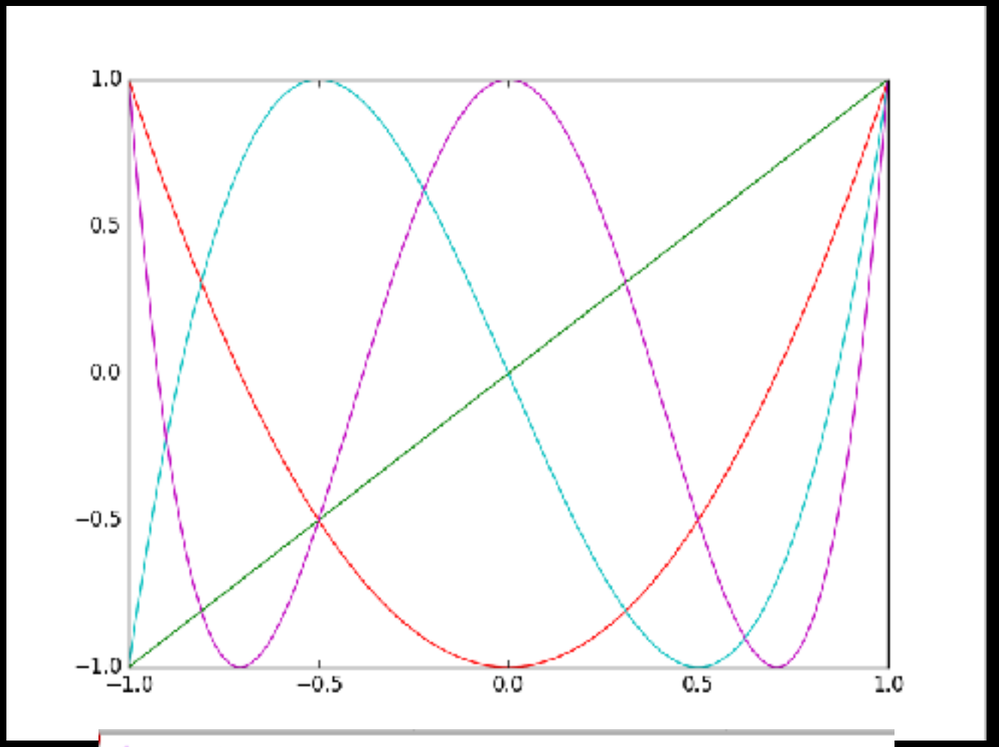
Phys 512
Lecture 6

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.

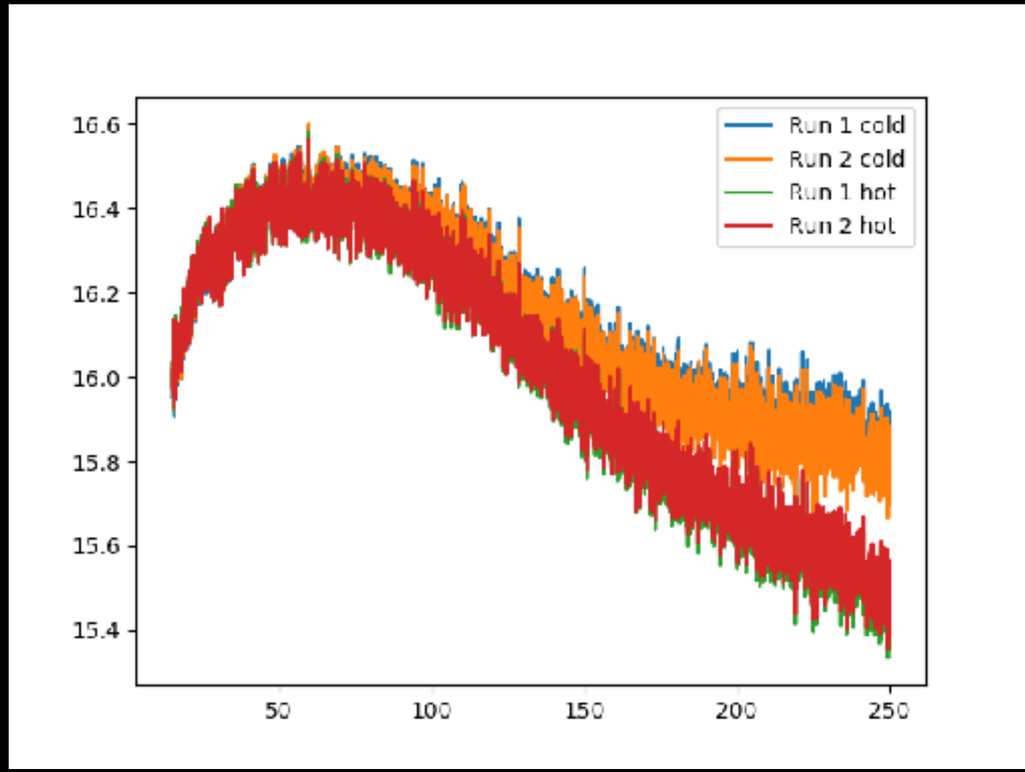
- Bound on error mean Chebyshev polynomials are great for modelling functions. Many implementations of functions in e.g. math library are based on them.
- Look at e.g. cos fit from -pi to pi.
- Why are odd terms zero?
- How many terms would you keep for single precision? For double?

```
>>> 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]
```

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?



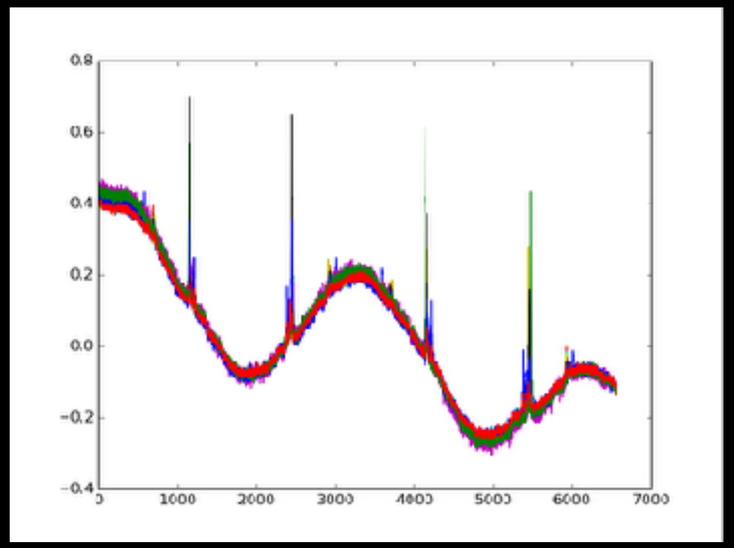
LLS Errors

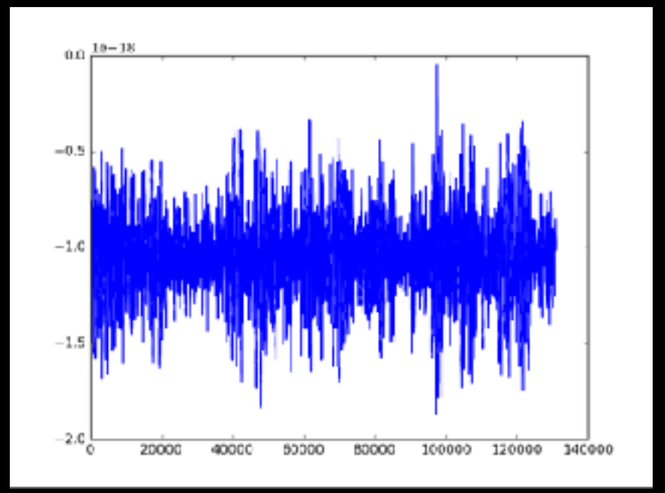
- $d=Am_t+n$ where d are the observed data, Am_t is the true data, and n is noise.
- We want to find $<(m-m_t)^2>$
- Well, $m = (A^TN^{-1}A)^{-1}A^TN^{-1}d$. $m_t=(A^TN^{-1}A)^{-1}A^TN^{-1}(d_t)$ where dt is true (noiseless) data.
- $\delta_m=m-m_t = (A^TN^{-1}A)^{-1}A^TN^{-1}n$.
- Expectation clearly = 0 (since $<n>=0$), but $< (m-m_t)(m-m_t)^T>=(A^TN^{-1}A)^{-1}$.
- So, errors on my parameters are sqrt(diagonal) of that. Which, happily, we already have!
- Errors in reconstructed data: $<(A\delta_m)(A\delta_m)^T>=A\ (A^TN^{-1}A)^{-1}\ A^T$.

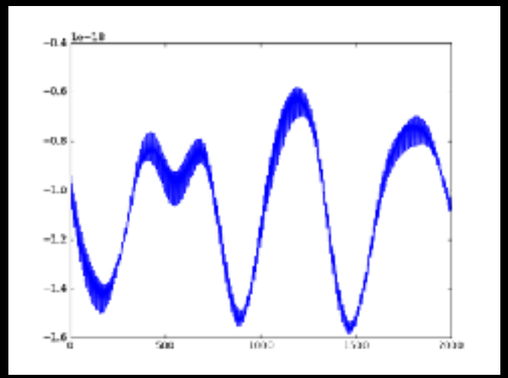
Correlated Noise

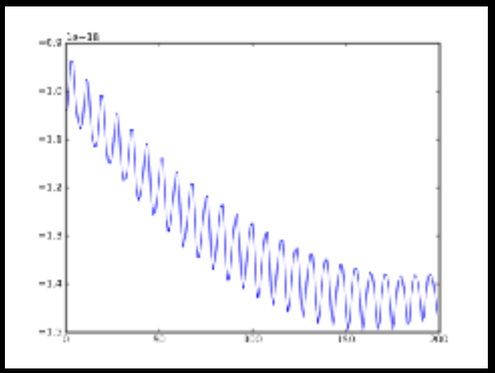
- So far, we have assumed that the noise is independent between data sets.

- 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}.

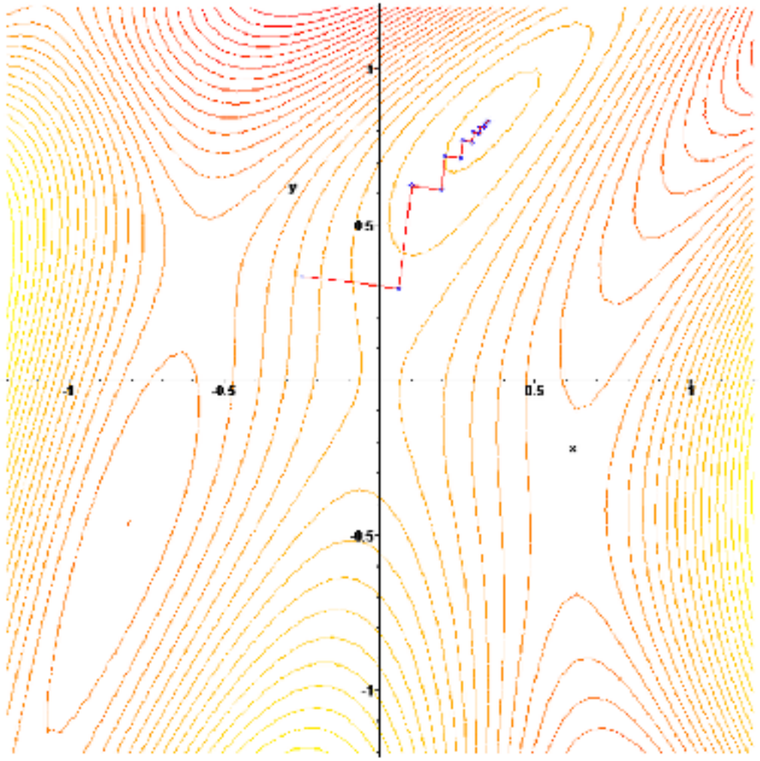
Nonlinear Fitting

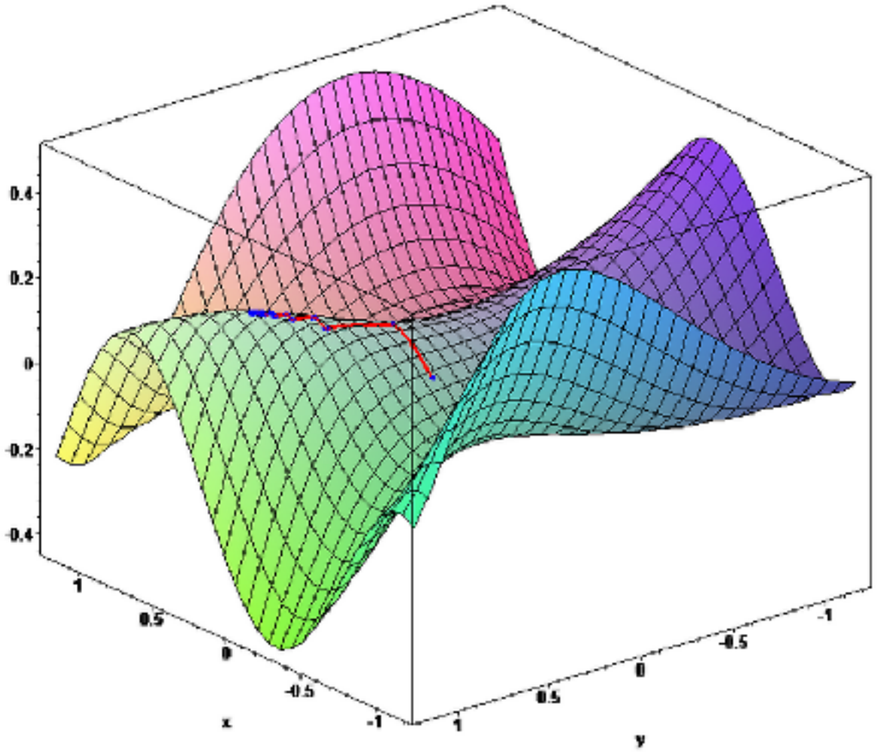
- Sometimes data depend non-linearly on model parameters
- Examples are Gaussian and Lorentzian $(a/(b+(x-c)^2)$
- Often significantly more complicated - cannot reason about global behaviour from local properties. May be multiple local minima
- Many methods reduce to how to efficiently find the "nearest" 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! $A_m^T N^{-1} A_m\ \delta m = A_m^T N^{-1} r$

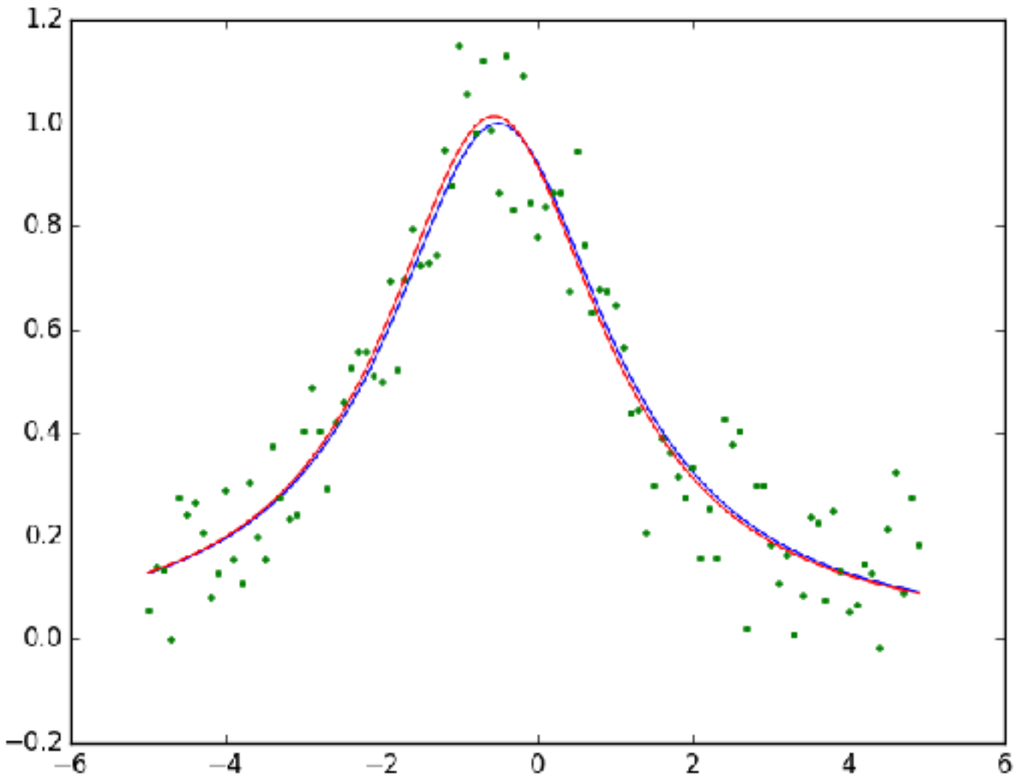
How to Implement

- 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
```



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_pred=P/Q for polynomials P and Q. Furthermore, set $Q_0=1$ (why can we do this?). So take Q -> 1+xQ, where the new Q is one degree lower order.
- But I can solve this! y_pred(1+xQ)=P, or y_pred=P-y_pred*x*Q.
- This is now a linear problem and we can use the full framework of LLS.

Newton's Method LS Ratfuns

- What is gradient of χ^2 w.r.t numerator parameters?
  - Pi/Q
- What is gradient of χ^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: 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.

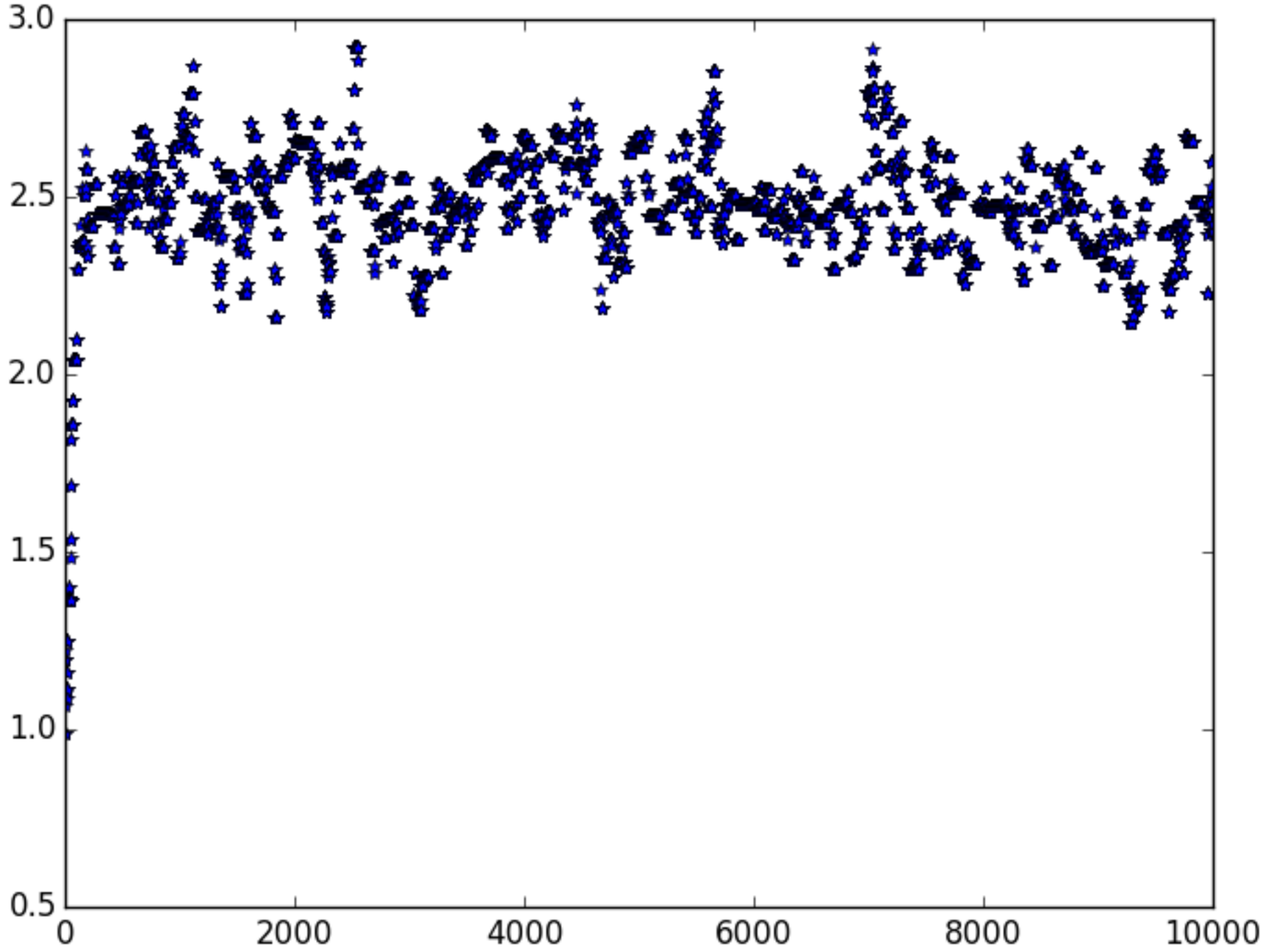
Output

```
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.

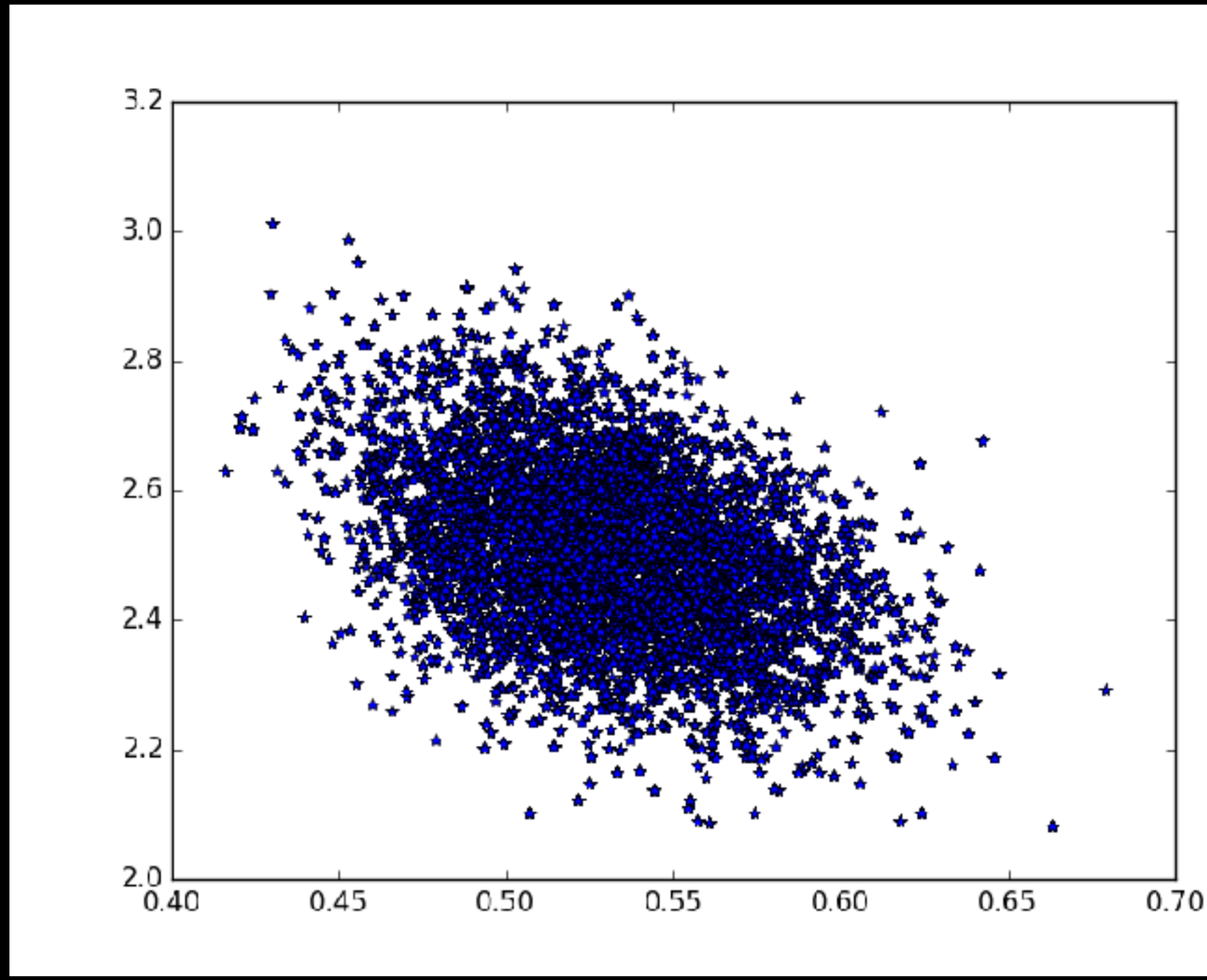
What Chain Looks Like



- 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.

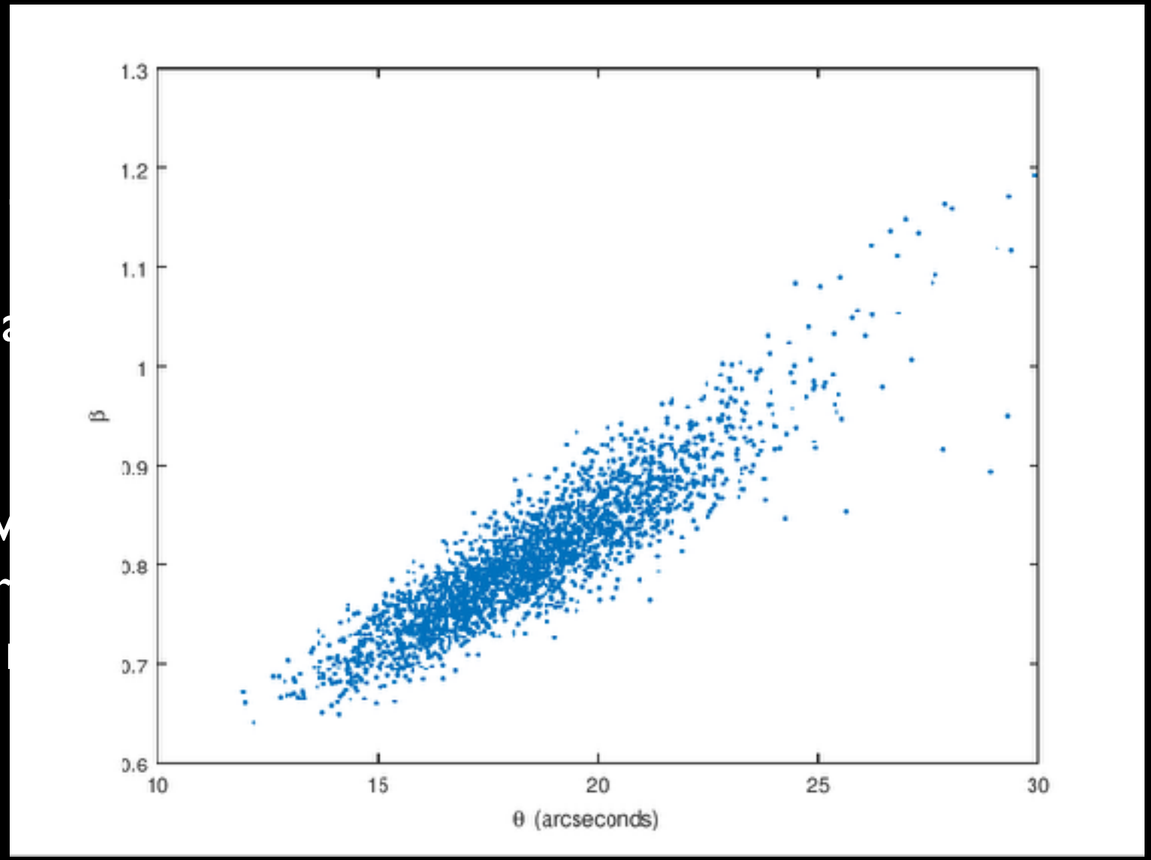
Covariances

- 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.