2D-FFTs

- Since these aren't familiar…
- $F(k_x,k_y)=\sum\sum f(x,y)\exp[-2\pi i\,(k_x x/N+k_y y/M)]$, x from 0 to N-1, y from 0 to M-1.
- If patch is square (N==M), $= \sum\sum f(\mathbf{x})\exp(-2\pi i \mathbf{k}\cdot\mathbf{x}/N)$. Holds in higher dimensions as well.
- wavevector **k** has $|\mathbf{k}|=(k_x^2+k_y^2)^{1/2}$.

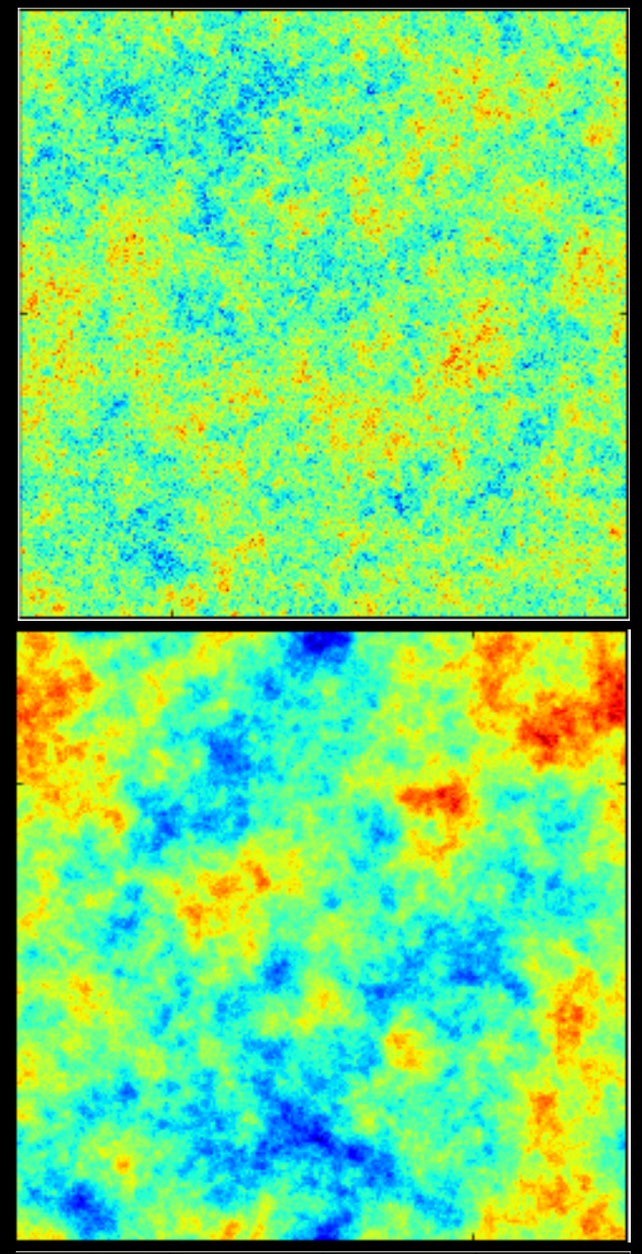
How would I Simulate 2D-Data?

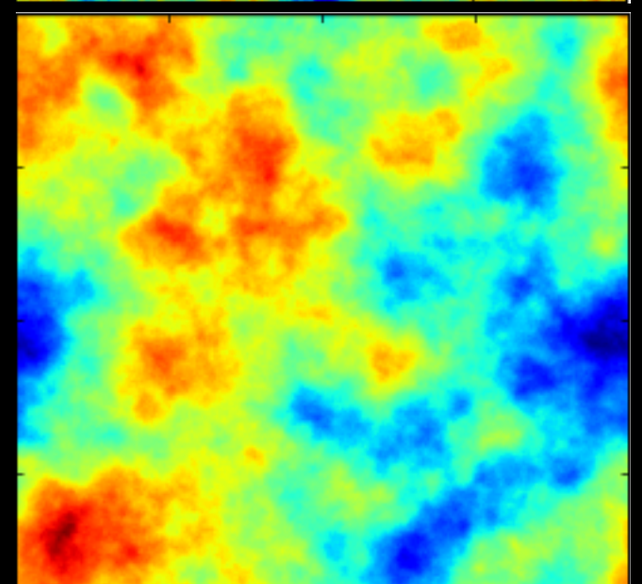
- $\langle F(\mathbf{k})^* F(\mathbf{k'})\rangle = P(|k|)\delta_{kk'}$ for some power spectrum $P(|k|)$

- So, make (suitably phased) 2D random data, and rescale by sqrt(P(|k|).

- Let’s see code…

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

plt.ion()
npix=2000
k=np.arange(npix)
k[k>npix/2]=k[k>npix/2]-npix

kx=np.repeat([k],npix,axis=0)
ky=kx.transpose()
k=np.sqrt(kx**2+ky**2)

myamp=1/(1.0+k)**1.5

#A simple way of generating Gaussian random #'s with
#appropriate phase relations to be the transform of a real
#field is to take the transform of a real field
map=np.random.randn(npix,npix)
mapft=np.fft.fft2(map)

#of course, we could also generate data with the appropriate symmetry, where
#F(kx,ky)=F(-kx,-ky)^*, with special care taking along kx=0,ky=0 axes
mapft2_r=np.random.randn(npix,npix)
mapft2_r[1:,0]=mapft2_r[1:,0]+np.flipud(napft2_r[1:,0])
mapft2_r[0,1:]=mapft2_r[0,1:]+np.flipud(napft2_r[0,1:])
mapft2_r[1:,1:]=mapft2_r[1:,1:]+np.flipud(np.fliplr(mapft2_r[1:,1:]))
mapft2_i=np.random.randn(npix,npix)
mapft2_i[1:,0]=mapft2_i[1:,0]-np.flipud(napft2_i[1:,0])
mapft2_i[0,1:]=mapft2_i[0,1:]-np.flipud(napft2_i[0,1:])
mapft2_i[1:,1:]=mapft2_i[1:,1:]-np.flipud(np.fliplr(mapft2_i[1:,1:]))
mapft2_i[0,0]=0

map_back=np.fft.ifft2(mapft*myamp)
map_back=np.real(map_back)
map_back2=np.real(np.fft.ifft2(mapft2*myamp))
```





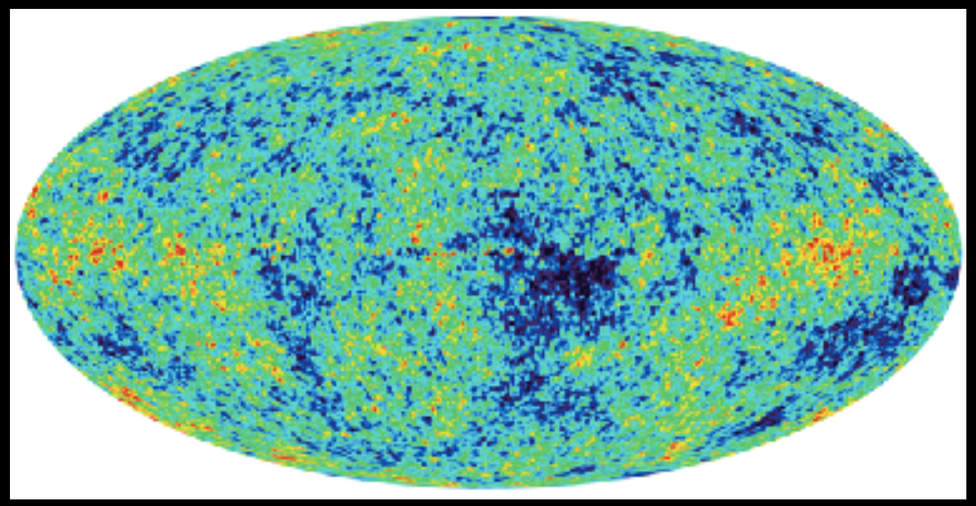
Top: FFT-based 2D simulation code
Right: outputs for $P(k) \sim k^{(-2,-3,-4)}$. Which is which?

Parameter Extraction from CMB

- Once we've mapped the CMB, and measured its power spectrum, we want to learn about the universe.
- Power spectrum depends on various parameters (Ω_k,Ω_b,Ω_m,Ω_Λ, n_s, τ…).
- Dependence is (often highly!) non-linear, so we can't use the linear least-squares we've done so far.
- How does CMB depend on parameters?

Early On…

- Perturbations initially are sound waves
- Gravity drives collapse, photon pressure provides restoring force.
- Photons coupled to matter by free electrons. They disappear at recombination.
- Wave properties get locked in when electrons go away.

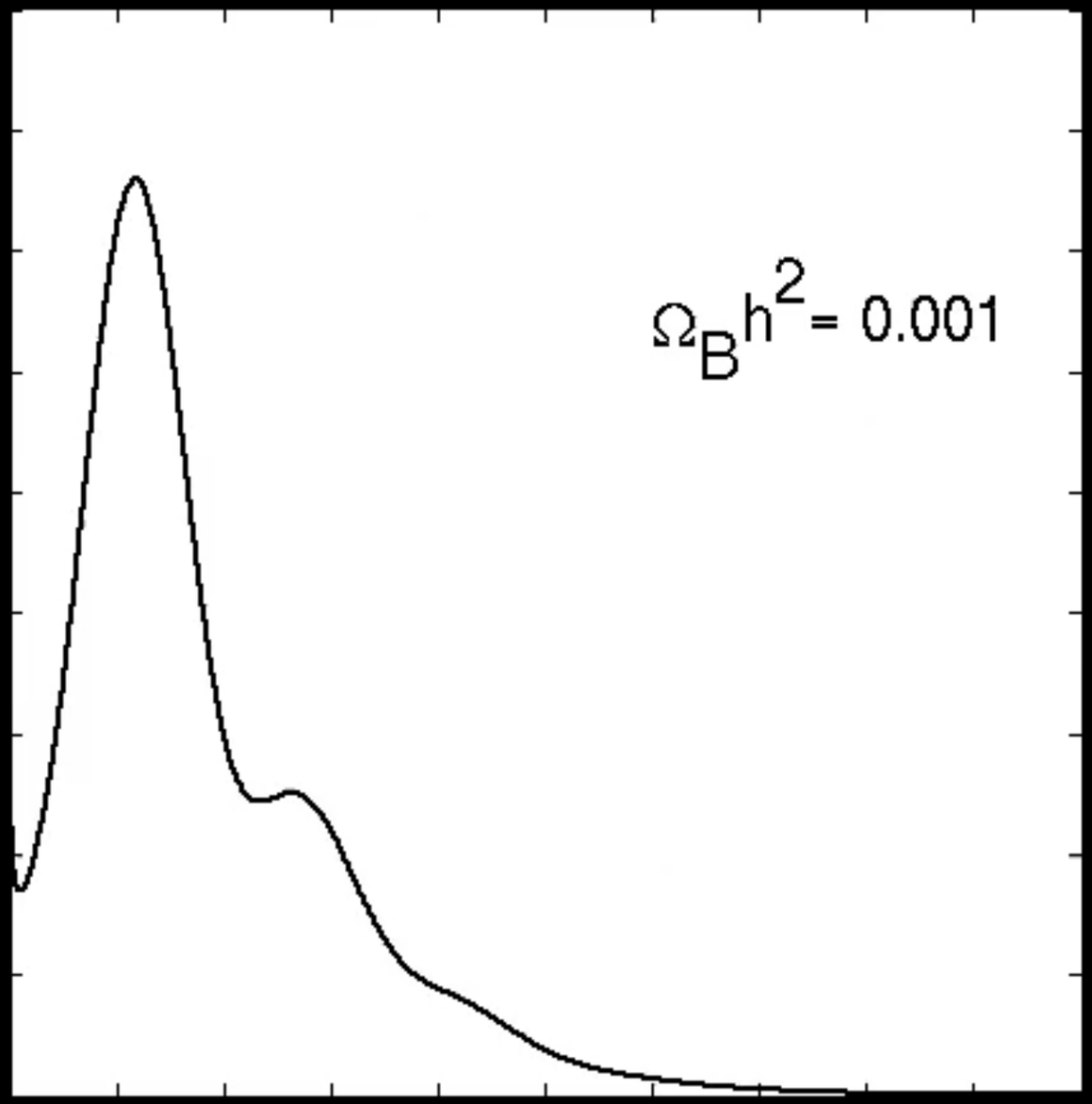


Parameter Example: Baryon Density

Matter wants to collapse. It drags light with it, but the light doesn't want to be squeezed. The more matter there is, the more it can compress the photons. So, more baryons means brighter patches.

Photons also diffuse ("Silk damping"). More electrons means less diffusion, and so power on small scales falls off less quickly.

(spectrum also falls of since we're really averaging over a finite-thickness shell)

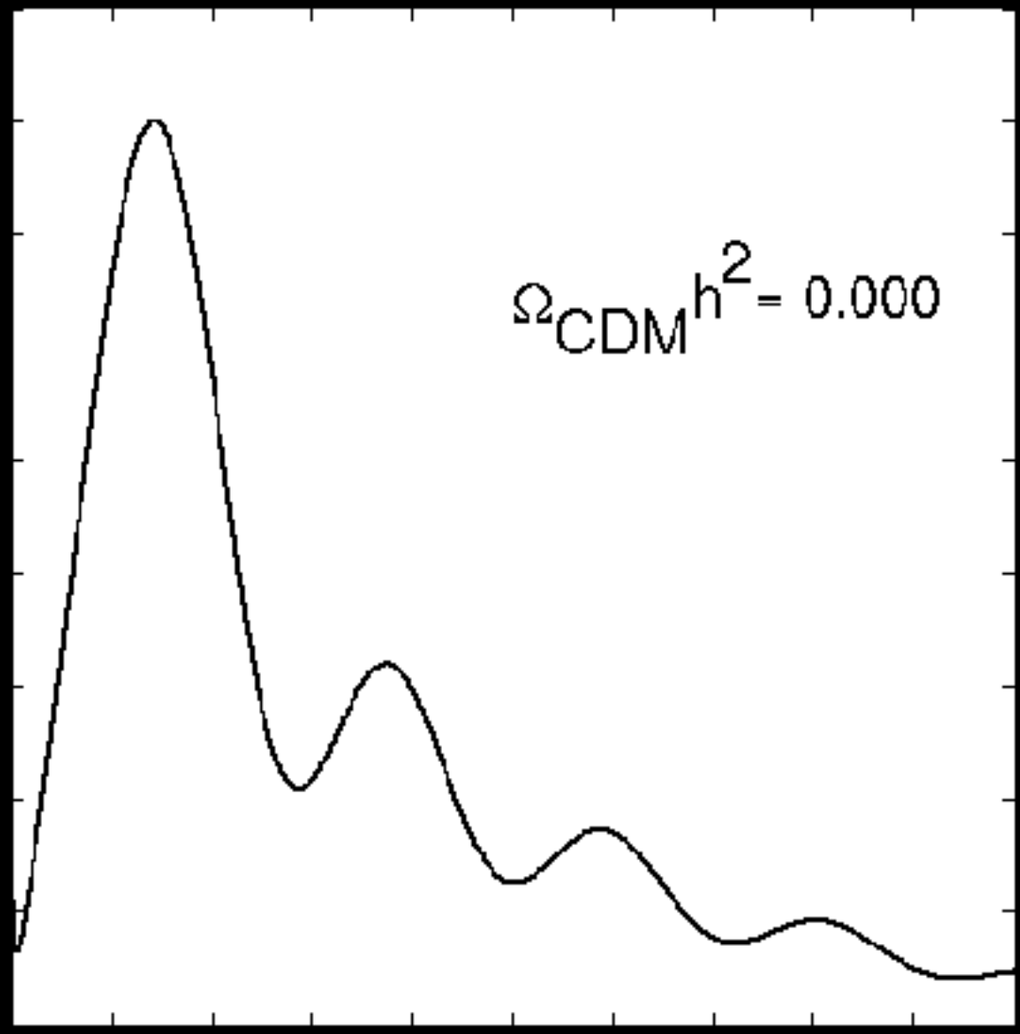


Parameter Example: Dark Matter

Dark matter doesn't interact with light. No pressure means the dark matter just collapses.

Dark matter tries to pull baryons with it through gravity, so 1st, 3rd etc. peaks DM works with baryons. 2nd, 4th etc. peaks, DM works against baryons.

Lots of DM + lots of baryons = big 1st, 3rd peak, smaller 2nd, 4th... Can basically read off baryon/DM ratio from relative even/odd peak heights.



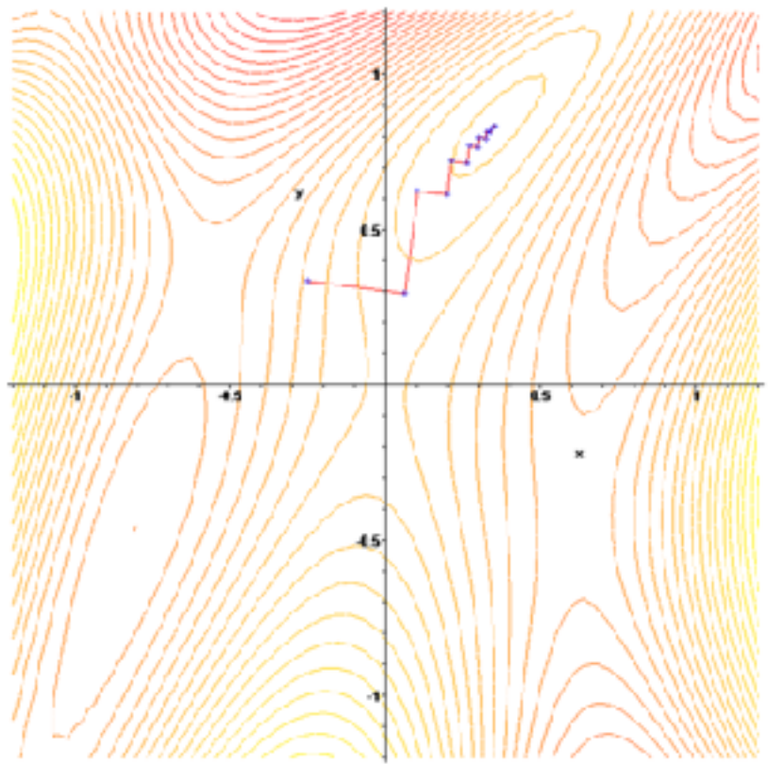
Nonlinear Fitting

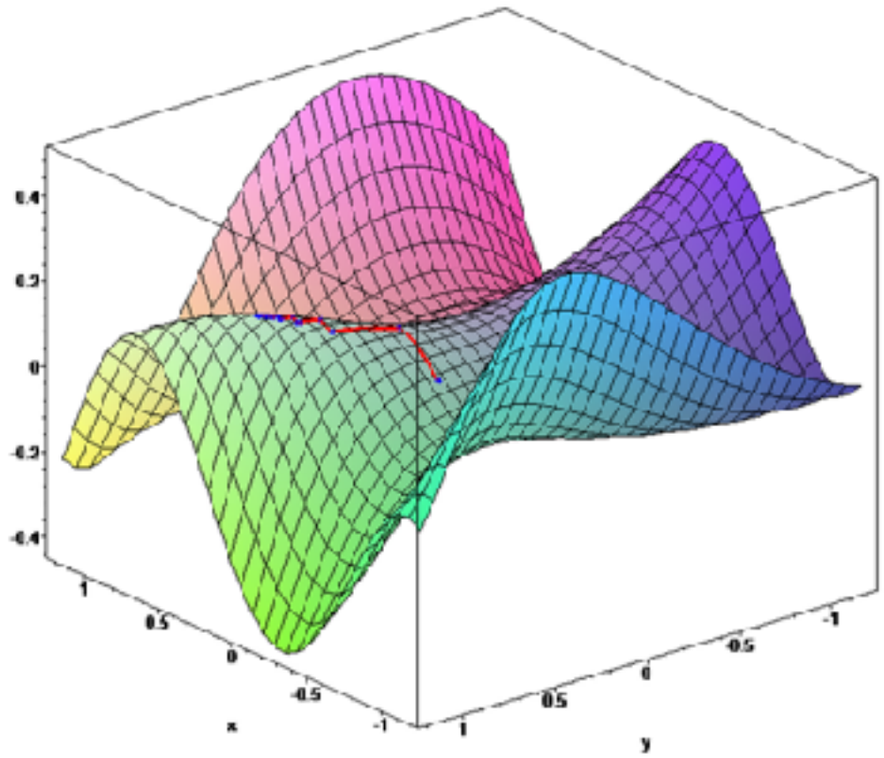
- Sometimes data depend non-linearly on model parameters
- Examples are Gaussian and Lorentzian $(a/(b+(x-c)^2)$
- Often significantly more complicated - cannot reason about global behaviour from local properties. May be multiple local minima
- Many methods reduce to how to efficiently find the "nearest"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

e gradient descent algorithm in action. (1: contour)

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: <d>=Am. Nonlinear: <d>=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! $A_m^T N^{-1} A_m\ \delta m = A_m^T N^{-1} r$

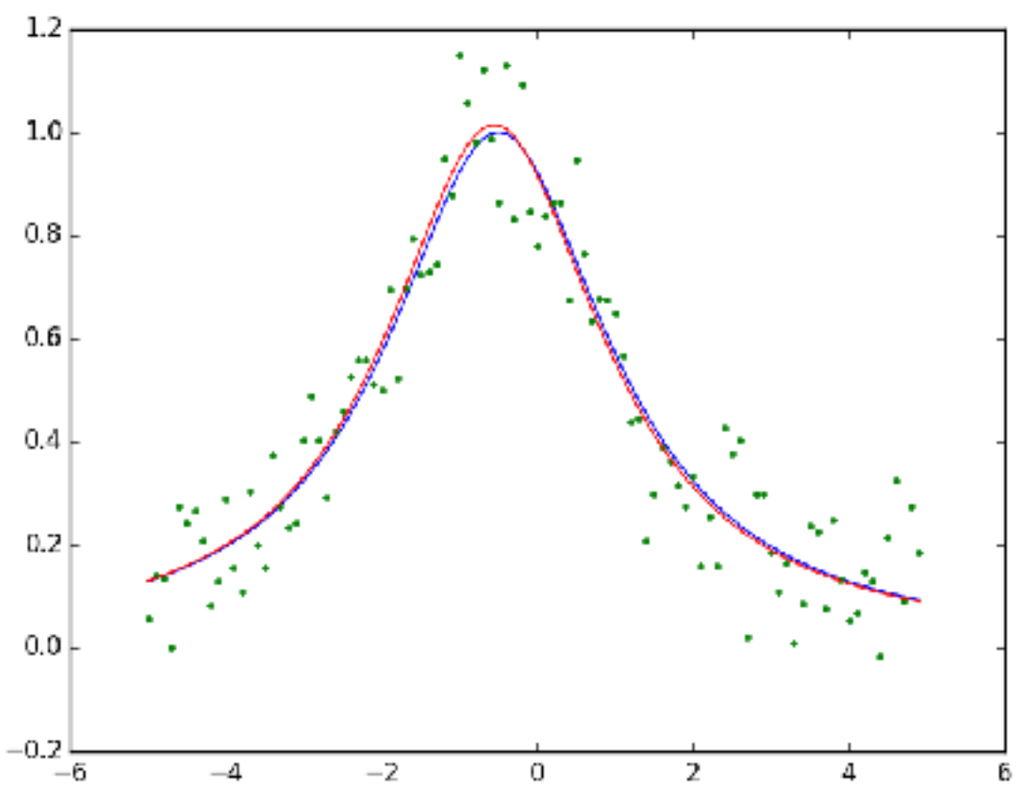
How to Implement

- 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?).
- Curvature estimate of $\chi 2$ again $(A_m^T N^{-1} A_m)$, and estimated errors on parameters again $< \delta m \, \delta m^T > = (A_m^T N^{-1} A_m)^{-1}$.

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.

Comments on Errors

- We have estimates of errors from Newton's/Levenberg-Marquardt.
- BUT. They for errors to be correct, they rely on Taylor expansion of χ^2 being valid.
- That maybe true. But may not, e.g. multiple local minima.
- Since science requires error bars, we need to check if this is correct before quoting errors.
- banana-shaped contours extremely common - curvature matrix cannot capture them. Don't know if this is the case until you've mapped out likelihood surface.

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.

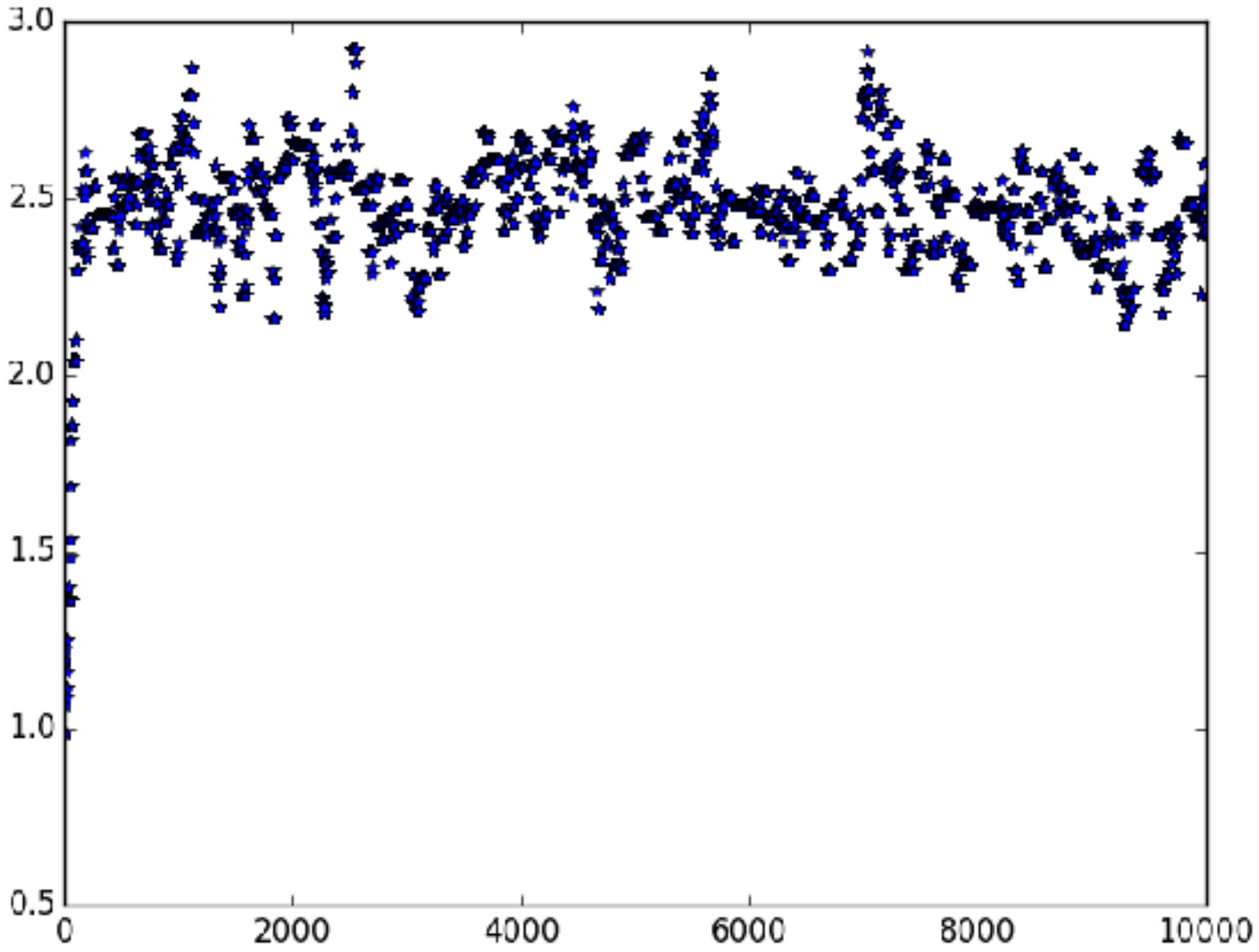
Output

```
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

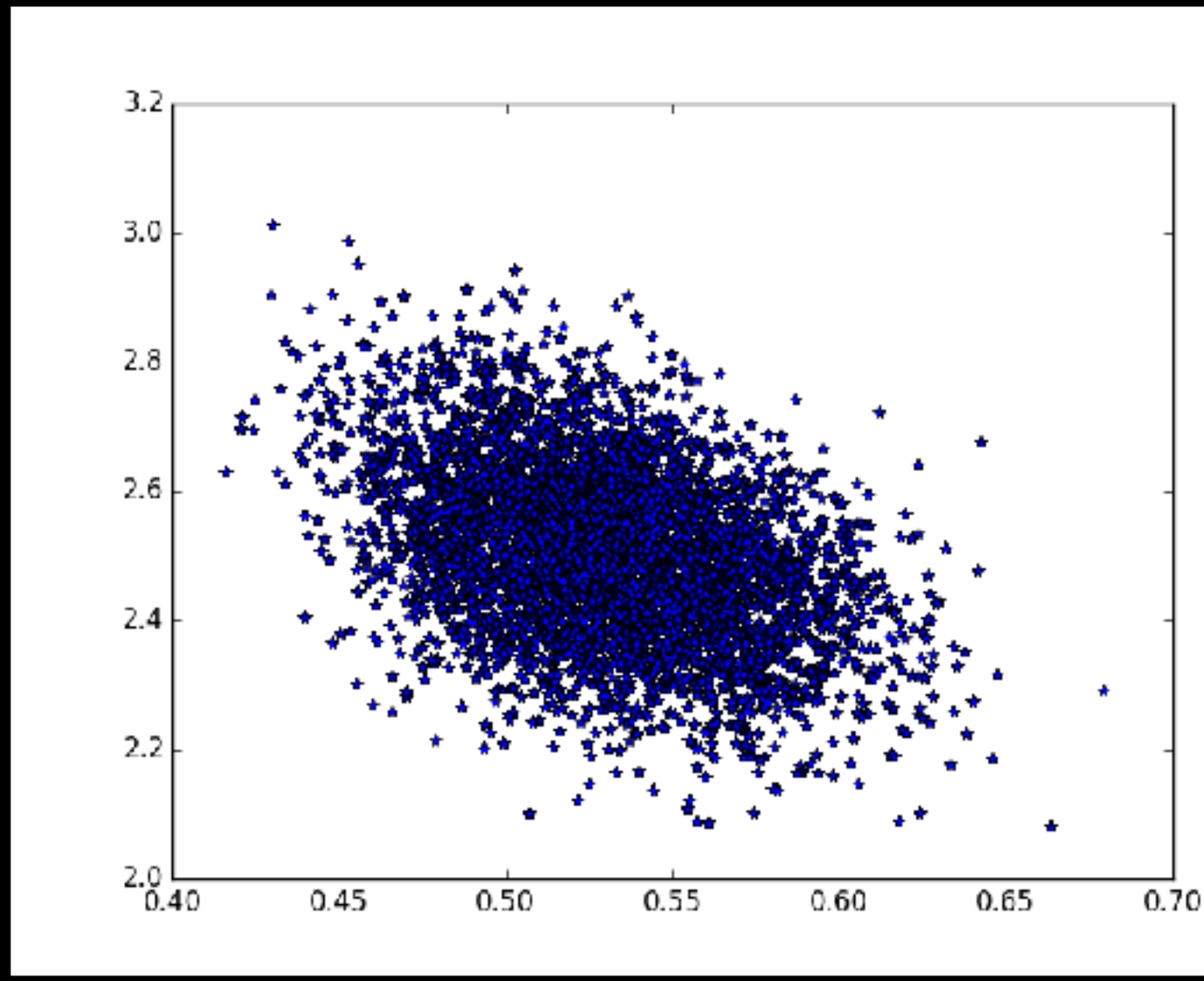


- Here's the samples for one parameter. Note big shift at beginning: we started at a wrong position, but chain quickly moved to correct value.

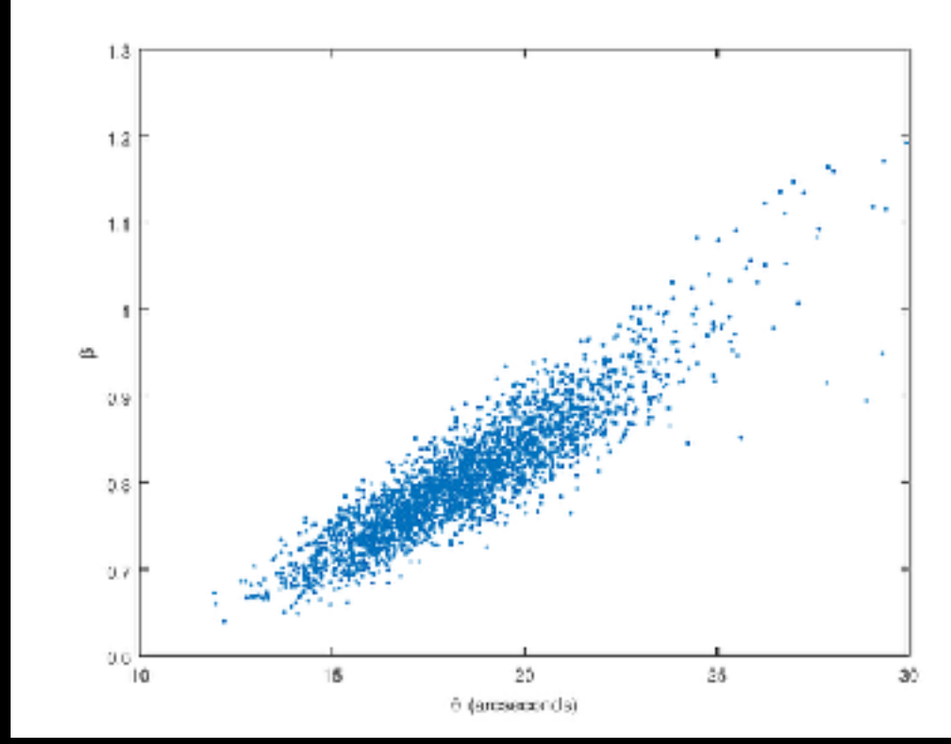
- Initial part is called "burn-in", and should be removed from chain.

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.

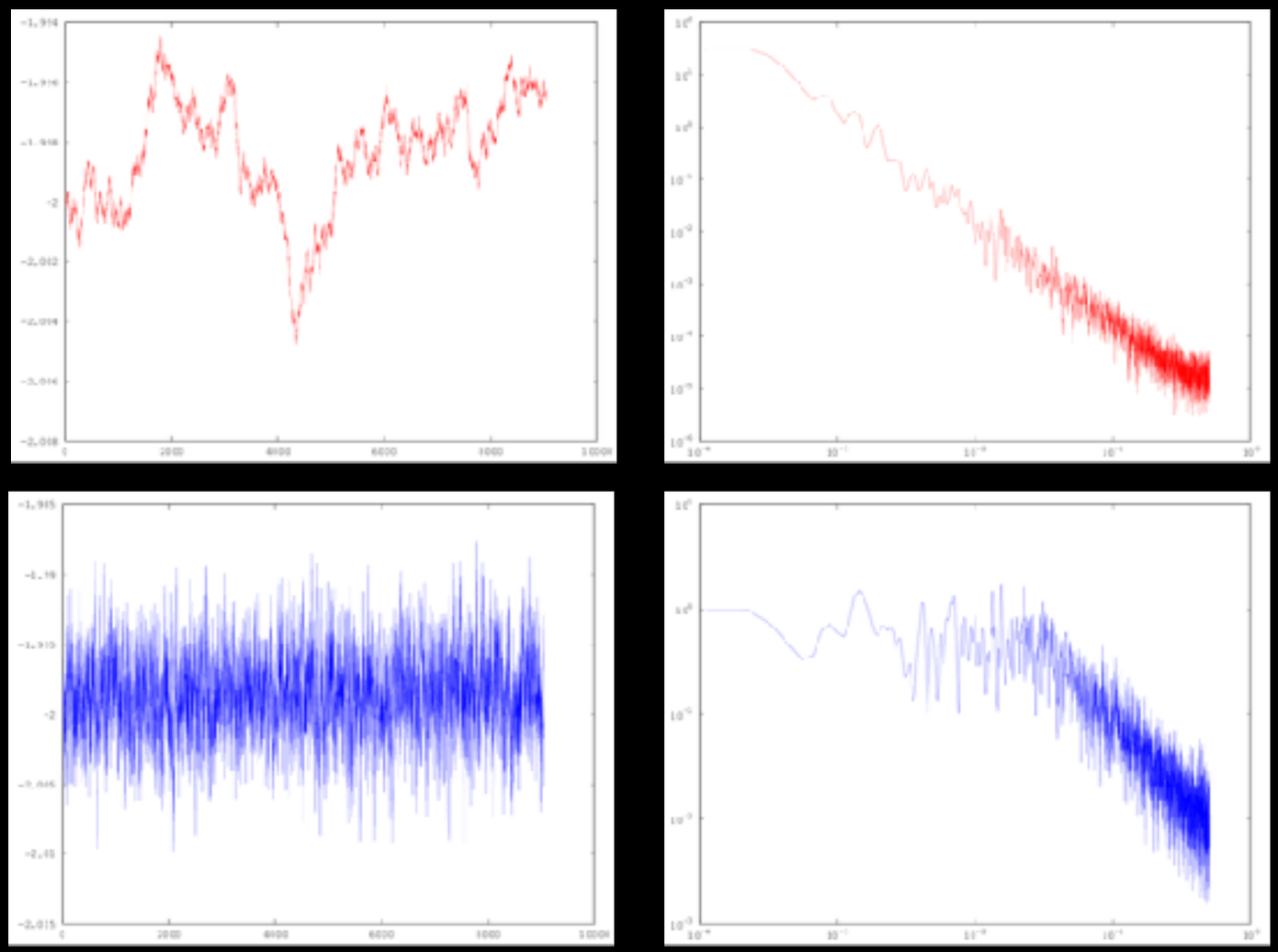
- Chains *work* independent of step size. However, they work *faster* with a good trial step size. Too large steps, we spend all our time sampling crazy land. Too small and we only move around slowly, so takes many samples to get to a new place.

- 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.

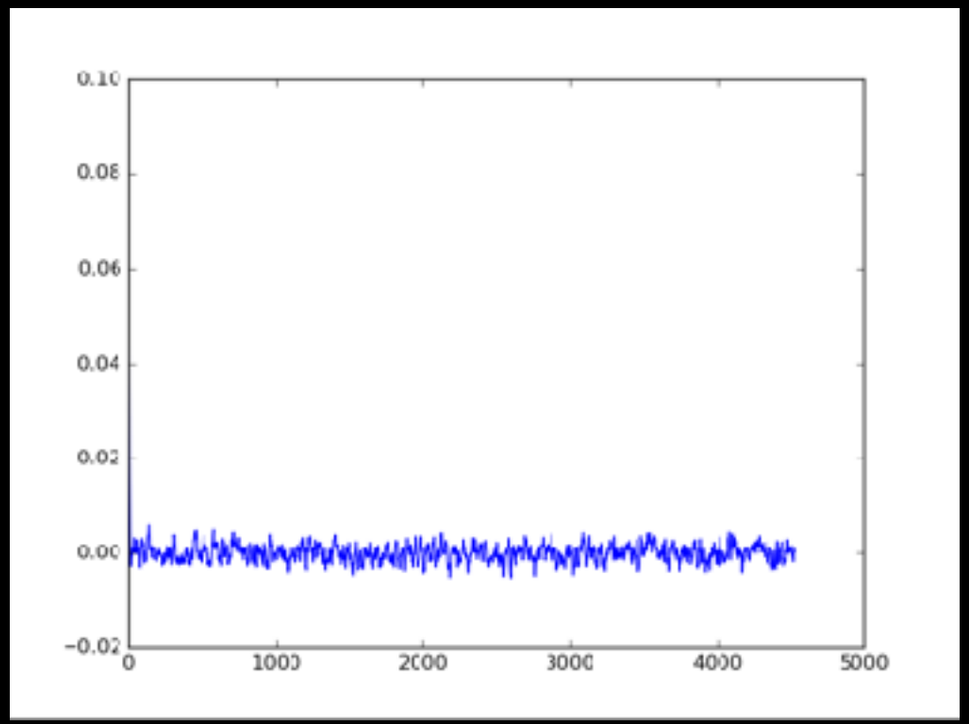
Single-Chain Convergence

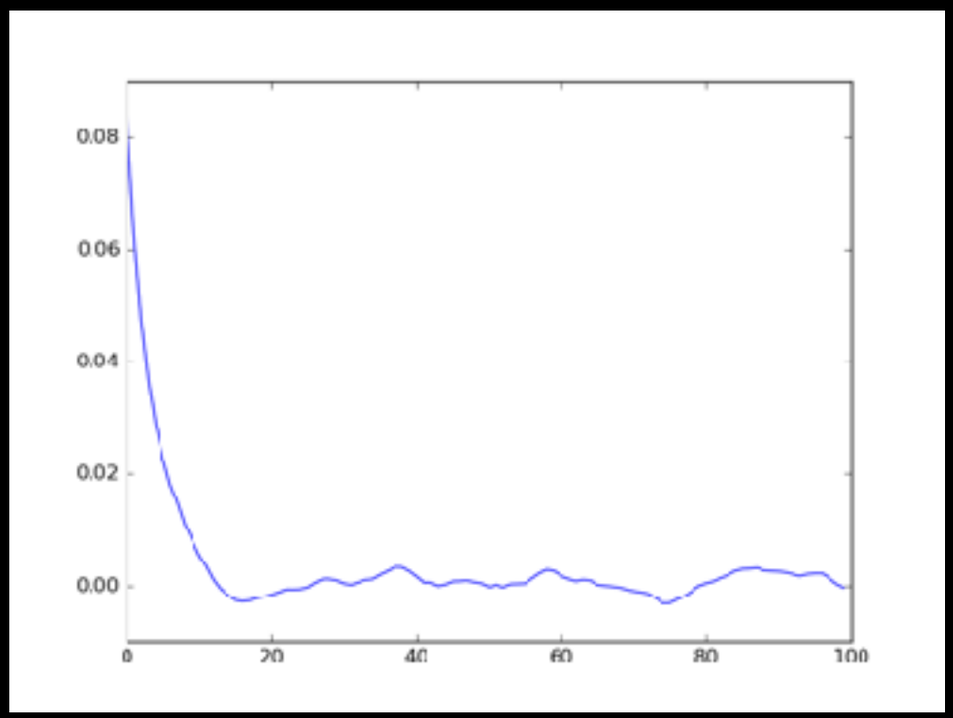
- Chains eventually forget their past.
- If you plot chain samples, then eventually they should look like white noise
- FT of converged chain should be flat for large scales (low k)
- top: unconverged chain.
  bottom:converged chain.



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?





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
```

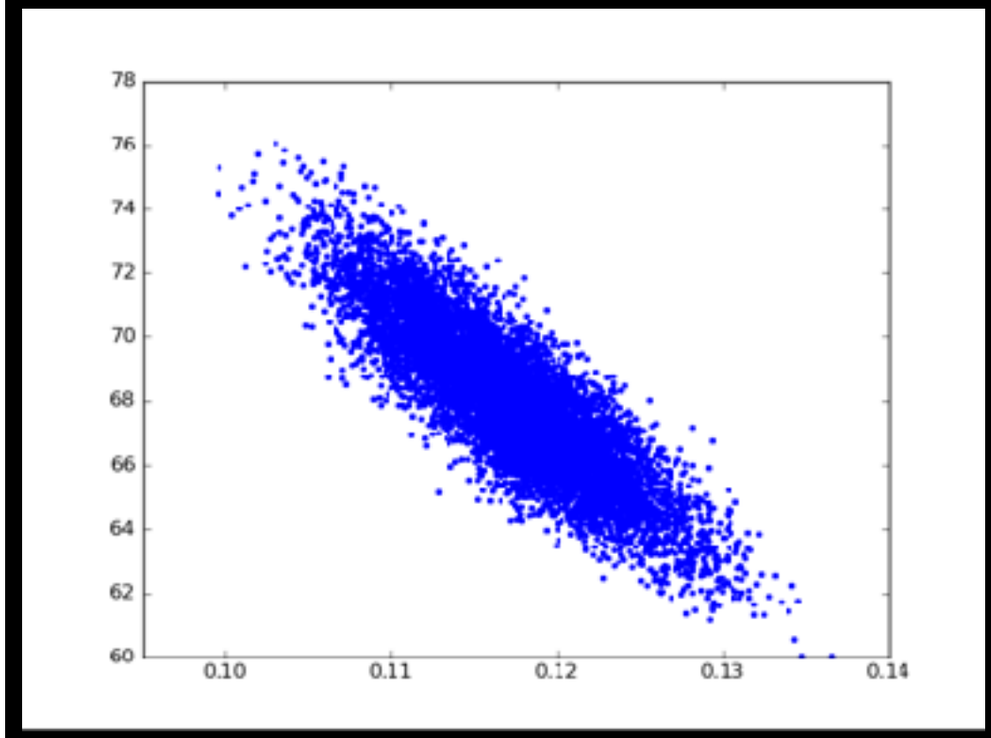
Output

- 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.        ]]
```