ASTR 119: Session 13
Orbital Mechanics

Outline

1) No homework this week!
2) Code Check In
3) Visualization of the Day
4) Coupled ordinary Differential Equations
5) Orbital Mechanics
6) 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.

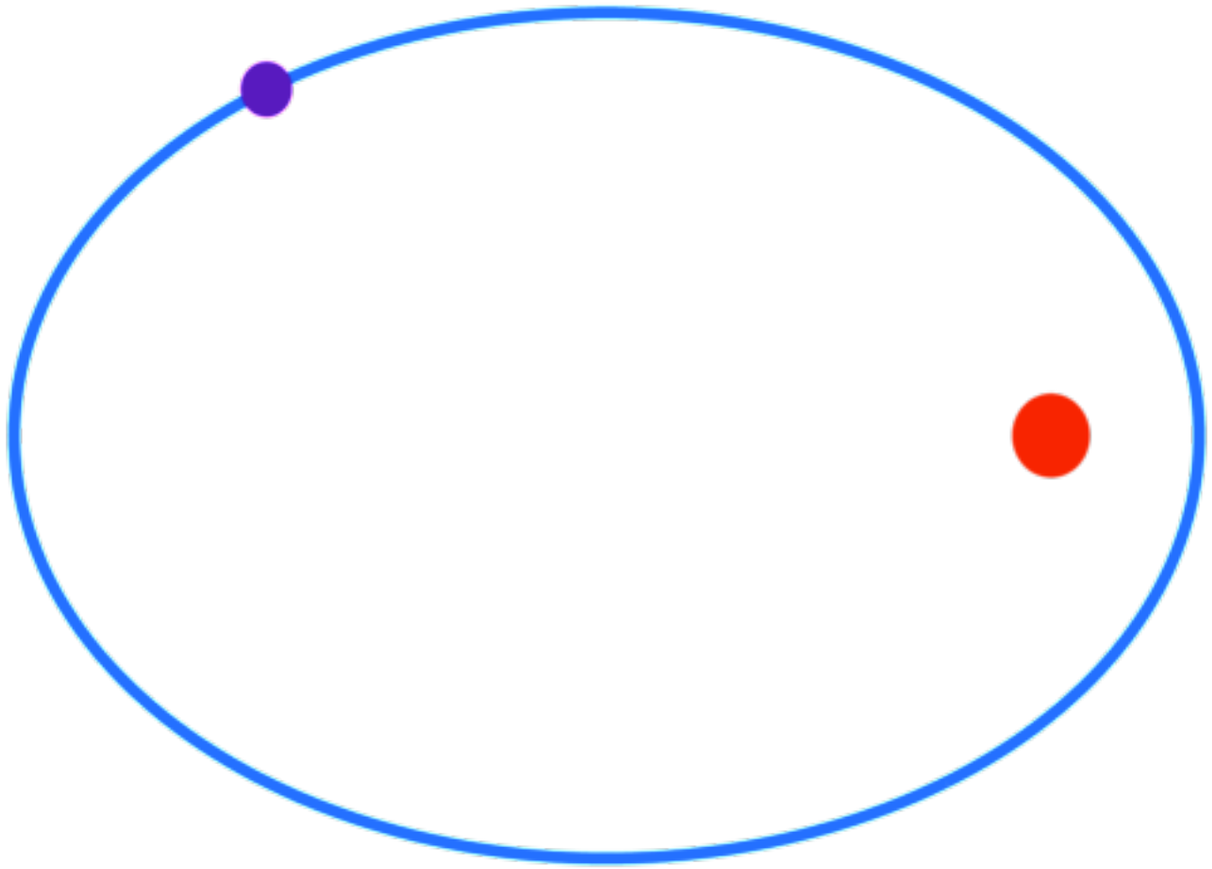
6) Print the final value of x in scientific notation using 2 decimal places.

You will have ~ 5 minutes.

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

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

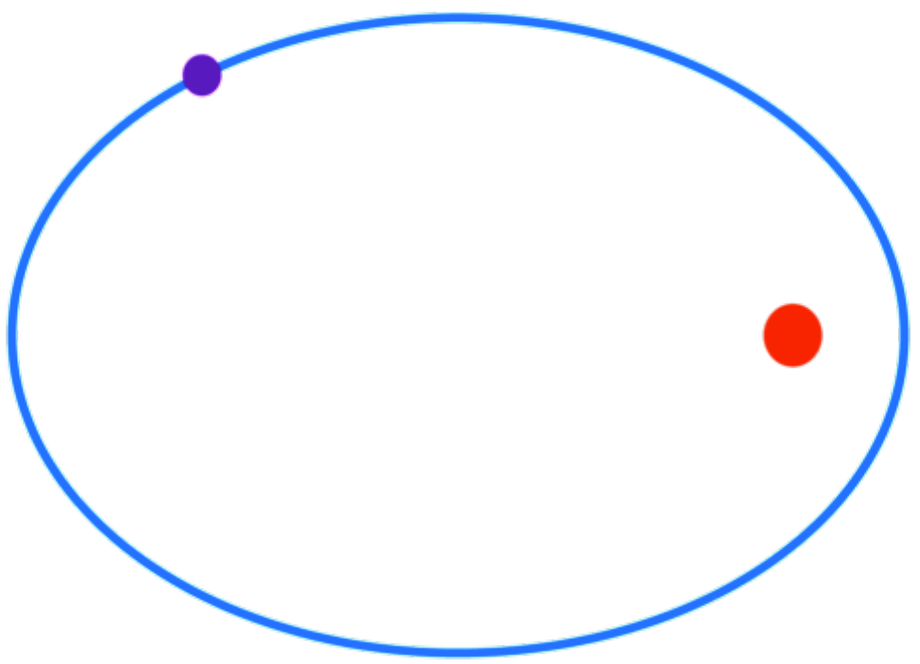
Semi-major axis of orbit *a*:

$2a = r_0 + r_1$

Eccentricity *e*:

Measure of non-circularity of an orbit

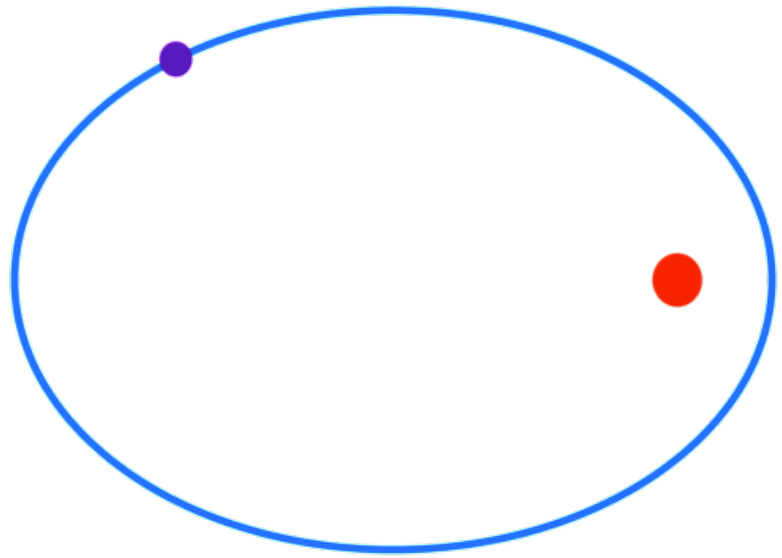
- ***e* < *1*: ellipse**
- ***e* = *0*: circle**
- ***e* = *1*: parabola**
- ***e* > *1*: hyperbola**



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

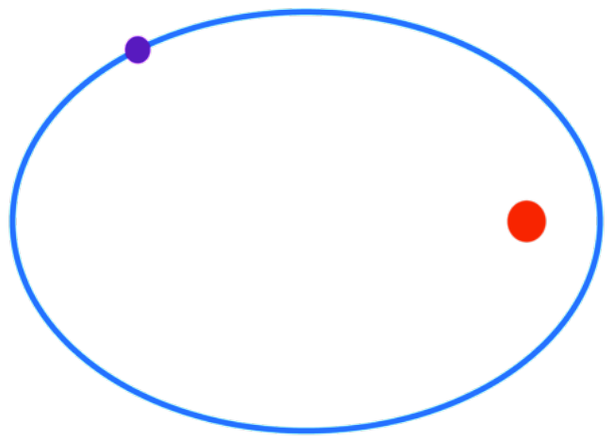
$$1\,\mathrm{solar\,mass} = 1M_\odot = 1.98892 \times 10^{30}\mathrm{kg}$$

Astronomical Unit (Mean distance between Earth and Sun):

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

1 year (Earth orbital time)

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



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

```
        #eccentricity
        p.e = 0.01671123

    #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 = solar_circular_velocity(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
    solar_gravitational_acceleration(p)

    #set timestep
    p.dt = calc_dt(p)

```

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.

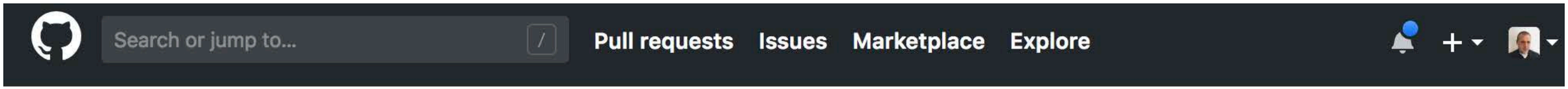
Solar System Model

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.

Save Your Work

Make a GitHub project “astr-119-session-13”, 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.