ASTR 119: Session 14

Orbital Mechanics

Outline

1) Code Check In
2) HW #6 Cash-Karp Solution
3) Final projects + visualizations
4) Orbital Mechanics
5) Save your work to GitHub

Computing/Programming Check-In

1) Create a jupyter notebook, and name it astr-119-code-check-in.ipynb.

2) Add a markdown line that describes each cell in the notebook. Each of the following instructions should be its own cell, in order.

3) Import numpy and matplotlib.pyplot as usual.

4) Declare integer i, set equal to zero. Declare a float x, set equal to 119.

5) Use a for loop, iterate i from 0 to 119 inclusive. For each even value (including 0) of i, add 3 to x. For each odd value of i, subtract 5 from x.

6) Print the final value of x in scientific notation using 2 decimal places.

You will have ~ 5 minutes.

TAs will be sending sign-up information, starting in section this week.

HW#6 — Solution

1) Repeat the exercise from the 11/6 session, but use the Cash-Karp Runge-Kutta method with adaptive stepwise control

2) Evolve the system of equations:

$$\frac{dy}{dx} = z \qquad \frac{dz}{dx} = -y$$

Use the initial conditions y(x=0) = 0 and dydx(x=0) = 1, and evolve over the range [0, 2*pi].

3) Plot the analytical solutions for y(x) and dy/dx(x) over the specified range, and the numerical solution.

4) Plot the absolute error for the numerical solutions of y(x) and dy/dx(x) over the specified range.

5) Call the repository “astr-119-hw-5” and the notebook “hw-5.ipynb”.

ADAPTIVE STEP-SIZE CONTROL

Consider the general form of the fifth-order Runge-Kutta method:

$$k_1 = hf(x_n, y_n) \qquad k_2 = hf(x_n + c_2 h, y_n + a_{21} k_1)$$

$$k_3 = hf(x_n + c_3 h, y_n + a_{31} k_1 + a_{32} k_2)$$

$$k_4 = hf(x_n + c_4 h, y_n + a_{41} k_1 + a_{42} k_2 + a_{43} k_3)$$

$$k_5 = hf(x_n + c_5 h, y_n + a_{51} k_1 + a_{52} k_2 + a_{53} k_3 + a_{54} k_4)$$

$$k_6 = hf(x_n + c_6 h, y_n + a_{61} k_1 + a_{62} k_2 + a_{63} k_3 + a_{64} k_4 + a_{65} k_5)$$

$$y_{n+1} = y_n + b_1 k_1 + b_2 k_2 + b_3 k_3 + b_4 k_4 + b_5 k_5 + b_6 k_6 + O(h^6)$$

$$y^\star_{n+1} = y_n + b^\star_1 k_1 + b^\star_2 k_2 + b^\star_3 k_3 + b^\star_4 k_4 + b^\star_5 k_5 + b^\star_6 k_6 + O(h^5)$$

$$\Delta \equiv y_{n+1} - y^\star_{n+1} = \sum_{i=1}^{6} (b_i - b^\star_i) k_i$$

Cash-Karp Solution

Create a notebook to perform the Cash-Karp implementation of a Runge-Kutta integration.

In [8]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
```

Cash-Karp Solution

Define a coupled set of ODEs to integrate

In [122]:

```
def dfdx(x,f):

    #d2y/dx2 = -y
    #define :

    # y = f[0]
    # dy/dx = z
    # z = f[1]

    y = f[0]
    z = f[1]

    #return derivatives
    dydx = np.zeros_like(f)
    dydx[0] = z
    dydx[1] = -1*y

    return dydx
```

Cash-Karp Solution

Define the core of the Cash-Karp method

In [127]:

```
def cash_karp_core_mv(x_i,y_i,nv,h,f):

    #cash karp is defined in terms
    #of weighting variables
    ni  = 7
    nj  = 6
    ci  = np.zeros(ni)
    aij = np.zeros( (ni,nj) )
    bi  = np.zeros(ni)
    bis = np.zeros(ni)

    #input values for ci, aij, bi, bis
    ci[2] = 1./5.
    ci[3] = 3./10.
    ci[4] = 3./5.
    ci[5] = 1.
    ci[6] = 7./8.
```

Cash-Karp Solution

Still cash_karp_core_mv()

```
18
19      #j = 1
20      aij[2,1] = 1./5
21      aij[3,1] = 3./40.
22      aij[4,1] = 3./10.
23      aij[5,1] = -11./54.
24      aij[6,1] = 1631./55296.
25
26      #j = 2
27      aij[3,2] = 9./40.
28      aij[4,2] = -9./10.
29      aij[5,2] = 5./2.
30      aij[6,2] = 175./512.
31
32      #j = 3
33      aij[4,3] = 6./5.
34      aij[5,3] = -70./27.
35      aij[6,3] = 575./13824.
36
37      #j = 4
38      aij[5,4] = 35./27.
39      aij[6,4] = 44275./110592.
40
41      #j = 5
42      aij[6,5] = 253./4096.
43
```

Cash-Karp Solution

Still cash_karp_core_mv()

```
    #bi
    bi[1] = 37./378.
    bi[2] = 0.
    bi[3] = 250./621.
    bi[4] = 125./594.
    bi[5] = 0.0
    bi[6] = 512./1771.

    #bis
    bis[1] = 2825./27648.
    bis[2] = 0.0
    bis[3] = 18575./48384.
    bis[4] = 13525./55296.
    bis[5] = 277./14336.
    bis[6] = 1./4.

    #define the k array
    ki = np.zeros((ni,nv))

    #compute ki
    for i in range(1,ni):
        #compute xn+1 for i
        xn = x_i
        for j in range(1,i+1):
            xn += ci[j]*h

        #compute temp y
        yn = y_i.copy()
        for j in range(1,i):
            yn += aij[i,j]*ki[j,:]

        #get k
        ki[i,:] = h*f(xn,yn)
```

Cash-Karp Solution

Still cash_karp_core_mv()

```
78      #get ynpo, ynpo*
79      ynpo  = y_i.copy()
80      ynpos = y_i.copy()
81      #print("ni = ",ni, ynpo, ynpos)
82      for i in range(1,ni):
83          ynpo  += bi[i] *ki[i,:]
84          ynpos += bis[i]*ki[i,:]
85          #print(i,ynpo[0],ynpos[0])
86          #print(i,ynpo[0],ynpos[0],bi[i]*ki[i,0],bis[i],*ki[i,0])
87
88      #get error
89      Delta = np.fabs(ynpo-ynpos)
90
91      #print("INSIDE Delta",Delta,ki[:,0],ynpo,ynpos)
92
93
94      #return new y and delta
95      return ynpo, Delta
```

Cash-Karp Solution

Define an adaptive step size driver for Cash-Karp

8]:

```
def cash_karp_mv_ad(dfdx,x_i,y_i,nv,h,tol):



    #define a safety scale
    SAFETY = 0.9
    H_NEW_FAC = 2.0

    #set a maximum number of iterations
    imax = 1000

    #set an iteration variable
    i = 0

    #create an error
    Delta = np.full(nv,2*tol)

    #remember the step
    h_step = h

```

Cash-Karp Solution

```
16
17      #remember the step
18      h_step = h
19
20      #adjust the step
21      while(Delta.max()/tol>1.0):
22
23          #get our new y and error estimate
24          y_ipo, Delta = cash_karp_core_mv(x_i,y_i,nv,h_step,dfdx)
25
26          #if the error is too large, take a smaller step
27          if(Delta.max()/tol>1.0):
28
29              #our error is too large, decrease step
30              h_step *= SAFETY * (Delta.max()/tol)**(-0.25)
31
32          #check iteration
33          if(i>=imax):
34              print("Too many iterations in cash_karp_mv_ad()")
35              raise StopIteration("Ending after i = ",i)
36
37          #iterate
38          i += 1
```

Still cash_karp_mv_ad()

Cash-Karp Solution

Still cash_karp_mv_ad()

```
36
37          #iterate
38          i += 1
39
40      #next time, try a bigger step
41      h_new = np.fmin(h_step * (Delta.max()/tol)**(-0.9), h_step * H_NEW_FAC)
42
43      #return the answer and step info
44      return y_ipo, h_new, h_step
45
46
47
48
```

Cash-Karp Solution

Define a wrapper for Cash-Karp

9]:

```
def cash_karp_mv(dfdx,a,b,y_a,tol,verbose=False):

    #dfdx is the derivative wrt x
    #a is the lower bound
    #b is the upper bound
    #y_a are the boundary condition at a
    #tol is the tolerance

    #define our starting step
    xi = a
    yi = y_a.copy()

    #define an initial starting step
    h = 1.0e-4 * (b-a)

    #set a max number of iterations
    imax = 1000

    #set an iteration variable
    i = 0

    #how many variables?
    nv = len(y_a)

    #set the initial conditions
    x = np.full(1,a)
    y = np.full( (1,nv), y_a)

```

Cash-Karp Solution

Still cash_karp_mv()

```
31
32      #loop until we reach b
33      while(flag):
34
35          #calculate y_i+1, step info
36          y_ipo, h_new, h_step = cash_karp_mv_ad(dfdx,xi,yi,nv,h,tol)
37
38          #update the step for next time
39          h = h_new
40
41          #prevent an overshoot
42          if(xi+h_step>b):
43
44              #limit step to end at b
45              h = b - xi
46
47              #recompute y_i+1
48              y_ipo, h_new, h_step = cash_karp_mv_ad(dfdx,xi,yi,nv,h,tol)
49
50              #we're done
51              flag = False
52
53          #update the values
54          xi += h_step
55          yi = y_ipo.copy()
56
57          #add the step
58          x = np.append(x,xi)
59          y_ipo = np.zeros((len(x),nv))
60          y_ipo[0:len(x)-1,:] = y[:]
61          y_ipo[-1,:] = yi[:]
62          del y
63          y = y_ipo
```

Cash-Karp Solution

Still cash_karp_mv()

```
65
66
67          #prevent too many iterations
68          if(i>=imax):
69              print("Maximum iterations reached.")
70              raise StopIteration("Iteration number = ",i)
71
72          #iterate
73          i += 1
74
75          #output some information
76          if(verbose):
77              s = "i = %3d\tx = %9.8f\ty = %9.8f\th = %9.8f\tb = %9.8f" % (i,xi,yi[0],h_step,b)
78              print(s)
79
80          #if we're done, exit
81          if(xi==b):
82              flag = False
83
84      #return the answer
85      return x, y
86
```

Cash-Karp Solution

Perform the integration

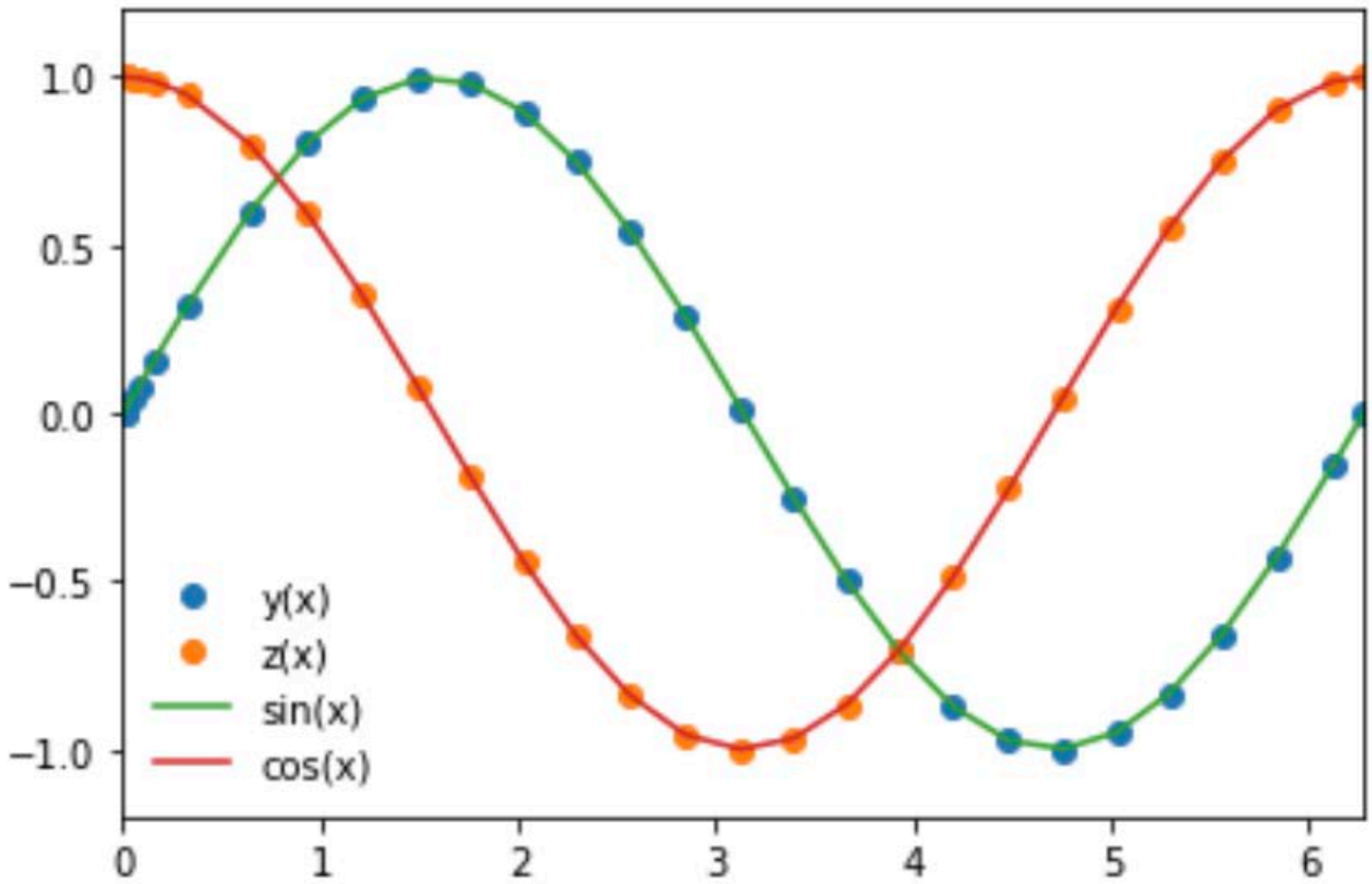
30]:
```
a = 0.0
b = 2.0*np.pi
y_0 = np.zeros(2)
y_0[0] = 0.0
y_0[1] = 1.0
nv = 2

tolerance = 1.0e-6
x, y = cash_karp_mv(dfdx,a,b,y_0,tolerance,verbose=True)
plt.plot(x,y[:,0],'o',label='y(x)')
plt.plot(x,y[:,1],'o',label='z(x)')
plt.plot(x,np.sin(x),label='sin(x)')
plt.plot(x,np.cos(x),label='cos(x)')
plt.xlim([0,2*np.pi])
plt.ylim([-1.2,1.2])
plt.legend(frameon=False)
```

Cash-Karp Solution

```
i =   1 x = 0.00062832  y = 0.00062832  h = 0.00062832  b = 6.28318531
i =   2 x = 0.00188496  y = 0.00188495  h = 0.00125664  b = 6.28318531
i =   3 x = 0.00439823  y = 0.00439822  h = 0.00251327  b = 6.28318531
i =   4 x = 0.00942478  y = 0.00942464  h = 0.00502655  b = 6.28318531
i =   5 x = 0.01947787  y = 0.01947664  h = 0.01005310  b = 6.28318531
i =   6 x = 0.03958407  y = 0.03957373  h = 0.02010619  b = 6.28318531
i =   7 x = 0.07979645  y = 0.07971180  h = 0.04021239  b = 6.28318531
i =   8 x = 0.16022123  y = 0.15953660  h = 0.08042477  b = 6.28318531
i =   9 x = 0.32107077  y = 0.31558279  h = 0.16084954  b = 6.28318531
i =  10 x = 0.64276986  y = 0.59941495  h = 0.32169909  b = 6.28318531
i =  11 x = 0.93739384  y = 0.80601851  h = 0.29462398  b = 6.28318531
i =  12 x = 1.20675386  y = 0.93446544  h = 0.26936002  b = 6.28318531
i =  13 x = 1.49426997  y = 0.99707369  h = 0.28751611  b = 6.28318531
i =  14 x = 1.76344767  y = 0.98150050  h = 0.26917769  b = 6.28318531
i =  15 x = 2.03076151  y = 0.89606839  h = 0.26731385  b = 6.28318531
i =  16 x = 2.29758389  y = 0.74731321  h = 0.26682237  b = 6.28318531
i =  17 x = 2.56844699  y = 0.54227799  h = 0.27086310  b = 6.28318531
i =  18 x = 2.84768738  y = 0.28969237  h = 0.27924039  b = 6.28318531
i =  19 x = 3.12869484  y = 0.01289739  h = 0.28100745  b = 6.28318531
i =  20 x = 3.39732623  y = -0.25295549 h = 0.26863139  b = 6.28318531
i =  21 x = 3.66411066  y = -0.49906425 h = 0.26678443  b = 6.28318531
i =  22 x = 3.93155133  y = -0.71032491 h = 0.26744067  b = 6.28318531
i =  23 x = 4.20391433  y = -0.87348904 h = 0.27236300  b = 6.28318531
i =  24 x = 4.48601277  y = -0.97448720 h = 0.28209844  b = 6.28318531
i =  25 x = 4.76275451  y = -0.99873304 h = 0.27674175  b = 6.28318531
i =  26 x = 5.03073667  y = -0.94975497 h = 0.26798215  b = 6.28318531
i =  27 x = 5.29729916  y = -0.83376264 h = 0.26656249  b = 6.28318531
i =  28 x = 5.56553774  y = -0.65761502 h = 0.26823858  b = 6.28318531
i =  29 x = 5.83966631  y = -0.42912109 h = 0.27412857  b = 6.28318531
i =  30 x = 6.12503238  y = -0.15749453 h = 0.28536607  b = 6.28318531
i =  31 x = 6.28318531  y = 0.00000015  h = 0.15815293  b = 6.28318531
```

Cash-Karp Solution

```
: <matplotlib.legend.Legend at 0x118f64ac0>
```

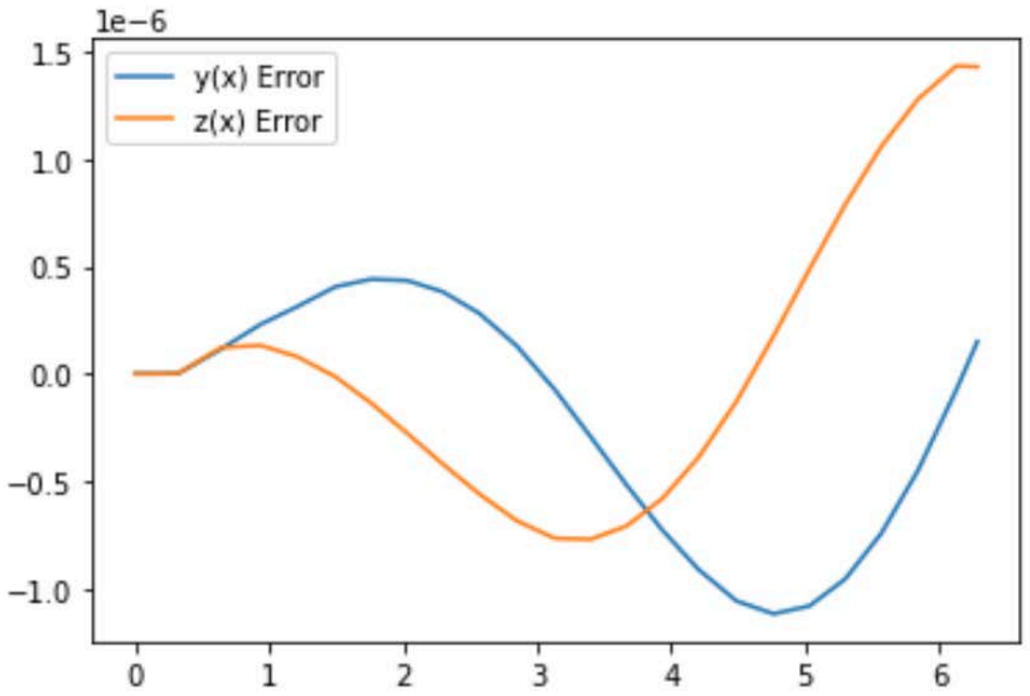


Cash-Karp Solution

Plot the error

31]:
```
plt.plot(x, y[:,0]-np.sin(x), label="y(x) Error")
plt.plot(x, y[:,1]-np.cos(x), label="z(x) Error")
plt.legend()
```

31]: <matplotlib.legend.Legend at 0x118fc7f70>



Final Projects

1) You will be asked to perform one of four final projects, working in a **group of up to 4 people of your choosing from your SECTION**. If you would like to be assigned to a group, please let me know immediately.

2) Please notify me of your group by **8:00am Thursday Nov 19**. Anyone who does not respond by **8:00am Thursday** letting me know their group or their desire to be assigned to a group will work on their own final project.

3) You will choose from one of the following 4 project topics, described in the following slides:
    1) Damped, driven pendulum
    2) Logistic map and chaos
    3) Astronomical source detection
    4) Monte Carlo Integration

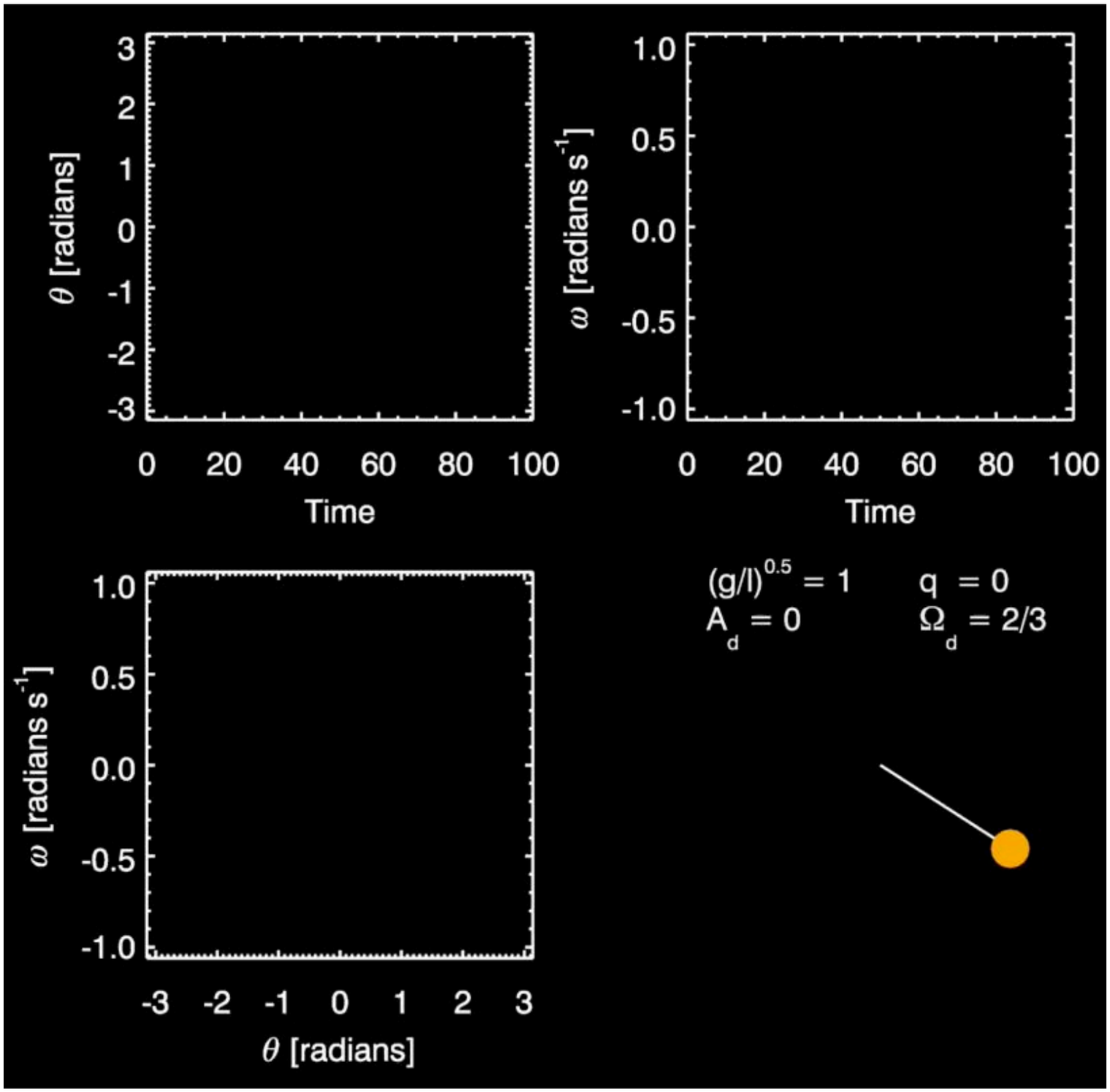
4) Groups must select a final project topic by **Tues Nov 24**. Detailed instructions for each project will be distributed Thursday Nov 19.

5) Groups should be organized through GitHub, invite your group members and us (TAs) as collaborators on your project. Tag us (TAs) when your final project is ready to grade.

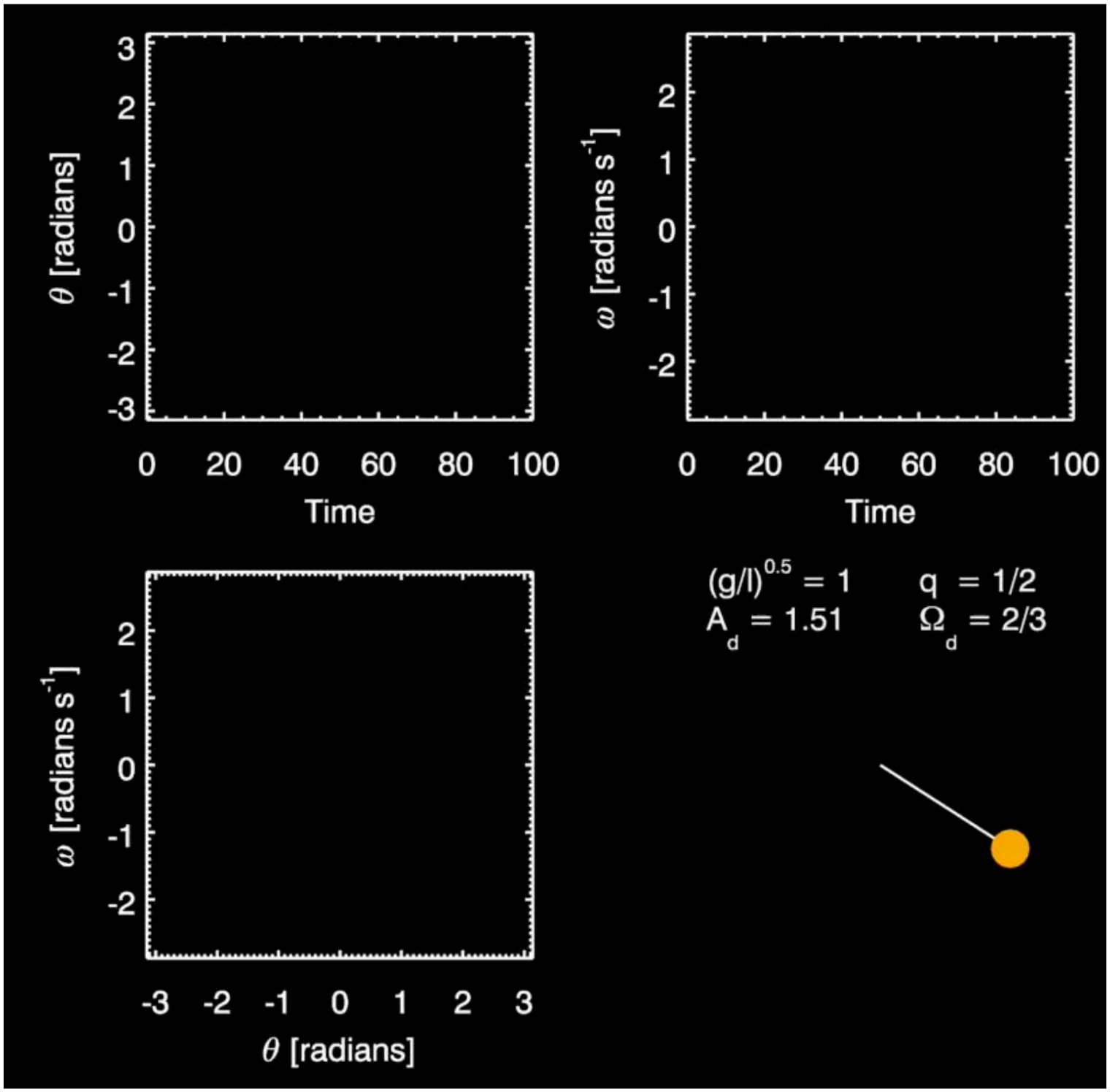
6) Each python module should indicate which student authored which part of the code.

7) Final projects are due (tags must happen by) **Tuesday, December 15, 2020 at 3pm.**

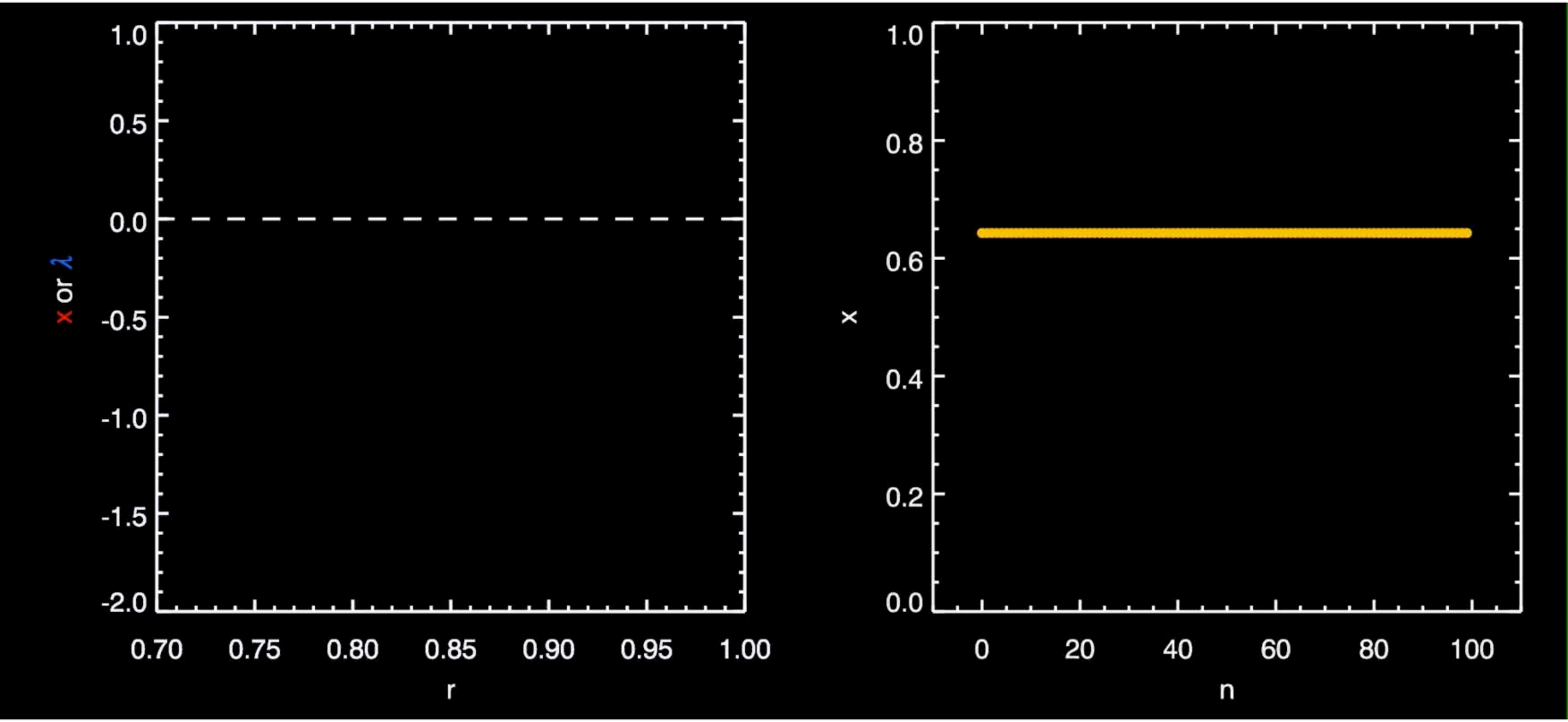
Final Projects 1: Forced, Damped Pendulum



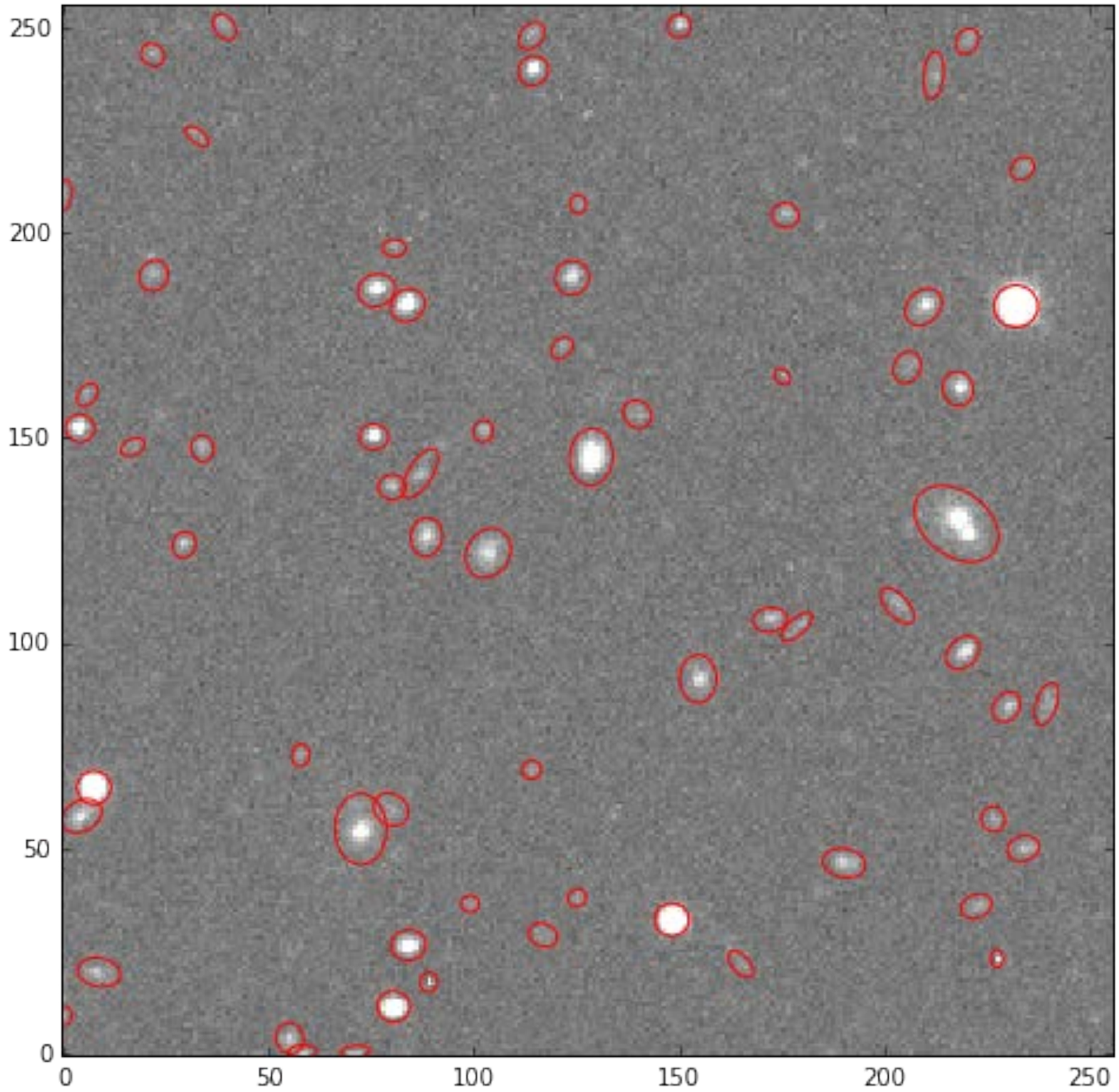
Final Projects 1: Forced, Damped Pendulum



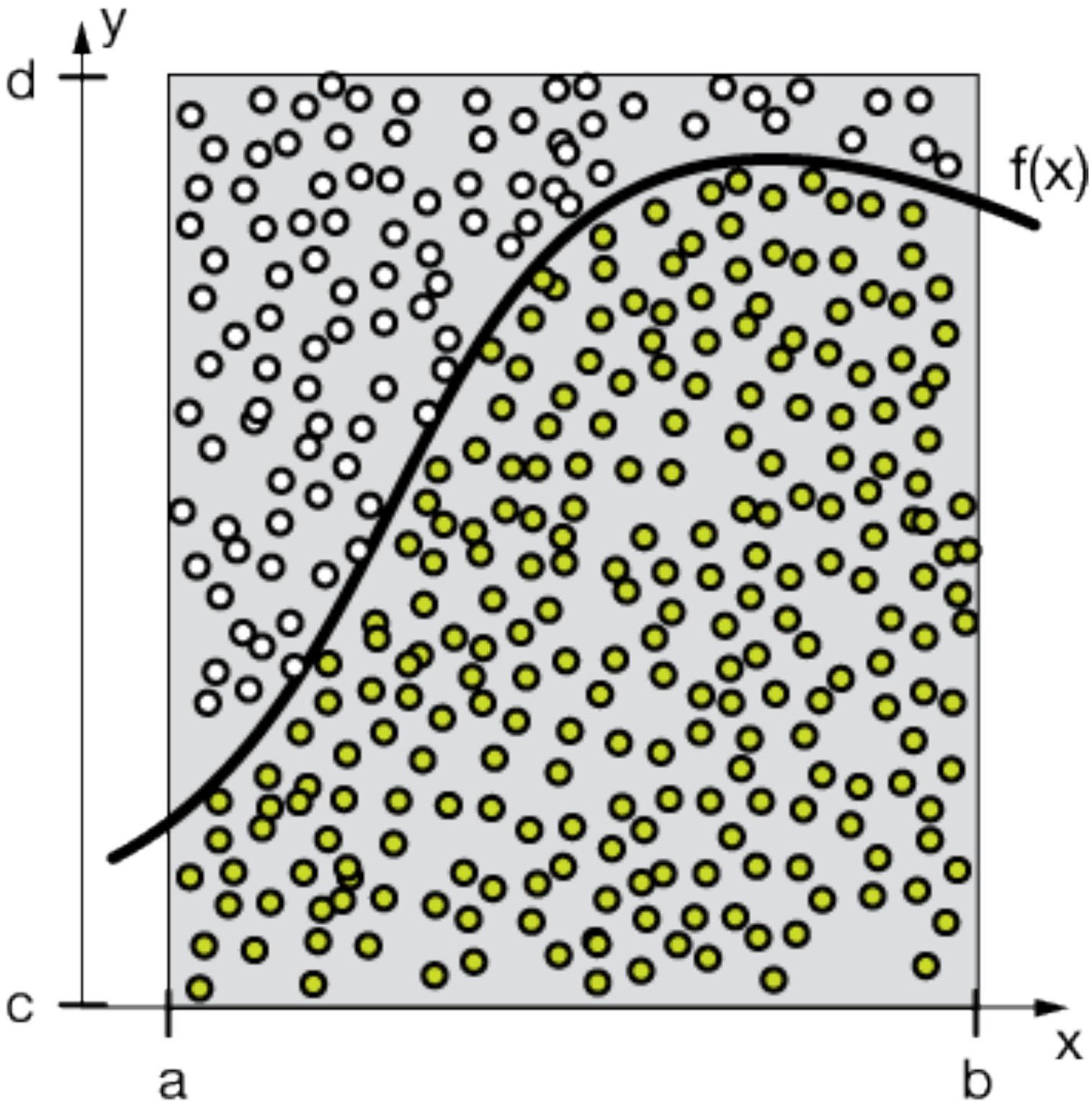
Final Projects 2: Logistic Map and Chaos



Final Projects 3: Astronomical Source Detection

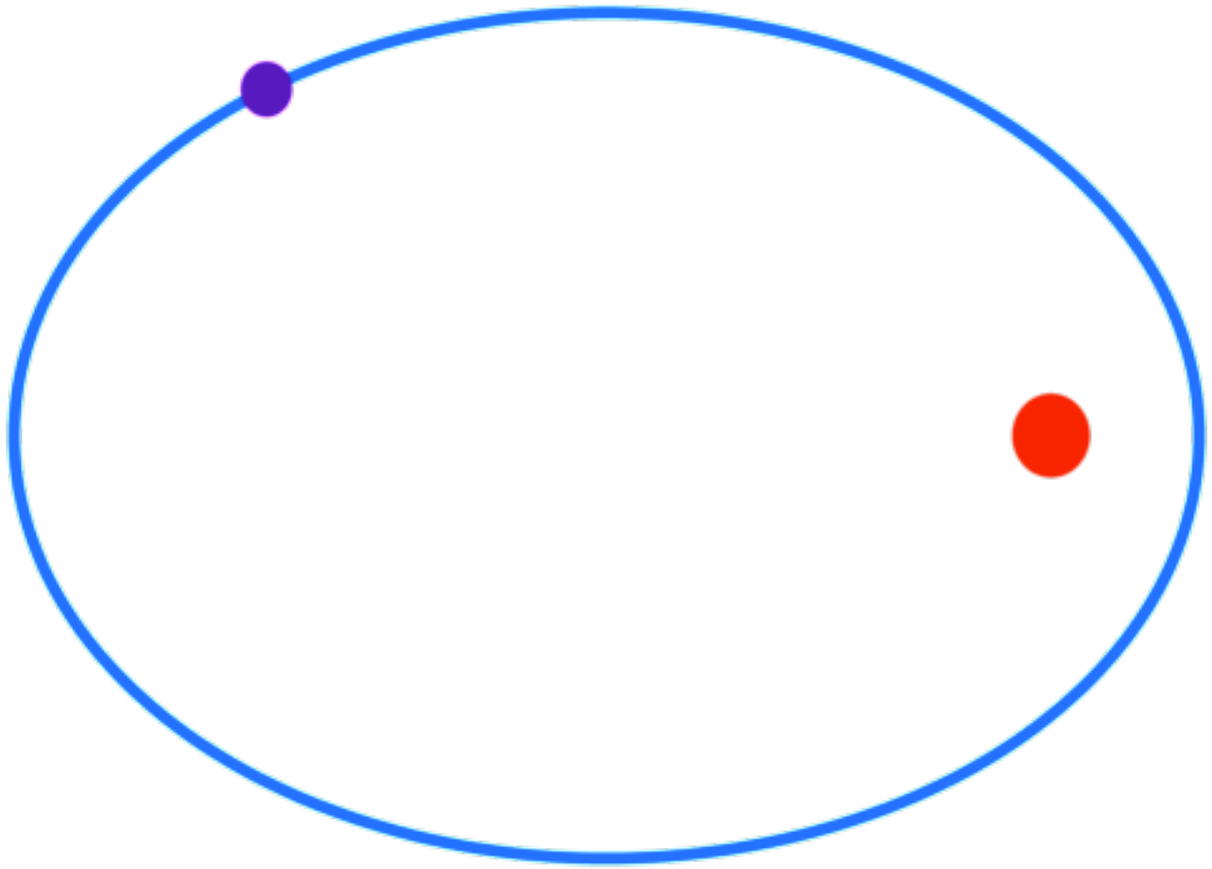


Final Projects 4: Monte Carlo Integrator



ORBITAL MECHANICS DEFINITIONS:

Consider the solar system, where the Sun effectively sits at the center of the system and the planets orbit about the Sun.



We are going to define a few useful quantities...

ORBITAL MECHANICS DEFINITIONS:

Perihelion distance, r_0:

Distance of closest approach in an elliptical orbit about the Sun

Aphelion distance, r_1:

Furthest distance in an elliptical orbit about the Sun

Semi-major axis of orbit *a*:

$2a = r_0 + r_1$

Eccentricity *e*:

Measure of non-circularity of an orbit

- $e < 1$: **ellipse**
- $e = 0$: **circle**
- $e = 1$: **parabola**
- $e > 1$: **hyperbola**

ORBITAL MECHANICS DEFINITIONS:

Relation between perihelion and aphelion distances:

$r_1 = r_0 (1+e)/(1-e)$

Velocity for a circular orbit:

$acceleration = v_c^2 / r = GM/r^2$

$v_c = (GM/r)^{1/2}$

Velocity at perihelion for an orbit with a given eccentricity:

$v_e = v_c(1+e)^{1/2}$, where v_c is evaluated at the perihelion r_0.

ORBITAL MECHANICS DEFINITIONS:

Solar Mass

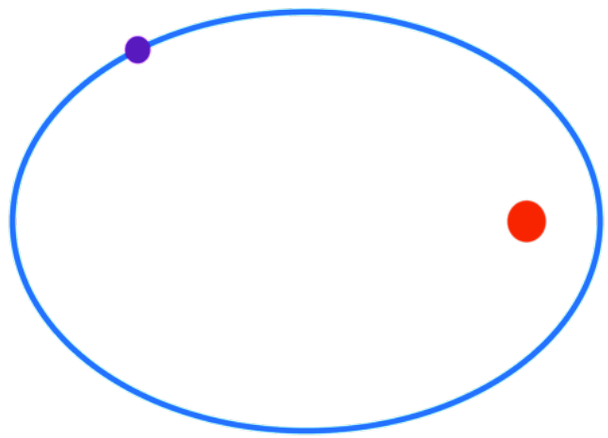
$$1\,\text{solar mass} = 1M_\odot = 1.98892 \times 10^{30}\text{kg}$$

Astronomical Unit (Mean distance between Earth and Sun):

$$1\,\text{AU} = 1.495979 \times 10^{11}\text{m}$$

1 year (Earth orbital time)

$$1\,\text{year} = 365.25\,\text{days} = 3.15567 \times 10^{7}\text{s}$$



ORBITAL MECHANICS DEFINITIONS:

Speed of light in solar system units

$$c = 2.997925 \times 10^8 \mathrm{m/s}$$

$$= 2.997925 \times 10^8 \mathrm{m/s} \left(\frac{1\mathrm{AU}}{1.495979 \times 10^{11}\mathrm{m}} \right) \left(\frac{3.15576 \times 10^7 \mathrm{s}}{1\mathrm{yr}} \right)$$

$$= 63421.1\,\mathrm{AU/yr}$$

MODELING OUR SOLAR SYSTEM

To make a model of our Solar System, we will begin by assuming all the bodies lie in a single plane. Hence, our system is two dimensional. We will only consider the effect of gravity:

$$\mathbf{F}_{ij} = -G\frac{m_i m_j}{r_{ij}^3}\mathbf{r}_{ij}$$

In many cases, a single object provides the dominant force. In our Solar System, the Sun is by far the most important but Jupiter also provides interesting perturbative effects. For our models, we will generally restrict ourselves to considering the gravitational force provided by just a few key objects.

VARIABLE TIMESTEPS

We have already discussed the importance of variable timesteps, both in the context of many body systems and for ODE integration in general. In a Solar System model, it's essential.

Consider the orbits of Mercury and Pluto. Mercury whizzes around the Sun at approximately 48km/s, while Pluto drifts slowly at about 5km/s. So, for Mercury we need to use a relatively small timestep to accurately capture its orbit. For Pluto, a larger timestep would do nicely.

VARIABLE TIMESTEPS

What if you wanted to model Mercury and Pluto simultaneously?

What timestep should you use? If we use a short timestep appropriate for Mercury's orbit, Pluto will use many more timesteps than necessary. If we use a long timestep appropriate for Pluto's orbit, then Mercury's orbit will be inaccurate.

While the former choice (short timesteps) does not hurt the accuracy, it does hurt the run-time since we will spend a lot of time unnecessarily evolving Pluto's orbit.

Since run-time will become an increasingly important issue, it is important to develop schemes that increase computational efficiency.

VARIABLE TIMESTEPS

In short, variable timesteps are motivated by two main reasons:

1) Variable timesteps preserve a similar accuracy of integration for all the bodies in the system.

2) Variable timesteps use the computational resources most efficiently.

VARIABLE TIMESTEPS

In a variable timestepping scheme, each object has its own timestep. In principle, this is not difficult but it does increase the amount of bookkeeping in a code.

In particular, there are two issues that need to be solved:

1) How do we pick the timestep for each object?

2) We must evolve all the objects in a synchronous manner.

PICKING THE TIMESTEP

There are a number of ways to choose a timestep for an object. One useful scheme is to force each object to move no more than some fixed distance ε in a timestep Δt. In the case of Mercury and Pluto, this would result in roughly ten times as long a timestep for Pluto as for Mercury, since its velocity is one-tenth as large.

An appropriate timestep criterion might be something like:

$$\Delta t = \frac{\epsilon}{|\mathbf{v}|}$$

So long as |v| is constant, (as in a nearly circular orbit) this will work fine. But more generally this will fail when |v| gets very small but the acceleration remains large.

PICKING THE TIMESTEP

But what about acceleration? We can choose the minimum of:

$$\Delta t = \frac{\epsilon}{|\mathbf{v}|} \qquad \Delta t = \frac{\epsilon}{\sqrt{|\mathbf{a}|}}$$

This will be our approach.

How do we pick ε? A smaller value will result in a smaller timestep. Our choice is therefore determined by the accuracy, but in practice it is usually selected by trial and error. In the case of a softened gravitational force law, we can relate ε to the softening length.

SYNCHRONIZATION

Another issue that arises with variable timesteps when each object has its own timestep and each object is evolved at different rates.

This is not optimal because usually we need to compute the force on between various objects, or plot their positions, or compute energy conservation, etc., for a specific time for all objects.

In our Mercury and Pluto example, if we simply advanced Mercury and Pluto three timesteps from a fixed initial time, then Mercury will not be nearly as far advanced in time as Pluto since Pluto has a much longer timestep. What we would rather have is that Mercury takes *many* steps for a single step of Pluto, such that at the end of Pluto's step Mercury and Pluto end up at the same time.

We call this *synchronization.*

SYNCHRONIZATION

One method for performing such synchronization is to choose an overall large timestep for the entire system, typically taken to be a constant (but could be variable -- like the timestep of Pluto).

Within the large timestep, there is a loop over all the bodies in the system. For each body, you compute its own timestep using our criteria, and then have another loop that evolves how many every timesteps that are needed to cover the full large timestep.

Since each object's timestep will not be an even multiple of the large timestep, you have to be careful not to exceed the large timestep on the last small timestep.

In the end there will be 3 nested loops:

1) The large timestep (main) loop
2) The loop over the number of bodies within 1)
3) The loop for each body's timestep within 2)

Verlet Equations

We could model position and velocity as follows:

$$x_{i+1} = x_i + v_i \Delta t + \frac{1}{2} a_i \Delta t^2 + O(\Delta t^3)$$

$$v_{i+1} = v_i + \frac{1}{2}(a_i + a_{i+1}) \Delta t + O(\Delta t^3)$$

These equations have the advantages that they are "self-starting", are second-order accurate, and easy to implement as long as the acceleration only depends on position (as is the case with gravity).

Unfortunately, for orbital motion the Verlet equations suffer from a small but serious deficiency....

LEAPFROG INTEGRATION

In the first few orbits, the motion determined by the Verlet equation is very close to the "real" motion. However, after many orbital periods the orbit will gradually diverge from the correct orbit.

This divergence can be slowed by reducing the timestep at the cost of computational time. But, in general, this scheme conserves energy and angular momentum poorly over large numbers of periods and is ill suited to the study of orbital motion.

Fortunately, there is an integration scheme that does not suffer from this gradually divergent behavior. While this scheme is also second-order accurate like the Verlet scheme, and therefore the error at each step is similar, on average the errors of this new scheme tend to average out rather than add coherently.

LEAPFROG INTEGRATION

We call this new scheme *leapfrog integration* for reasons that will become apparent, and it is a better choice for evolving systems over many dynamical times.

The leapfrog scheme has the advantage of being "symplectic", which means that the integrator is time-reversible. You can integrate the system forwards or backwards and arrive at the same ending or starting position.

For symplectic integration schemes, higher order errors tend to cancel out *on average* and hence such schemes maintain approximately the proper orbit forever.

LEAPFROG INTEGRATION

In the leapfrog scheme, the positions and velocities are "leapfrogged" over each other, with one being advanced between the full timesteps (e.g., 0, 1, 2, 3 ...) and the other being advanced between "halfsteps" (e.g., 1/2, 3/2, 5/2....). A full timestep thus progresses as follows:

$$x_{i+\frac{1}{2}} = x_i + \frac{1}{2} v_i \Delta t$$

$$v_{i+1} = v_i + a_{i+\frac{1}{2}} \Delta t$$

$$x_{i+1} = x_{i+\frac{1}{2}} + \frac{1}{2} v_{i+1} \Delta t$$

These equations are similar to the Verlet equations, but the acceleration at the half timestep is used to evolve the velocity. The order of accuracy is the same as the Verlet equations.

LEAPFROG INTEGRATION

One complication with the leapfrog method is that it is not precisely self-starting. The very first advance of position from x_0 to $x_{1/2}$ is only first-order accurate. Hence, if one uses the leapfrog scheme starting from t=0, the first halfstep is first-order accurate, and hence the entire calculation becomes first-order accurate!

Fortunately, this can be easily remedied. To do so, the initialization of the position at the *very first* halfstep must be evolved according to a second-order accurate equation:

$$x_{i+\frac{1}{2}} = x_i + \frac{1}{2} v_i \Delta t + \frac{1}{4} a_i \Delta t^2 + O(\Delta t^3)$$

This is done *once*, and then you continue on with the leapfrog scheme.

Solar System Model

We want to:

1) Make a model of the solar system to evolve planetary orbits.

2) Start with the orbits of Mercury, Venus, and Earth about the Sun.

3) Allow for non-circular orbits, accounting for eccentricity.

4) Use a symplectic integrator to average down energy and angular momentum errors.

5) We want to use variable time steps for the planets, governed by a global time step for which we record the current properties of the planets.

Solar System Model

Since we are dealing with “planets” as the primary logical unit of our model, it makes sense to design our code around a “Planet” object. This object will have the following members:

1) Position in two dimensions (x,y)
2) Velocity in two dimensions (vx, vy)
3) Acceleration in two dimensions (ax, ay)
4) A time (which may differ from the global time of the system)
5) A timestep (which may differ from the global time step of the system)
6) An eccentricity for its orbit (sets the initial conditions)
7) A semi-major axis (sets the initial conditions)
8) An iterative variable that tracks the number of time steps this planet has taken.

Solar System Model

Create a simple solar system model

In [1]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
from collections import namedtuple
```

Define a planet class

In [3]:
```
class planet():
    "A planet in our solar system"
    def __init__(self,semimajor,eccentricity):
        self.x   = np.array(2) #x and y position
        self.v   = np.array(2) #x and y velocity
        self.a_g = np.array(2) #x and y acceleration
        self.t  = 0.0          #current time
        self.dt = 0.0          #current timestep
        self.a  = semimajor    #semimajor axis of the orbit
        self.e  = eccentricity #eccentricity of the orbit
        self.istep  = 0        #current integer timestepl

```

Solar System Model

The basic physical model that we have is that the planets orbit about the Sun, with the gravitational force supplying the centripetal acceleration to maintain the orbit. As a result, there is a minimum amount of information about the solar system that we need.

1) Mass of the sun
2) The gravitational constant G
3) The circular velocity about the sun at any location.
4) The gravitational acceleration about the sun at any location.

Solar System Model

Define a dictionary with some constants

In [ ]:
```
1 solar_system = { "M_sun":1.0, "G":39.4784176043574320}
```

Solar System Model

Define some functions for setting circular velocity, and acceleration

In [6]:
```
def solar_circular_velocity(p,solar_system):

    G = solar_system["G"]
    M = solar_system["M_sun"]
    r = ( p.x[0]**2 + p.x[1]**2 )**0.5

    #return the circular velocity
    return (G*M/r)**0.5

```

In [ ]:
```
def solar_gravitational_acceleration(p, solar_system):

    G = solar_system["G"]
    M = solar_system["M_sun"]
    r = ( p.x[0]**2 + p.x[1]**2 )**0.5

    #acceleration in AU/yr/yr
    a_grav = -1.0*G*M/r**2

    #find the angle at this position
    if(p.x[0]==0.0):
        if(p.x[1]>0.0):
            theta = 0.5*np.pi
        else:
            theta = 1.5*np.pi
    else:
        theta = np.atan(p.x[1],p.x[0])

    #set the x and y components of the velocity
    p.a_g[0] = a_grav * np.cos(theta)
    p.a_g[1] = a_grav * np.sin(theta)
```

Solar System Model

Compute the timestep

In [ ]:

```
def calc_dt(p):

    #integration tolerance
    ETA_TIME_STEP = 0.0004

    #compute timestep
    eta = ETA_TIME_STEP
    v = (p.v[0]**2 + p.v[1]**2)**0.5
    a = (p.a_g[0]**2 + p.a_g[1]**2)**0.5
    dt = eta * fp.min(1./np.fabs(v),1./fabs(a))

    return dt
```

Solar System Model

As with any differential, we need to set the initial conditions. Since we are treating the planets as independent (currently), we can initialize the planets independently. For each planet, we need to

1) Set the semi-major axis of the orbit.
2) Set the eccentricity.
3) Initialize the position at time t=t_init.
4) Initialize the velocity at time t=t_init.
5) Calculate the initial acceleration at time t = t_init.
6) Use the initial velocity and acceleration to determine the time step.

Solar System Model

Define the initial conditions

In [ ]:

```
def SetPlanet(p, i):

    AU_in_km = 1.495979e+8 #an AU in km

    #circular velocity
    v_c = 0.0      #circular velocity in AU/yr
    v_e = 0.0      #velocity at perihelion in AU/yr

    #planet-by planet initial conditions

    #Mercury
    if(i==0):
        #semi-major axis in AU
        p.a = 57909227.0 #AU_in_km

        #eccentricity
        p.e = 0.20563593

    #Venus
    elif(i==1):
        #semi-major axis in AU
        p.a = 108209475.0 #AU_in_km

        #eccentricity
        p.e = 0.00677672

    #Earth
    elif(i==2):
        #semi-major axis in AU
        p.a = 1.0

        #eccentricity
        p.e = 0.01671123
```

Solar System Model

```
32          #eccentricity
33          p.e = 0.01671123
34
35      #set remaining properties
36      p.t = 0.0
37      p.x[0] = p.a*(1.0-p.e)
38      p.x[1] = 0.0
39
40      #get equiv circular velocity
41      v_c = solar_circular_velocity(p)
42
43      #velocity at perihelion
44      v_e = v_c*(1 + p.e)**0.5
45
46      #set velocity
47      p.v[0] = 0.0      #no x velocity at perihelion
48      p.v[1] = v_e      #y velocity at perihelion (counter clockwise)
49
50      #calculate gravitational acceleration from Sun
51      solar_gravitational_acceleration(p)
52
53      #set timestep
54      p.dt = calc_dt(p)
55
```

Cell cont.

Solar System Model

As we've discussed, for conservative systems it is desirable to use a symplectic integration method that obeys a Hamiltonian of the system. The leapfrog method is one such method, so let's use that. To do so, we'll need:

1) A function to take the first position step (special for the first step only).
2) A function to take a full step in position.
3) A function to take a full step in velocity.

Solar System Model

Write leapfrog integrator

In [ ]:
```
def x_first_step(x_i, v_i, a_i, dt):
    #x_1/2 = x_0 + 1/2 v_0 Delta_t + 1/4 a_0 Delta t^2
    return x_i + 0.5*v_i*dt + 0.25*a_i*dt**2
```

In [ ]:
```
def v_full_step(x_i, v_i, a_ipoh, dt):
    #v_i+1 = v_i + a_i+1/2 Delta_t
    return v_i + a_ipoh*dt;
```

In [ ]:
```
def x_full_step(x_ipoh, v_ip1, a_ipoh, dt):
    #x_3/2 = x_1/2 + v_i+1 Delta t
    return x_ipoh + v_ip1*dt;
```

Solar System Model

We have chosen an integration method, but now we need to consider all the overhead involved in calculating the orbits. We'll want a file to save the information about the solar system over time, and we'll want to write to it in a convenient format. We'll need to evolve the global system many timesteps, and evolve each planet over its own time step potentially many times per global timestep. So, we'll need:

1) A driving routine to perform the bookkeeping of the orbital integration.
2) A routine to open the file we'll write the data out to.
3) A routine to write the data to file each global timestep.
4) To update the position and velocity according to the leapfrog integrator.
5) Update the planetary time steps according to their velocity and acceleration.

Solar System Model

We'd like to save the results of the simulation and use that data to visualize the solar system. We want our data files to be efficient and easy to use (these are sometimes at cross purposes). Also, we want the position and velocities to be written at the same time (but they are offset by 1/2 time step in our integrator). We need to:

1) Open a data file.
2) Write to the data file in a useful format.
3) Synchronize the position and velocity data when writing to the file.

Solar System Model

Write a function to save the data to file

```
]:
```

```
def SaveSolarSystem(p, n_planets, t, dt, istep, ndim):

    #loop over the number of planets
    for i in range(n_planets):

        #define a filename
        fname = "planet.%s.txt" % p[i].name

        if(istep==0):
            #create the file on the first timestep
            fp = open(fname,"w")
        else:
            #append the file on subsequent timesteps
            fp = open(fname,"a")

        #compute the drifted properties of the planet
        v_drift = np.zeros(ndim)

        for k in range(ndim):
            v_drift[k] = p[i].v[k] + 0.5*p[i].a_g[k]*p[i].dt

        #write the data to file
        s = "%6d\t%6.5f\t%6.5f\t%6d\t%6.5f\t%6.5f\t% 6.5f\t% 6.5f\t% 6.5f\t% 6.5f\t% 6.5f\t% 6.5f\n" % \
            (istep,t,dt,p[i].istep,p[i].t,p[i].dt,p[i].x[0],p[i].x[1],v_drift[0],v_drift[1],      \
             p[i].a_g[0],p[i].a_g[1])
        fp.write(s)


        #close the file
        fp.close()
```

Solar System Model

Write a function to evolve the solar system

]:

```
def EvolveSolarSystem(p,n_planets,t_max):

    #number of spatial dimensions
    ndim = 2

    #define the first timestep
    dt = 0.5/365.25

    #define the starting time
    t = 0.0

    #define the starting timestep
    istep = 0

    #save the initial conditions
    SaveSolarSystem(p,n_planets,t,dt,istep,ndim)

    #begin a loop over the global timescale
    while(t<t_max):

        #check to see if the next step exceeds the
        #maximum time.  If so, take a smaller step
        if(t+dt>t_max):
            dt = t_max - t # limit the step to align with t_max

        #evolve each planet
        for i in range(n_planets):
```

Solar System Model

Cell cont

```
26          #evolve each planet
27          for i in range(n_planets):
28
29              while(p[i].t<t+dt):
30
31                  #special case for istep==0
32                  if(p[i].istep==0):
33
34                      #take the first step according to a verlet scheme
35                      for k in range(ndim):
36                          p[i].x[k] = x_first_step(p[i].x[k],p[i].v[k],p[i].a_g[k],p[i].dt)
37
38                      #update the acceleration
39                      p[i].a_g = SolarGravitationalAcceleration(p[i])
40
41                      #update the time by 1/2dt
42                      p[i].t += 0.5*p[i].dt
43
44                      #update the timestep
45                      p[i].dt = calc_dt(p[i])
46
47                  #continue with a normal step
48
49                  #limit to align with the global timestep
50                  if(p[i].t + p[i].dt > t+dt):
51                      p[i].dt = t+dt-p[i].t
```

Solar System Model

Cell cont

```
                #limit to align with the global timestep
                if(p[i].t + p[i].dt > t+dt):
                    p[i].dt = t+dt-p[i].t

                #evolve the velocity
                for k in range(ndim):
                    p[i].v[k] = v_full_step(p[i].v[k],p[i].a_g[k],p[i].dt)

                #evolve the position
                for k in range(ndim):
                    p[i].x[k] = x_full_step(p[i].x[k],p[i].v[k],p[i].a_g[k],p[i].dt)

                #update the acceleration
                p[i].a_g = SolarGravitationalAcceleration(p[i])

                    #update by dt
                    p[i].t += p[i].dt

                    #compute the new timestep
                    p[i].dt = calc_dt(p[i])

                    #update the planet's timestep
                    p[i].istep+=1

            #now update the global system time
            t+=dt
```

Solar System Model

Cell cont

```
            #now update the global system time
            t+=dt

            #update the global step number
            istep += 1

            #output the current state
            SaveSolarSystem(p,n_planets,t,dt,istep,ndim)

        #print the final steps and time
        print("Time t = ",t)
        print("Maximum t = ",t_max)
        print("Maximum number of steps = ",istep)

        #end of evolution
```

Solar System Model

Create a routine to read in the data

```
]: def read_twelve_arrays(fname):
       fp = open(fname,"r")
       fl = fp.readlines()
       n = len(fl)
       a = np.zeros(n)
       b = np.zeros(n)
       c = np.zeros(n)
       d = np.zeros(n)
       f = np.zeros(n)
       g = np.zeros(n)
       h = np.zeros(n)
       j = np.zeros(n)
       k = np.zeros(n)
       l = np.zeros(n)
       m = np.zeros(n)
       p = np.zeros(n)
       for i in range(n):
           a[i] = float(fl[i].split()[0])
           b[i] = float(fl[i].split()[1])
           c[i] = float(fl[i].split()[2])
           d[i] = float(fl[i].split()[3])
           f[i] = float(fl[i].split()[4])
           g[i] = float(fl[i].split()[5])
           h[i] = float(fl[i].split()[6])
           j[i] = float(fl[i].split()[7])
           k[i] = float(fl[i].split()[8])
           l[i] = float(fl[i].split()[9])
           m[i] = float(fl[i].split()[10])
           p[i] = float(fl[i].split()[11])

       return a,b,c,d,f,g,h,j,k,l,m,p
```

Solar System Model

Perform the integration of the solar system

45]:

```
#set the number of planets
n_planets = 3

#set the maximum time of the simulation
t_max = 2.0

#create empty list of planets
p = []

#set the planets
for i in range(n_planets):

    #create an empty planet
    ptmp = planet(0.0,0.0)

    #set the planet properties
    SetPlanet(ptmp,i)

    #remember the planet
    p.append(ptmp)

#evolve the solar system
EvolveSolarSystem(p,n_planets,t_max)
```

```
Time t =  2.000000000000041
Maximum t =  2.0
Maximum number of steps =  1464
```

Solar System Model

Read the data back in for every planet

```
fname = "planet.Mercury.txt"
istepMg,tMg,dtMg,istepM,tM,dtM,xM,yM,vxM,vyM,axM,ayM = read_twelve_arrays(fname)
```

```
fname = "planet.Earth.txt"
istepEg,tEg,dtEg,istepE,tE,dtE,xE,yE,vxE,vyE,axE,ayE = read_twelve_arrays(fname)
```

```
fname = "planet.Venus.txt"
istepVg,tVg,dtVg,istepV,tV,dtV,xV,yV,vx,vyV,axV,ayV = read_twelve_arrays(fname)
```

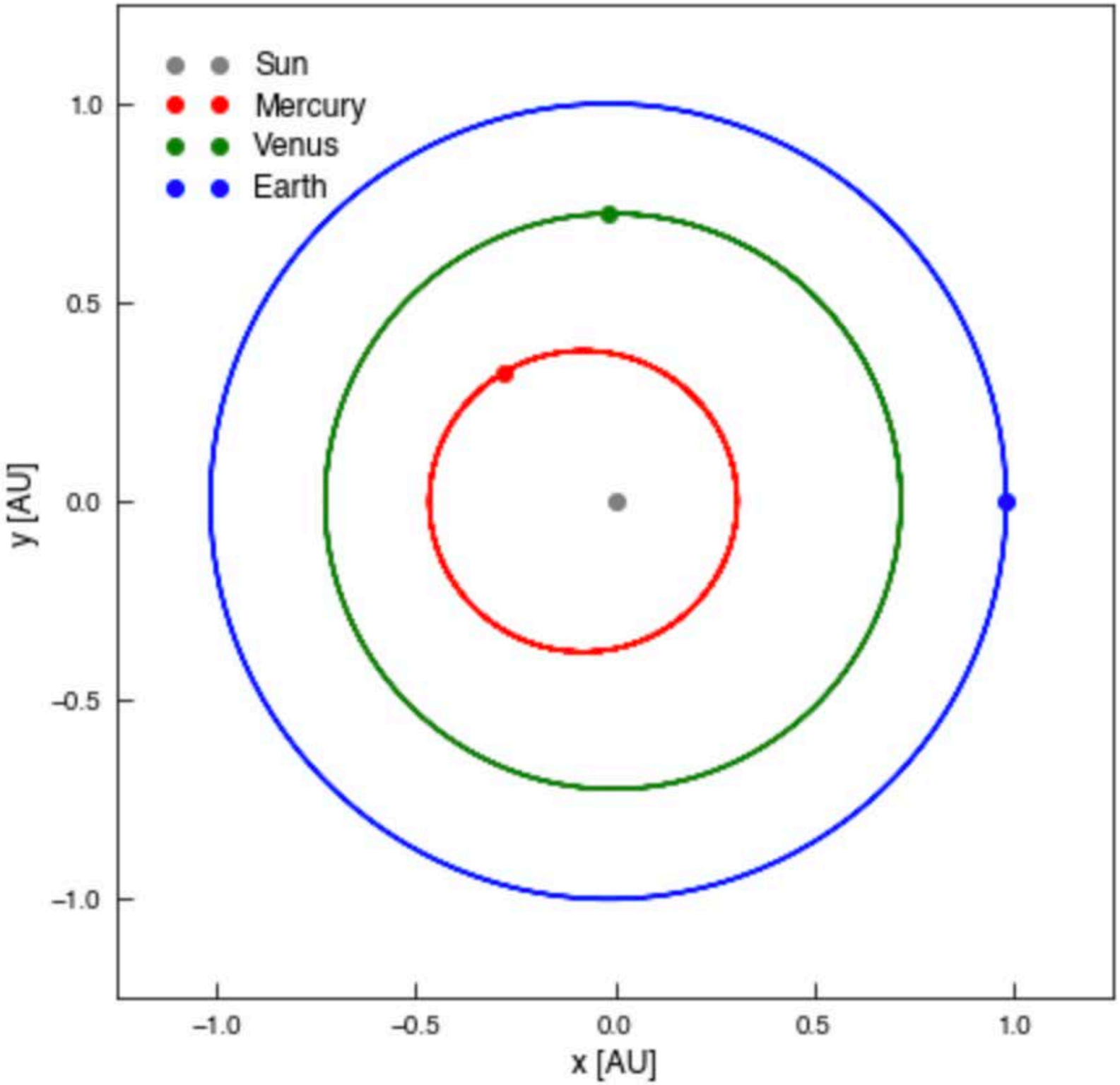
Solar System Model

Plot the data

```
:
```

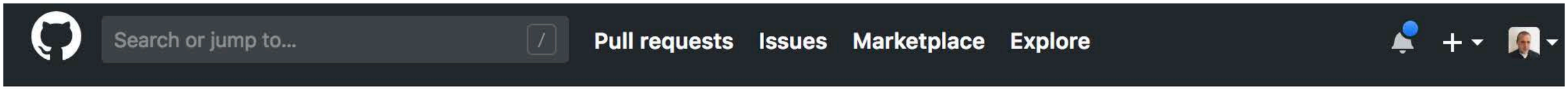
```
fig = plt.figure(figsize=(7,7))

xSun = [0.0]
ySun = [0.0]
plt.plot(xSun,ySun,'o',color="0.5",label="Sun")

plt.plot(xM,yM,color="red")
plt.plot(xM[-1],yM[-1],'o',color="red",label="Mercury")

plt.plot(xV,yV,color="green")
plt.plot(xV[-1],yV[-1],'o',color="green",label="Venus")

plt.plot(xE,yE,color="blue")
plt.plot(xE[-1],yE[-1],'o',color="blue",label="Earth")



plt.xlim([-1.25,1.25])
plt.ylim([-1.25,1.25])
plt.xlabel('x [AU]')
plt.ylabel('y [AU]')
plt.axes().set_aspect('equal')
plt.legend(frameon=False,loc=2)
```

Solar System Model



Save Your Work

Make a GitHub project “astr-119-session-14”, and commit the programs my_first_jupyter_notebook.ipynb and test_matplotlib.ipynb you made today.



Create a new repository

A repository contains all the files for your project, including the revision history.

Owner **Repository name**

brantr / astr-119-session-5

Great repository names are short and memorable. Need inspiration? How about **fantastic-spork**.

Description (optional)

We learned a new trick! -- Jupyter notebooks.

Public
Anyone can see this repository. You choose who can commit.

Private
You choose who can see and commit to this repository.