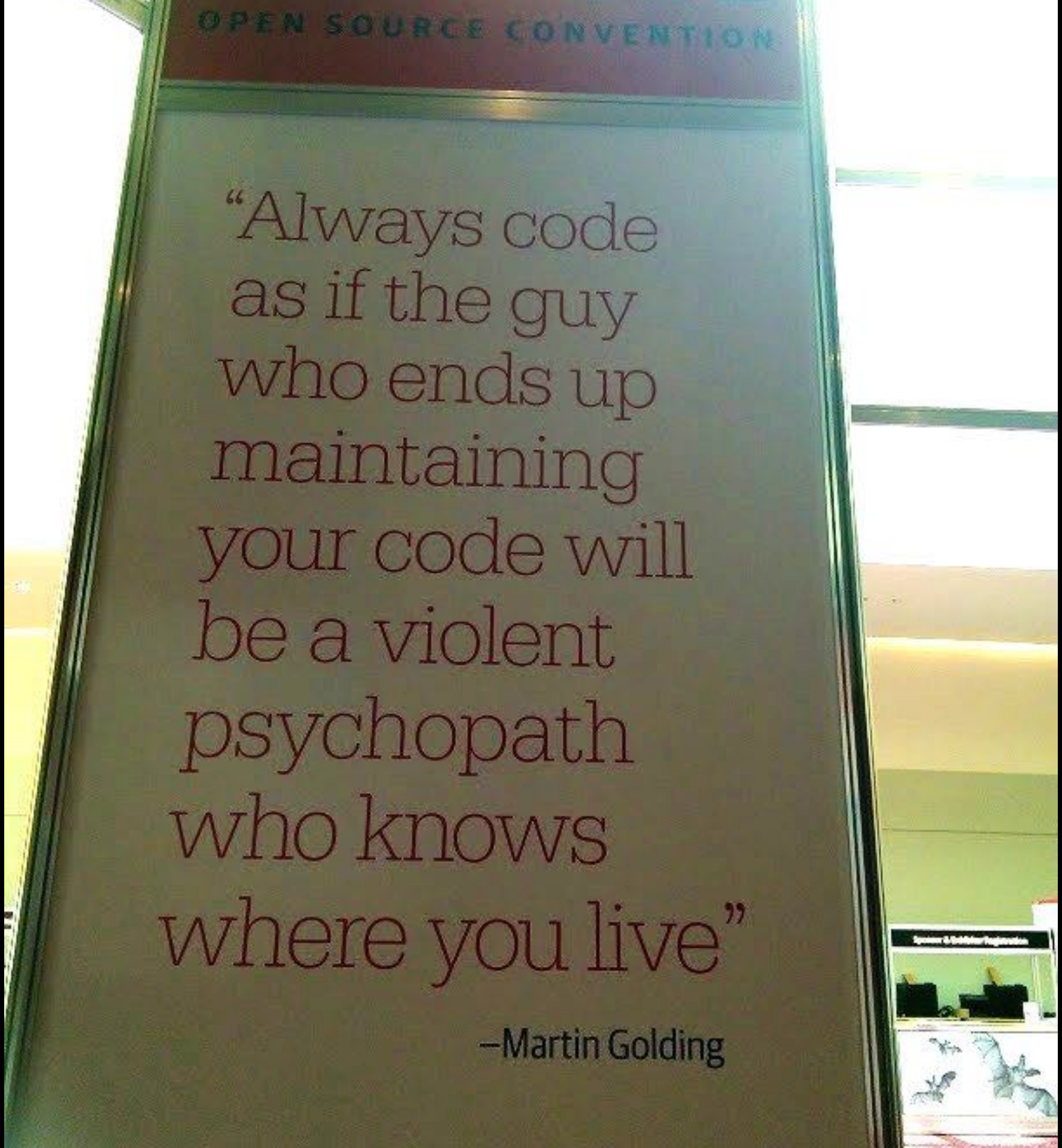
Computational Physics
Fourier Transforms


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

Fourier Transforms

- Functions can be represented in many different ways
- We normally use "real" space - $f(x)$
- Generally, arbitrarily many transforms exist to represent functions in different spaces - $F(y)=Af(x)$ for some matrix A and some new variable *y*. Iff A is invertible, $f(x)=A^{-1}F(y)$
- One important basis nature has picked out is complex exponentials/sines and cosines. Fundamental across physics, particularly quantum mechanics.

Fundamental Definition

- $F(k)=\int f(x)*\exp(-2\pi ikx)dx$ (where $k=1/\omega$)
- Integral gets rid of *x*, replaces with *k*. New function has amplitude and phase as a function of *k*.
- Quantum mechanics - de Broglie says $p = \hbar k$. So, Fourier transform position to get momentum.
- Fourier transform electric field $E(t)$ to get frequency spectrum.
- Fourier transform to get fast correlations, convolutions, many other things.

DFT (Discrete FT)

- Computers don't do continuous. Not enough RAM for starters…
- Function exists over finite range in x at finite number of points.
- If input function has *n* points, output can only have *n* k's.
- Gives rise to discrete Fourier Transform (DFT)
- $F(k)=\sum f(x)\exp(-2\pi ikx/N)$ for N points and $0 \leq k < N$
- What would DFT of $f(0)=1$, otherwise $f(x)=0$ look like?
- What would DFT of $f(x)=1$ look like?
- DFTs have subtle behaviours not seen in continuous, infinite FTs.

Inverse

- One way to think about DFT is as a matrix multiply.
- $F(k)=Af$, $A_{mn}=\exp(2\pi i\ mn/N)$
- But look - $A_{mn}==A_{nm}$, so matrix is symmetric.
- Also, columns are orthogonal under conjugation: $\sum\exp(-2\pi ikx)\exp(2\pi ik'x)=\sum\exp(2\pi i(k'-k)x)$. N if $k'==k$, otherwise 0.
- So, $A^{-1}=1/N*\mathrm{conj}(A)$. $IFT=1/N\ \sum F(k)\exp(-2\pi ikx)$.
- Get back to where we started by just doing another DFT with a sign flip, then divide by # of data points.
- Alternative: divide by $\sqrt{(N)}$ in both DFT and IFT, (not standard computationally)

Numpy Complex

```
import numpy
def exp_prod(m,n,N):
    #define imaginary unity
    J=numpy.complex(0,1)
    #now rest of code is just like for real numbers
    x=numpy.arange(0.0,N)*2*J*numpy.pi/N
    return numpy.sum(numpy.exp(-1*x*m)*numpy.exp(x*n))
if __name__=="__main__":
    print exp_prod(0,0,8)
    print exp_prod(2,4,8)
    print exp_prod(3,3,8)
    print exp_prod(0,7,8)
```

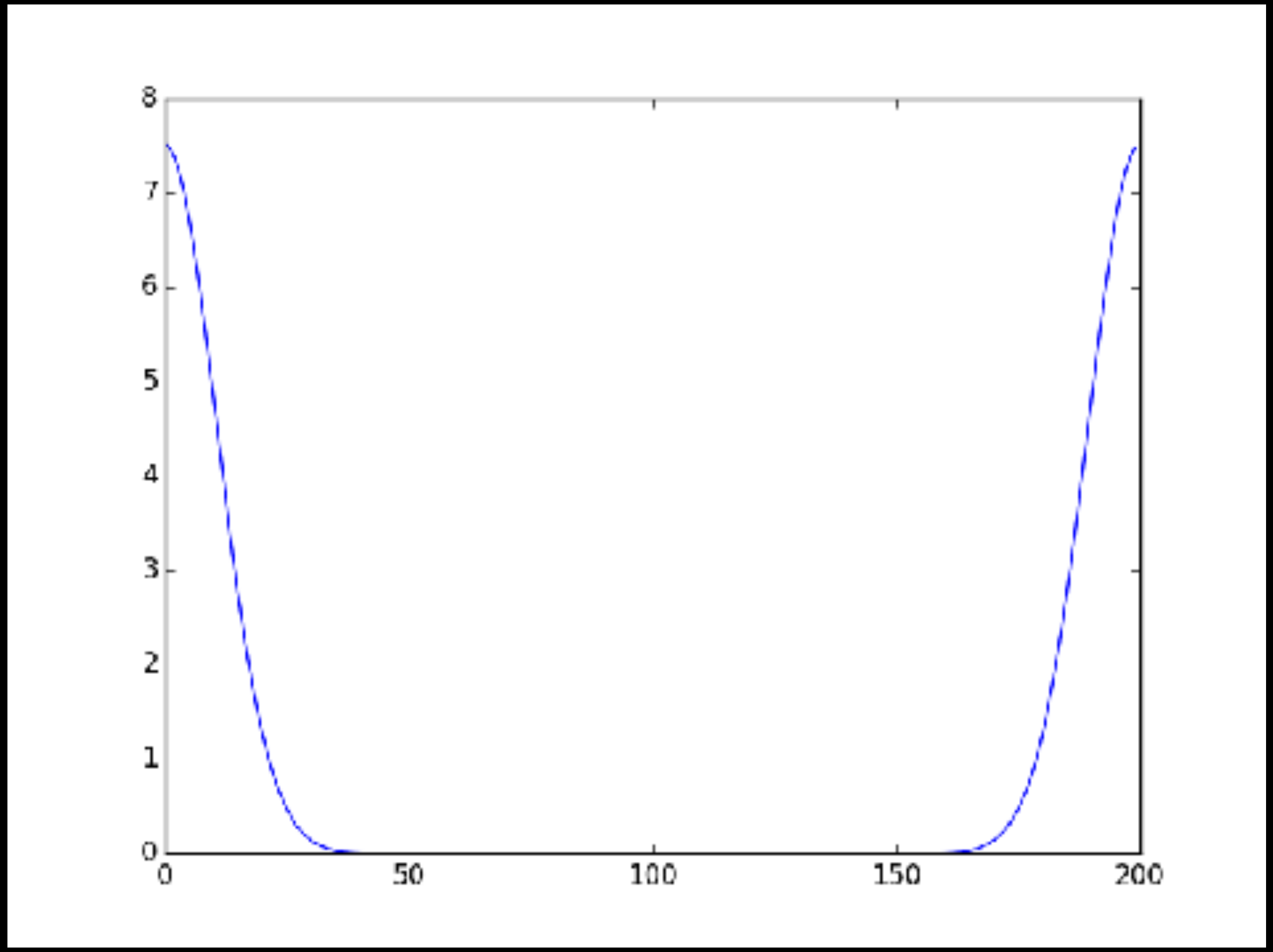
```
Jonathans-MacBook-Pro:lecture4 sievers$ python dft_columns.py
(8+0j)
(-4.28626379702e-16+4.4408920985e-16j)
(8+0j)
(3.44169137634e-15-1.11022302463e-15j)
Jonathans-MacBook-Pro:lecture4 sievers$
```

- Let’s check orthogonality, need complex #’s.
- numpy.complex(re,im) will make a complex #
- numpy functions usually defined for complex #’s.

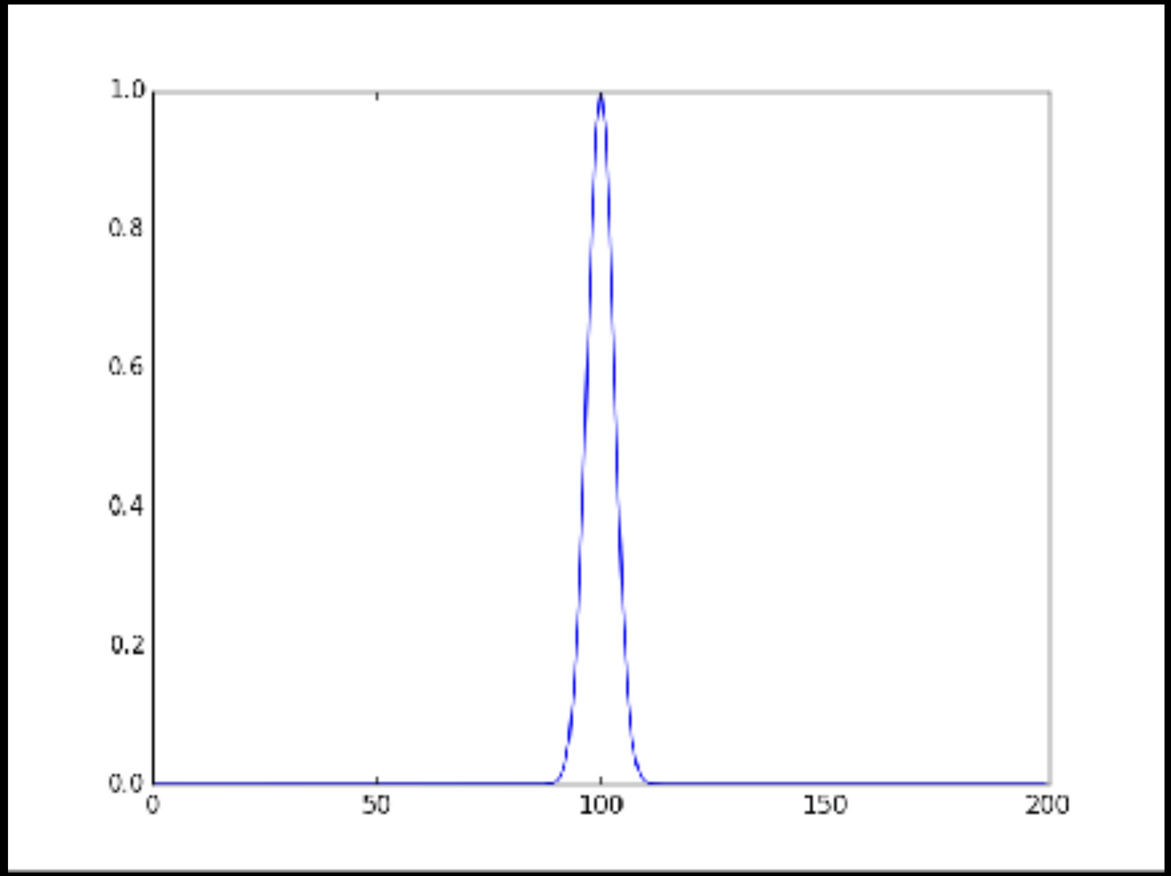
DFTs with Numpy

- Numpy has many Fourier Transform operations
- (for reasons to be seen) they are called *Fast* Fourier Transforms - FFT is one way of implementing DFTs.
- FFT's live in a submodule of numpy called FFT
- xft=numpy.fft.fft(x) takes DFT
- x=numpy.fft.ifft(x) takes inverse DFT
- Numpy normalizes such that f==fft(ifft(f))==ifft(fft(f))

DFT in Action




```
import numpy
from matplotlib import pyplot as plt

x=numpy.arange(-10,10,0.1)
y=numpy.exp(-0.5*x**2/(0.3**2))
yft=numpy.fft.fft(y)
plt.plot(numpy.abs(yft))
plt.savefig('gauss_dft')
plt.show()
```



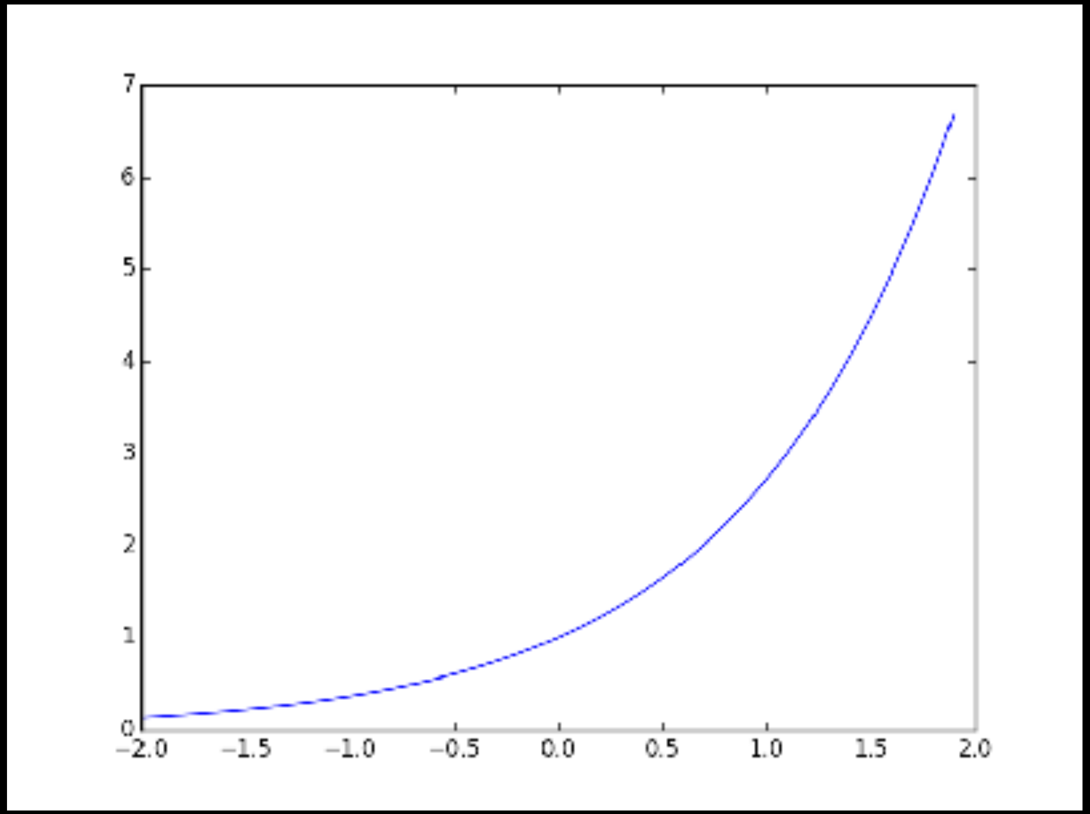
- Right: input Gaussian
- Top: DFT of the Gaussian

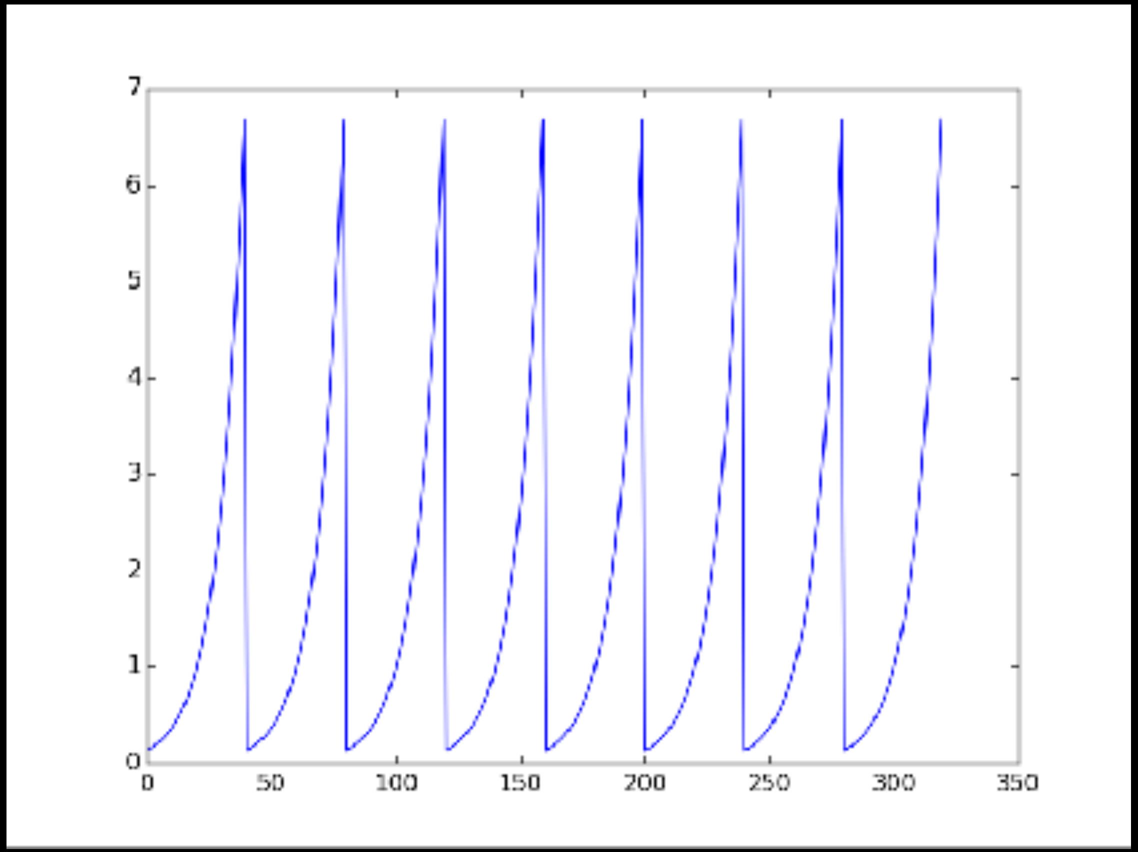
Periodicity

- $f(x)=\sum F(k) \exp(-2\pi ikx/N)$
- What is $f(x+N)$? $\sum F(k) \exp(-2\pi ik(x+N)/N)$
- $=\sum F(k) \exp(-2\pi ik)\exp(-2\pi ikx/N)$.
- $\exp(-2\pi ik)=1$ for integer k, so $f(x+N)==f(x)$
- DFT's are periodic - they just repeat themselves ad infinitum.

Periodicity

```
import numpy
from matplotlib import pyplot as plt

x=numpy.arange(-2,2,0.1)
y=numpy.exp(x)
plt.plot(x,y)
plt.savefig('fft_exp')
plt.show()
yy=numpy.concatenate((y,y))
yy=numpy.concatenate((yy,yy))
yy=numpy.concatenate((yy,yy))
plt.plot(yy)
plt.savefig('fft_exp_repeating')
plt.show()
```





- You may think you're taking top transform. You're not - you're taking the bottom one.
- In particular, jumps from right edge to left will strongly affect DFT

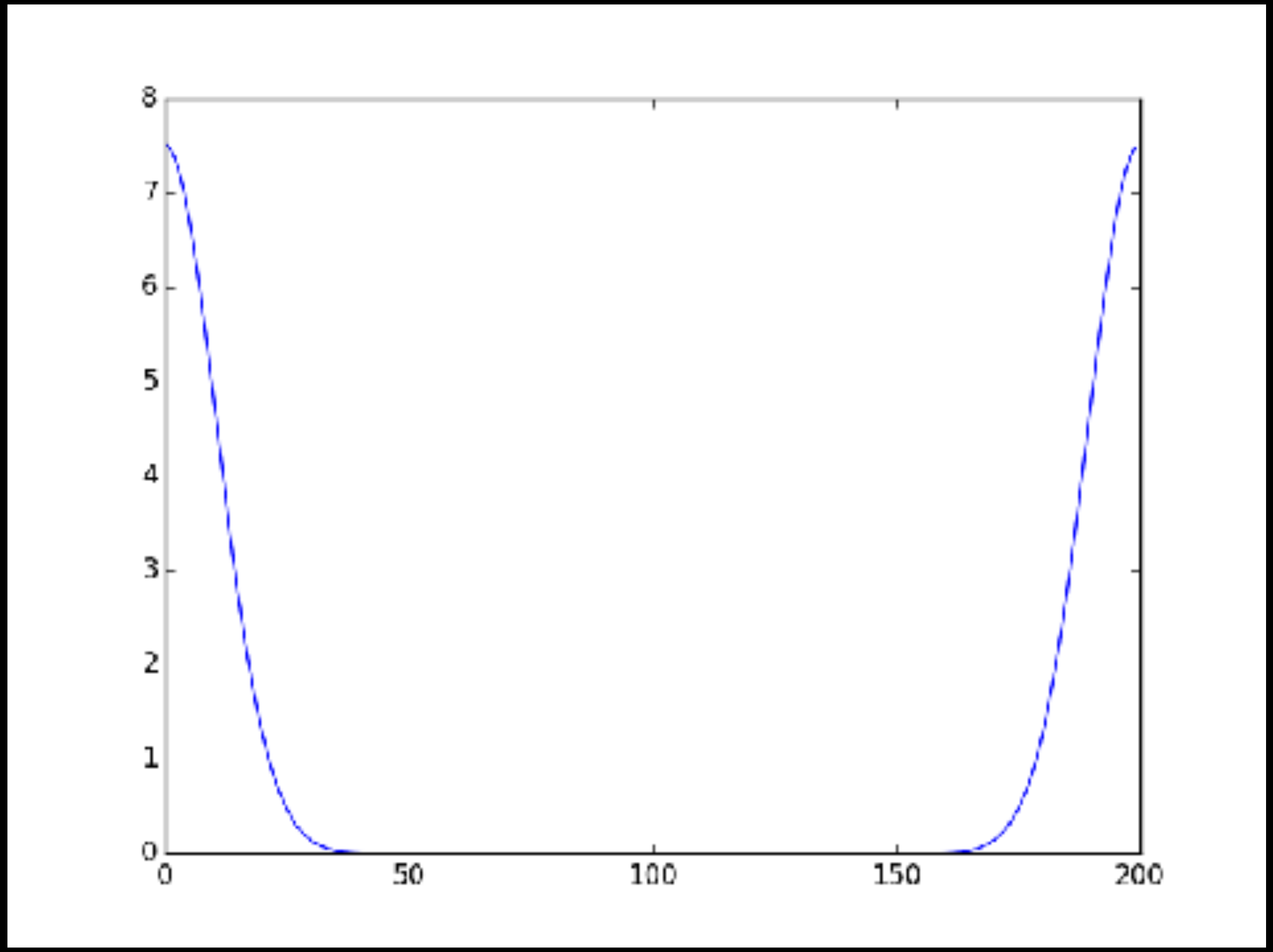
Aliasing

- $f(x)=\sum F(k)\exp(2\pi ikx/N)$
- What if I had higher frequency, $k>N$? let $k^*=k-N$ (i.e. k^* low freq.)
- $f(x)=\sum F(k)\exp(2\pi i(k^*+N)x/N=\sum F(k)\exp(2\pi ix)\exp(2\pi ik^*x/N)$
- for x integer, middle term goes away: $\sum F(k^*+N)\exp(2\pi ik^*x/N)$
- High frequencies behave exactly like low frequencies - power has been *aliased* into main frequencies of DFT.
- Always keep this in mind! Make sure samples are fine enough to prevent aliasing.

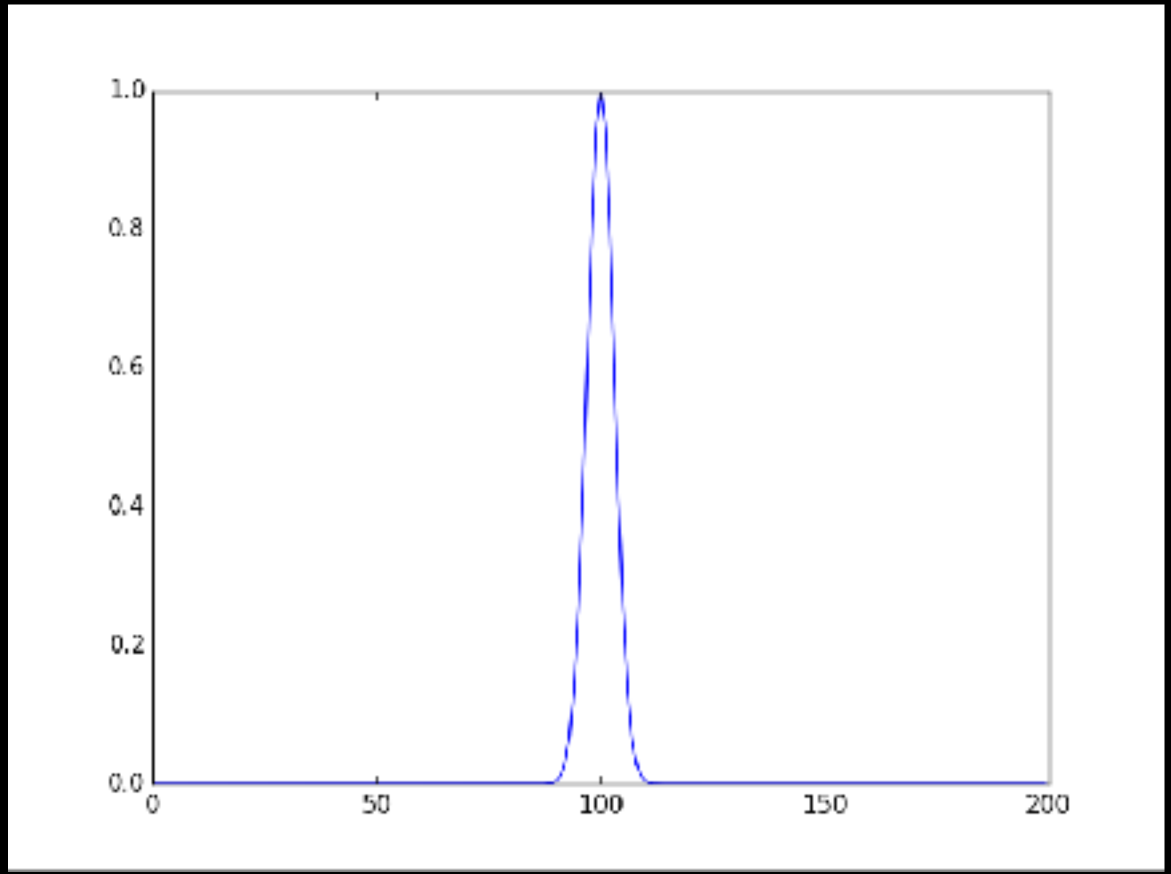
Negative Frequencies

- All frequencies that are N apart behave identically
- DFT has frequencies up to (N-1).
- Frequency (N-1) equivalent to frequency (-1). You will do better to think of DFT as giving frequencies (-N/2,N/2) than frequencies (0,N-1)
- *Sampling theorem*: if function is band-limited - highest frequency is ν - then I get full information if I sample *twice* per frequency, $dt=1/(2\nu)$. Factor of 2 comes from aliasing.

DFT in Action, Redux




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

x=numpy.arange(-10,10,0.1)
y=numpy.exp(-0.5*x**2/(0.3**2))
yft=numpy.fft.fft(y)
plt.plot(numpy.abs(yft))
plt.savefig('gauss_dft')
plt.show()
```



- FFT makes more sense now - negative frequencies have been aliased to high frequency.

Flipping

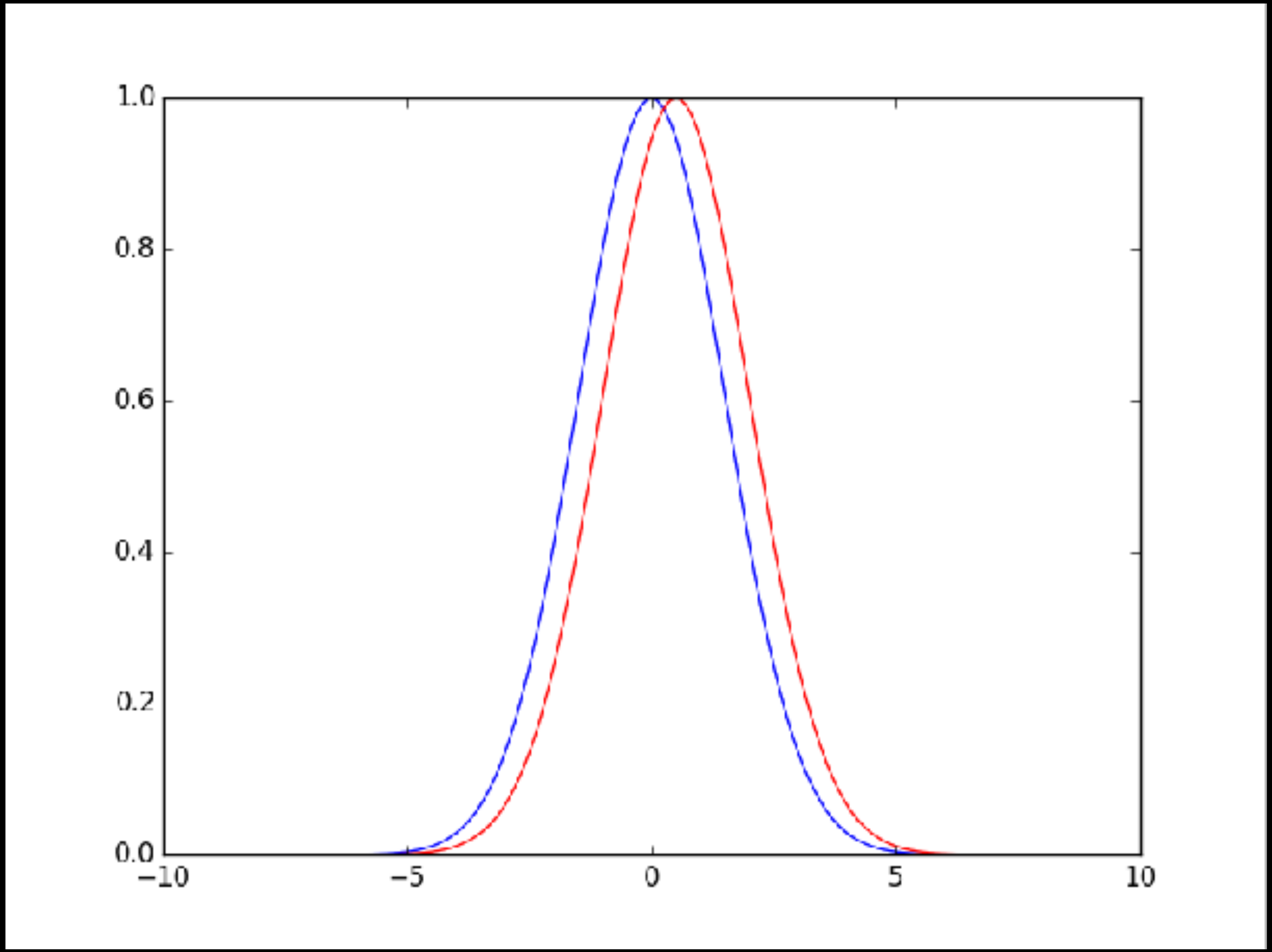
- What is DFT of $f(-x)$?
- $\sum f(-x)\exp(-2\pi ikx/N)$, $x^*=-x$, $\sum f(x^*)\exp(-2\pi ik(-x)/N)$
- $DFT(f(-x))=\sum f(x)\exp(2\pi ikx/N) = conj(F(k))$

Shifting

- What is FFT(x+dx)? $\sum f(x+dx)\exp(-2\pi ikx/N)$.
- $x^*=x+dx$: $F(k)=\sum f(x^*)\exp(-2\pi ik(x^*-dx)/N)$
- $F(k)=\exp(2\pi ikdx/N)\sum f(x^*)\exp(-2\pi ikx^*/N)$
- So, just apply a phase gradient to the DFT to shift in x

Shifting Example

```
import numpy
from matplotlib import pyplot as plt

x=numpy.arange(-10,10,0.1)
y=numpy.exp(-0.5*x**2/(1.5**2))
N=x.size
kvec=numpy.arange(N)
yft=numpy.fft.fft(y)
J=numpy.complex(0,1)
dx=5.0;
yft_new=yft*numpy.exp(-2*numpy.pi*J*kvec*dx/N)
y_new=numpy.real(numpy.fft.ifft(yft_new))
plt.plot(x,y)
plt.plot(x,y_new,'r')
plt.savefig('shifted_gaussian')
plt.show()
```



Real Data Symmetry

- If I know F(k), what is F(N-k) if f(x) is real?
- F(N-k)=F(-k) (from alias theorem)
- F(-k)=∑f(x)exp(-2πi(-k)x/N. let x*=-x
- F(-k)=∑f(-x*)exp(2πikx*/N) = conj(F(k)) by flipping
- So, if f(x) is real, F(k)=conj(F(N-k))
- If N even, F(N/2)=conj(F(N/2)), so F(N/2) must be real.

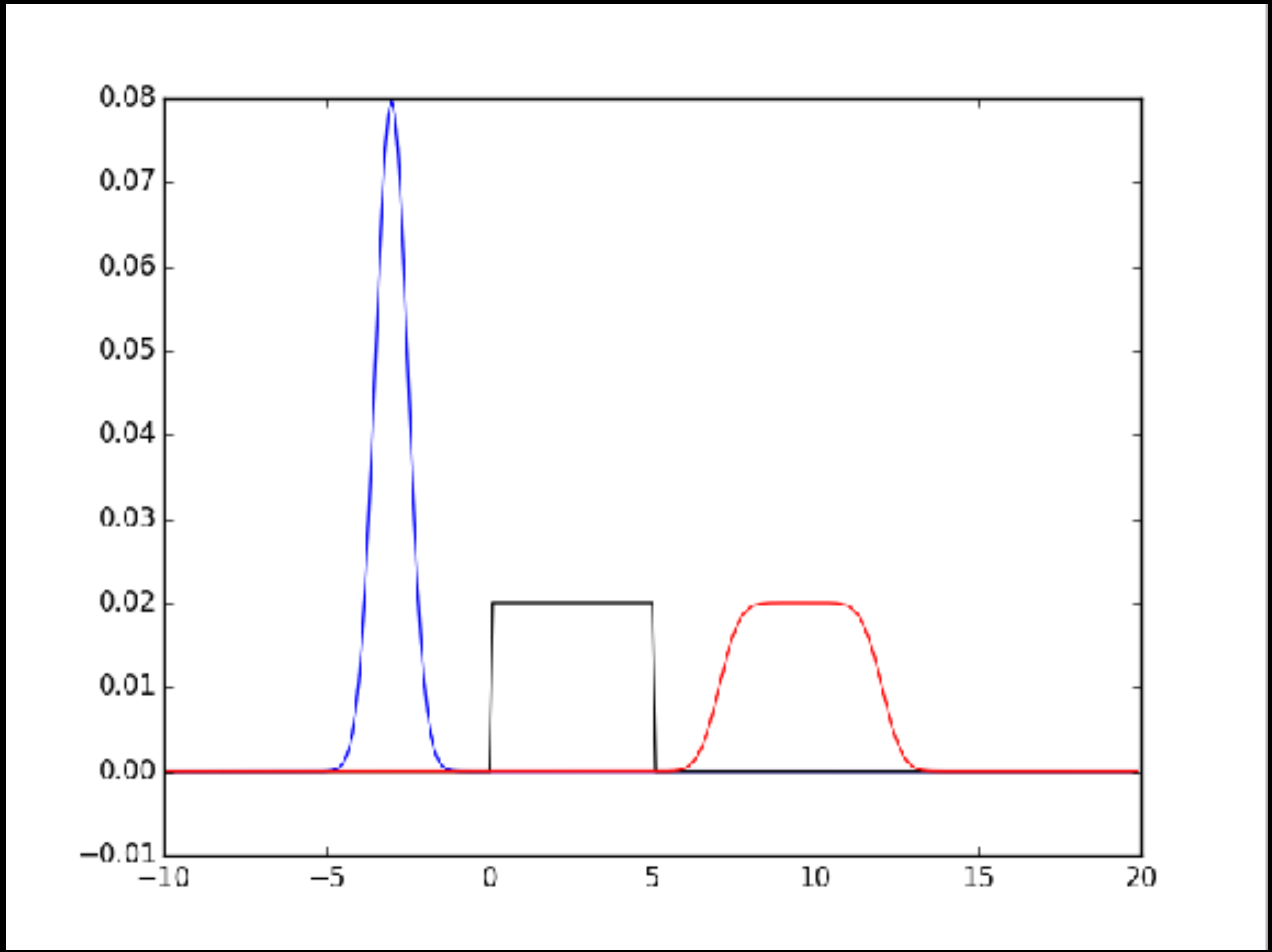
```
>>> x=numpy.random.randn(8)
>>> xft=numpy.fft.fft(x)
>>> for xx in xft:
...   print xx
...
(-4.53568815727+0j)
(-0.174046761579+2.08827239558j)
(2.15348308858+2.32162497273j)
(-0.423040513854-3.72126858798j)
(2.75685372591+0j)
(-0.423040513853+3.72126858798j)
(2.15348308858-2.32162497273j)
(-0.174046761579-2.08827239558j)
>>>
```

Convolution Theorem

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

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()
```



Aliasing

- $f(x)=\sum F(k)\exp(2\pi ikx/N)$
- What if I had higher frequency, $k>N$? let $k^*=k-N$ (i.e. k^* low freq.)
- $f(x)=\sum F(k)\exp(2\pi i(k^*+N)x/N=\sum F(k)\exp(2\pi ix)\exp(2\pi ik^*x/N)$
- for x integer, middle term goes away: $\sum F(k^*+N)\exp(2\pi ik^*x/N)$
- High frequencies behave exactly like low frequencies - power has been *aliased* into main frequencies of DFT.
- Always keep this in mind! Make sure samples are fine enough to prevent aliasing.

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 $n\log n$.

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

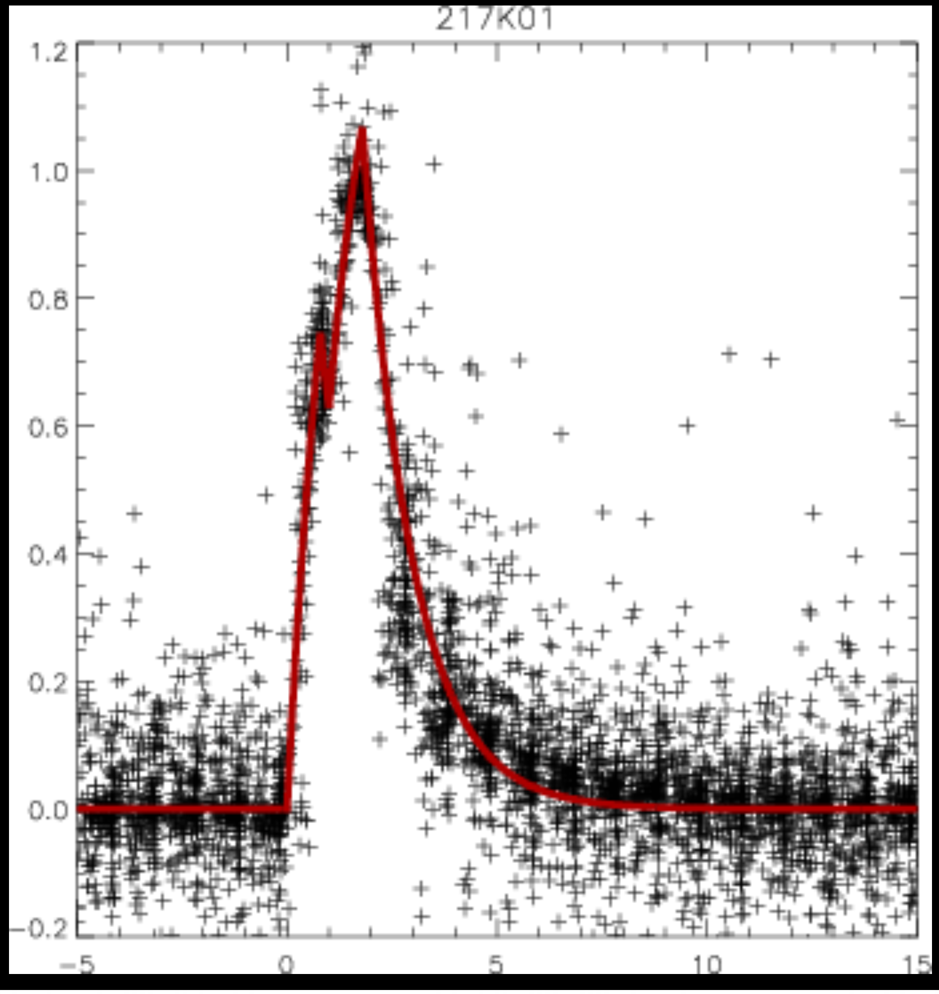
- Should you write your own FFT code? (no)
- Should you understand what is going on under the hood? (yes)

```
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
```

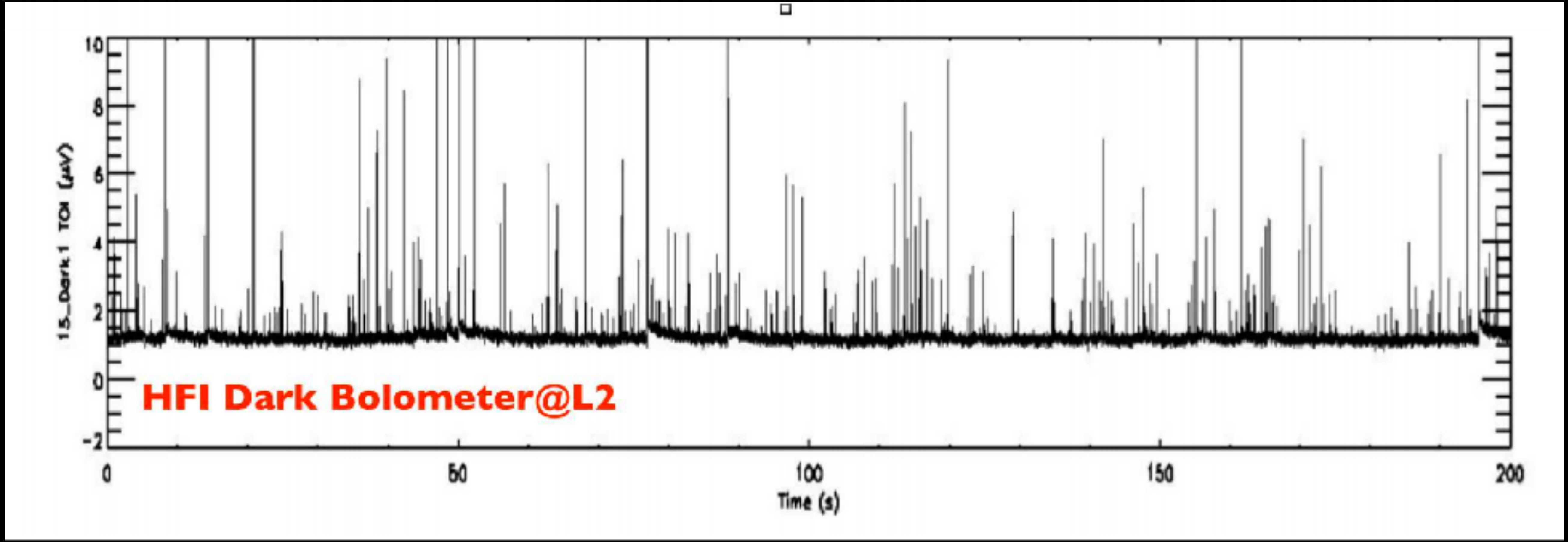
```
>>> execfile('time_ffts.py')
Reference time was  0.00335788726807
Extending by one increased time by a factor of  2594.13178074
Extending by two increased time by a factor of  5.54963078671
Extending by  14 increased time by a factor of  1.8124112468
```

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

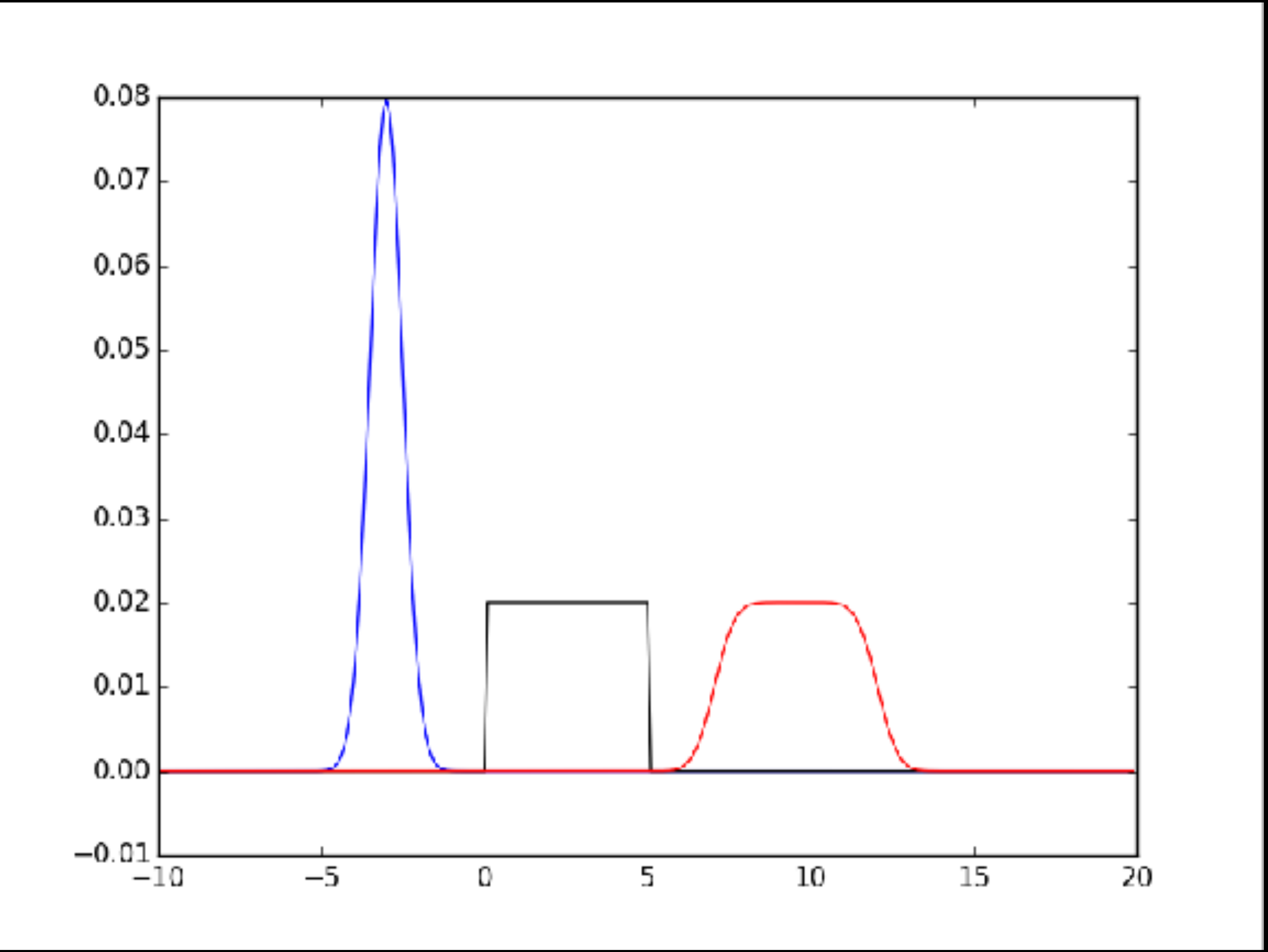


Cosmic Rays from Planck Satellite



Convolution Example

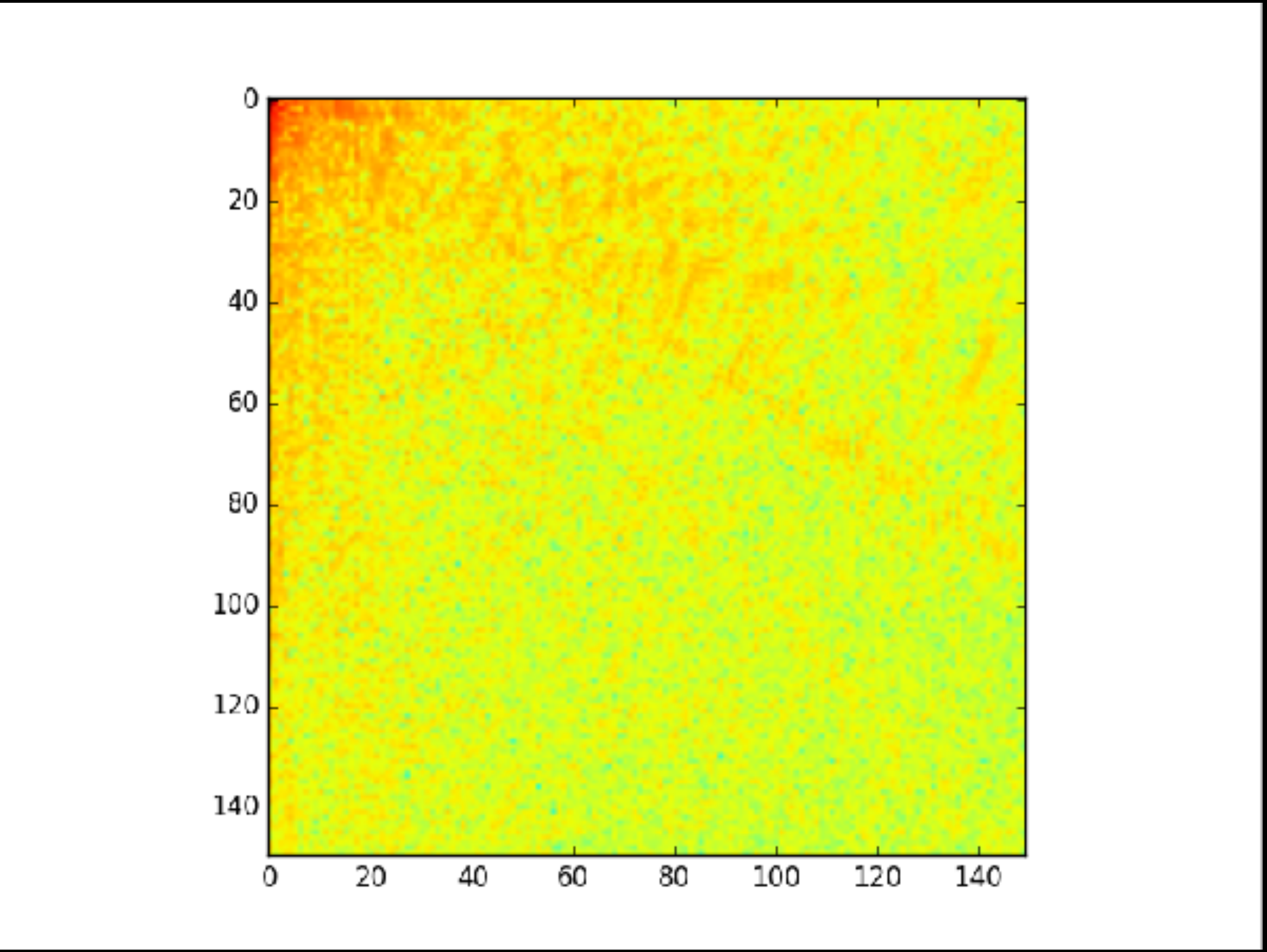
```
from numpy import arange,exp,real
from numpy.fft import fft,ifft
from matplotlib import pyplot as plt
def conv(f,g):
    ft1=fft(f)
    ft2=fft(g)
    return real(ifft(ft1*ft2))

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

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

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

othed_map=numpy.zeros(meerkat.shape)
smoothed_map=numpy.zeros(meerkat.shape)
x_smooth=3.5
x_restore=4
 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

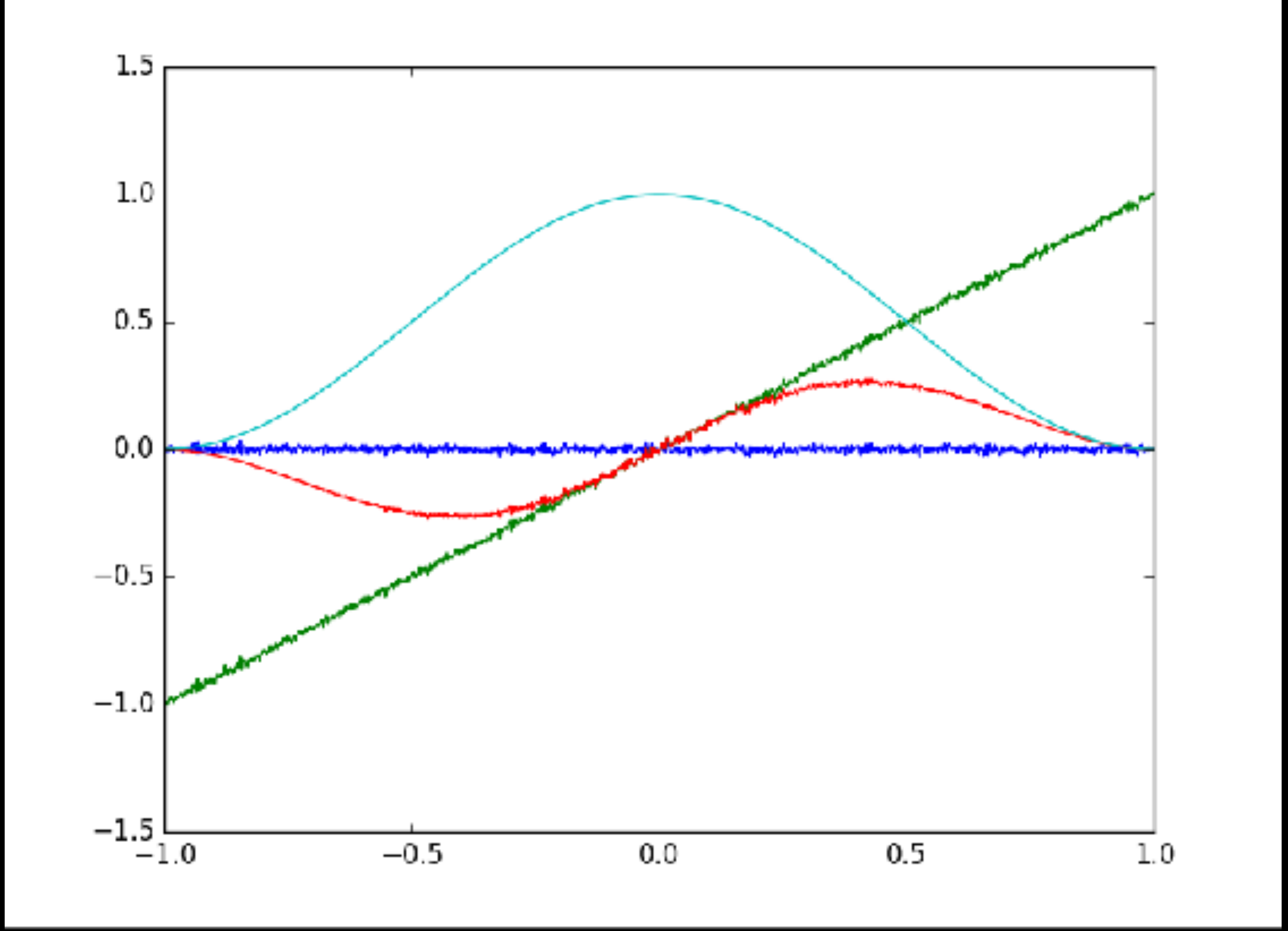
- Jumps around edges cause high frequency power in FFTs. This is a bad thing.
- Standard solution: multiply by a window that goes to zero (or some very small value) at edges.
- There are many possible windows, depending on what you want to do: Hamming, Hanning, cos… 28 listed on wikipedia page.
- If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

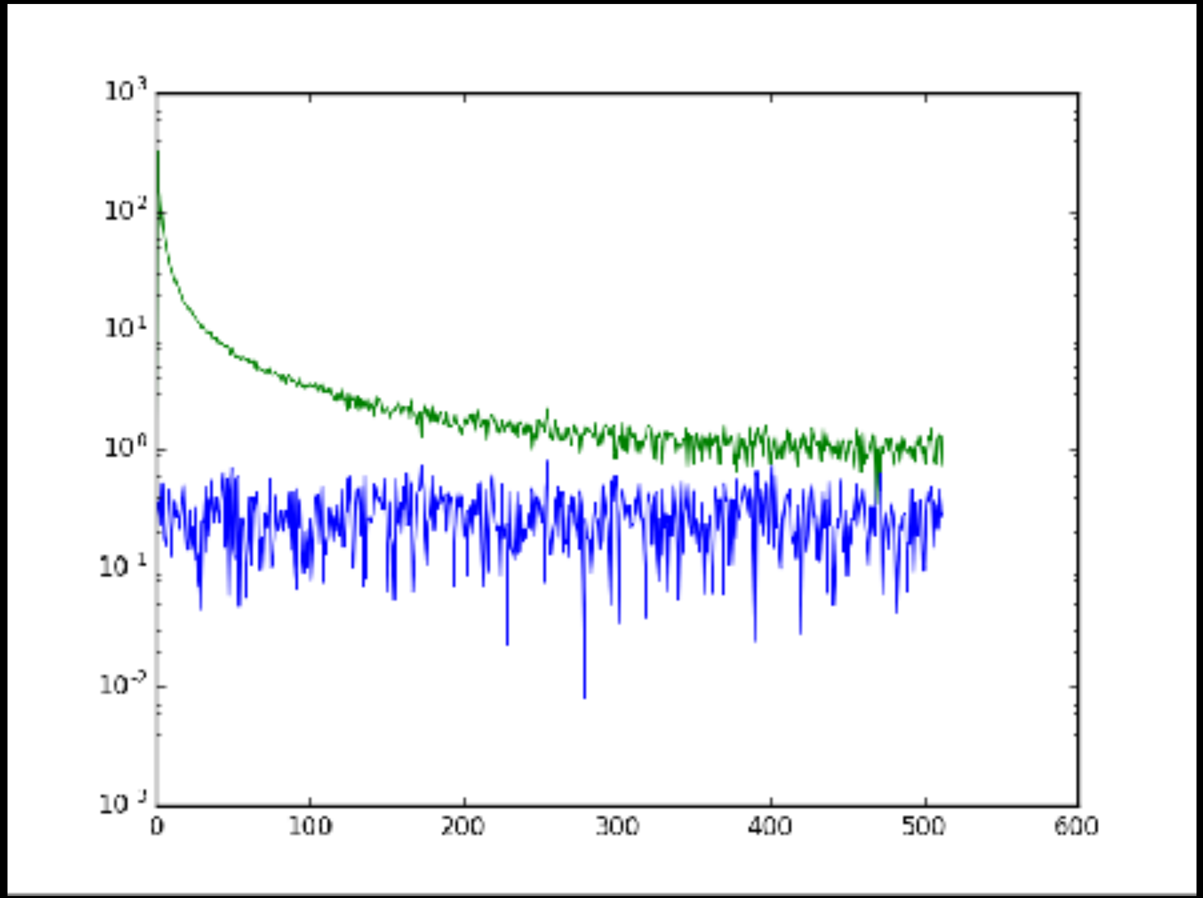
What does FT look like?

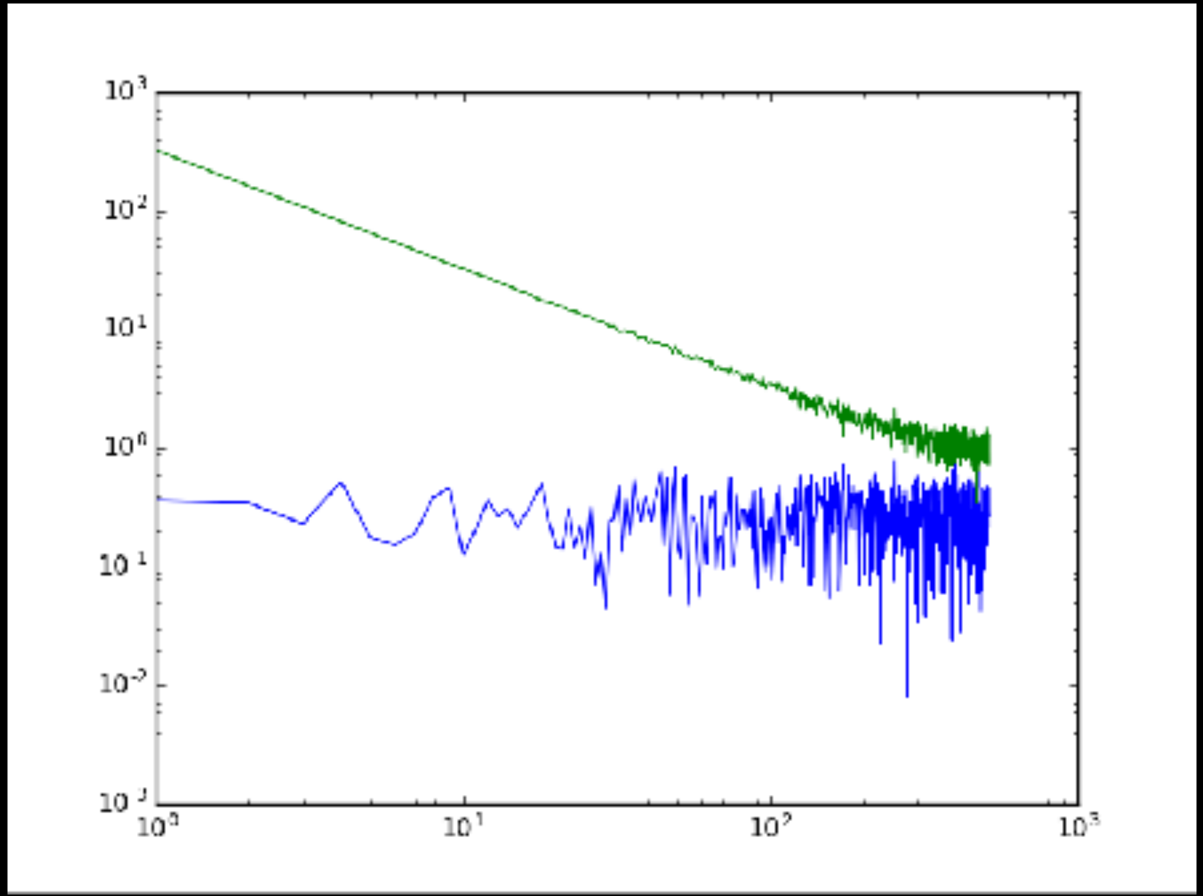



```
import numpy
from matplotlib import pyplot as plt
plt.ion();

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?

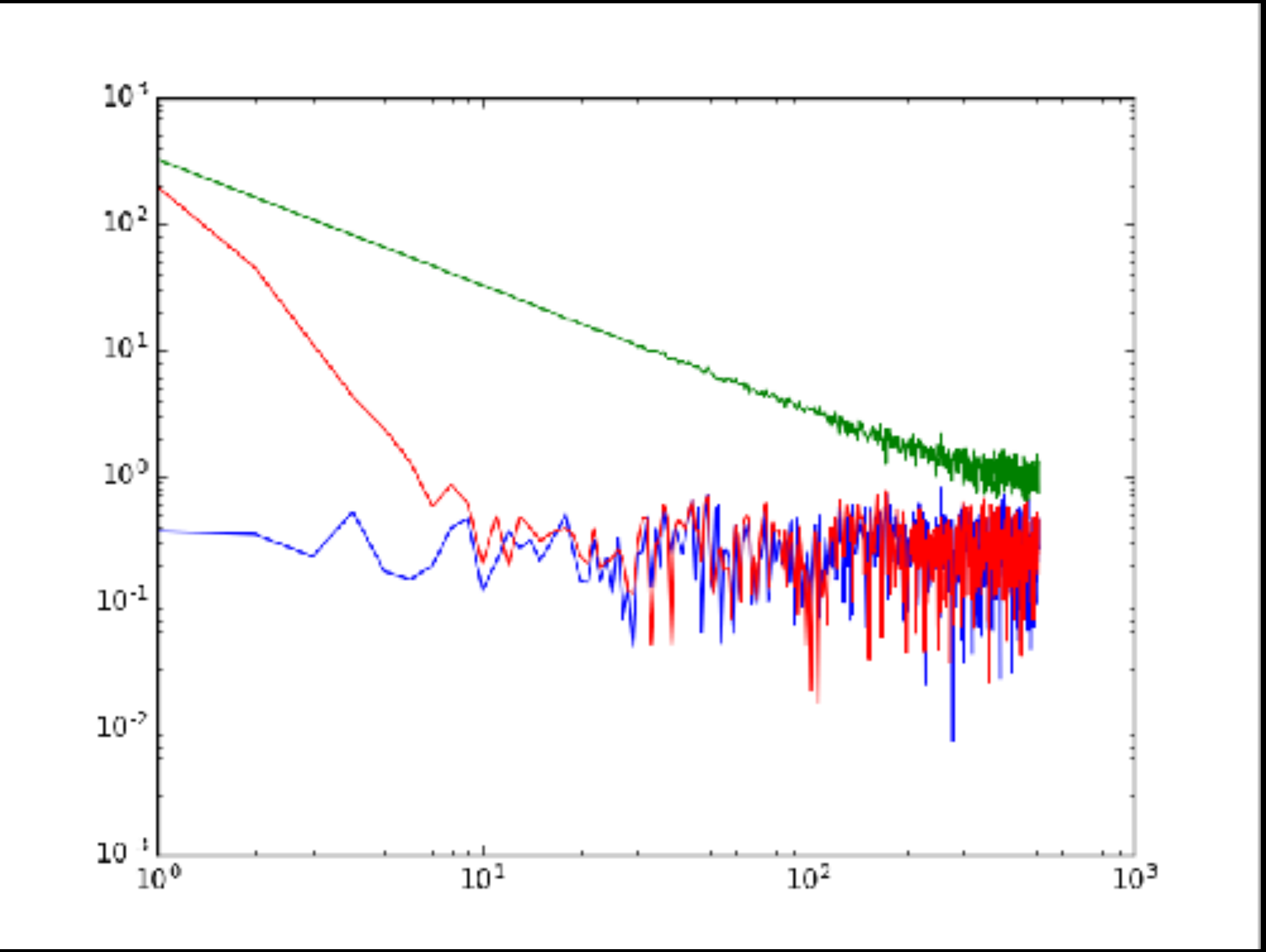




Adding Window

- 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')
```