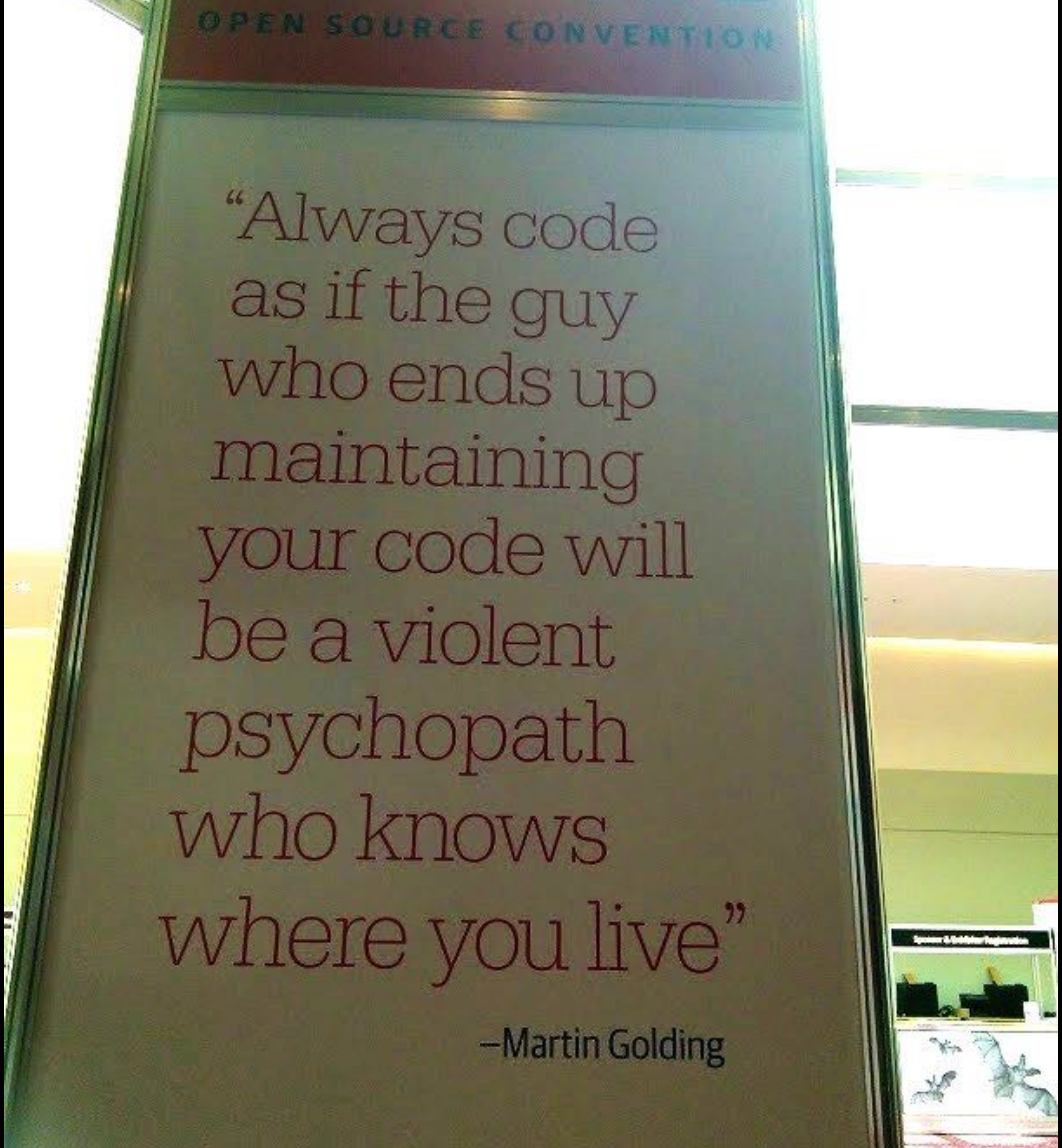
Computational Physics
Lecture 5

sieversj@ukzn.ac.za

git clone https://github.com/ukzncompphys/lecture5_2018.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

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=$F_{even}+\exp(-2\pi ik^*/N+i\pi)F_{odd}=F_{even}-W_kF_{odd}$.

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 *nlogn*.

Sample FFT

- Routine uses *recursion* - function calls itself. Recursion can be very powerful, but also easy to goof.
- numpy.concatenate will combine arrays - note that they have to be passed in as a tuple, hence the extra set of parenthesis
- Modern FFT routines deal with arbitrary length arrays. Fastest Fourier Transform in the West (FFTW) standard packaged - usually used by numpy.

```
from numpy import concatenate,exp,pi,arange,complex
def myfft(vec):
    n=vec.size
    #FFT of length 1 is itself, so quit
    if n==1:
        return vec
    #pull out even and odd parts of the data
    myeven=vec[0::2]
    myodd=vec[1::2]

    nn=n/2;
    j=complex(0,1)
    #get the phase factors
    twid=exp(-2*pi*j*arange(0,nn)/n)

    #get the dfts of the even and odd parts
    eft=myfft(myeven)
    oft=myfft(myodd)

    #Now that we have the partial dfts, combine them with
    #the phase factors to get the full DFT
    myans=concatenate((eft+twid*oft,eft-twid*oft))
    return myans
```

```
>>> import myft
>>> x=numpy.random.randn(32)
>>> xft1=numpy.fft.fft(x)
>>> xft2=myft.myfft(x)
>>> print numpy.sum(numpy.abs(xft1-xft2))
2.33937690259e-13
>>>
```

In Practice

- Should you write your own FFT code? (no)
- Should you understand what is going on under the hood? (yes)

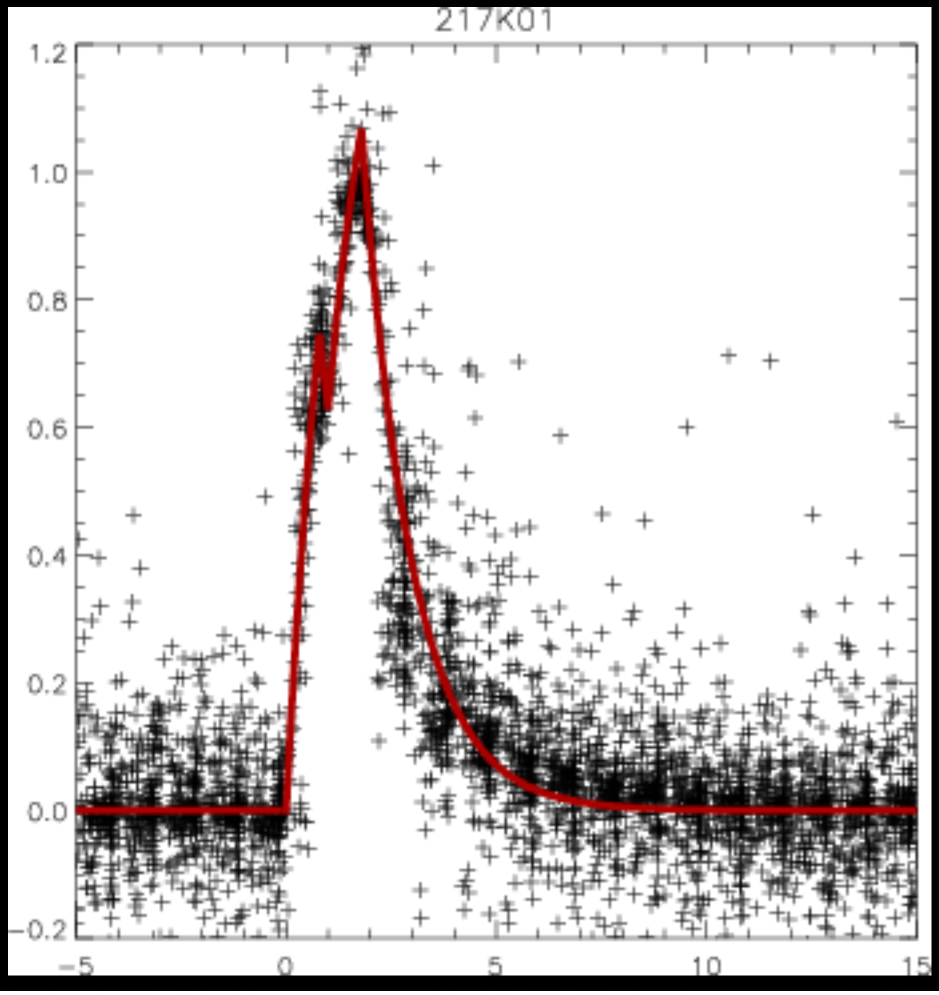
```
import numpy
import time

n=2**16
x=numpy.random.randn(n)
t1=time.time();y=numpy.fft.fft(x);t2=time.time();t_ref=t2-t1
x=numpy.random.randn(n+1) #this is a prime
t1=time.time();y=numpy.fft.fft(x);t2=time.time();t_plus1=t2-t1
x=numpy.random.randn(n+2) #this is has largest factor 331
t1=time.time();y=numpy.fft.fft(x);t2=time.time();t_plus2=t2-t1
x=numpy.random.randn(n+14) #this is has largest factor 23
t1=time.time();y=numpy.fft.fft(x);t2=time.time();t_plus14=t2-t1
print 'Reference time was ',t_ref
print 'Extending by one increased time by a factor of ',t_plus1/t_ref
print 'Extending by two increased time by a factor of ',t_plus2/t_ref
print 'Extending by  14 increased time by a factor of ',t_plus14/t_ref
```

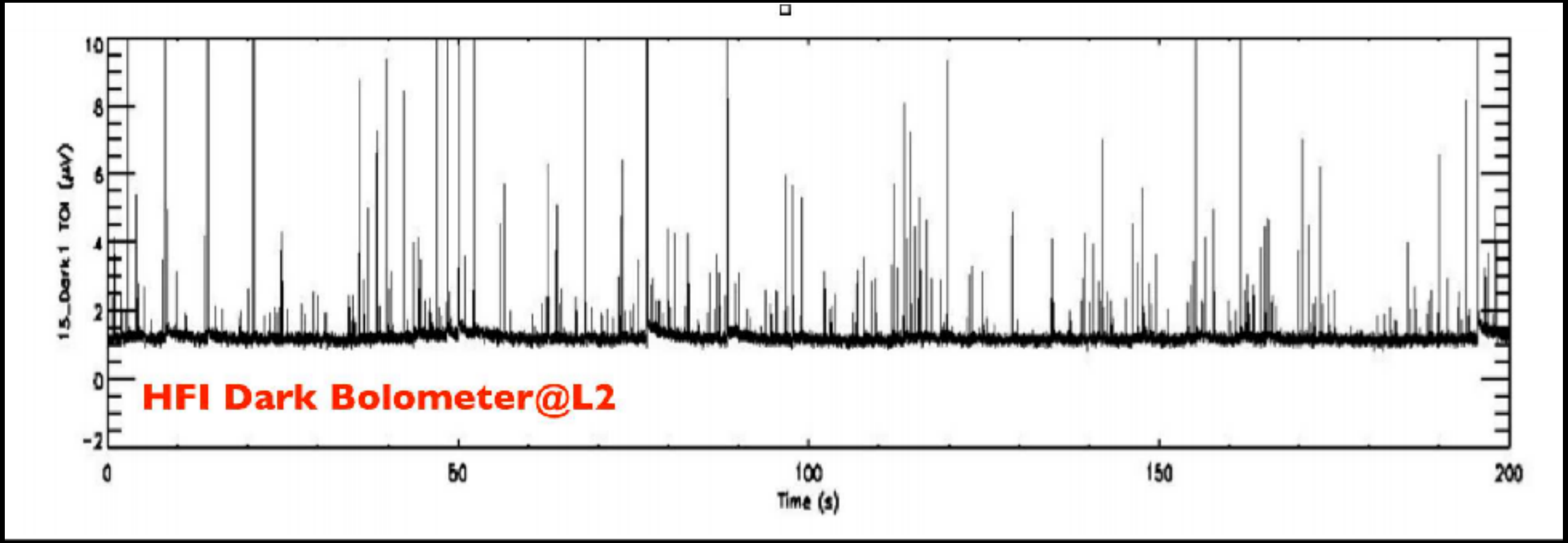
```
>>> execfile('time_ffts.py')
Reference time was  0.00335788726807
Extending by one increased time by a factor of  2594.13178074
Extending by two increased time by a factor of  5.54963078671
Extending by  14 increased time by a factor of  1.8124112468
```

Convolution Theorem

- Convolution defined to be $conv(y)=f\otimes g==\int f(x)g(y-x)dx$
- $\sum_x \sum F(k)\exp(2\pi ikx)\ \sum conj(G(k'))\exp(2\pi ik'x)\exp(-2\pi ik'y/N)$
- Reorder sum: $\sum\sum F(k)conj(G(k'))\exp(-2\pi ik'y/N)\sum_x \exp(2\pi i(k+k')x)$
- equals zero unless k'==-k. Cancels one sum, conjugates G
- $f\otimes g=\sum F(k)G(k)\exp(2\pi iky/N)$=ift(dft(f)*dft(g))
- So, to convolve two functions, multiply their DFTs and take the IFT

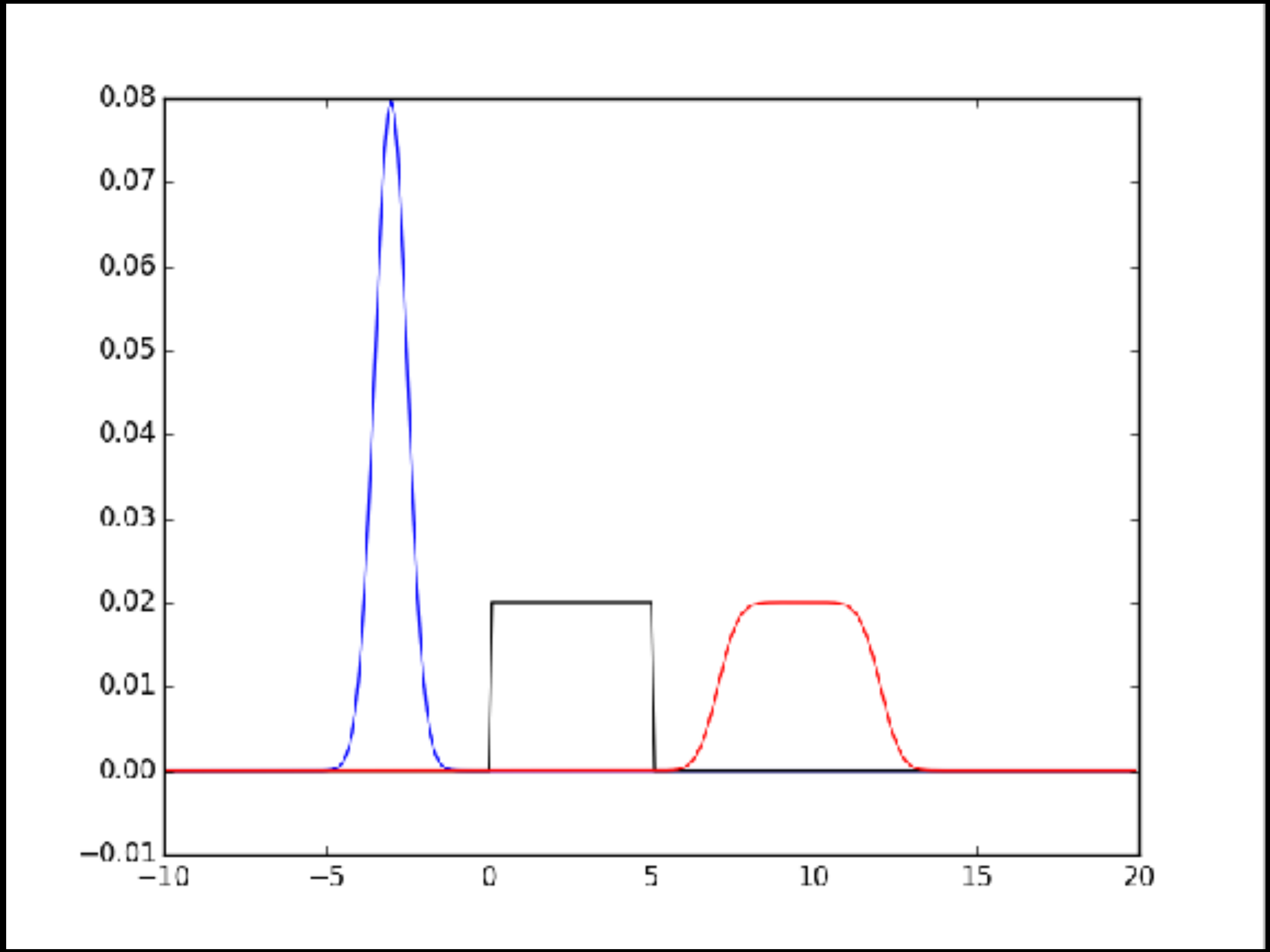
Cosmic Rays from Planck Satellite





Convolution Example

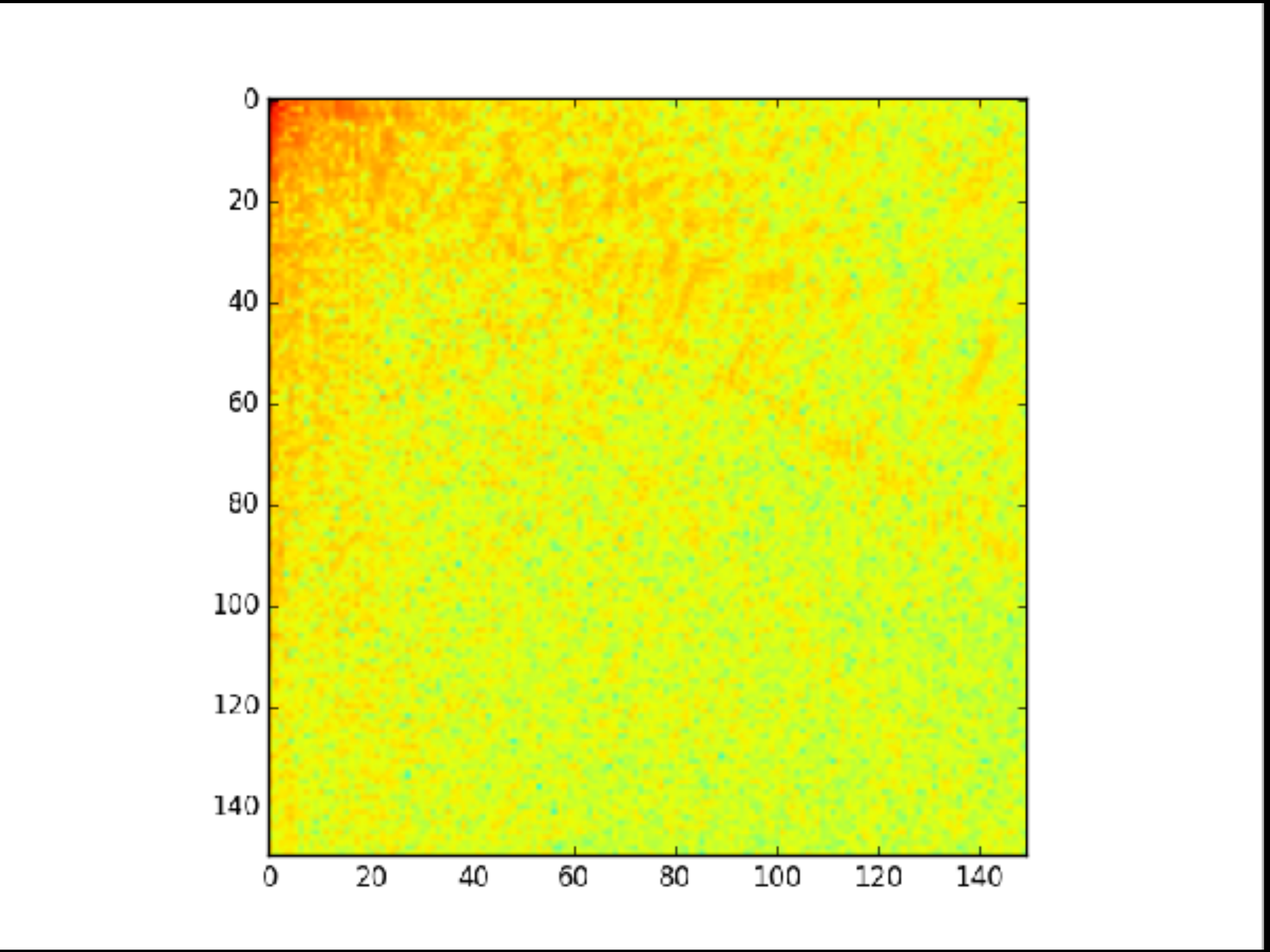
```
from numpy import arange,exp,real
from numpy.fft import fft,ifft
from matplotlib import pyplot as plt
def conv(f,g):
    ft1=fft(f)
    ft2=fft(g)
    return real(ifft(ft1*ft2))

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()
```



2D Fourier Transforms

- Fourier transform defined in 2 dimensions:
- $F(k,l)=\int\int f(x,y)\exp(-ikx)\exp(-ily)dxdy$
- 2D FT's extremely common in image processing.
- JPEGs in fact are based on picking out modes from image FT's.

numpy.fft.fft2



Smoothing Images

- Out-of-focus images are convolutions.
- Can defocus an image by convolving with a blurry kernel
- Let’s fuzz out map by a Gaussian.

```python
def get_fft_vec(n):
    vec=numpy.arange(n)
    vec[vec>n/2]=vec[vec>n/2]-n
    return vec
def smooth_map(map,npix,smooth=True):
    nx=map.shape[0]
    ny=map.shape[1]
    xind=get_fft_vec(nx)
    yind=get_fft_vec(ny)

    #make 2 1-d gaussians of the correct lengths
    sig=npix/numpy.sqrt(8*numpy.log(2))
    xvec=numpy.exp(-0.5*xind**2/sig**2)
    xvec=xvec/xvec.sum()
    yvec=numpy.exp(-0.5*yind**2/sig**2)
    yvec=yvec/yvec.sum()

    #make the 1-d gaussians into 2-d maps using numpy.repeat
    xmat=numpy.repeat([xvec],ny,axis=0).transpose()
    ymat=numpy.repeat([yvec],nx,axis=0)

    #if we didn't mess up, the kernel FT should be strictly real
    kernel=numpy.real(numpy.fft.fft2(xmat*ymat))

    #get the map Fourier transform
    mapft=numpy.fft.fft2(map)
    #multiply/divide by the kernel FT depending on whath we're after
    if smooth:
        mapft=mapft*kernel
    else:
        mapft=mapft/kernel
    #now get back to the convolved map with the inverse FFT
    map_smooth=numpy.fft.ifft2(mapft)

    #since numpy gets imaginary parts from roundoff, return the real part
    return numpy.real(map_smooth)
```

smooth_map.py

```
port numpy
om matplotlib import pyplot as plt
port smooth_map

erkat=plt.imread('meerkat_small.jpg')

othed_map=numpy.zeros(meerkat.shape)
smoothed_map=numpy.zeros(meerkat.shape)
ix_smooth=3.5
ix_restore=4
r i in range(3):
   tmp=numpy.squeeze(meerkat[:,:,i])
   tmp_smooth=smooth_map.smooth_map(tmp,npix_smooth)
   smoothed_map[:,:,i]=tmp_smooth
   tmp2=smooth_map.smooth_map(tmp_smooth,npix_restore,False)
   unsmoothed_map[:,:,i]=tmp2
```

Deconvolution

- Well, if I smear out by multiplying FT's, I can unsmear by dividing, right?
- If yes, worth billions and billions of your favo(u)rite currency. Save those fuzzy pictures…
- Maybe. Let's try smoothing image by 3.5 pixels, then unsmoothing by 4.
- What happened?

Deconvolution

- smoothing lowers high-frequency signal
- unsmoothing *must* raise it back up.
- if there's any noise, it gets amplified by unsmoothing.
- If I smooth, then write to jpg, I round to nearest integer. Equivalent to adding noise.
- So, think those fuzzy license plates in google maps are safe?

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…

Windowing

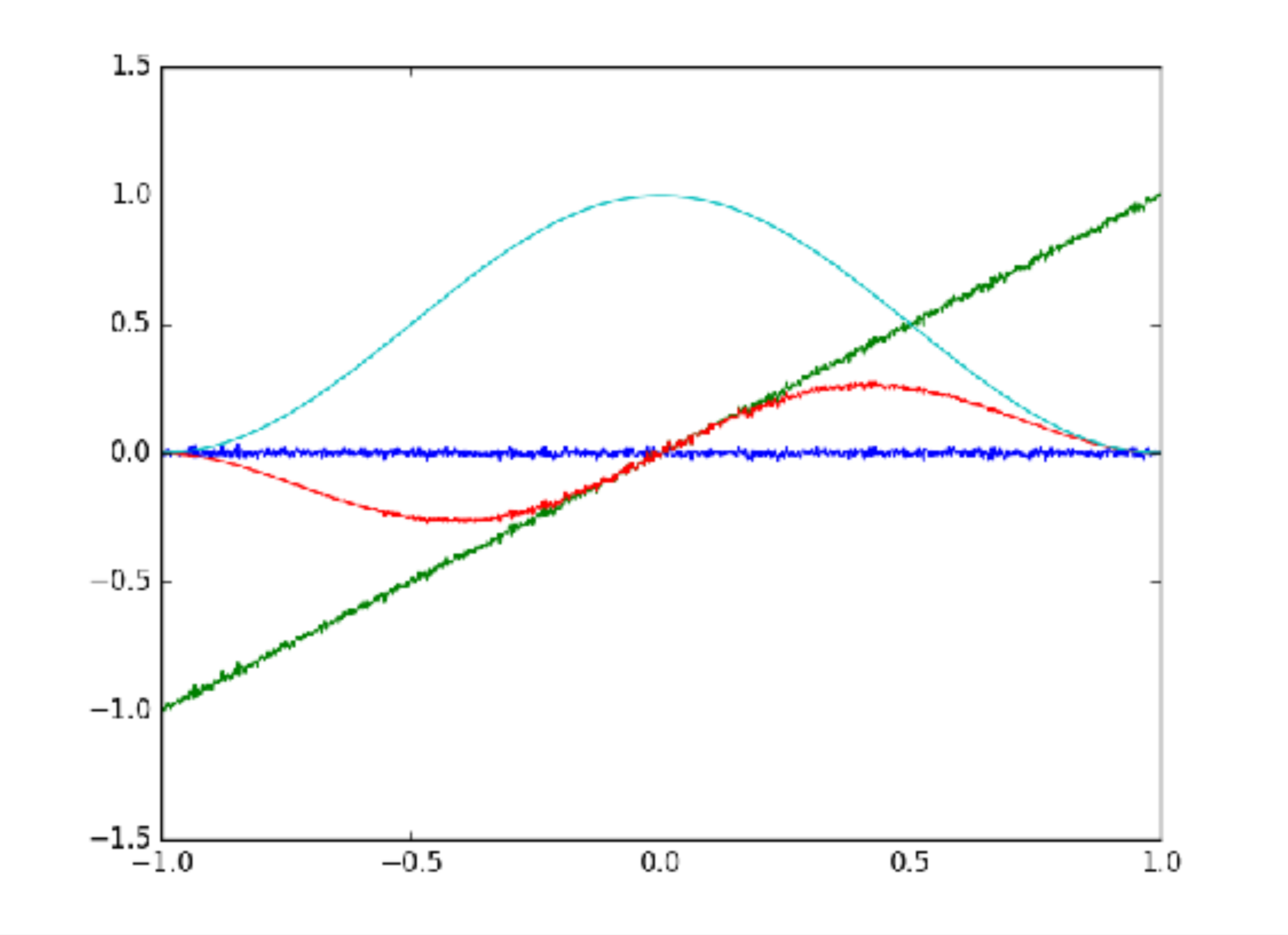
- Jumps around edges cause high frequency power in FFTs. This is a bad thing.
- Standard solution: multiply by a window that goes to zero (or some very small value) at edges.
- There are many possible windows, depending on what you want to do: Hamming, Hanning, cos… 28 listed on wikipedia page.
- If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

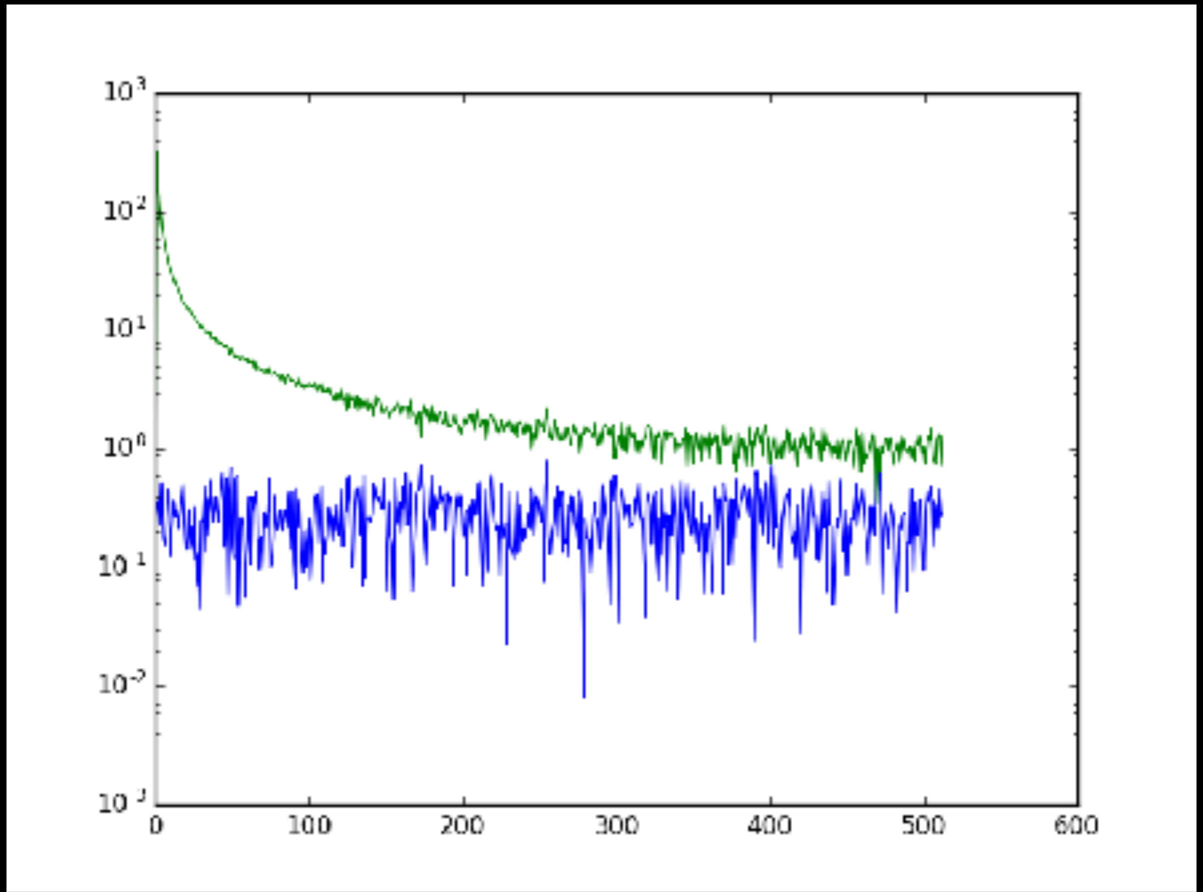
If I take a piece of noisy data from a smooth long-term signal, smooth part may look like similar to a line.

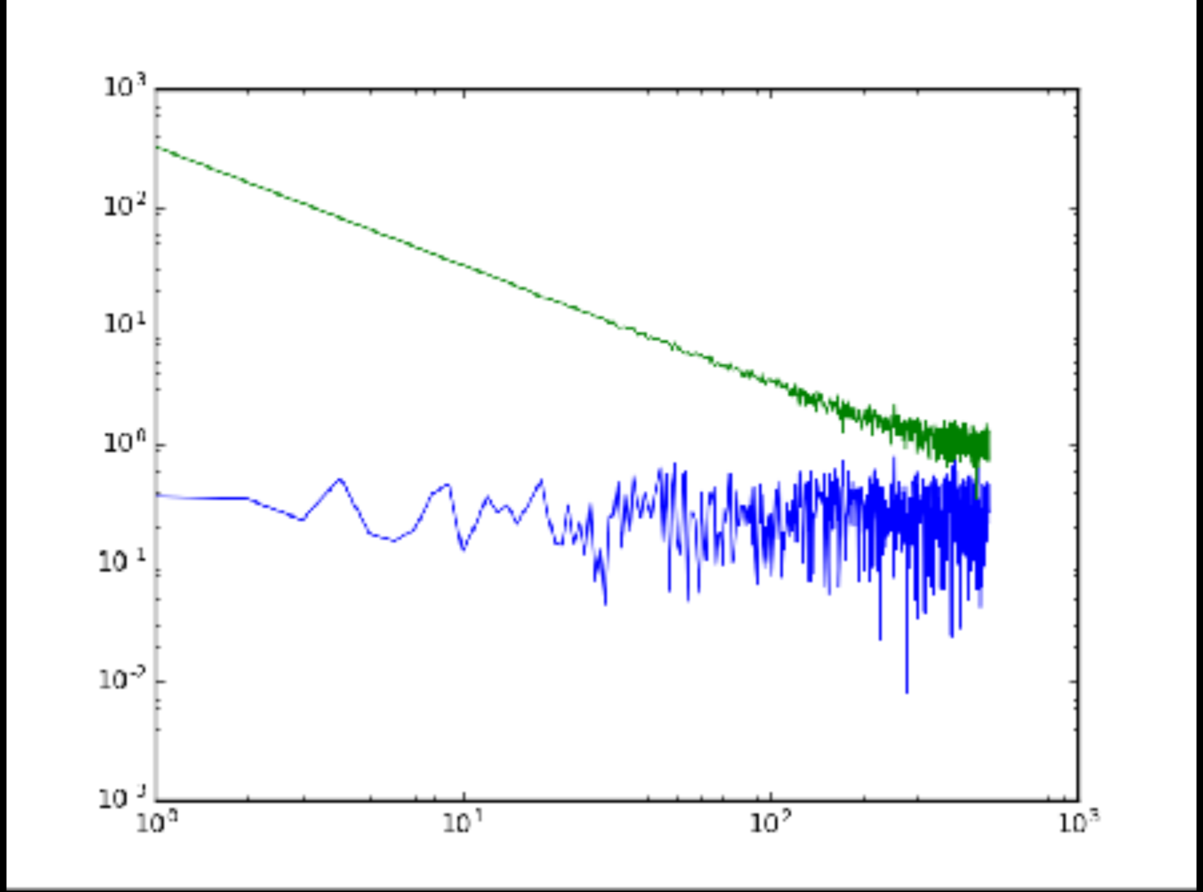
What does FT look like?




```
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)
```

Effects of Adding Slope



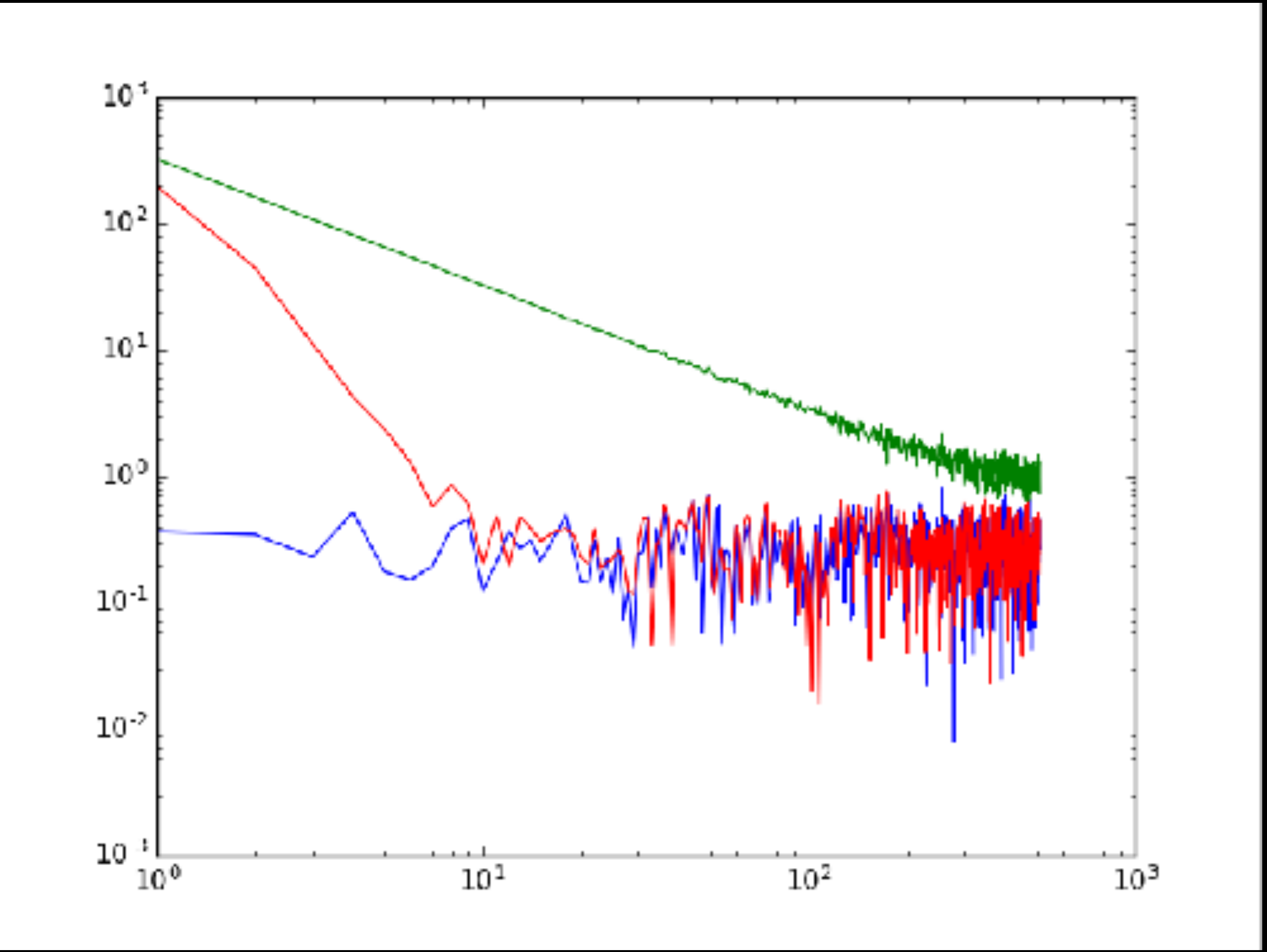


- Even though we know long-term signal is smooth, by taking piece we raise noise level in FT. This is a bad thing.
- Why does the FT look like a line in log-log space?

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')
```

Tutorial Problems (final version Wed.)

- Write a function that will shift an array by an arbitrary amount using a convolution (yes, I know there are easier ways to do this). The function should take 2 arguments - an array, and an amount by which to shift the array. Plot a gaussian that started in the centre of the array shifted by half the array length. (10)

- The correlation function f★g is ∫f(x)g(x+y). Through a similar proof, one can show f★g=ift(dft(f)*conj(dft(g))). Write a routine to take the correlation function of two arrays. Plot the correlation function of a Gaussian with itself. (10)

- Using the results of part 1 and part 2, write a routine to take the correlation function of a Gaussian (shifted by an arbitrary amount) with itself. How does the correlation function depend on the shift? Does this surprise you? (10)

- The circulant (wrap-around) nature of the dft can sometimes be problematic. Write a routine to take the convolution of two arrays *without* any danger of wrapping around. You may wish to add zeros to the end of the input arrays. (10)

Tutorial Bonus Problem

- 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)