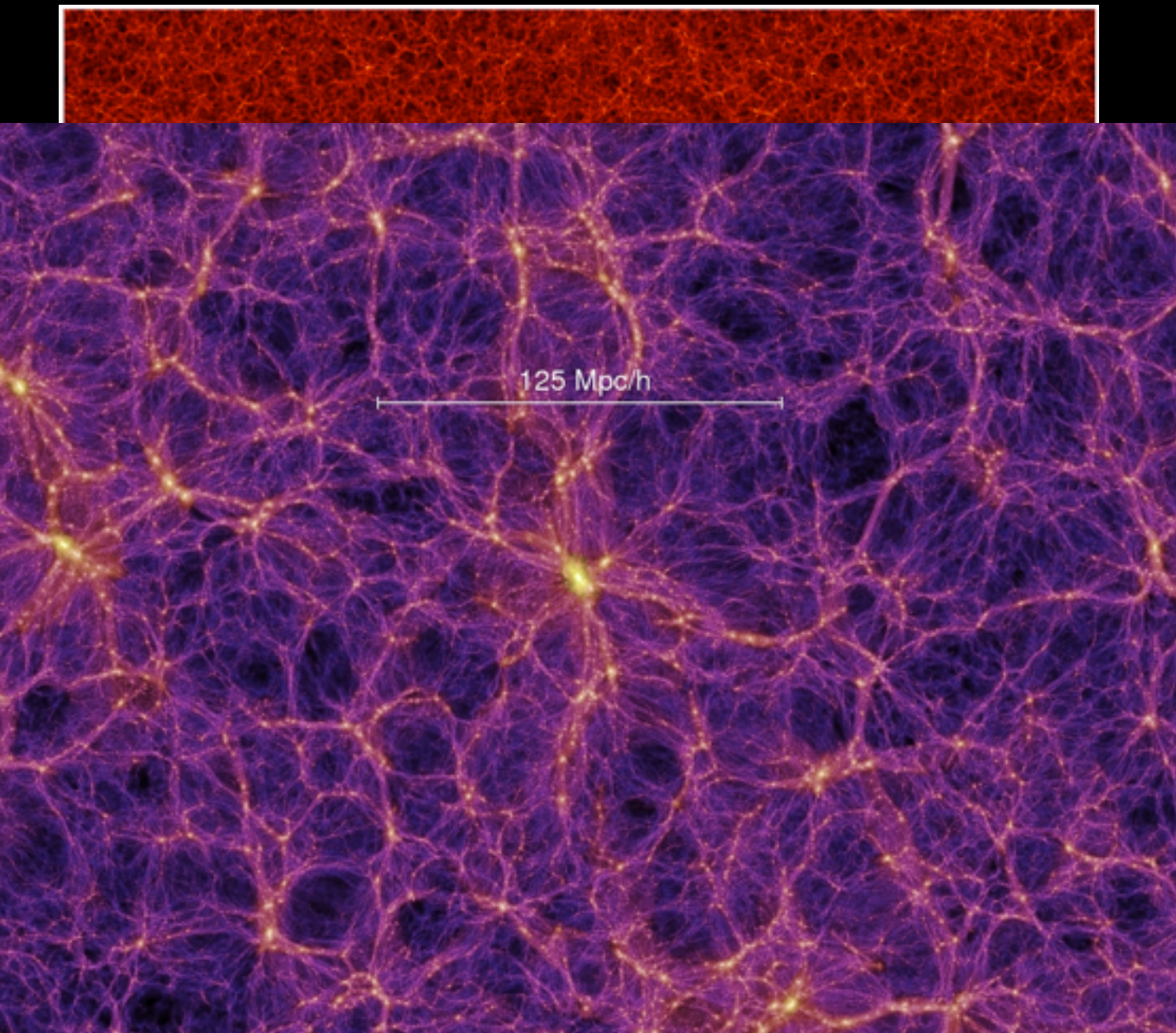# Computational Physics
Lecture 8

sieversj@ukzn.ac.za

git clone https://github.com/ukzncompphys/lecture8_2016.git

N-Body

- Dominant force in the universe on large scales is gravity.

- Physical systems often too complex to deal with analytically. Computer simulations often key to understanding.

- Wide variety of problems in e.g. astrophysics involve matter fields evolving with gravity.

- Evolution of 2 masses is called the "2-body problem." With many (many) objects, called the "n-body problem," or just n-body.

- N-body simulations are key to understanding the universe around us. Also useful in chemistry, economics…
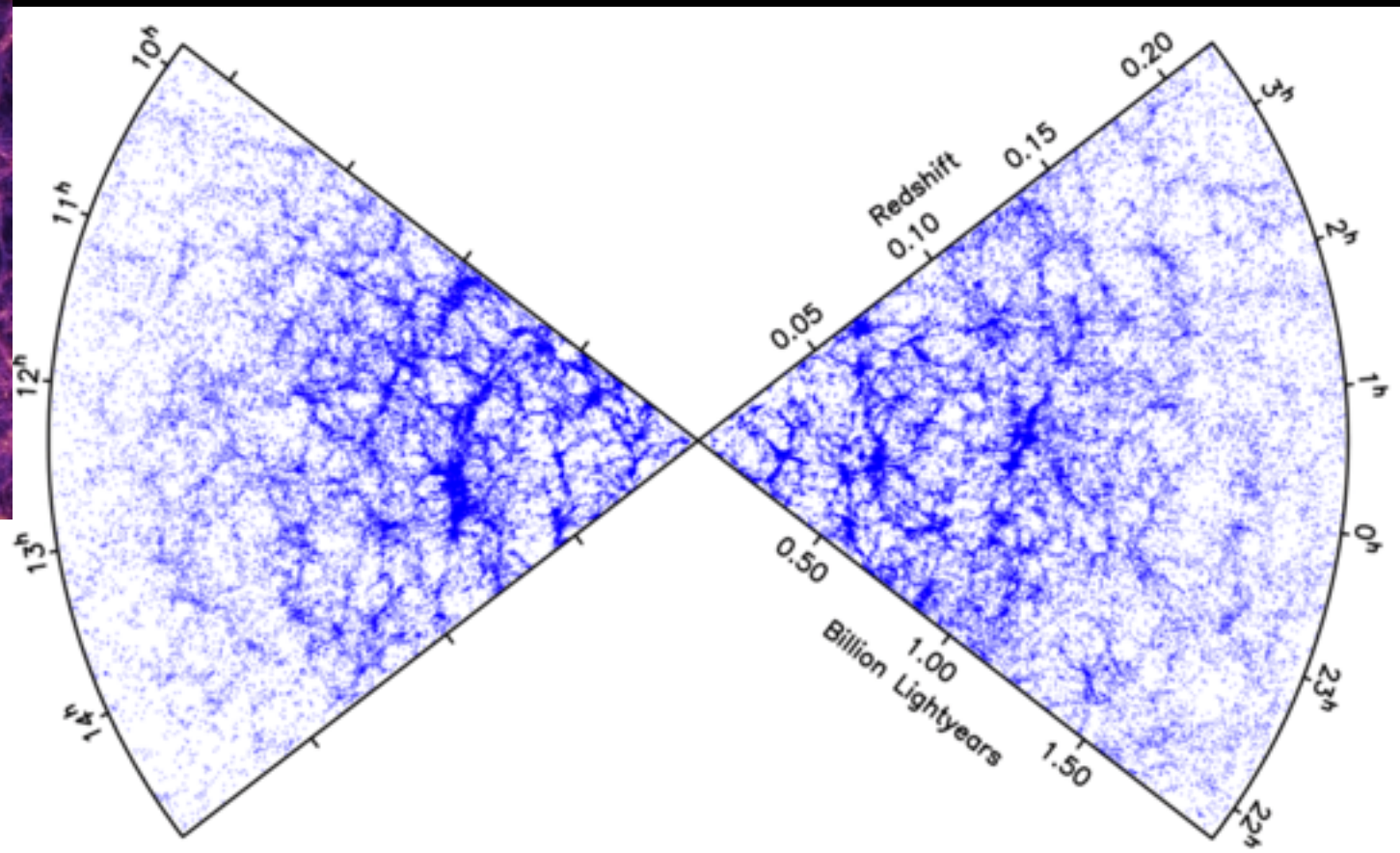
Cosmology

125 Mpc/h

MacFarland, Colberg, White (München), Jenkins,
Pearce, Frenk (Durham), Evrard (Michigan),
Couchman (London, CA), Thomas (Sussex),
Efstathiou (Cambridge), Peacock (Edinburgh)

$2000 \times 2000 \times 20\,(\text{Mpc}/h)^3$

Simulation (left) dark matter,
bottom is galaxy data.

We use simulations like this to
interpret observations of galaxy
clustering.

Galaxy Clusters

- Galaxy clusters are biggest objects in universe - 10^{15} solar masses.

- Picture from millenium simulation, total of 10^{10} particles.

- Need simulations to interpret galaxy cluster data.

2 Mpc/h
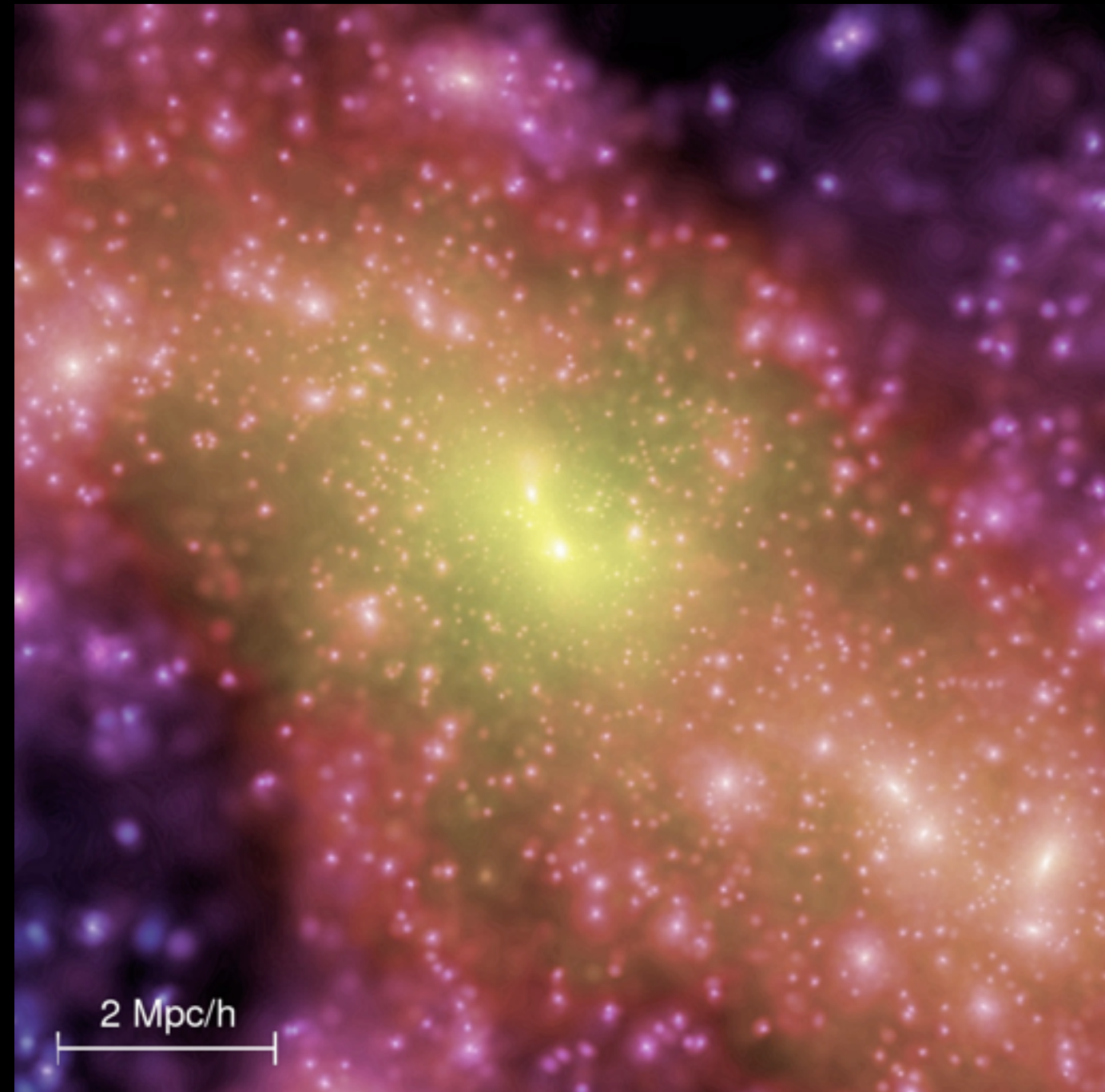
Solar System

- more than 2 bodies usually unstable, systems kick out lightest objects.

- Is the solar system stable? Could Earth get kicked out of its orbit and become inhospitable to life?
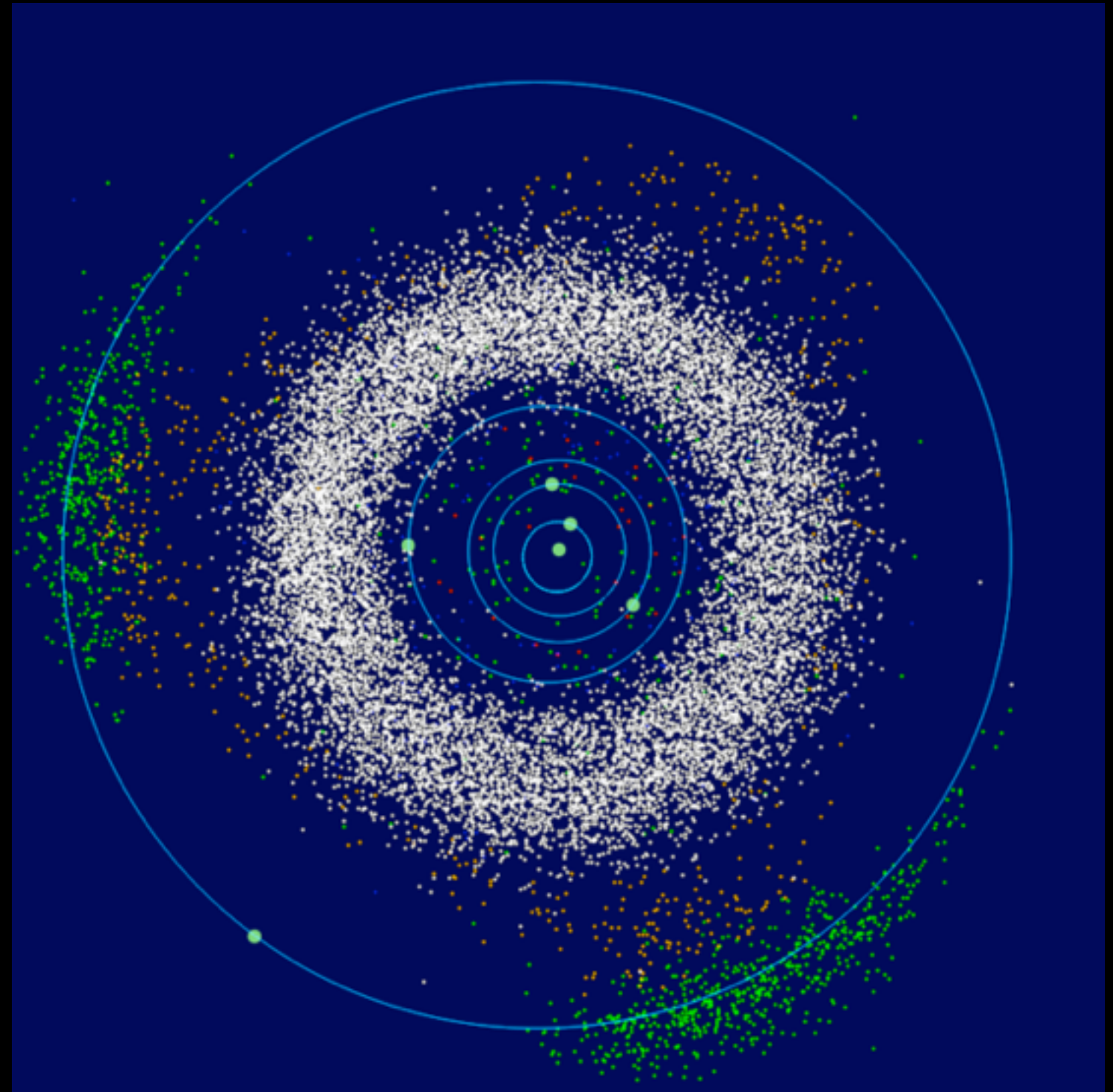
Classical n-body

- We'll approximate system as a collection of masses, interact only through gravity.

- What is the minimum information we need per particle?

Classical n-body

- We'll approximate system as a collection of masses, interact only through gravity.

- What is the minimum information we need per particle?

- Each particle needs its own mass, position, and velocity.

Gravity

- F=-Gm_1m_2/r^2. F=ma. For many particles, dv/dt=-$\sum Gm_2/r_{12}^2$.

- dx/dt=v. Definition of velocity.

- Leaves us with coupled system:
  $d/dt[x_i, v_i \quad] \quad = [v, -\sum Gm_j/r_{ij}^2.]$

- Solve system of equations, and we're done!

How do we solve?

- if $dx/dt=v$, and $x_t=v_0$, then $x_{t+\delta} \approx x_0 + \delta_t v$ from some "average" value of v.

- So, take discrete steps in time. Then take each particle, and use its velocity to update positions

- Also have to update velocities using accelerations: $dv_i/dt = -\sum Gm_j/r_{ij}^2$ for $i \neq j$. Note the force is a vector, so we rewrite $1/r^2$ as $r/|r|^3$

- For sufficiently small time step, we should have an accurate solution.

Let's look at two particles: two_particles_simple.py

```python
import numpy
from matplotlib import pyplot as plt
#let's start two partcles in what should be a circular orbit
x0=0;y0=0;vx0=0;vy0=0.5
x1=1;y1=0;vx1=0;vy1=-0.5;

#for simplicity, let's assume G&m are all equal to 1
dt=0.01
tmax=5
```

Code to do two bodies.
First, set initial conditions,
integration time, and step
size.

```python
for t in numpy.arange(0,tmax,dt):
    dx=x0-x1
    dy=y0-y1
    rsquare=dx*dx+dy*dy
    r=numpy.sqrt(rsquare)
    r3=r*r*r
    #calculate the x and y  force components
    fx0=dx/r3;
    fy0=dy/r3
    #forces on particle 1 must be opposite of particle 0
    fx1 =-fx0
    fy1 =-fy0
    #update particle positions
    x0 +=dt*vx0
    y0 +=dt*vy0
    vx0 +=-dt*fx0
    vy0 +=-dt*fy0

    x1+= dt*vx1
    y1+=dt*vy1
    vx1 -= dt*fx1
    vy1 -= dt*fy1
```

```python
plt.clf()
plt.plot(x0,y0,'rx')
plt.plot(x1,y1,'b*')
plt.ylim(-1.5,1.5)
plt.xlim(-1,2)

plt.draw()
pot=-1.0/r
kin=0.5*(vx0*vx0+vy0*vy0+vx1*vx1+vy1*vy1)
print 'kin and pot are '  + repr(kin) + '    ' + repr(pot) + '   ' + repr(pot+kin)
```

Let's look at two particles

```python
import numpy
from matplotlib import pyplot as plt
#let's start two partcles in what should be a circular orbit
x0=0;y0=0;vx0=0;vy0=0.5
x1=1;y1=0;vx1=0;vy1=-0.5;

#for simplicity, let's assume G&m are all equal to 1
dt=0.01
tmax=5
```

Now let's loop over time steps. First calculate force with r/r^3, then update positions and velocities.

```python
for t in numpy.arange(0,tmax,dt):
    dx=x0-x1
    dy=y0-y1
    rsquare=dx*dx+dy*dy
    r=numpy.sqrt(rsquare)
    r3=r*r*r
    #calculate the x and y  force components
    fx0=dx/r3;
    fy0=dy/r3
    #forces on particle 1 must be opposite of particle 0
    fx1 =-fx0
    fy1 =-fy0
    #update particle positions
    x0 +=dt*vx0
    y0 +=dt*vy0
    vx0 +=-dt*fx0
    vy0 +=-dt*fy0

    x1+= dt*vx1
    y1+=dt*vy1
    vx1 -= dt*fx1
    vy1 -= dt*fy1
```

```python
plt.clf()
plt.plot(x0,y0,'rx')
plt.plot(x1,y1,'b*')
plt.ylim(-1.5,1.5)
plt.xlim(-1,2)

plt.draw()
pot=-1.0/r
kin=0.5*(vx0*vx0+vy0*vy0+vx1*vx1+vy1*vy1)
print 'kin and pot are '  + repr(kin) + '    ' + repr(pot) + '   ' + repr(pot+kin)
```

Let's look at two particles

```python
import numpy
from matplotlib import pyplot as plt
#let's start two partcles in what should be a circular orbit
x0=0;y0=0;vx0=0;vy0=0.5
x1=1;y1=0;vx1=0;vy1=-0.5;

#for simplicity, let's assume G&m are all equal to 1
dt=0.01
tmax=5
```

```python
for t in numpy.arange(0,tmax,dt):
    dx=x0-x1
    dy=y0-y1
    rsquare=dx*dx+dy*dy
    r=numpy.sqrt(rsquare)
    r3=r*r*r
    #calculate the x and y  force components
    fx0=dx/r3;
    fy0=dy/r3
    #forces on particle 1 must be opposite of particle 0
    fx1 =-fx0
    fy1 =-fy0
    #update particle positions
    x0 +=dt*vx0
    y0 +=dt*vy0
    vx0 +=-dt*fx0
    vy0 +=-dt*fy0

    x1+= dt*vx1
    y1+=dt*vy1
    vx1 -= dt*fx1
    vy1 -= dt*fy1
```

Finally, plot particle positions, calculate the kinetic and potential energy, and print the energies to the screen.

```python
plt.clf()
plt.plot(x0,y0,'rx')
plt.plot(x1,y1,'b*')
plt.ylim(-1.5,1.5)
plt.xlim(-1,2)

plt.draw()
pot=-1.0/r
kin=0.5*(vx0*vx0+vy0*vy0+vx1*vx1+vy1*vy1)
print 'kin and pot are '  + repr(kin) + '   ' + repr(pot) + '   ' + repr(pot+kin)
```

Higher Order

- The force changes during a timestep. We will build up inaccuracy due to ignoring this.

- We could be more accurate - what if we take a trial step, then calculate the force there and replace the effective force by the average of the two forces?

- Likewise, we can get a trial final velocity, why not use the average of the initial and final?

- This will give us higher accuracy

Leapfrog

- Another simple scheme for higher order is *leapfrog*.

- If I say my positions and velocities are half a step out of sync then the velocity is the average velocity over the position timestep.

- Likewise, position is the average position over the next velocity timestep

- Get 2nd order for no extra work!

- Downside - can't change timestep size.

Softening

- How big should a timestep be?

- Depends on forces. If force is big enough that velocity changes by a lot, then method will be inaccurate.

- for $f=1/r^2$, force can get arbitrarily big. Bad!

- Solution is to use *softening* - particles are really fuzzy balls, so once they get close enough, force *drops*.

- Possible solutions: $F \sim r/((r^2+eps^2)^{3/2}$. Or, $a=r^3$, if $a<a_0$, $a=a_0$. $F=r/a$.

- Then for large distances, force is unchanged, but goes to zero for small distances.

Many Particles

- We can do the same thing for many particles. Particles should collapse together into a ball.

- Softening is key!

- Let's watch:

How Much Work Does This Take?

- Right now, we calculate the forces on every pair of particles. Total work scales like n^2 - running big simulations is a problem!

- Instead, let's look at potential from a distribution of many particles. If the potential of 1 particle is P(r), then what is the potential from a field?

- $P_{tot}(r') = \int P(r-r')\rho(r)d^3r$.

- Wait, have we seen this operation before?

Modern N-body

- We have! Global potential is just potential from a single particle convolved with the density field.

- Force is just the gradient of the potential

- FFT takes $n\log n$, for billions of particles, $n\log n <<< n^2$.

- So, a scheme is to have a grid on which we will calculate the density, then sum particles into their nearest grid cell, and convolve with the desired potential.

- Once have potential, for each particle, calculate gradient of potential at its position. Now we can run for millions of particles on a desktop instead of thousands!

Tutorial

- Let's add some methods to the class from the last tutorial so we can use it in an n-body simulation. First, add a method to initialize # of particles witih random positions in 2-D (numpy.random.randn() will get gaussian random numbers). (5)

- Next, write a method that calculates the forces on the particles using a softened potential (10)

- Now write a method that will update the particle positions and velocities using a timestep. (5)

- Finally, plot the total kinetic and potential energies as a function of time, and show that the total energy is approximately conserved (5)

Tutorial bonus

- Write a 2nd order integrator for your n-body class. You'll need to make copies of your particles. When you look at the total energy, is conservation better or worse? (10)

Final Project

- One possible final project is to write an *nlogn* nbody simulator (unless you have another idea). If you think you would like to do this, you can start working on it.

- Core routines will be to calculate the 2-d density field from particle positions, to calculate the global potential from the distribution, and to calculate the particle forces from the potential field.

- See project_guidelines.pdf

Another Possible Project

- Ever notice how old-timey movie reels sound very different?

- Much of this is due to response of old microphones, which had very limited frequency ranges.

- With sounddevice, you should be able to record your voice into python.

- Use FFTs to filter your voice (with a fake British accent!) to sound old-timey.

- Write your original and filtered voices into a file you can play.

1-page Music Player

- Python module sounddevice will play sounds (Linux you can just write to /dev/audio, but sounddevice has other nice features)

- Open-source faad library will read a variety of music files

- Combination of the two can be used to write a 1-page player that reads all your music tracks, shuffles them, then plays the while printing out info on what you're playing.

- ctrl-c interrupts current play and moves on to next song.

- See shuffle.py