

Computational Physics

Lecture 7

sieversj@ukzn.ac.za

git clone https://github.com/ukzncompphys/lecture7_2018.git

First Tutorial

```
import numpy as np
from matplotlib import pyplot as plt
```

```
def simpsons(fun,x0,x1,n):
    #first, let's check we got a valid input for n.
    if not(type(n)==int):
        print 'number of points must be an integer'
        return None
    if n&1==0:
        print 'number of points must be odd in simpsons'
        return None
    x=np.linspace(x0,x1,n)
    dx=x[1]-x[0]
    y=fun(x) #this calls the function we passed in at the x points
    even_sum=y[2:-1:2].sum() #make sure we don't pick up last point
    odd_sum=y[1::2].sum()
    tot=y[0]+y[-1]+4*odd_sum+2*even_sum
    return dx*tot/3
```

```
def simple_int(fun,x0,x1,n):
    x=np.linspace(x0,x1,n)
    dx=x[1]-x[0]
    y=fun(x)
    return dx*y.sum()
```

```
n=[11,31,101,301,1001]
```

```
nvec=np.asarray(n)
errvec=np.zeros(len(n))
errvec_simple=np.zeros(len(n))
for i in range(len(n)):
    err=simpsons(np.cos,0,np.pi/2,n[i])-1
    print 'Simpsons error on ',n[i],' points is ',err
    errvec[i]=np.abs(err)
    errvec_simple[i]=simple_int(np.cos,0,np.pi/2,n[i])-1
```

```
plt.ion()
plt.clf();
plt.plot(nvec,errvec)
ax=plt.gca()
ax.loglog() #here's another way to specify log axes
```

```
pp=np.polyfit(np.log(nvec),np.log(errvec),1)
print 'Error goes like number of point to the power ',pp[0]
pred=np.polyval(pp,np.log(nvec))
plt.plot(nvec,np.exp(pred))
```

```
plt.plot(nvec,errvec_simple)
pp_simple=np.polyfit(np.log(nvec),np.log(errvec_simple),1)
print 'Error for simple integrator goes like number of point to the power ',pp_simple[0]
```

```
plt.legend(['Simpsons Error','Power Law Fit','Simple Integrator Error'])
```

Dictionaries

- Dictionaries very useful built-in datatype in python
- Dictionaries have *keys*, each key stores a *value*.
- You can create a dictionary with {}
- Use square brackets to get the values

Dictionaries in Action

```
>>> x={} #initialize an empty dictionary
>>> x[0]=4
>>> x[-1]='sandwich'
>>> x['quest']='to find the holy grail'
>>> print x['quest']
to find the holy grail
>>> y=0
>>> print x[y]
4
>>> y=-1
>>> print x[y]
sandwich
>>> print x
{0: 4, 'quest': 'to find the holy grail', -1: 'sandwich'}
>>> for key in x.keys():
...     print 'key ' + repr(key) + ' has value ' + repr(x[key])
...
key 0 has value 4
key 'quest' has value 'to find the holy grail'
key -1 has value 'sandwich'
>>>
```

See how to assign and reference dictionary entries.

`d.keys()` gets all keys
you can use “del” to delete a dictionary entry (or any variable for that matter)

```
>>> sandwich={'bread' : 'rye','cheese' : 'swiss', 'meat' : 'pastrami','toppings': ('mustard','mayo')}
>>> for key in sandwich.keys():
...     print 'key ' + repr(key) + ' has value ' + repr(sandwich[key])
...
key 'cheese' has value 'swiss'
key 'toppings' has value ('mustard', 'mayo')
key 'meat' has value 'pastrami'
key 'bread' has value 'rye'
>>>
```

```
>>> del sandwich['bread']
>>> print sandwich.keys()
['cheese', 'toppings', 'meat']
>>>
```

Make a dictionary containing cubes from 1 to 10

```
Jonathans-MacBook-Pro:lecture5 sievers$ more cube_dict.py
#file cube_dict.py
cubes={}
for x in range(1,11):
    cubes[x]=x**3

for xx in cubes.keys():
    print repr(xx) + ' cubed is ' + repr(cubes[xx])
Jonathans-MacBook-Pro:lecture5 sievers$ █
```

```
>>> execfile('cube_dict.py')
1 cubed is 1
2 cubed is 8
3 cubed is 27
4 cubed is 64
5 cubed is 125
6 cubed is 216
7 cubed is 343
8 cubed is 512
9 cubed is 729
10 cubed is 1000
>>> █
```

Classes

- Python is *object-oriented*. That means things called objects contain data and contain methods (i.e. functions) that do things with the data.
- You have seen this in action: e.g. `vec=numpy.ones(10);print vec.sum()`
- This is very different from e.g. C or (classic) Fortran. In C, you need to know how to sum an array. In Python, the array knows how to sum itself
- In python, objects are called *classes*. You can define them in files with the *class* keyword.
- data/methods of a class are accessed with a period ‘.’. The first argument to any method is the instance of the class itself. It is customary (and strongly encouraged) to name that variable “self”.

Beginnings of a complex variable class

```
#class_example1.py
import numpy
class Complex:
    #__init__ is a special function. When you create a new
    #instance of a class, if it exists in the class definition,
    #__init__ will get called. __init__ is assumed to return the first value

    def __init__(self,r=0,i=0):
        self.r=r
        self.i=i

if __name__=='__main__':
    num=Complex()
    print 'real part of num is ' + repr(num.r)
    print 'imaginary part of num is ' + repr(num.i)

    num2=Complex(2,5)
    print 'real part of num2 is ' + repr(num2.r)
    print 'imaginary part of num2 is ' + repr(num2.i)

    #we can assign new data to classes whenever we want.
    #you probably want to be really careful with this however
    num2.len=numpy.sqrt(num2.r**2+num2.i**2)
    print 'length of num2 is ' + repr(num2.len)
```

Left: a bare-bones
complex number class.

Below: output

```
-uu-:---F1 class_example1.py All L1 (Python)--
Loading python...done
```

```
Jonathans-MacBook-Pro:lecture5 sievers$ python class_example1.py
real part of num is 0
imaginary part of num is 0
real part of num2 is 2
imaginary part of num2 is 5
length of num2 is 5.3851648071345037
Jonathans-MacBook-Pro:lecture5 sievers$
```

Class Methods

- We've made a class that can hold complex numbers.
- Right now the class just holds numbers, it doesn't do anything.
- We did take an absolute value, but we had to know at the command line how to do that.
- Let's add a method to the class to take its absolute value

Methods ctd.

```
#class_example2.py
import numpy
class Complex:
    #__init__ is a special function. When you create a new
    #instance of a class, if it exists in the class definition,
    #__init__ will get called. __init__ is assumed to return the first value

    def __init__(self,r=0,i=0):
        self.r=r
        self.i=i
    def abs(self):
        return numpy.sqrt(self.r**2+self.i**2)

if __name__=='__main__':

    num=Complex(2,5)
    print 'real part of num is ' + repr(num.r)
    print 'imaginary part of num is ' + repr(num.i)
    myabs=num.abs()
    print 'absolute value is ' + repr(myabs)
```

We have added an `abs()` method to the complex class. Now you can get the absolute value without having to know anything about complex numbers.

```
Jonathans-MacBook-Pro:lecture5 sievers$ python class_example2.py
real part of num is 2
imaginary part of num is 5
absolute value is 5.3851648071345037
Jonathans-MacBook-Pro:lecture5 sievers$
```

What's the difference?

```
#class_example3.py

import numpy
class Complex:
    def __init__(self, r=0, i=0):
        self.r=r
        self.i=i
    def abs(self):
        return numpy.sqrt(self.r**2+self.i**2)
#####
#
# What is the difference between these two classes?
#
#####
class Complex2:
    def __init__(self, r=0, i=0):
        self.r=r
        self.i=i
def abs(self):
    return numpy.sqrt(self.r**2+self.i**2)

if __name__=='__main__':

    num=Complex(2,5)
    print num.abs()
    num2=Complex2(2,5)
    print num2.abs()
```

Classes Complex and Complex2 look similar, but they might have different behaviour. Why?

```
Jonathans-MacBook-Pro:lecture5 sievers$ python class_example3.py
5.38516480713
Traceback (most recent call last):
  File "class_example3.py", line 28, in <module>
    print num2.abs()
AttributeError: Complex2 instance has no attribute 'abs'
Jonathans-MacBook-Pro:lecture5 sievers$
```

What's the difference?

```
#class_example3.py
```

```
import numpy
class Complex:
    def __init__(self, r=0, i=0):
        self.r=r
        self.i=i
    def abs(self):
        return numpy.sqrt(self.r**2+self.i**2)
#####
#
# What is the difference between these two classes?
#
```

```
#####
class Complex2:
    def __init__(self, r=0
        self.r=r
        self.i=i
def abs(self):
    return numpy.sqrt(sel

if __name__=='__main__':
    num=Complex(2,5)
    print num.abs()
    num2=Complex2(2,5)
    print num2.abs()

>>> abs(-3)
3
>>> execfile('class_example3.py')
5.38516480713
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    File "class_example3.py", line 28, in <module>
        print num2.abs()
AttributeError: Complex2 instance has no attribute 'abs'
>>> abs(num2)
5.3851648071345037
>>> abs(num)
5.3851648071345037
>>> abs(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    File "class_example3.py", line 20, in abs
        return numpy.sqrt(self.r**2+self.i**2)
AttributeError: 'int' object has no attribute 'r'
>>>
```

Always remember your indenting! By not indenting we closed the Complex2 definition *and* defined a global function abs that replaced the built-in function.

Python Uses References

- Python uses *references*. If *a* is an instance of a class, and you say *b=a*, then the contents of *b* will point to the same memory as the contents of *a*.
- This means that if I then change *b*, *a* will also change.
- General rule is if you change/assign a piece of *b*, same piece of *a* will change.
- Be very careful - don't change values inside of functions unless you meant to.

```
>>> a=Complex(3,5)
>>> b=a
>>> print a.r
3
>>> b.r=5
>>> print a.r
5
>>> █
```

Copy

- Because of this, it is often customary to have a `copy()` function.
- Copy should make a fully distinct version of the instance.

```
#class_example4.py

import numpy
class Complex:
    def __init__(self, r=0, i=0):
        self.r=r
        self.i=i
    def copy(self):
        return Complex(self.r, self.i)
    def abs(self):
        return numpy.sqrt(self.r**2+self.i**2)

if __name__=='__main__':

    num=Complex(2,5)
    num2=num.copy()
    num2.r=10
    print 'real part of num is ' + repr(num.r)
    print 'real part of num2 is ' + repr(num2.r)
```

```
Jonathans-MacBook-Pro:lecture5 sievers$ python class_example4.py
real part of num is 2
real part of num2 is 10
Jonathans-MacBook-Pro:lecture5 sievers$
```


Overloading

- The operators in python (e.g. +,-,*...) just map to a set of special functions. You can use them on your classes if you include methods with those names.
- Extending the behaviour of the default operators is called overloading.
- `__add__` is the keyword for '+'. `__repr__` is the keyword for printing things.
- If you want to do this, you can google to get the rest of the special function names.
- Note that `a+b` is shorthand for `a.__add__(b)` so as written `a+2` will work, but `2+a` won't. Why?

```
#overload.py

import numpy
class Complex:
    def __init__(self,r=0,i=0):
        self.r=r
        self.i=i
    def copy(self):
        return Complex(self.r,self.i)
    def __add__(self,val):
        ans=self.copy()
        if isinstance(val,Complex):
            ans.r=ans.r+val.r
            ans.i=ans.i+val.i
        else:
            ans.r=ans.r+val
        return ans
    def __repr__(self):
        if (self.i<0):
            return repr(self.r)+' - '+repr(-1*self.i) +'i'
        else:
            return repr(self.r)+' + '+repr(self.i) +'i'
```

```
>>> from overload import Complex
>>> a=Complex(2,5)
>>> b=Complex(4,-3)
>>> c=a+b
>>> print c
6 + 2i
>>> d=a+b+2
>>> print d
8 + 2i
>>>
```

Try/Except

- Sometimes things go wrong. Say a method is given invalid input
- Python has *try/except*. The code will execute the *try* block. As soon as that hits an error it jumps to the *except* block.
- If there is no error, *except* is skipped.
- Optionally, you can include a *finally* clause that always gets executed after the *try/except*. Useful for e.g. freeing memory/closing files etc.

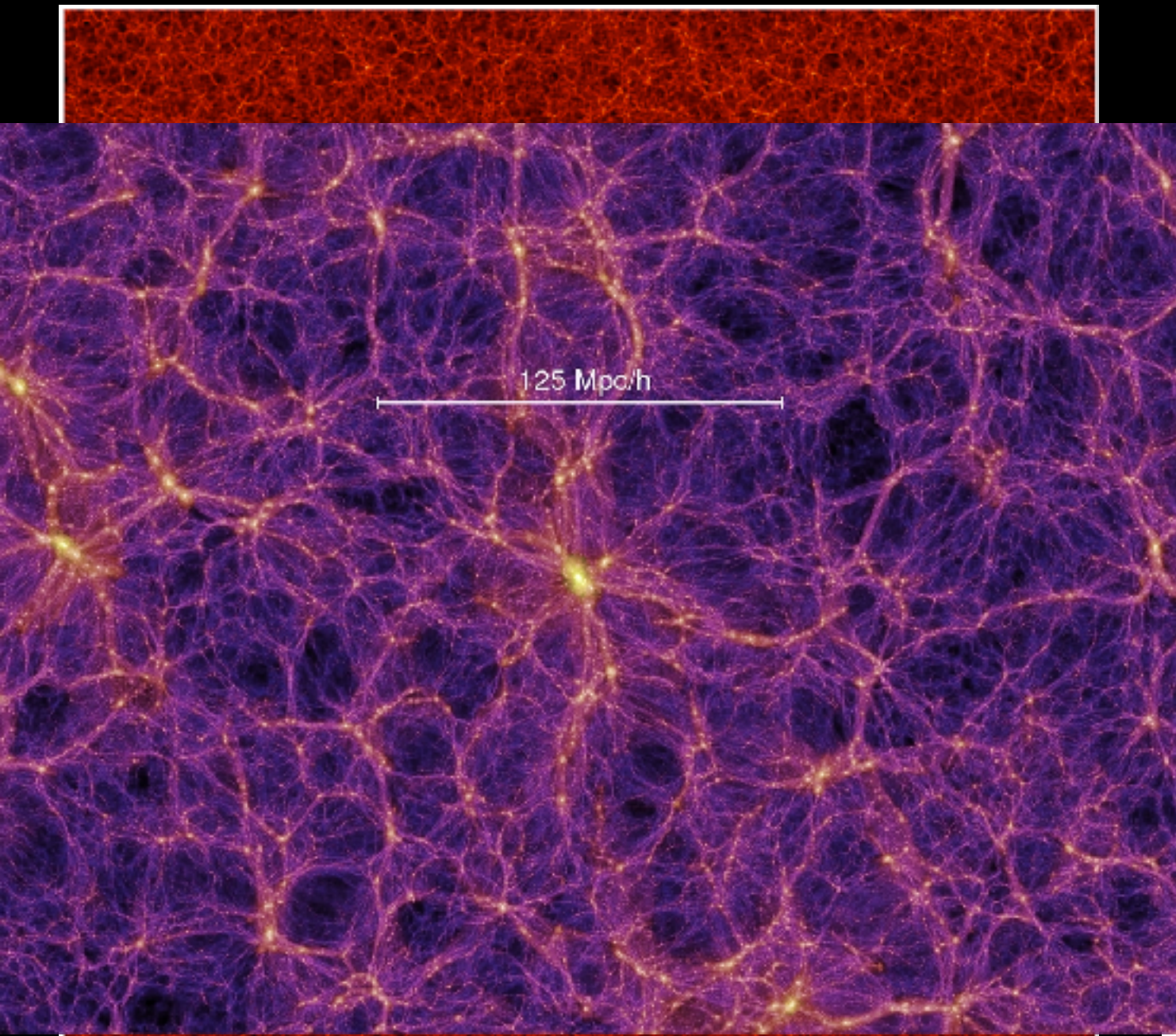
```
def __add__(self, val):  
    ans=self.copy()  
    if isinstance(val, Complex):  
        ans.r=ans.r+val.r  
        ans.i=ans.i+val.i  
    else:  
        try:  
            ans.r=ans.r+val  
        except:  
            print 'Invalid type in Complex.__add__'  
            ans=None  
    return ans
```

```
>>> a=Complex(2,5)  
>>> b=3  
>>> c=a+b  
>>> print c  
5 + 5i  
>>> b='abc'  
>>> print a+b  
Invalid type in Complex.__add__  
None  
>>> █
```

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

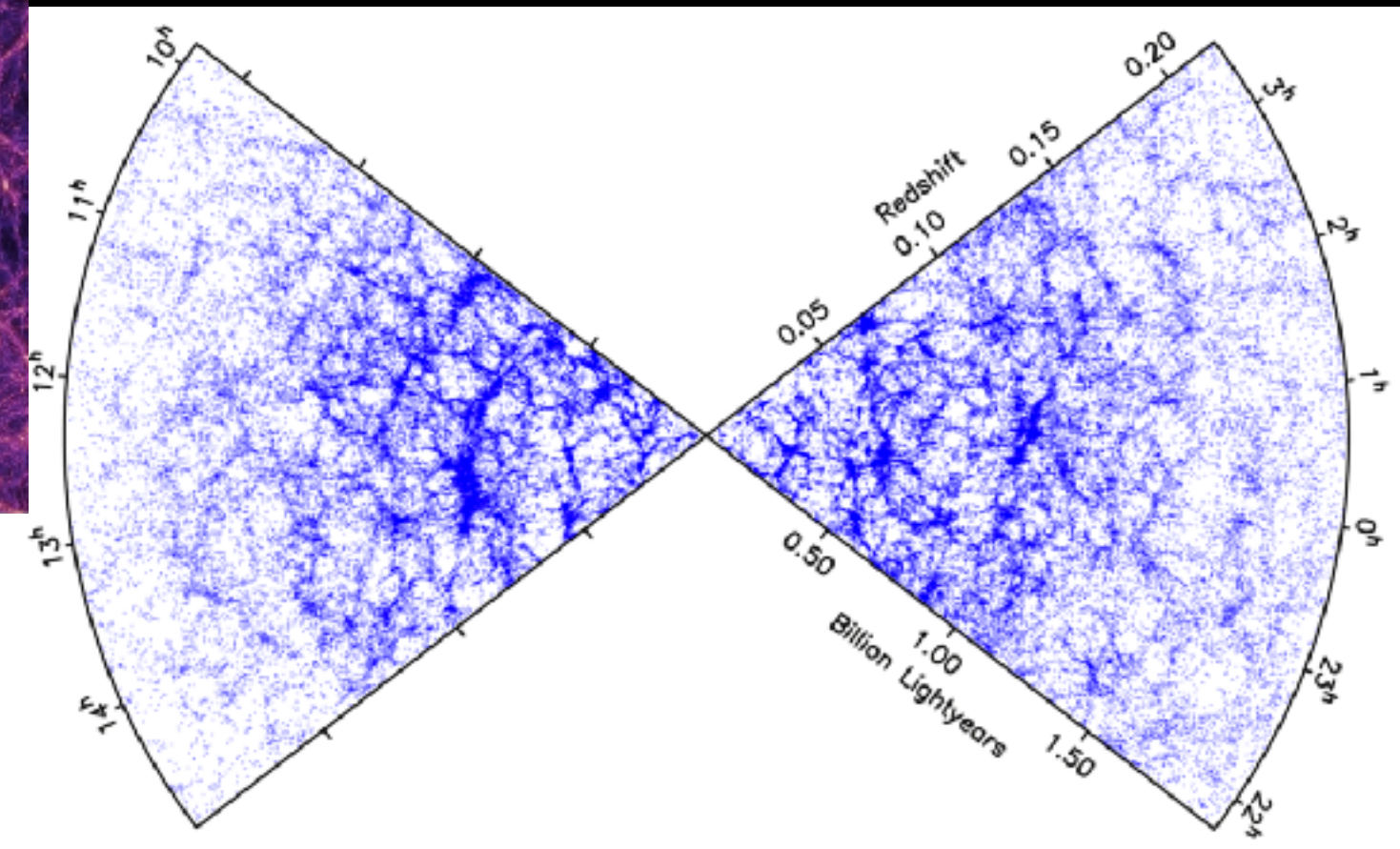


Simulation (left) dark matter,
bottom is galaxy data.

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

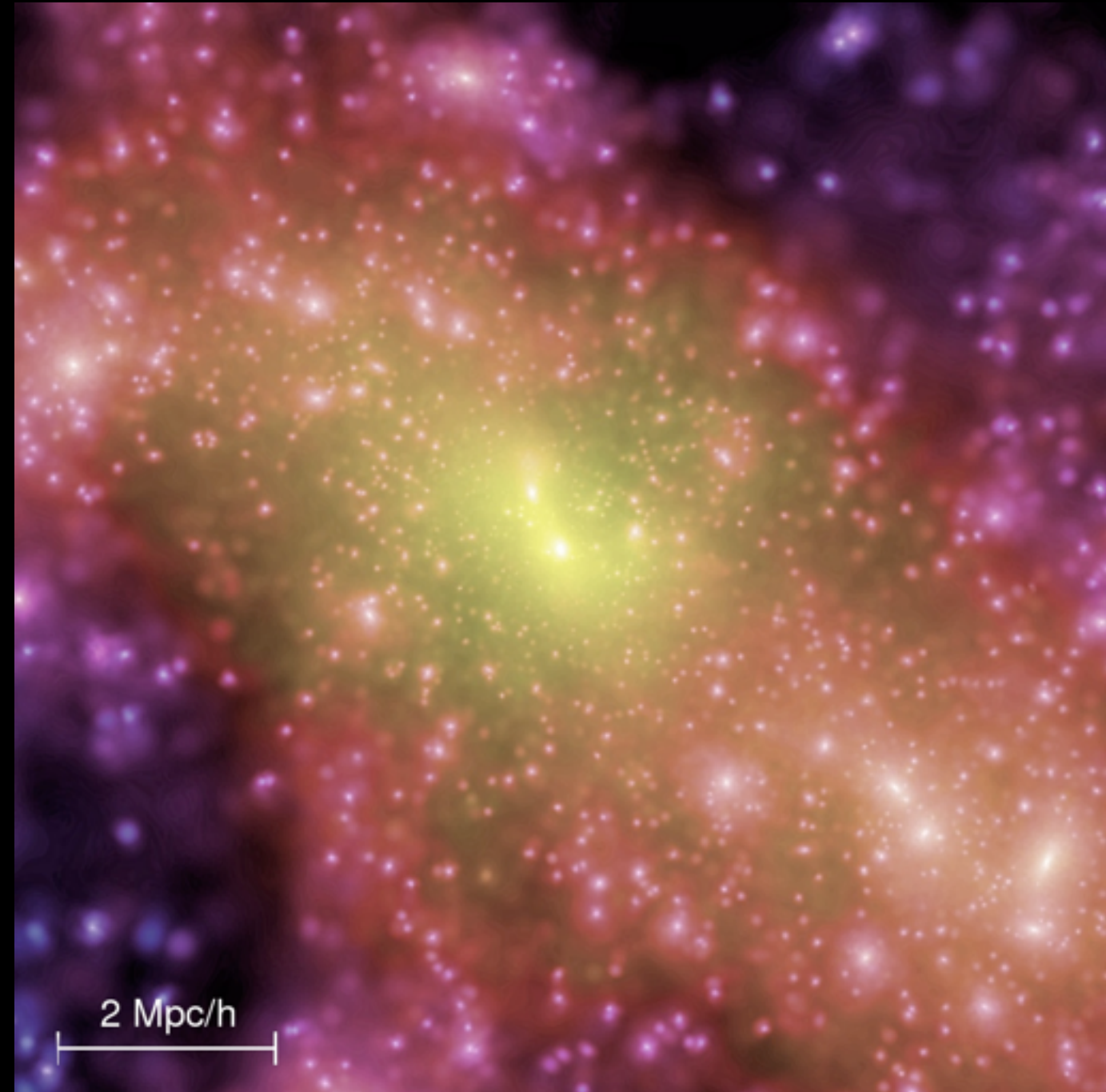
MacFarland, Colberg, White (München), Jenkins,
Pearce, Frenk (Durham), Evrard (Michigan),
Couchman (London, CA), Thomas (Sussex),
Efstathiou (Cambridge), Peacock (Edinburgh)

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



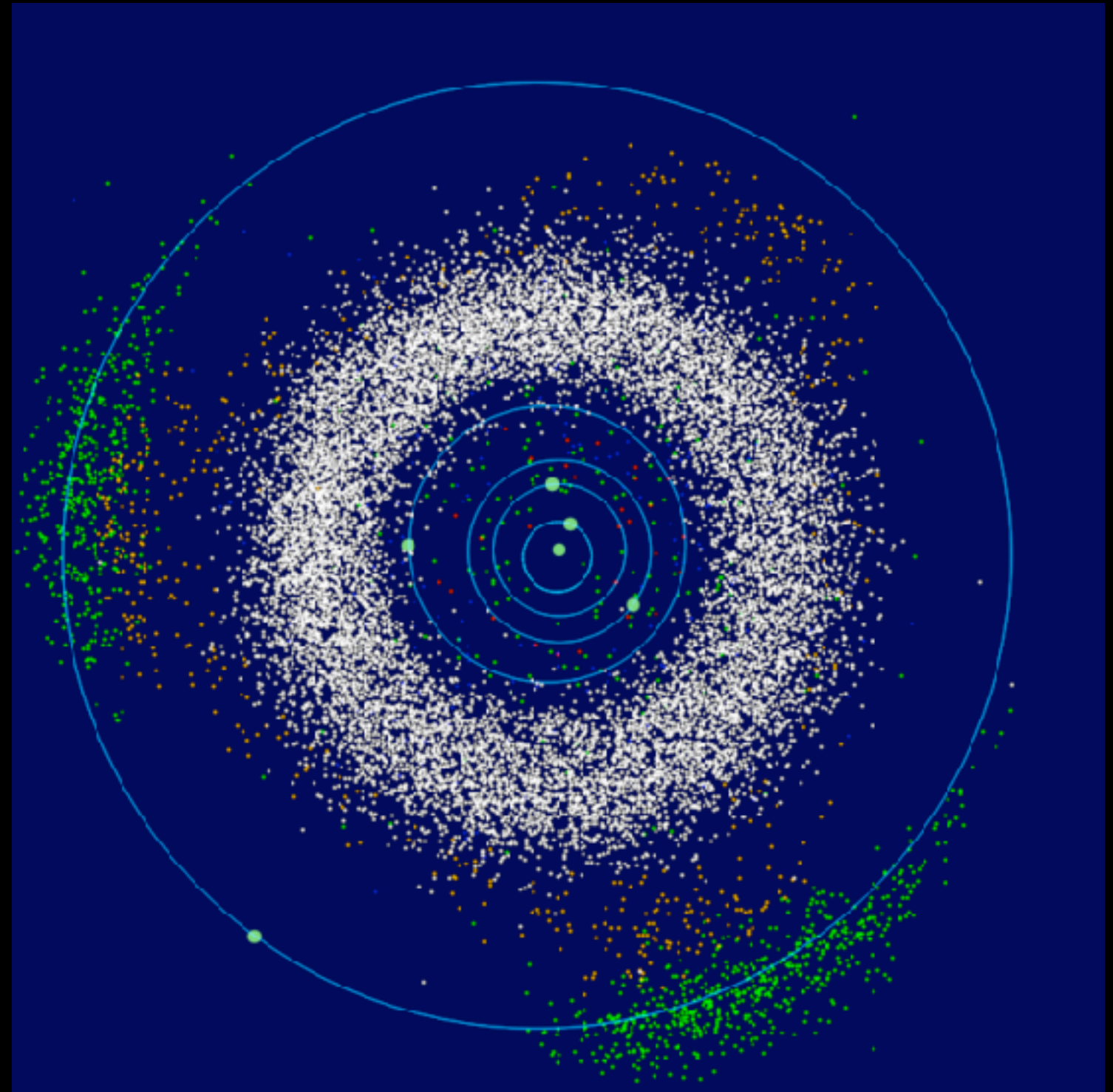
Galaxy Clusters

- 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.



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:
 $d/dt[x_i, v_i] = [v, -\sum Gm_j/r_{ij}^2.]$
- 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

```
import numpy
from matplotlib import pyplot as plt
#let's start two particles 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.



```
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
```

```
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

```
import numpy
from matplotlib import pyplot as plt
#let's start two particles 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.

```
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
```

```
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

```
import numpy
from matplotlib import pyplot as plt
#let's start two particles 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
```

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



```
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
```

```
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: $F \sim r/(r^2+\epsilon^2)^{3/2}$. 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?
- $P_{\text{tot}}(r') = \int P(r-r') \rho(r) d^3r$.
- 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, $n \log n \ll n^2$.
- 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 Problems (Due Monday)

- stay tuned...