Computer Lab Assignment 12

Radioactive Decay, Keplerian Orbits, and Runge-Kutta Method

(1) In lecture 18, we discussed a numerical algorithm to solve the following ordinary differential equation to calculate how fast the unstable isotope carbon-14 decays:

$$\frac{dc_{14}}{dt} = -k\, c_{14}(t)$$

Please use t=0 and c_{14}=1 as initial condition and a decay constant of $k=0.00012097\ year^{-1}$. Write your own 10 lines of Python code to solve this equation using the discretized form:

$$c(t+\Delta t) = c(t)\ \left[1 - k\Delta t\right]$$

Use a step size of 10 years. At the end, make a plot that compares your numerical solution to the analytical solution, $c(t) = c(0)\ e^{-kt}$. What is the half-life of carbon-14? (In this context, half-life has nothing to do with drinking unicorn blood but instead it refers to the time that it takes for half of the carbon-14 atoms to decay away.)

(2) Now we want to use the same method to solve Newton's equation for a planet orbiting the sun. For simplicity assume that the sun has mass m_S=10^6 and remains stationary. Instead of a single variable *c*(t), we now have four variables that change with time,

$$\vec{r} = (x,y) \qquad \vec{v} = (v_x, v_y)$$

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

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

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

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

For simplicity, set G=3×10^{-6}, Δt=10^{-3} and integrate for 10 time units. As initial condition, we recommend x=1, y=0, v_x=0, and v_y=2. Make a x-y plot with the following commands:

```
plt.plot(xx,yy,'r-')
plt.plot(0,0,'*',mfc='w',ms=10)
plt.gca().set_aspect('equal', adjustable='box')
plt.show()
```

If this is anything like an ellipse then you have succeeded in this part of the lab. Congratulations!

(3) Now we ask you to re-write the code by introducing a vector $\vec{y}$ = [x , y , v_x , v_y] with

```
y = np.concatenate((r,v))
```

and a Python function that computes $\frac{d\vec{y}}{dt} = f(t,\vec{y})$. Your code should still use the Euler method and should do *exactly* the same calculation as before. Review lecture 19.

```
def KeplerODE(t,y):
    global mp,ms,G

    r = y[0:2]
    v = y[2:4]

    drdt = …

    F    = …
    a    = …
    dvdt = …

    return np.concatenate((drdt,dvdt))
```

In your main code you call this function with

```
dydt = kepler_ode(t,y);
```

and you update the vector y at every time step with

```
y = y + dydt * dt
```

Remove the old formulae that updated the vectors *r* and *v*. Instead extract *r* and *v* every time from vector *y*. Now make sure that your code still works and the plots are the same. Well done if it does!

Now you have to three options to proceed. You are encouraged to implement the following five equations of the Runge-Kutta method yourself. Then complete part 4A. It only requires five lines of code. Alternatively, if you are short of time, you can choose between two pre-packaged ODE solvers provided by the SciPy package. Follow part 4B or 4C. There is very little difference between these two.

(4A) Instead of calling `KeplerODE` function once per time step Δt (called h below), we want to call it 4 times as specified in the Runge-Kutta algorithm:

$$\vec{F}_1 = \vec{f}(t_n, \vec{y}_n)$$
$$\vec{F}_2 = \vec{f}(t_n + \tfrac{h}{2}, \vec{y}_n + \tfrac{h}{2}\vec{F}_1)$$
$$\vec{F}_3 = \vec{f}(t_n + \tfrac{h}{2}, \vec{y}_n + \tfrac{h}{2}\vec{F}_2)$$
$$\vec{F}_4 = \vec{f}(t_n + h, \vec{y}_n + h\vec{F}_3)$$

Introduce intermediate vectors $F_1 \ldots F_4$ and compute the much more accurate new *y* vector

$$\vec{y}_{n+1} = \vec{y}_n + \frac{h}{6}\left[\vec{F}_1 + 2\vec{F}_2 + 2\vec{F}_3 + \vec{F}_4\right]$$

Now run the new code and see if you still get an ellipse.

(4B) Here you use the function `solve_ivp` that allows you to choose between different integration methods. A popular one is RK45, which compare the deviations between a 4th and a 5th order Runge-Kutta integration scheme to adjust the time step automatically. Smaller steps will be used when the planet is near the sun when it moves faster.

```
from scipy.integrate import solve_ivp

y = np.concatenate((r0,v0)) # set initial conditions
sol = solve_ivp(KeplerODE,[0,tMax],y,method='RK45',max_step=0.01)

print(sol)
plt.plot(sol.y[0,:],sol.y[1,:],'r-')
plt.plot(0,0,'*',mfc='w',ms=10)
plt.gca().set_aspect('equal', adjustable='box')
plt.show()
```

(4C) SciPy also provides the second routine `odeint` to solve ODEs. They adopted the different convention for the order of the t and y arguments in the `KeplerODE` function. y comes before t. Instead of changing my original `KeplerODE` function, I wrote a wrapper routine `KeplerODE2`

```
def KeplerODE2(y,t):
     return KeplerODE(t,y)

from scipy.integrate import odeint

y0 = np.concatenate((r0,v0))
t = np.arange(0.0, tMax, dt)

yt = odeint(KeplerODE2, y0, t)

print(yt)
plt.plot(yt[:,0],yt[:,1])
plt.plot(0,0,'*',mfc='w',ms=10)
plt.gca().set_aspect('equal', adjustable='box')
plt.show()
```

(5) Regardless whether you opted for parts (4A), (4B), or (4C), you are now in the position to compare the accuracy of our original Euler methods with higher-order integration techniques like Runge-Kutta method. A common approach to gauge accuracy of a method is to compare the total energy at the beginning and at the end. Please plot the kinetic, potential, and total energy as a function of time.

$$E_{kin} = \frac{1}{2} m \vec{v}^2 \qquad E_{pot} = -\frac{G m_S m_P}{r} \qquad E_{tot} = E_{kin} + E_{pot}$$

The Runge-Kutta method requires the `KeplerODE` function to be called four times per time step. This is clearly more work than is needed in a single Euler step. Can you somehow verify why everyone prefers the Runge-Kutta methods nevertheless?