Phys 512
Lecture 4

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.
- I suggest you pick one and sick with it.

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 sigular 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=LLT.

χ^2

- The PDF of a Gaussian is $\exp(-0.5(x-\mu)^2/\sigma^2)/\sqrt{2\pi\sigma^2}$ with mean μ and standard deviation σ.
- 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 μ_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 χ^2. We can find the maximum likelihood model by minimizing χ^2.

Linear least-squares

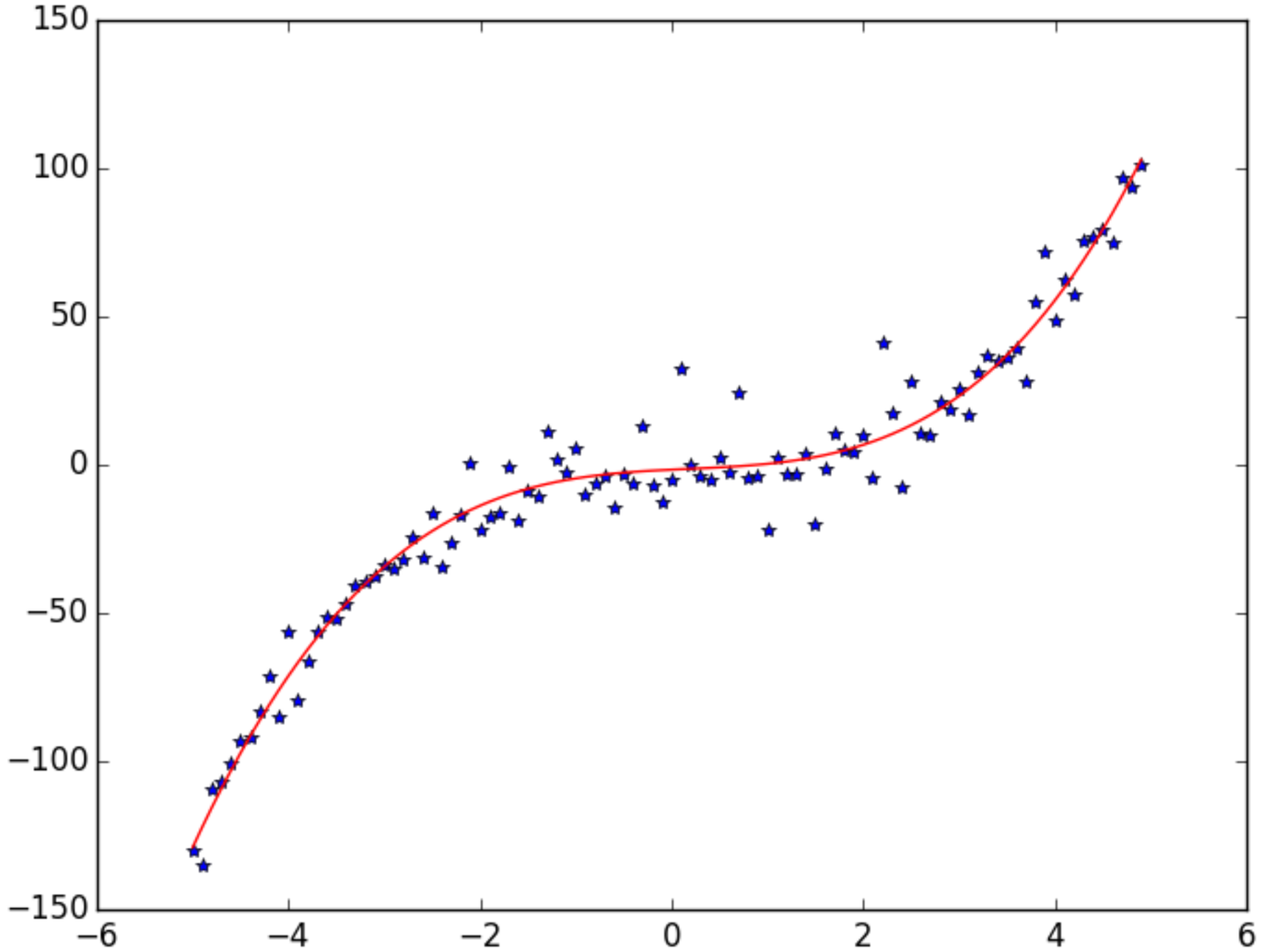
- Rewrite $\chi 2$ with matrices: $(x-\mu)^T N^{-1}(x-\mu)$ for noise covariance matrix N. If N has diagonal elements σ^2, this is identical to previous.
- Let's take simple case that our model depends linearly on a small number of parameters: $\mu_i=\sum A_{ij}m_j$ for model parameters m and matrix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

Least Squares: $\chi^2=(x-Am)^T N^{-1}(x-Am)$

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

Exa

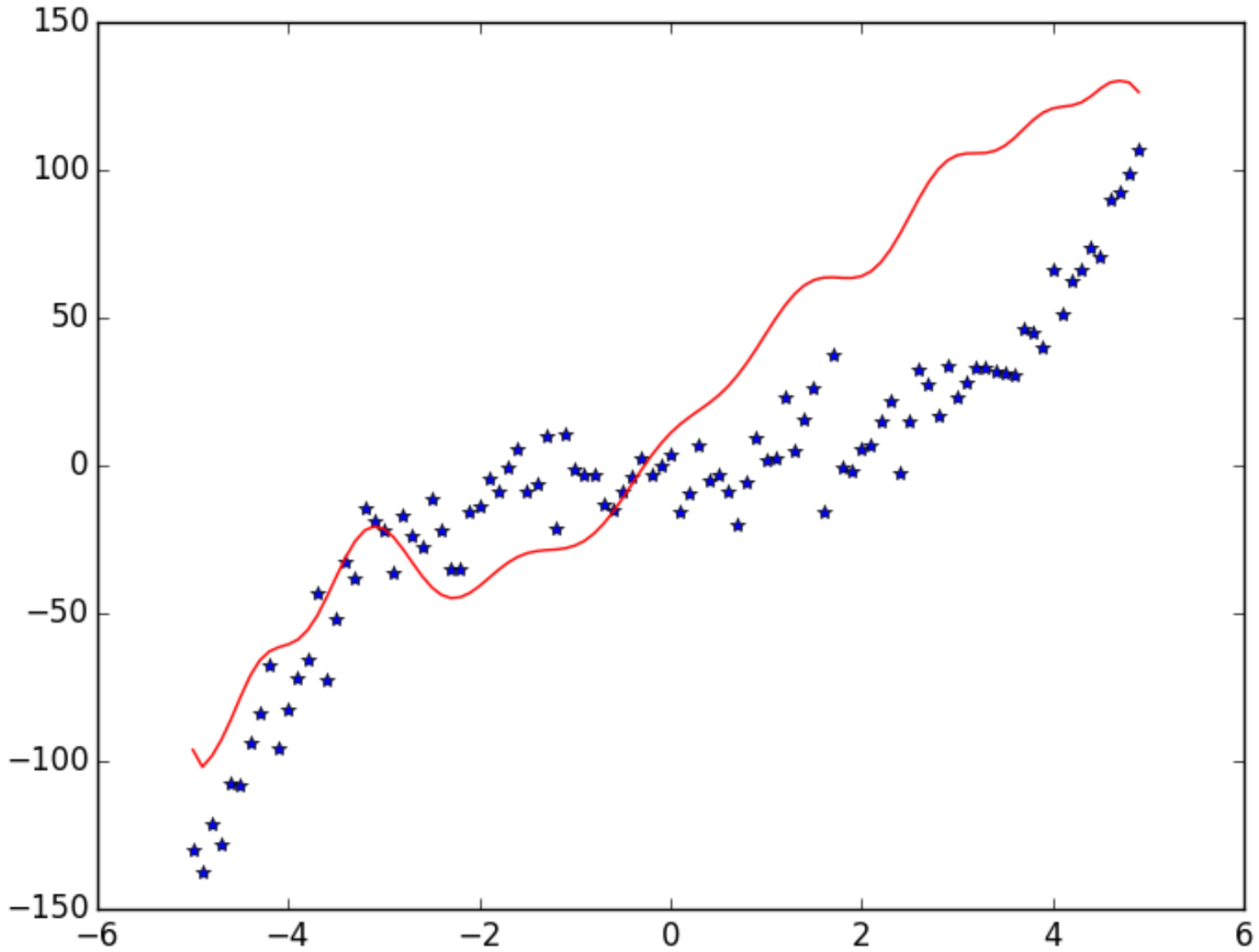
```
import numpy
from matplot
t=numpy.aran
x_true=t**3-
x=x_true+10*

npoly=5  #le
ndata=t.size
A=numpy.zero
A[:,0]=1.0
for i in ran
    A[:,i]=A
#Let's ignor
#m=(A^TA)^{-
A=numpy.matr
d=numpy.matr
lhs=A.transp
rhs=A.transp
fitp=numpy.l
pred=A*fitp
plt.clf();pl
plt.draw()
```



tc. 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 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 $A=USV^T$, 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: $(USV^T)^TUSV^Tm=(USV^T)^Tx$. $VSU^TUSV^Tm=VSU^Tx$
- U^TU=identity, so cancels. $VS^2V^Tm=VSU^Tx$. S^2 squares the condition number, so that was bad. We can analytically cancel left-hand V and one copy of S: $SV^Tm=U^Tx$. Then $m=VS^{-1}U^Tx$
- 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. 100th 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

```python
import numpy
def get_legendre_mat(t,npoly):
    #key relation:  (n+1)P_(n+1)=(2n+1)tP_n - nP_(n-1)
    mat=numpy.zeros([t.size,npoly])
    mat[:,0]=1.0
    if npoly>1:
        mat[:,1]=t
    for i in range(1,npoly-1):
        mat[:,i+1]=((2.0*i+1)*t*mat[:,i]-i*mat[:,i-1])/(i+1.0)
    mat=numpy.matrix(mat)
    return mat

if __name__=='__main__':
    dt=0.001
    t=numpy.arange(-5+dt/2.0,5,dt)
    for npoly in numpy.arange(5,100,5):
        mat=get_legendre_mat(t/5,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) + ' Legendre matrix has co
```

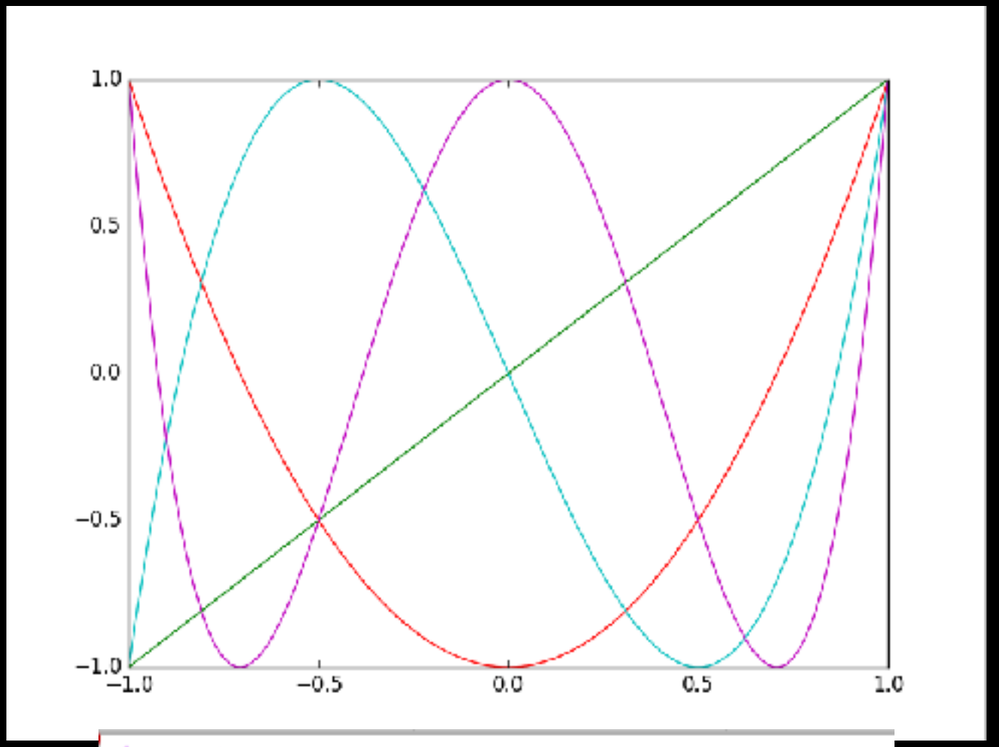
```
>>> execfile('cond_example_legendre.py')
5 order poynomial matrix has condition number 9.0000026999767648
10 order poynomial matrix has condition number 19.00005415034467
15 order poynomial matrix has condition number 29.000294368595334
20 order poynomial matrix has condition number 39.000963550102306
25 order poynomial matrix has condition number 49.002402810934953
30 order poynomial matrix has condition number 59.00505642599736
35 order poynomial matrix has condition number 69.009477057966521
40 order poynomial matrix has condition number 79.016336167849929
45 order poynomial matrix has condition number 89.026442681092632
50 order poynomial matrix has condition number 99.040774215288522
55 order poynomial matrix has condition number 109.06052705286851
60 order poynomial matrix has condition number 119.08719407465288
65 order poynomial matrix has condition number 129.12268493401126
70 order poynomial matrix has condition number 139.16951135267718
75 order poynomial matrix has condition number 149.23107516419981
80 order poynomial matrix has condition number 159.31212210407367
85 order poynomial matrix has condition number 169.41946763316335
90 order poynomial matrix has condition number 179.56317279103277
95 order poynomial matrix has condition number 189.75845697330035
>>>
```

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: $T_n=\cos(n*\arccos(x))$, $-1 \le x \le 1$
- 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 ± 1, and are more-or-less uniform between ± 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?

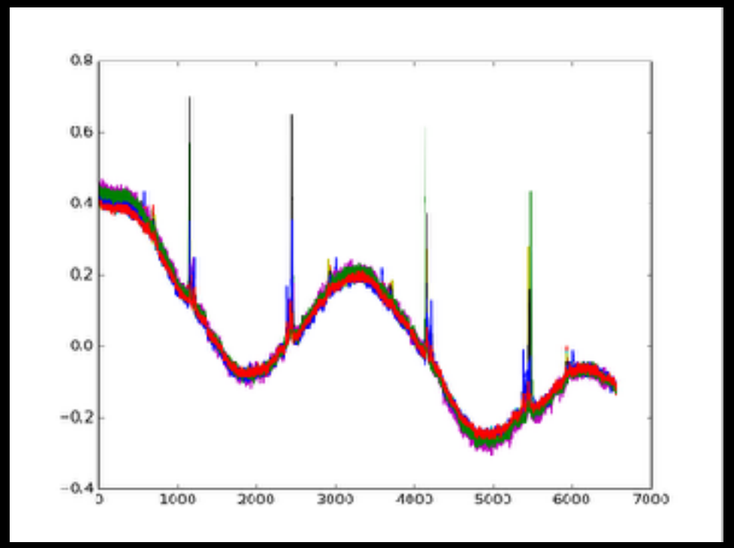
```
>>> execfile("fit_chebyshev.py")
  0  -3.0424e-01   9.6317e-16
  2  -9.7087e-01   2.5647e-16
  4   3.0285e-01  -2.9023e-16
  6  -2.9092e-02  -1.1972e-16
  8   1.3922e-03  -2.7269e-16
 10  -4.0190e-05  -3.2028e-17
 12   7.7828e-07  -6.0301e-17
 14  -1.0827e-08  -5.7433e-16
 16   1.1351e-10  -3.4716e-16
 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
```

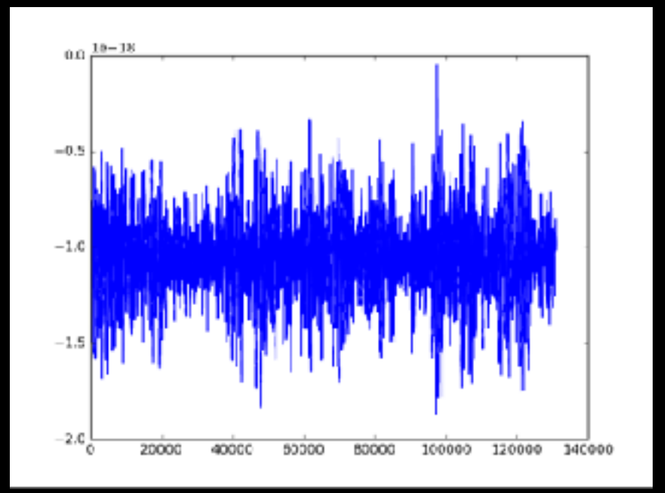
```
import numpy
from numpy.polynomial import chebyshev
x=numpy.arange(-1,1,1e-3)
y=numpy.cos(x*numpy.pi)
order=50
pp=chebyshev.chebfit(x,y,order)
for i in range(0,order,2):
    #note formatted output here, similar to C
    print '%3d %12.4e %12.4e'% (i,pp[i],pp[i+1])
    #print 2*i,pp[2*i],pp[2*i+1]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

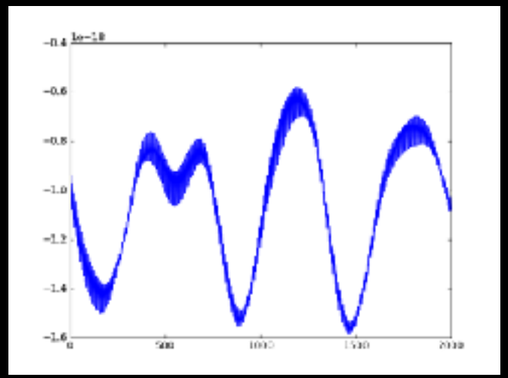
Correlated Noise

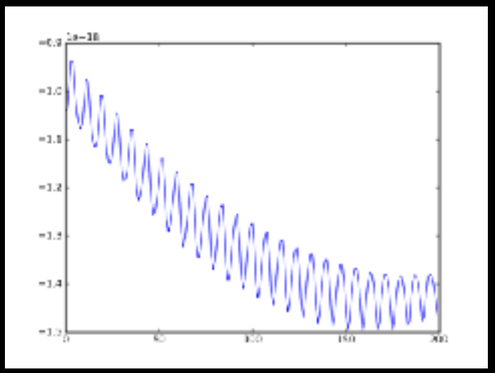
- So far, we have assumed that the noise is independent between data sets.

- 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->Vr, N->$VNV^T$, and $\chi 2$ expression is unchanged in new, rotated space.
- Furthermore, (fairly) easy to show that $<N_{ij}>=<r_ir_j>$.
- So, we can work in this new, rotated space without ever referring to original coordinates. Just need to calculate noise covariances N_{ij}.

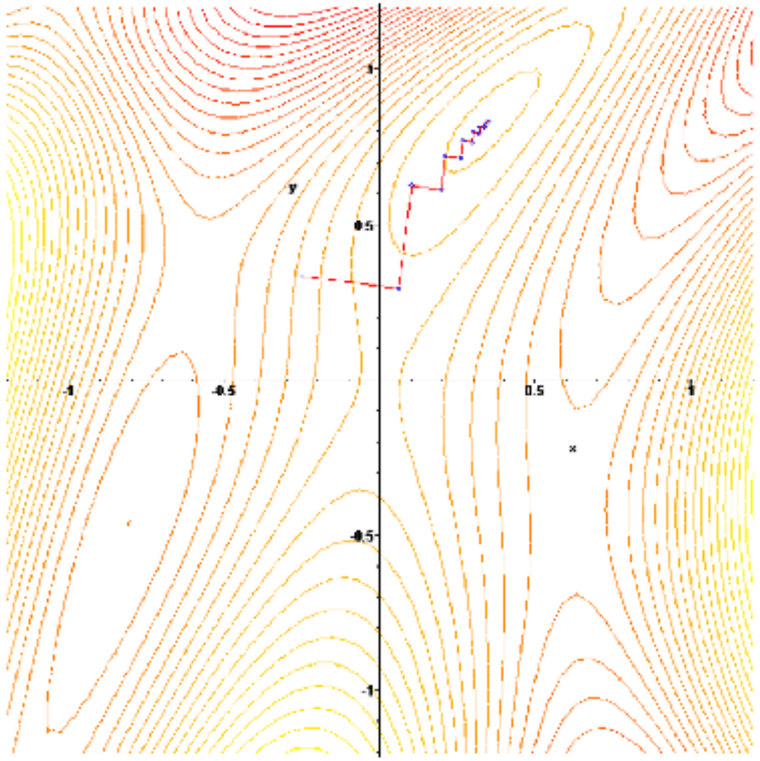
Nonlinear Fitting

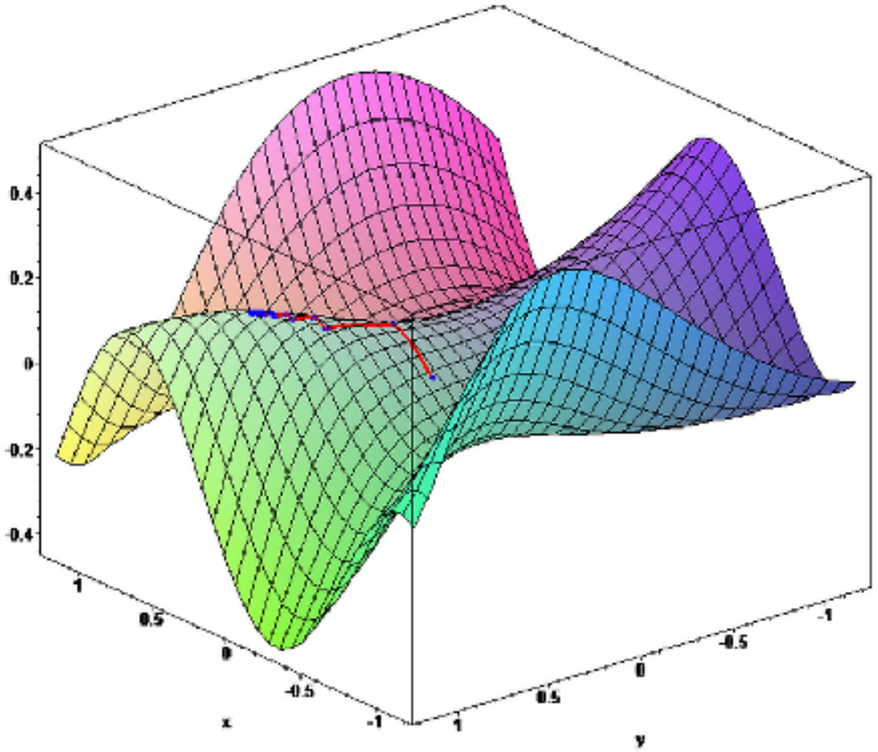
- Sometimes data depend non-linearly on model parameters
- Examples are Gaussian and Lorentzian $(a/(b+(x-c)^2)$
- Often significantly more complicated - cannot reason about global behaviour from local properties. May be multiple local minima
- Many methods reduce to how to efficiently find the "nearest" 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.

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

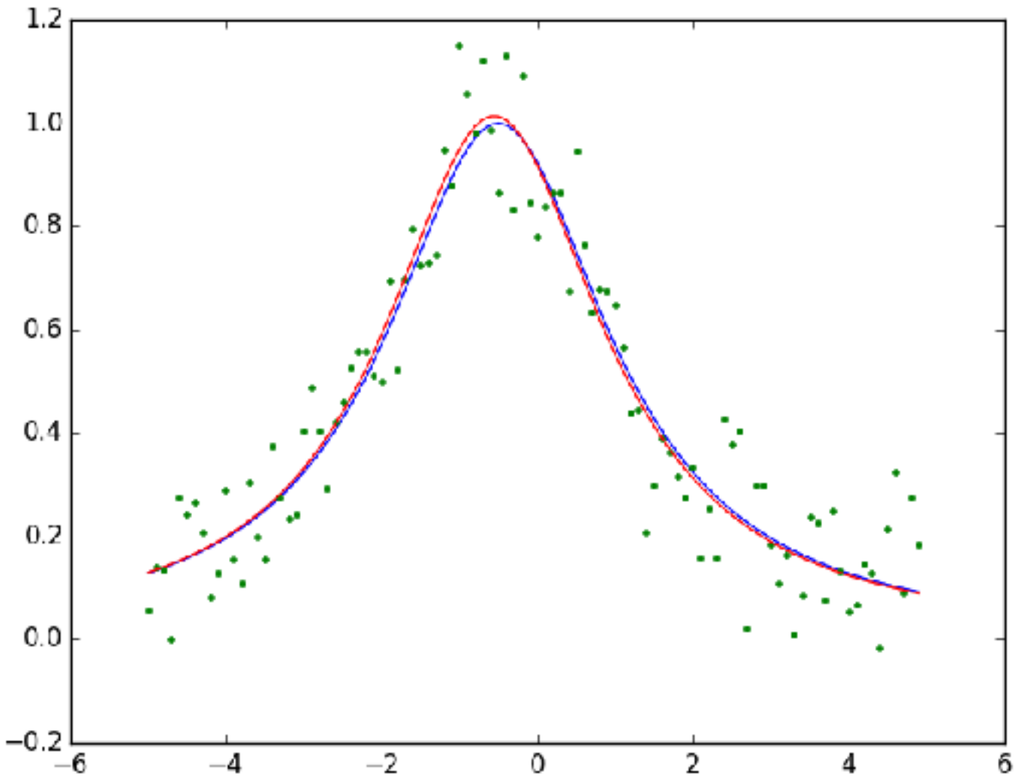
How to Implement

- Start with a guess for the parameters: m_0.
- Calculate model $A(m_0)$ and local gradient A_m.
- 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?).

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
```



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: curve->curve+Λ*diag(curve). For Λ=0 this is Newton, for large Λ it's downhill.
- Scheme: if fit is improving, make Λ small. If it isn't working, make Λ 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 χ^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.

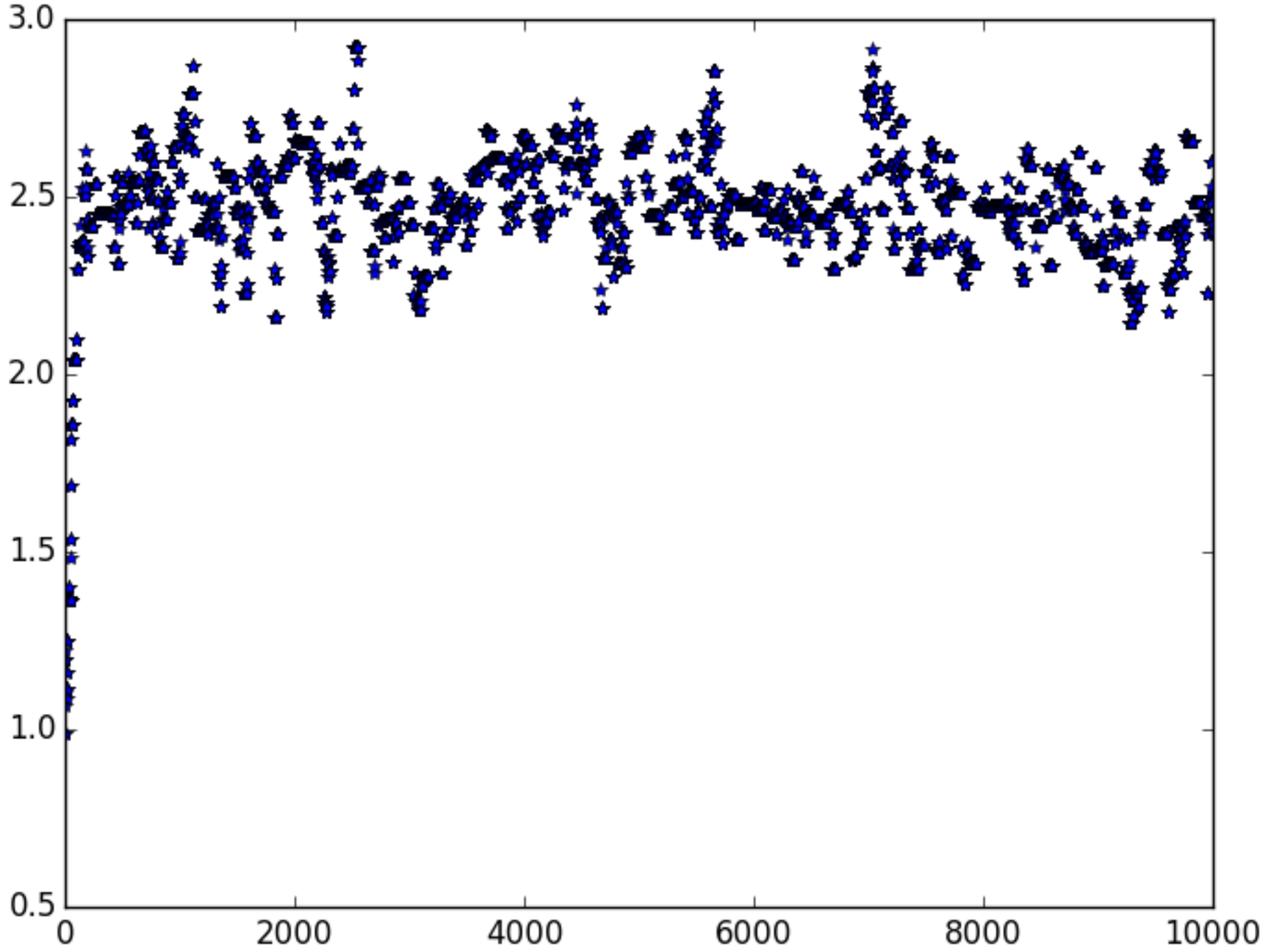
Output

```
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, 0.039835404907011 54]
```

- 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.

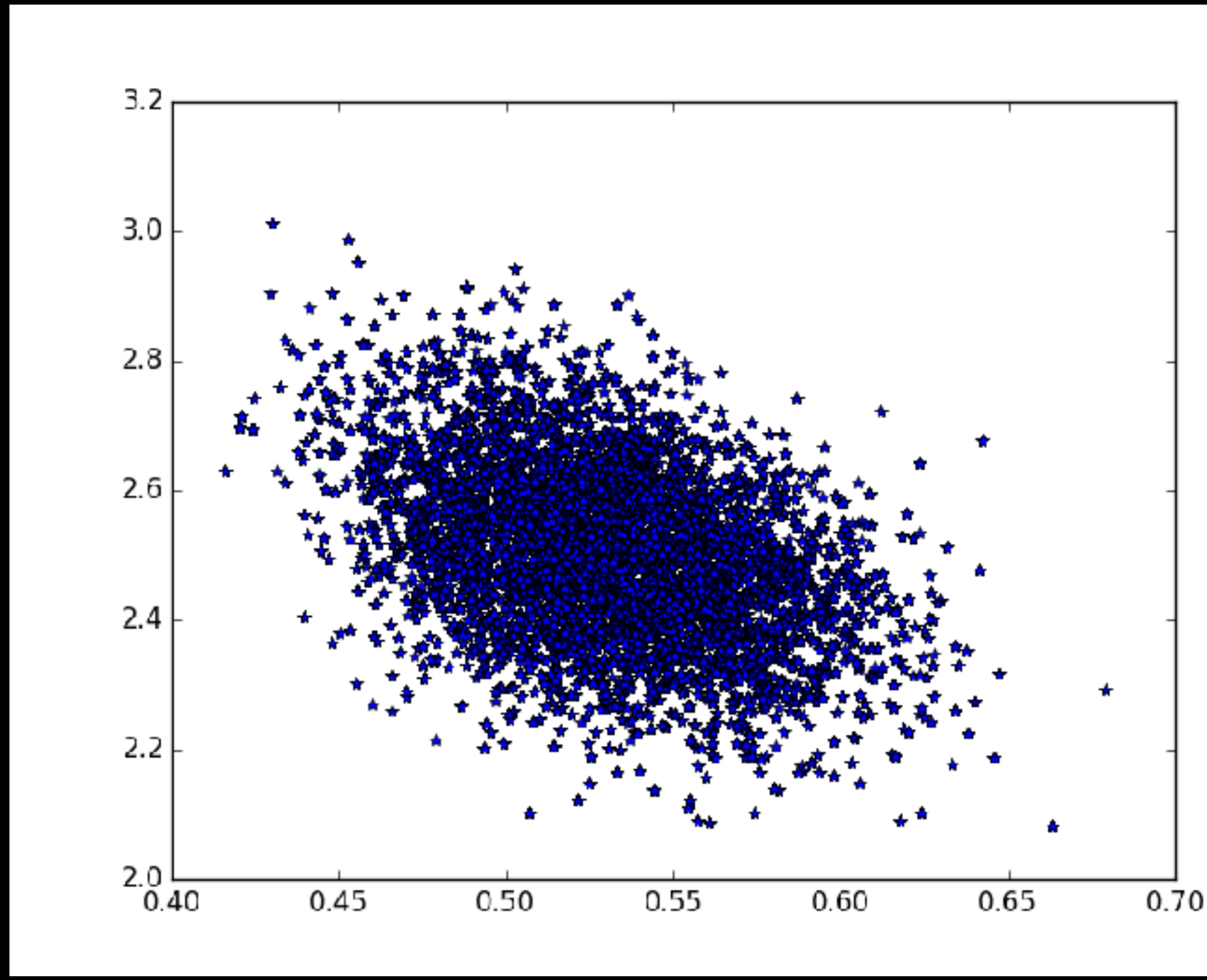
What Chain Looks Like



- 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.

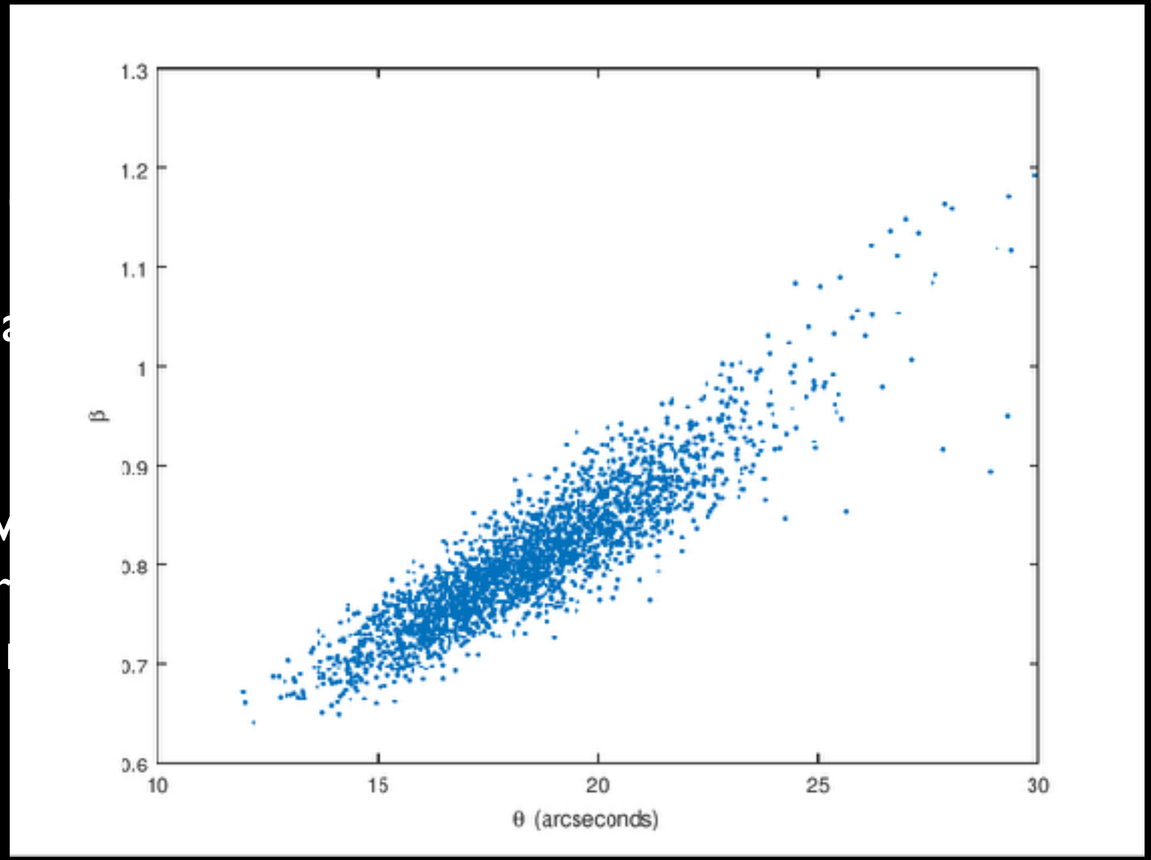
Covariances

- 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 When to Fold 'em

- Trick in doing MCMC is knowing when to
- One standard technique is to run many cha
  vs. expected scatter.
- Chains *work* independent of step size. Hov
  step size. Too large steps, we spend all our
  and we only move around slowly, so takes
- If parameters are correlated, you probably
- 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.