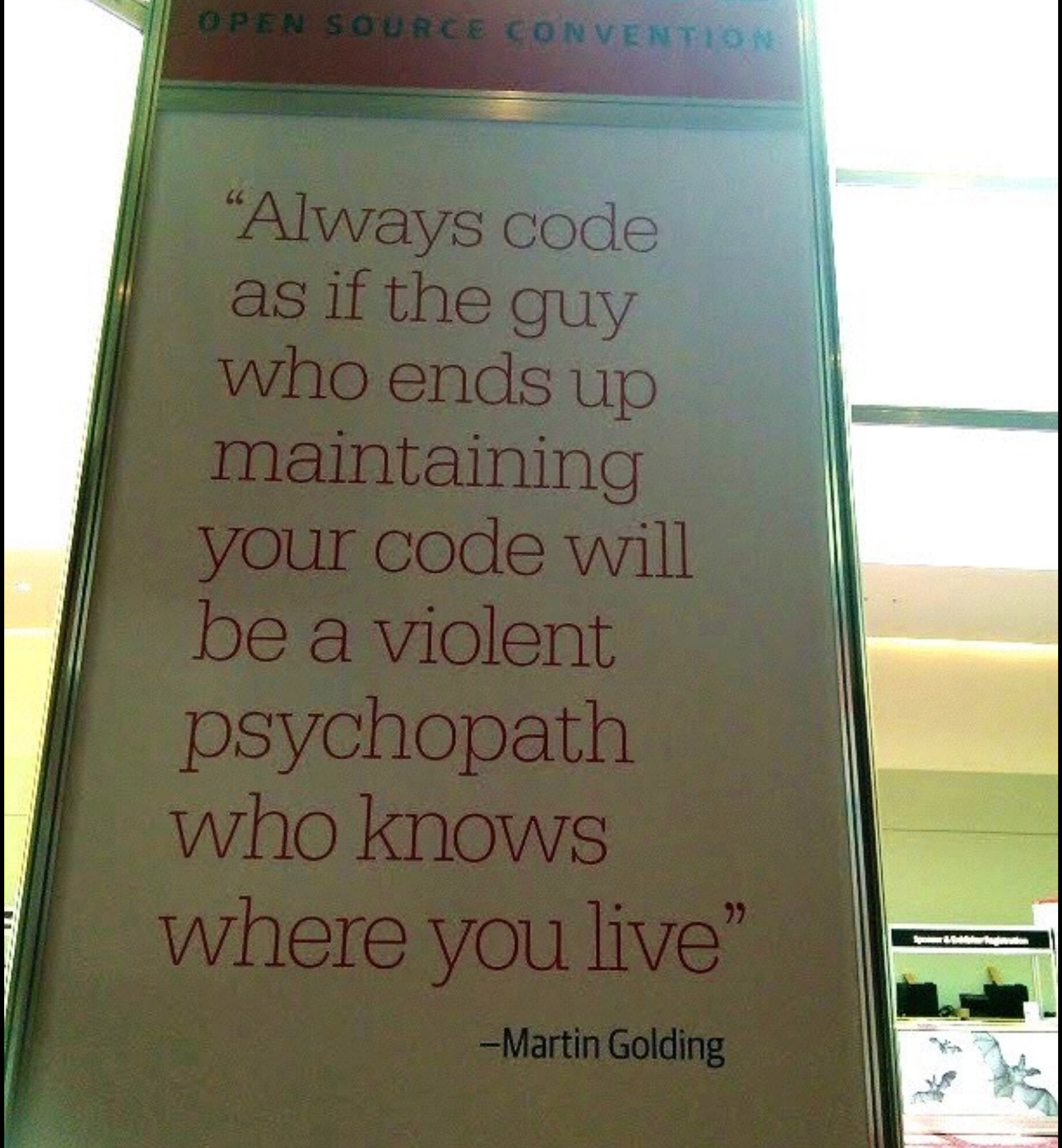
Computational Physics
Lecture 5

sieversj@ukzn.ac.za
Tutor: Hazmatally (hazmatally@gmail.com)

git clone https://github.com/ukzncompphys/lecture5_2017.git

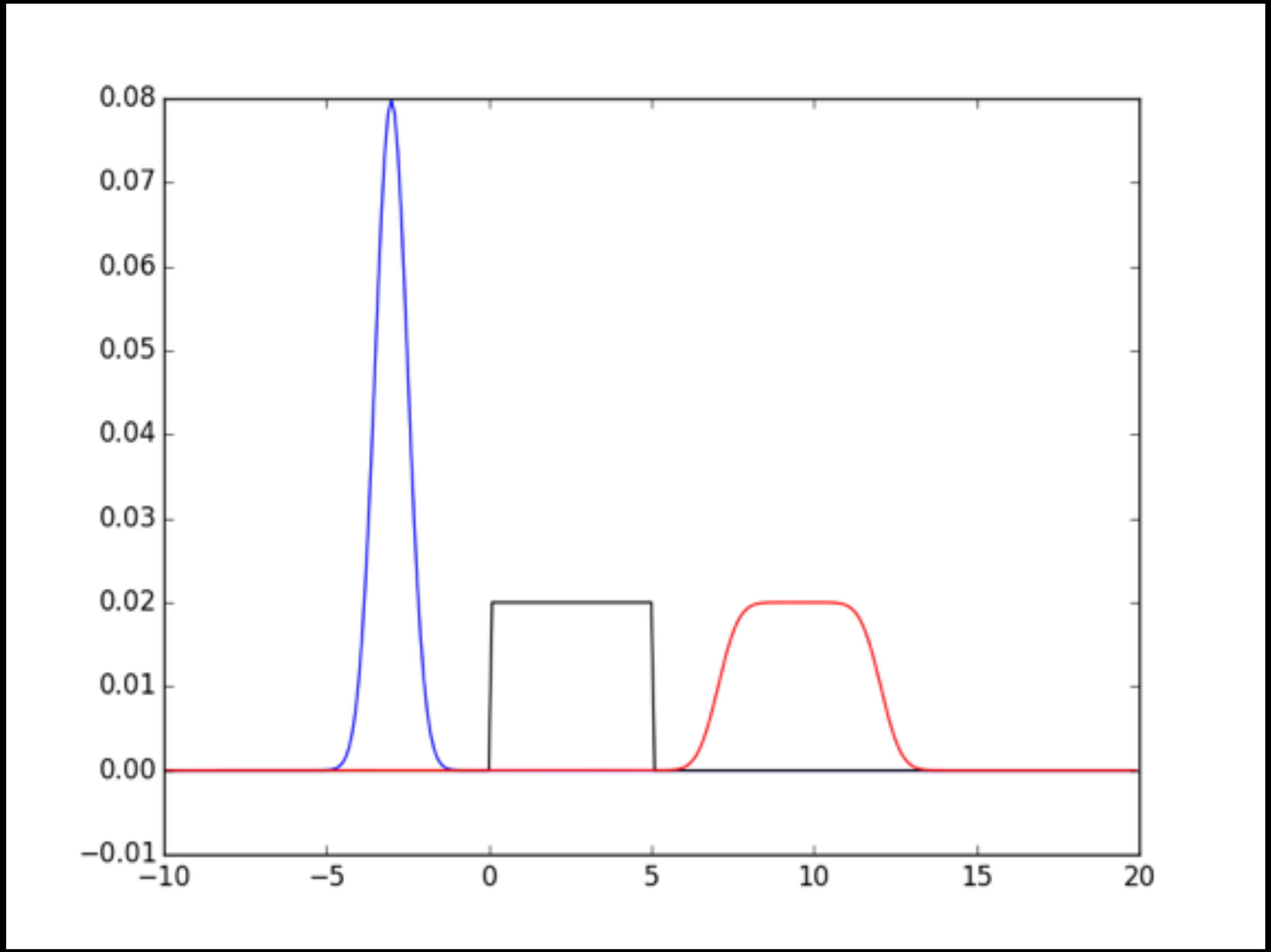

OPEN SOURCE CONVENTION
"Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live"
–Martin Golding

Convolution Example

```
from numpy import arange,exp,real
from numpy.fft import fft,ifft
from matplotlib import pyplot as plt
def conv(f,g):
    ft1=fft(f)
    ft2=fft(g)
    return real(ifft(ft1*ft2))

x=arange(-10,20,0.1)
f=exp(-0.5*(x+3)**2/0.5**2)
g=0*x;g[ (x>0)&(x<5)]=1
g=g/g.sum()
f=f/f.sum()
h=conv(f,g)

plt.plot(x,f,'b')
plt.plot(x,g,'k')
plt.plot(x,h,'r')
plt.savefig('convolved')
plt.show()
```



Fast Fourier Transform

- How many operations does a DFT take?
- Have an N by N matrix operating on a vector of length N - clearly N^2 operations, right?
- Nope! Otherwise we'd never use them. What's actually going on?
- Note DFT=$\sum f(x)\exp(-2\pi ikx/N) = \sum f_{even}(x)\exp(-2\pi ik(2x)/N) + \sum f_{odd}(x)\exp(-2\pi ik(2x+1)/N)$
- $=F_{even}+\exp(-2\pi ik/N)F_{odd}$. Let $W_k=\exp(-2\pi ik/N)$
- if $k>N/2$, then $k^*=k-N/2$ and DFT=$F_{even}+\exp(-2\pi ik^*/N+i\pi)F_{odd}=Feven-W_kF_{odd}$.

FFT cont'd

- So $F(k)=F_{even}(k)+W_kF_{odd}(k)$ $(k<N/2)$ or $F_{even}(k)-W_kF_{odd}(k)$ $(k>=N/2)$
- So, can get *all* the frequencies if I have 2 half-length FFTs.
- Well, just do the same thing again. FFT of a single element is itself.
- This algorithm works for arrays whose length is a power of 2
- Popularized by Cooley/Tukey in early computer days. Later found to go back to Gauss in 1805. Changes computational work from n^2 to $n\log n$.

Windowing

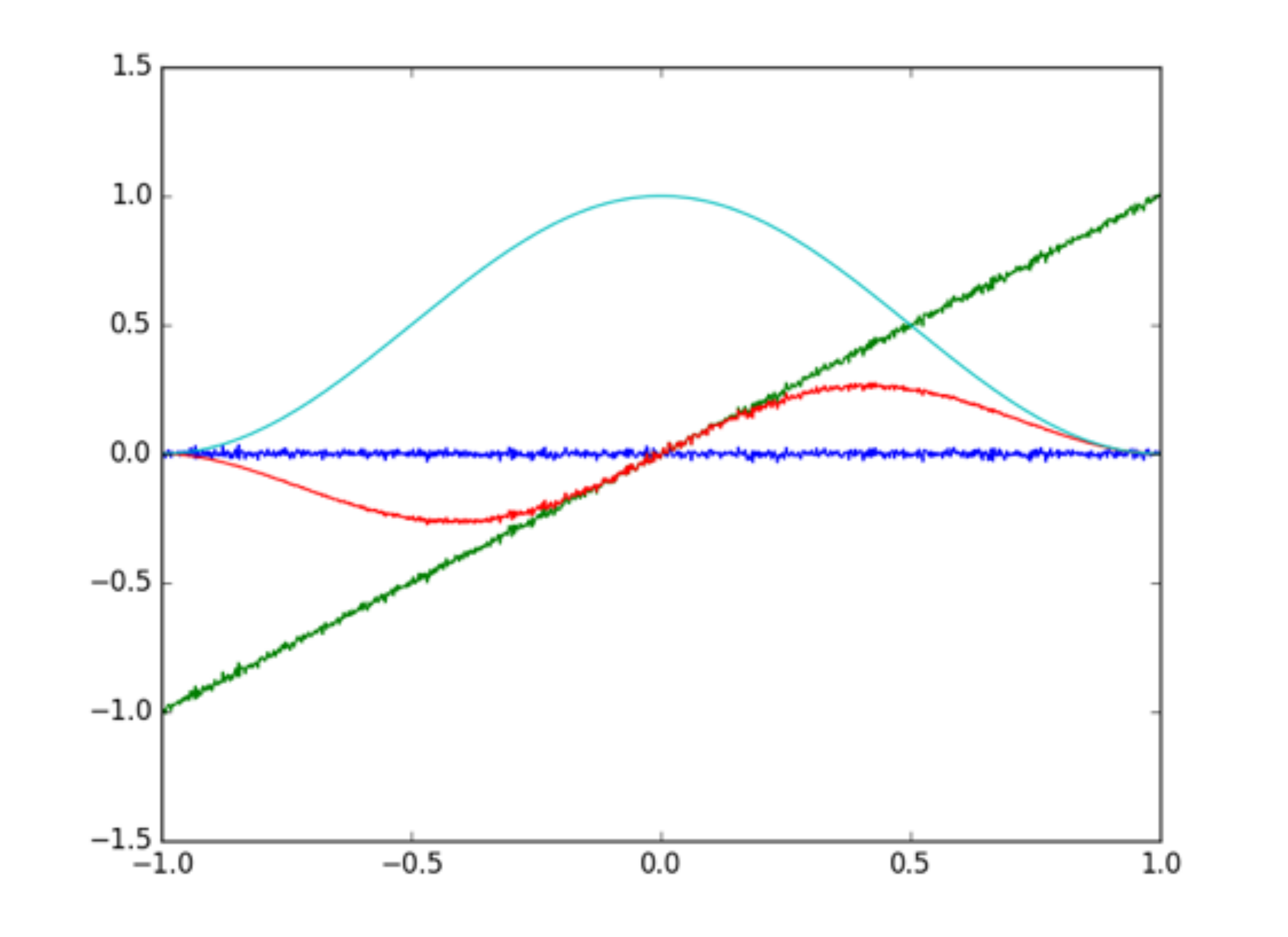
- Jumps around edges cause high frequency power in FFTs. This is a bad thing.
- Standard solution: multiply by a window that goes to zero (or some very small value) at edges.
- There are many possible windows, depending on what you want to do: Hamming, Hanning, cos… 28 listed on wikipedia page.
- If I multiply by window in real space, what have I done in Fourier space?

Window in Real Space

Use cos window that goes to zero on edges w/derivative zero.

If I take a piece of noisy data from a smooth long-term signal, smooth part may look like similar to a line.

What does FT look like?




```
import numpy
from matplotlib import pyplot as plt
plt.ion();

x=numpy.arange(1024);
x=x-1.0*x.mean();x=x/x[-1]
y1=0.01*numpy.random.randn(x.size)
y2=y1+x
window=0.5*(1+numpy.cos(x*numpy.pi))
y3=y2*window
plt.clf();plt.plot(x,y1);plt.plot(x,y2);plt.plot(x,y3)
plt.plot(x,window);plt.savefig('raw_data.png')

y1ft=numpy.fft.rfft(y1)
y2ft=numpy.fft.rfft(y2)
```

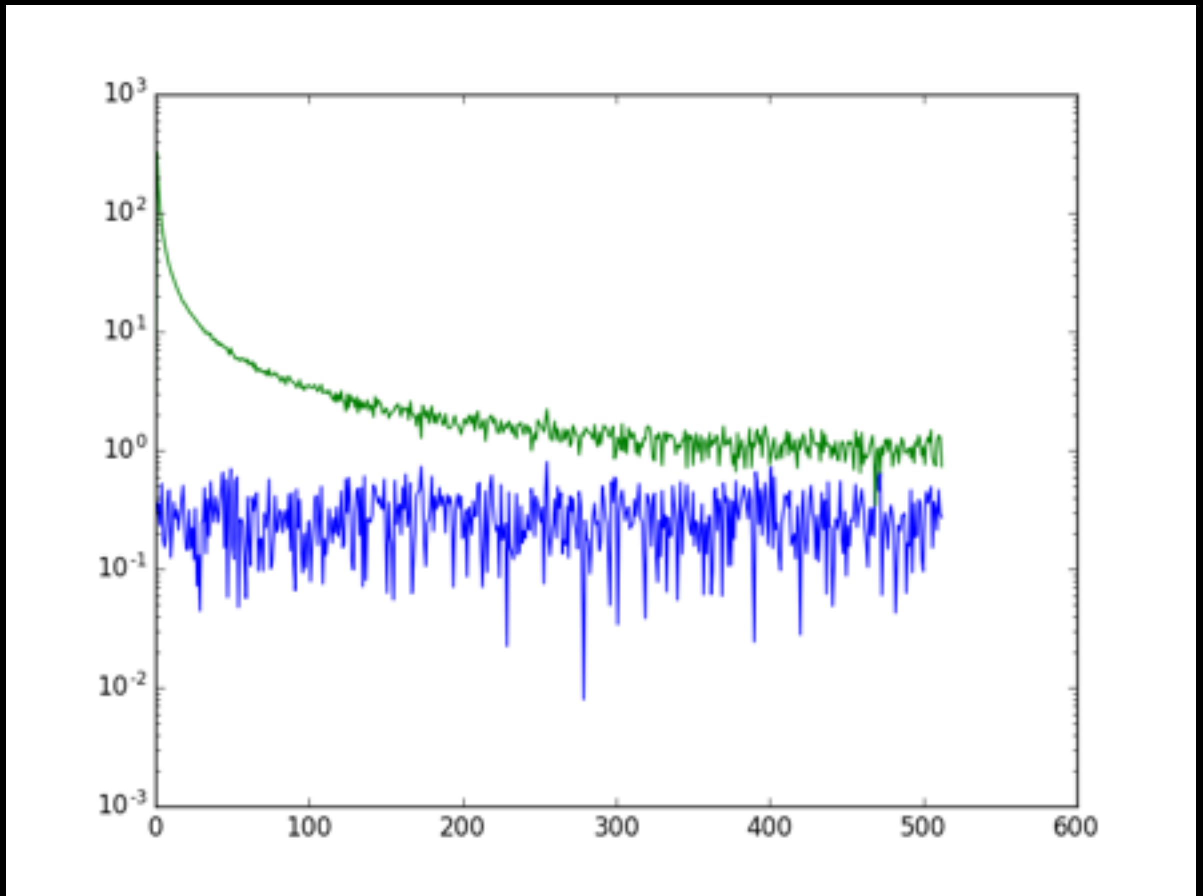
More FT asides

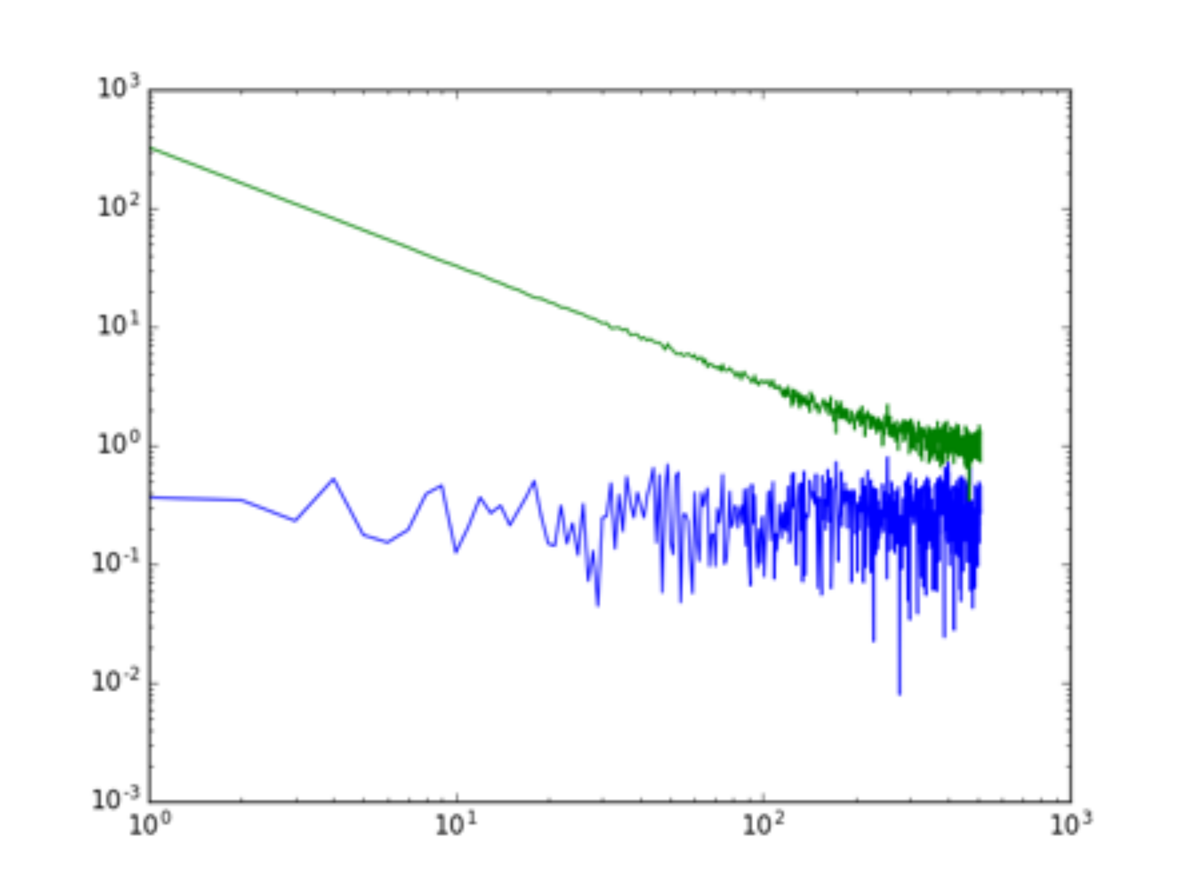
- What is the Fourier transform of a slope?
- What is the (expected) Fourier transform of random noise?
- We will be looking at plots that show the amplitude of the Fourier transforms against wavelength. The variance of the FT is called the *power spectrum*, and is fundamental in many areas of electronics, physics, astronomy…

Effects of Adding Slope

- Even though we know long-term signal is smooth, by taking piece we raise noise level in FT. This is a bad thing.

- Why does the FT look like a line in log-log space?

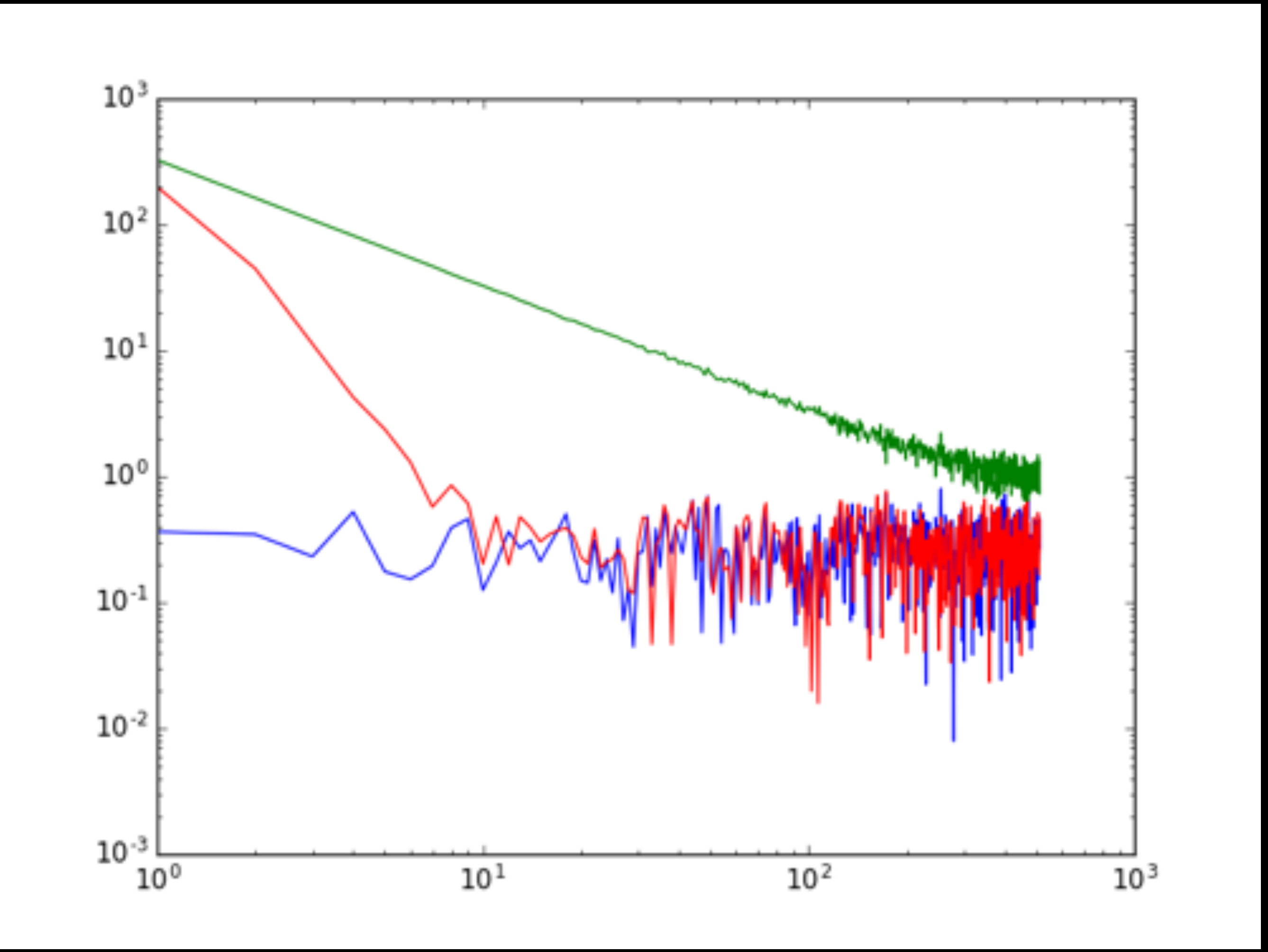




Adding Window

- Multiplying the data with a slope by the window makes the high-frequency power drop back down. This is usually considered a good thing.
- Low frequency power is still large - that's real, we do have a slope in our data.
- What am I doing with that normfac thing?

Parceval's theorem: FT is a unitary rotation, so length before/after must be the same. Windowing removes power, so scale back up by average amount of windowing loss.




```
window=0.5*(1+numpy.cos(x*numpy.pi))
y3=y2*window
#why am I doing this normfac thing?
normfac=numpy.sqrt(numpy.mean(window**2))
y3ft=numpy.fft.rfft(y3)
plt.plot(numpy.abs(y3ft/normfac));
plt.savefig('window_log.png')
```

End of Fourier Transforms

- Any questions?

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

```python
#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)


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

Left: a bare-bones complex number class.

Below: output

```
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 5.385164807134537
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 5.385164807134504
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()
```

```
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$
```

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

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)
5.385164807134537
>>> abs(num)
5.385164807134537
>>> 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.

```python
#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?

```python
#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.

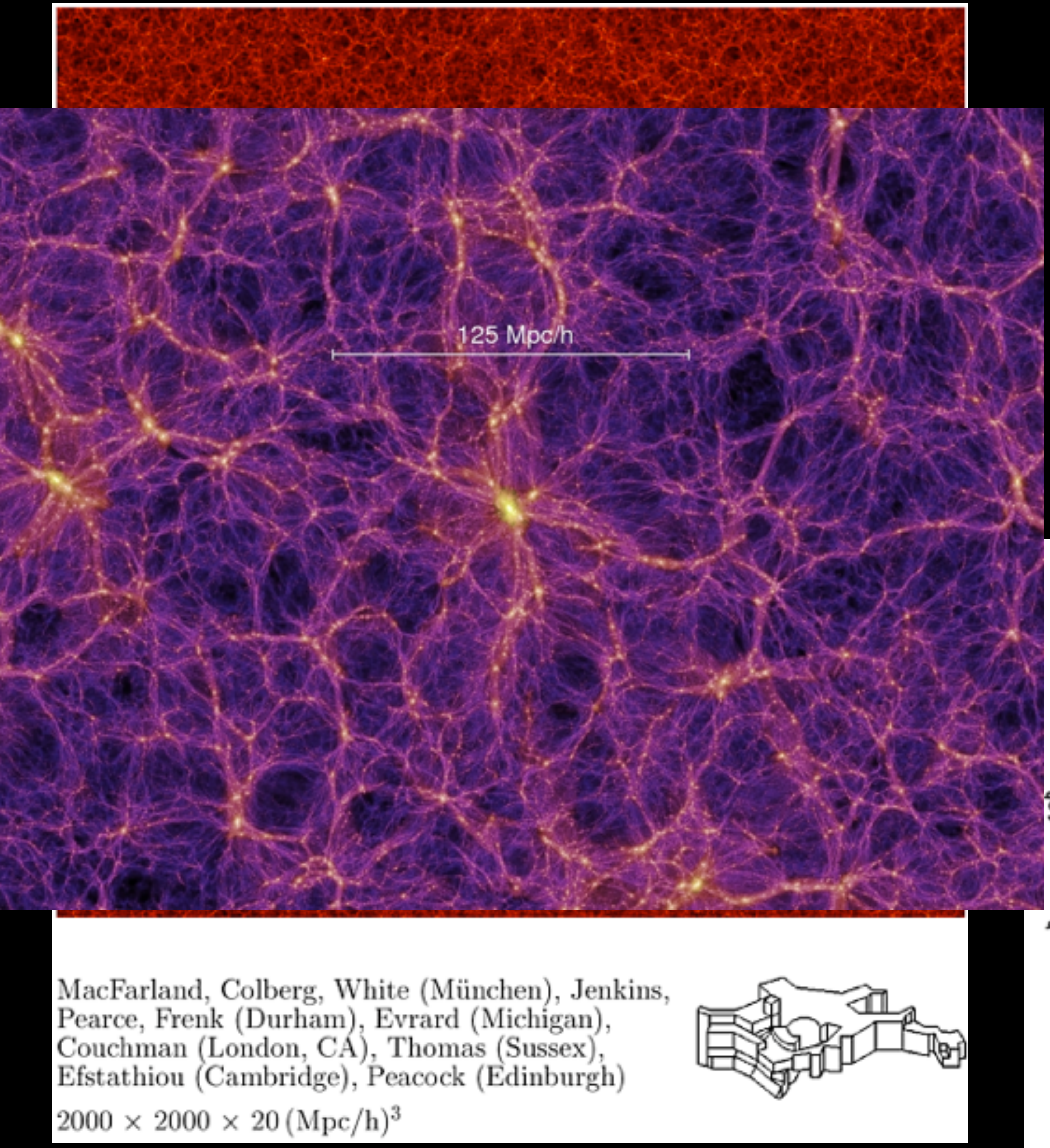
```
def __add__(self,val):
    ans=self.copy()
    if isinstance(val,Complex):
        ans.r=ans.r+val.r
        ans.i=ans.i+val.i
    else:
        try:
            ans.r=ans.r+val
        except:
            print 'Invalid type in Complex.__add__'
            ans=None
    return ans
```

```
>>> 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

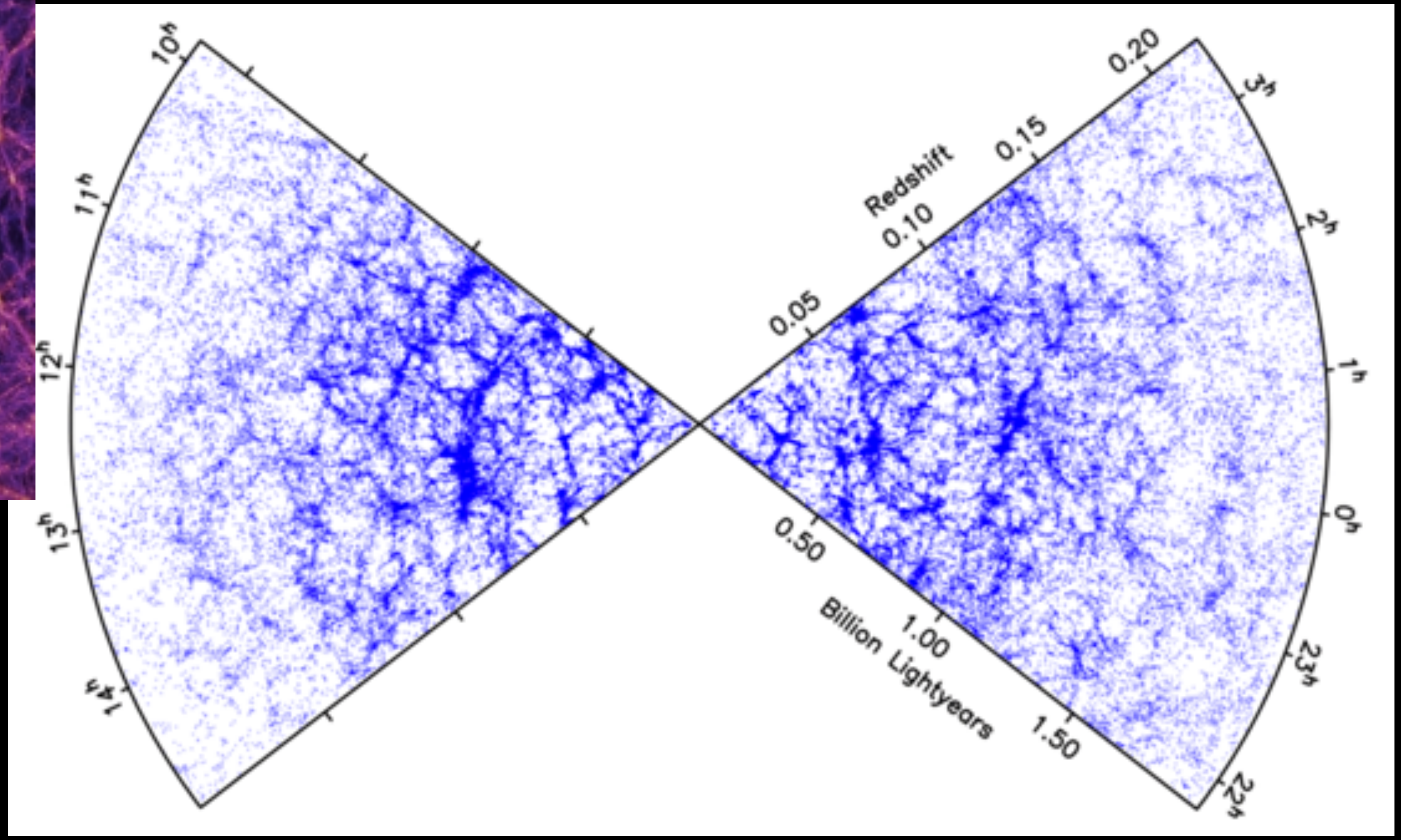
- Dominant force in the universe on large scales is gravity.
- Physical systems often too complex to deal with analytically. Computer simulations often key to understanding.
- Wide variety of problems in e.g. astrophysics involve matter fields evolving with gravity.
- Evolution of 2 masses is called 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



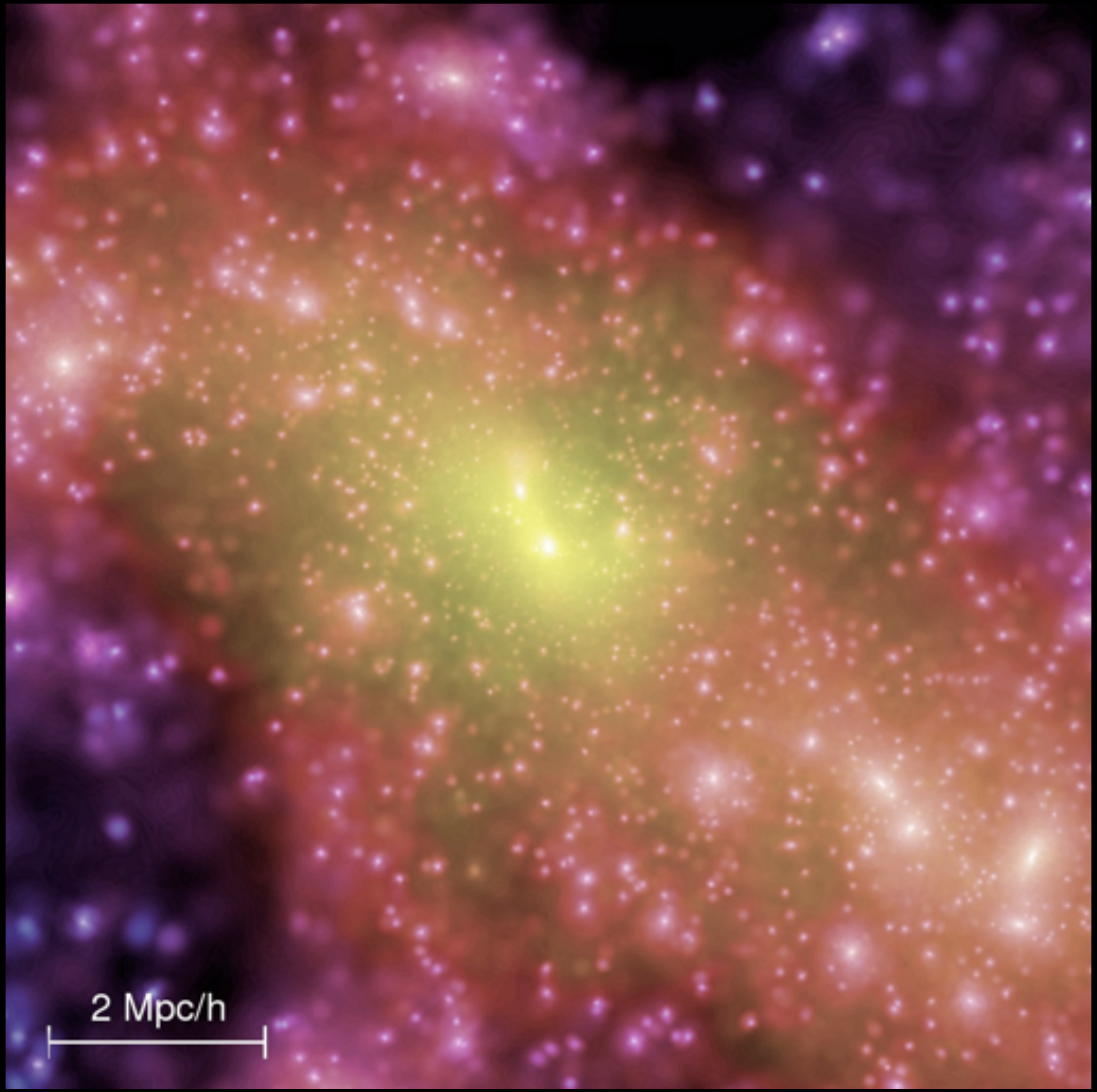
Simulation (left) dark matter, bottom is galaxy data.

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



Galaxy Clusters

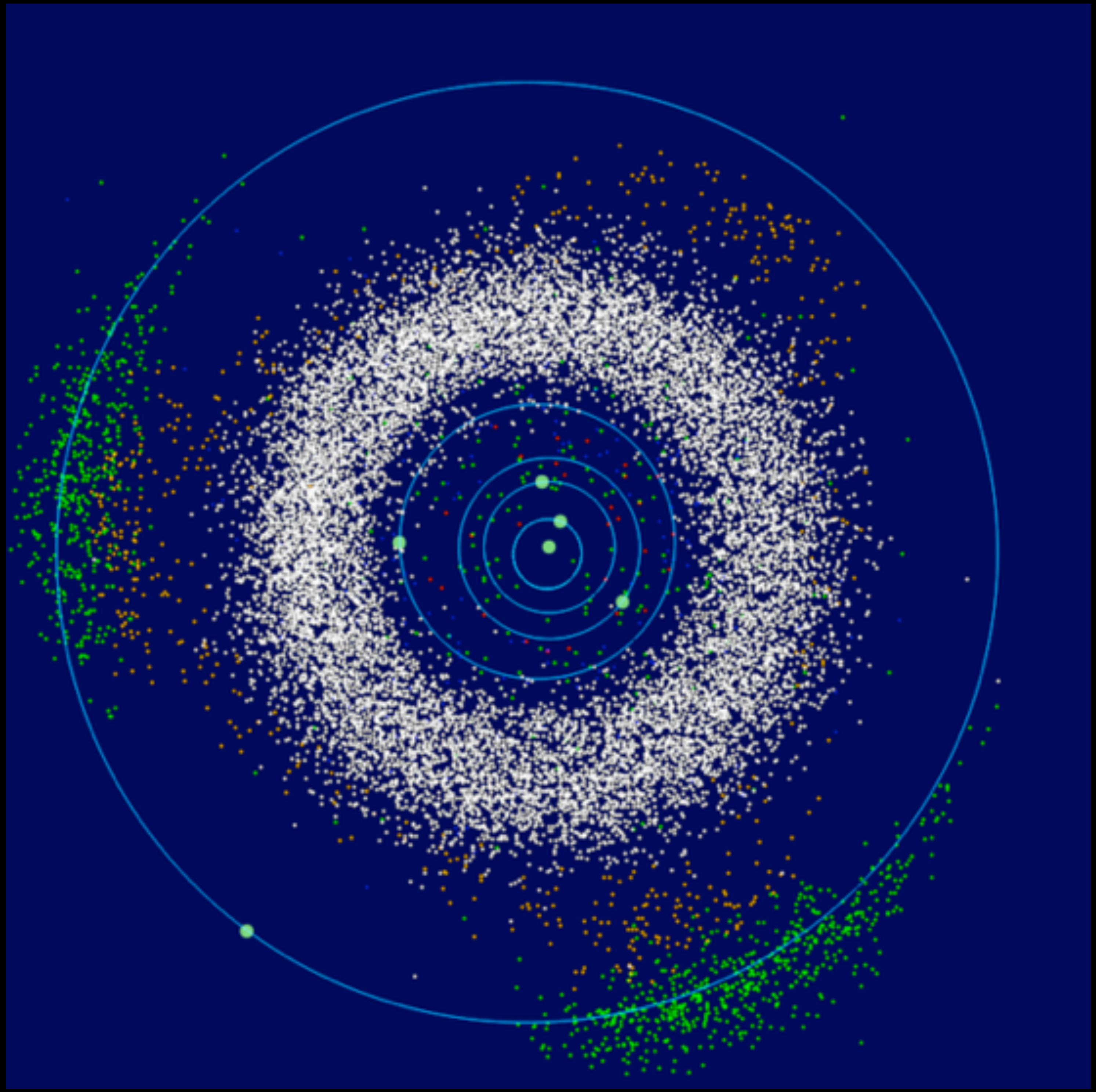
- Galaxy clusters are biggest objects in universe - 10^{15} solar masses.
- Picture from millenium simulation, total of 10^{10} particles.
- Need simulations to interpret galaxy cluster data.



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

Tutorial Problems (Due next week)

- Complete the complex definition to support -,*, and / (__sub__, __mul__, and __div__). Recall that a/b = a*conj(b)/(b*conj(b)). Show from a few sample cases that your functions work. (10)

- Next lecture we will look at n-body simulations. In preparation, write a class that contains masses and x and y positions for a collection of particles. The class should also contain a dictionary that can contain options. Two entries in the dictionary should be the # of particles and G (gravitational constant). The class should also contain a method that calculates the potential energy of every particle, sum(G $m_1 m_2/r_{12}$) . (10)

Tutorial Bonus Problems

- Bonus: extend the complex class to also support arbitrary (i.e. non-integer) powers (keyword is __pow__). +3 if the routine works if a^b works for complex *a* and real *b*, +5 if it works for complex a and complex *b*. (you may ignore branch cuts) (10).

- (if you haven't already done this) You have a sample code that calculates an FFT of an array whose length is a power of 2. Using that routine as a guideline, write an FFT routine that works on an array whose length is a power of 3 (e.g. 9, 27, 81). Verify that it gives the same answer as numpy.fft.fft (10)