UNIVERSITY OF SUSSEX
Scientific Computing
Tutor: Dr. Ilian Iliev, Office: Pev 3 4A5

Problem Sheet 8 (Problem 2 will be assessed)

1. The system of differential equations describing the motion of a satellite around the Earth is described by Newton's law of gravitation:

$$x'' = \frac{-x}{(x^2+y^2)^{3/2}}, \qquad y'' = \frac{-y}{(x^2+y^2)^{3/2}} \tag{1}$$

   (a) Simplify this system to a system of 4 first-order differential equations by introducing $z = x'(t)$ and $w = y'(t)$ (i.e. speeds of the satellite in the x- and y-directions).

   (b) Solve the system you obtained in a) using the 4th order explicit Runge-Kutta method for $x(0) = 4$, $y(0) = 0$, $v(0) = 0$ and $w(0) = 0.5$. Take stepsize $h = 0.25$ and run to $t_{max} = 50$. Plot the **position** of the satellite (i.e. x vs. y) at each time-step. You should see an almost circular orbit (otherwise you have a bug) - make sure you say **axis('equal')** to ensure the plot axes have the same length.

   (c) In order to get some more interesting orbits, keep $x(0)$, $y(0)$, and $v(0)$ fixed, but:

   (i) change $w(0)$ to $w(0) = 0.6$ (with $h = 0.5$ and $t_{max} = 150$) and

   (ii) change $w(0)$ to $w(0) = 0.8$ (with $h = 0.5$ and $t_{max} = 200$).

   Plot all three orbits. What cases do they represent physically?

Solution: (a) By using the (standard) method suggested, we convert the system of 2 second-order equations into one of 4 first-order equations:

$$\begin{aligned} x' &= z \\ z' &= \frac{-x}{(x^2+y^2)^{3/2}} \\ y' &= w \\ w' &= \frac{-y}{(x^2+y^2)^{3/2}} \end{aligned}$$

(b) and (c) As usual, we first import the required modules and set up our system of equations to be solved.

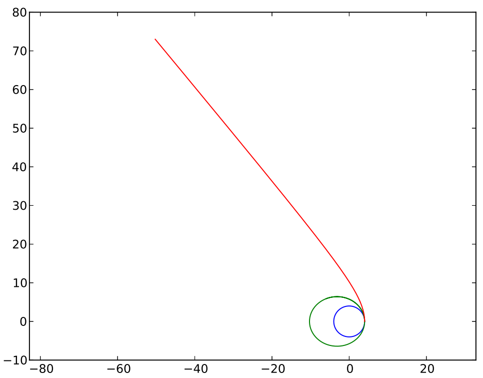
```
from numpy import *
from printSoln import *
from run_kut4 import *
import pylab as pl

def f(x,y):
    f=zeros(4)
    f[0]=y[1]
    f[1]=-y[0]/(y[0]**2+y[2]**2)**1.5
    f[2]=y[3]
    f[3]=-y[2]/(y[0]**2+y[2]**2)**1.5
    return f
```

Then, we solve this system of equations by calling the Runge-Kutta code with the requested initial conditions, stepsize and time period, and then plots the result:

```
x = 0.0 # Start of integration
xStop = 50.0 # End of integration
y = array([4., 0., 0., 0.5]) # Initial values of {y}
h = 0.25 # Step size
freq = 1 # Printout frequency

X,Y = integrate(f,x,y,xStop,h)
#printSoln(X,Y,freq)

pl.axis('equal')

pl.plot(Y[:,0],Y[:,2])
pl.show()

raw_input("Press return to exit")


xStop = 150.0 # End of integration
y = array([4., 0., 0., 0.6]) # Initial values of {y}
h = 0.5 # Step size
```

```
X,Y = integrate(f,x,y,xStop,h)
#printSoln(X,Y,freq)

pl.plot(Y[:,0],Y[:,2])
pl.show()

raw_input("Press return to exit")

xStop = 200.0 # End of integration
y = array([4., 0., 0., 0.8]) # Initial values of {y}
h = 0.5 # Step size

X,Y = integrate(f,x,y,xStop,h)
#printSoln(X,Y,freq)

pl.plot(Y[:,0],Y[:,2])
pl.show()

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

Note that we should plot 'x' vs. 'y', not vs. time (i.e. spatial trajectory, not vs. time). Results are shown in the Figure.



The result from (b) is a perfect circular orbit, while the higher initial velocities in (c) yield a more elliptical orbit (lower value) or a hyperbolic escape orbit (higher value).

2. a) Modify the provided Runge-Kutta code to use the Euler method, instead and integrate the system

$$\begin{aligned} y' &= z \\ z' &= -y \end{aligned}$$

with $y(0) = 0, z(0) = 1$ over the interval [0,20] with 200 steps.

b) Do the same using the 4th order Runge-Kutta method with the provided code.

c) Plot y from both methods and (on a separate plot) their absolute errors. What do you observe? What is the correct solution? What do you think is the reason for these results? Roughly how many points do we need for the Euler method to produce an acceptable (visually indistinguishable) solution?

Solution:

a) The modification of the Runge-Kutta code provided for doing the Euler method instead is done by simply replacing

```
def integrate(F,x,y,xStop,h):

    def run_kut4(F,x,y,h):
        K0 = h*F(x,y)
        K1 = h*F(x + h/2.0, y + K0/2.0)
        K2 = h*F(x + h/2.0, y + K1/2.0)
        K3 = h*F(x + h, y + K2)
        return (K0 + 2.0*K1 + 2.0*K2 + K3)/6.0
```

with

```
def integrate0(F,x,y,xStop,h):

    def euler(F,x,y,h):
        K0 = h*F(x,y)
        return K0
```

Note that I re-named both functions, so that they do not get confused with the corresponding Runge-Kutta functions.

We then import the needed modules and define our RHS function:

```
from numpy import array,zeros
from printSoln import *
from run_kut4 import *
from euler0 import *
import pylab as pl

def f(x,y):
    f=zeros(2)
    f[0]=y[1]
    f[1]=-y[0]
    return f

x = 0.0 # Start of integration
xStop = 20.0 # End of integration
y = array([0., 1.]) # Initial values of {y}
h = 0.1 #0.001 # Step size
freq = 1 # Printout frequency
```

Note that because this is a system, the function needs to be a vector. Integrating this system with ICs $y(0) = 0, z(0) = 1$ over the interval [0,20] with 200 steps is done by:

```
x = 0.0 # Start of integration
xStop = 20.0 # End of integration
y = array([0., 1.]) # Initial values of {y,y'}
h = 0.1  # Step size
freq = 1 # Printout frequency

X0,Y0 = integrate0(f,x,y,xStop,h)
printSoln(X0,Y0,freq)
```

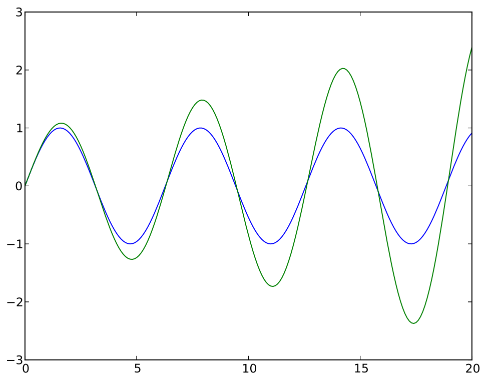
b) Do the same using the 4th order Runge-Kutta method.

```
X,Y = integrate(f,x,y,xStop,h)
printSoln(X,Y,freq)
```

c) We plot the results of (a) and (b) by:

```
pl.plot(X,Y[:,0])

pl.show()

raw_input("Press return to exit")

pl.plot(X0,Y0[:,0])
pl.show()

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

which gives:



The correct solution is, of course $y(x) = sin(x)$. Absolute errors are plotted with:

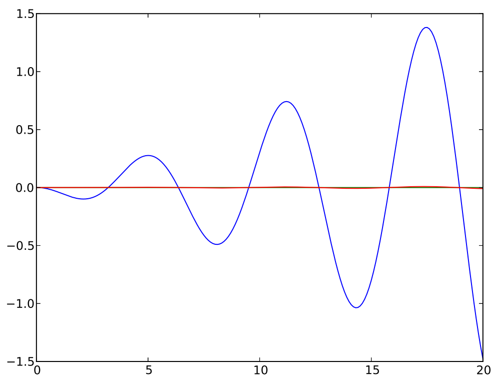
```
pl.clf()

y_exact=sin(X0)

errorEuler=y_exact-Y0[:,0]
errorRK4=y_exact-Y[:,0]

pl.plot(X0,errorEuler)
pl.plot(X0,errorRK4)
pl.show()

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

Clearly, the errors of the Euler method with step size of $h = 0.1$ are unacceptably large. I order to decrease them to acceptable level (almost indistinguishable from the RK4 method errors) we need $h \approx 0.001$:

```
h = 0.001 # Step size
```

```
freq = 1 # Printout frequency

X1,Y1 = integrate0(f,x,y,xStop,h)
printSoln(X1,Y1,freq)

y_exact1=sin(X1)

errorEuler2=y_exact1-Y1[:,0]

pl.plot(X1,errorEuler2)

pl.show()

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

However, we note that albeit the results are acceptable, it takes much longer to obtain the answer. The results for all errors are shown in the plot.



The problem observed is that the Euler method, which is low-order continuously over-shoots the correct solution.