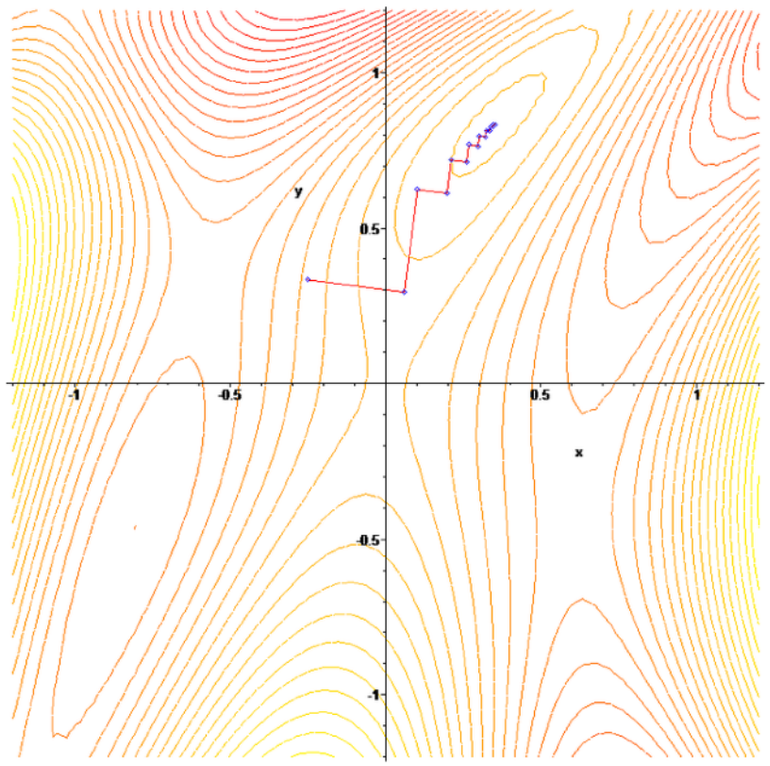
Lecture 7

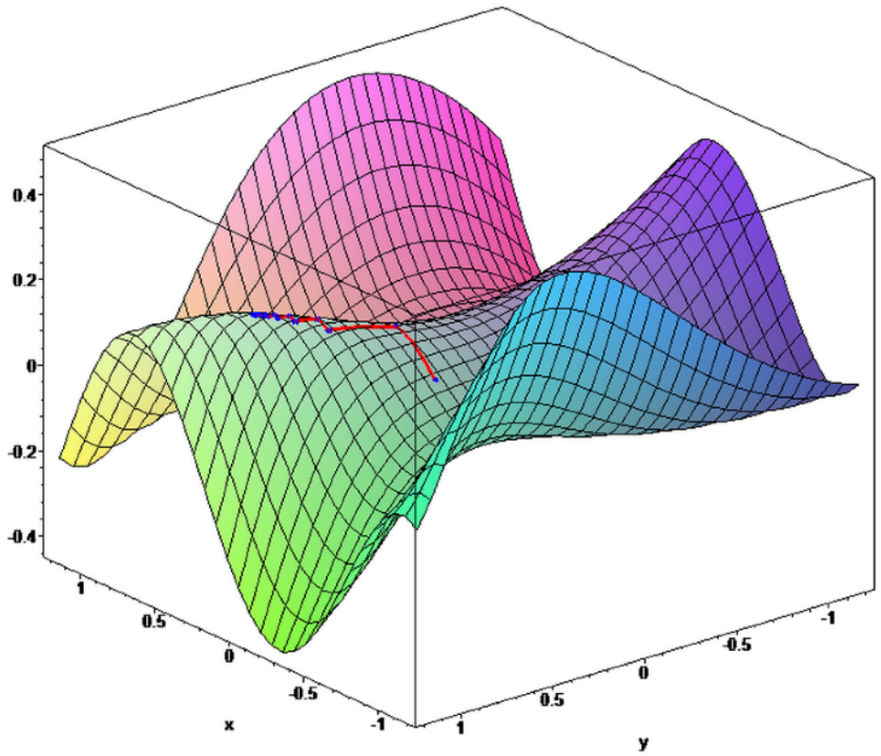
Non-linear least-squares w/stationary noise, matched filters.

Steepest Descent

The "Zig-Zagging" nature of the method is also evident below, where the gradient ascent method is applied to $F(x,y) = \sin\left(\frac{1}{2}x^2 - \frac{1}{4}y^2 + 3\right)\cos(2x + 1 - e^y)$.

e gradient descent algorithm in action. (1: contour)





From wikipedia. Zigagging is inefficient.

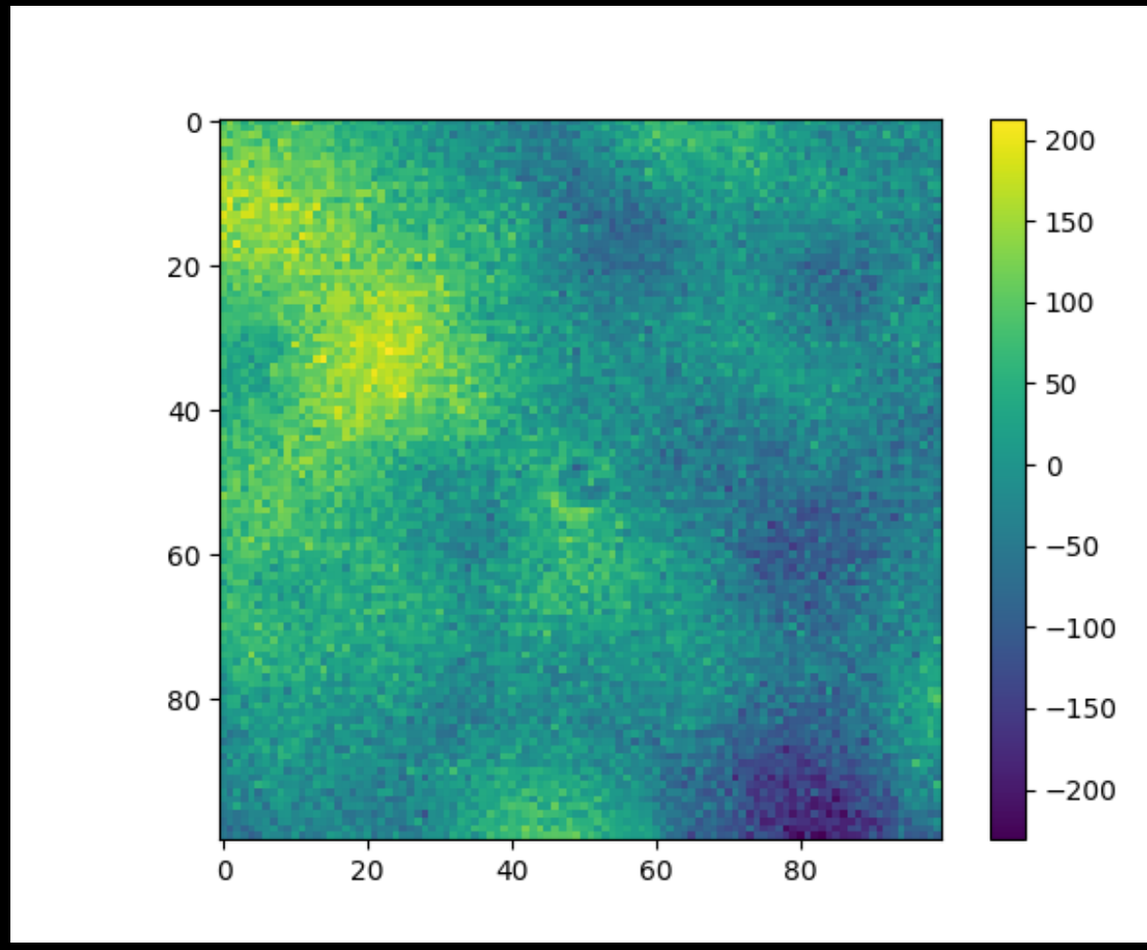
How to Implement

- Start with a guess for the parameters: m_0.
- Calculate model $A(m_0)$ and local gradient A_m. Gradient can be done analytically, but also often numerically.
- Solve linear system $A_m^T N^{-1} A_m \, \delta m = A_m^T N^{-1} r$
- Set $m_0 \rightarrow m_0 + \delta m$.
- Repeat until δm is "small". For χ^2, change should be << 1 (why?).

Levenbert-Marquardt

- Newton's method can sometimes (often) fail, if the step is too large.
- In such cases, the curvature estimate is not a good approximation over the region we care about.
- We want a shorter step that is more downhill. One way to do that is to introduce a parameter λ, and increase the diagonal of the LHS by a factor of $(1+\lambda)$.
- If $\lambda=0$, we have Newton's method. If λ is large, LHS inverse is small, so step is short. It is also guaranteed to be in a downhill direction.
- Levenberg-Marquardt: take a step. If step fails, increase λ and try new, shorter step. If step succeeds, accept the step and decrease λ.
- Lots of ways to do this, NR method is silly. If $\lambda << 1$, just make it 0. If you ever fail, set λ to be order unity right away.

ACT Map Example



- Look at fit_act_flux_newton.py

- This implements numerical derivatives w/ Newton's method to fit a Gaussian (including sigma, dx, dy) to the ACT data.

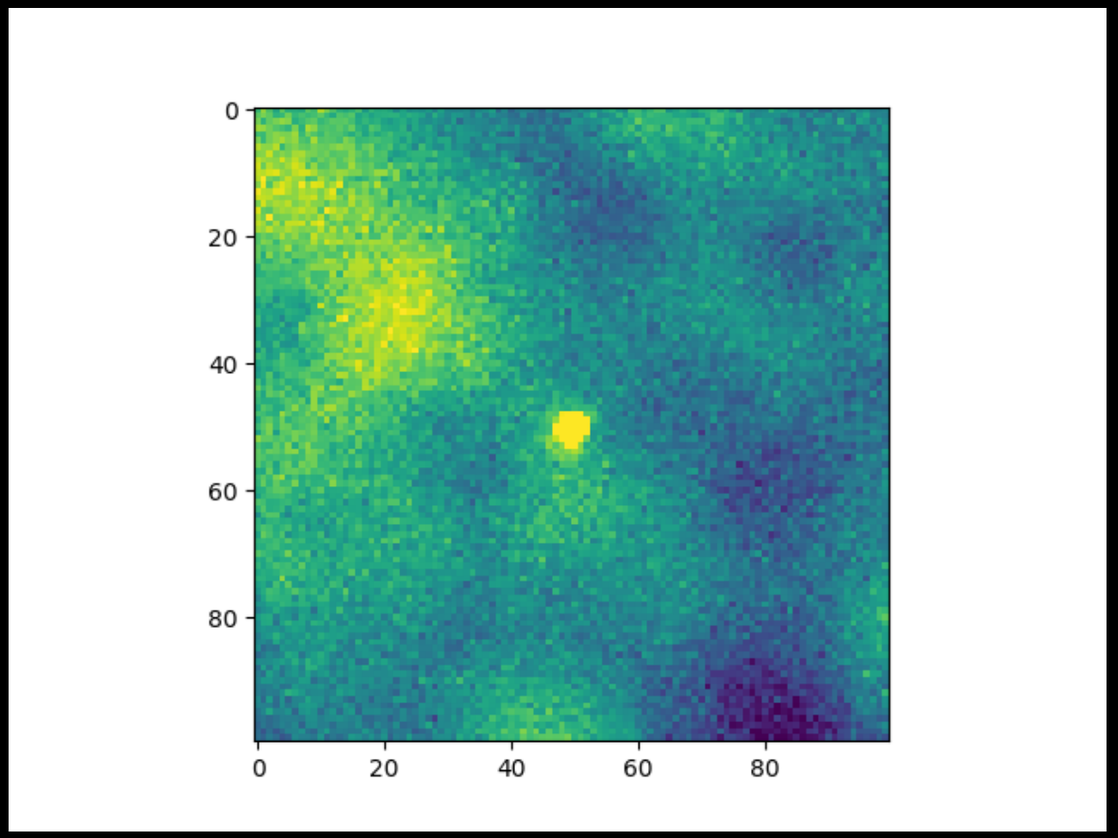
- How should we estimate the noise, and hence the parameter uncertainties?

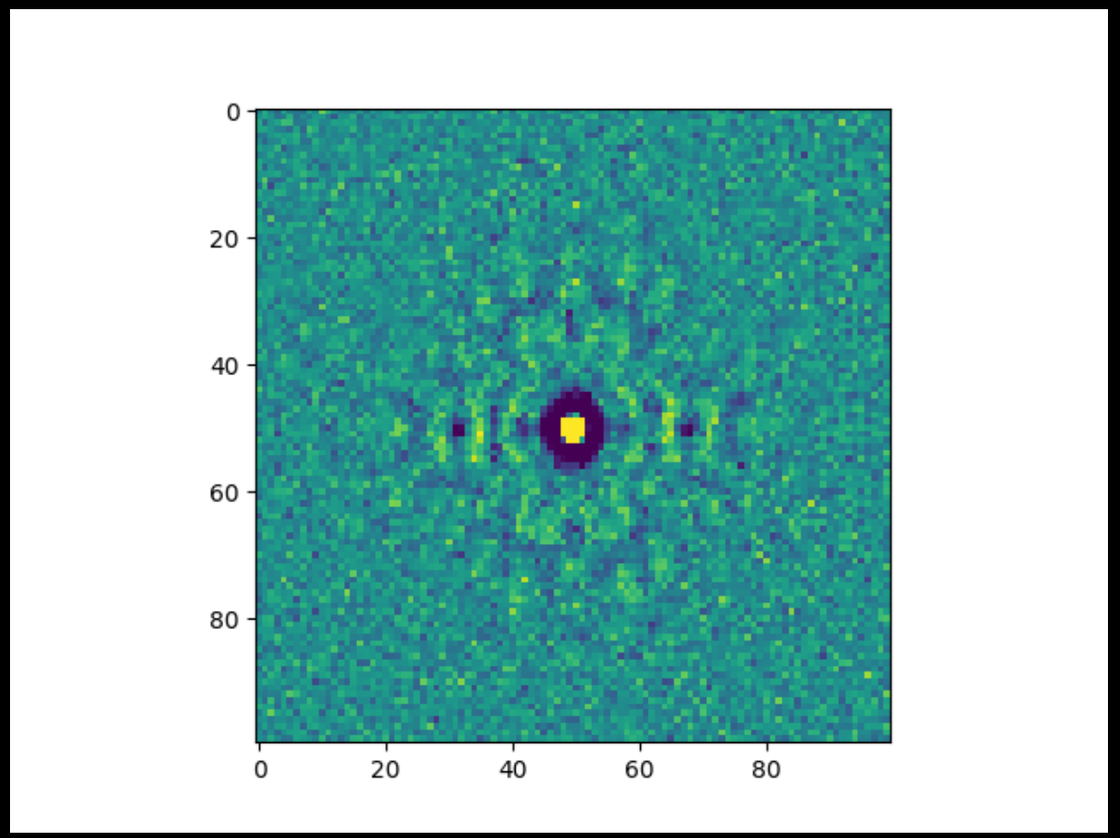
- Think about how we would do this accounting for the correlated noise?

ACT w/ Correlated Noise

- Now we have everything we need to get a real error bar!
- We will assume noise is stationary inside patch
- Noise model will be $|FT|^2$, with some smoothing.
- LHS: $A^T N^{-1} A$. For $N^{-1}A$, we can FT A, then divide by our (smoothed) FT of the patch, and FT back.
- RHS: $A^T N^{-1} d$. We could FT d, but we already have $N^{-1}A$, so $(N^{-1}A)^T d$ is just the dot product of something we already have with the data. Sanity check is that $A^T(N^{-1}d)$ gives same answer.
- NB - there are some normalization factors to do with FT, plus I make 4 reflected copies to avoid edge effects. (More discussion soon…)

Output: left original, right $N^{-1}d$






```
best-fit improvement is  9258.461141888942
best-fit amplitude is  3131.6195091919985
rhs two ways is  45247.759297511904 45247.759297511904
amp/errs are  0.9799892049119797 0.004653847062334384 0.009947632452080320 0.4678
3464153434196
in temperature:  3068.953312899911 14.574078253202229
```

Output: The full noise model has 2x smaller errorbar than pretending noise is white. The fitting has figured out that noise is on large scales, but we have signal on small scales and taken advantage of that.

Matched Filters

- Let's say we know what a signal looks like, but we don't know *where* to look for it.
- In 1D, this would be $A(x-t)^T N^{-1} A(x-t)m = A(x-t)^T N^{-1} d(x)$
- Iff noise is stationary, I can let $x'=x-t$, then N stays the same. $A^T N^{-1} Am = A^T N^{-1} d(x'+t)$. LHS is just a number, RHS I get $\sum (N^{-1}A) d(x'+t)$.
- This is a cross-correlation, and can be done efficiently in Fourier space.
- Called matched filters, but never forget underneath you are doing a least-squares fit for a model against every location in the data.

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?

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)
```

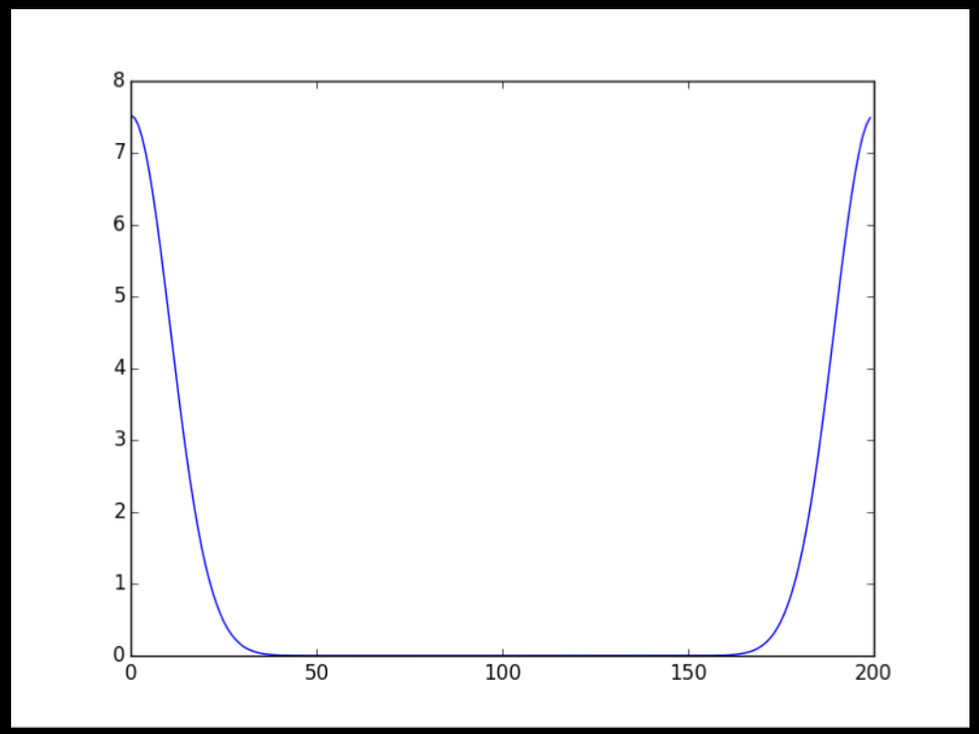
```
Jonathans-MacBook-Pro:lecture4 sievers$ python dft_columns.py
(8+0j)
(-4.28626379702e-16+4.4408920985e-16j)
(8+0j)
(3.44169137634e-15-1.1102302463e-15j)
Jonathans-MacBook-Pro:lecture4 sievers$
```

- 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

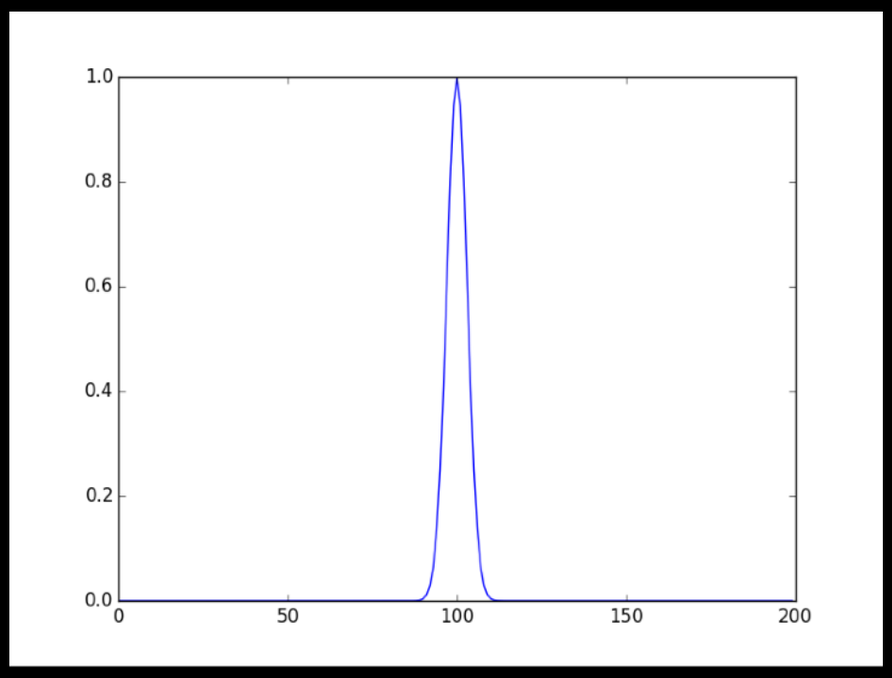
- Numpy has many Fourier Transform operations
- (for reasons to be seen) they are called *Fast* Fourier Transforms - FFT is one way of implementing DFTs.
- FFT's live in a submodule of numpy called FFT
- xft=numpy.fft.fft(x) takes DFT
- x=numpy.fft.ifft(x) takes inverse DFT
- Numpy normalizes such that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

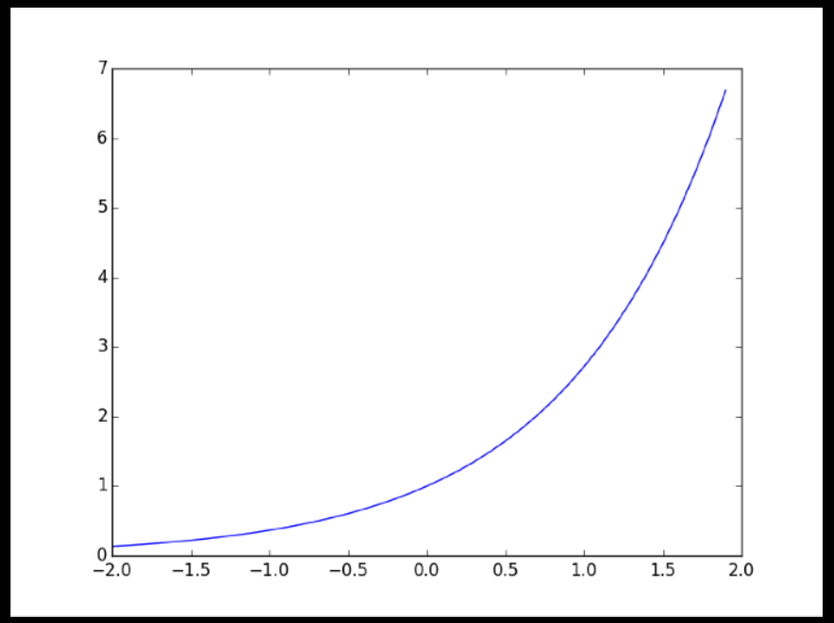
Periodicity

- $f(x)=\sum F(k)\ \exp(2\pi ikx/N)$
- 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.

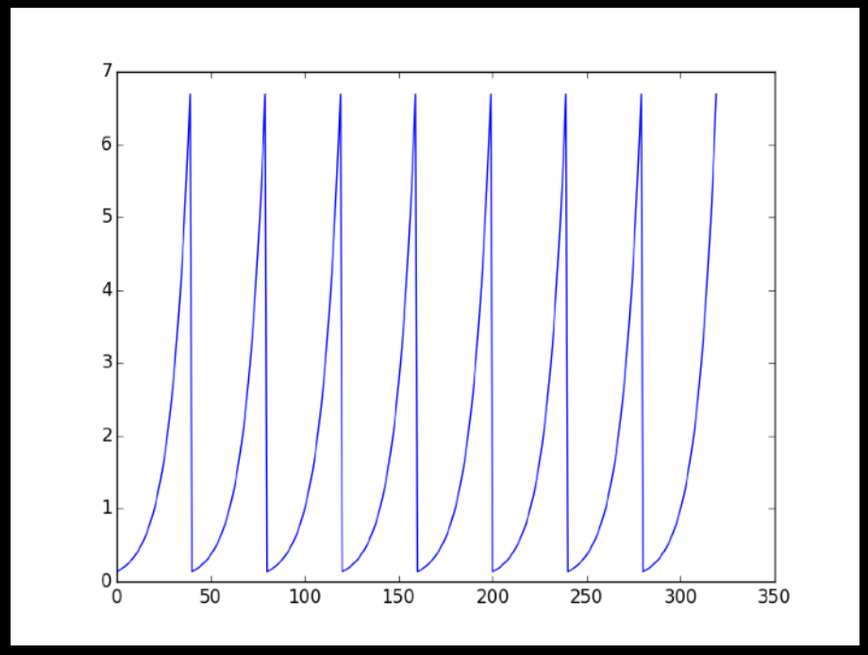
Periodicity

```
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



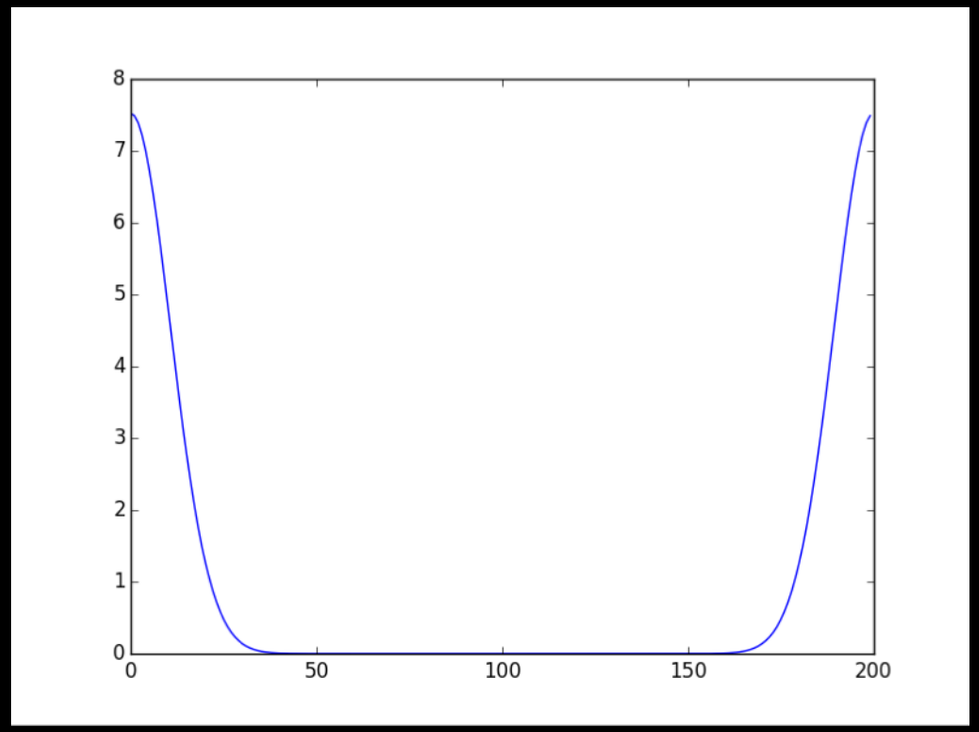
Aliasing

- $f(x)=\sum F(k)\ \exp(2\pi i k x/N)$
- What if I had higher frequency, k>N? let $k^*=k-N$ (i.e. k^* low freq.)
- $f(x)=\sum F(k)\ \exp(2\pi i (k^*+N)x/N=\sum F(k)\exp(2\pi i x)\exp(2\pi i k^* x/N)$
- for x integer, middle term goes away: $\sum\ F(k^*+N)\exp(2\pi i k^* x/N)$
- 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

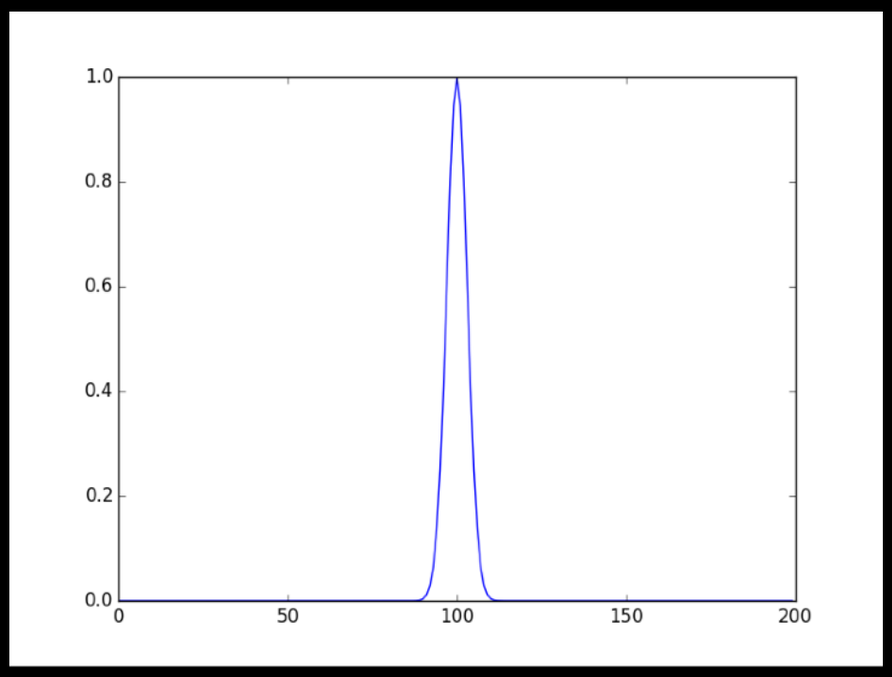
- All frequencies that are N apart behave identically
- DFT has frequencies up to (N-1).
- Frequency (N-1) equivalent to frequency (-1). You will do better to think of DFT as giving frequencies (-N/2,N/2) than frequencies (0,N-1)
- *Sampling (Nyquist) theorem*: if function is band-limited - highest frequency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

DFT in Action, Redux




```
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

Convolution Theorem

- Convolution defined to be conv(y)=f⊗g==∫f(x)g(y-x)dx
- $\sum_x \sum F(k)\exp(2\pi ikx) \sum \text{conj}(G(k'))\exp(2\pi ik'x)\exp(-2\pi ik'y/N)$
- Reorder sum: $\sum\sum F(k)\text{conj}(G(k'))\exp(-2\pi ik'y/N)\sum_x \exp(2\pi i(k+k')x)$
- equals zero unless k'==-k. Cancels one sum, conjugates G
- $f\otimes g=\sum F(k)G(k)\exp(2\pi iky/N)=\text{ift}(\text{dft}(f)*\text{dft}(g))$
- So, to convolve two functions, multiply their DFTs and take the IFT
- Cross correlation similar - we get f(x+y) instead, and we get $\text{corr}(f,g)=\text{IDFT}(FG^*)$ (*i*.e. take the conjugate of one DFT).

Matched Filters Redux

- With convolution/correlation theorems in hand, we can now efficiently do a least-squares fit at every point in our data.

- FT the signal. FT the data. Conjugate one, multiply, and IDFT. If you want normalized amplitude, divide by sum of model^2.

- Adding correlated noise trivial! We've already FFT'd, so after we FT the data/model, we divide by the noise power spectrum (and similar for the LHS).

- NB - easy to get your factors of N, sqrt(N) etc. wrong, so do be careful if you want properly normalized…

- Often, we don't know exactly what the noise is, in which case make the RHS using your best guess, but then estimate noise by looking at scatter in MF output.