UNIVERSITY OF SUSSEX
Scientific Computing
Tutor: Dr. Ilian Iliev, Office: Pev 2 5A14

Problem Sheet 9 (Problem 2 will be assessed)

1. Consider the differential equation (called van der Pol oscillator):

$$x''(t) = -x - \epsilon(x^2 - 1)x'(t), x(0) = 0.5, x'(0) = 0,$$

where ϵ is a positive constant. As the value of ϵ increases, this equation becomes increasingly stiff.

(a) Convert this equation to two first-order equations.

(b) Run the Euler method code (which you made in the last workshop) with $\epsilon = 10$, and $[a, b] = [0, 10\pi]$ and different choices of step-size: $h = 0.02, 0.005, 0.001$. Plot all solutions, along with the exact solution (obtained by using `scipy.integrate.odeint`**Note that the in-build method has a different interface to the function defining the equation(s) compared to the supplied functions (the arguments are reversed)! Result is also formatted differently, check the help.**.) in one figure using linear axes, and the absolute errors of all solutions in another figure using semi-log axes. What do you observe?

(c) Solve the same equation using the Runge-Kutta method of 4th order provided in class and $h = 0.02$. What do you observe?

What do you conclude based on your results from (b) and (c)?

Solution:

(a) We convert this equation to two first-order equations by setting $z = dx/dt$:

$$\begin{aligned} x' &= z \\ z' &= -x - \epsilon(x^2 - 1)z \end{aligned}$$

(b) We first implement the system of equations as a (vector) function:

```
import numpy as np
import scipy
```

```
from scipy import integrate
from printSoln import *
from euler0 import *
import pylab as pl

epsi=10.

def f(t,y):
    f=np.zeros(2)
    f[0]=y[1]
    f[1]=-y[0]-epsi*(y[0]**2-1)*y[1]
    return f

def g(y,t):
    f=np.zeros(2)
    f[0]=y[1]
    f[1]=-y[0]-epsi*(y[0]**2-1)*y[1]
    return f
```

Note that (as discussed in class) we need two functions with different order of arguments, one to use with the supplied RK4 solver, one for the Python internal solver.

We then solve this system using the forward Euler method and the in-build Python ODE integrator over the requested interval and using the given initial conditions e.g. as follows:

```
x = 0.0 # Start of integration
xStop = 10.*np.pi # End of integration
y = np.array([0.5,0]) # Initial value of {y}
freq = 1 # Printout frequency

h = 0.02 # Step size
X1,Y1 = integrate0(f,x,y,xStop,h)
printSoln(X1,Y1,freq)

x1=np.linspace(0.0,xStop,np.size(X1))

z1=scipy.integrate.odeint(g,y,x1)

Yexact1=z1[:,0]
Yexact1=Yexact1.reshape(np.size(x1))
```

```
error1=abs((Y1[:,0]-Yexact1))

h = 0.005 # Step size
X2,Y2 = integrate0(f,x,y,xStop,h)
printSoln(X2,Y2,freq)

x2=np.linspace(0.0,xStop,np.size(X2))

z2=scipy.integrate.odeint(g,y,x2)

Yexact2=z2[:,0]
Yexact2=Yexact2.reshape(np.size(x2))

error2=abs((Y2[:,0]-Yexact2))


h = 0.001 # Step size

X3,Y3 = integrate0(f,x,y,xStop,h)
printSoln(X3,Y3,freq)

x3=np.linspace(0.0,xStop,np.size(X3))

z3=scipy.integrate.odeint(g,y,x3)

Yexact3=z3[:,0]
Yexact3=Yexact3.reshape(np.size(x3))

error3=abs((Y3[:,0]-Yexact3))
```

The results are the plotted using, e.g:

```
pl.plot(X3,Yexact3)

pl.show()

raw_input("Press return to exit")
```

```
pl.plot(X1,Y1[:,0])
pl.plot(X2,Y2[:,0])
pl.plot(X3,Y3[:,0])

pl.show()

raw_input("Press return to exit")

pl.clf()

pl.semilogy(X1,error1)
pl.semilogy(X2,error2)
pl.semilogy(X3,error3)

pl.show()

raw_input("Press return to exit")
```

which yields Figure 2.

The large stepsize solution (green) is incorrect, with errors larger than 100% (blue). The middle stepsize solution (red) is still significantly off, typically by at least 10% (green), and by up to several hundred percent in some intervals, and also getting worse over time. The finest-step solution (cyan) is relatively closer to the 'exact' solution (blue), albeit not ideal (off by 1% to few percent, red).

(c) Here we solve the same system using the forward Runge-Kutta method of 4th order method:

```
import numpy as np
import scipy
from scipy import integrate
from printSoln import *
from run_kut4 import *
import pylab as pl

epsi=10.

def f(t,y):
    f=np.zeros(2)
    f[0]=y[1]
```

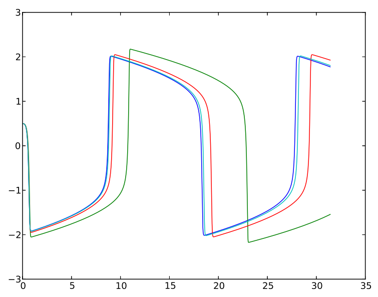

Figure 1: Solutions for Problem 1(a).

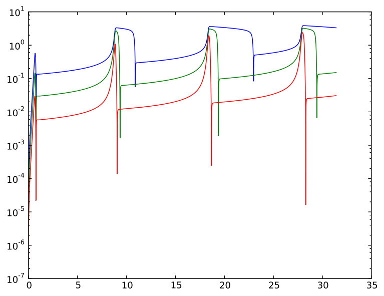

Figure 2: Errors for Problem 1(a).

```
    f[1]=-y[0]-epsi*(y[0]**2-1)*y[1]
    return f

def g(y,t):
    f=np.zeros(2)
    f[0]=y[1]
    f[1]=-y[0]-epsi*(y[0]**2-1)*y[1]
    return f

x = 0.0 # Start of integration
xStop = 10.*np.pi # End of integration
y = np.array([0.5,0]) # Initial value of {y}
freq = 1 # Printout frequency

h = 0.02 # Step size
X1,Y1 = integrate(f,x,y,xStop,h)
```

```
printSoln(X1,Y1,freq)

x1=np.linspace(0.0,xStop,np.size(X1))

z1=scipy.integrate.odeint(g,y,x1)

Yexact1=z1[:,0]
Yexact1=Yexact1.reshape(np.size(x1))

error1=abs((Y1[:,0]-Yexact1))
```

The solutions using each of the three step sizes and the 'exact' solution are plotted as follows:

```
pl.plot(X1,Yexact1)

pl.show()

raw_input("Press return to exit")

pl.plot(X1,Y1[:,0])

pl.show()

raw_input("Press return to exit")

pl.clf()

pl.semilogy(X1,error1)

pl.show()

raw_input("Press return to exit")
```

which yields Figure 4.

Clearly, the solution is much better than what we found with the Euler method (where the long step one did not even yield any result), however the errors could still be substantial - up to 10^-4 for the coarse steps, but typically less than 10^{-5} for the finer timesteps. However, the errors do grow over time.

The implicit methods used for stiff equations are generally uncon-

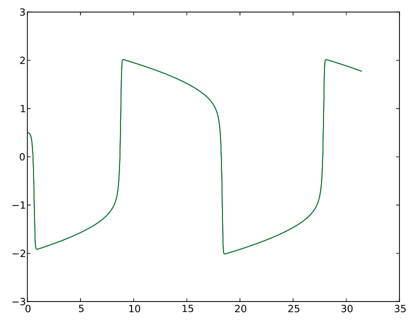

Figure 3: Results for Problem 2(b).

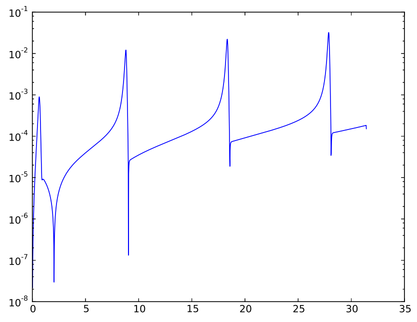

Figure 4: Errors for Problem 2(b).

ditionally stable (but not necessarily very precise, which depends on their order, instead).

2. The equations

$$\dot{u} = -a(u - v) \tag{1}$$

$$\dot{v} = cu - v - uw \tag{2}$$

$$\dot{w} = -bw + uv \tag{3}$$

known as the Lorenz equations, are encountered in the theory of fluid dynamics (atmospheric convection) and other areas of physics - lasers, electric circuits and chemical reactions. Letting $a = 5.0$, $b = 0.9$, and $c = 8.2$, use `scipy.integrate.odeint` to solve these equations from $t = 0$ to 10 with the initial conditions $u(0) = 0$, $v(0) = 1.0$, and $w(0) = 2.0$ and plot $u(t)$. Repeat the solution with $c = 8.3$ and same

a and b as above. What conclusions can you draw from the results? Again for $c = 8.3$ plot the spatial trajectory u vs. v. The result you find is called Lorentz attractor and is related to the discovery of chaotic systems. [40]

Solution: We start by importing the required modules and setting up the parameter values and the system of equations:

```
import numpy as np
import scipy
from scipy import integrate
import pylab as pl

a=5.0
b=0.9
c=8.2

def g(y,t):
    g=np.zeros(3)
    g[0]=-a*(y[0]-y[1])
    g[1]=c*y[0]-y[1]-y[0]*y[2]
    g[2]=-b*y[2]+y[0]*y[1]
    return g
```

Note that when using the Python internal integrator the function arguments need to be switched compared to what we used for Runge-Kutta.

Integrating this system with ICs $u(0) = 0, v(0) = 1$ and $w(0 = 2$ over the interval [0,100] with 10000 steps is done by:

```
x = 0.0 # Start of integration
xStop = 100.0 # End of integration
y = np.array([0.0,1.0,2.0]) # Initial value of {y}

x1=np.linspace(0.0,xStop,10000)
z1=scipy.integrate.odeint(g,y,x1)

Yexact1=z1[:,0]
Yexact1=Yexact1.reshape(np.size(x1))

pl.plot(x1,Yexact1)
```

The result of the in-build routine are also in a different matrix form, so need to be re-shaped from a column to a row vector.

We then do the same with the slightly different value for the c parameter

```
c=8.3

z2=scipy.integrate.odeint(g,y,x1)

Yexact2=z2[:,0]
Yexact2=Yexact2.reshape(np.size(x1))

#error=abs((Y1[:,0]-Yexact1))

pl.plot(x1,Yexact2)

pl.show()

raw_input("Press return to exit")
```

Results from the calculations are shown in Figure 5.

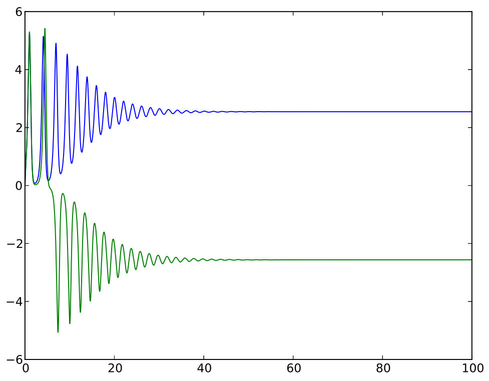

Figure 5: Solutions for Problem 2.

Clearly, the solution is very unstable and even a small change in the parameters results in complete change in the result. The system exhibits a chaotic behaviour.

Finally, we plot the spatial trajectory in x vs. y (or rather a 2D projection of the 3D spatial trajectory), as follows:

```
pl.clf() #clear the plot

Yexact3=z2[:,1]
Yexact3=Yexact3.reshape(np.size(x1))

pl.plot(Yexact2,Yexact3)

pl.show()
```

The result is shown in Figure 6.

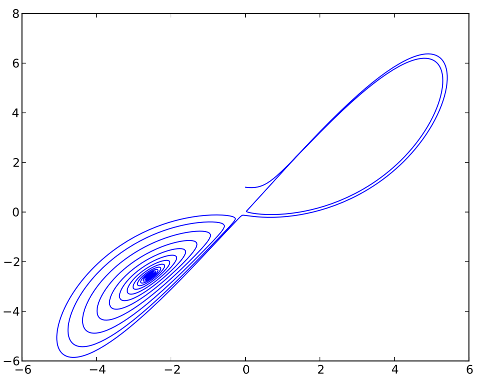

Figure 6: Lorentz attractor trajectory for Problem 2.

This is called Lorentz attractor for the trajectory. A slight change in the parameters (e.g. $c = 8.5$) could result in a different attracting point.