Phase spaces of Dynamical Systems

Tomack Gilmore

1 Approximating the phase space of a pendulum

The oscillating motion of a pendulum of length l is described by the equation

$$\ddot{\theta}(t) = -\frac{g}{l}\sin(\theta(t)), \tag{1}$$

where g is the gravitational constant. If we let $\gamma(t) = \dot{\theta}(t)$ be the angular velocity of the pendulum then (1) may be written as the following system of equations

$$\frac{d}{dt}\begin{pmatrix} \theta(t) \\ \gamma(t) \end{pmatrix} = \begin{pmatrix} \gamma(t) \\ -\omega^2 \sin(\theta(t)) \end{pmatrix} = \begin{pmatrix} F_1(\theta(t),\gamma(t)) \\ F_2(\theta(t),\gamma(t)) \end{pmatrix} = F(\theta(t),\gamma(t)). \tag{2}$$

We use the following

$$f(x,y) \approx f(a,b) + \left.\frac{\partial f(x,y)}{\partial x}\right|_{a,b}(x-a) + \left.\frac{\partial f(x,y)}{\partial y}\right|_{a,b}(y-b)$$

to linearise F around a given point (a,b). If we let $\theta^* = \gamma^* = 0$ we obtain

$$F_1(\theta+\theta^*, \gamma+\gamma^*) = F_1(\theta,\gamma) = \gamma$$

and

$$F_2(\theta,\gamma) = -\frac{g}{l}\theta,$$

so that the linear approximation of F as a first-order system can be written as

$$\frac{d}{dt}\begin{pmatrix} \theta \\ \gamma \end{pmatrix} \approx \begin{pmatrix} \gamma \\ -\omega^2\theta \end{pmatrix}.$$

The following code defines the function from (2):

```
function xdot = q1eq1( t,x )
g=9.80665;% This is the gravitational acceleration.
xdot(1)=x(2);%This calculates F_1 in terms of initial conditions.
xdot(2)=-g*sin(x(1));%This calculates F_2 in terms of initial conditions.
xdot=xdot';%This returns a vector.
end
```

We then call the following procedure (q1proc1.m) which produces a plot of $\theta(t)$ for $l = 1$, $\gamma(0) = 0$ and various initial angles $\theta = \pi/6, \pi/3$ and $3\pi/4$ (see Figure 1):

```
t=0;%intialises time at 0
T=10;%specifies size of time interval
dt=0.01;%specifies time steps
x0(1)=pi/6;%specifies initial condition theta(0)=pi/6
x0(2)=0;%specifies initial condition gamma(0)=0
[ts1, sol1]=ode45('q1eq1', [t:dt:T], x0);%solves for initial conditions
x0(1)=pi/3;%changes initial condition so that theta(0)=pi/3
[ts2, sol2]=ode45('q1eq1', [t:dt:T], x0);%solves for new initial conditions
x0(1)=3*pi/4;%changes intitial condition so that theta(0)=3pi/4
[ts3, sol3]=ode45('q1eq1', [t:dt:T], x0);%solves for new initial conditions
plot(ts1,sol1(:,1),ts2,sol2(:,1),ts3,sol3(:,1))%plots trajectories for theta
```

We then execute the following commands to produce the plots of $\gamma(t)$ and the phase space trajectory in Figure 1:

```
>> plot(ts1,sol1(:,2),ts2,sol2(:,2),ts3,sol3(:,2))
>> plot(sol1(:,1),sol1(:,2),sol2(:,1),sol2(:,2),sol3(:,1),sol3(:,2))
```

It is clear from these plots that as we increase the initial angle the frequency of both $\gamma(t)$ and $\theta(t)$ decreases and the amplitude increases. Also the amplitude of $\gamma(t)$ increases much quicker than the amplitude of $\theta(t)$. This implies that if we increase the initial angle then the pendulum will oscillate less frequently about the point of equilibrium as one would expect.

We now define a new function that calculates the linearisation of F:

```
function xdot = q1eq2( t,x )
g=9.80665;%set gravitational acceleration
xdot(1)=x(2);%calculates linear approx. of F_1 in terms of initial condition
xdot(2)=-g*x(1);%calculates linear approx of F_2 in terms of initial cond.
xdot=xdot';%returns vector
end
```

We call it in the following procedure (q1proc2.m):

```
t=0;%intialises time at 0
T=6;%specifies time interval
dt=0.01;%specifies increment length of time
x0(2)=0;%sets initial condition of gamma(0)=0
x0(1)=pi/4;%sets initial condition theta(0)=pi/4
[ts1, sol1]=ode45('q1eq1', [t:dt:T], x0);%solves ODE for above conditions
[tsl1,soll1]=ode45('q1eq2',[t:dt:T], x0);%solves linear approx. for above
                                          %conditions
```

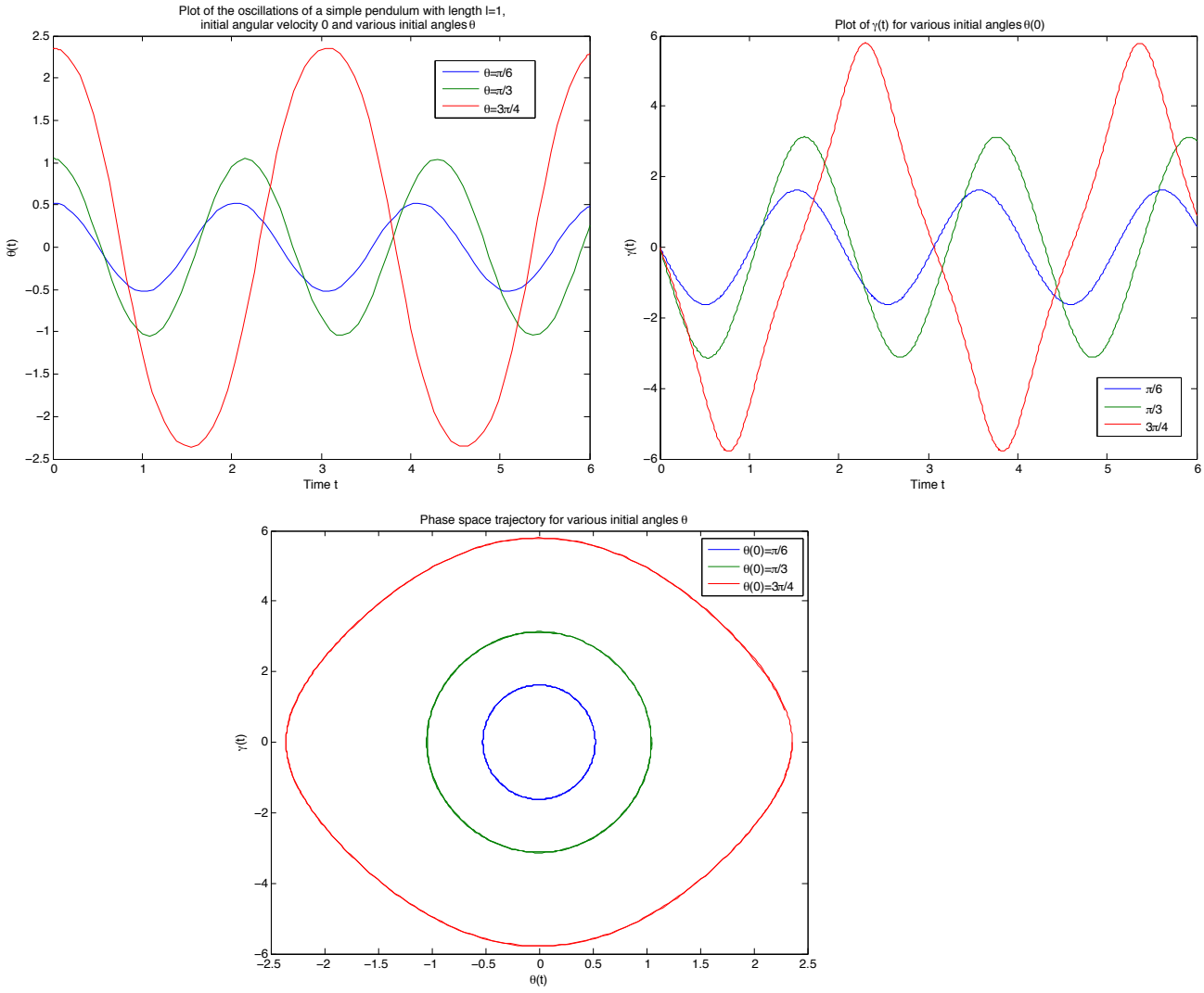
Plot of the oscillations of a simple pendulum with length l=1,
initial angular velocity 0 and various initial angles θ
θ=π/6
θ=π/3
θ=3π/4
θ(t)
Time t
Plot of γ(t) for various initial angles θ(0)
π/6
π/3
3π/4
γ(t)
Time t
Phase space trajectory for various initial angles θ
θ(0)=π/6
θ(0)=π/3
θ(0)=3π/4
γ(t)
θ(t)

Figure 1

```
x0(1)=pi/3;%sets new theta(0)=pi/3
[ts2, sol2]=ode45('q1eq1', [t:dt:T], x0);%solves ODE for above conditions
[tsl2, soll2]=ode45('q1eq2', [t:dt:T], x0);%solves linear approx. for above
                                                %conditions
x0(1)=3*pi/5;%sets new theta(0)=3pi/5
[ts3, sol3]=ode45('q1eq1', [t:dt:T], x0);%solves ODE for above conditions
[tsl3, soll3]=ode45('q1eq2', [t:dt:T], x0);%solves linear approx. for above
                                                %conditions
```

We then execute the following commands to plot the trajectories of F and its linear approximation for various initial angles (see Figure 2):

```
>> plot(sol1(:,1),sol1(:,2),soll1(:,1),soll1(:,2))
>> plot(sol2(:,1),sol2(:,2),soll2(:,1),soll2(:,2))
>> plot(sol3(:,1),sol3(:,2),soll3(:,1),soll3(:,2))
```

Clearly as the initial angle increases from zero our linear approximation of the trajectory deviates further and further from the true trajectory of the system. This would make sense as our linear approximation was about $\theta(0) = 0$, so the further away our initial angle is from 0 the less accurate our approximation will be.

The total energy $E(t)$ of the nonlinear pendulum can be written as the sum of its kinetic energy $K(t) = (ml^2\gamma(t)^2)/2$, and its potential energy $U(t) = mgl(1 - \cos(\theta(t))$

$$E(t) = \frac{ml^2\gamma^2}{2} + mgl(1 - \cos\theta) \tag{3}$$

Differentiating (3) with respect to t gives:

$$\dot{E} = ml^2\gamma(\gamma') + \theta' mgl\sin\theta = 0, \tag{4}$$

as $l = m = 1$, $\gamma = \theta'$ and $\gamma' = -g\sin\theta$. We see that the energy of the system is conserved.

We now implement Euler's method for (1) by calling the following procedure (eulerq1.m):

```
t0=0;%initialises time
T=10;%specifies size of time interval
dt=0.001;%specifies size of time step
n=(T-t0)/dt;%specifies number of steps
tspan=[t0:dt:T];%specifies time range as a vector
clear x;
clear y;

x0=pi/6;%sets initial angle theta
y0=0;%sets initial angle gamma
g=9.80665;%sets gravitational constant
```

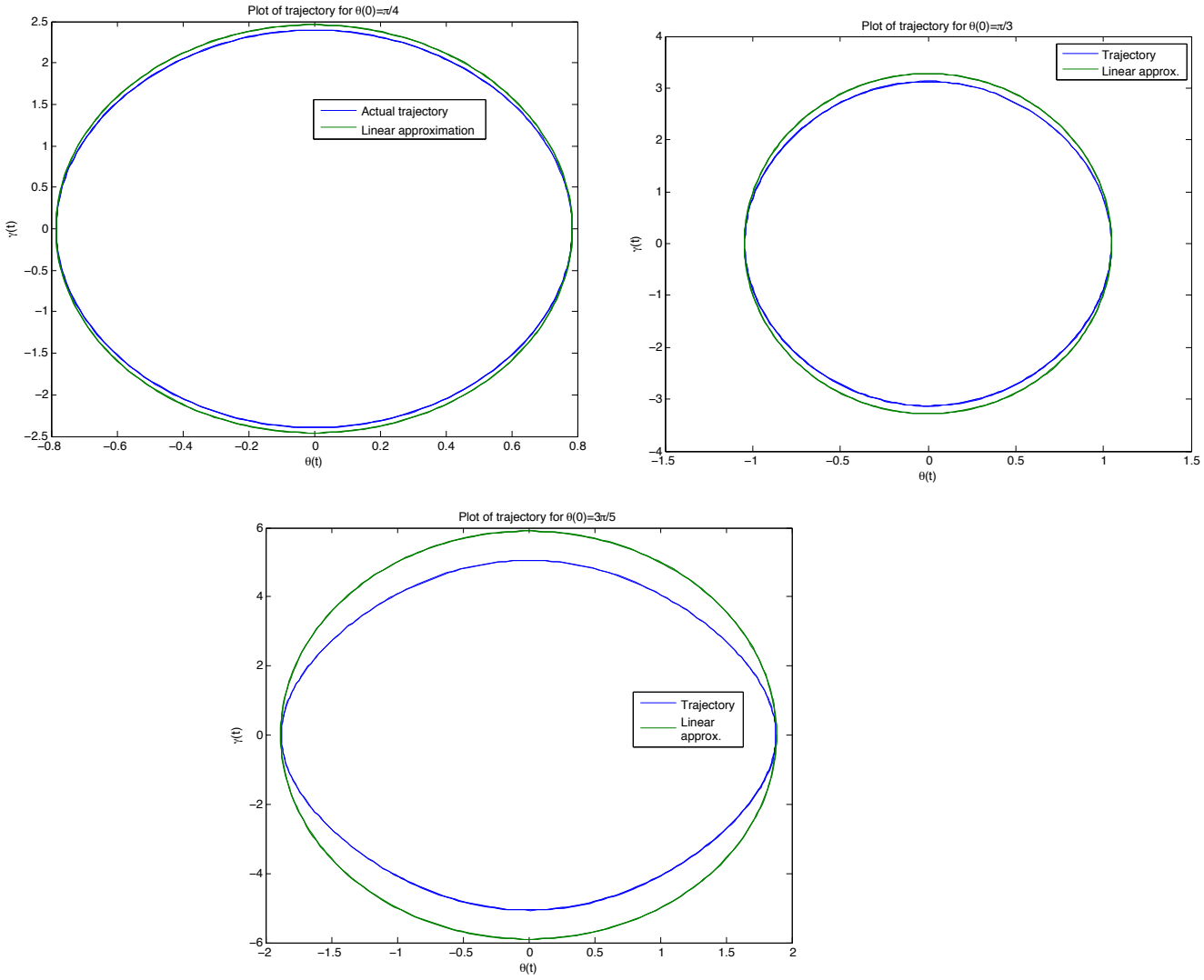
Plot of trajectory for θ(0)=π/4
Actual trajectory
Linear approximation
γ(t)
θ(t)
Plot of trajectory for θ(0)=π/3
Trajectory
Linear approx.
γ(t)
θ(t)
Plot of trajectory for θ(0)=3π/5
Trajectory
Linear approx.
γ(t)
θ(t)

Figure 2

```
x(1)=x0;%specifies initial angle theta as first entry in solution vector x
y(1)=y0;%specifies initial angle gamma as first entry in solution vector y
for i=1:n%initialises counter to run from 1 to n
    x(i+1)=x(i)+dt*y(i);%solves for each theta as time progresses
    y(i+1)=y(i)+dt*(-g*sin(x(i)));%solves for each gamma as time progresses
end
```

We then execute the following commands to generate the plots found in Figure 3 for various time integration steps:

```
>> q1proc1
>> eulerq1
>> plot(tspan,x,tspan,y,ts1,sol1(:,1),ts1,sol1(:,2))
```

Clearly as we decrease the size of the time steps the Euler method gets much closer to our solution that uses ode45, however it is clear that as time increases the Euler method will deviate fairly rapidly away from the more accurate solution obtained using ode45. We can also plot the phase space trajectories (Figure 4) for different size time steps by executing the following command:

```
>>plot(x,y,sol1(:,1),sol2(:,2))
```

Again we can clearly see that as time increases the Euler method spirals away from the solution obtained by ode45, which would wrongly imply that the energy of the system is not conserved. As we increase the number of time steps we obtain a better estimate, but the smaller time step we use the more time it takes to compute the Euler estimate of the solution for the system.

We now implement the Runge-Kutta method by calling the following procedure (rkq1.m)

```
t0=0;%sets initial time
T=10;%specidfies size of time interval
dt=0.001;%specisfies size of time steps
n=(T-t0)/dt;%specifies number of time steps
tspan=[t0:dt:T];%specifies time range as a vector
clear x;
clear y;

x0=pi/6;%sets initial condition for theta
y0=0;%sets initial condition for gamma
g=9.80665;%sets the gravitational constant

x(1)=x0;%sets initial condition as first entry in the solution vextor x
y(1)=y0;%sets initial condition as first entry in the solution vextor y
```

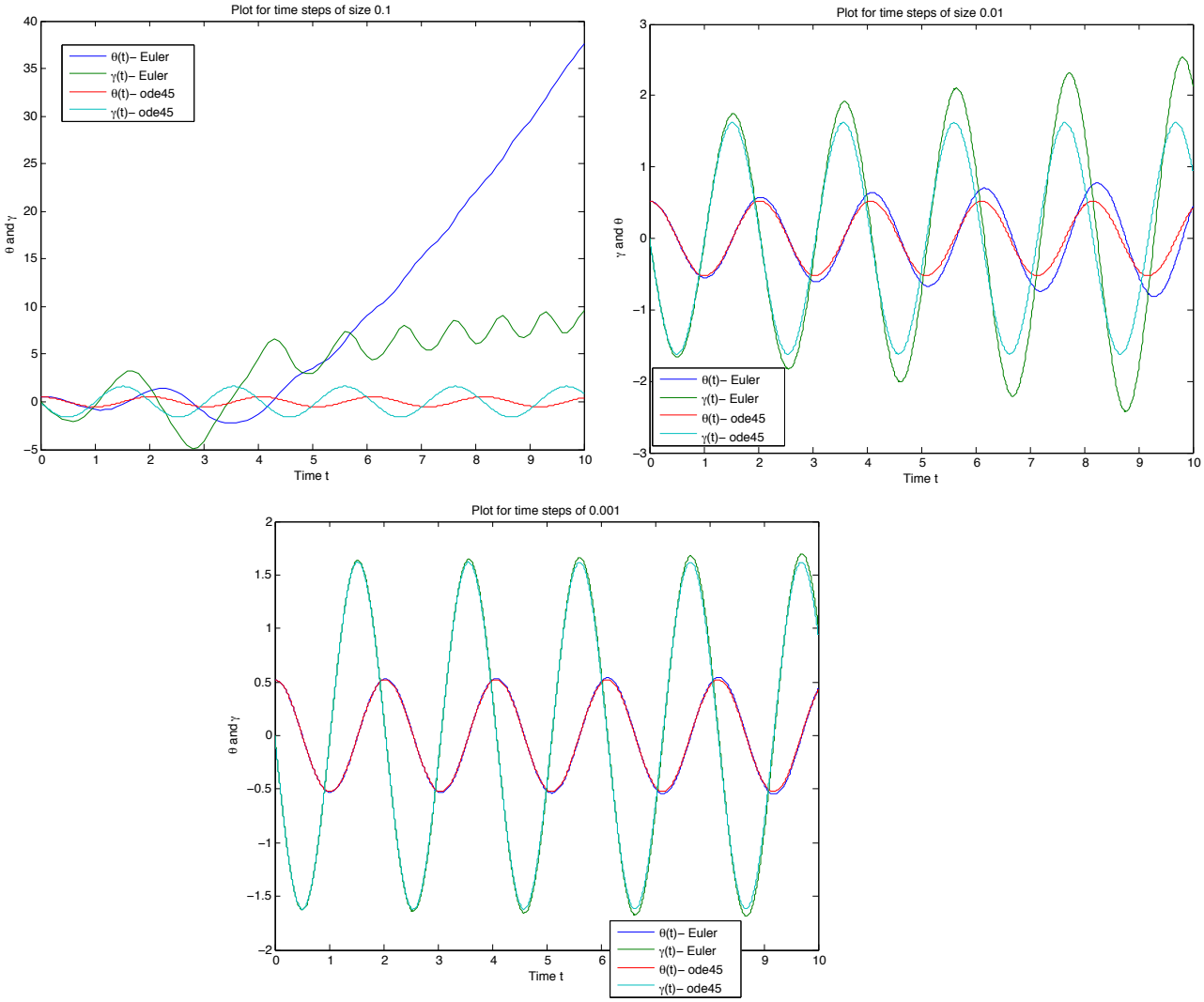
Plot for time steps of size 0.1
θ and γ
Time t
θ(t)– Euler
γ(t)– Euler
θ(t)– ode45
γ(t)– ode45
Plot for time steps of size 0.01
γ and θ
Time t
θ(t)– Euler
γ(t)– Euler
θ(t)– ode45
γ(t)– ode45
Plot for time steps of 0.001
θ and γ
Time t
θ(t)– Euler
γ(t)– Euler
θ(t)– ode45
γ(t)– ode45

Figure 3

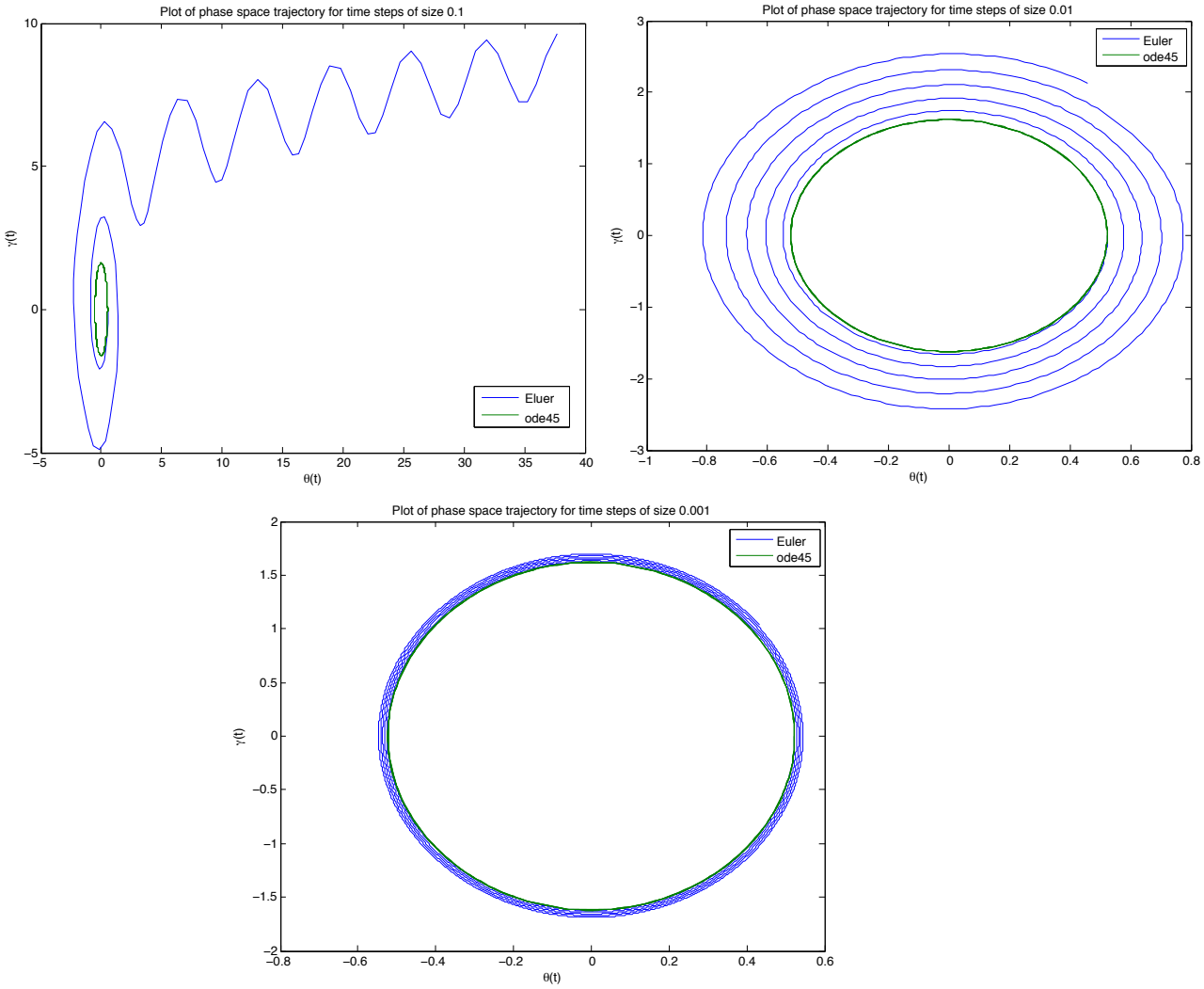
Plot of phase space trajectory for time steps of size 0.1
Eluer
ode45
γ(t)
θ(t)
Plot of phase space trajectory for time steps of size 0.01
Euler
ode45
γ(t)
θ(t)
Plot of phase space trajectory for time steps of size 0.001
Euler
ode45
γ(t)
θ(t)

Figure 4

```
for i=1:n%initialises counter from 1 to n
    ax=y(i);%calculates left estimate of force for theta
    bx=y(i)+ax*dt/2;%estimates midpoint force
    cx=y(i)+bx*dt/2;%refines estimate of midpoint force
    dx=y(i)+cx*dt/2;%refines estimate of midpoint force
    x(i+1)=x(i)+(ax+2*bx+2*cx+dx)*dt/6;%estimates force on the right side
                                        %of the time interval
    ay=-g*sin(x(i));%as above for gamma
    by=-g*sin(x(i)+ax*dt/2);
    cy=-g*sin(x(i)+bx*dt/2);
    dy=-g*sin(x(i)+cx*dt/2);
    y(i+1)=y(i)+(ay+2*by+2*cy+dy)*dt/6;
end
```

We plot the trajectories for this method (Figure 5) by executing the following commands:

```
>> plot(tspan,x,tspan,y,ts1,sol1(:,1),ts1,sol1(:,2))
>> plot(x,y,sol1(:,1),sol1(:,2))
```

Clearly the Runge-Kutta method is a much better approximation for the solution than the one obtained by Euler's method. Both the phase space and trajectory for the Runge-Kutta method is a closer fit to the solution obtained by ode45 for the same initial conditions. However both methods show that energy is slowly dissipated as time goes on. We have shown above that the energy is conserved (in which case the phase space trajectory is an ellipse), however the result in Figure 4 is to be expected given that we are numerically estimating the solution.

2 The Lotka-Volterra predator-prey model

The Lotka-Volterra predator-prey model is given by the following differential equations:

$$\begin{aligned} \dot{B} &= aB - bBL \\ \dot{L} &= -cL + dBL, \end{aligned} \tag{5}$$

where B is the number of baboons, L the number of leopards and a, b, c, $d > 0$. The first equation calculates the rate of growth over time of the baboon population, the second for leopards. The lower case letters all represent certain population parameters. The birth rate of the baboons is denoted by a, b denotes the mortality rate of the baboons per leopard- it can be thought of the average "hunting ability" of the leopards. We also have the mortality rate of the leopards denoted by c and d denotes the reproductive rate of the leopards according to how many baboons they capture i.e. how many leopards we can make out of a certain number of baboons. The model makes a number of unrealistic assumptions- mainly that there is no limit on the amount of food available to the baboons;

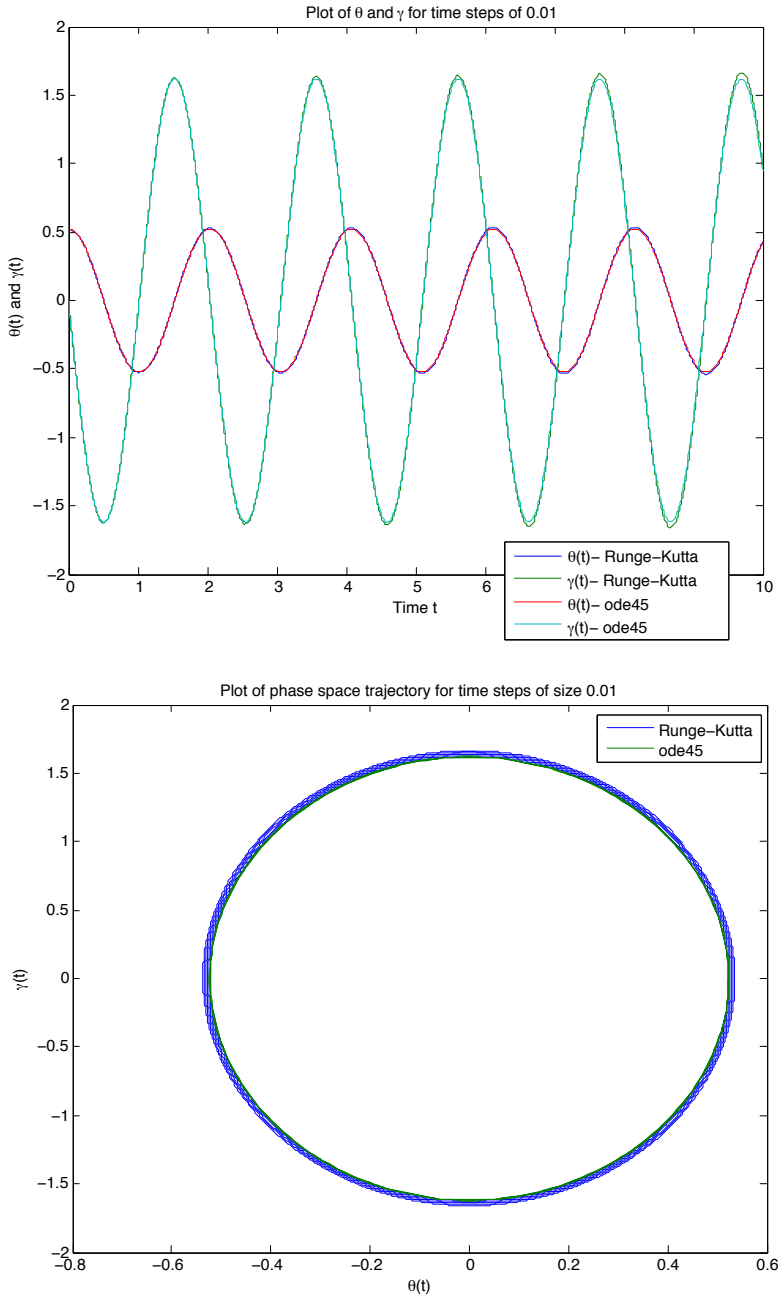

Figure 5: Trajectories for Runge-Kutta method with $x(0) = \pi/6$ and $y(0) = 0$

that the leopards only ever eat baboons; that the rate of change of the population is dependent on its size; and that the environment is constant and never changes in favour of either species.

In order to find the fixed points of the system we consider

$$\begin{aligned} aB^* - bL^*B^* &= 0, \\ dL^*B^* - cL^* &= 0. \end{aligned}$$

We see immediately that $B^* = c/d$ and $L^* = a/b$. Now letting $x = B/B^*$ and $y = L/L^*$ we can rewrite (5) as

$$\begin{aligned} x' &= ax(1-y) \\ y' &= cy(x-1), \end{aligned} \quad . \tag{6}$$

By letting $s = at$ we obtain

$$\frac{1}{a}\frac{d}{ds}\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x(1-y) \\ \mu y(x-1) \end{pmatrix} \tag{7}$$

where $\mu = c/a$.

We can plot the model by first defining the following function (ltkavolt.m):

```
function xdot = lotkavolt( t ,x )
mu=4.5;%sets mu parameter value
xdot(1)=x(1)*(1-x(2));%calculates the rate of growth of baboons
xdot(2)=mu*x(2)*(x(1)-1);%calculates the rate of growth of leopards
xdot=xdot';%returns a vector
end
```

We call the function in the following procedure (lotkavoltproc.m)

```
t=0;%sets initial time
T=30;%sets end of time interval
dt=0.01;%sets size of time increment
clear ts;%clears ts
clear sol;%clears sol

x0(1)=6;%sets initial population of baboons
x0(2)=4;%sets initial population of leopards

[ts, sol]=ode45('lotkavolt', [t:dt:T], x0);%solves differential equation
plot(ts,sol(:,1),ts,sol(:,2))%returns trajectory of baboons and leopards
                             %over time
```

This produces the plots found in Figure 6 from which it is clear that we the model predicts cycles. However these plots also show how unrealistic the model is for at times

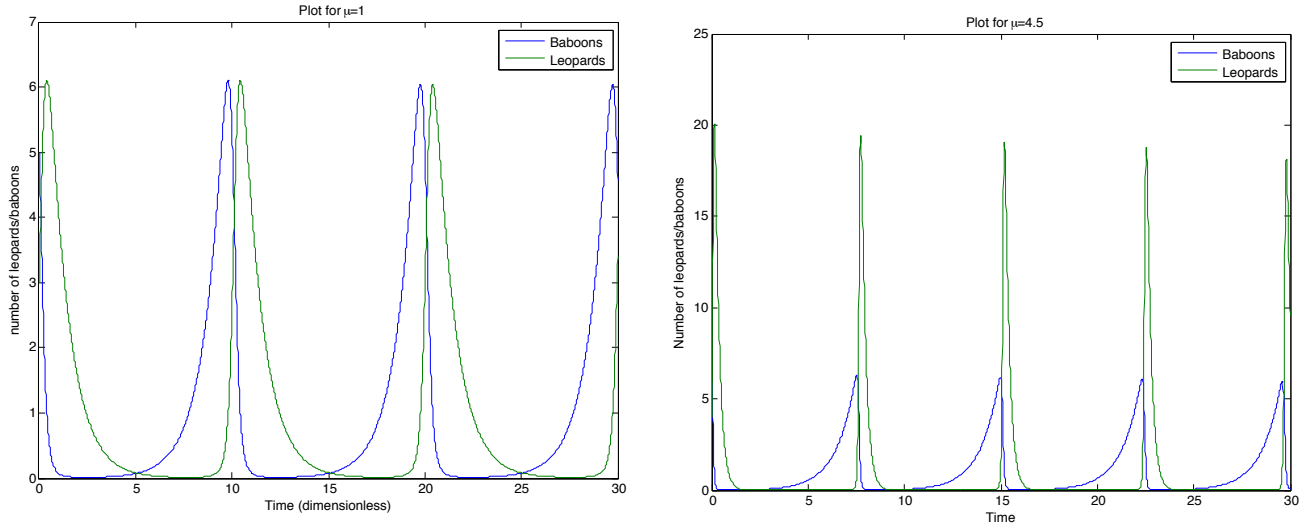

Figure 6: (Left) plot of trajectory for initial population of 6 baboons and 4 leopards where $\mu = 1$. (Right) Plot for initial population of 5 baboons, 4 leopards and $\mu = 4.5$

the population of leopards and baboons both go almost to zero and then shoot up again. We can see this by considering the phase space trajectory for the same conditions (Figure 7). Clearly the system frequently returns to zero in both cases, which does not make sense intuitively. It does not seem reasonable to suddenly go from 0 baboons to lots of baboons/leopards so quickly.

To find the conserved quantity in terms of the dimensionless variables consider equation (7) again which can be written as

$$\frac{1}{xya}\frac{dx}{ds}\frac{dy}{ds} = \frac{dy}{ds}\left(\frac{1}{y} - 1\right) \quad \text{and} \quad \frac{1}{xya}\frac{dy}{ds}\frac{dx}{ds} = \frac{dx}{ds}\left(\mu - \frac{1}{x}\right).$$

Taking the difference of these equations we have that $0 = \frac{dy}{ds}\left(\frac{1}{y} - 1\right) + \frac{dx}{ds}\left(\frac{1}{x} - \mu\right)$. Now we need to find a function $H(x, y)$ such that

$$\frac{d}{ds}H(x, y) = \frac{dy}{ds}\frac{\partial}{\partial y}H(x, y) + \frac{dx}{ds}\frac{\partial}{\partial x}H(x, y) = 0.$$

So we need a function such that $\frac{\partial}{\partial y}H(x, y) = \frac{1}{y} - 1$ and $\frac{\partial}{\partial x}H(x, y) = \frac{1}{x} - \mu$. Integrating we obtain $H(x, y) = \ln y - y + c_y(x) = \ln x - \mu x + c_x(y)$ so we can conclude that the quantity $H(x, y) = \ln y + \ln x - \mu x - y$ is conserved for the system.

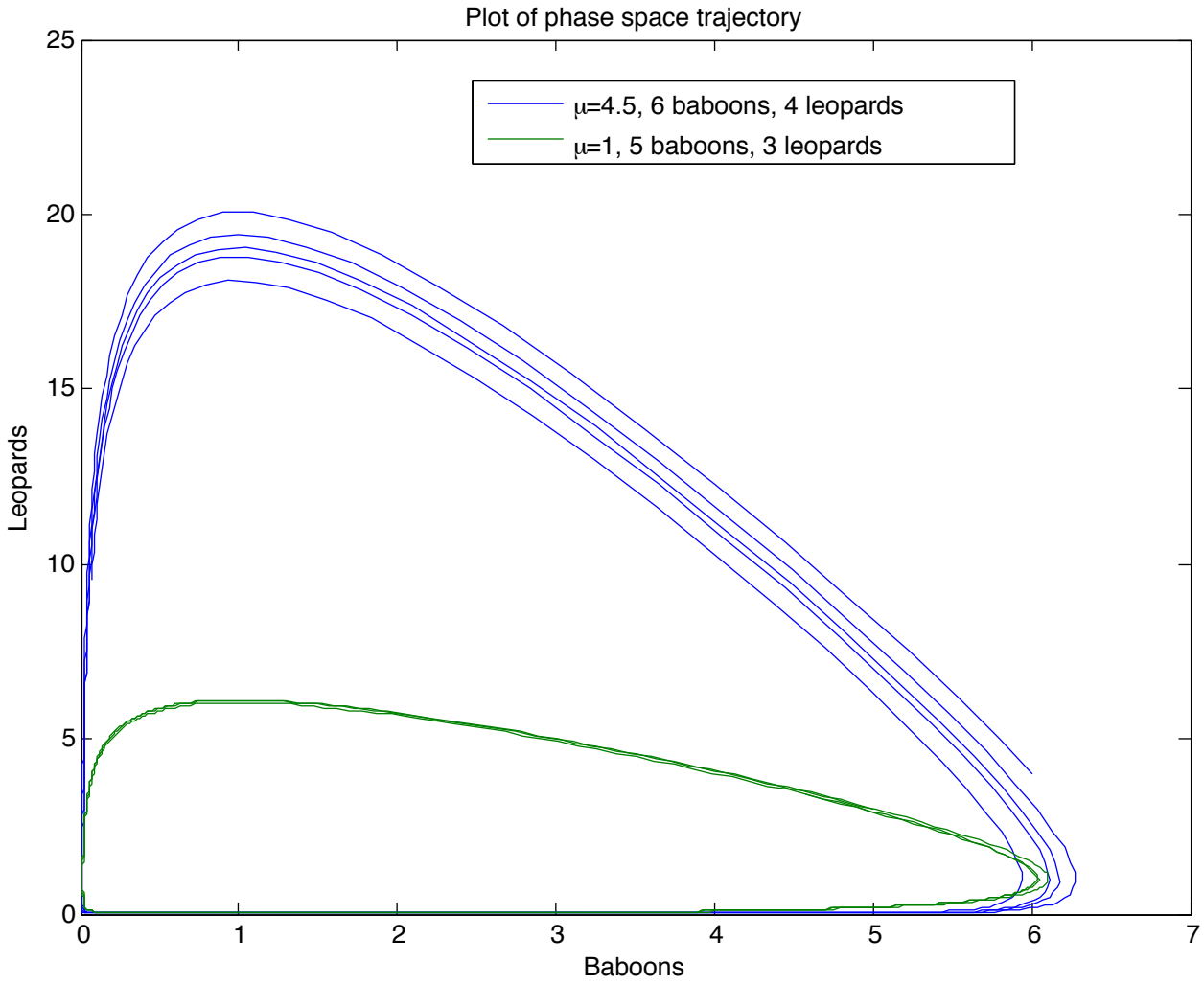
Plot of phase space trajectory
μ=4.5, 6 baboons, 4 leopards
μ=1, 5 baboons, 3 leopards
Leopards
Baboons
0
5
10
15
20
25
0
1
2
3
4
5
6
7

Figure 7

3 The Lorenz attractor

The Lorenz equations are given in vector notation by

$$\frac{d}{dt}\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} \sigma(y-x) \\ x(\rho-z)-y \\ xy-\beta z \end{pmatrix}.$$

We plot the trajectory for each component by first defining the following function (lorenz.m):

```
function xdot = lorenz( t,x )
sigm=10;%sets the vaue of sigma
rho=28;%sets the value of rho
bet=8/3;%sets the value of beta
xdot(1)=sigm*(x(2)-x(1));%calculates x'
xdot(2)=x(1)*(rho-x(3))-x(2);%calculates y'
xdot(3)=x(1)*x(2)-bet*x(3);%calculates z'
xdot=xdot';%returns solution as a vector
end
```

which we call in the following procedure (lorenzproc.m):

```
t=0;%sets initial time
T=50;%sets size of time interval
dt=0.001;%sets size of time increment

x0(1)=0.1;%sets initial value for x
x0(2)=0.1;%sets initial value for y
x0(3)=0.1;%sets initial value for z

[ts, sol]=ode45('lorenz', [t:dt:T], x0);%solves system numerically
```

and we plot the trajectories for each component by executing the following commands:

```
>> plot(ts,sol(:,1))
>> plot(ts,sol(:,2))
>> plot(ts,sol(:,3))
```

This results in the plots found in Figure 8. There is no discernible repeating pattern, and we may assume that the dynamics of the system are non-linear. To see this more fully we plot the phase space trajectory of the system (Figure 9) by entering the following command:

```
>> plot3(sol(:,1),sol(:,2),sol(:,3))
```

Clearly the flow of the system is chaotic and no line intersects any other line. The dynamical system evolves in a complex non-repeating pattern. The trajectory seems to be

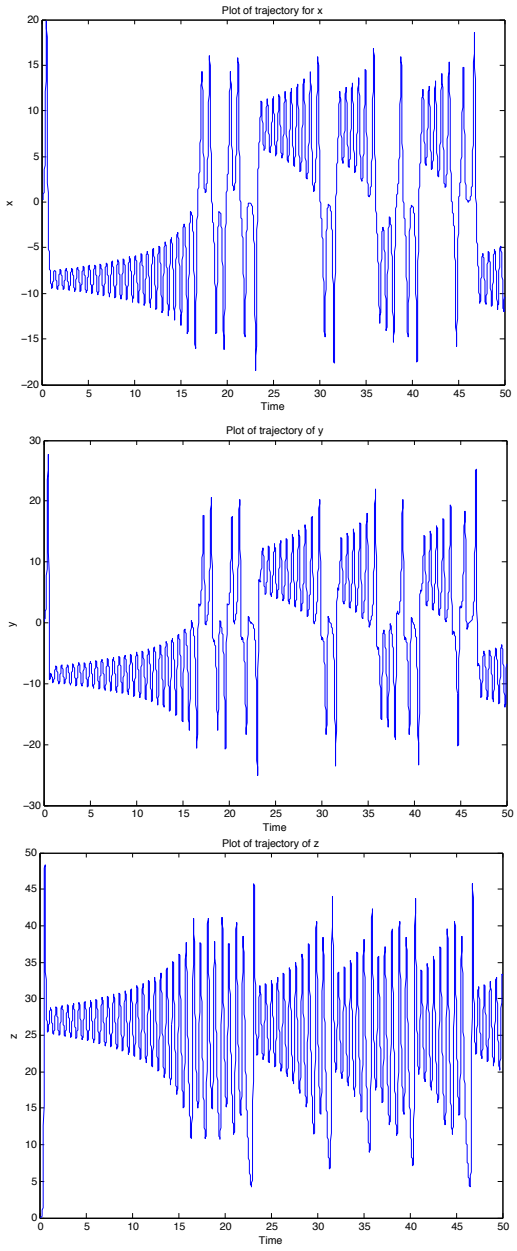

Figure 8: Plot of trajectory for x (top), y (middle), z (bottom)

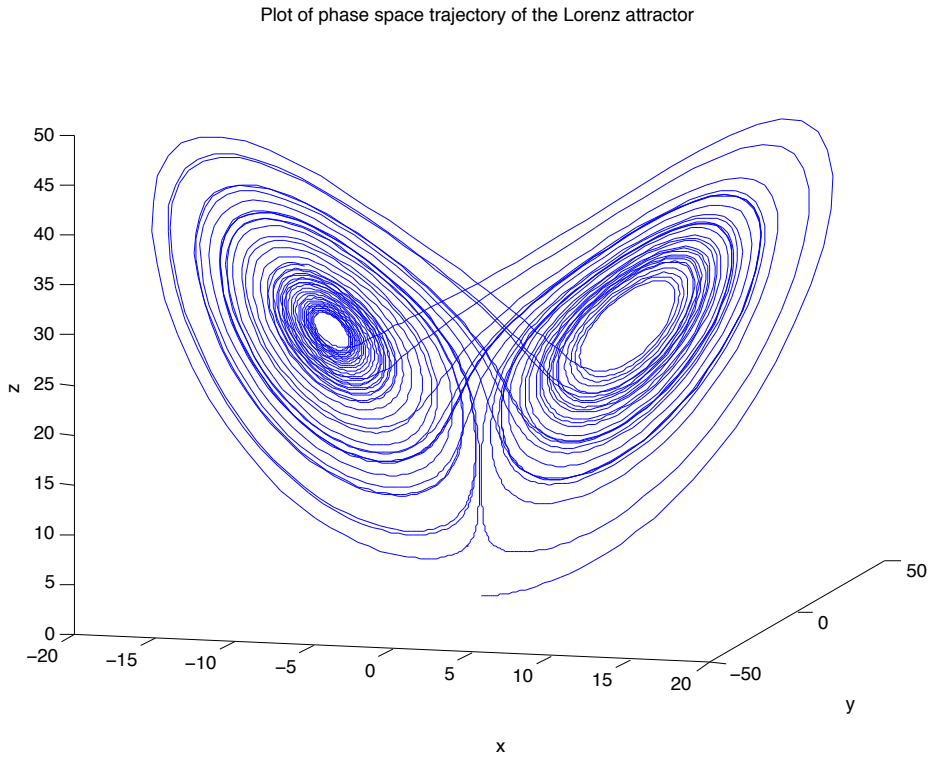
Plot of phase space trajectory of the Lorenz attractor
50
45
40
35
30
25
20
15
10
5
0
z
−20
−15
−10
−5
0
5
10
15
20
x
50
0
−50
y

Figure 9

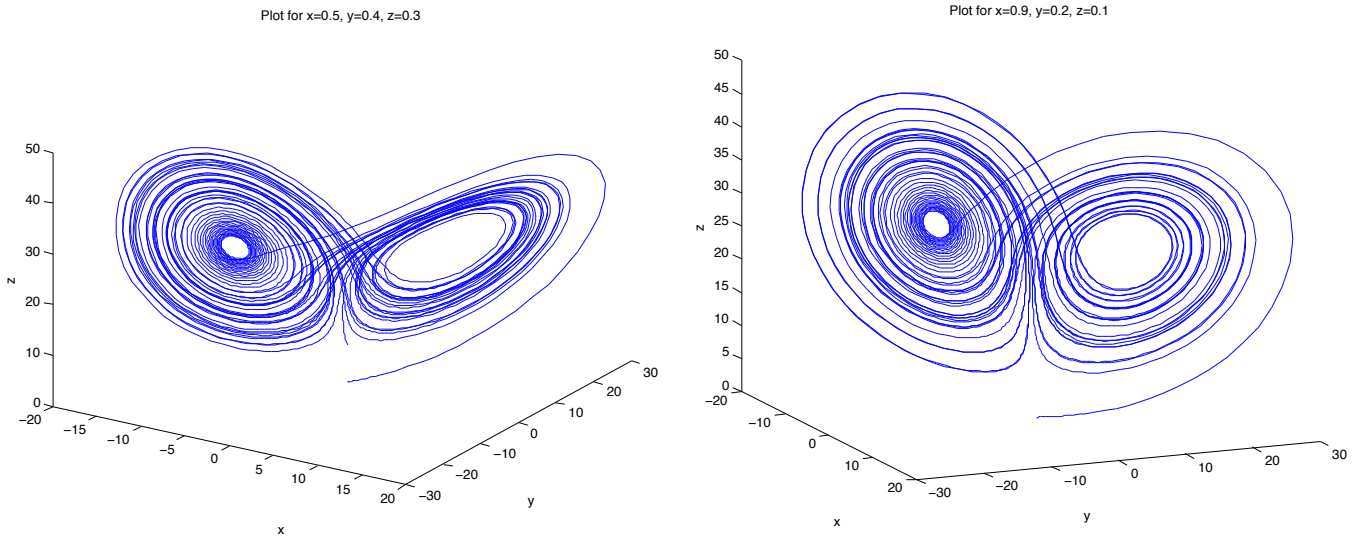

Figure 10: Plot of Lorenz attractor for two different initial conditions.

attracted to the two central circles, hence the name strange attractor. Figure 10 shows that for different initial conditions the system's trajectory will eventually follow a similar trajectory to that of Figure 9.

The plot for the trajectory of x with initial condition $x = 0.1$ and also $x = 0.2$ may be found in Figure 11. It seems that the two different initial conditions follow a similar trajectory until time runs beyond 15, where they become completely out of phase. They then exhibit similar behaviour beyond the 40 mark, gradually becoming out of phase again.

We implement Euler's method for the Lorenz equations by calling the procedure (eulerq3.m):

```
t0=0;%sets initial time
T=50;%sets size of time interval
dt=0.01;%sets size of time increment
n=(T-t0)/dt;%sets number of increments
tspan=[t0:dt:T];%sets increments as a row vector

clear x;
clear y;
clear z;

sigma=10;%sets sigma parameter
rho=28;%sets rho parameter
beta=8/3;%sets beta parameter
```

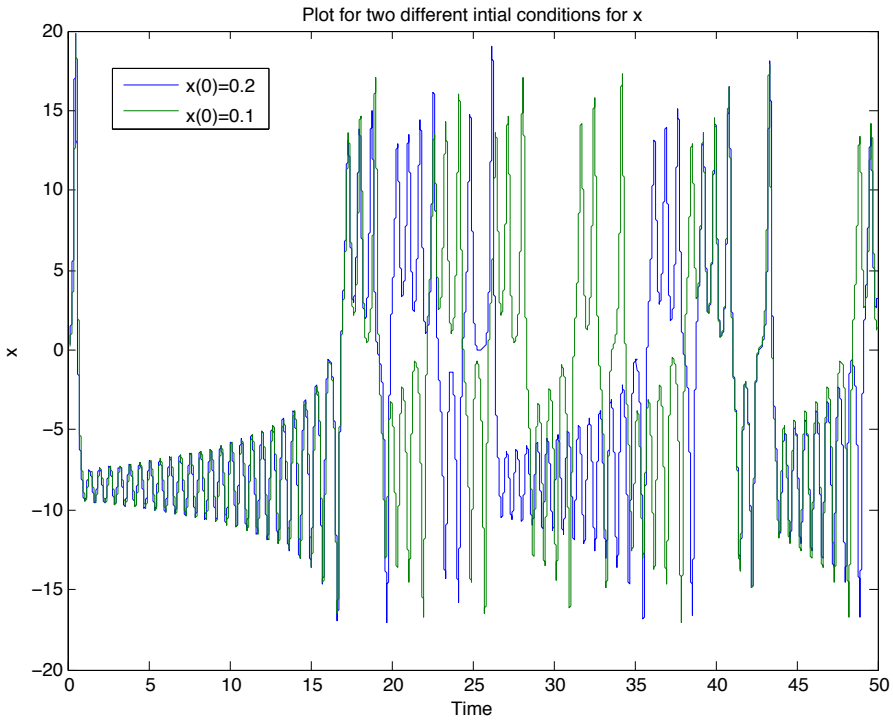
Plot for two different intial conditions for x
x(0)=0.2
x(0)=0.1
x
Time
20
15
10
5
0
−5
−10
−15
−20
0
5
10
15
20
25
30
35
40
45
50

Figure 11

```
x0=0.1;%sets initial condition for x
y0=0.1;%sets initial condition for y
z0=0.1;%sets initial condition for z

x(1)=x0;%sets initial condition for x as for entry in solution vector x
y(1)=y0;%sets initial condition for y as for entry in solution vecto y
z(1)=z0;%sets initial condition for z as for entry in solution vectr z
for i=1:n
    x(i+1)=x(i)+dt*sigma*(y(i)-x(i));%calculates Euler estimate for
                                      %x value at each time increment
    y(i+1)=y(i)+dt*(x(i)*(rho-z(i))-y(i));%calculates Euler estimate for
                                       %y value at each time increment
    z(i+1)=z(i)+dt*(x(i)*y(i)-beta*z(i));%calculates Euler estimate for
                                      %z value at each time increment
end
```

We plot the trajectory for x by executing the following commands:

```
>> lorenzproc
>> eulerq3
>> plot(ts,sol(:,1),tspan,x)
```

which results in the plot found in Figure 12. This clearly shows that the trajectory for the Euler estimate is not even close to the solution obtained using ode45. The two solutions for x agree almost nowhere and so Euler's method is pretty useless for approximating the lorenz attractor. We might be inclined to think that Euler's method is generally no good for approximating the behaviour of chaotic dynamical systems.

4 The Duffing oscillator

The Duffing oscillator is a nonlinear dynamical system defined by the second order differential equation

$$\ddot{x} + \alpha\dot{x} + x + \mu x^3 = \cos(\omega t), \tag{8}$$

where α controls the size of the damping, μ determines the nonlinearity of the restoring force and $\cos(\omega t)$ determines the amount of forcing applied to the system. Note that if $\mu = 0$ the equation describes a damped and driven simple harmonic oscillator. Equation (8) may be written as a vector system of two first order differential equations in the following way:

$$\frac{d}{dt}\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} y \\ \cos(\omega t) - \alpha y - x - \mu x^3 \end{pmatrix} \tag{9}$$

We solve this system in Matlab by first defining the following function (eqDO.m):

```
function xdot = eqDO(t,x)
alpha=0.06667;%sets parameter value for alpha
mu=0.13333;%sets parameter value for mu
omega=1;%sets parameter value for omega
xdot(1)=x(2);%calcultes dx/dt
xdot(2)=-alpha*x(2)-x(1)-mu*x(1)^3+cos(omega*t);%calculates dy/dt
xdot=xdot';%returns the solution vector
end
```

We then call the procedure eqDOproc.m:

```
t0=0;%sets initial time
T=50;%sets size of time interval
dt=0.001;%sets size of time increment
tspan=[0:dt:T];%sets time increments as a row vector

x0(1)=0;%sets initial condition for x'
x0(2)=0;%sets intitial condition for x''

[ts, sol]=ode45('eqDO',[t0:dt:T],x0);%solves system
plot(sol(:,1),sol(:,2)) %plots phase space trajectory
```

We can see in Figure 13 that the system exhibits a stable fixed point ($\mu = 4$, $\alpha = 0.5$, $\omega = 0$) when the forcing of the system is constant, a limit cycle ($\mu = 4$, $\alpha = 0.5$, $\omega = 1$) and chaotic behaviour ($\mu = 0.13333$, $\alpha = 0.06667$, $\omega = 1$). As the non-linearity of the system increases the system becomes increasingly chaotic.

5 Runge-Kutta vs. Euler

Given the equation

$$\dot{x}(t) = -x(t)$$

we call the following procedure (eulerq5.m) to compare the maximum difference between Euler's method and the exact solution on a log-log plot for various time steps (Figure 14):

```
t0=0;%sets initial condition
T=5;%sets size of time interval
x0=2.0;%sets initial condition at time=0
dt=[0.0125, 0.025, 0.05, 0.1, 0.2, 0.5, 1];%this generates a vector for the
                                           %various values of dt
for j=1:7%initialises a loop that will run over the entries of dt
n=(T-t0)/dt(j);%calculates the number of time steps we take for each dt
ts=[t0:dt(j):T];%specifices the time steps as a vector
```

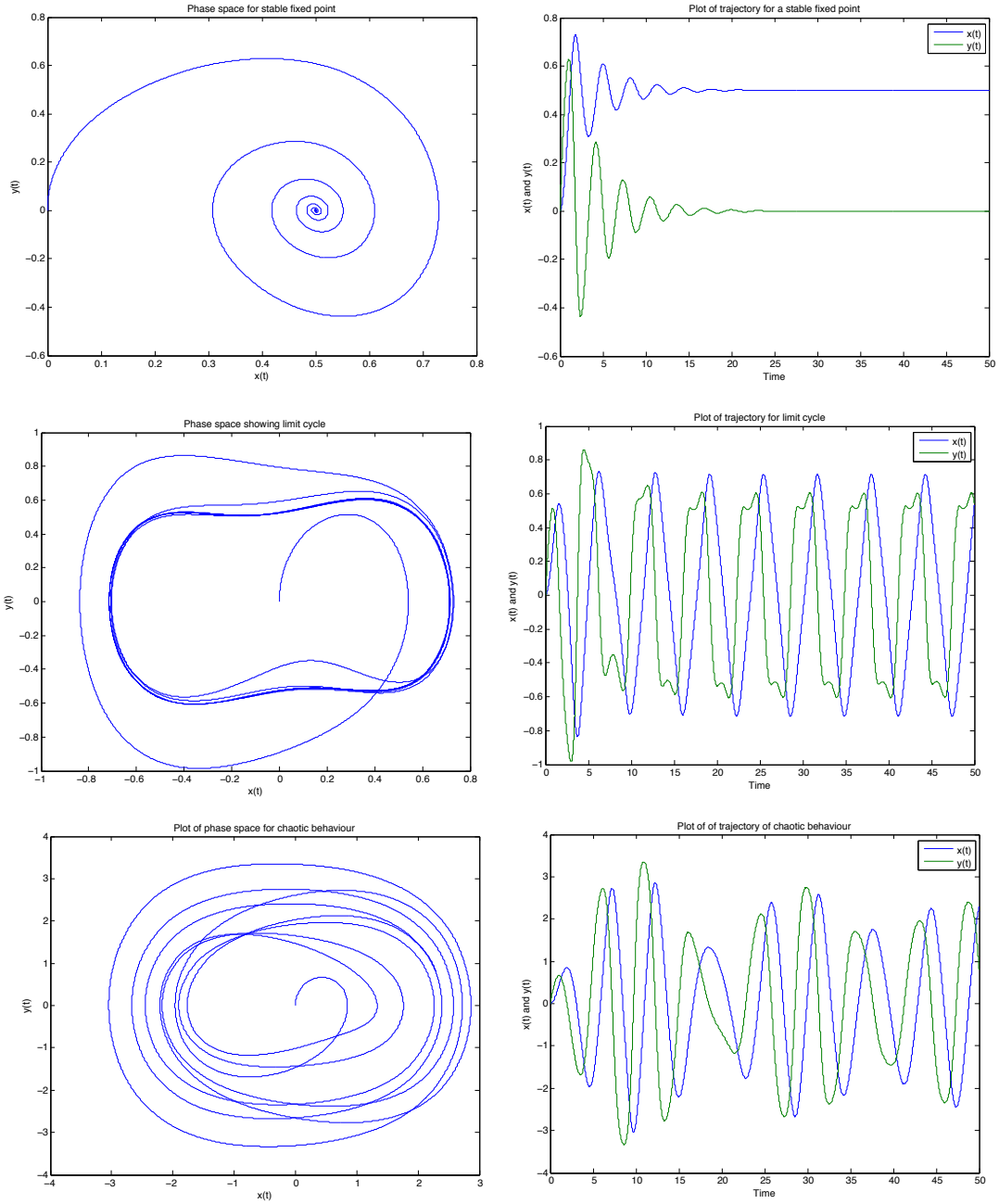

Figure 12: System exhibits a stable fixed point (top), a limit cycle (middle), chaotic behaviour (bottom).

```
clear x;%clears the vector x
x(1)=x0;%sets initial condition as first entry in solution vector
    for i=1:n%initialises loop to implement Euler's method
        x(i+1)=x(i)-x(i)*dt(j);%implements Euler's method
    end
 xexact=x0.*exp(-ts);%calculates exact solution for time steps specified
 y(j)=max(abs(x-xexact));%calculates the biggest difference between the two
                         %solutions
end
plot(log(dt),log(y))%plots the vector of differences (y) against the
                    %corresponding dt values
```

We then call the following procedure (rkq5.m) to compare the maximum difference between the Runge-Kutta method and the exact solution for various time-steps (Figure 14):

```
t0=0;%sets initial time
T=5;%sets size of time interval
x0=2.0;%sets initial condition for time=0
dtr=[0.0125, 0.025, 0.05, 0.1, 0.2, 0.5, 1];%specifies various sizes of
                                            %time-step as a vector
for j=1:7%intialises loop that runs over dtr
    clear x;%clears x
    x(1)=x0;%sets initial condition as first entry in solution vector
    n=(T-t0)/dtr(j);%specifies number of time steps taken according to dt
    ts=[t0:dtr(j):T];%sets time steps as a vector
    for i=1:n%initialises loop to implement Runge-Kutta method
        a=-x(i);%calculates left point estimate for solution
        b=-(x(i)+a*dtr(j)/2);%calculates first midpoint estimate
        c=-(x(i)+b*dtr(j)/2);%calculates second midpoint estimate
        d=-(x(i)+c*dtr(j));%calculates right point estimate
        x(i+1)=x(i)+(a+2*b+2*c+d)*dtr(j)/6;%calculates average over
                                           %time-step
    end
    xexact=x0.*exp(-ts);%calculates exact solution over the vector ts
    yr(j)=max(abs(x-xexact));%calculates biggest difference for each dt
end
plot(log(dtr),log(yr))%plots log-log plot of differences
```

The error in estimation using Euler's method is almost 1 when the time step is 1, whereas the error using Runge-Kutta's method is e^{-4}, which is much smaller for the same time step. As we can see from Figure 14 the error in estimation for both methods decreases as the size of the time step decreases, however the Runge-Kutta method does so at a much faster rate as expected and we can see that it is a much more efficient method for calculating

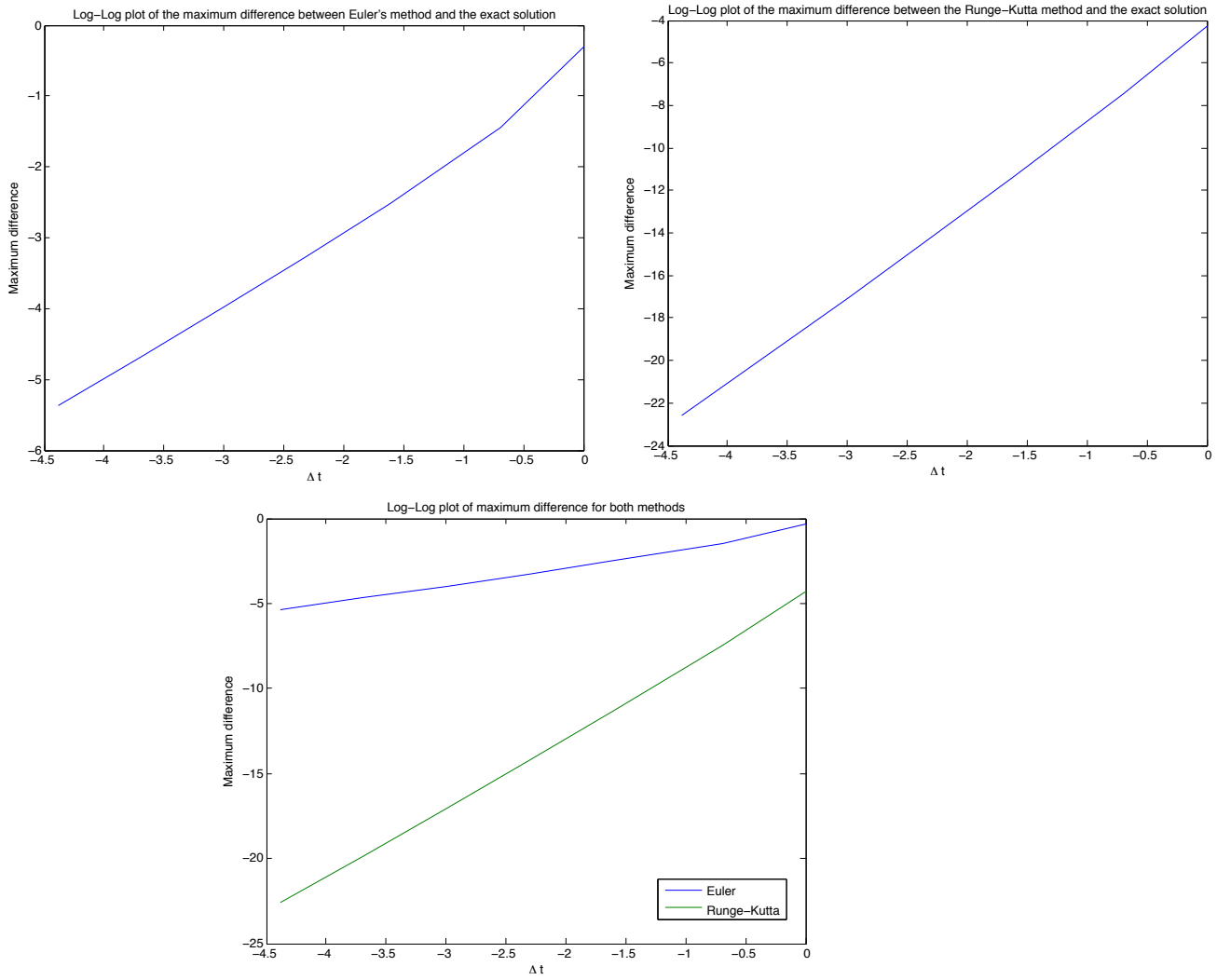

Figure 13: Clockwise from top-left: Plot for Euler method, plot for Runge-Kutta method, plot of both methods.

solutions of this type. The plots also hint at the relation between the errors- the Euler method is has a second order error, the Runge-Kutta a fourth order error, and the errors at each point as we take smaller and smaller time steps roughly seem to reflect this. We also have this interesting increase as we move from the time step of size 0.5 to 1 for the Euler method, I'm not sure how this comes about, but it implies that the Euler method gets worse at a faster rate the bigger the time step we take.