

# Phys 512

## Lecture 5

# Model Fitting

- Sometime we have data. Sometimes we'd like to model it.
- Variety of techniques used, depending on what noise, model look like.
- Nearly all of these techniques require linear algebra, so we will start with numpy linear algebra.

# Matrices vs. Arrays

- Unlike say Matlab, arrays and matrices are different in numpy.
- However, you can treat arrays like matrices and matrices like arrays. This can be confusing.
- `*` overloaded for matrices to be matrix multiply, but not for arrays. As of python3, `@` is overloaded for arrays to be matrix multiply.
- I suggest you pick one and stick with it. People usually use arrays (and we will do so in this course).

# Linear Algebra Operations

- Several matrix operations/factorizations built into numpy
- `numpy.dot(a,b)` does matrix multiplication if `a,b` are matrices.
- `numpy.linalg.inv` inverts a square matrix
- `numpy.linalg.eig` takes eigenvalues/eigenvectors
- `numpy.linalg.svd` takes singular value decomposition  $A=USV^T$  where  $S$  diagonal,  $U,V$  columns orthogonal
- `numpy.linalg.qr` takes QR decomposition  $A=QR$  where  $Q$  is orthogonal,  $R$  triangular
- `numpy.linalg.chol` takes Cholesky decomposition of a positive-definite matrix  $A=LL^T$ .

$$\chi^2$$

- The PDF of a Gaussian is  $\exp(-0.5(x-\mu)^2/\sigma^2)/\sqrt{2\pi\sigma^2}$  with mean  $\mu$  and standard deviation  $\sigma$ .
- If we have a bunch of data points, which may have different means and standard deviations, then the joint PDF is the product of the PDFs.
- It is often more convenient to work with the log. For many points,  $\log(\text{PDF}) = \sum -0.5(x_i - \mu_i)^2/\sigma_i^2 - 0.5 \log(2\pi\sigma_i^2)$
- Usually, we know the variance of our data, and want our model to predict the expected value of  $x_i$ , which is  $\mu_i$ . When we compare models, the second part is constant, so we ditch it. log likelihood becomes:  $-0.5 \sum (x_i - \mu_i)^2/\sigma_i^2$ .
- $\sum (x_i - \mu_i)^2/\sigma_i^2$  is  $\chi^2$ . We can find the maximum likelihood model by minimizing  $\chi^2$ .

# Linear least-squares

- Rewrite  $\chi^2$  with matrices:  $(\mathbf{x}-\boldsymbol{\mu})^T \mathbf{N}^{-1} (\mathbf{x}-\boldsymbol{\mu})$  for noise covariance matrix  $\mathbf{N}$ . If  $\mathbf{N}$  has diagonal elements  $\sigma^2$ , this is identical to previous.
- Let's take simple case that our model depends linearly on a small number of parameters:  $\mu_i = \sum A_{ij} m_j$  for model parameters  $m$  and matrix  $A$  that transforms to predicted values. In matrixese:  $\boldsymbol{\mu} = A\mathbf{m}$
- One example:  $x(t)$  is a polynomial in time. Then  $\mu_i = \sum t_i^j c_j$ .
- With this parameterization,  $\chi^2 = (\mathbf{x} - A\mathbf{m})^T \mathbf{N}^{-1} (\mathbf{x} - A\mathbf{m})$



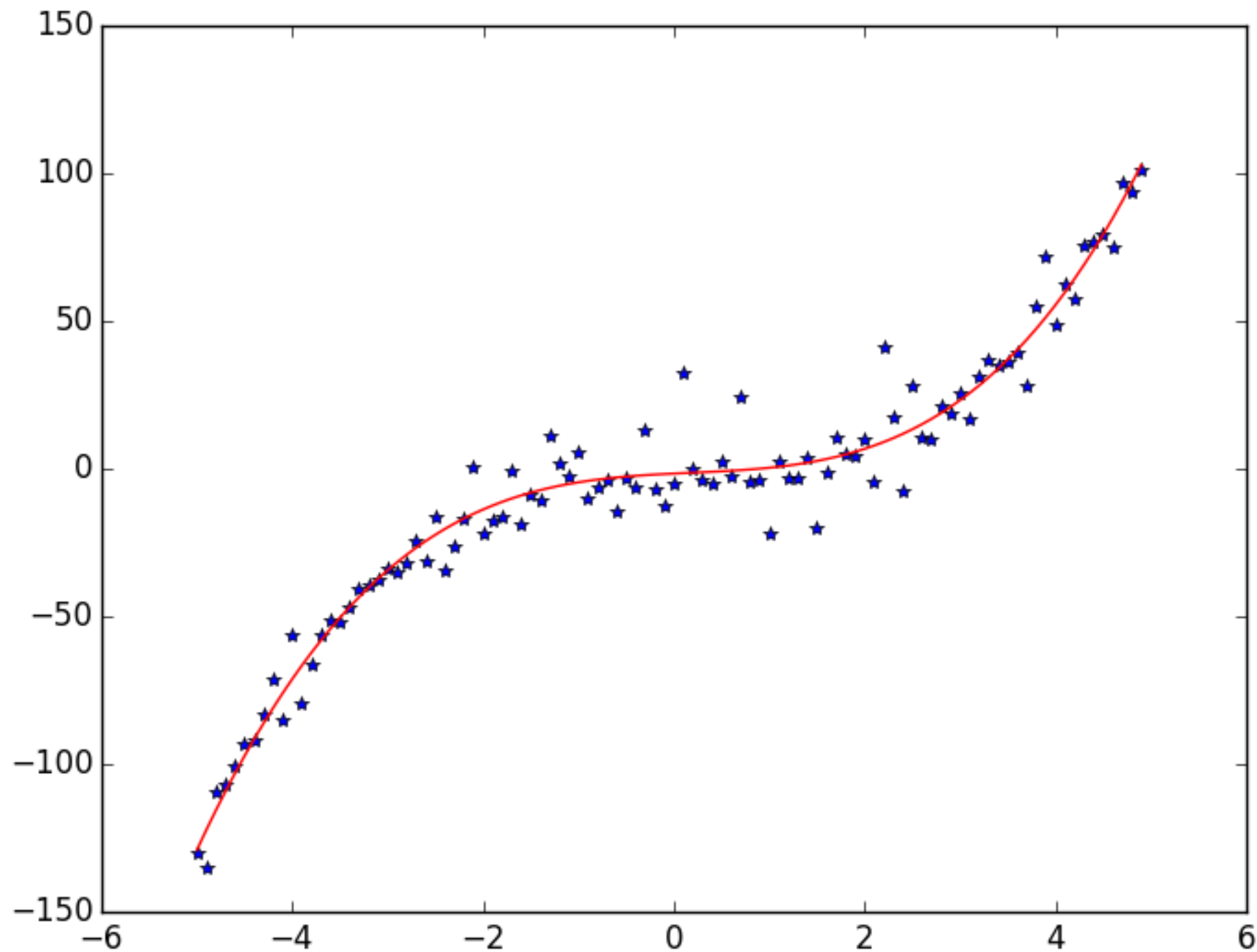
# Least Squares: $\chi^2 = (\mathbf{x} - \mathbf{A}\mathbf{m})^T \mathbf{N}^{-1} (\mathbf{x} - \mathbf{A}\mathbf{m})$

- To find best-fitting model, minimize  $\chi^2$ . Calculus on matrices works like regular calculus, as long as no orders get swapped.
- $\partial \chi^2 / \partial \mathbf{m} = -\mathbf{A}^T \mathbf{N}^{-1} (\mathbf{x} - \mathbf{A}\mathbf{m}) + \dots = 0$  (at minimum)
- We can solve for  $\mathbf{m}$ :  $\mathbf{A}^T \mathbf{N}^{-1} \mathbf{A} \mathbf{m} = \mathbf{A}^T \mathbf{N}^{-1} \mathbf{x}$ . Or,  $\mathbf{m} = (\mathbf{A}^T \mathbf{N}^{-1} \mathbf{A})^{-1} \mathbf{A}^T \mathbf{N}^{-1} \mathbf{x}$

# Example: Polynomial Regression

```
import numpy
from matplotlib
t=numpy.arange(-5,5)
x_true=t**3-
x=x_true+10*

npoly=5 #le
ndata=t.size
A=numpy.zeros(ndata,npoly)
A[:,0]=1.0
for i in range(1,npoly):
    A[:,i]=A[:,i-1]*t
#Let's ignore the noise
#m=(A^TA)^{-1}A^Ty
A=numpy.linalg.pinv(A)
d=numpy.linalg.pinv(A)
lhs=A.transpose()
rhs=A.transpose()*x
fitp=numpy.linalg.pinv(lhs)*rhs
pred=A*fitp
plt.clf();plt.plot(t,x,'b*')
plt.plot(t,pred,'r')
```

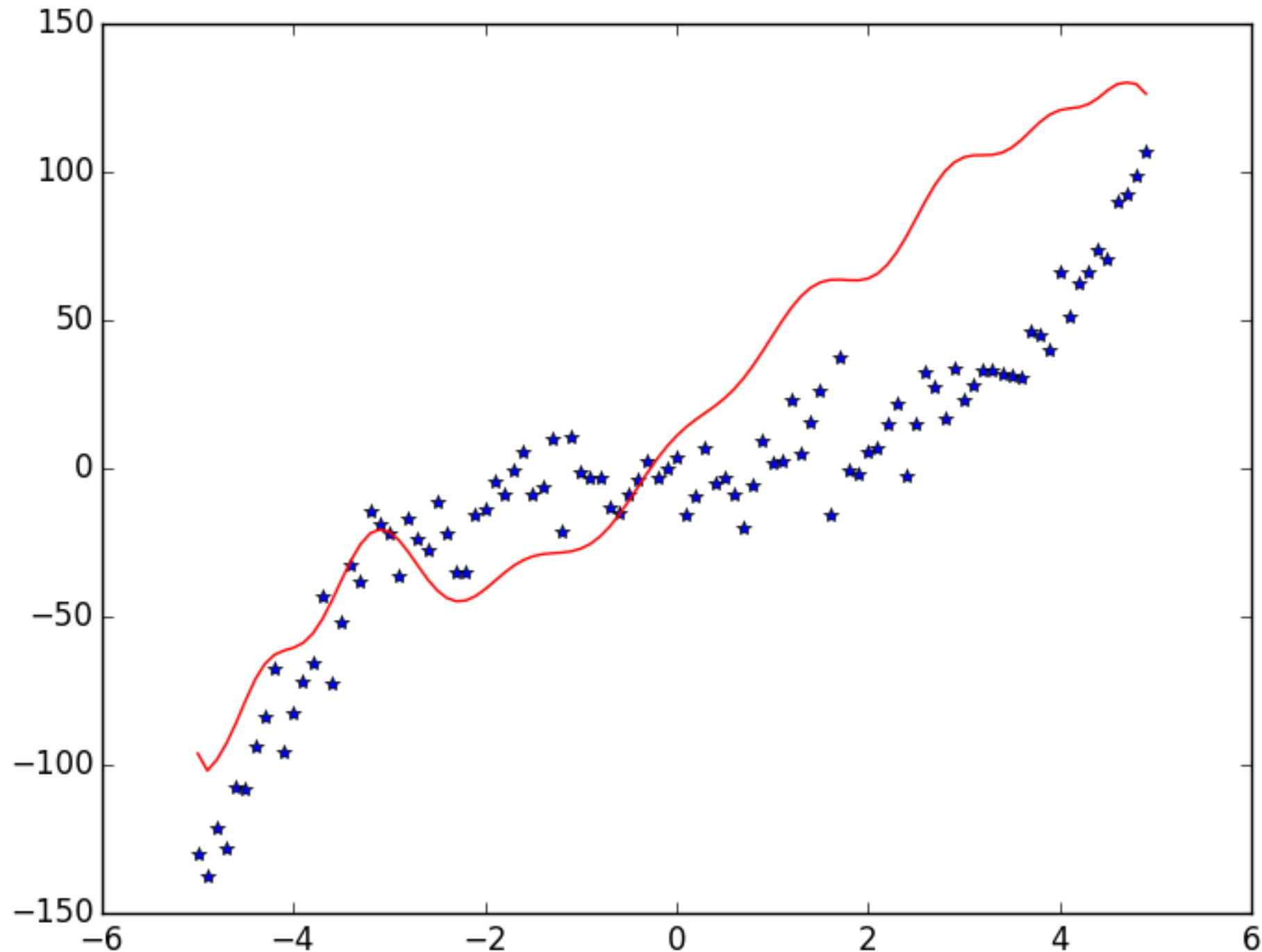


etc. as matrices rather

s live in numpy.linalg,



# Higher Order



```
import numpy
from matplotlib import pyplot as plt
t=numpy.arange(-5,5,0.1)
x_true=t**3-0.5*t**2
x=x_true+10*numpy.random.randn(t.size)

npoly=25 #let's fit 4th order polynomial
ndata=t.size
A=numpy.zeros([ndata,npoly])
A[:,0]=1.0
for i in range(1,npoly):
    A[:,i]=A[:,i-1]*t
#Let's ignore noise for now. New equations are:
#m=(A^TA)^{-1}*(A^Td)
A=numpy.matrix(A)
d=numpy.matrix(x).transpose()
lhs=A.transpose()*A
rhs=A.transpose()*d
fitp=numpy.linalg.inv(lhs)*rhs
pred=A*fitp
plt.clf();plt.plot(t,x,'*');plt.plot(t,pred,'r');
plt.draw()
plt.savefig('polyfit_example_high.png')
```

# Condition # and Roundoff

- Recall that the eigenvalues of a symmetric matrix are real, and the eigenvectors are orthogonal. So,  $(A^T N^{-1} A)$  can be re-written  $V^T \Lambda V$ , where  $\Lambda$  is diagonal and  $V$  is orthogonal (so  $V^{-1} = V^T$ ).
- $(ABC)^{-1} = C^{-1} B^{-1} A^{-1}$ , so  $\text{inverse} = V^{-1} \Lambda^{-1} (V^T)^{-1} = V^T \Lambda^{-1} V$ .
- If a bunch of eigenvalues are really small, they will be huge in the inverse. Double precision numbers are good to  $\sim 16$  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  $A^T A m = A^T x$ .
- Singular value decomposition (SVD) factors matrix  $A = U S V^T$ , where  $S$  is diagonal, and  $U$  and  $V$  are orthogonal, and  $V$  is square. For symmetric,  $U = V$ ,  $S = \text{eigenvalues}$ , but SVD works for any matrix.
- Solutions:  $(U S V^T)^T U S V^T m = (U S V^T)^T x$ .  $V S U^T U S V^T m = V S U^T x$
- $U^T U = \text{identity}$ , so cancels.  $V S^2 V^T m = V S U^T x$ .  $S^2$  squares the condition number, so that was bad. We can analytically cancel left-hand  $V$  and one copy of  $S$ :  $S V^T m = U^T x$ . Then  $m = V S^{-1} U^T x$
- NB - 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. 100<sup>th</sup> order polynomials.
- Main issue is that simple polynomials are ill-conditioned:  $x^{20}$  looks a lot like  $x^{22}$ .

```
import numpy
from matplotlib import pyplot as plt
t=numpy.arange(-5,5,0.1)
x_true=t**3-0.5*t**2
x=x_true+10*numpy.random.randn(t.size)

npoly=20
ndata=t.size
A=numpy.zeros([ndata,npoly])
A[:,0]=1.0
for i in range(1,npoly):
    A[:,i]=A[:,i-1]*t

A=numpy.matrix(A)
d=numpy.matrix(x).transpose()
#Make the svd decomposition, the extra False
#is to make matrices compact
u,s,vt=numpy.linalg.svd(A,False)
#s comes back as a 1-d array, turn it into a 2-d matrix
sinv=numpy.matrix(numpy.diag(1.0/s))
fitp=vt.transpose()*sinv*(u.transpose()*d)
```

# 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

```
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

# Back to Polynomials...

- I mentioned there are several versions people use. Why?
- I mean, isn't least-squares always best?
- Not always... Sometimes you want least-bad behaviour over the whole range. Least squares will often trade off bad at edges for even better in middle.
- Enter Chebyshev polynomial.



# Chebyshev Polynomials

- Chebyshev polynomials are defined (among other ways) as:  $T_n = \cos(n \cdot \arccos(x))$ ,  $-1 \leq x \leq 1$
- Similar to Legendre have recurrence relation:  $T_{n+1} = 2xT_n - T_{n-1}$ , with  $T_0 = 1$  and  $T_1 = x$ .
- $T_n$  are bounded by  $\pm 1$ , and are more-or-less uniform between  $\pm 1$  throughout range.
- So, if coefficients drop as  $n$  increases, if you truncate series, maximum error anywhere is sum of absolute values of coefficients.



```
import numpy
from numpy.polynomial import chebyshev
from matplotlib import pyplot as plt
x=numpy.arange(-1,1,1e-3)
plt.ion()
plt.clf();
t_m=0*x+1.0;
t_0=x;
plt.plot(x,t_m)
plt.plot(x,t_0)
for ord in range(2,5):
    t_n=2*t_0*x-t_m
    plt.plot(x,t_n)
    t_m=t_0
    t_0=t_n
plt.savefig('cheb_pols.png')
```

# Cheb ctd.

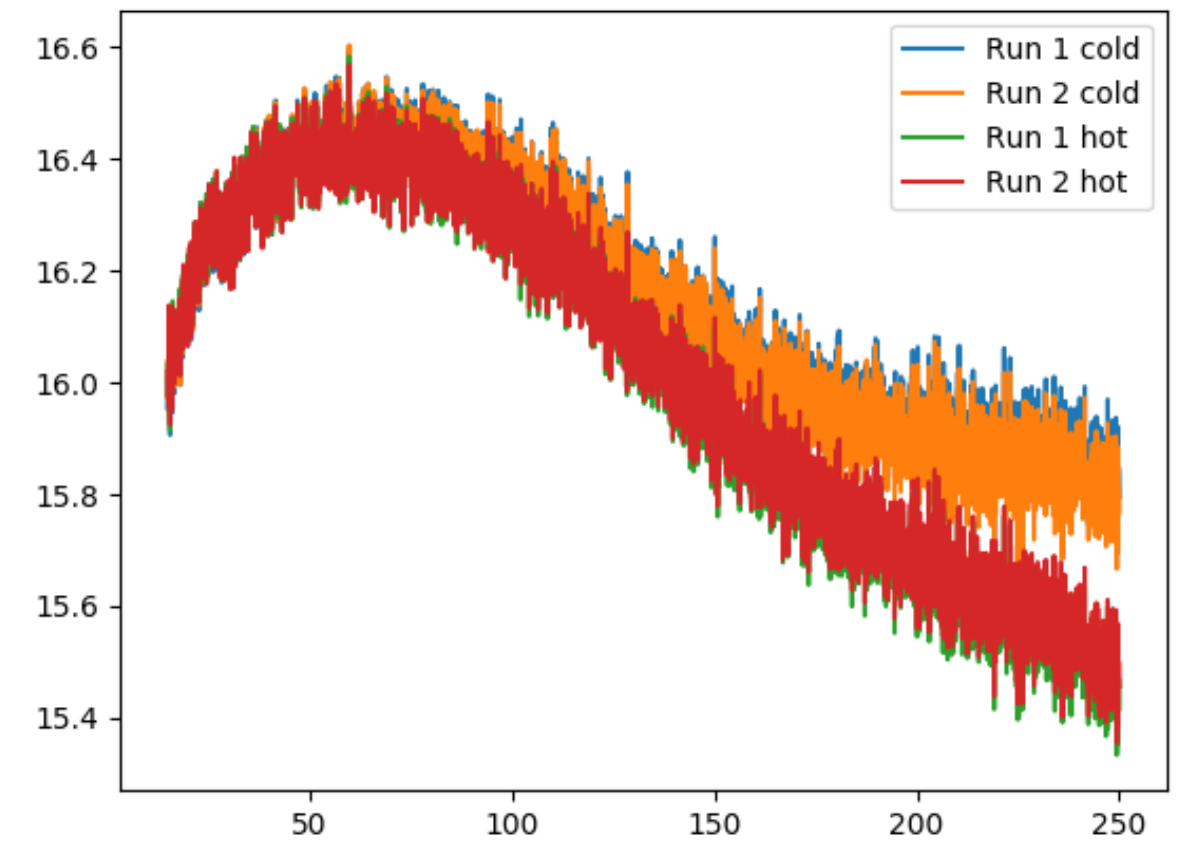
- Bound on error mean Chebyshev polynomials are great for modelling functions. Many implementations of functions in e.g. math library are based on them.
- Look at e.g. cos fit from  $-\pi$  to  $\pi$ .
- Why are odd terms zero?
- How many terms would you keep for single precision? For double?

```
>>> execfile("fit_chebyshev.py")
0   -3.0424e-01   9.6317e-16
2   -9.7087e-01   2.5647e-16
4    3.0285e-01  -2.9023e-16
6   -2.9092e-02  -1.1972e-16
8    1.3922e-03  -2.7269e-16
10  -4.0190e-05  -3.2028e-17
12   7.7828e-07  -6.0301e-17
14  -1.0827e-08  -5.7433e-16
16   1.1351e-10  -3.4716e-16
18  -9.2925e-13  -2.7599e-16
20   5.9186e-15  -1.1678e-16
22   2.7256e-16  -7.4016e-17
24   6.1067e-17  -1.0598e-16
26   2.5820e-16  -3.6700e-16
28  -1.1187e-16  -3.3971e-16
30  -2.6300e-16  -3.6472e-16
```

```
import numpy
from numpy.polynomial import chebyshev
x=numpy.arange(-1,1,1e-3)
y=numpy.cos(x*numpy.pi)
order=50
pp=chebyshev.chebfit(x,y,order)
for i in range(0,order,2):
    #note formatted output here, similar to C
    print '%3d %12.4e %12.4e' % (i,pp[i],pp[i+1])
    #print 2*i,pp[2*i],pp[2*i+1]
```

# Example: Amplifier Gain

- We have amplifiers for radio telescopes which amplify incoming signals.
- The gain varies as a function of frequency - if I want to know absolute signal level, need to correct for that.
- Plot shows two different runs for an amplifier at each of two different temperatures.
- How would I model the gain vs. frequency? vs. temperature?
- How would I decide on error bars?

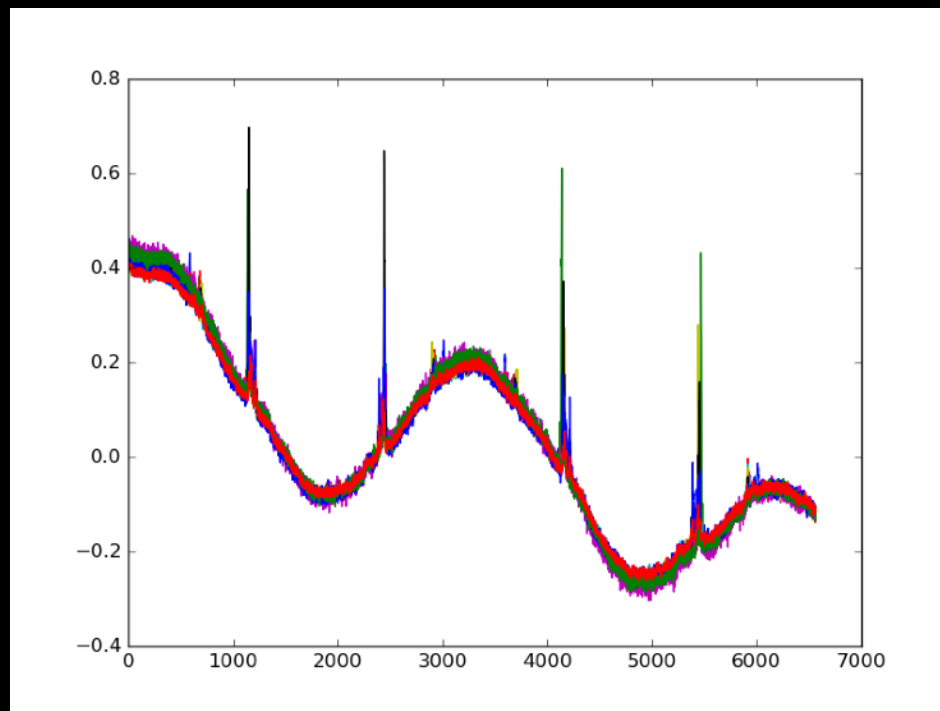


# Error Bars

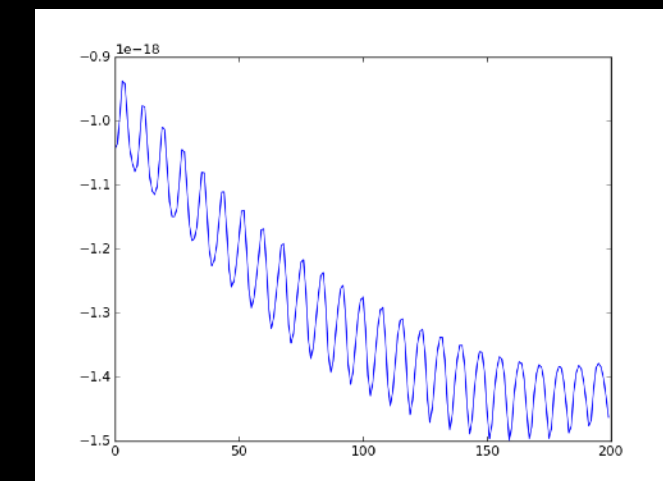
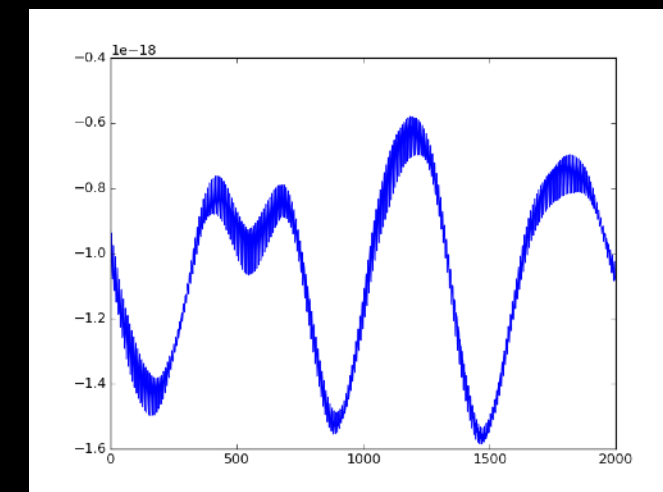
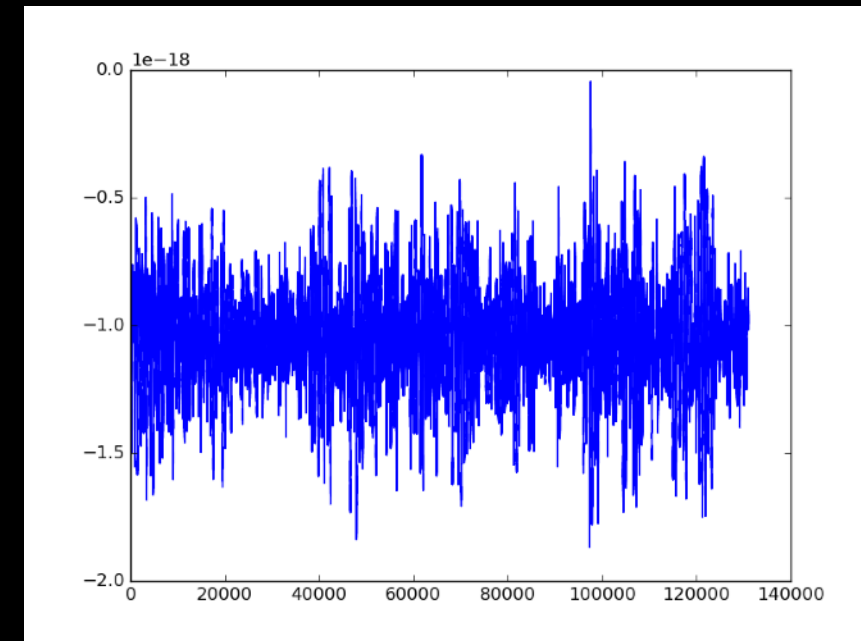
- What is the (co)variance of my model parameters?
- $d_t = Am_t$ , where  $d_t = \text{true}$  (noiseless data),  $m_t = \text{true model}$ .
- $A^T N^{-1} A m_t = A^T N^{-1} d_t$
- $A^T N^{-1} A (m - m_t) = A^T N^{-1} (d - d_t) = A^T N^{-1} n$  where  $n$  is the actual noise I got
- $(m - m_t) = (A^T N^{-1} A)^{-1} A^T N^{-1} n$ .
- $\langle (m - m_t)(m - m_t)^T \rangle = \langle (A^T N^{-1} A)^{-1} A^T N^{-1} n ((A^T N^{-1} A)^{-1} A^T N^{-1} n)^T \rangle$   
 $= \langle (A^T N^{-1} A)^{-1} A^T N^{-1} n n^T N^{-1} A (A^T N^{-1} A)^{-1} \rangle$
- But,  $\langle n n^T \rangle = N$ , so goes to  $(A^T N^{-1} A)^{-1} A^T N^{-1} N N^{-1} A (A^T N^{-1} A)^{-1} = (A^T N^{-1} A)^{-1}$
- Parameter covariance is just the (inverse) of the matrix we may have already inverted!

# 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  $\chi^2$  already can handle this.
- Let  $V$  be an orthogonal matrix, so  $VV^T = V^TV = I$ , and  $d - Am = r$  (for residual)
- $\chi^2 = r^T N^{-1} r = r^T V^T V N^{-1} V^T V r$ . Let  $r \rightarrow Vr$ ,  $N \rightarrow VNV^T$ , and  $\chi^2$  expression is unchanged in new, rotated space.
- Furthermore, (fairly) easy to show that  $\langle N_{ij} \rangle = \langle r_i r_j \rangle$ .
- 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  $d_{\text{new}} = V d_{\text{old}}$ , then I can generate  $d_{\text{old}}$  and rotate to get  $d_{\text{new}}$ .
- 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  $Lg$ , where  $g$  is a vector of zero-mean, unit-variance Gaussian random deviates.

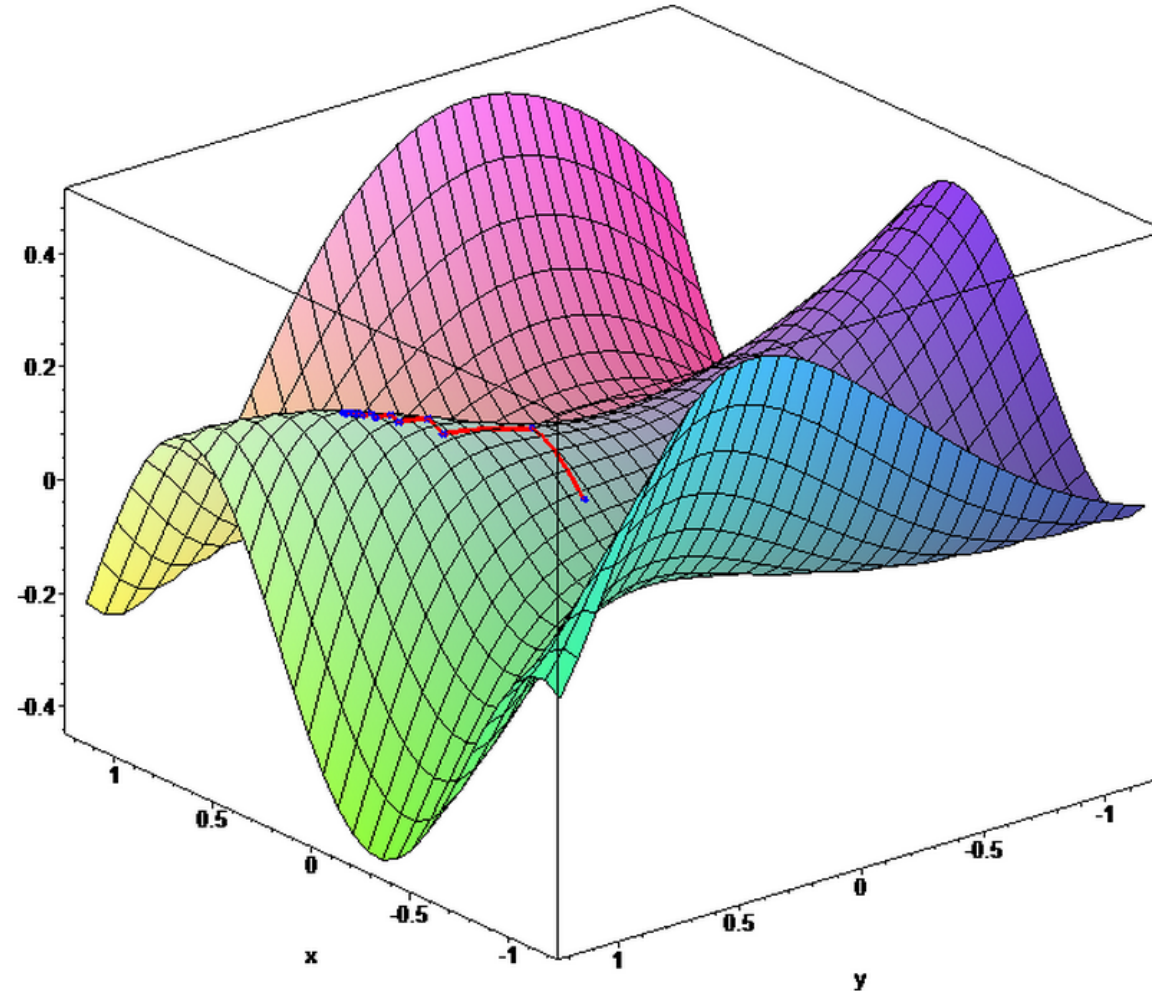
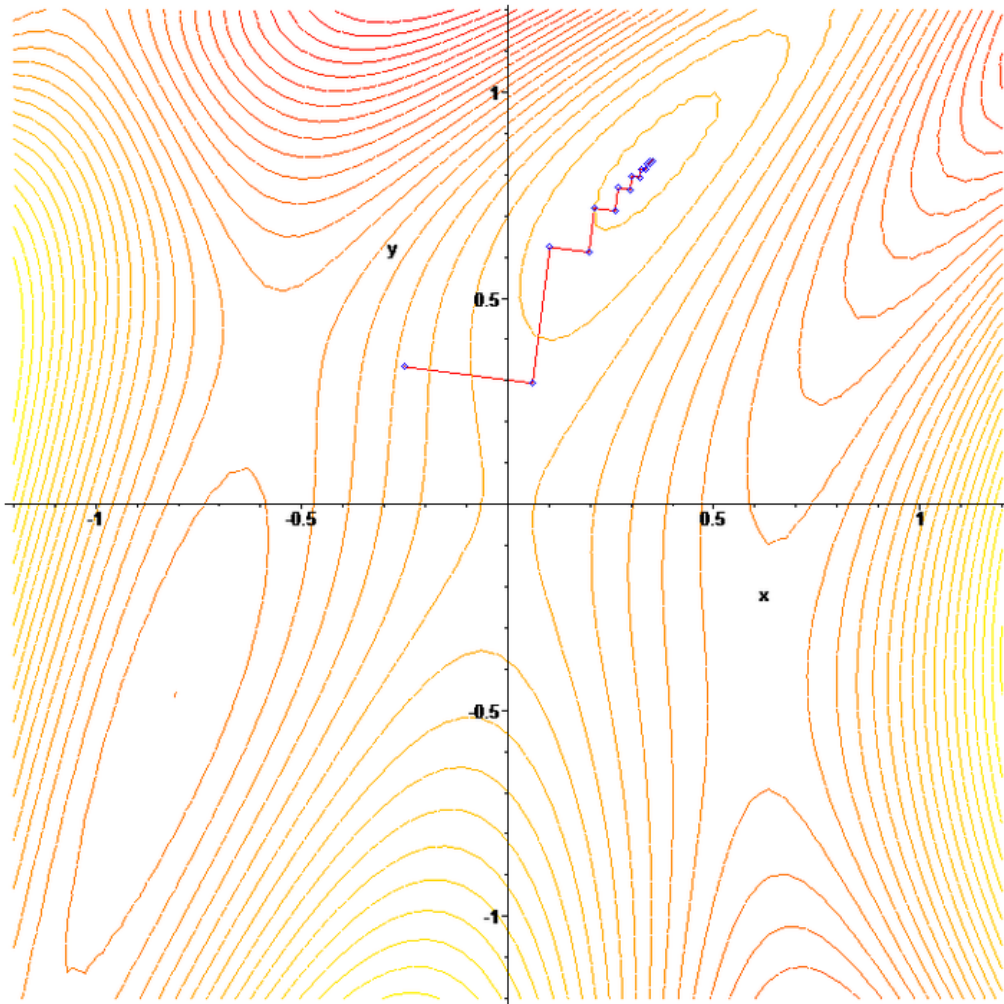
# Nonlinear Fitting

- 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  $F(x, y) = \sin\left(\frac{1}{2}x^2 - \frac{1}{4}y^2 + 3\right) \cos(2x + 1 - e^y)$ .



From wikipedia. Zigagging is inefficient.

# 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  $\delta m$  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!  $A_m^T N^{-1} A_m \delta m = A_m^T N^{-1} r$

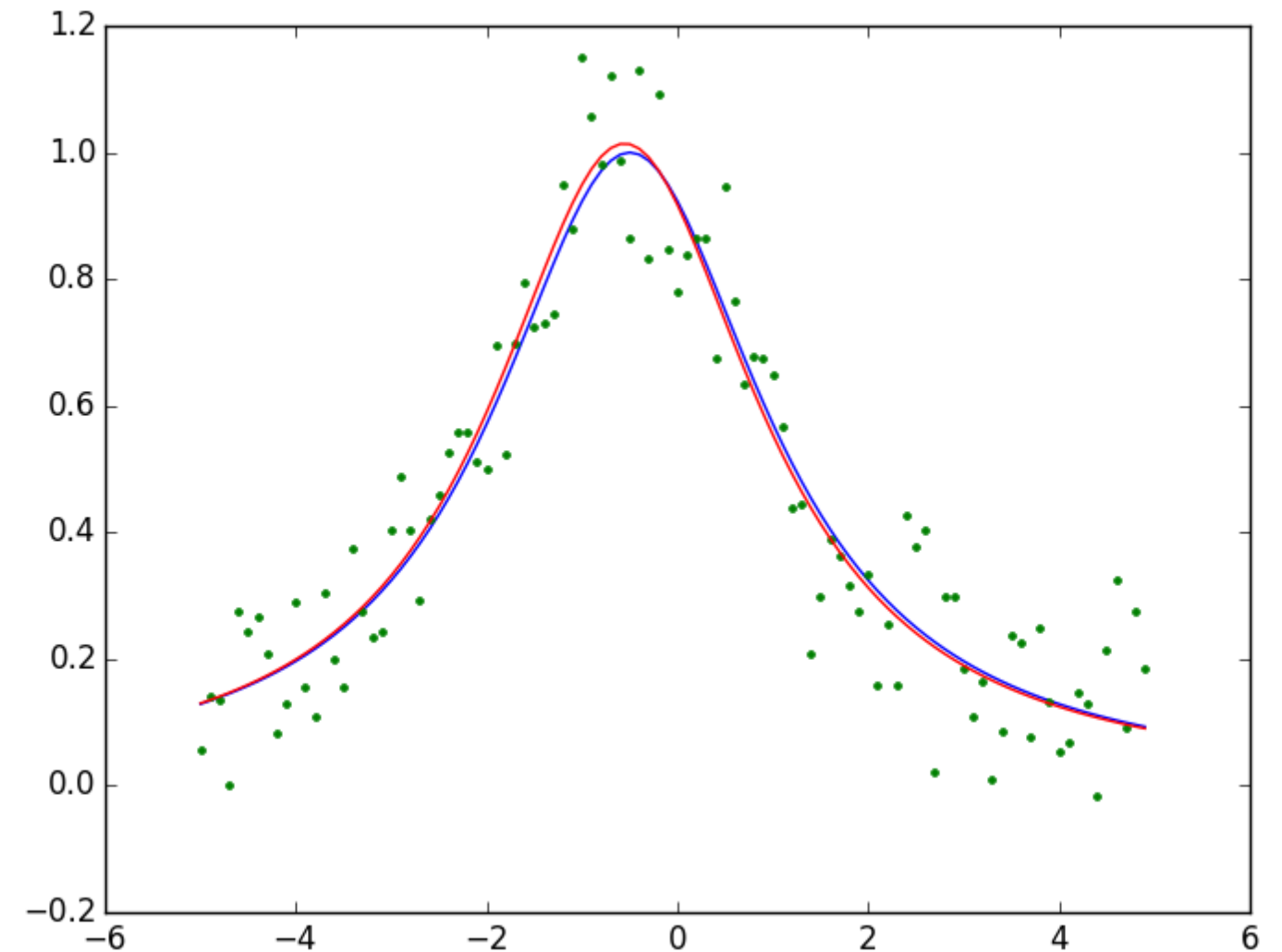
# How to Implement

- Start with a guess for the parameters:  $m_0$ .
- Calculate model  $A(m_0)$  and local gradient  $A_m$ .
- 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  $\delta m$  is “small”. For  $\chi^2$ , change should be  $\ll 1$  (why?).

# Newton's Method in Action

```
def calc_lorentz(p,t):  
    y=p[0]/(p[1]+(t-p[2])**2)  
    grad=numpy.zeros([t.size,p.size])  
    #now differentiate w.r.t. all the parameters  
    grad[:,0]=1.0/(p[1]+(t-p[2])**2)  
    grad[:,1]=-p[0]/(p[1]+(t-p[2])**2)**2  
    grad[:,2]=p[0]*2*(t-p[2])/(p[1]+(t-p[2])**2)**2  
    return y,grad
```

```
for j in range(5):  
    pred,grad=calc_lorentz(p,t)  
    r=x-pred  
    err=(r**2).sum()  
    r=numpy.matrix(r).transpose()  
    grad=numpy.matrix(grad)  
  
    lhs=grad.transpose()*grad  
    rhs=grad.transpose()*r  
    dp=numpy.linalg.inv(lhs)*(rhs)  
    for jj in range(p.size):  
        p[jj]=p[jj]+dp[jj]  
    print p,err
```



# Example: Rational Function Fits

- Rational functions (ratio of polynomials) are often better behaved than equivalent degree-of-freedom polynomial fits.
- Reasons include :
  - high order polynomials shoot off steeply outside constrained region, while rational functions can behave better, even going to zero.
  - Poles lead to Taylor series non-convergence - we've already seen this cause problems. Rational functions can gracefully deal.
- Rational functions are very mildly nonlinear.

# Starting Guess

- One of the hardest things in nonlinear fitting is a good enough starting guess. Rule of thumb is this takes more human time than actual fits.
- Take simple case of as many parameters as points.
- $y_{\text{pred}} = P/Q$  for polynomials  $P$  and  $Q$ . Furthermore, set  $Q_0 = 1$  (why can we do this?). So take  $Q \rightarrow 1 + xQ$ , where the new  $Q$  is one degree lower order.
- But I can solve this!  $y_{\text{pred}}(1 + xQ) = P$ , or  $y_{\text{pred}} = P - y_{\text{pred}} * x * Q$ .
- This is now a linear problem and we can use the full framework of LLS.

# Newton's Method LS Ratfun

- What is gradient of  $\chi^2$  w.r.t numerator parameters?
  - $P_i/Q$
- What is gradient of  $\chi^2$  w.r.t denominator parameters?
  - $-P/Q^2 Q_i$ .
- Could we take second derivatives if we wanted to?



# Levenberg-Marquardt

- Sometimes Newton's method doesn't converge
- In this case maybe we should just go downhill for a bit and then try again
- One way of doing this is Levenberg-Marquardt:  $\text{curve} - \frac{\text{curve}}{\text{curve} + \Lambda \cdot \text{diag}(\text{curve})}$ . For  $\Lambda=0$  this is Newton, for large  $\Lambda$  it's downhill.
- Scheme: if fit is improving, make  $\Lambda$  small. If it isn't working, make  $\Lambda$  larger until it starts working again.
- This and many other minimizers are in `scipy.optimize`.

# MCMC

- Nonlinear problems can be very tricky. Big problem - there can be many local minima, how do I find global minimum? Linear problem easier since there's only one minimum.
- One technique: Markov-Chain Monte Carlo (MCMC). Picture a particle bouncing around in a potential. It normally goes downhill, but sometimes goes up.
- Solution: simulate a thermal particle bouncing around, keep track of where it spends its time.
- Key theorem: such a particle traces the PDF of the model parameters, and distribution of the full likelihood is the same as particle path.
- Using this, we find not only best-fit, but confidence intervals for model parameters.

# MCMC, ctd.

- Detailed balance: in steady state, probability of state going from a to b is equal to going from b to a (“detailed balance”).
- Algorithm. Start a particle at a random position. Take a trial step. If trial step improves  $\chi^2$ , take the step. If not, *sometimes* accept the step, with probability  $\exp(-0.5\delta\chi^2)$ .
- After waiting a sufficiently long time, take statistics of where particle has been. This traces out the likelihood surface.

# MCMC Driver

```
def run_mcmc(data, start_pos, nstep, scale=None):
    nparam=start_pos.size
    params=numpy.zeros([nstep, nparam+1])
    params[0, 0:-1]=start_pos
    cur_chisq=data.get_chisq(start_pos)
    cur_pos=start_pos.copy()
    if scale==None:
        scale=numpy.ones(nparam)
    for i in range(1, nstep):
        new_pos=cur_pos+get_trial_offset(scale)
        new_chisq=data.get_chisq(new_pos)
        if new_chisq<cur_chisq:
            accept=True
        else:
            delt=new_chisq-cur_chisq
            prob=numpy.exp(-0.5*delt)
            if numpy.random.rand()<prob:
                accept=True
            else:
                accept=False
        if accept:
            cur_pos=new_pos
            cur_chisq=new_chisq
        params[i, 0:-1]=cur_pos
        params[i, -1]=cur_chisq
    return params
```

- Here's a routine to make a fixed-length chain.
- As long as our data class has a `get_chisq` routine associated with it, it will work.
- Big loop: take a trial step, decide if we accept or not. Add current location to chain.

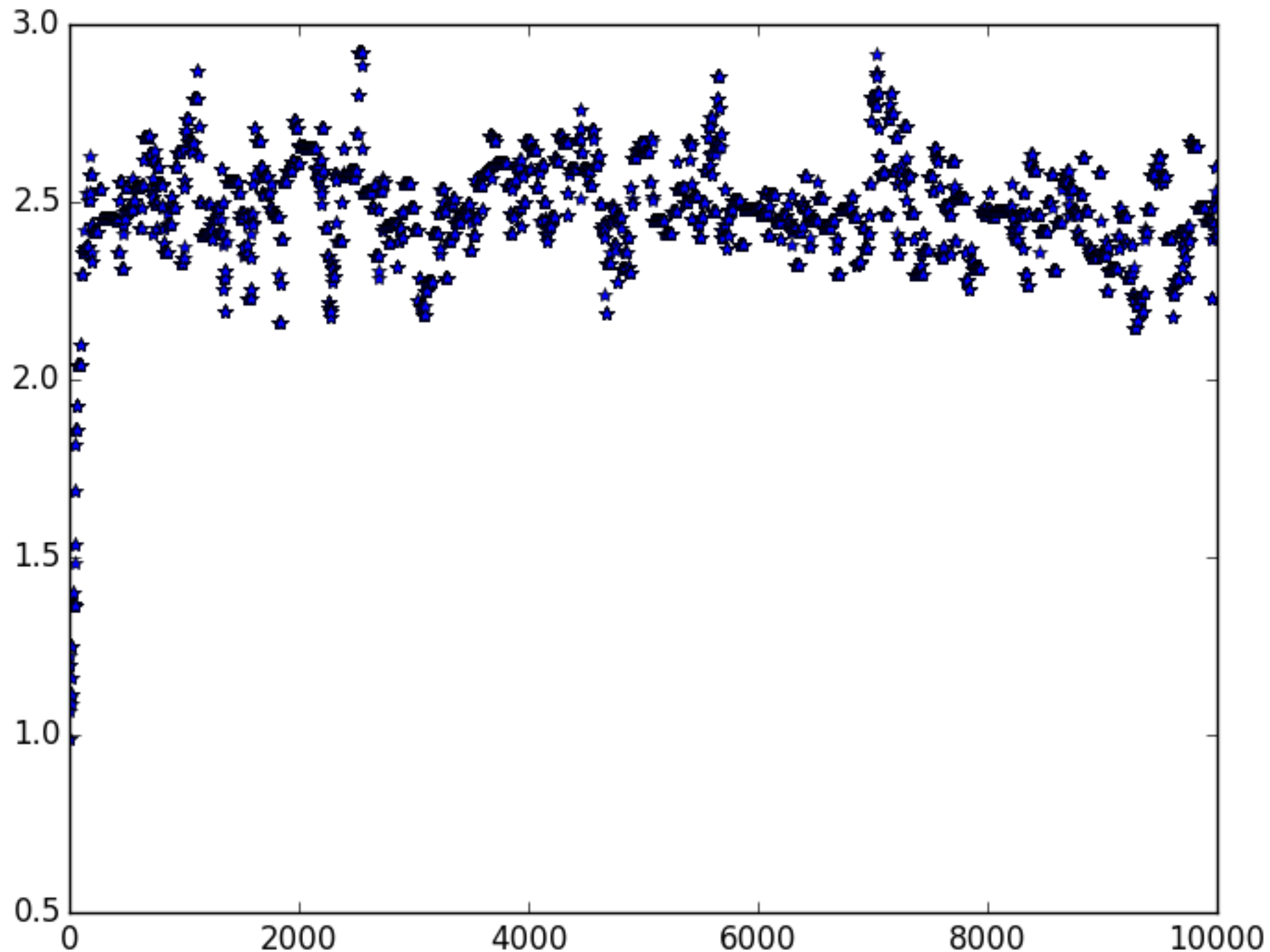
# Output

```
if __name__=='__main__':  
    #get a realization of a gaussian, with noise added  
    t=numpy.arange(-5,5,0.01)  
    dat=Gaussian(t,amp=2.5)  
  
    #pick a random starting position, and guess some errors  
    guess=numpy.array([0.3,1.2,0.3,-0.2])  
    scale=numpy.array([0.1,0.1,0.1,0.1])  
    nstep=10000  
    chain=run_mcmc(dat,guess,nstep,scale)  
    #nn=numpy.round(0.2*nstep)  
    #chain=chain[nn:,:]  
  
    #pull true values out, compare to what we got  
    param_true=numpy.array([dat.sig,dat.amp,dat.cent,dat.offset])  
    for i in range(0,param_true.size):  
        val=numpy.mean(chain[:,i])  
        scat=numpy.std(chain[:,i])  
        print [param_true[i],val,scat]
```

```
>>> execfile('fit_gaussian_mcmc.py')  
[0.5, 0.48547765442013036, 0.031379203158769478]  
[2.5, 2.5972175915216877, 0.16347041731916298]  
[0.0, 0.039131754036757782, 0.030226015774759099]  
[0.0, 0.0031281155414288856, 0.03983540490701154]
```

- Main: set up data first. Then call the chain function. Finally, compare output fit to true values.
- Parameter estimates are just the mean of the chain. Parameter errors are just the standard deviation of the chain.

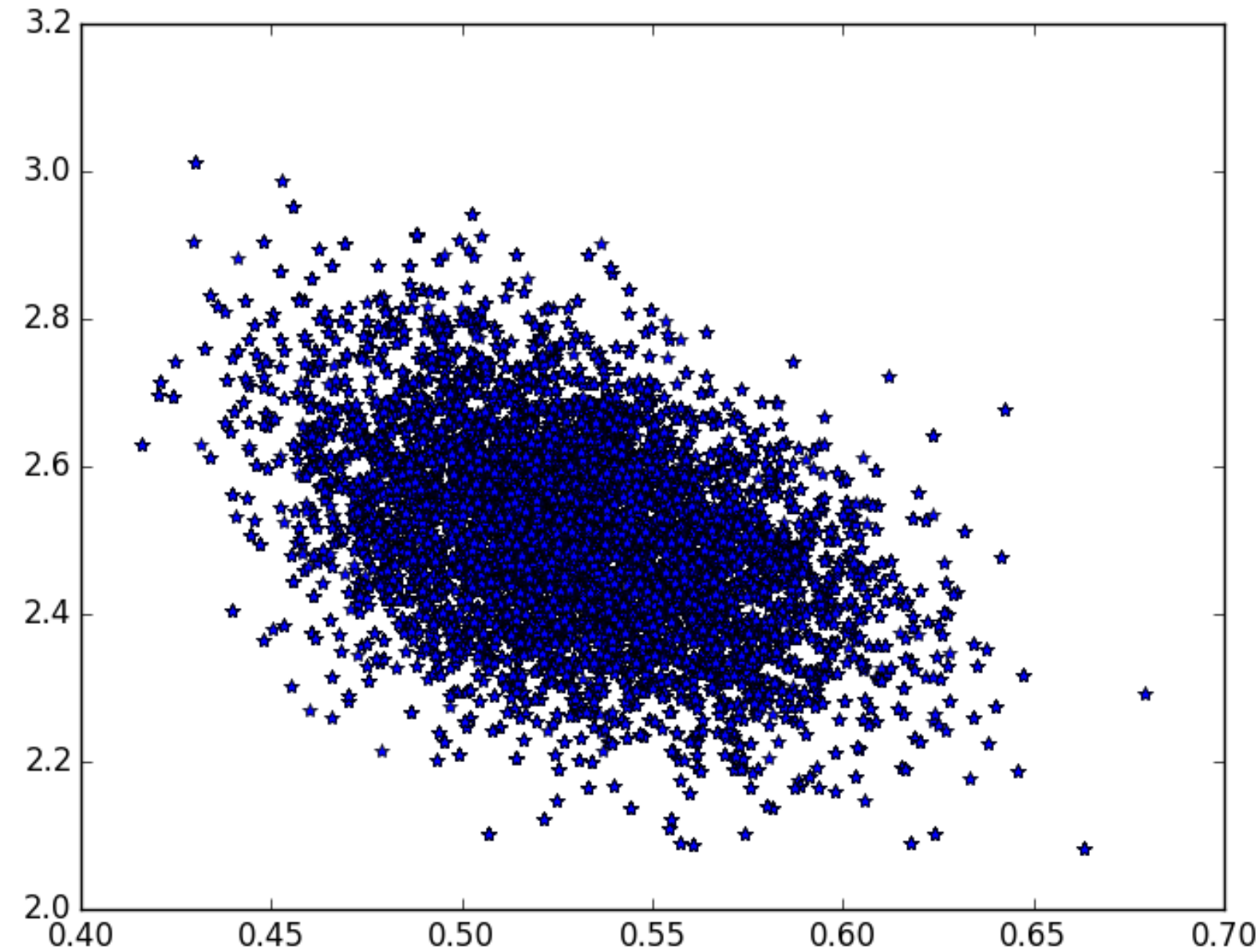
# What Chain Looks Like



- Here's the samples for one parameter. Note big shift at beginning: we started at a wrong position, but chain quickly moved to correct value.
- Initial part is called “burn-in”, and should be removed from chain.

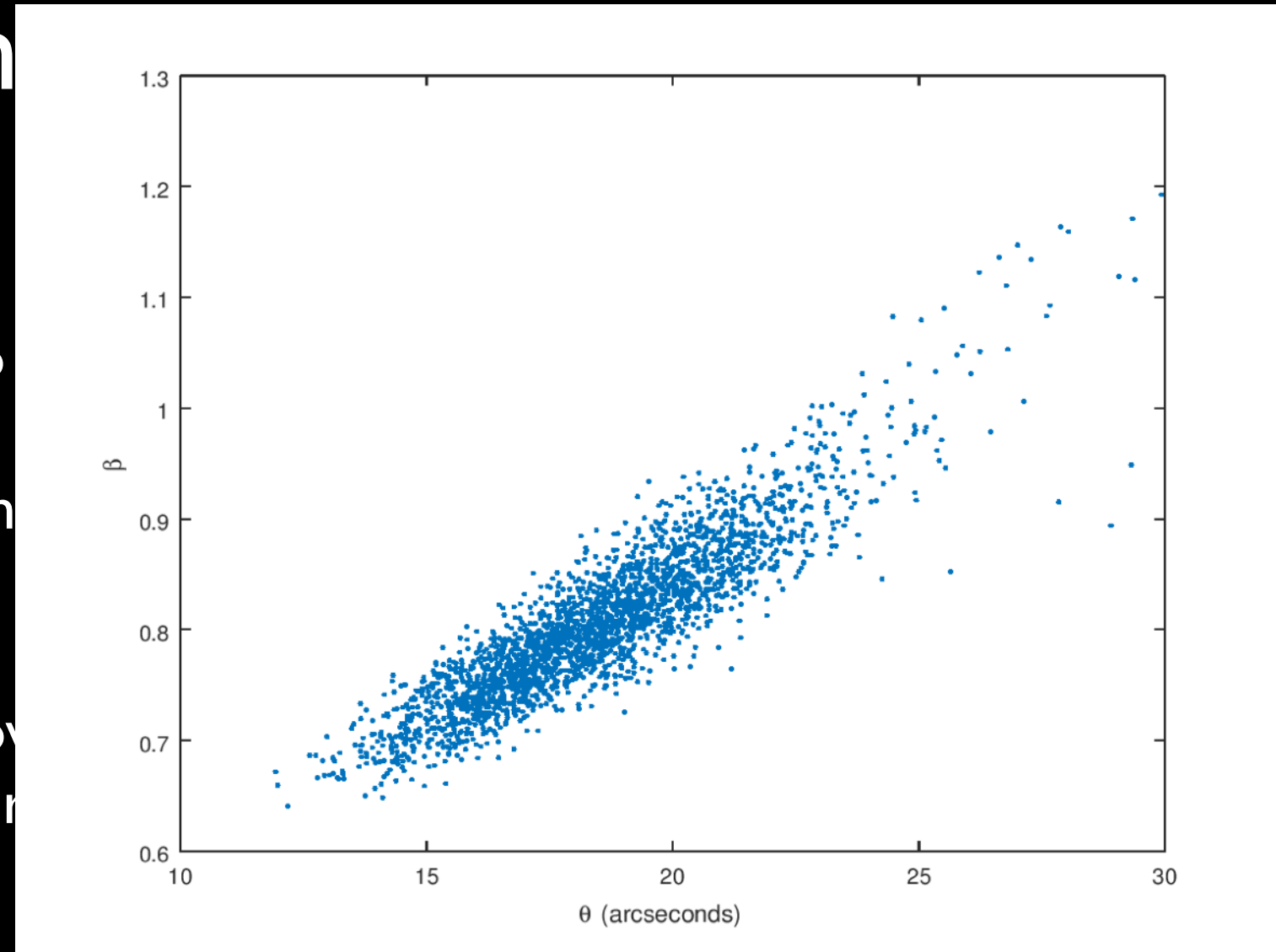
# Covariances

- Naturally get parameter covariances out of chains. Just look at covariance of samples!
- Very powerful way of tracing out complicated multi-dimensional likelihoods.



# You Gotta Know Wh

- Trick in doing MCMC is knowing when to
- One standard technique is to run many chains vs. expected scatter.
- Chains *work* independent of step size. How step size. Too large steps, we spend all our and we only move around slowly, so takes
- If parameters are correlated, you probably want steps to know about that. (See bit on correlated variates for technical details.)
- Good rule of thumb is you want to accept ~25% of your samples. Run for a bit, then adjust step size and start new chain.





# Single-Chain Convergence

- Chains eventually forget their past.
- If you plot chain samples, then eventually they should look like white noise
- FT of converged chain should be flat for large scales (low  $k$ )
- top: unconverged chain.  
bottom: converged chain.

