ASTR 119: Session 16

Orbital Mechanics
Random Numbers

Outline

1) Visualization of the Day
2) Final projects
3) Orbital Mechanics
4) Final project organization time
5) Save your work to GitHub

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.

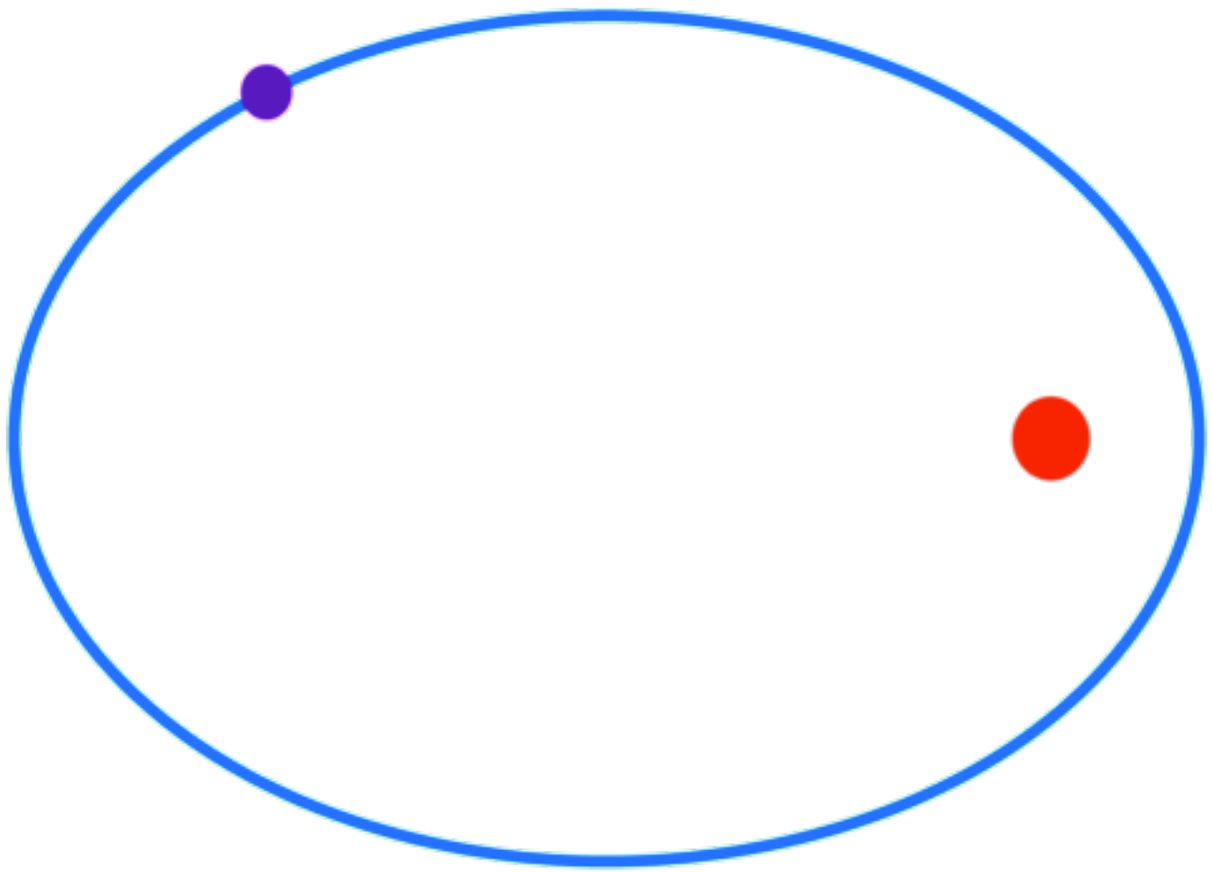
5) Groups should be organized through GitHub, invite your group members and us (TAs) as collaborators on your project. Tag us (TAs) when your final project is ready to grade.

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

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

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

            while(p[i].t<t+dt):

                #special case for istep==0
                if(p[i].istep==0):

                    #take the first step according to a verlet scheme
                    for k in range(ndim):
                        p[i].x[k] = x_first_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 the time by 1/2dt
                    p[i].t += 0.5*p[i].dt

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

                #continue with a normal step

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

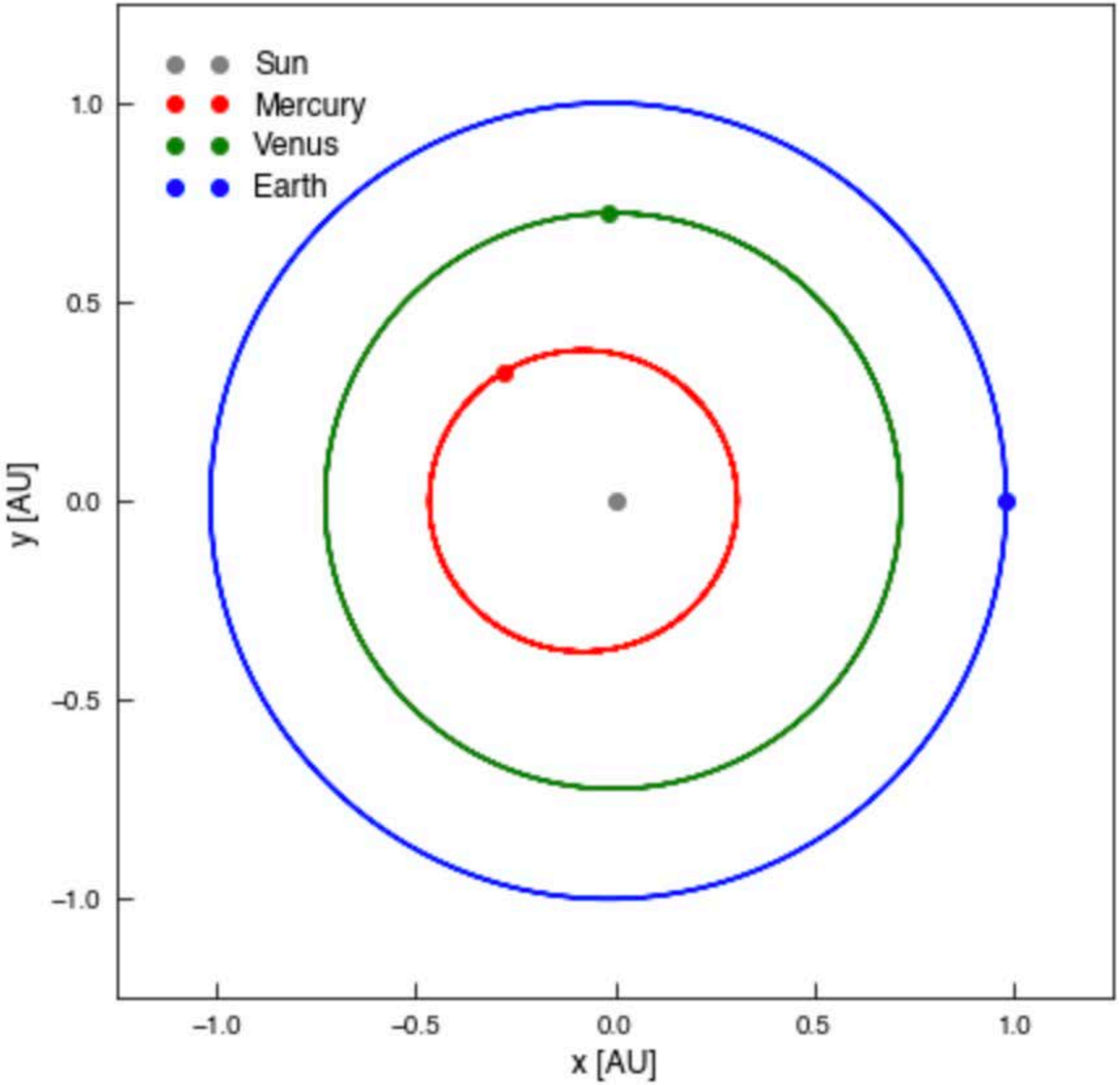
```
fname = "planet.Venus.txt"
istepVg,tVg,dtVg,istepV,tV,dtV,xV,yV,vx,vyV,axV,ayV = read_twelve_arrays(fname)
```

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



Random Number Generation

In science, we often have to “fake it till we make it”, meaning that in order to understand our experiments and their systematic uncertainties we need to model our data. Given both natural randomness and measurement error, these models amount to random number generation.

Often, we would like to generate random numbers distributed from a wide variety of distributions. If we can calculate the integral of the probability density distribution we wish to pull random numbers from, it is possible to use uniformly distributed random numbers to generate random numbers from a general distribution.

Given the complexity of this problem, let’s take some time to learn about publicly available libraries to help us with a variety of computational tasks including random number generation. This will be useful for many problems you might encounter in the future.

Random Number Generation

Use a random number generator to simulate a simple Gaussian process

In [2]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
```

Set some properties of the random system

In [5]:
```
n_samples = 10000 #number of random samples to use
n_bins    = 100   #number of bins for our histogram
sigma     = 1.0   #rms width of the gaussian
mu        = 0.0   #mean of the gaussian
```

Random Number Generation

Generate the random numbers using numpy

In [15]:
```
x = np.random.normal(mu,sigma,n_samples)
print(x.min())
print(x.max())
```

```
-3.76402898364
3.70499724783
```

Define a function to plot a Gaussian

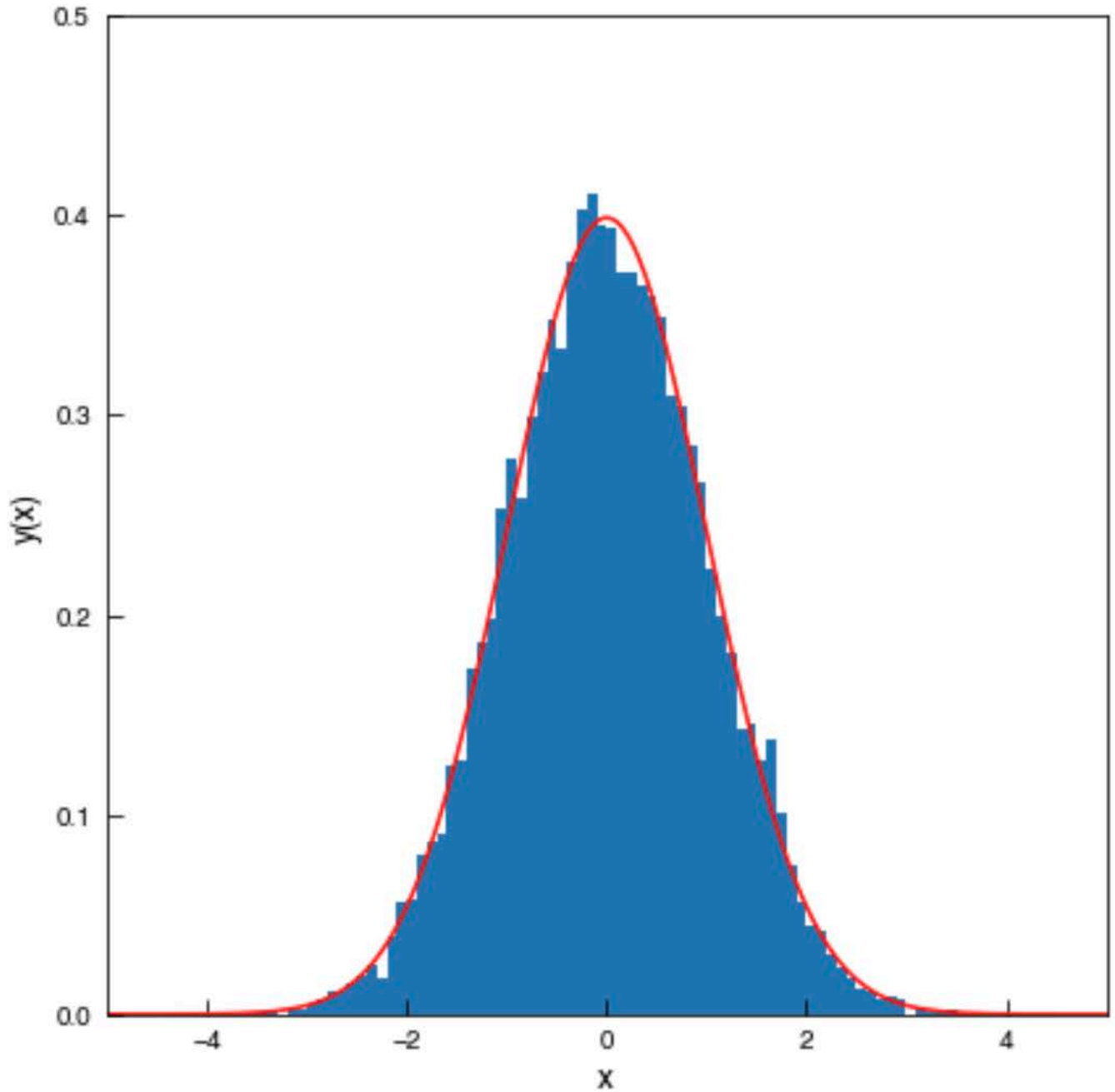
In [19]:
```
def gaussian(x,mu,sigma):
    return 1./(2.0*np.pi*sigma**2)**0.5 * np.exp(-0.5*((x-mu)/sigma)**2)
```

Random Number Generation

Create a histogram of the data

In [24]:
```
fig = plt.figure(figsize=(7,7))
y_hist, x_hist, ignored = plt.hist(x, bins=n_bins, range=[-5,5], density=True)
xx = np.linspace(-5.0,5.0,1000)
plt.plot(xx,gaussian(xx,mu,sigma),color="red")
plt.ylim([0,0.5])
plt.xlim([-5,5])
plt.gca().set_aspect(20)
plt.xlabel('x')
plt.ylabel('y(x)')
plt.show()
```

Random Number Generation



Monte Carlo Integration

One very useful application of random number generation is to perform Monte Carlo integration. Monte Carlo integration uses random numbers to sample a space and compute the integral of a given function by finding the ratio of the region under the curve to the total area probed by the samples.

I wanted to provide a simple example of Monte Carlo integration — the computation of Pi.

Monte Carlo Integration

Perform a simple Monte Carlo integration to compute Pi

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

Set some parameters of the integration

In [18]:
```
n = 10000     #number of samples for the integration
```

Monte Carlo Integration

Make some uniformly sampled variables [-1,1]

In [19]:
```
x = np.random.uniform(-1,1,n)
y = np.random.uniform(-1,1,n)
```

Find the number of samples within the unit circle

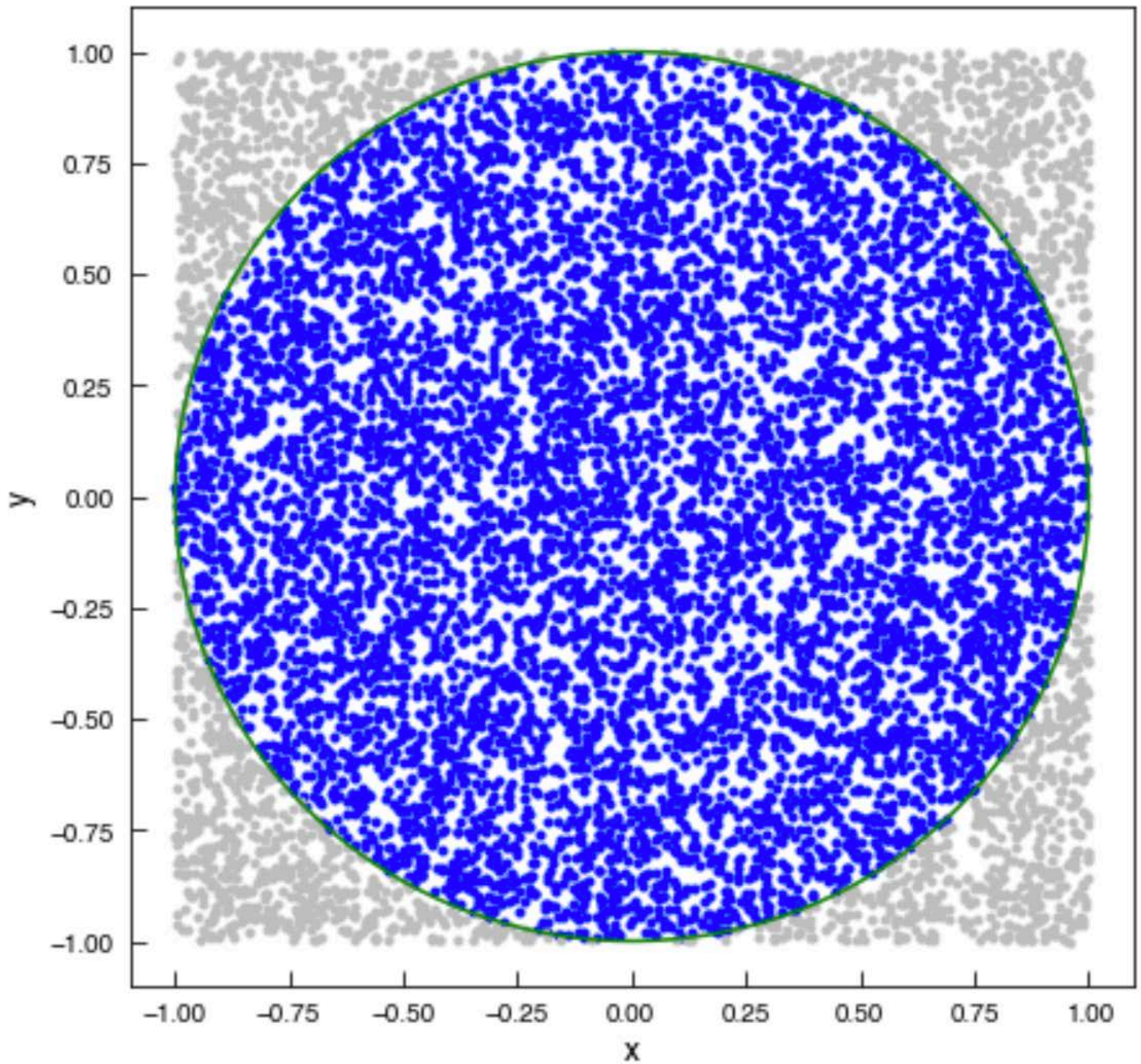
In [20]:
```
ir = np.where((x**2 + y**2)<1.0 )[0]
ur = np.where((x**2 + y**2)>=1.0 )[0]
```

Monte Carlo Integration

Plot the samples and the circle

In [21]:
```
fig = plt.figure(figsize=(7,7))
plt.xlim([-1.1,1.1])
plt.ylim([-1.1,1.1])
plt.plot(x[ir],y[ir],'.',color="blue")
plt.plot(x[ur],y[ur],'.',color="0.75")
theta = np.linspace(0,2*np.pi,1000)
xc = np.cos(theta)
yc = np.sin(theta)
plt.plot(xc,yc,color="green")

plt.xlabel('x')
plt.ylabel('y')
plt.show()
```

Monte Carlo Integration



Monte Carlo Integration

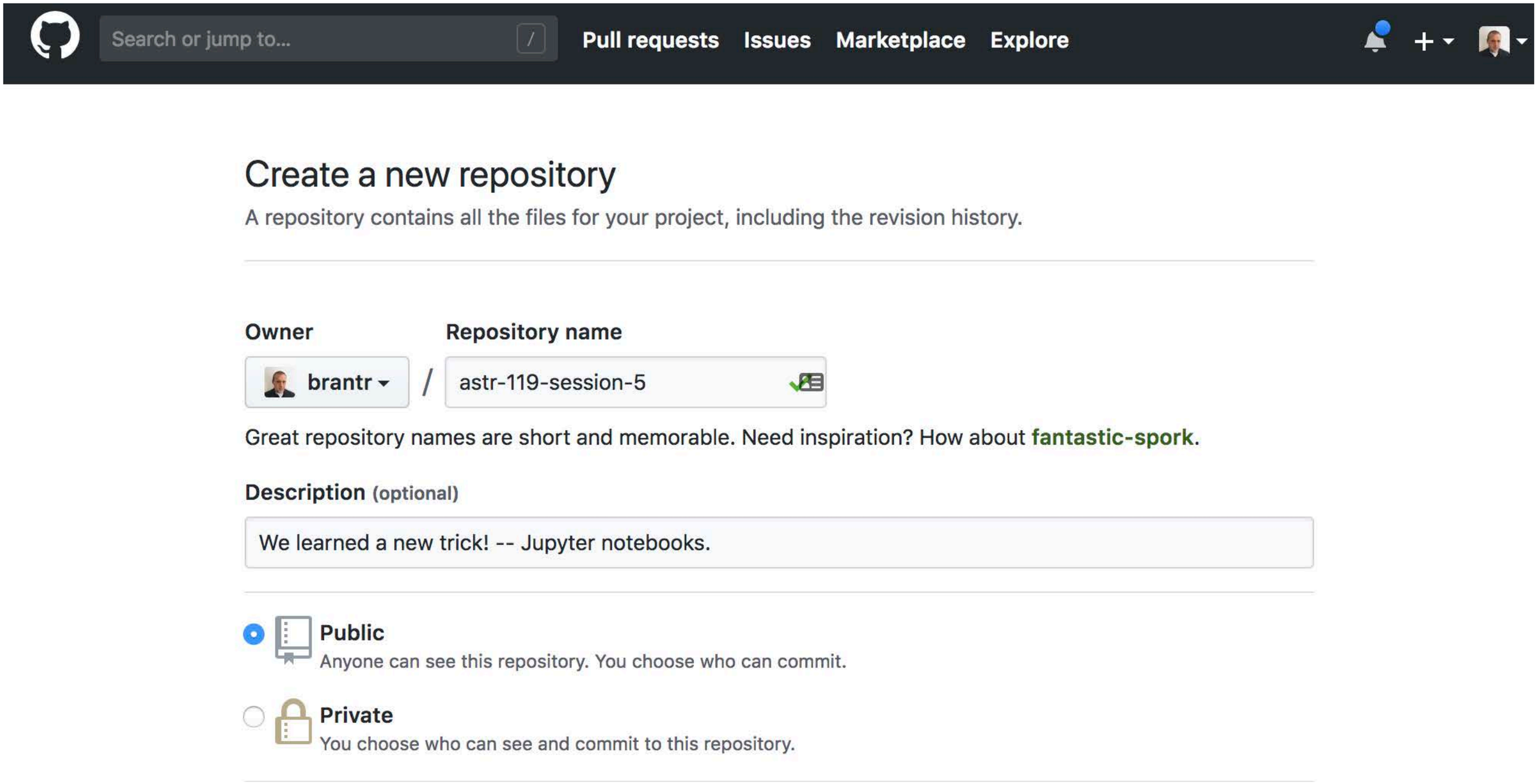
Report the result for Pi

In [24]:
```
pi_approx = 4.0*len(ir)/float(n)

error_pi = (pi_approx-np.pi)/np.pi

print("Number of samples = ",n)
print("Approximate pi    = ",pi_approx)
print("Error in approx   = ",error_pi)
```

```
Number of samples =  10000
Approximate pi    =  3.1476
Error in approx   =  0.0019121977520996127
```

Save Your Work

Make a GitHub project “astr-119-session-17”, and commit the programs you made today.



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

Define a planet class

In [2]:
```
class planet():
    "A planet in our solar system"
    def __init__(self,semimajor,eccentricity):
        self.x   = np.zeros(2) #x and y position
        self.v   = np.zeros(2) #x and y velocity
        self.a_g = np.zeros(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
        self.name   = ""       #name for the planet

```

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

Define some functions for setting circular velocity, and acceleration

In [4]:
```
def SolarCircularVelocity(p):

    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

```

Solar System Model

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

```
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

```
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

```
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

Cell cont.

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

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

        #eccentricity
        p.e = 0.01671123

        #name
        p.name = "Earth"

    #set remaining properties
    p.t = 0.0
    p.x[0] = p.a*(1.0-p.e)
    p.x[1] = 0.0

    #get equiv circular velocity
    v_c = SolarCircularVelocity(p)

    #velocity at perihelion
    v_e = v_c*(1 + p.e)**0.5

    #set velocity
    p.v[0] = 0.0     #no x velocity at perihelion
    p.v[1] = v_e     #y velocity at perihelion (counter clockwise)

    #calculate gravitational acceleration from Sun
    p.a_g = SolarGravitationalAcceleration(p)

    #set timestep
    p.dt = calc_dt(p)

```

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

```
]: 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
```

```
]: 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;
```

```
]: 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.

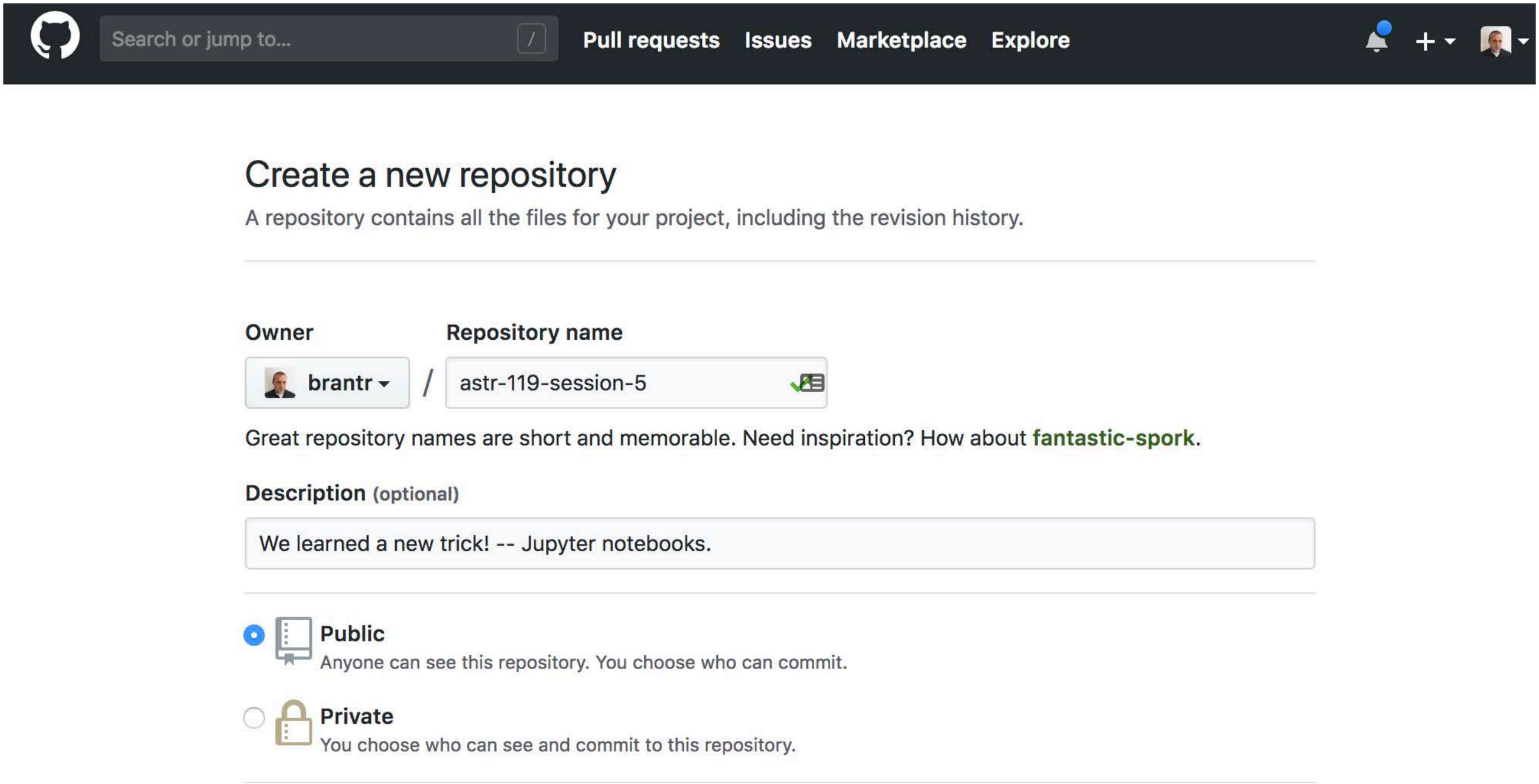
Solar System Model

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()
```

Save Your Work

Make a GitHub project “astr-119-session-16”, and commit the programs my_first_jupyter_notebook.ipynb and test_matplotlib.ipynb you made today.



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 [2]:
```
class planet():
    "A planet in our solar system"
    def __init__(self,semimajor,eccentricity):
        self.x   = np.zeros(2) #x and y position
        self.v   = np.zeros(2) #x and y velocity
        self.a_g = np.zeros(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
        self.name   = ""       #name for the planet

```

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

Define some functions for setting circular velocity, and acceleration

In [4]:
```
def SolarCircularVelocity(p):

    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

```

Solar System Model

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

```
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

```
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

```
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

Cell cont.

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

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

        #eccentricity
        p.e = 0.01671123

        #name
        p.name = "Earth"

    #set remaining properties
    p.t = 0.0
    p.x[0] = p.a*(1.0-p.e)
    p.x[1] = 0.0

    #get equiv circular velocity
    v_c = SolarCircularVelocity(p)

    #velocity at perihelion
    v_e = v_c*(1 + p.e)**0.5

    #set velocity
    p.v[0] = 0.0     #no x velocity at perihelion
    p.v[1] = v_e     #y velocity at perihelion (counter clockwise)

    #calculate gravitational acceleration from Sun
    p.a_g = SolarGravitationalAcceleration(p)

    #set timestep
    p.dt = calc_dt(p)

```

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

```
]: 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
```

```
]: 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;
```

```
]: 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;
```