AST 304, Fall 2022

Stars

Group Computational Project 1

In this project you shall implement three algorithms for solving a system of first-order ordinary differential equations (ODEs). These algorithms—forward Euler, second-order Runge-Kutta, and fourth-order Runge-Kutta—are described in appendix A.3 of the course pack.

A good test problem for our ODE solver is Kepler's problem—two point masses orbiting about their common center of mass. The problem is useful because it has an analytical solution against which you can compare your numerical solution.

1 Kepler's problem

Recall that by working in the center-of-mass frame, we can reduce the problem to that of a single particle of reduced mass $\mu \equiv m_1 m_2/(m_1 + m_2)$ obeying the equations of motion

$$\frac{\mathrm{d}\boldsymbol{r}}{\mathrm{d}t} = \boldsymbol{v}, \qquad \frac{\mathrm{d}\boldsymbol{v}}{\mathrm{d}t} = -\frac{GM}{r^3}\boldsymbol{r}, \tag{1}$$

where $M = m_1 + m_2$ is the total mass and $\boldsymbol{r} = \boldsymbol{r}_2 - \boldsymbol{r}_1$ is the vector pointing from particle 1 to particle 2. The motion is in a plane, so $\boldsymbol{r}$ and $\boldsymbol{v}$ are two dimensional vectors: $\boldsymbol{r} = (x, y)$ and $\boldsymbol{v} = (v_x, v_y)$.

The total energy of the system is the sum of kinetic (K) and potential (Ω) energies:

$$E = K + \Omega = \frac{1}{2}\mu v^2 - \frac{GM\mu}{r}, \tag{2}$$

with $v^2 = \boldsymbol{v} \cdot \boldsymbol{v}$ and $r = \sqrt{\boldsymbol{r} \cdot \boldsymbol{r}}$. If $E < 0$, then the motion is an elliptical orbit with semi-major axis

$$a = \frac{GM\mu}{2|E|}, \tag{3}$$

and orbital period

$$T = \frac{\pi}{\sqrt{2}} GM\mu^{3/2}|E|^{-3/2}. \tag{4}$$

Using equation (3) to express E in terms of a, substituting this expression into equation (4), and then squaring, we recover Kepler's third law relating the period and semi-major axis:

$$T^2 = \frac{4\pi^2}{GM} a^3. \tag{5}$$

We can test our numerical solution by checking that the orbit has the correct period, semi-major axis, and energy.

2 The reduced equations

Before numerically solving our system of equations (1), it is a good idea to scale our variables. First, note that equation (1) doesn't depend on the reduced mass μ. We therefore remove it from our computation by tracking the energy per reduced mass,

$$\epsilon \equiv \frac{E}{\mu} = \frac{v^2}{2} - \frac{GM}{r}, \tag{6}$$

rather than the energy.

Next, notice that for planetary orbits it is convenient to measure GM in units of $GM_\odot$ and distances in units of AU. The dimension of $GM_\odot$ is

$$\text{energy} \times \text{length}/\text{mass} \sim \text{length}^3/\text{time}^2.$$

It follows that our unit of energy per mass is $GM_\odot/(1\,\text{au})$. Since the energy per unit mass ϵ has dimensions $(\text{length}/\text{time})^2$, the unit of velocity in these units is $\sqrt{GM_\odot/(1\,\text{au})}$. From the unit of velocity and length, we construct the unit of time, length/velocity, as $(1\,\text{au})^{3/2}/(GM_\odot)^{1/2}$. With these units and defining $m = M/M_\odot$, the equations of motion reduce to

$$\frac{\mathrm{d}\boldsymbol{r}}{\mathrm{d}t} = \boldsymbol{v}, \qquad \frac{\mathrm{d}\boldsymbol{v}}{\mathrm{d}t} = -\frac{m}{r^3}\boldsymbol{r}. \tag{7}$$

In these units the energy per reduced mass, semi-major axis, and orbital period are

$$\epsilon = \frac{v^2}{2} - \frac{m}{r} \tag{8}$$

$$a = \frac{m}{2|\epsilon|} \tag{9}$$

$$T = \frac{\pi}{\sqrt{2}} m |\epsilon|^{-3/2}. \tag{10}$$

To specify our orbit, we choose the mass (in units of $M_\odot$) m and the semi-major axis (in AU) a. This sets ϵ via eq. (9). For example, suppose we set the total mass

and semi-major axis to $m = 1$ and $a = 1$, as would be appropriate for Earth's orbit. In these units the energy per mass is

$$\epsilon = -\frac{m}{2a} = -\frac{1}{2}. \tag{11}$$

The period T is then found from eq. (10). For $m = 1$, $a = 1$, the period in our reduced units is

$$T = 2\pi.$$

To find the period in seconds, multiply T by our unit of time, $\sqrt{(1\,\mathrm{au})^3/GM_\odot} = 5.02 \times 10^6$ s. For $m = 1$, $a = 1$, the period is therefore $2\pi \times 5.02 \times 10^6$ s $= 1.00$ yr, as expected.

Once we have the mass, semi-major axis, energy, and period, we then need to set our initial conditions $x_0, y_0, v_{x,0}, v_{y,0}$ that satisfy this energy via eq. (8). We have 4 initial variables and only one constraint, so we can freely pick any 3 of the initial variables.

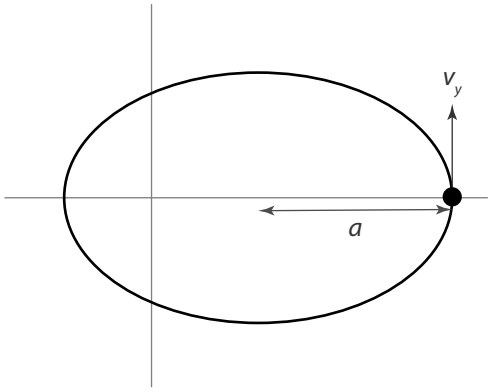

Figure 1: Layout of the ellipse.

For convenience, let's orient the orbit so the semi-major axis is along x, and let's start our particle at apastron as shown in Fig. 1. In this setup $y_0 = 0$, $v_{x,0} = 0$ and $a < x_0 < 2a$. Once we've set x_0, y_0, and $v_{x,0}$, we determine $v_{y,0}$ from the reduced energy per mass, eq. (11):

$$v_{y,0} = \sqrt{2\epsilon + 2m/x_0}. \tag{12}$$

3 Assignment

1. Implement routines that advance the solution to a system of ODE's by one step h for the forward Euler, second-order Runge-Kutta, and fourth-

order Runge-Kutta. The partially completed routines are in the file `ode.py` in your project repository. Once you have the routines written, you will need to test them. The file `test_ode.py`. This test integrates the simpler system of equations

$$z(t=0) = \{0.0, 1.0\}, \qquad f(t, z, \omega) = \{\omega z_1, -\omega z_0\}, \tag{13}$$

for which you can verify that the solution is $z(t) = \{\sin(\omega t), \cos(\omega t)\}$. Run `test_ode.py` and compare the output of this test with the file `sample_output`. They should be close.

2. Next, complete the functions in `kepler.py` that compute the kinetic, potential, and total energies, all per unit reduced mass, as well as the function

```
65 def derivs(t,z,m):
```

which computes $\mathrm{d}\boldsymbol{r}/\mathrm{d}t$, $\mathrm{d}\boldsymbol{v}/\mathrm{d}t$ following eq. [7].

3. Also in the file `kepler.py` you will find a routine

```
84 def integrate_orbit(z0,m,tend,h,method='RK4'):
```

that integrates the equations of motion from $0 < t \leq t_{\rm end}$. Notice that this routine takes an optional parameter `method` with default value `'RK4'`: if you call the routine with `method='Euler'` it will use the `ode.fEuler` method, and similarly for `'RK2'` and `'RK4'`. This allows you to switch between integration methods without having to rewrite code. Finally, there is a routine

```
143 def set_initial_conditions(a, m, e):
```

that computes the initial position and velocity, as well as the energy and orbital period, given the semi-major axis, mass, and eccentricity.

You will need to complete the routines in `kepler.py`, including the documentation.

4. Write a python script that uses the functions in `ode.py` and `kepler.py` to do the following.

For each of the three integration methods, integrate the equations of motion over 3 orbital periods, and compute the relative error in the total energy at the end of this time. Take the semi-major axis $a = 1$ and total mass $m = 1$. Make the ellipse have an eccentricity of $e = 0.5$ so that

$x_0 = (1+e)a = 1.5a$. Do each integration for a range of stepsizes $h = h_0, h_0/2, h_0/4, \ldots h_0/1024$, where $h_0 = 0.1\,T$ with T being the expected orbital period. Plot the error in the energy as a function of h. Does it scale as expected? Is it better to use linear or logarithmic axes in plotting the error?

5. For the smallest and largest values of h for each of the three integration methods, plot the particle trajectory. Does the orbit close? Is it an ellipse? Does it have the correct semi-major axis? Also plot the energies—potential, kinetic, and total—as a function of time. Put the energies all on the same plot.

4 Useful Tips

1. In this problem you will need to compute quantities such as $v^2 = \boldsymbol{v}\cdot\boldsymbol{v}$ and $r = \sqrt{\boldsymbol{r}\cdot\boldsymbol{r}}$. The `numpy` module has a `dot` function for computing the dot-product of two arrays and a `norm` function for finding the norm of a vector. For example,

```
In [1]: import numpy as np

In [2]: x = np.array([3.0,4.0])

In [3]: from numpy.linalg import norm

In [4]: r = norm(x)

In [5]: print(r)
5.0

In [6]: r2 = np.dot(x,x)

In [7]: print(r2)
25.0
```

2. In many problems the function $f(t,z)$ may depend on other parameters. For example, consider our test problem

$$\frac{\mathrm{d}}{\mathrm{d}t}\begin{Bmatrix} z_0 \\ z_1 \end{Bmatrix} = \omega \begin{Bmatrix} z_1 \\ -z_0 \end{Bmatrix}, \tag{14}$$

with $z(t = 0) = \{0.0, 1.0\}$. The solution is $z_0 = \sin(\omega t)$, $z_1 = \cos(\omega t)$. We would like to pass the parameter ω to our function $z' = f(t, z, \omega)$, but we don't want to write a special forward Euler routine for each problem. `Python` has a handy trick for handling this. The interface for our step routine is

```
15 def fEuler(f,t,z,h,args=()):
```

This says that our step routine takes an additional argument `args`, but the *default* value of `args`—the value that `args` takes if we omit it when calling `fEuler`—is a empty tuple[1] `()`. Then, later in the routine when we call the function `f` we can pass `args` like so:

```
45     return z + h*f(t,z,*args)
```

The `*args` directive turns the tuple `args` into an additional list of parameters for the function `f`. If `args` is empty—the default—then nothing is passed. If, for example, we called `fEuler` with `args=(w,x,y)`, then `f(t,z,*args)` would be converted to `f(t,z,w,x,y)`.

If `f` only takes one additional parameter, then writing `args=(w,)` is cumbersome, so we add the code

```
35     if not isinstance(args,tuple):
36         args = (args,)
```

This converts `args` into a tuple if it isn't one already.

[1] In `Python`, a *tuple* is a sequence of objects; unlike a list the tuple cannot be changed after it is defined.