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.

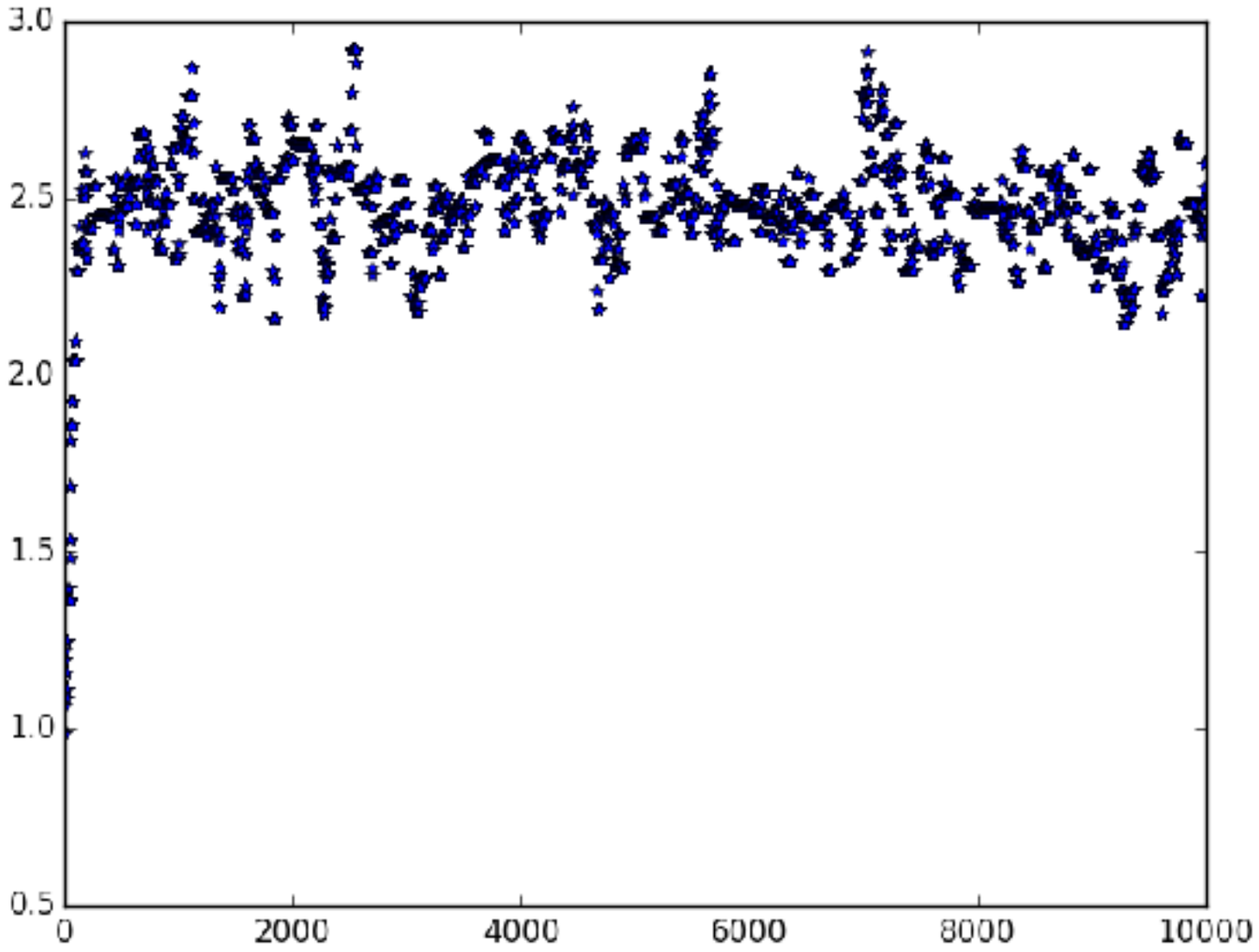
Output

```
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, 0.48547765442813036, 0.031379203158769478]
[2.5, 2.5972175915216877, 0.16347041731916298]
[0.0, 0.039131754036757782, 0.030226015774759099]
[0.0, 0.0031281155414288856, 0.039835404907011541]
```

- 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

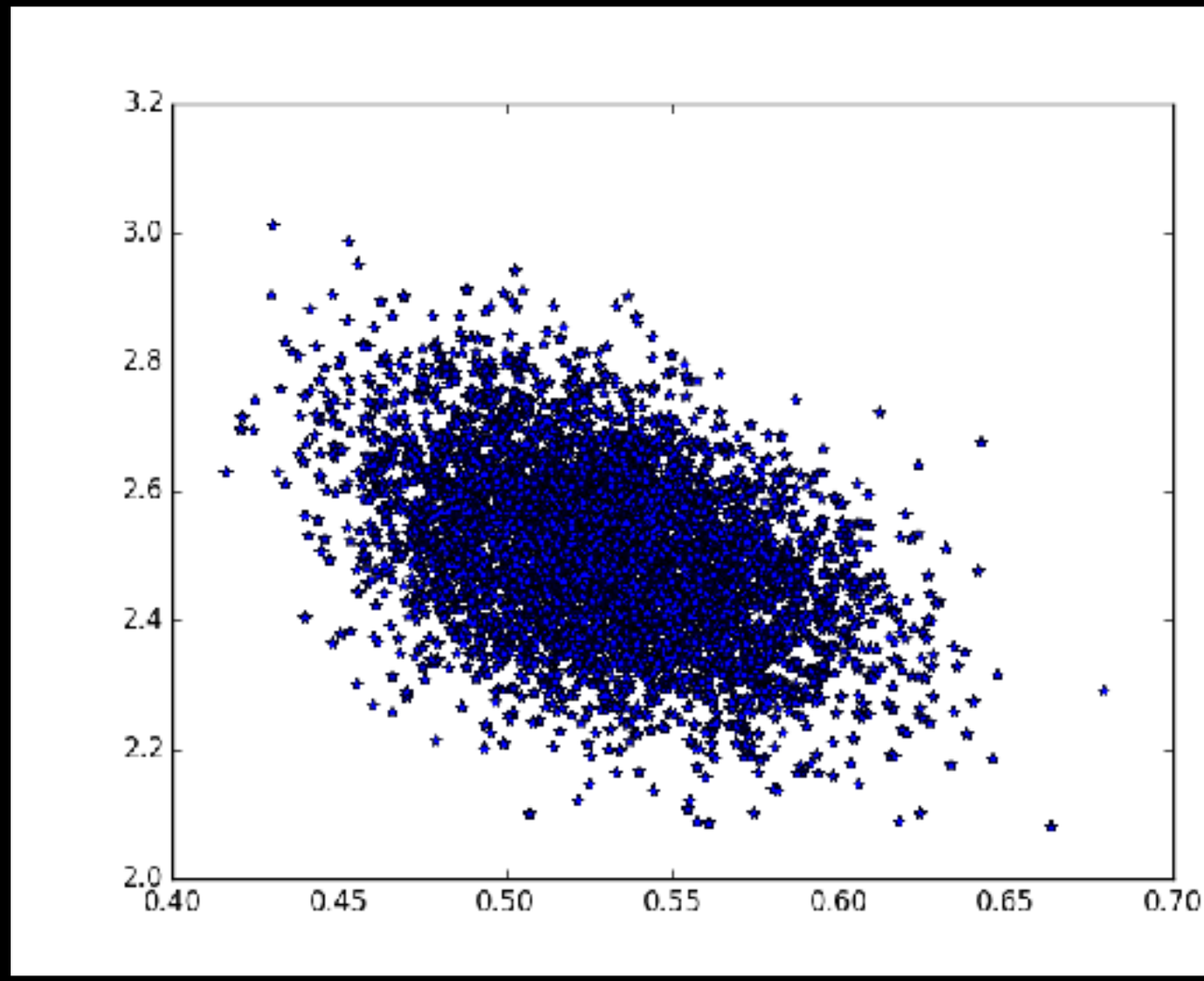


- Here's the samples for one parameter. Note big shift at beginning: we started at a wrong position, but chain quickly moved to correct value.

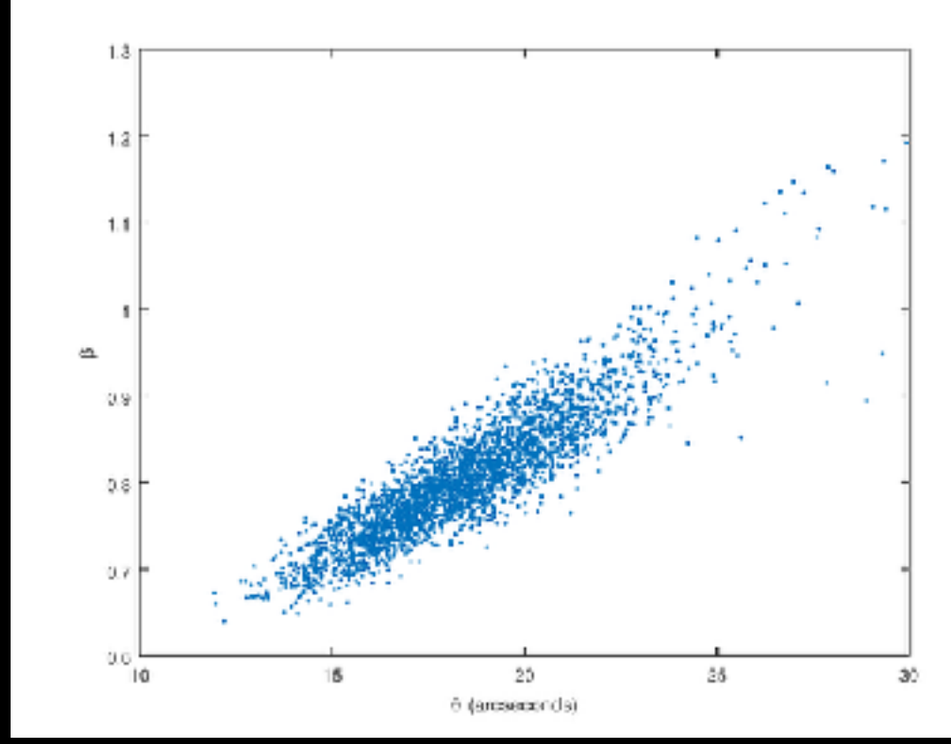
- Initial part is called "burn-in", and should be removed from chain.

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 stop.

- One standard technique is to run many chains, then look at scatter between them vs. expected scatter.

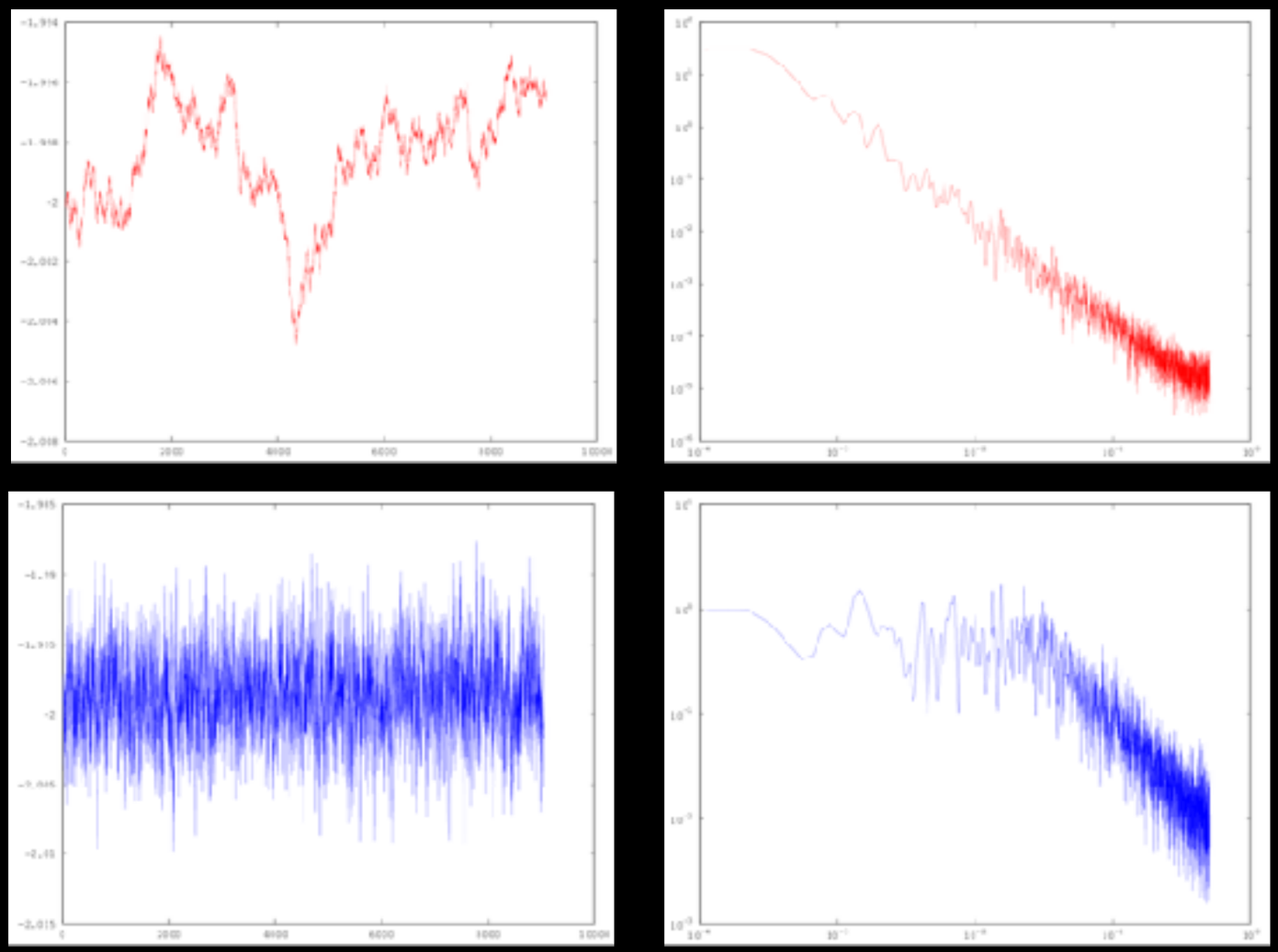
- Chains *work* independent of step size. However, they work *faster* with a good trial step size. Too large steps, we spend all our time sampling crazy land. Too small and we only move around slowly, so takes many samples to get to a new place.

- If parameters are correlated, you probably want steps to know about that.

- 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.

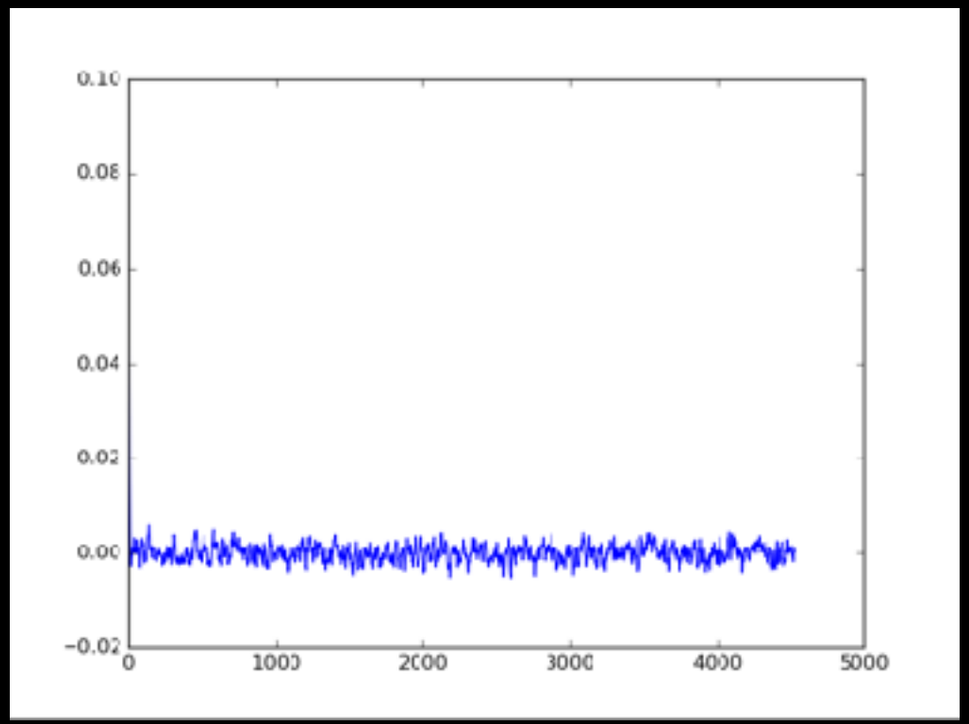
Single-Chain Convergence

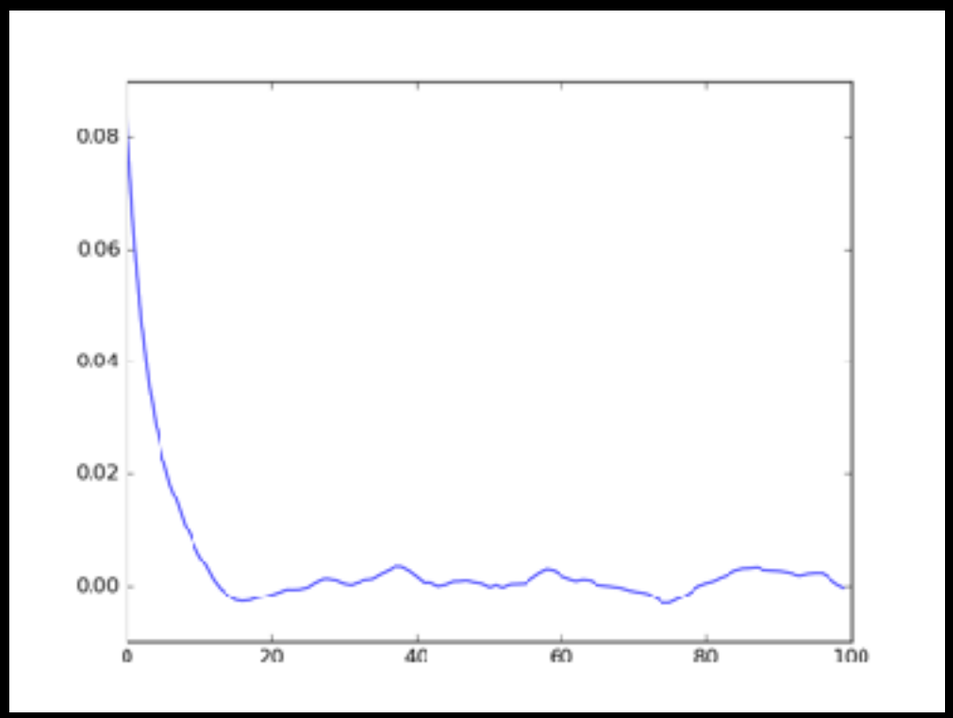
- Chains eventually forget their past.
- If you plot chain samples, then eventually they should look like white noise
- FT of converged chain should be flat for large scales (low k)
- top: unconverged chain.
  bottom:converged chain.



Convergence Via Correlation Function

- Can look at length over which correlation goes to zero.
- Gives an estimate of # of independent chain samples.
- Error on mean is σ/sqrt(n_{eff})
- What is error on the errors?
- How well have we determined 2-σ errors?





```
Jonathans-MacBook-Pro-3:lecture_14 sievers$ python sigma_limits.py
probs are  [9, 63, 336, 390] with  400  independent samples
mean/std of -2 sigma is  -1.9758189964 0.134298859411
mean/std of -1 sigma is  -0.998708960776 0.0756213449049
mean/std of +1 sigma is  1.0007693527 0.0770698963299
mean/std of +2 sigma is  1.98323454583 0.136960169072
Jonathans-MacBook-Pro-3:lecture_14 sievers$ python sigma_limits.py
probs are  [9, 63, 336, 390] with  400  independent samples
mean/std of -2 sigma is  -1.9793256697 0.133239056699
mean/std of -1 sigma is  -1.00218022111 0.0750195733187
mean/std of +1 sigma is  0.997494456703 0.0738194974937
mean/std of +2 sigma is  1.97591077979 0.131859607605
Jonathans-MacBook-Pro-3:lecture_14 sievers$ python sigma_limits.py
probs are  [9, 63, 336, 390] with  400  independent samples
mean/std of -2 sigma is  -1.97932914448 0.137403438897
mean/std of -1 sigma is  -0.993297829502 0.0764195279129
mean/std of +1 sigma is  1.00148930758 0.0726167903993
mean/std of +2 sigma is  1.97754400032 0.1359435569
```

```
import numpy as np
from scipy.special import erfc

npoints=400
niter=1000

plus1=np.zeros(niter)
minus1=np.zeros(niter)
plus2=np.zeros(niter)
minus2=np.zeros(niter)

m1_ind=1-0.5*erfc(-1/np.sqrt(2))
p1_ind=1-0.5*erfc(1/np.sqrt(2))

m2_ind=1-0.5*erfc(-2/np.sqrt(2))
p2_ind=1-0.5*erfc(2/np.sqrt(2))


m1_ind=np.int(m1_ind*npoints)
m2_ind=np.int(m2_ind*npoints)
p1_ind=np.int(p1_ind*npoints)
p2_ind=np.int(p2_ind*npoints)

print 'probs are ',[m2_ind,m1_ind,p1_ind,p2_ind], 'with ',npoints,' independent samples'

for ii in range(niter):
    dat=np.random.randn(npoints)
    dat.sort()
    plus1[ii]=dat[p1_ind]
    plus2[ii]=dat[p2_ind]

    minus1[ii]=dat[m1_ind]
    minus2[ii]=dat[m2_ind]

print 'mean/std of -2 sigma is ',np.mean(minus2),np.std(minus2)
print 'mean/std of -1 sigma is ',np.mean(minus1),np.std(minus1)
print 'mean/std of +1 sigma is ',np.mean(plus1),np.std(plus1)
print 'mean/std of +2 sigma is ',np.mean(plus2),np.std(plus2)
```

CMB Chains

```
def update_model(params,cosmology):
    np=len(param_list)
    assert(np==len(cosmology))
    p2=params.copy()
    np_normal=np-np_power
    p2.set_cosmology(ombh2=cosmology[0],omch2=cosmology[1],H0=cosmology[2],tau=cosmology[3])
    p2.InitPower.set_params(As=cosmology[4],ns=cosmology[5])
    return p2

def wmap_chisq(cosmology,wmap,pars_in):
    t0=time.time()
    try:
        pars=update_model(pars_in,cosmology)
        results=camb.get_results(pars)
        power=results.get_cmb_power_spectra(pars,CMB_unit='muK')['total']
    except:
        return bad_chisq
    t1=time.time()
    inds=np.asarray(wmap[:,0],dtype='int')
    pred=power[inds,0]
    chisq=np.sum( (pred-wmap[:,1])**2/wmap[:,2]**2)
    t2=time.time()
    #print t1-t0,t2-t1
    return chisq
```

- CAMB (Antony Lewis maintainer) will calculate predicted CMB power spectra from models.

- Python wrapper exists(!)

- Most experiments publish a likelihood code as part of release. WMAP9 simple enough can get OK results just from raw power spectrum.

CMB Chains ctd.

```
parscamb_CAMBparams()
Click to close this tab; Option-click to close all tabs except this one
                                                    98]
                                                    , 4.57e-02,   2.8138e-09,   9.763e-01]
mycosmo=np.asarray(mycosmo)

par2=update_model(pars,mycosmo)
t1=time.time()
results=camb.get_results(par2)
t2=time.time()

errs=np.asarray([2.4e-4, 7.31e-4,0.7,1.8e-3,7e-12,0.005])/2
wmap=np.loadtxt('wmap_tt_spectrum_9yr_v5.txt')
chisq=wmap_chisq(mycosmo,wmap,pars)

ombh=np.linspace(0.022,0.024,50)
cosmo_use=mycosmo.copy()
```

```
nsamp=1000
chains=np.zeros([nsamp,1+len(mycosmo)])
for iter in range(nsamp):
    new_cosmo=mycosmo+np.random.randn(len(errs))*errs
    new_chisq=wmap_chisq(new_cosmo,wmap,pars)
    accept=False
    if new_chisq<bad_chisq:
        thresh=np.exp(-0.5*(new_chisq-chisq))
        if np.random.rand()<thresh:
            accept=True
    print iter, ' new_chisq is',new_chisq, ' and accept ',accept,new_cosmo
    if accept:
        chisq=new_chisq
        mycosmo=new_cosmo
    chains[iter,0]=chisq
    chains[iter,1:]=mycosmo
```

- WMAP code I've posted does an OK job reproducing parameters.
- It uses non-correlated errors. How could we improve that?

CMB Chains ctd.

- WMAP code I've posted does an OK job reproducing parameters.

- It uses non-correlated errors. How could we improve that?

  - Standard trick is to run shorter chain to estimate covariance, then sample from that.

```
pars=camb.CAMBparams()
Click to close this tab; Option-click to close all tabs except this one                    98]
mycosmo=np.asarray(mycosmo)                                  4.57e-02,   2.0138e-09,   9.763e-01]

par2=update_model(pars,mycosmo)
t1=time.time()
results=camb.get_results(par2)
t2=time.time()

errs=np.asarray([2.4e-4, 7.31e-4,0.7,1.8e-3,7e-12,0.005])/2
wmap=np.loadtxt('wmap_tt_spectrum_9yr_v5.txt')
chisq=wmap_chisq(mycosmo,wmap,pars)

ombh=np.linspace(0.022,0.024,50)
cosmo_use=mycosmo.copy()
```

```
nsamp=1000
chains=np.zeros([nsamp,1+len(mycosmo)])
for iter in range(nsamp):
    new_cosmo=mycosmo+np.random.randn(len(errs))*errs
    new_chisq=wmap_chisq(new_cosmo,wmap,pars)
    accept=False
    if new_chisq<bad_chisq:
        thresh=np.exp(-0.5*(new_chisq-chisq))
        if np.random.rand()<thresh:
            accept=True
    print iter, ' new_chisq is',new_chisq, ' and accept ',accept,new_cosmo
    if accept:
        chisq=new_chisq
        mycosmo=new_cosmo
    chains[iter,0]=chisq
    chains[iter,1:]=mycosmo
```

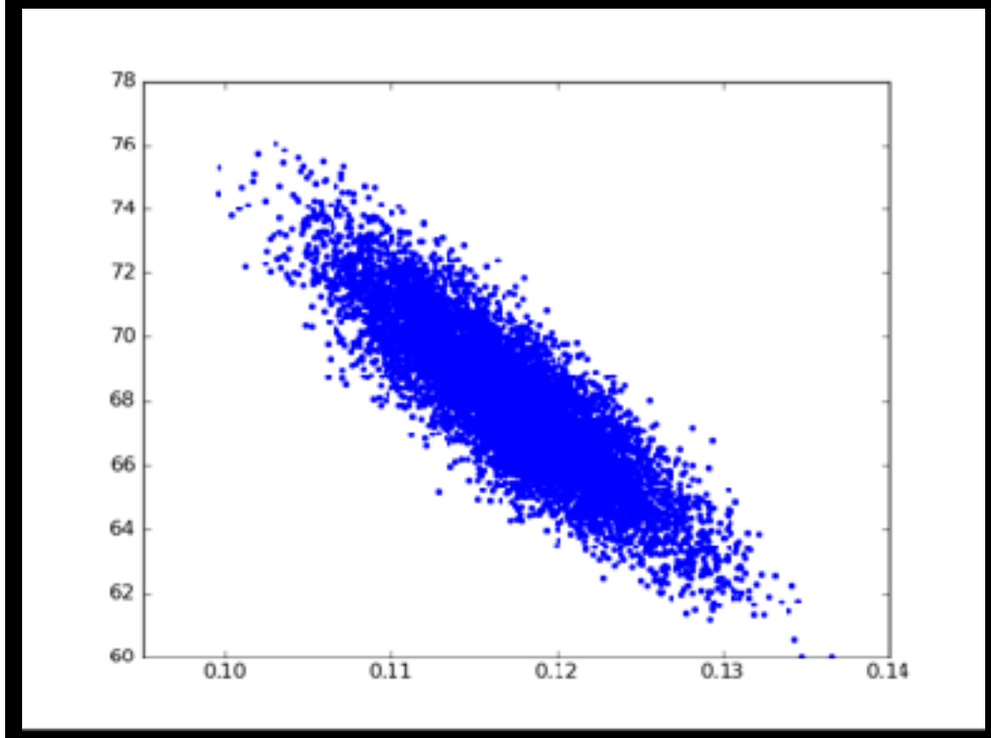
Output

- I’ve posted chain output of a simple version of WMAP9 (plus polarization).

- Means, standard deviations etc. can be calculated from the chains, as can covariances.

- Script also checks for highly correlated parameters

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

chains=np.loadtxt('wmap9_pol_corr_chains_v4_hippo.txt')
for i in range(1,chains.shape[1]):
    dat=chains[:,i].copy()
    myval=np.mean(dat)
    mystd=np.std(dat)
    dat=dat-myval
    datft=np.fft.rfft(dat)
    mycorr=np.fft.irfft(datft*np.conj(datft))
    mylen=np.min(np.where(mycorr<0))
    nsamp=len(dat)/mylen
    print 'mean/error on parameter ',i-1,' are ',myval,mystd,' with roughly ',nsamp,'

chains_norm=chains[:,1:].copy()
for i in range(chains_norm.shape[1]):
    chains_norm[:,i]=chains_norm[:,i]-chains_norm[:,i].mean()
    chains_norm[:,i]=chains_norm[:,i]/chains_norm[:,i].std()
mycorr=np.dot(chains_norm.transpose(),chains_norm)/chains_norm.shape[0]
print 'correlation matrix is: '
print mycorr

plt.ion()
plot_thresh=0.5
npar=mycorr.shape[0]
print 'npar is',npar
for i in range(npar):
    for j in range(i+1,npar):
        if np.abs(mycorr[i,j])>plot_thresh:
            plt.clf();
            plt.plot(chains[:,i+1],chains[:,j+1],'.')
            outname='wmap_corrs_'+repr(i)+'_'+repr(j)+'.png'
            plt.savefig(outname)
```

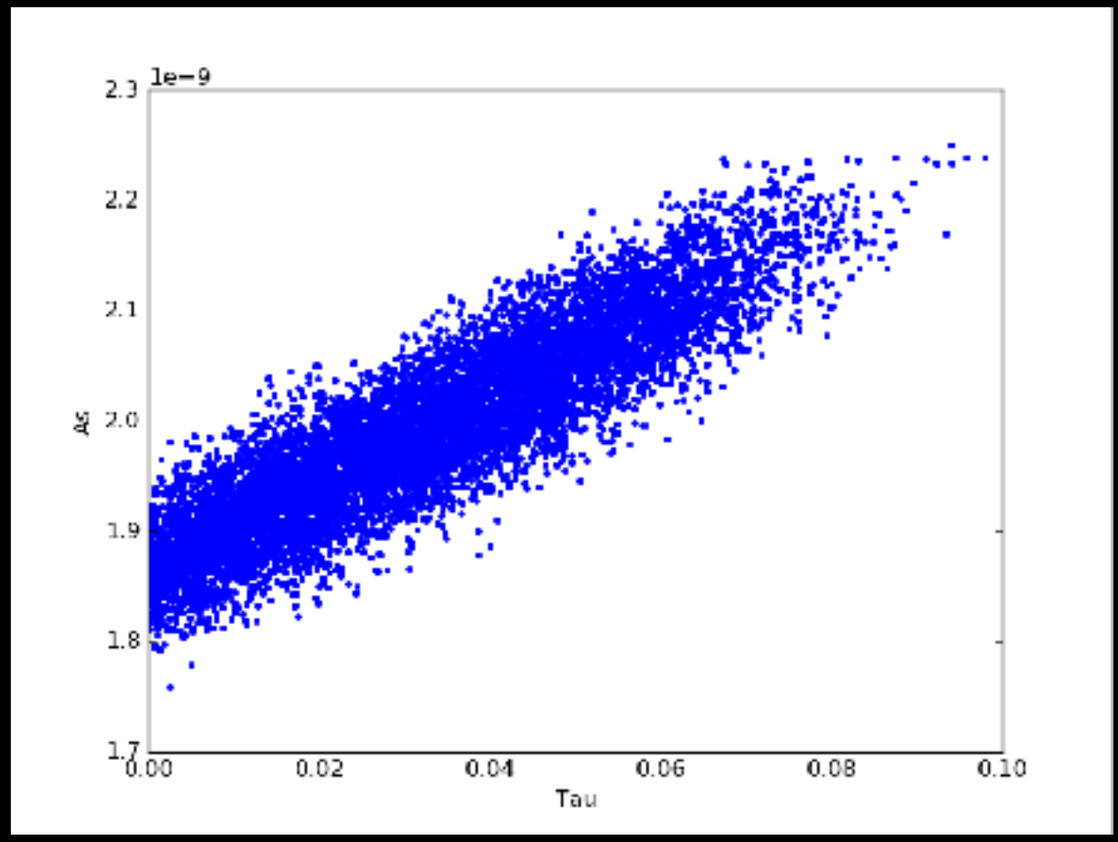



```
mean/error on parameter  0  are  0.0222860762053 0.000538203996813  with roughly  410  indepenent samples
mean/error on parameter  1  are  0.117182927182 0.0049507805274  with roughly  319  indepenent samples
mean/error on parameter  2  are  67.965357016 2.24277182627  with roughly  410  indepenent samples
mean/error on parameter  3  are  0.0319033090969 0.0203133500902  with roughly  555  indepenent samples
mean/error on parameter  4  are  1.99028245937e-09 8.65373625888e-11  with roughly  576  indepenent samples
mean/error on parameter  5  are  0.962840948863 0.0136196826567  with roughly  441  indepenent samples
correlation matrix is:
[[ 1.         -0.24947737  0.61816937  0.19303752  0.21011013  0.84147807]
 [-0.24947737  1.         -0.86078837 -0.14412739  0.23631121 -0.43294914]
 [ 0.61816937 -0.86078837  1.          0.19556806 -0.07516771  0.70242859]
 [ 0.19303752 -0.14412739  0.19556806  1.          0.9124564   0.22797675]
 [ 0.21011013  0.23631121 -0.07516771  0.9124564   1.          0.1865787 ]
 [ 0.84147807 -0.43294914  0.70242859  0.22797675  0.1865787   1.        ]]
```

Parameter Ranges

- Sometimes we know a parameter must be within a range. E.g optical depth to reionization must not be negative.

- If you have limits like this, always check if your chains butt up against them before reporting limits.



Of course…

- This likelihood is not right
- We are measuring *variance* of data. This is not χ^2.
- Real thing MCMC is working on is relative likelihood.
  Accept with probability like(new_state)/like(old_state).
- What is likelihood from power spectrum?

Back to Gaussian

- Gaussian PDF is $\exp(-(x-\mu)^2/2\sigma^2)/\sqrt{(2\pi\sigma^2)}$

- For χ^2 we could assume that σ is constant.

- This is not the case when we are estimating variances. The denominator matters.

Likelihood from Many Variables

- As usual, for independent gaussians, likelihood is just the product of individual likelihoods
- Also as usual, log likelihood is usually more convenient.
  For CMB, we can also set expectation μ=0.
- $\log(L)=-1/2\ \sum x^2/\sigma^2-1/2\sum\log(2\pi\sigma^2)$
- we can safely ditch 2π (unless you vary over non-Euclidean geometries…)
- Leaves $\log(L)=-1/2\ \chi 2-1/2\sum\log(\sigma^2)$

To Matrix Expression

- For uncorrelated data, $N_{ii}=\sigma_i^2$., $N_{ij}=0$ if $i\neq j$.
- Determinant of a diagonal matrix is product of diagonal elements. So, $\sum\log(\sigma_i^2)=\log(\text{product}(\sigma_i^2))=\log(|N|)$
- So, our final result is $-2\log(L)=d^T N^{-1} d+\log(|N|)$
- As you might expect, once we have a proper linear algebra expression, this also works for correlated noise.

Likelihood Shape

- Once we have correlated data, we come up with a model for the noise, then calculate the likelihood of the observed data given this model.

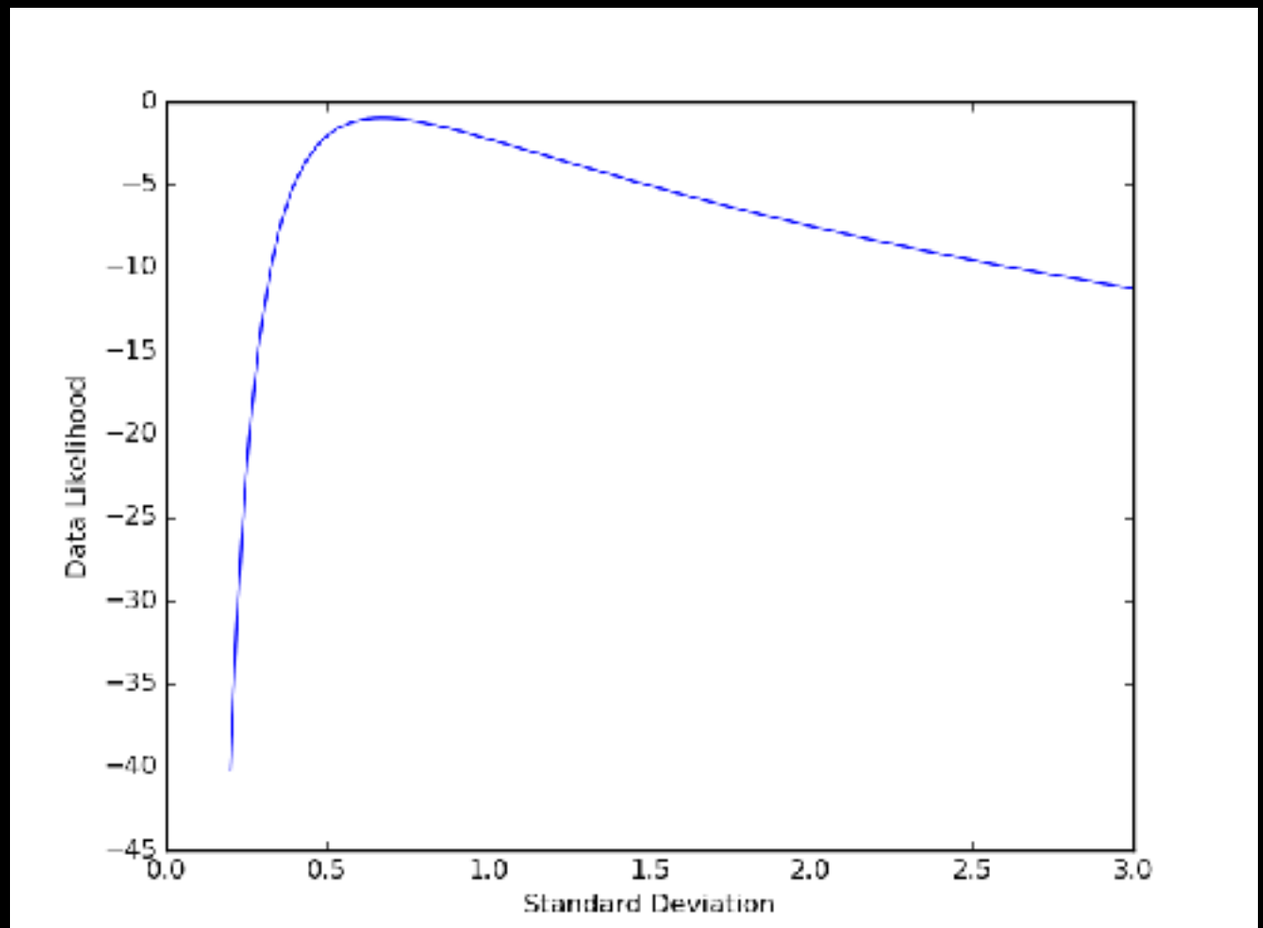
- This model is *nonlinear* - we can't write down a global solution like we could with χ^2.

- The probability we roll the dice and get very large variance is exponentially small.

- The probability we roll the dice and get very small variance is polynomial small. Shape is skewed.

Likelihood Shape

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

plt.ion()

n=10
x=np.random.randn(n)
sig=np.linspace(0.2,3,1000)
chisq=np.sum(x**2)/sig**2

logdet=n*np.log(sig**2)

loglike=-0.5*chisq-0.5*logdet
plt.clf()
plt.plot(sig,loglike)
plt.xlabel('Standard Deviation')
plt.ylabel('Data Likelihood')
plt.savefig('variance_likelihood.png')
```



Fitting a Power Spectrum

- This shape is important for low-ell, e.g. measurements of the optical depth to reionization.

- How would we estimate the power spectrum?

- $\langle d_i d_j \rangle = N_{ij,instrumental} + Cov_{ij}(sky)$

- $Cov_{ij}(sky) = \sum Y_{lm}(\theta_i \phi_j) Y_{lm}(\theta_j, \phi_j) C_l$.

- If we know the instrumental noise, we can tweak the C_l to maximize the likelihood.

- Looking at the likelihood surface gives us the correlations between the C_l. This gives an estimate of error bars.

Likelihood Codes

- Rather than the simplistic χ^2, in real life we'll have to use a full likelihood. Important for optical depth, n_s, tensor-to-scalar ratio…

- Experiments these days usually put out likelihood code that does this properly.

- WMAP has done so, to get it right you actually need maps and noise covariance matrices.

- Do this if you want to do parameters in real life!