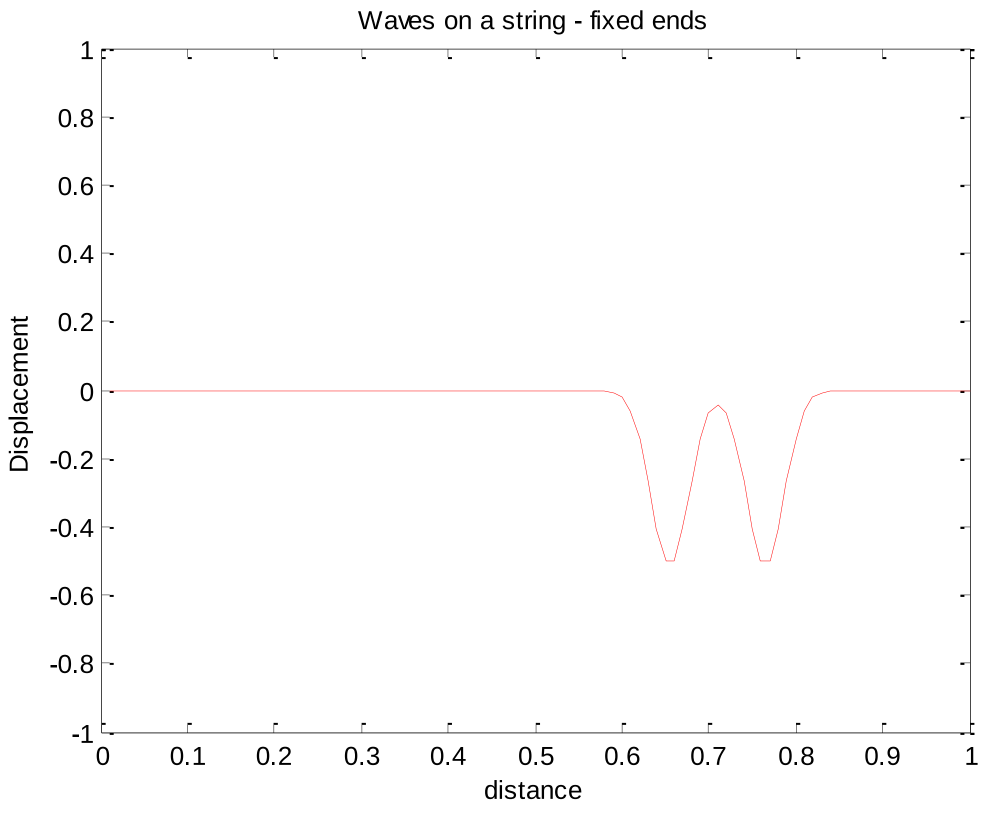


Figure 38. Waves propagating on a string with fixed ends

6.1.1 Waves on a string with free ends

Here we simulate the motion of a string using the wave equation, however, the string is free at the ends. (imagine the string tied to a massless ring which slides frictionlessly up and down a vertical pole). There are 2 files here

1. waves_free
2. propagate_free

```
% Solution of wave equation with free ends for string
%  based on 'Computational Physics' book by N Giordano and H Nakanishi
clear;
string_dimension=100;
time_loops=1500;
% Preallocate matrices for speed;
x=1/string_dimension:1/string_dimension:1;
x_scale=1:1:string_dimension;
y_next =zeros(1,string_dimension);
signal_data=zeros(1,time_loops);
elapsed_time=zeros(1,time_loops);
% Initialise string position
k=1000;
x_0=0.5;
 delta_t=3.33e-5;
initial_position=exp(-k.*(x-x_0).^2);
y_current =initial_position;
y_previous = initial_position;
initial_time=0;
time=initial_time;
for time_step = 1:time_loops;
        time=time+delta_t;
        [y_next]=propagate_free(y_current, y_previous);
        y_previous=y_current;
        y_current=y_next;
        clf;
        plot(x_scale/string_dimension, y_current,'r');
        title('Waves on a string - free ends');
        xlabel('distance');
        ylabel('Displacement');
        axis([0 1 -1 1]);
        hold on;
        drawnow;
end;


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
function [y_next] = propagate_free(y_current, y_previous)

 % c=300, delta_x=0.01, which makes delta_t = delta_x/c = 0.01/300 = 3.33e-5
 % since r=c*delta_t/delta_x and for the Figure 6.2 example
 % delta_t=delta_x/c, giving r=1.
r=1;
M=size(y_current,2);       % Vector size = number of columns

% there are 3 vectors containing positional information for the whole
% string,;    y_new, y_current and y_old
% preallocate memory for speed
% This function calculates the new shape of the string after one time step

% HERE IS THE ORIGINAL CODE
% for i=2:M-1;                        %This loop index takes care of the fact that the boundaries are
fixed
%       y_next(i) = 2*(1-r^2)*y_current(i)....
%                   -y_previous(i)....
%                   +r^2*(y_current(i+1)+y_current(i-1));
%     end;
% HERE IS THE VECTORISED CODE, WHICH IS MORE EFFICIENT

y_next=zeros(1,M);

 i=2:1:M-1;                           %This  index ignores the boundaries
     y_next(i) = 2*(1-r^2)*y_current(i)....
                 -y_previous(i)....
                 +r^2*(y_current(i+1)+y_current(i-1));

%                 Update position of free ends to those of nearest neighbours

y_next(1)=y_next(2);
y_next(M)=y_next(M-1);
```

6.2 Frequency spectrum of waves on a string

Here we simulate the motion of a string using the wave equation.

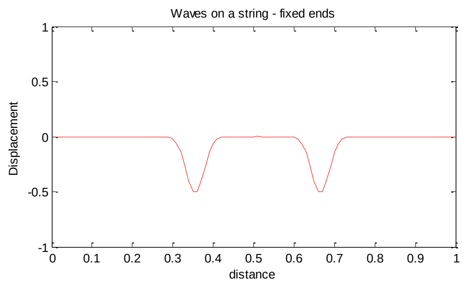
There are 2 files here

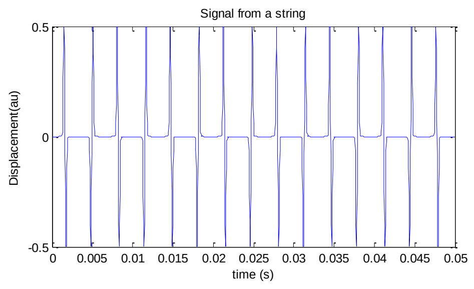
1. waves
2. propagate

```
% Solution of wave equation for string
%  based on 'Computational Physics' book by N Giordano and H Nakanishi
%
clear;
string_dimension=100;
time_loops=1500;
% Preallocate matrices for speed;
x=1/string_dimension:1/string_dimension:1;
x_scale=1:1:string_dimension;
y_next =zeros(1,string_dimension);
signal_data=zeros(1,time_loops);
elapsed_time=zeros(1,time_loops);
% Initialise string position
k=1000;
x_0=0.5;
 delta_t=3.33e-5;
 f_sample=1/delta_t;
initial_position=exp(-k.*(x-x_0).^2);
y_current =initial_position;
y_previous = initial_position;
initial_time=0;
time=initial_time;
for time_step = 1:time_loops;
        time=time+delta_t;
        [y_next]=propagate(y_current, y_previous);
        y_previous=y_current;
        y_current=y_next;
        clf;
        subplot(2,2,1);
        plot(x_scale/string_dimension, y_current,'r');
        title('Waves on a string - fixed ends');
        xlabel('distance');
        ylabel('Displacement');
        axis([0 1 -1 1]);
        hold on;
        drawnow;
        %%%%%%%

         % Record displacement at 5 percent from left end of the string for future plot
         signal_data(time_step)=y_current(5);
        elapsed_time(time_step)=time;
end;
        subplot(2,2,2);
        % plot displacement at 5 percent from left end of the string
        % using suitable scaling
         plot(elapsed_time,signal_data);
        title('Signal from a string');
```

```
          xlabel('time (s)');
          ylabel('Displacement(au)');

          % Generate FFT and calculate the power spectrum. The power spectral density,
          % a measurement of the energy at various frequencies, is equal
          % to the sum of the real and imaginary components of the
          % FFT. You can multiply the result of the FFT by its complex
          % conjugate in order to calculate it.

          f_sample=1/delta_t;

          NFFT = 2^(nextpow2(length(signal_data))); % No. of points in DFT=Next power of 2 up
from length of signal_data
          fft_value = fft(signal_data,NFFT);    % Perform (NFFT point) DFT padding out with zeros
so length of fft_value is NFFT

         Num_Unique_Pts=ceil((NFFT+1)/2); % only half points are unique due to nature of FFT.
Calculate number
         fft_value=fft_value(1:Num_Unique_Pts); % throw away half the points

          %Calculate the scaled power spectrum normalised by dividing
          % by the length of the signal data vector
            power_spectrum=  fft_value.*conj(fft_value)/(length(signal_data));

            % Since we dropped half the FFT, we multiply the power_spectrum by 2 to keep the same
energy.
          % The DC component and Nyquist component, if it exists, are unique and should not be
multiplied by 2.

                if rem(NFFT, 2) % odd NFFT excludes Nyquist point
                  power_spectrum(2:end) = power_spectrum(2:end)*2;
                else
                  power_spectrum(2:end -1) = power_spectrum(2:end -1)*2;
                end

           % Calculate scaled frequency scale

           f =  (0:Num_Unique_Pts-1)*f_sample/NFFT;
          subplot(2,2,3);

%         Power spectrum is symmetric so plot first half

          plot(f,power_spectrum,'g');
           axis([0 3000 0 6]);
           title('Power spectrum');
          xlabel('frequency (Hz)');
          ylabel('Power(au)');


%           Matlab offers the facility to play sounds. It would be nice to signal the end of program
execution with the tone that would be heard from
%           this string,. You would use the displacement from the vector 'signal_data' for this. Note
that 1500 samples are played in 0.05s,
%           therefore each sample should be played for 0.05/1500 =    3.3333e-05 seconds.The
sampling frequency is, therefore,1/3.3333e-05 = 30 kHz.
%           So the sampling rate is 1/3.3333e-5 =30 kHz

           sound(signal_data, 30e3);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
function [y_next] = propagate(y_current, y_previous)

 % c=300, delta_x=0.01, which makes delta_t = delta_x/c = 0.01/300 = 3.33e-5
 % since r=c*delta_t/delta_x and for the Figure 6.2 example
 % delta_t=delta_x/c, giving r=1.

r=1;
M=size(y_current,2);      % Vector size = number of columns

% there are 3 vectors containing positional information for the whole
% string,;    y_new, y_current and y_old
% preallocate memory for speed
% This function calculates the new shape of the string after one time step

% HERE IS THE ORIGINAL CODE
% for i=2:M-1;                              %This loop index takes care of the fact that the boundaries are
fixed
%       y_next(i) = 2*(1-r^2)*y_current(i)....
%                     -y_previous(i)....
%                     +r^2*(y_current(i+1)+y_current(i-1));
%     end;
% HERE IS THE VECTORISED CODE, WHICH IS MORE EFFICIENT

y_next=zeros(1,M);
 i=2:1:M-1;                             %This loop index takes care of the fact that the boundaries are
fixed
      y_next(i) = 2*(1-r^2)*y_current(i)....
                    -y_previous(i)....
                    +r^2*(y_current(i+1)+y_current(i-1));
```





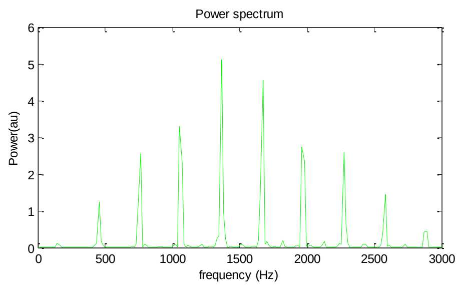


Figure 39. Signal from a vibrating string and Power spectrum. Signal excited with Gaussian pluck centred at the middle of the string and the displacement 5% from the end of the string was recorded.

7. Random Systems

7.1 Random walk simulation

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Solution of random walk problem
%  based on 'Computational Physics' book by N Giordano and H Nakanishi
% Solution by Kevin Berwick
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
clear;
number_of_walkers=500;
number_of_steps=100;
step_number=zeros(1,number_of_steps);
x2ave=zeros(1,number_of_steps);
step_number_array=[1:1:number_of_steps];

for  r=1:number_of_walkers;
%    initialise position
        x=0;
        y=0;

       for i=1:number_of_steps;

                  if rand<0.5;
                     x=x+1;
                  else
                     x=x-1;
                  end;
                  % Accumulate value of x^2 , the squared displacement,  for each step number

                  x2ave(i)=x2ave(i)+x^2;
      end;

end;

% Divide by number of walkers

x2ave= x2ave/number_of_walkers;
plot(step_number_array, x2ave, 'g');
title('Random walk');
xlabel('Step number');
ylabel('x^2');
```

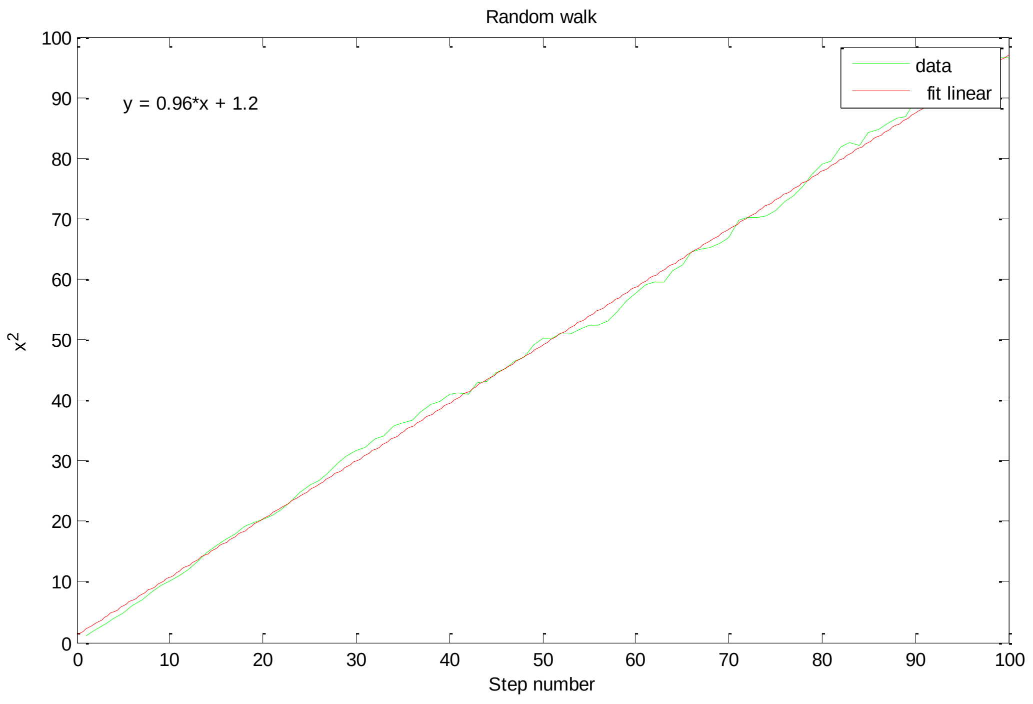


Figure 40. x^2 as a function of step number. Step length = 1. Average of 500 walks. Also shown is a linear fit to the data.

7.1.1 Random walk simulation with random path lengths.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Solution of random walk problem
%  based on 'Computational Physics' book by N Giordano and H Nakanishi
% Solution by Kevin Berwick
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
clear;
number_of_walkers=500;
number_of_steps=100;
step_number=zeros(1,number_of_steps);
x2ave=zeros(1,number_of_steps);
step_number_array=[1:1:number_of_steps];

for  r=1:number_of_walkers;
%    initialise position
        x=0;
        y=0;

       for i=1:number_of_steps;

                 if rand<0.5;
                    x=x+rand;
                 else
                    x=x-rand;
                 end;

                 % Accumulate value of x^2 , the squared displacement,  for each step number

                 x2ave(i)=x2ave(i)+x^2;
       end;

end;

% Divide by number of walkers

x2ave= x2ave/number_of_walkers;
plot(step_number_array, x2ave, 'g');
title('Random walk with random step length');
xlabel('Step number');
ylabel('x^2');
```

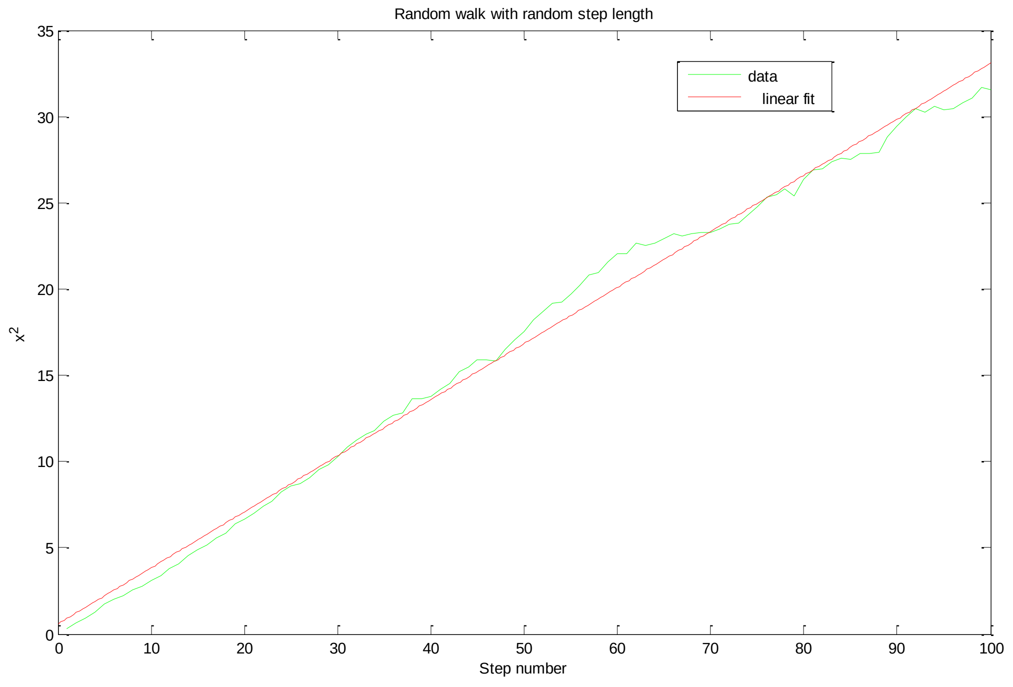


Figure 41. x^2 as a function of step number. Step length = random value betwen +/-1. Average of 500 walks. Also shown is a linear fit to the data.

10. Quantum Mechanics

10.2 Time independent Schrodinger equation. Shooting method.

Here we solve the time independent Schrodinger equation in one dimension for the particle in a box problem.

There are 2 files here

1. One_D_Schrodinger_Shooting
2. calculate_psi

Here is the code

```
%
%  Program to calculate wave function
%  based on 'Computational Physics' book by N Giordano and H Nakanishi
%  Section 10.2
%  by Kevin Berwick
%

%  Initialise

clear;
N=200;
delta_x=0.01;
E_initial=1.879;
delta_E=0.1;
x=(delta_x: delta_x: N*delta_x);

% Create half the potential well

V= zeros(1, N);
V(100:N) =1000;

% Create an intial vector to hold the wavefunction


b= 1.5;                        %  suggested cutoff parameter
%
% Implement psi_prime(0)=0 for an even parity solution by
% letting psi_in(0) and psi_in(-1)=0; Since this is the center of the well, we use indices 200 and 199
for these positions.
% Initialise last_diverge which keeps track of the diverging trend to zero
% since we don't know this direction yet

last_diverge=0;

% If delta_E is small enough then the current E is acceptable. We define a minimum value for this
quantity here

minimum_delta_E=0.005;

% initialise E

E=E_initial;

% MAIN LOOP
```

```
while abs(delta_E)>minimum_delta_E;
%    Initialise
   psi= zeros(1, N);
   psi(1)=1;
   psi(2)=1;

% Calculate wavefunction

  [psi,i]=calculate_psi(psi, N, delta_x, E, b,V);

  % Visualise results with movie
     plot(x, psi,'r');
     title('Square well');
     axis([0 2 -2 2]);
     xlabel('distance');
     ylabel('Wavefunction');
     drawnow;
     pause(0.5);

      if sign(psi(i+1))~=sign(last_diverge);
                    % If last value of psi evaluated before
                    % breakout from calculate_psi function and
                    % last diverge are of different signs, turn
                    % round direction of varying E and halve its
                    % value
        delta_E=-delta_E/2;
      end;
         E=E+delta_E
         last_diverge=sign(psi(i+1));
 end;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%  Function to calculate wave function
%  based on 'Computational Physics' book by N Giordano and H Nakanishi
%  Section 10.2
%  by Kevin Berwick
%

function [psi,i] = calculate_psi(psi, N, delta_x, E, b,V)
%This function calculates psi
% Make psi_prime(0) =0 for an even parity solution;

for i=2:N-1;
            psi(i+1)=2*psi(i)-psi(i-1)-2*(E-V(i))*delta_x^2*psi(i);
            if abs(psi(i+1)) > b; %  if psi is diverging, exit the loop;
                return;
            end;
end;
```

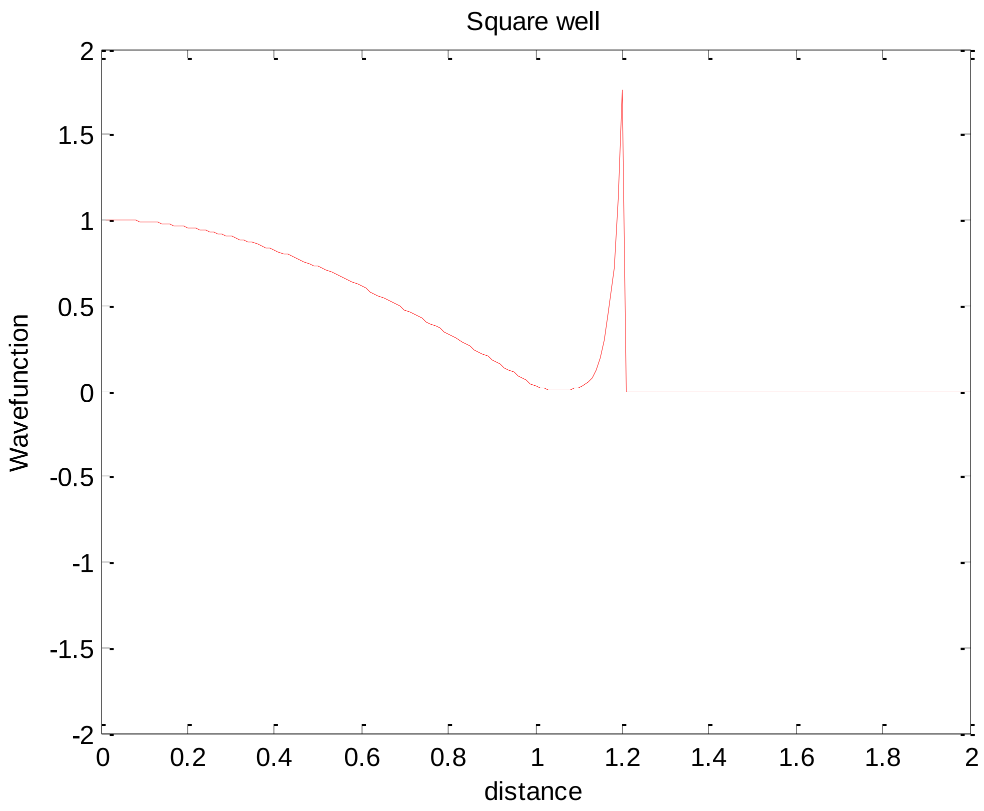


Figure 42. Calculated wavefunction using the shooting method. The wall(s) of the box are at x=(-)1. The value of Vo used was 1000 giving ground-state energy of 1.23. Analytical value is 1.233. Wavefunctions are not normalised.

10.5 Wavepacket construction

This is a little program to illustrate wavepacket construction. Here is the code.

```
%
% Program to illustrate wavepacket construction
% based on 'Computational Physics' book by N Giordano and H Nakanishi
% Section 10.5
% by Kevin Berwick
%

% Initialise and set up initial waveform

clear;
x=(0:0.0005:1);
x_0=0.4;
C=25;
sigma_squared=1e-3;
delta_x=0.0005;
delta_t=0.2;
k_0=500;

psi=C*exp(-(x-x_0).^2/sigma_squared).*exp(1i*k_0*x);

subplot(2,2,1);
```

```
plot(x,real(psi), 'b');
title('Real part of wavefunction ');
xlabel('distance');
ylabel('Re(wavefunction)');

subplot(2,2,2);
plot(x,imag(psi),'r');
title('Imaginary part of wavefunction');
xlabel('distance');
ylabel('Im(wavefunction)');

subplot(2,2,3);
plot(x,(conj(psi).*psi),'k');
title('Probability of finding particle  ');
xlabel('distance');
ylabel('psi*conj(psi)');
```

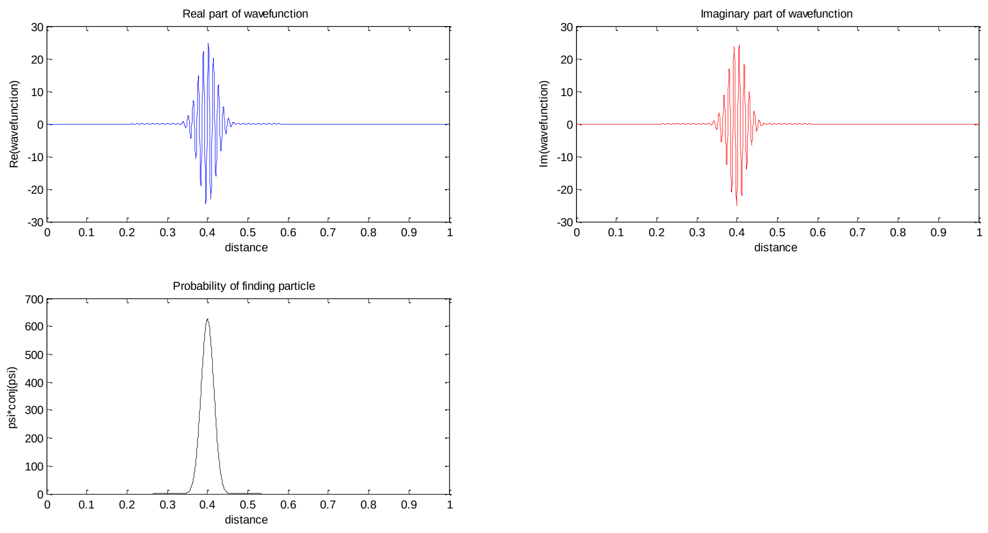


Figure 43. Composition of wavepacket. ko = 500, x0=0.4, sigma^2=0.001.

10.3 Time Dependent Schrodinger equation in One dimension. Leapfrog method.

Here we solve the time dependent Schrodinger equation in one dimension. A variety of scenarios can be modelled using this code. Representative results for a potential wall and cliff are presented .

There are 3 files here.

time_dep_SE_1D.m

real_psi.m

imag_psi.

```
% %
% %  Program to illustrate solution of Time Dependent Schrodinger equation
% % using leapfrog algorithm
% %  based on 'Computational Physics' book by N Giordano and H Nakanishi
% %  Section 10.5
% %  by Kevin Berwick
% %
%
% %  Initialise and set up initial waveform


clear;
N=1000;
x= linspace(0,1,N);

% Set up intial wavepacket;

x_0=0.4;
C=10;
sigma_squared=1e-3;
k_0=500;

% Discretisation parameters

delta_x=1e-3;
delta_t=5e-8;


%
%  Generate an intial wavepacket
%
 psi=C*exp(-(x-x_0).^2/sigma_squared).*exp(1i*k_0*x);

%
% % Extract the real and imaginary parts of the wavefunction
%
R_initial=real(psi);
I_initial=imag(psi);


%  Build a potential cliff . Create a region with a cliff  at x=0.6. To the
%  left, V=0, to the right,  V=-1e6.
```

```
V= zeros(1, N);
V(600:N) =-1e6;

% Initialise current real and imaginary parts of psi

 I_current=I_initial;
 R_current=R_initial;

% Initial run of Im(psi) to start off leapfrog process;

% t=t+delta_t/2;

[I_next] = imag_psi(N, I_current, R_current, delta_t, delta_x, V);

% % Do the leapfrog!!
%
 for time_step = 1:15000;


% evaluate R at delta_t, 2*delta_t, 3*delta_t.......
%     Time is incremented by  t=t+delta_t/2 every call;

    [R_next]=real_psi(N, R_current, I_current, delta_t, delta_x, V);

    R_current=R_next;
   % evaluate I at (3/2)*delta_t, (5/2)*delta_t............
   % Time is incremented by  t=t+delta_t/2 every call;

    [I_next] = imag_psi(N, I_current, R_current, delta_t, delta_x, V);

    % calculate psi*psi  with R(t) and  I(t+delta_t/2) and I(t-delta_t/2)

    prob_density=R_current.^2+I_next.*I_current;

    I_current=I_next;

% Visualise results with movie. Plot every 10 calculations for speed

      if rem(time_step, 10)== 0;

            plot(x, prob_density,'-b','LineWidth',2);
            title('Reflection from cliff');
            axis([0 1 0 200]);
            xlabel('x');
            ylabel('Probability density');
           drawnow;

      end;
 end;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%
% % Calculate the imaginary part of the wavefunction at time
% t=t+delta_t/2,, t + 3*delta_t/2 etc
% % given the value at time t.

function [I_next]= imag_psi(N, I_current, R_current, delta_t, delta_x, V)
 I_next= zeros(1,N);
s=delta_t/(2*delta_x^2);
```

```
for x=2:N-1;
    % Calculate the imaginary part of the wavefunction at time t=t+delta_t,
    % given the value at time t.
      I_next(x)=I_current(x) +s*(R_current(x+1)-2*R_current(x)+R_current(x-1))...
                        -delta_t*V(x).*R_current(x);
     % Boundary conditions

                       I_next(1)=I_next(2);
                       I_next(N)=I_next(N-1);

end;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% % Calculate the real part of the wavefunction at time t=t+delta_t,
% t+2*delta_t etc....
% % given the value at time t. Vectorise for speed.

function [R_next]= real_psi(N, R_current, I_current, delta_t, delta_x, V)
R_next= zeros(1,N);
s=delta_t/(2*delta_x^2);
for x=2:N-1;
    % Calculate the real part of the wavefunction at time t=t+delta_t,
    % given the value at time t. Vectorise for speed.
      R_next(x)=R_current(x) - s*(I_current(x+1)-2*I_current(x)+I_current(x-1))...
                        +delta_t*V(x).*I_current(x);

  % Boundary conditions

                       R_next(1)=R_next(2);
                       R_next(N)=R_next(N-1);

end;
```

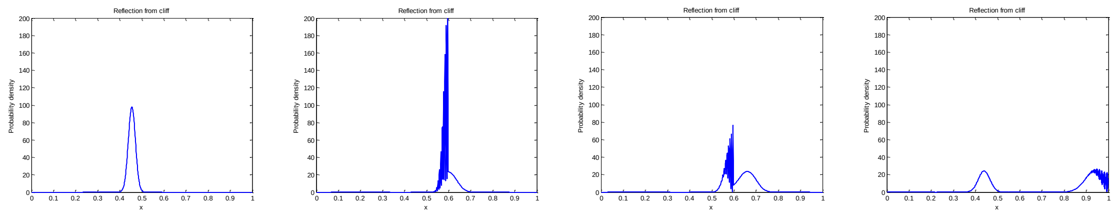


Figure 44. Wavepacket reflection from potential cliff at x=0.6. The potential was V=0 for x<0.6 and V=-1e6 for x>0.6. Values used for initial wavepacket were x_0=0.4,C=10, sigma_squared=1e-3, k_0=500. Simulation used delta_x=1e-3, delta_t=5e-8. Time progresses left to right.

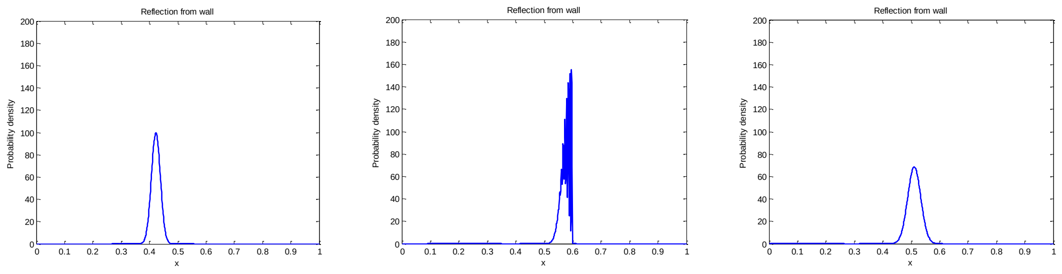


Figure 45. Wavepacket reflection from potential wall at x=0.6. The potential was V=0 for x<0.6 and V=1e6 for x>0.6. Values used for initial wavepacket were x_0=0.4,C=10, sigma_squared=1e-3, k_0=500. Simulation used delta_x=1e-3, delta_t=5e-8. Time progresses left to right.

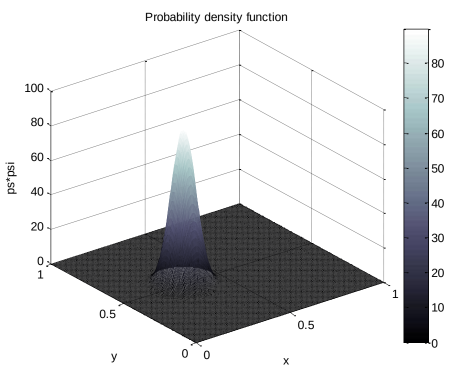
10.4 Time Dependent Schrodinger equation in two dimensions. Leapfrog method.

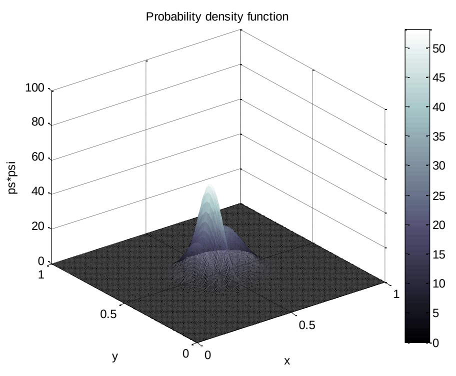
Here we extend the solution of the the time dependent Schrodinger equation to two dimensions. A variety of scenarios can be modelled using this code. Representative results for a potential wall and cliff are presented .

There are 3 files here.

1. time_dep_SE_2D.m
2. imag_psi_2D.m
3. real_psi_2D.m

```
% %
% %  Program to illustrate solution of 2D Time Dependent Schrodinger equation
% % using leapfrog algorithm
% %  based on 'Computational Physics' book by N Giordano and H Nakanishi
% %  Section 10.5
% %  by Kevin Berwick
% %
%
% %  Initialise and set up initial waveform


clear;
N=200;

% Set up intial wavepacket;

x_0=0.25;
y_0=0.5;
C=10;
sigma_squared=0.01;
k_0=40;

% Discretisation parameters

delta_x=1/200;
delta_t=0.00001;

%
% Build a mesh for the values of the probability density function

a= linspace(0, 1, N);

%Use meshgrid to calculate the grid matrices for the x- and y-coordinates,
%using same resolution and scales in x and y directions.

[x,y] = meshgrid(a);


% Create a 2D potential cliff

V=zeros(N,N);
V(:, 100:200)=-1e3;
% % Create a 2D potential wall
% V=zeros(N,N);
% V(:, 100:200)=1e3;
% Calculate psi

 psi_stationary=C*exp(-(x-x_0).^2/sigma_squared).*exp(-(y-y_0).^2/sigma_squared);
 plane_wave = exp(1i*k_0*x);%+1i*k_0*y);
 psi_z=psi_stationary.*plane_wave;


% % % Extract the real and imaginary parts of the wavefunction
 %
 R_initial=real( psi_z);
 I_initial=imag( psi_z);


% % Initialise current real and imaginary parts of psi
%
 I_current=I_initial;
 R_current=R_initial;
%
% % Initial run of Im(psi) to start off leapfrog process;
%
%
[I_next] = imag_psi(N, I_current, R_current, delta_t, delta_x, V);
%
% % % Do the leapfrog!!
% %
 for time_step = 1:2000;

% evaluate R at delta_t, 2*delta_t, 3*delta_t.......
%    Time is incremented by  t=t+delta_t/2 every call;

    [R_next]=real_psi_2D(N, R_current, I_current, delta_t, delta_x, V);

    R_current=R_next;

   % evaluate I at (3/2)*delta_t, (5/2)*delta_t............
   % Time is incremented by  t=t+delta_t/2 every call;

    [I_next] = imag_psi_2D(N, I_current, R_current, delta_t, delta_x, V);

    % calculate psi*psi  with R(t) and  I(t+delta_t/2) and I(t-delta_t/2)

    prob_density=R_current.^2+I_next.*I_current;

    I_current=I_next;

% Visualise results with movie. Plot every 10 timesteps for speed

      if rem(time_step, 10)== 0;

            surf(x,y, prob_density);
            title('Probability density function');
            xlabel('x');
            ylabel('y');
            zlabel('ps*psi');
             axis([0 1 0 1 0 100]);
             view(3);
             axis on;
             grid on;
             colormap('bone');
             light;
            lighting phong;
            camlight('left');
            shading interp;
            colorbar;
            drawnow;

        end;
 end;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%




%
% % Calculate the imaginary part of the wavefunction at time
% t=t+delta_t/2,, t + 3*delta_t/2 etc
% % given the value at time t. Vectorise for speed.

function [I_next]= imag_psi_2D(N, I_current, R_current, delta_t, delta_x, V)
I_next= zeros(N,N);
s=delta_t/(2*delta_x^2);
 x=2:N-1;
  y=2:N-1;
    % Calculate the imaginary part of the wavefunction at time t=t+delta_t,
    % given the value at time t.
      I_next(x,y)=I_current(x,y) +s*(R_current(x+1,y)-2*R_current(x,y)+R_current(x-
1,y)+R_current(x,y+1)-2*R_current(x,y)+R_current(x,y-1))...
                           -delta_t*V(x,y).*R_current(x,y);


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% % Calculate the real part of the wavefunction at time t=t+delta_t,
% t+2*delta_t etc....
% % given the value at time t. Vectorise for speed.

function [R_next]= real_psi_2D(N, R_current, I_current, delta_t, delta_x, V)
R_next= zeros(N,N);
s=delta_t/(2*delta_x^2);
 x=2:N-1;
  y=2:N-1;
    % Calculate the real part of the wavefunction at time t=t+delta_t,
    % given the value at time t.
      R_next(x,y)=R_current(x,y) - s*(I_current(x+1,y)-2*I_current(x,y)+I_current(x-
1,y)+I_current(x,y+1)-2*I_current(x,y)+I_current(x,y-1))...
                           +delta_t*V(x,y).*I_current(x,y);
```





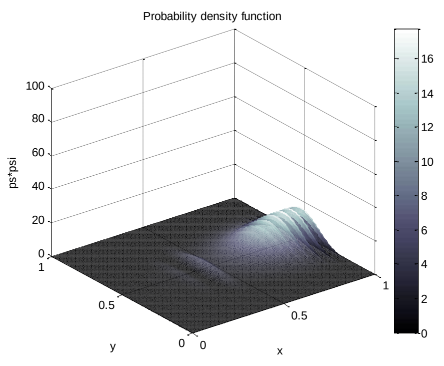
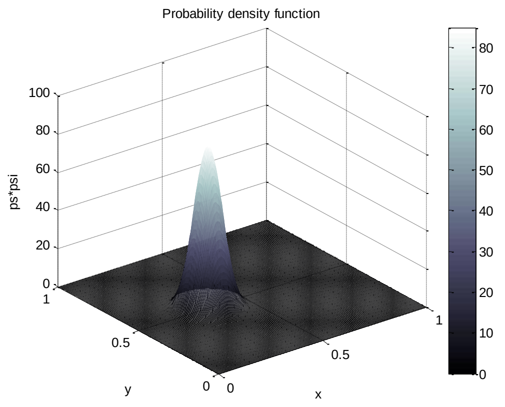
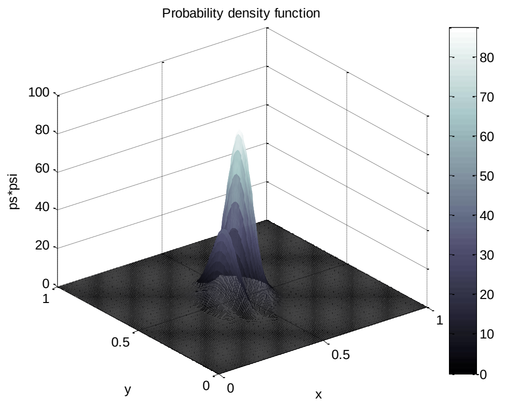


Figure 46.Wavepacket reflection from potential cliff at x=0.5. The potential was V=0 for x<0.5 and V=-1e3 for x>0.5. Values used for initial wavepacket were x_0=0.25, y_0=0.5,C=10, sigma_squared=0.01, k_0=40. Simulation used delta_x=0.005, delta_t=0.00001.





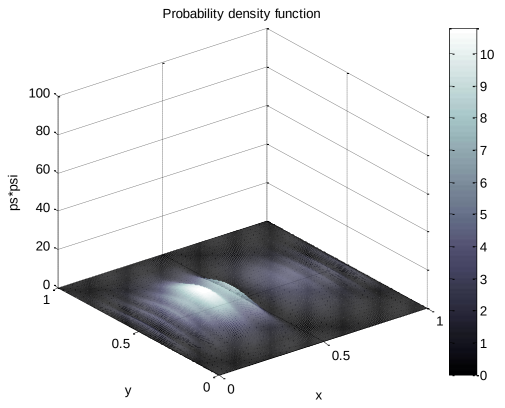


Figure 47. Wavepacket reflection from potential wall at x=0.5. The potential was V=0 for x<0.5 and V=1e3 for x>0.5. Values used for initial wavepacket were x_0=0.25, y_0=0.5,C=10, sigma_squared=0.01, k_0=40. Simulation used delta_x=0.005, delta_t=0.00001.