N-Body Gravitationally Interacting System!

Background!

- We approximate a gravitationally attractive multibody system by integrating the equation of motion
- Three different scenarios are considered:
  - Solar: Central massive body, smaller planets orbiting around
  - Lunar: Moon orbiting a planet as planet orbits a star
  - Random: Particles with a random range of masses in random positions, given velocity according to a Gaussian distribution
- Each scenario is ran for a range of step sizes, after which initial/final momentum & energy are compared

```python
class Particle(object):

    def __init__(self, mass, x, y, z, vx, vy, vz):
        self.mass = mass
        position = np.array([x,y,z])
        velocity = np.array([vx,vy,vz])

        self.position = position
        self.velocity = velocity

        self.position_list = [position]
        self.velocity_list = [velocity]


    def combine_particles(self, particle2, particle_list, crit_radius):

        distance = self.get_distance(particle2)
        r = (distance[0] ** 2.0 + distance[1] ** 2.0 + distance[2] ** 2.0) ** (1.0 / 2.0)

        #print (distance)
        #print (r)

        if r < crit_radius:

            print("Combining Particles")
            # print(particle_list)

            M = self.get_mass() + particle2.get_mass()
            new_v = (self.get_mass() * self.get_velocity_vector() + particle2.get_mass() * particle2.get_velocity_vector()) / M
            new_p = (self.get_position_vector() + particle2.get_position_vector()) / 2.0

            new_particle = Particle(M, new_p[0], new_p[1], new_p[2], new_v[0], new_v[1], new_v[2])
            particle_list.append(new_particle)

            particle_list.remove(self)
            particle_list.remove(particle2)
```

Math!

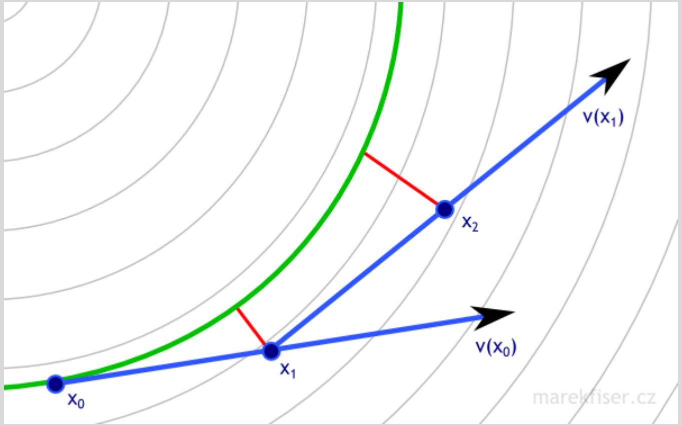
$$U = -G\frac{Mm}{r}$$

$$\vec{a} = \frac{\vec{F}}{m} = -G\frac{M}{r^2}\hat{r}$$

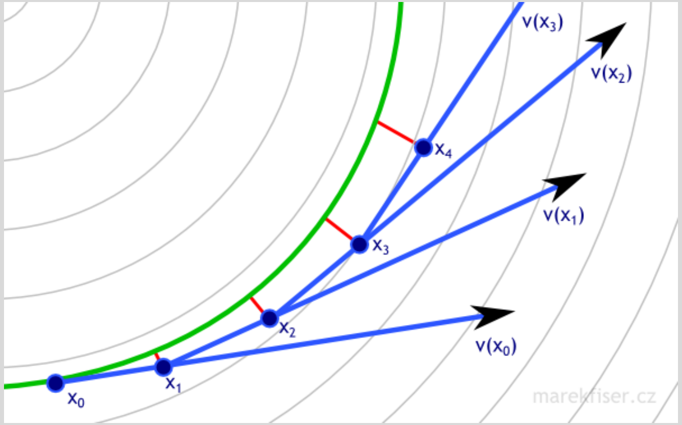
$$\vec{F} = -\nabla U = -G\frac{Mm}{r^2}\hat{r}$$

- Each vector is coded as a numpy array
- Movement is due to the net acceleration felt by all other particles
- Movement determined by integrator (RK4)

Integrator: Runge-Kutta 4th Order!



dt = 0.5



dt = 0.25

- Euler integration two different timesteps
- Vector field shown as grey lines
- Green curve represents actual path
- Blue is the Euler Method (RK1)
- Red lines shows error

Integrator: Runge-Kutta 4th Order!

$$\vec{k}_{1r_{i+1}} = \vec{v}_i \qquad \vec{k}_{1v_{i+1}} = \vec{a}(\vec{r}_i)$$

$$\vec{k}_{2r_{i+1}} = \vec{v}_i + \vec{k}_{1v_{i+1}}\frac{h}{2} \qquad \vec{k}_{2v_{i+1}} = \vec{a}(\vec{r}_i + \vec{k}_{1r_{i+1}}\frac{h}{2})$$

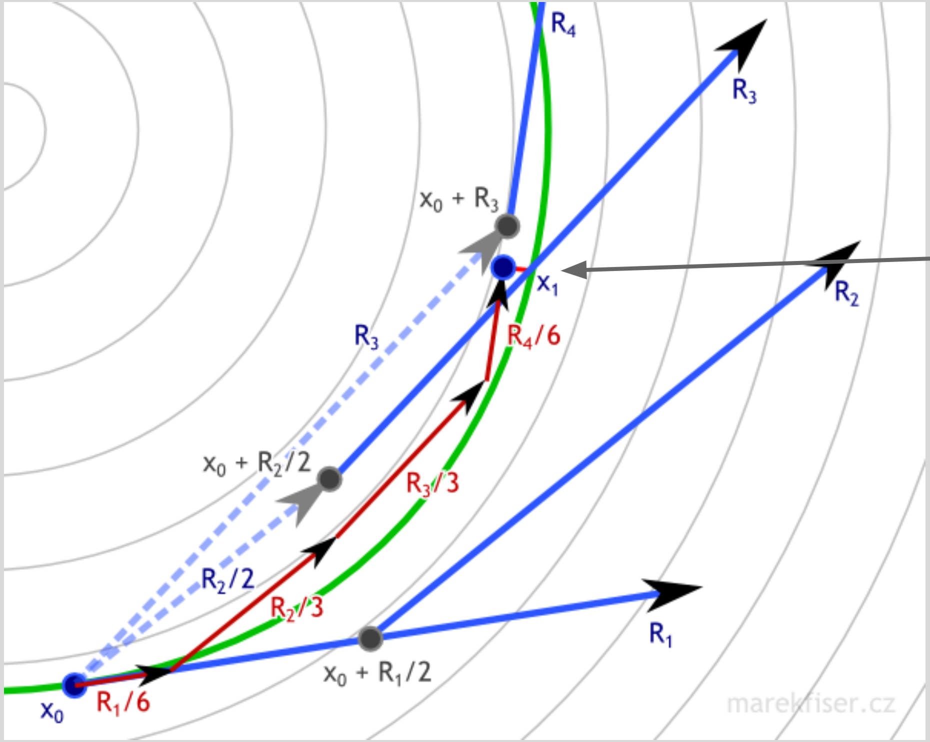
$$\vec{k}_{3r_{i+1}} = \vec{v}_i + \vec{k}_{2v_{i+1}}\frac{h}{2} \qquad \vec{k}_{3v_{i+1}} = \vec{a}(\vec{r}_i + \vec{k}_{2r_{i+1}}\frac{h}{2})$$

$$\vec{k}_{4r_{i+1}} = \vec{v}_i + \vec{k}_{3v_{i+1}}h \qquad \vec{k}_{4v_{i+1}} = \vec{a}(\vec{r}_i + \vec{k}_{3r_{i+1}}h)$$

$$\vec{r}_{i+1} = \vec{r}_i + \frac{h}{6}(\vec{k}_{1r_{i+1}} + 2\vec{k}_{2r_{i+1}} + 2\vec{k}_{3r_{i+1}} + \vec{k}_{4r_{i+1}})$$

$$\vec{v}_{i+1} = \vec{v}_i + \frac{h}{6}(\vec{k}_{1v_{i+1}} + 2\vec{k}_{2v_{i+1}} + 2\vec{k}_{3v_{i+1}} + \vec{k}_{4v_{i+1}})$$

Integrator: Runge-Kutta 4th Order!



RK4 (dt = 1)

$$\vec{r}_{i+1} = \vec{r}_i + \frac{h}{6}(\vec{k}_{1r_{i+1}} + 2\vec{k}_{2r_{i+1}} + 2\vec{k}_{3r_{i+1}} + \vec{k}_{4r_{i+1}})$$
$$\vec{v}_{i+1} = \vec{v}_i + \frac{h}{6}(\vec{k}_{1v_{i+1}} + 2\vec{k}_{2v_{i+1}} + 2\vec{k}_{3v_{i+1}} + \vec{k}_{4v_{i+1}})$$

Error in measurement: little red line

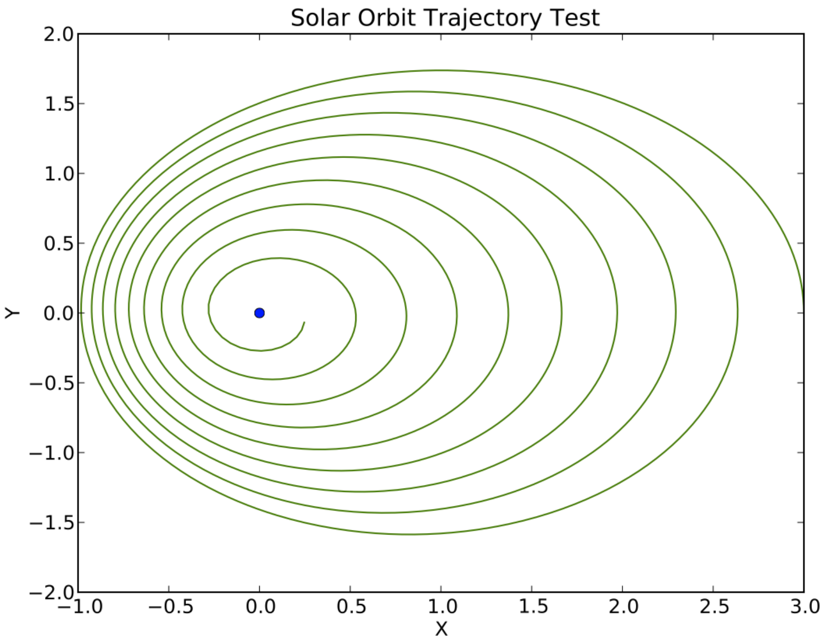
These coefficients are the slope of the function at three separate points during the timestep: the beginning, the mid-point and the end.

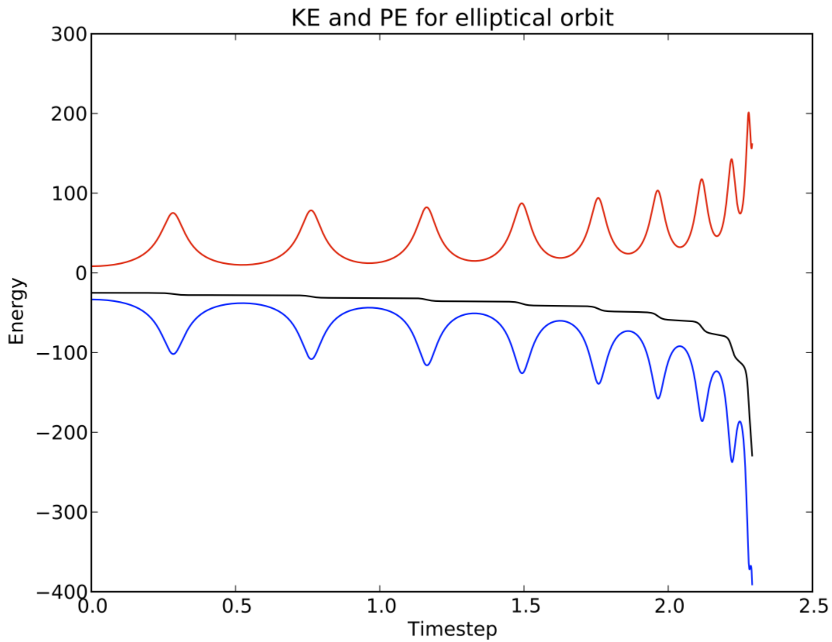
Acceleration Calculation: The Work Horse!

```python
def acceleration(self, position1):
    """Accepts particle's position; returns net acceleration"""
    ax, ay, az = 0, 0, 0
    for p2 in particles:
        if p2 != self:
            r = distance(self,p2)
            a = G*p2.mass / r**2.0
            a_x = a * (p2.position[0] - position1[0])/r
            a_y = a * (p2.position[1] - position1[1])/r
            a_z = a * (p2.position[2] - position1[2])/r
            ax += a_x
            ay += a_y
            az += a_z
    return (ax, ay, az)
```

Effects of Stepsize!

A Small Timestep is the Enemy of Energy Conservation

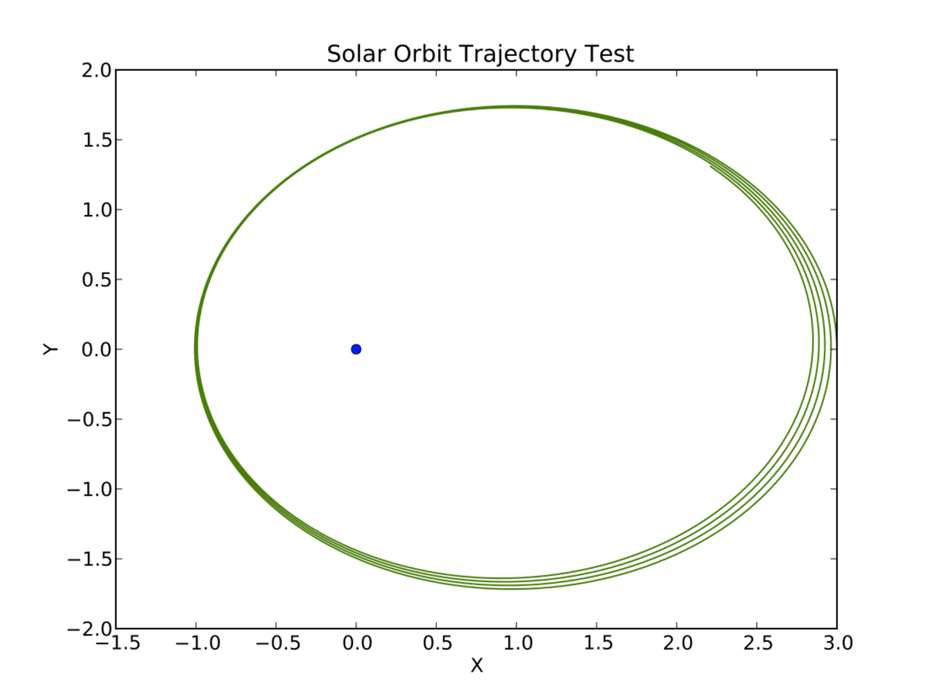


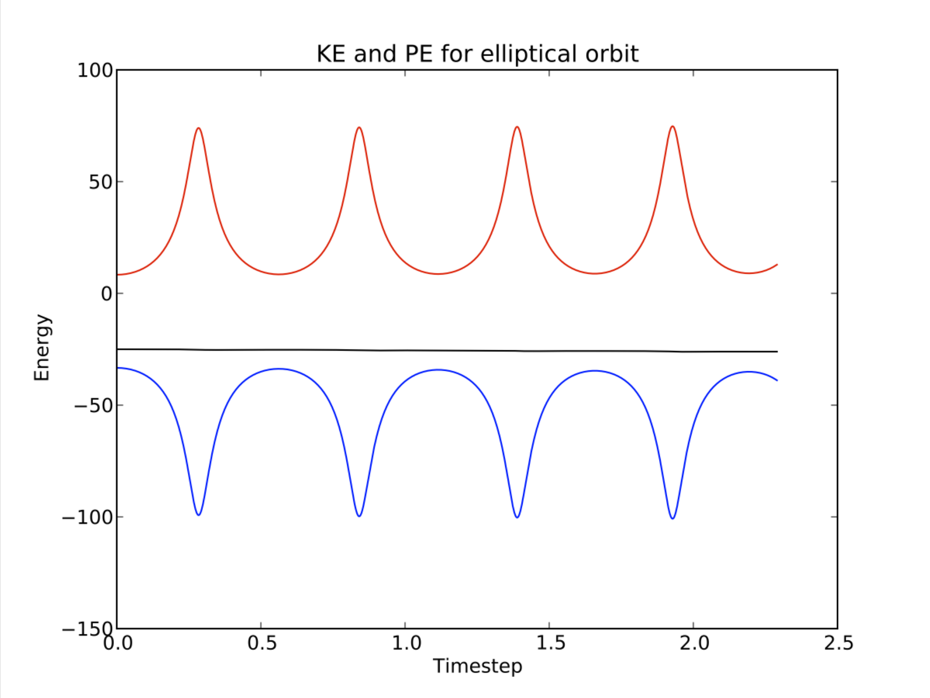


Solar Orbit Examples

h = 0.001

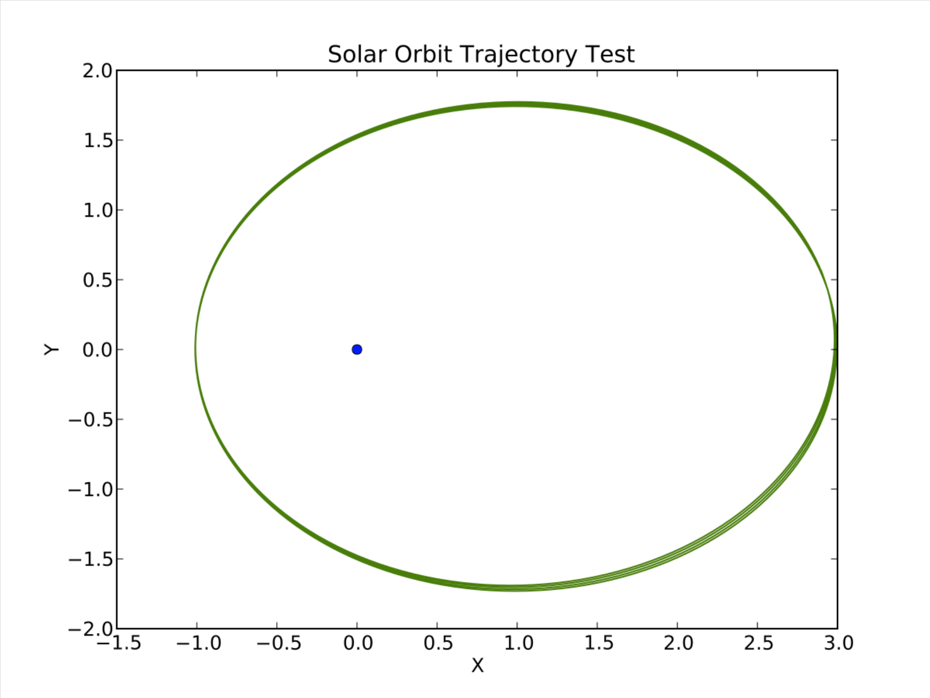
Effects of Stepsize!

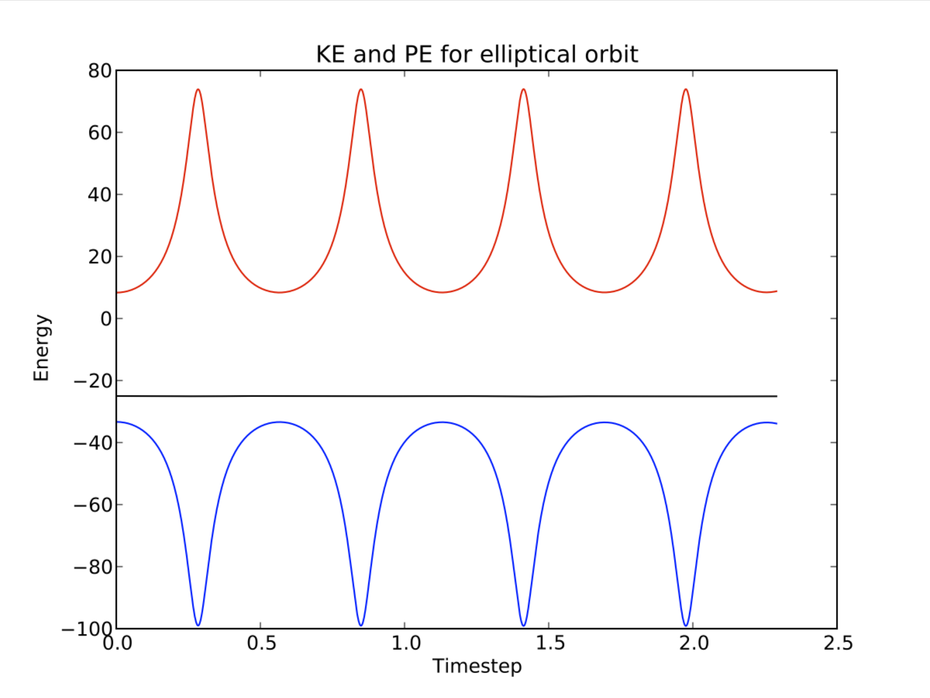




h = 0.0001

Effects of Stepsize!

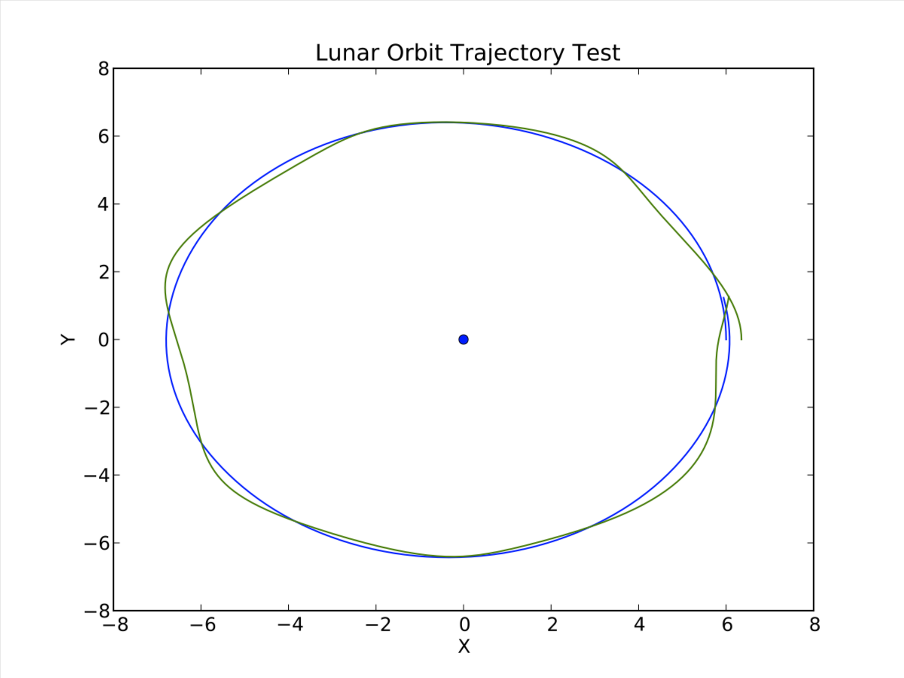


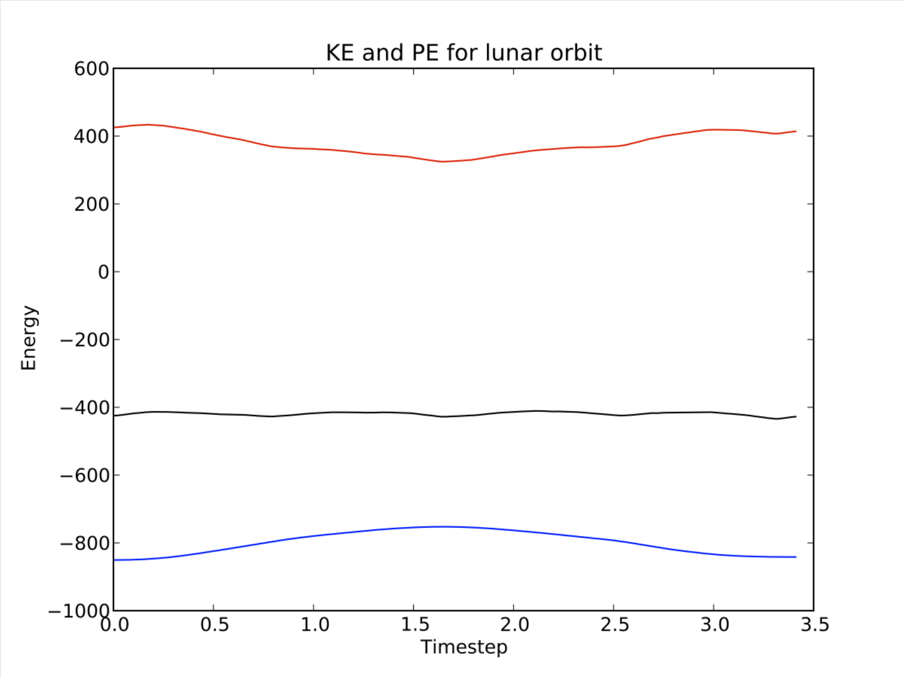


h = 0.00001

Lunar Orbit Example

h = 0.00001

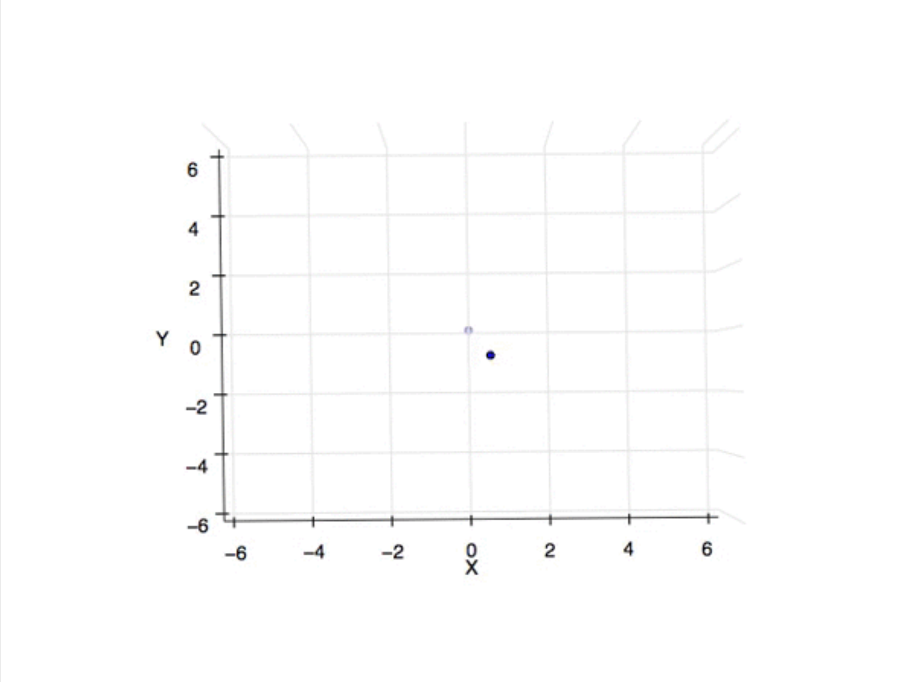


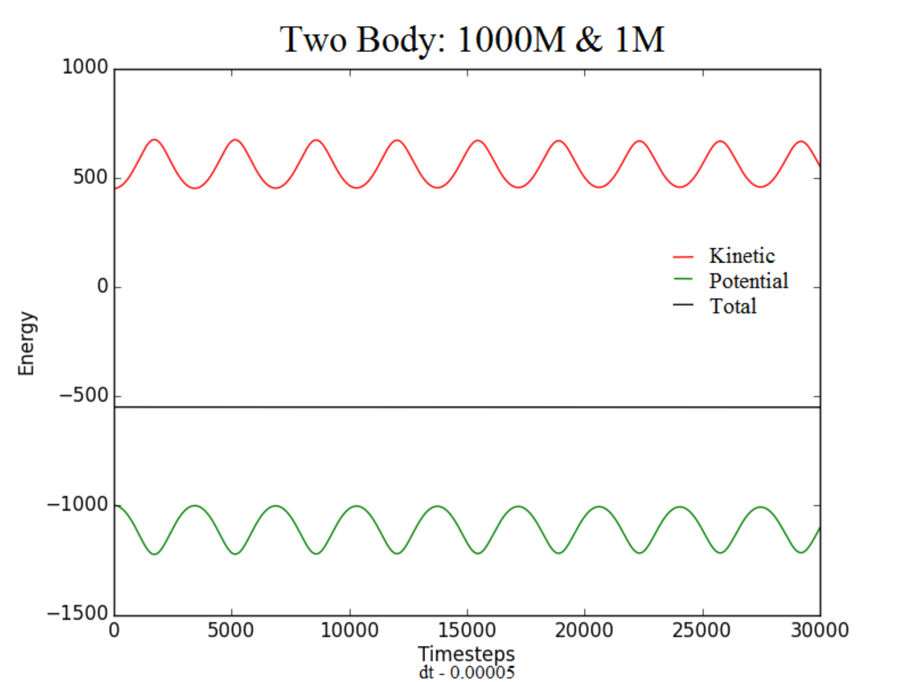


The lunar orbit highlights the problem with close interactions when dealing with discrete timesteps.

Animation!

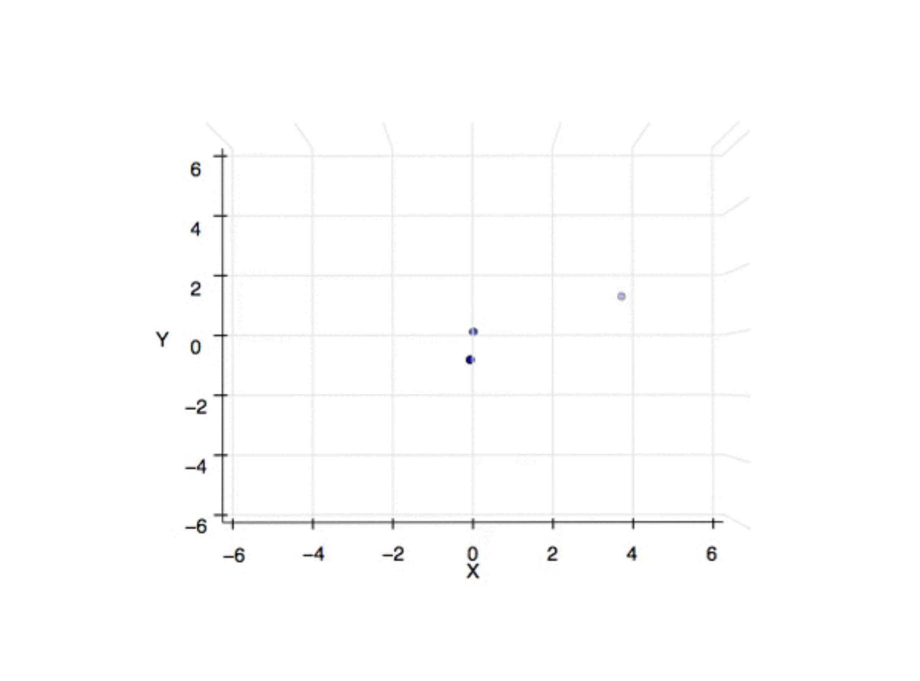
Two Body System

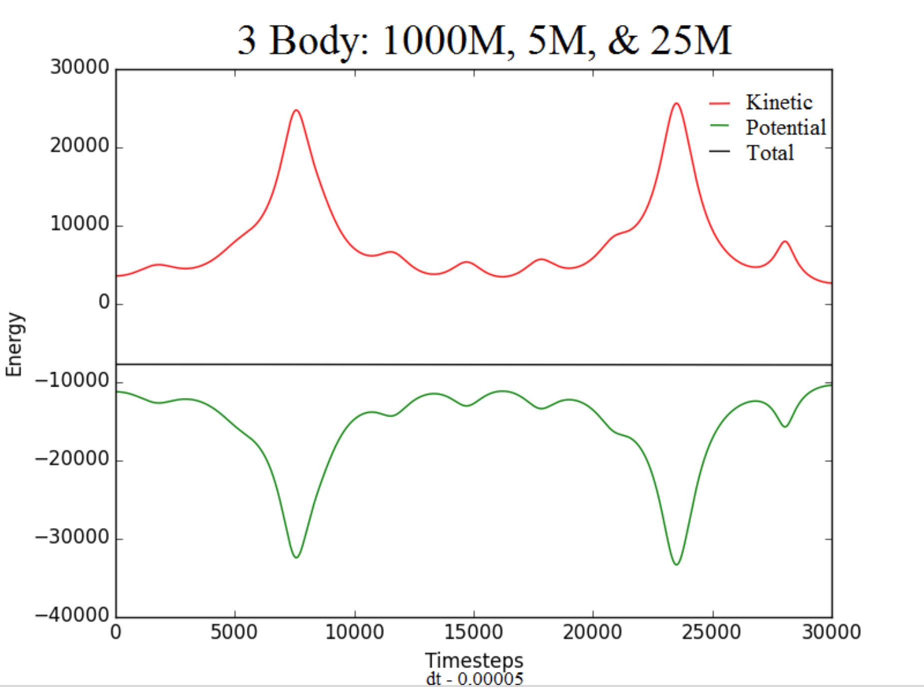




Animation!

Three Body System

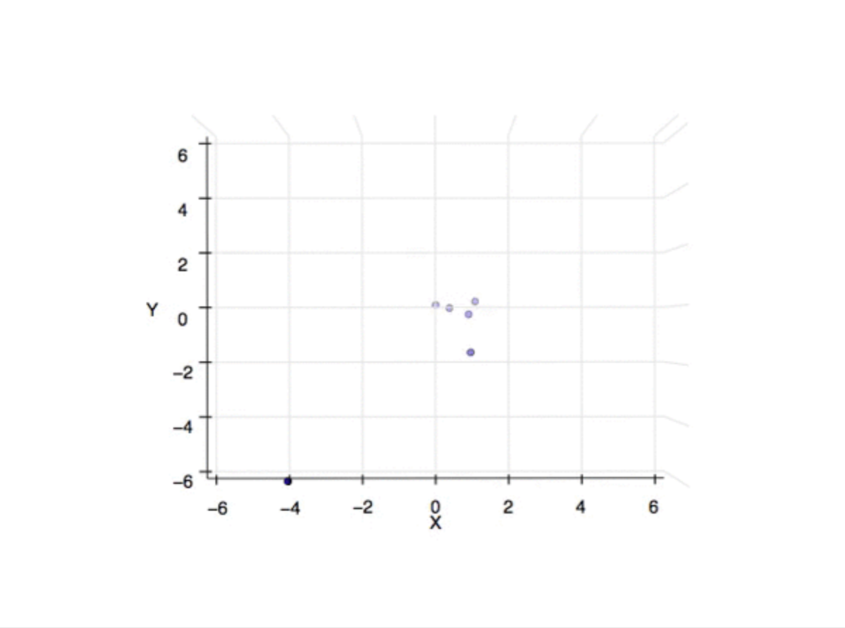


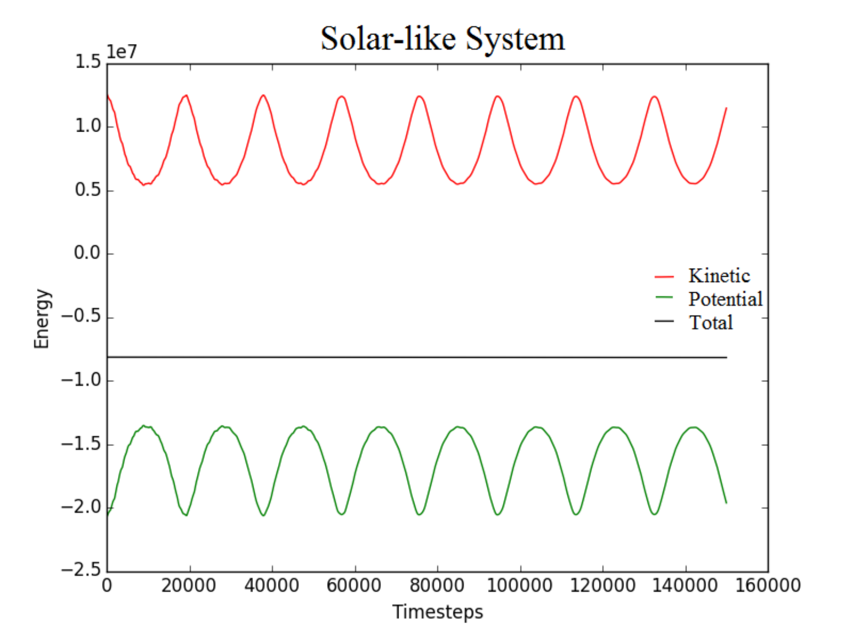


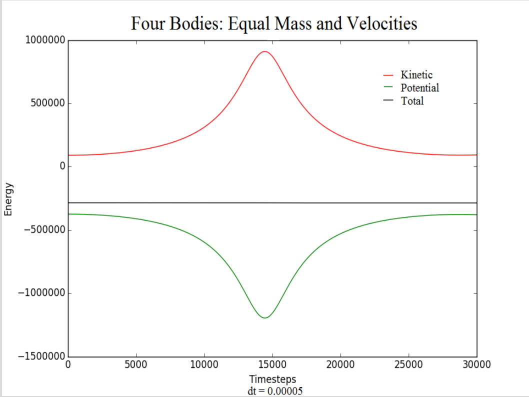
Animation!

- Similar masses to inner planets plus Jupiter
- Jupiter dominates

Solar-like System
(Mercury to Jupiter)

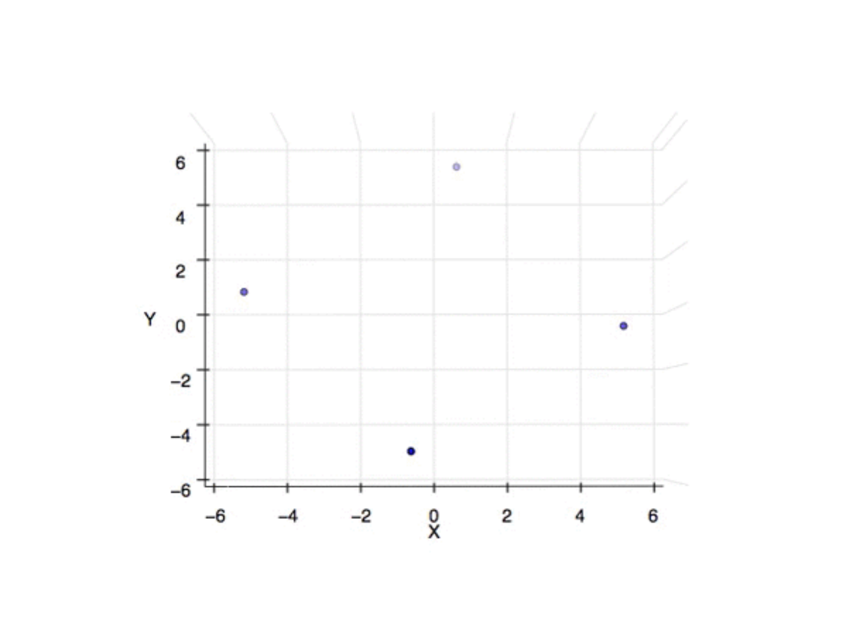


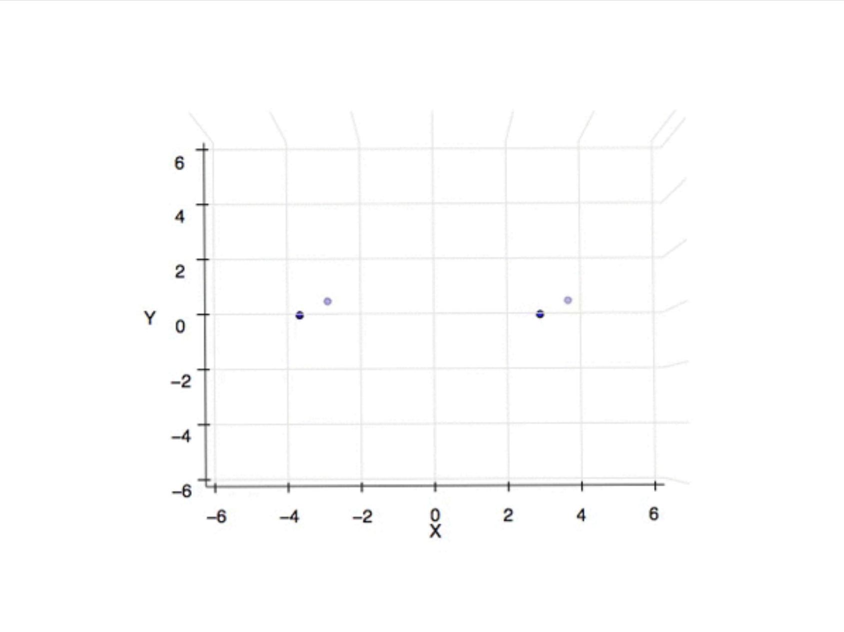


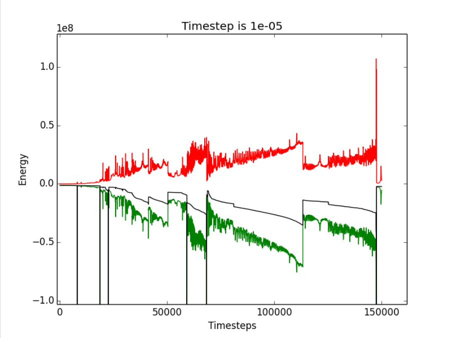


Animation!

Four Equal Mass System

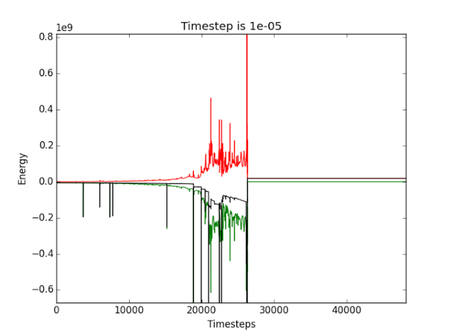


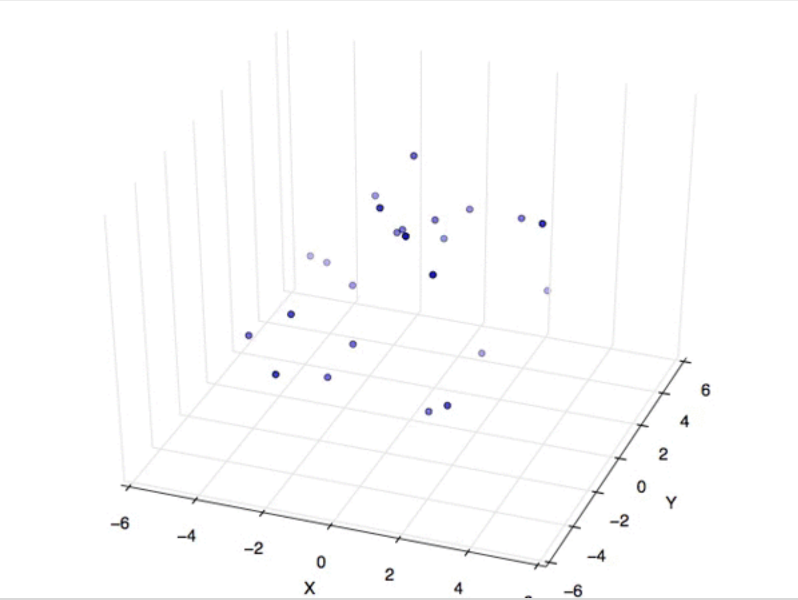




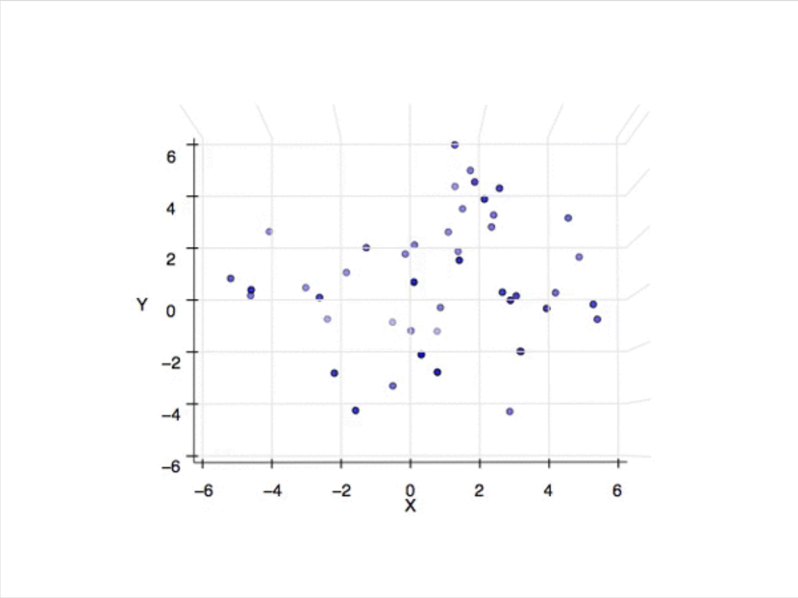
Animation!

N-body Collisional System



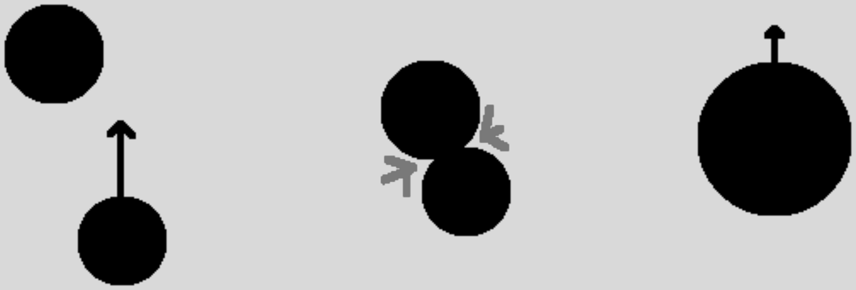


N = 25



N = 50

Energy Conservation!



Sources of Error

- Every collision produces some amount of error (inelastic collision)
- High velocities demand high timestep
    - Dynamical timestep

```
h = h_min * (top_velocity/current_max)
```

Room for Improvement!

- Import FORTRAN for workhorse calculations
- Don’t double record acceleration
- Initialize particles with a spin property that adds during collisions