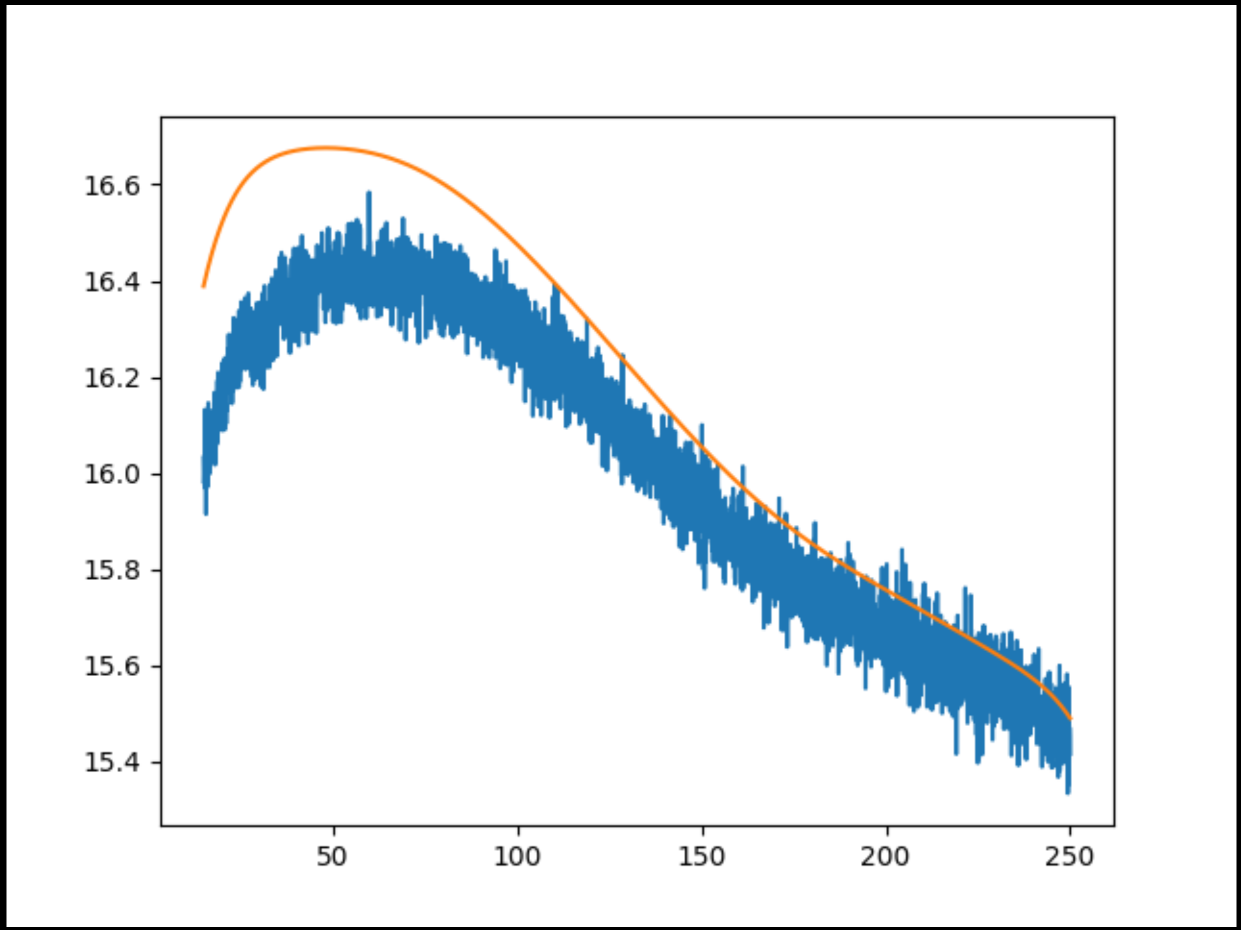
Lecture 4

SVD/Legendre fitting, correlated noise, non-linear least-squares

Higher Order

- More polynomials should be more accurate, right?
- Let's fit same problem but now to 10th order.
- Other than changing order, code is identical.
- What happens now?



Condition # and Roundoff

- Recall that the eigenvalues of a symmetric matrix are real, and the eigenvectors are orthogonal. So, $(A^TN^{-1}A)$ can be re-written $V^T\Lambda V$, where Λ is diagonal and V is orthognal (so $V^{-1}=V^T$).

- $(ABC)^{-1}=C^{-1}B^{-1}A^{-1}$, so inverse$=V^{-1}\Lambda^{-1}(V^T)^{-1}=V^T\Lambda^{-1}V$.

- If a bunch of eigenvalues are really small, they will be huge in the inverse. Double precision numbers are good to ~16 digits, so if spread gets bigger than 10^{16}, we'll lose information in the inverse.

- Ratio of largest to smallest eigenvalue is called the condition number. If it is large, matrices are ill-conditioned, and will present problems.

Condition # of Polynomial Matrices

- Condition # quickly blows up. So, we should have expected problems.

```
import numpy
def get_poly_mat(t,npoly):
    mat=numpy.zeros([t.size,npoly])
    mat[:,0]=1.0
    for i in range(1,npoly):
        mat[:,i]=t*mat[:,i-1]
    mat=numpy.matrix(mat)
    return mat

if __name__=='__main__':
    t=numpy.arange(-5,5,0.1)
    for npoly in numpy.arange(5,30,5):
        mat=get_poly_mat(t,npoly)
        mm=mat.transpose()*mat
        mm=mm+mm.transpose() #bonus symmetrization
        e,v=numpy.linalg.eig(mm)
        eabs=numpy.abs(e)
        cond=eabs.max()/eabs.min()
        print repr(npoly) + ' order poynomial matrix has condition number ' + repr(cond)
```

```
>>> execfile('cond_example.py')
5 order poynomial matrix has condition number 158940.69399024552
10 order poynomial matrix has condition number 2366966250887.5864
15 order poynomial matrix has condition number 2.722363799692467e+19
20 order poynomial matrix has condition number 2.2708595871810382e+25
25 order poynomial matrix has condition number 7.8912167454722334e+31
>>>
```

One Possibility: SVD

- Take noiseless case. Then solving $A^TAm=A^Tx$.
- Singular value decomposition (SVD) factors matrix $A=USV$, where S is diagonal, and U and V are orthogonal, and V is square. For symmetric, $U=V$, S=eigenvalues, but SVD works for any matrix.
- Solutions: $(USV)^TUSVm=(USV)^Tx$. $V^TSU^TUSVm=V^TSU^Tx$
- U^TU=identity, so cancels. $V^TS^2Vm=V^TSU^Tx$. S^2 squares the condition number, so that was bad. We can analytically cancel left-hand V and one copy of S: $SVm=U^Tx$. Then $m=V^TS^{-1}U^Tx$
- NB - some packages (e.g. matlab) have $A=USV^T$. Always sanity check.
- NB2 - this can be done even faster with QR

SVD Code

- Here's how to take singular value decompositions with numpy.
- This will work better than before, but still won't get us to e.g. 100th order polynomials.
- Main issue is that simple polynomials are ill-conditioned: x^{20} looks a lot like x^{22}.

```
import numpy as np

x=np.linspace(-2,2)
y=np.cos(x)+np.sin(x)

ord=5
n=len(x)
mat=np.zeros([n,ord+1])
for i in range(ord+1):
    mat[:,i]=x**i

u,s,v=np.linalg.svd(mat,0)
fitp=v.T@np.diag(1/s)@u.T@y
pred=mat@fitp
print('reconstruction error RMS is ',np.std(pred-y))
print('inverse condition number of SVD is ',s.min()/s.max())
```

(Bad) Solution: pinv

- Since small eigenvalues in $A^TN^{-1}A$ are the problem, we could zero them out in the inverse.
- np.linalg.pinv will do this, with adjustable threshold for small eigenvalues.
- This will give a sensible answer. Will it give a good answer?
- Probably not! We've thrown away information by zeroing out eigenvalues. Our fit does not actually have as much freedom as it should.
- Is this a problem? That is for you to decide…

Solution: Different Poly Basis

- There are several families of polynomials that have better properties (Legendre, Chebyshev…). Usually defined on (-1,1) through recursion relations.

- Legendre polynomials are constructed to be orthogonal on (-1,1), so condition number should be good. If our t range is different from (-1,1), rescale so that it is.

- Key relation: $(n+1)P_{n+1}(t)=(2n+1)tP_n(t)-nP_{n-1}(t)$ with $P_0=1$ and $P_1=t$.

- I pick up a power of *t* each time, so these are also polynomials, just written in linear combinations that have better condition number.

- Strongly encourage you to *never* fit regular polynomials. Always use Legendre, Chebyshev…

Legendre Code

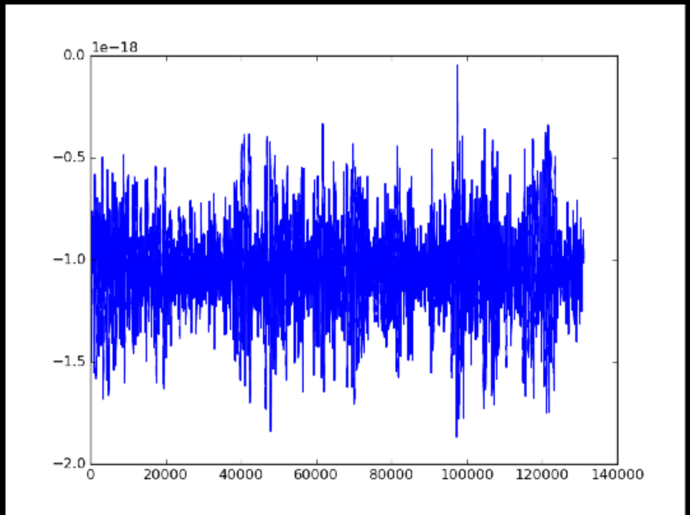
```python
import numpy as np
x=np.linspace(-1,1,1001)
ords=np.arange(5,101,5)
for ord in ords:
    #legvander is the numpy routine to make a matrix of legendre polynomials.
    #stands for legendre vandermond
    y=np.polynomial.legendre.legvander(x,ord)
    #the 0 argument to SVD says to keep the output in compact (rectangular)
    #form if the input matrix is rectangular
    u,s,v=np.linalg.svd(y,0)
    print('legendre condition number for order ',ord,' is ',s.max()/s.min())
```

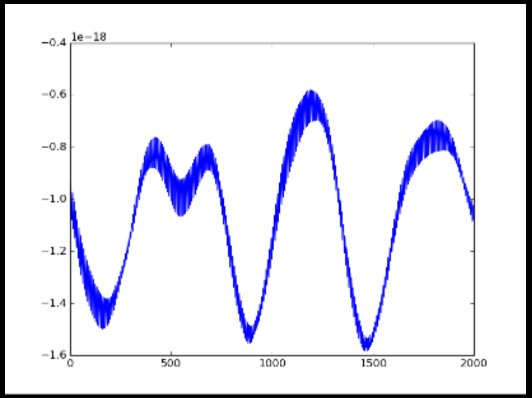
```
legendre condition number for order  5  is  3.3004122674582725
legendre condition number for order  10  is  4.542374337123092
legendre condition number for order  15  is  5.508348566496
legendre condition number for order  20  is  6.334121283482599
legendre condition number for order  25  is  7.070257461267389
legendre condition number for order  30  is  7.74091305773132
legendre condition number for order  35  is  8.360572554145357
legendre condition number for order  40  is  8.939241298418528
legendre condition number for order  45  is  9.484316948354397
legendre condition number for order  50  is  10.001668962700121
legendre condition number for order  55  is  10.496403074691885
legendre condition number for order  60  is  10.974228862874183
legendre condition number for order  65  is  11.445362702828856
legendre condition number for order  70  is  11.942607221492095
legendre condition number for order  75  is  12.571332359572134
legendre condition number for order  80  is  13.493593404203155
legendre condition number for order  85  is  14.8728748591077
legendre condition number for order  90  is  16.932477120003394
legendre condition number for order  95  is  20.0307649803339
legendre condition number for order  100  is  24.768705279365424
```

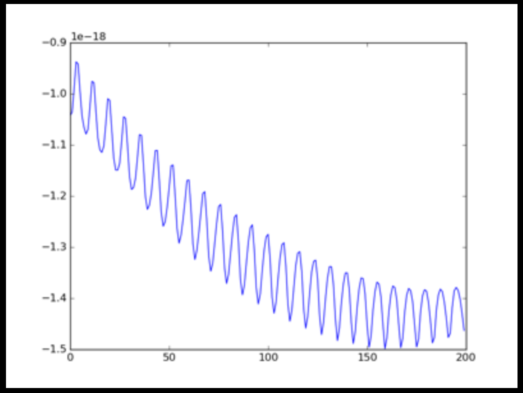
Correlated Noise

- So far, we have assumed that the noise is independent between data sets.

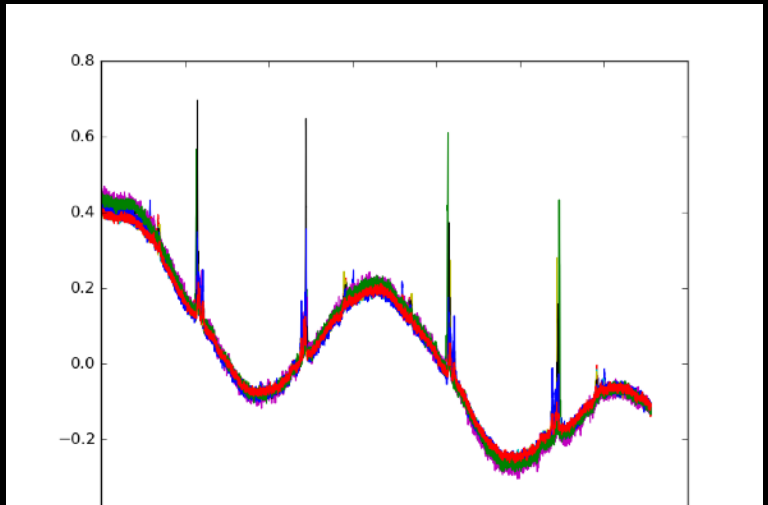
- Life is sometimes that kind, but very often not. We need tools to deal with this.







Right:LIGO data, with varying levels of zoom. Left: detector timestreams from Mustang 2 camera @GBT



Fortunately…

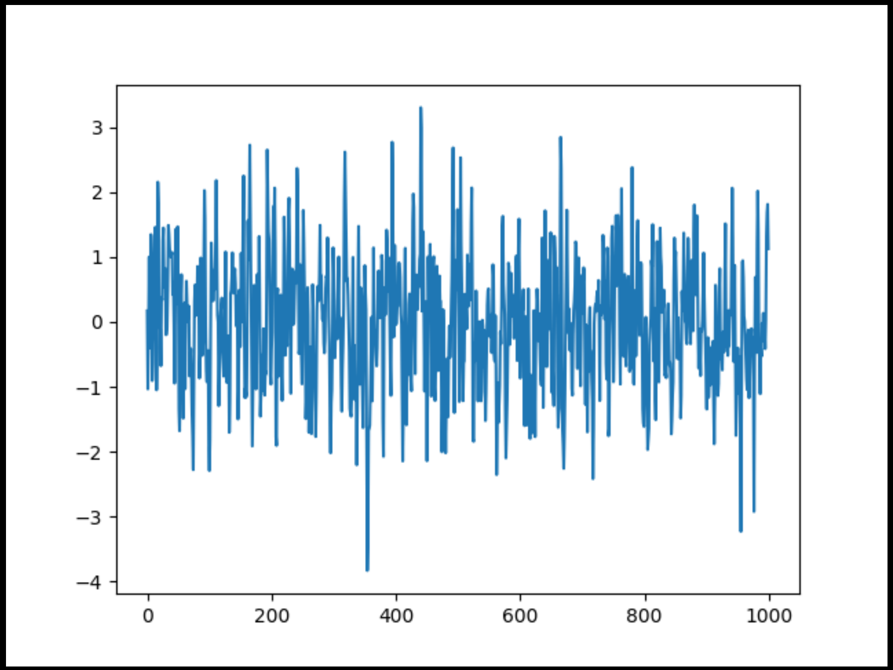
- Linear algebra expressions for χ^2 already can handle this.
- Let V be an orthogonal matrix, so $VV^T=V^TV=I$, and $d-Am=r$ (for residual)
- $\chi 2=r^TN^{-1}r$ $=r^TV^TVN^{-1}V^TVr$. Let $r\text{->}Vr$, $N\text{->}VNV^T$, and $\chi 2$ expression is unchanged in new, rotated space.
- Furthermore, (fairly) easy to show that $\langle N_{ij}\rangle=\langle r_ir_j\rangle$.
- So, we can work in this new, rotated space without ever referring to original coordinates. Just need to calculate noise covariances N_{ij}.

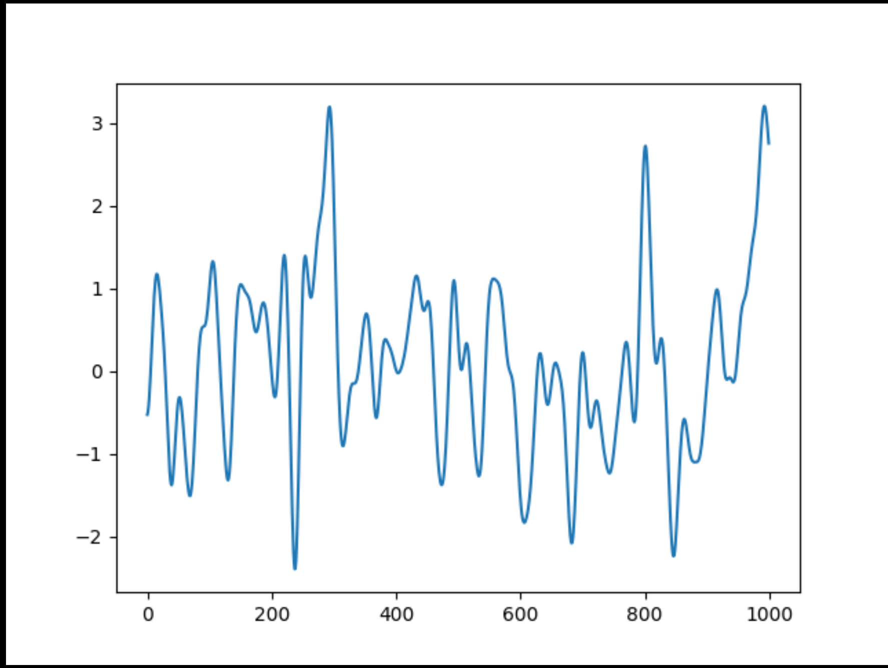
Generating Correlated Noise

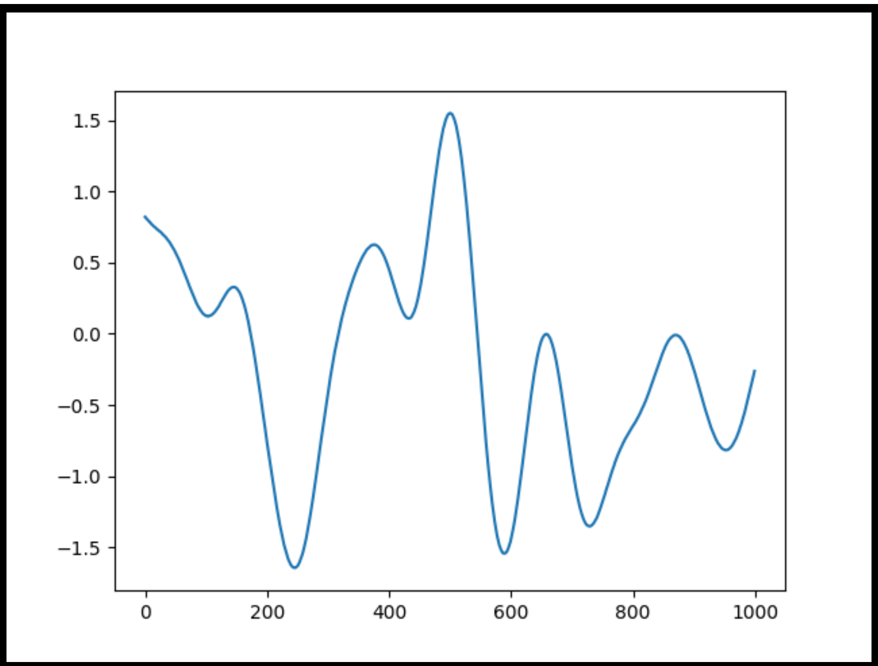
- Say we have a noise matrix N and want to create realizations from it. How do we do this?
- Same trick in reverse. If $d_{new}=Vd_{old}$, then I can generate d_{old} and rotate to get d_{new}.
- We can pick any matrix that diagonalizes N, since we know how to generate uncorrelated data.
- A particularly useful one is Cholesky (LU equivalent for positive-definite): $N=LL^T$. We can generate simulated data just by taking L*g*, where *g* is a vector of zero-mean, unit-variance Gaussian random deviates.
- Often a good idea to generate many simulations, average their outer products, and check empirical matrix agrees with expected.

Gaussian Correlated Noise

- Look at gauss_corrnoise.py
- <f(x)f(x+dx)>=exp(-x²/2σ²)
- top: σ=1, middle σ=10, bottom σ=50







Stationary Noise

- A common case for correlated noise is noise depends only on separation of points, $N_{ij}=f(|i-j|)$

- This is called *stationary* noise, since on average, noise properties don't change with time.

- We can invoke the power of Fourier transforms to enormously simplify stationary noise.

- Note - computer takes the discrete Fourier transform (DFT): $F(k)=\sum f(x)\exp(-2\pi ikx/N)$. I suggest you commit this to memory.

- IDFT: $f(x)=1/N \sum F(k)\exp(2\pi ikx/N)$

- Note: x,k integers, go from 0 to N-1

Stationary Noise 2

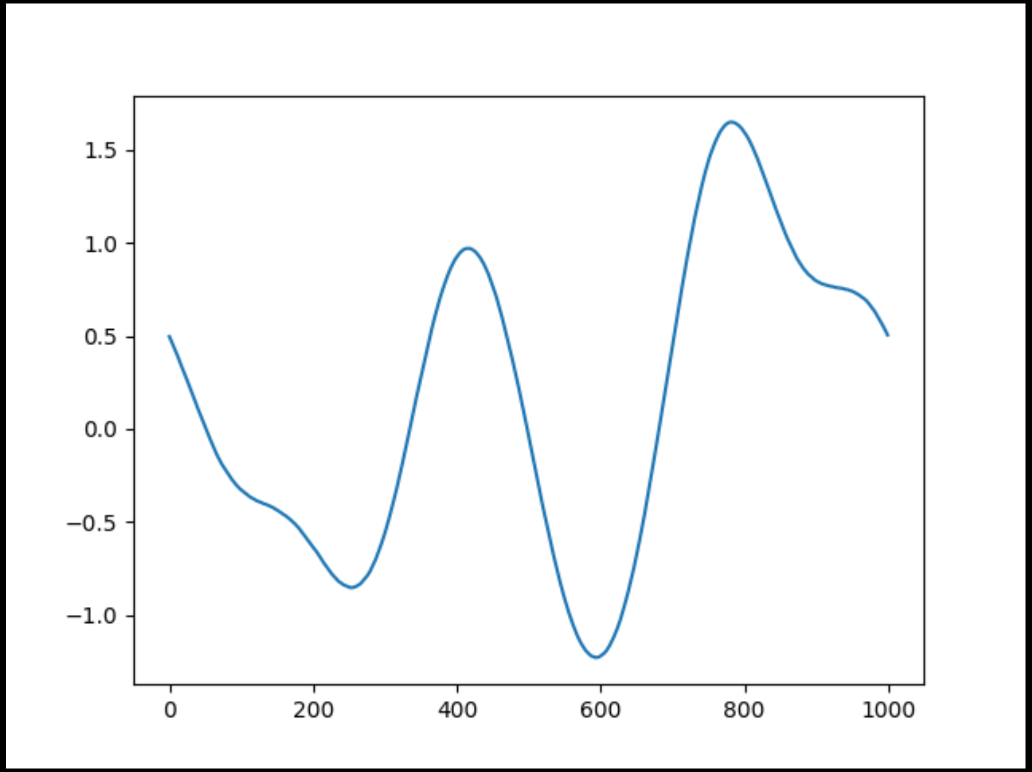
- Say we have noise where $\langle f(x)f(x+dx)\rangle=g(dx)$ (not of x)
- Fourier space: $\langle F(k)F^*(k')\rangle=\langle\sum f(x)\exp(-2\pi ikx/N)\sum f(x')\exp(2\pi ik'x'/N)$
- Can swap x' for x+dx, since sum is over all points, can sum over dx:
- $\langle F(k)F^*(k')\rangle=\langle\sum f(x)\exp(-2\pi ikx/N)\sum f(x+dx)\exp(2\pi ik'(x+dx)/N)\rangle$
- Reorder sum over x then dx: $\langle\sum \exp(2\pi ik'dx/N)\sum f(x)f(x+dx)\exp(2\pi ix(k'-k)/N\rangle$
- Now $\langle f(x)f(x+dx)\rangle=g(dx)$ (by assumption), can come out. $\sum \exp(2\pi ik'dx/N)g(dx)\sum \exp(2\pi ix(k'-k)/N)$
- Interior goes to N for k'=k, left with $N\sum g(dx)\exp(2\pi ikdx/N)\delta_{kk'}$, so Fourier transform of noise is diagonal.
- Further, variance of F(k) given by Fourier transform of correlation function g(dx).

Correlated Noise via DFT

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

n=1000
x=np.fft.fftfreq(n)*n
sig=100
mycorr=np.exp(-0.5*x**2/sig**2)
myps=np.fft.rfft(mycorr)

dat=np.random.randn(n)
datft=np.fft.rfft(dat)
dat_corr=np.fft.irfft(datft*np.sqrt(myps))
```



- Code is much shorter!
- And much (much) faster!

Correlated Noise Summary

- In real life, we frequently have noise that is correlated between data points.
- Our entire least-squares framework carries over, as long as $N_{ij}=\langle n_i n_j \rangle$
- We can generate correlated noise using e.g. Cholesky (or eigenvalues, but Cholesky faster)
- If the noise is stationary, the noise is diagonal in Fourier space.
- Can generate correlated stationary noise realization very fast with Fourier transforms, since $\langle F(k)^2 \rangle = DFT(g(dx))$ where $g(dx)=\langle f(x)f(x+dx) \rangle$

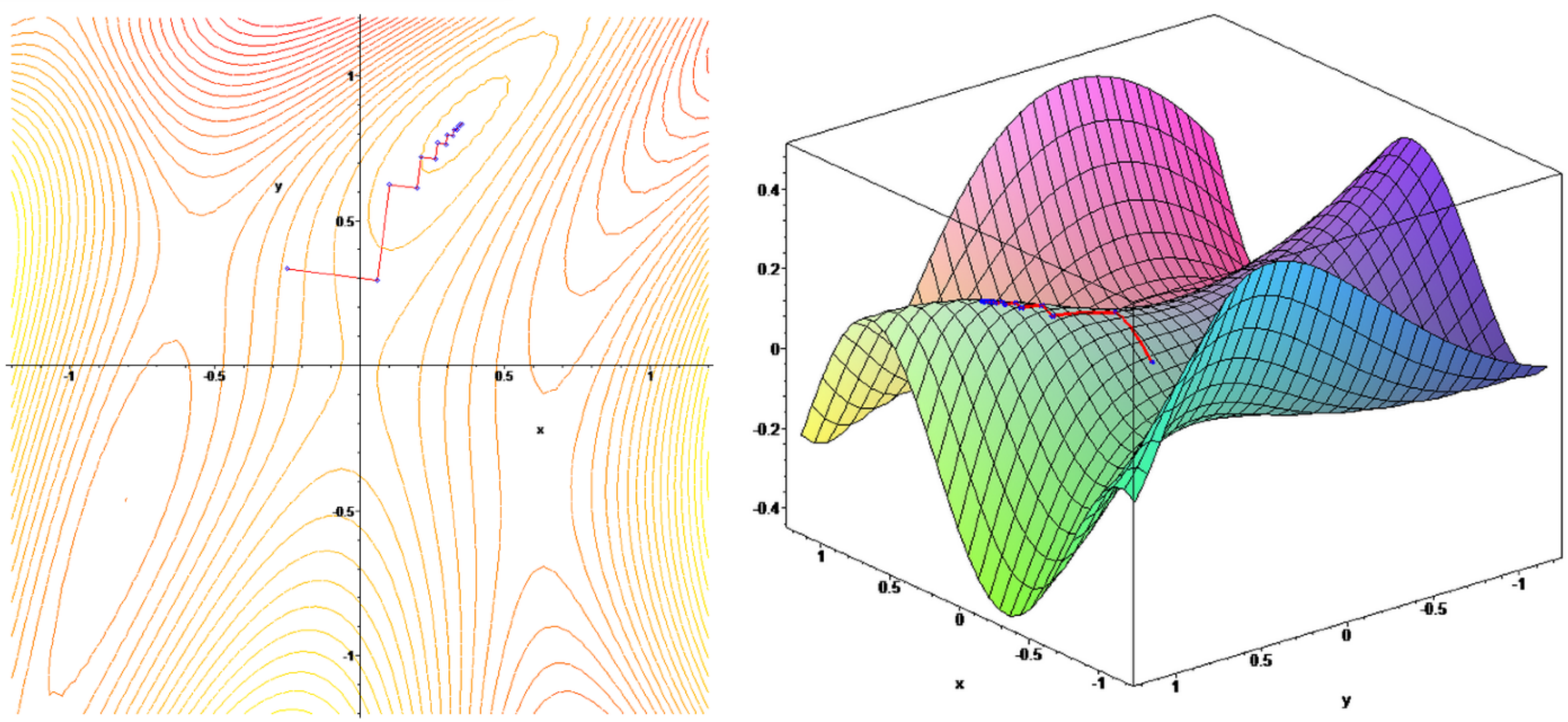
Nonlinear Fitting

- Sometimes data depend non-linearly on model parameters
- Examples are Gaussian and Lorentzian $(a/(b+(x-c)^2)$
- Often significantly more complicated - cannot reason about global behaviour from local properties. May be multiple local minima
- Many methods reduce to how to efficiently find the "nearest" minimum.
- One possibility - find steepest downhill direction, move to the bottom, repeat until we're happy. Called "steepest descent."
- How might this end badly?

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.

Worked Example

- What is the best-fit mean and error for a set of uncorrelated gaussian variables with same mean but individual errors?

- A=? Show that $A^T N^{-1} A = \sum(\sigma_i^{-2})$, $A^T N^{-1} d = \sum d_i/\sigma_i^2$.

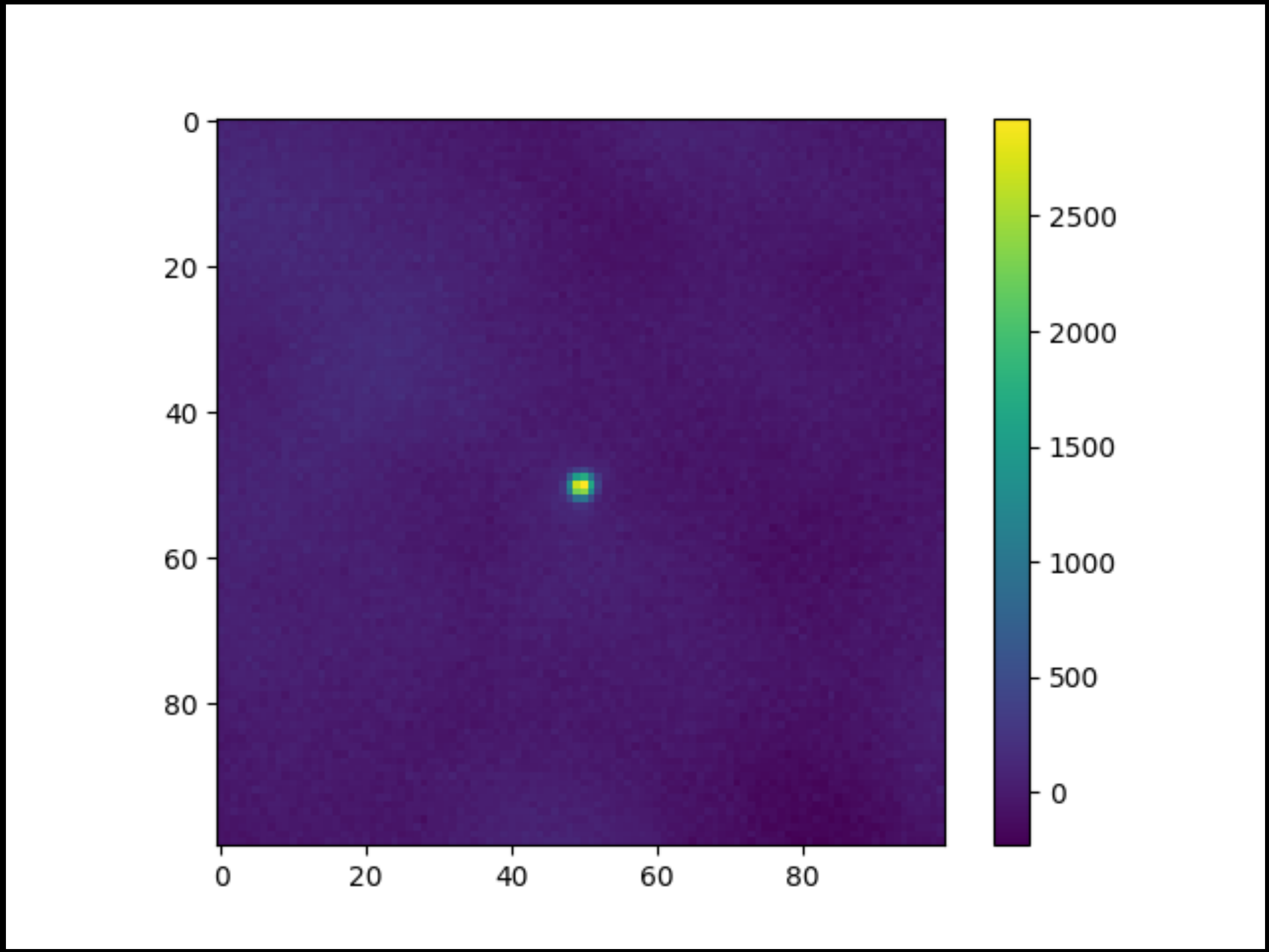
- Define weights $w_i = \sigma_i^{-2}$. Then $m = \sum w_i d_i / \sum w_i$. Variance of our estimator is $1/\sum w_i$.

Worked Example 2

- Let's assume that N is constant and diagonal, and we have a single paramers.
- Show LHS = $\sum(m_i^2/\sigma^2)$
- RHS=$\sum(d_i m_i/\sigma^2)$
- Best-fit amplitude is RHS/LHS = $\sum(d_i m_i)/\sum m_i^2$
- Error=1/sqrt(RHS) = σ/sqrt($\sum m_i^2$). If there are ~n model points with value ~1, this turns into σ/sqrt(n), as roughly expected.

Example - Source in ACT Data

- Let's fit the amplitude of a source in ACT data.
- Look at find_act_flux.py.
- Let's try to guess a Gaussian, fit amplitude to it.
- Map is saved as FITS. Ability to read/write/manipulate FITS images extremely useful in astronomy!



Better: Newton's Method

- linear: $\langle d \rangle = Am$. Nonlinear: $\langle d \rangle = A(m)$ $\chi^2=(d-A(m))^T N^{-1}(d-A(m))$
- If we're "close" to minimum, can linearize. $A(m)=A(m_0)+\partial A/\partial m * \delta m$
- Now have $\chi^2=(d-A(m_0)-\partial A/\partial m\ \delta m)^T N^{-1}(d-A(m_0)-\partial A/\partial m\ \delta m)$
- What is the gradient?

Newton's Method ctd

- Gradient trickier - $\partial A/\partial m$ depends in general on m, so there's a second derivative

- Two terms: $\nabla\chi^2=(-\partial A/\partial m)^T N^{-1}(d-A(m_0)-\partial A/\partial m\ \delta m)-(\partial^2 A/\partial m_i \partial m_j\ \delta m)^T N^{-1}(d-A(m_0)-\partial A/\partial m\ \delta m)$

- If we are near solution $d \approx A(m_0)$ and δm is small, so first term has one small quantity, second has two. Second term in general will be smaller, so usual thing is to drop it.

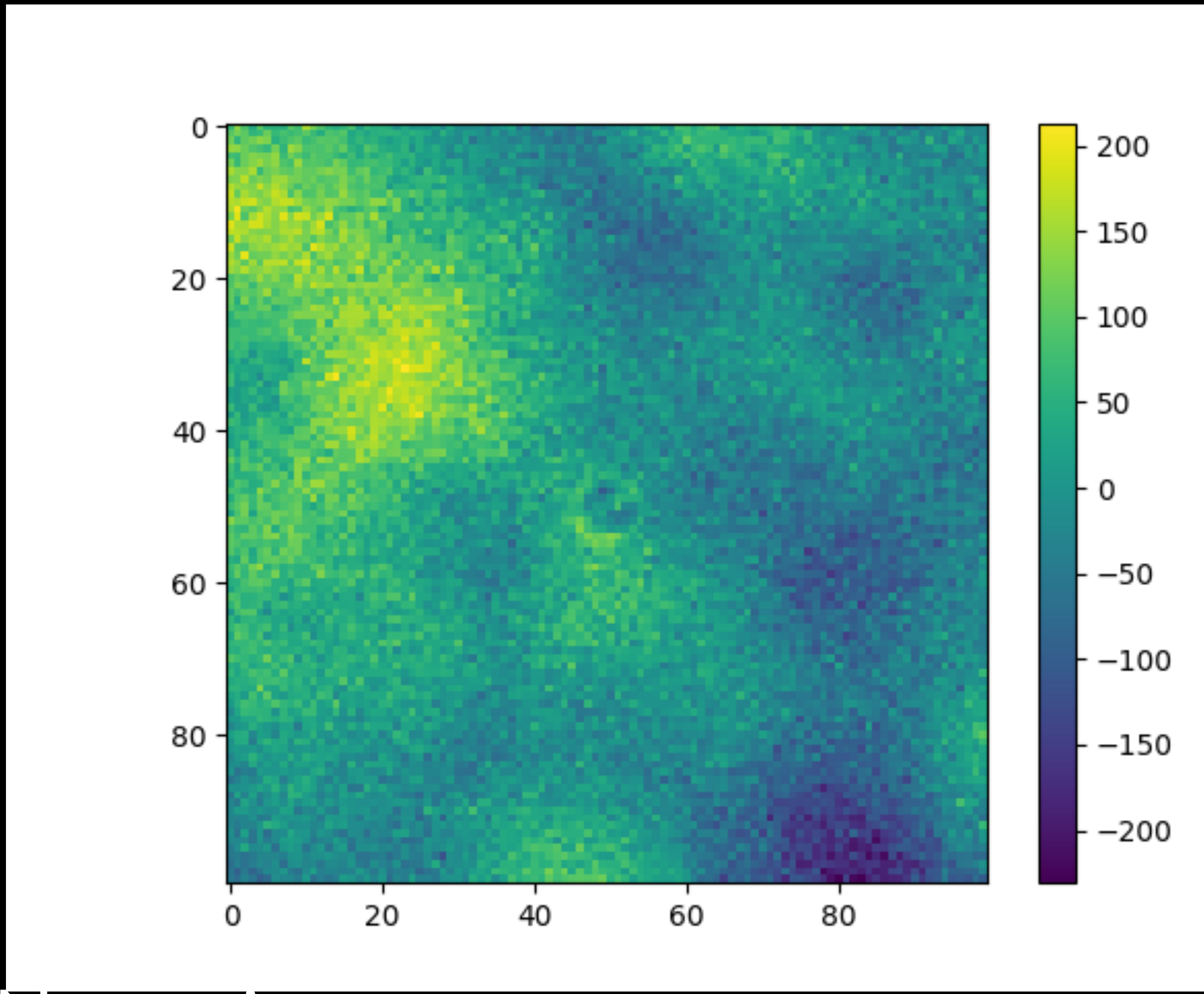
- Call $\partial A/\partial m$ A_m. Call $d-A(m_0)$ r. Then $\nabla\chi^2 \approx -A_m^T N^{-1}(r-A_m \delta m)$

- We know how to solve this! $A_m^T N^{-1} A_m\ \delta m = A_m^T N^{-1} r$

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?).

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?