ASTR 119: Session 15

Orbital Mechanics

Outline

1) Code Check In
2) Visualization of the Day
3) Final projects
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.

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.**

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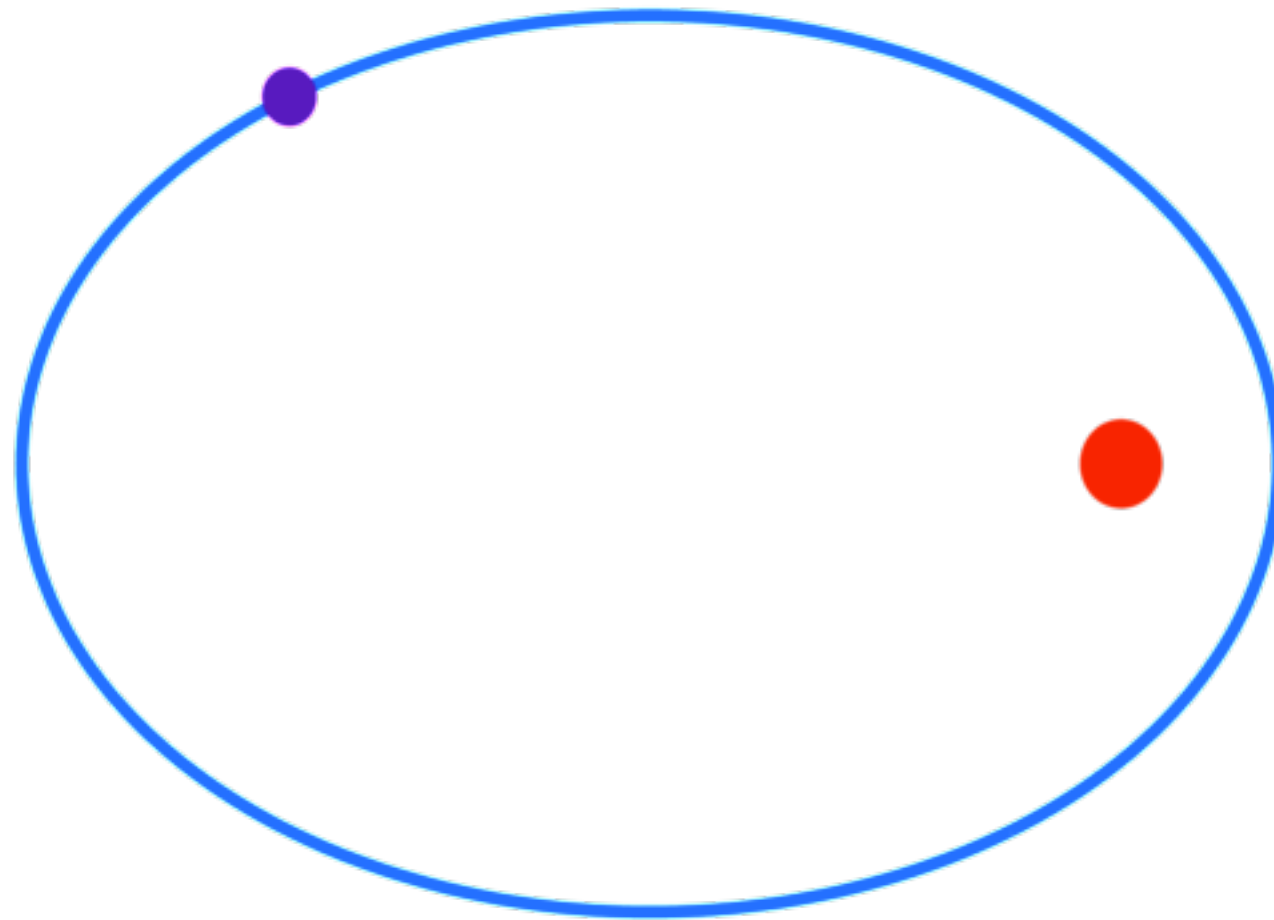
We are going to define a few useful quantities...

Solar System Model

Create a simple solar system model

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

Define a planet class

```
In [2]:   1  class planet():
          2      "A planet in our solar system"
          3      def __init__(self,semimajor,eccentricity):
          4          self.x   = np.zeros(2) #x and y position
          5          self.v   = np.zeros(2) #x and y velocity
          6          self.a_g = np.zeros(2) #x and y acceleration
          7          self.t  = 0.0           #current time
          8          self.dt = 0.0           #current timestep
          9          self.a  = semimajor     #semimajor axis of the orbit
         10          self.e  = eccentricity #eccentricity of the orbit
         11          self.istep  = 0         #current integer timestepl
         12          self.name   = ""        #name for the planet
         13
```

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 [35]:
```
1  solar_system = { "M_sun":1.0, "G":39.4784176043574320}
```

Define some functions for setting circular velocity, and acceleration

In [4]:
```
1  def SolarCircularVelocity(p):
2
3      G = solar_system["G"]
4      M = solar_system["M_sun"]
5      r = ( p.x[0]**2 + p.x[1]**2 )**0.5
6
7      #return the circular velocity
8      return (G*M/r)**0.5
9
```

Solar System Model

Write a function to compute the gravitational acceleration on each planet from the Sun

```python
def SolarGravitationalAcceleration(p):

    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.arctan2(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)
    return a_grav*np.cos(theta), a_grav*np.sin(theta)
```

Solar System Model

Compute the timestep

```python
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 * np.fmin(1./np.fabs(v),1./np.fabs(a)**0.5)

    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

```python
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

        #name
        p.name = "Mercury"

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

        #eccentricity
        p.e = 0.00677672

        #name
        p.name = "Venus"
```

Solar System Model

```python
30          #name
31          p.name = "Venus"
32
33      #Earth
34      elif(i==2):
35          #semi-major axis in AU
36          p.a = 1.0
37
38          #eccentricity
39          p.e = 0.01671123
40
41          #name
42          p.name = "Earth"
43
44      #set remaining properties
45      p.t = 0.0
46      p.x[0] = p.a*(1.0-p.e)
47      p.x[1] = 0.0
48
49      #get equiv circular velocity
50      v_c = SolarCircularVelocity(p)
51
52      #velocity at perihelion
53      v_e = v_c*(1 + p.e)**0.5
54
55      #set velocity
56      p.v[0] = 0.0        #no x velocity at perihelion
57      p.v[1] = v_e        #y velocity at perihelion (counter clockwise)
58
59      #calculate gravitational acceleration from Sun
60      p.a_g = SolarGravitationalAcceleration(p)
61
62      #set timestep
63      p.dt = calc_dt(p)
64
```

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

```python
]: 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
```

```python
]: def v_full_step(v_i, a_ipoh, dt):
       #v_i+1 = v_i + a_i+1/2 Delta t
       return v_i + a_ipoh*dt;
```

```python
]: 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

```python
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

```python
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

```python
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

```python
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
52
53              #evolve the velocity
54              for k in range(ndim):
55                  p[i].v[k] = v_full_step(p[i].v[k],p[i].a_g[k],p[i].dt)
56
57              #evolve the position
58              for k in range(ndim):
59                  p[i].x[k] = x_full_step(p[i].x[k],p[i].v[k],p[i].a_g[k],p[i].dt)
60
61              #update the acceleration
62              p[i].a_g = SolarGravitationalAcceleration(p[i])
63
64              #update by dt
65              p[i].t += p[i].dt
66
67              #compute the new timestep
68              p[i].dt = calc_dt(p[i])
69
70              #update the planet's timestep
71              p[i].istep+=1
72
73          #now update the global system time
74          t+=dt
```

Solar System Model

Cell cont

```
73          #now update the global system time
74          t+=dt
75
76          #update the global step number
77          istep += 1
78
79          #output the current state
80          SaveSolarSystem(p,n_planets,t,dt,istep,ndim)
81
82      #print the final steps and time
83      print("Time t = ",t)
84      print("Maximum t = ",t_max)
85      print("Maximum number of steps = ",istep)
86
87      #end of evolution
```

Solar System Model

Create a routine to read in the data

```python
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]:   1   #set the number of planets
       2   n_planets = 3
       3
       4   #set the maximum time of the simulation
       5   t_max = 2.0
       6
       7   #create empty list of planets
       8   p = []
       9
      10   #set the planets
      11   for i in range(n_planets):
      12
      13       #create an empty planet
      14       ptmp = planet(0.0,0.0)
      15
      16       #set the planet properties
      17       SetPlanet(ptmp,i)
      18
      19       #remember the planet
      20       p.append(ptmp)
      21
      22   #evolve the solar system
      23   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

```python
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
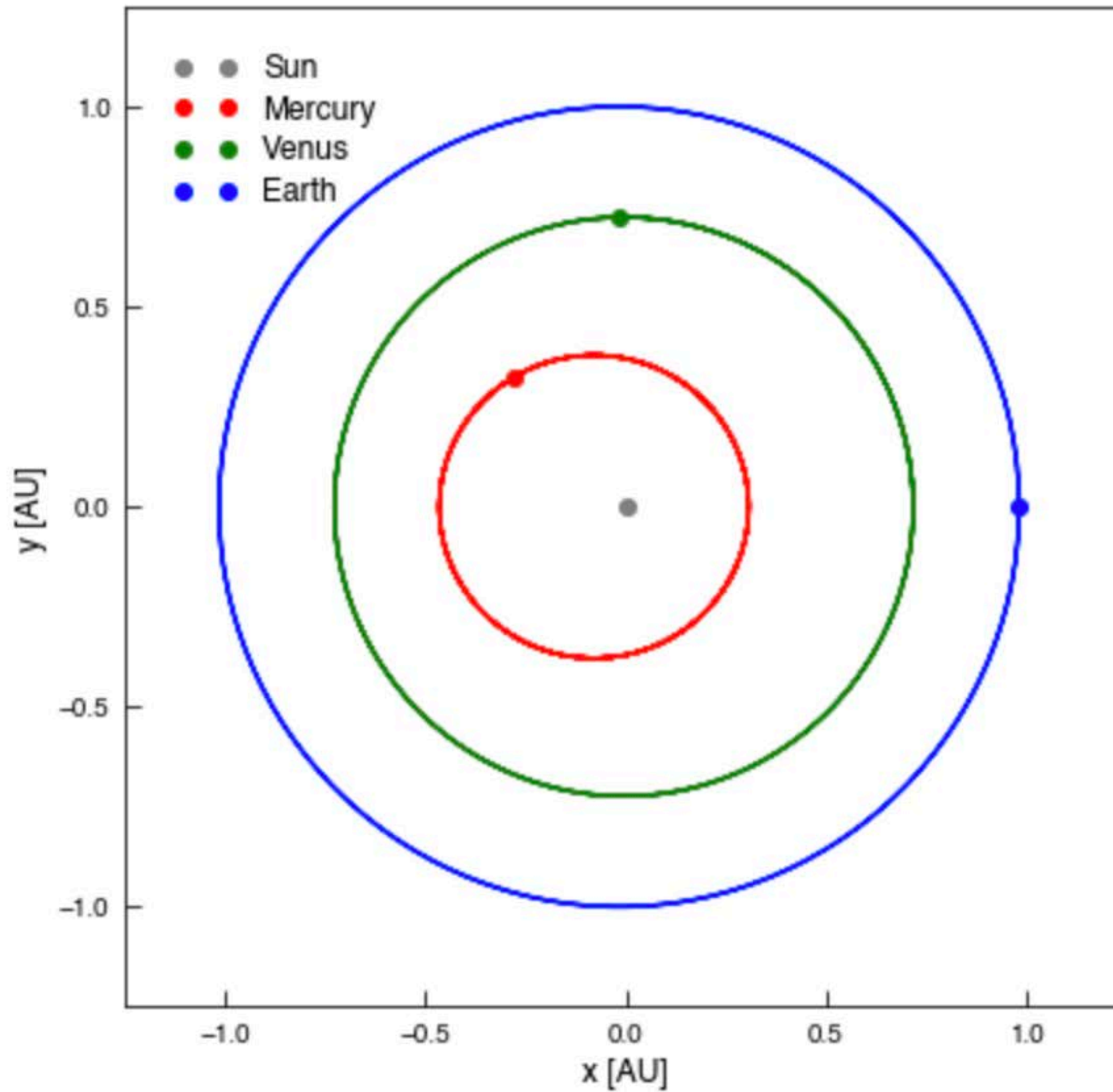
Save Your Work

Make a GitHub project "astr-119-session-15", and commit the programs my_first_jupyter_notebook.ipynb and test_matplotlib.ipynb you made today.