

CMB Parameters via MCMC

But First...

- The CMB information is encoded by the power spectrum.
- Full sky: $C_l = \langle a_{lm}^2 \rangle$, where expectation is over m .
- Small patch equivalent, we take the 2D Fourier transform, and average the variance in rings.

2D-FFTs

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

How would I Simulate 2D-Data?

- $\langle F(\mathbf{k})^* F(\mathbf{k}') \rangle = P(|\mathbf{k}|) \delta_{\mathbf{k}\mathbf{k}'} \text{ for some power spectrum } P(|\mathbf{k}|)$
- So, make (suitably phased) 2D random data, and rescale by $\sqrt{P(|\mathbf{k}|)}$.
- 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)

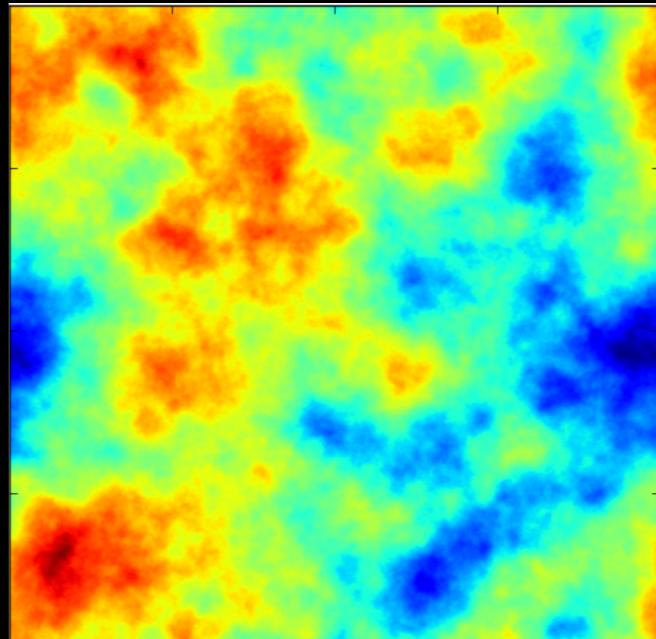
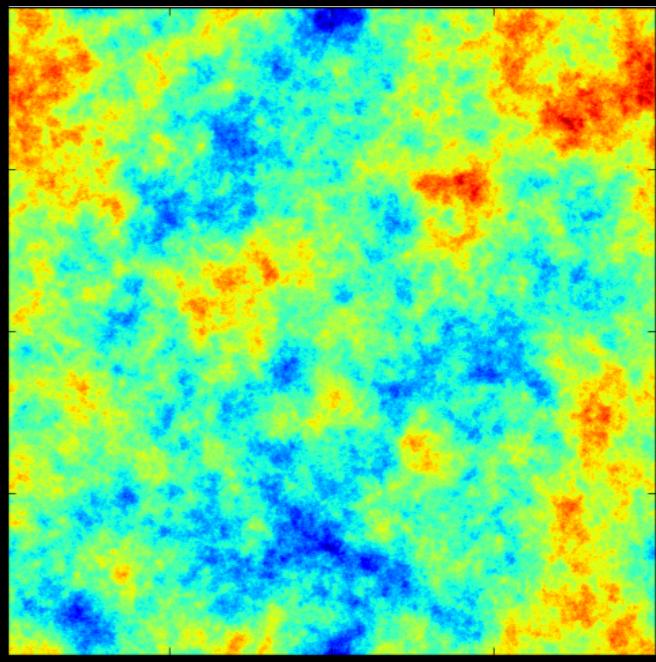
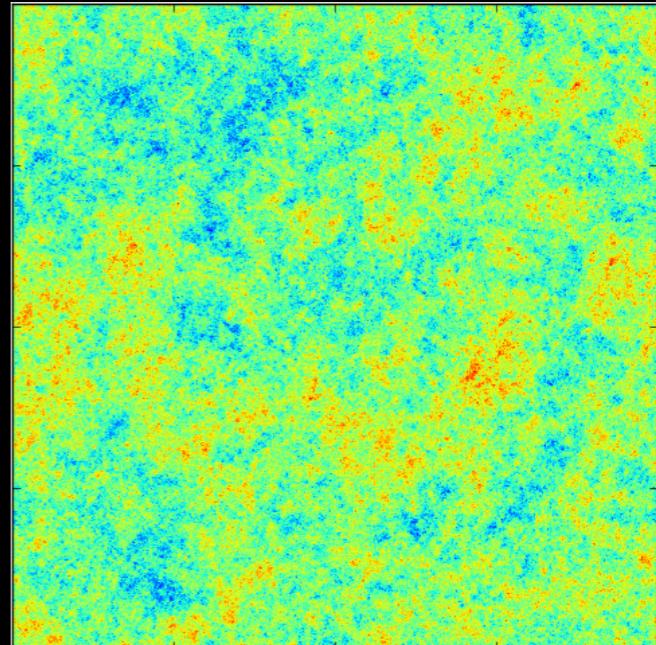
myamp=1/(1.0+k)**1.5

#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(mapft2_r[1:,0])
mapft2_r[0,1:]=mapft2_r[0,1:]+np.flipud(mapft2_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(mapft2_i[1:,0])
mapft2_i[0,1:]=mapft2_i[0,1:]-np.flipud(mapft2_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))

```



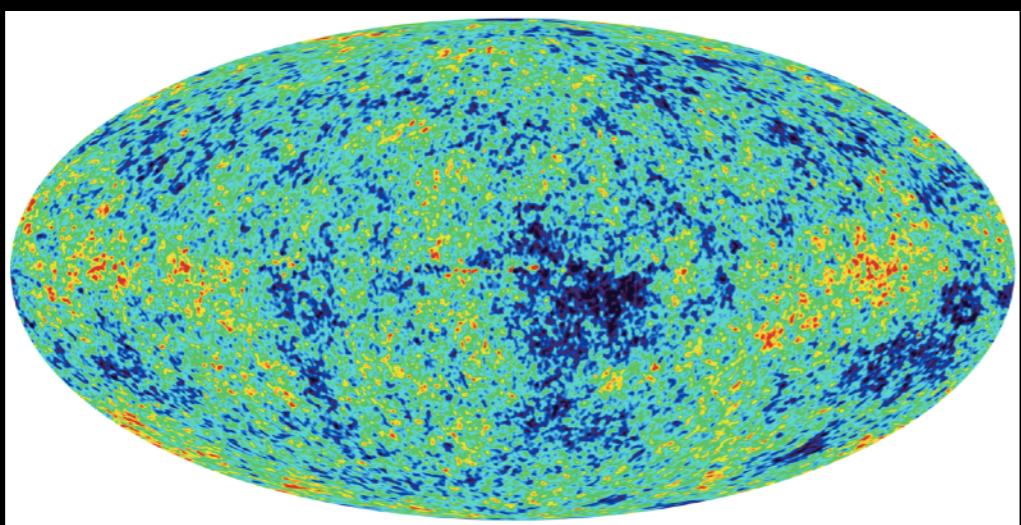
Top: FFT-based 2D simulation code
 Right: outputs for $P(k) \sim k^{(-2,-3,-4)}$. Which is which?

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 ($\Omega_k, \Omega_b, \Omega_m, \Omega_\Lambda, n_s, \tau \dots$).
- 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.

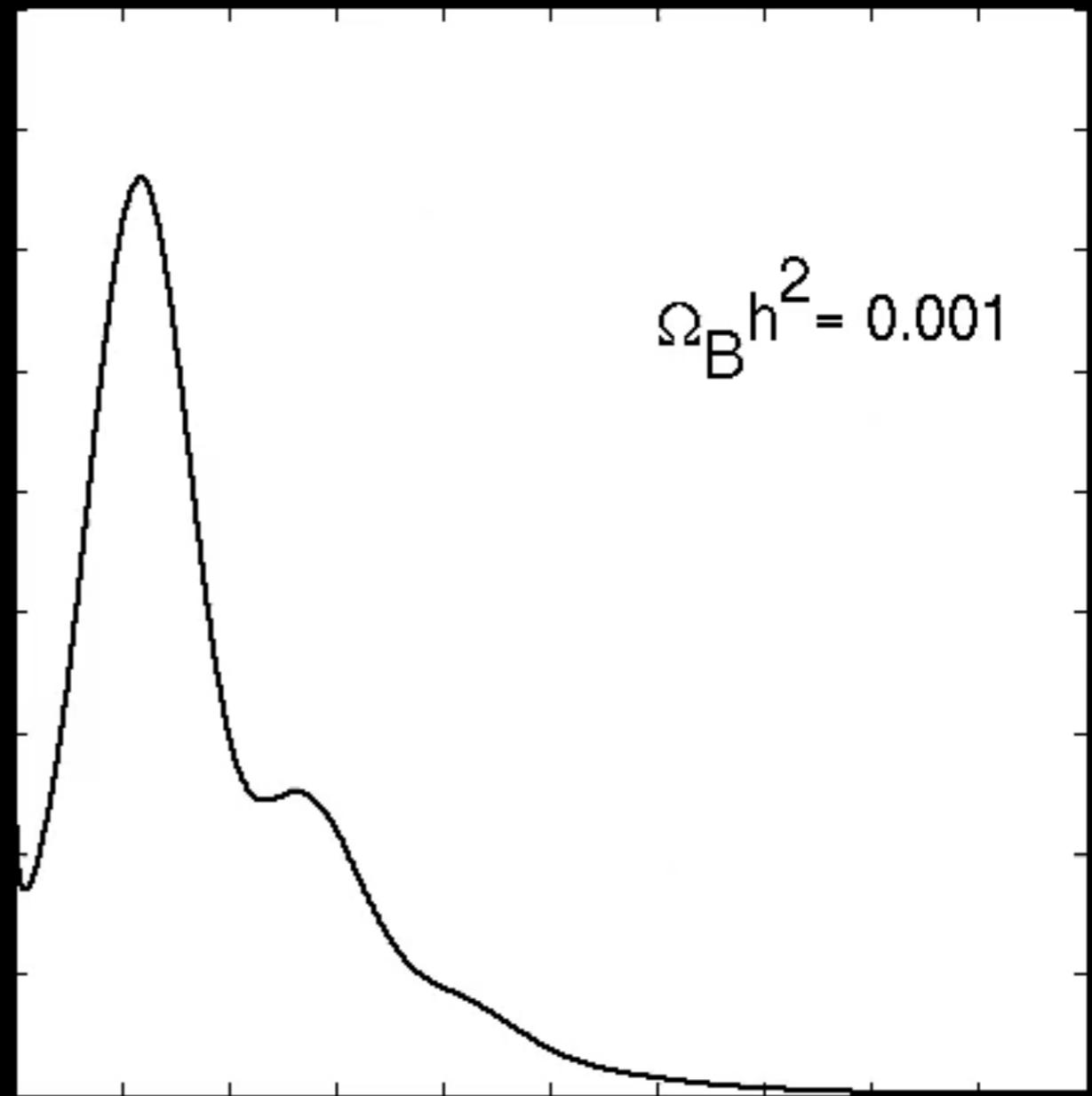


Parameter Example: Baryon Density

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 off since we're really averaging over a finite-thickness shell)

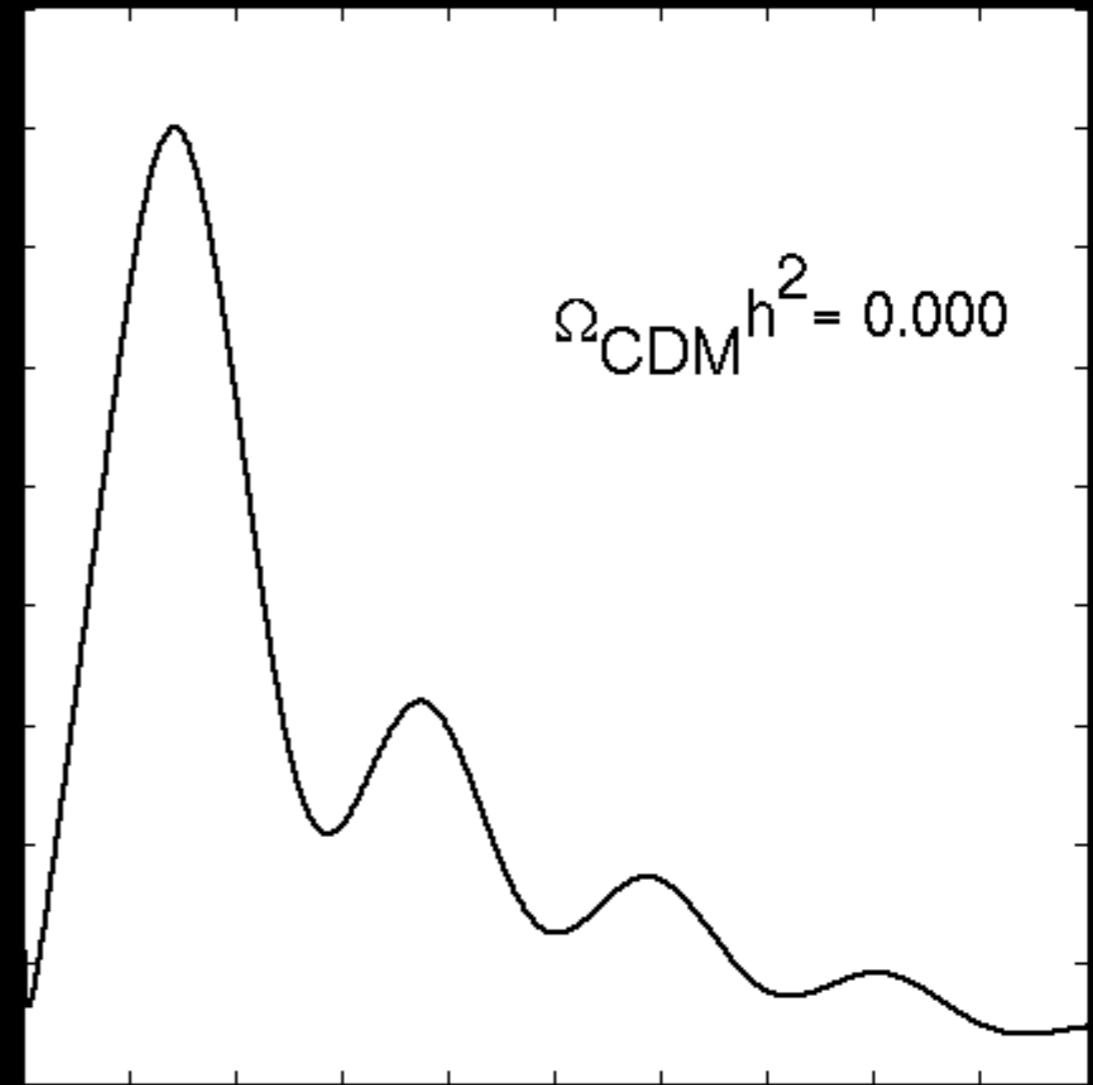


Parameter Example: Dark Matter

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.

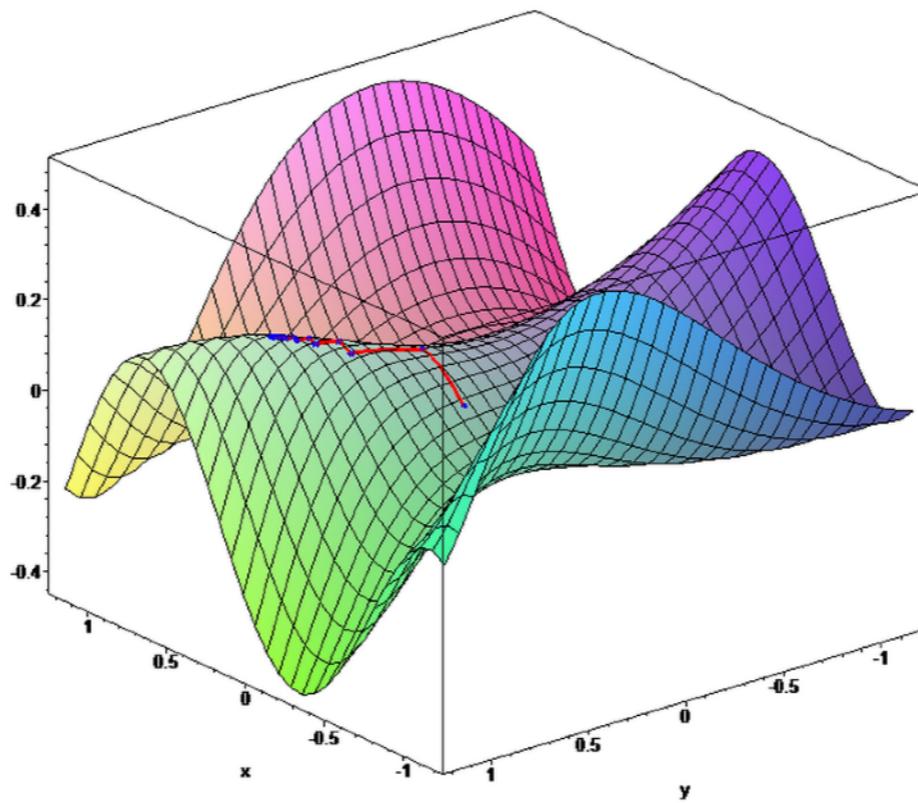
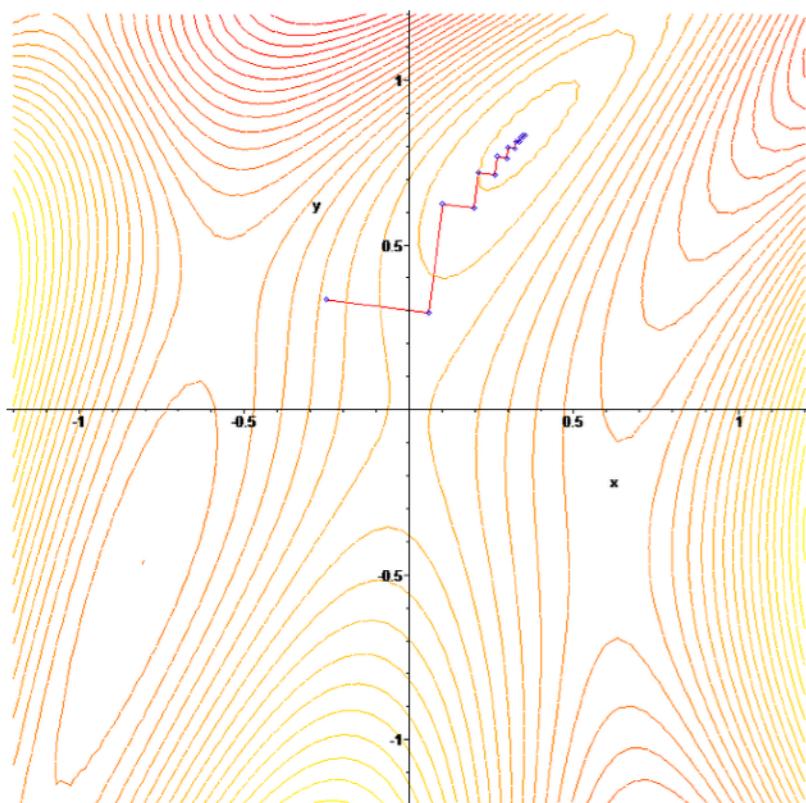


Nonlinear Fitting

- 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

The "Zig-Zagging" nature of the method is also evident below, where the gradient descent 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. Zigzagging 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

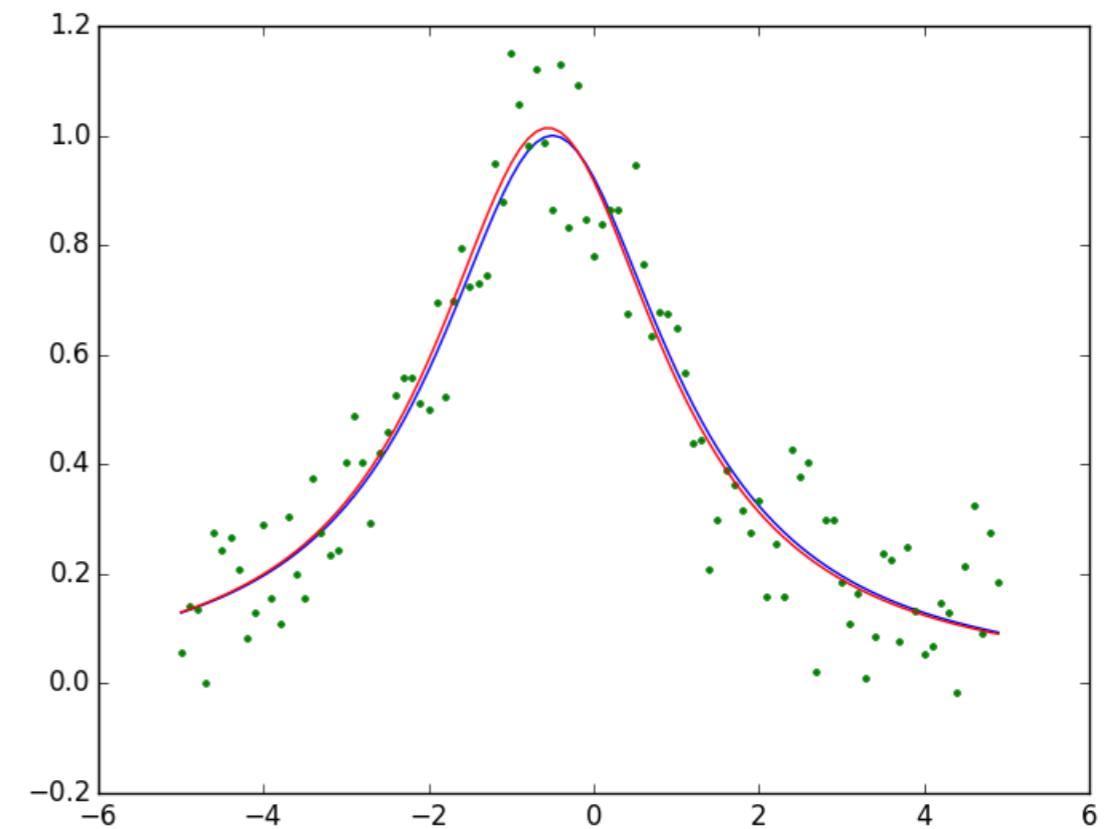
- Start with a guess for the parameters: m_0 .
- Calculate model $A(m_0)$ and local gradient A_m .
- Solve linear system $A_m^T N^{-1} A_m \delta m = A_m^T N^{-1} r$
- Set $m_0 \rightarrow m_0 + \delta m$.
- Repeat until δm is “small”. For χ^2 , change should be $<< 1$ (why?).
- Curvature estimate of χ^2 again ($A_m^T N^{-1} A_m$), and estimated errors on parameters again $\langle \delta m \delta m^T \rangle = (A_m^T N^{-1} A_m)^{-1}$.

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-
 \rightarrow curve+ Λ *diag(curve). For $\Lambda=0$ 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)

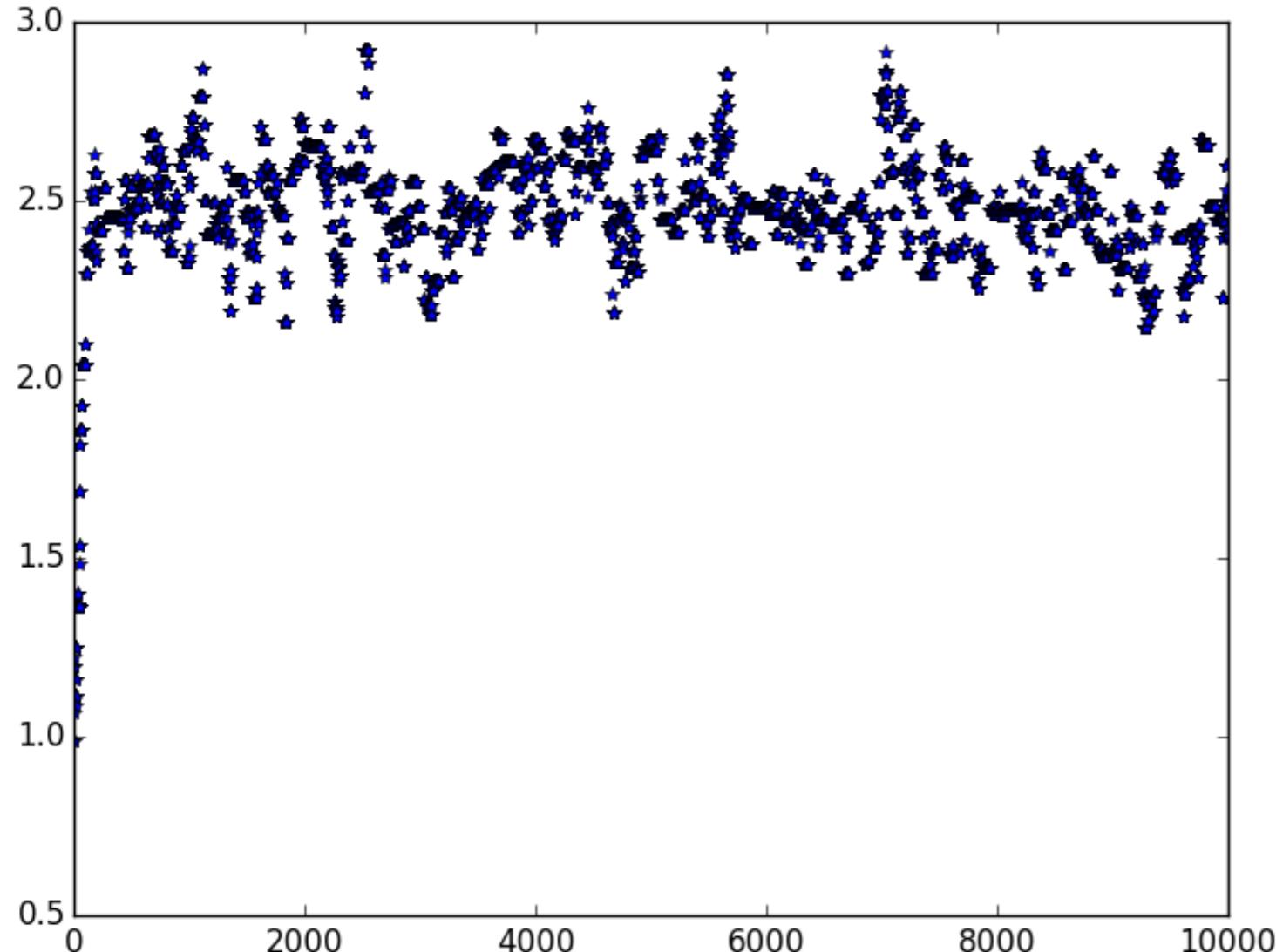
    #pick a random starting position, and guess some errors
    guess=numpy.array([0.3,1.2,0.3,-0.2])
    scale=numpy.array([0.1,0.1,0.1,0.1])
    nstep=10000
    chain=run_mcmc(dat,guess,nstep,scale)
    #nn=numpy.round(0.2*nstep)
    #chain=chain[nn:,:]

    #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]
```

- 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.

```
>>> 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]
```

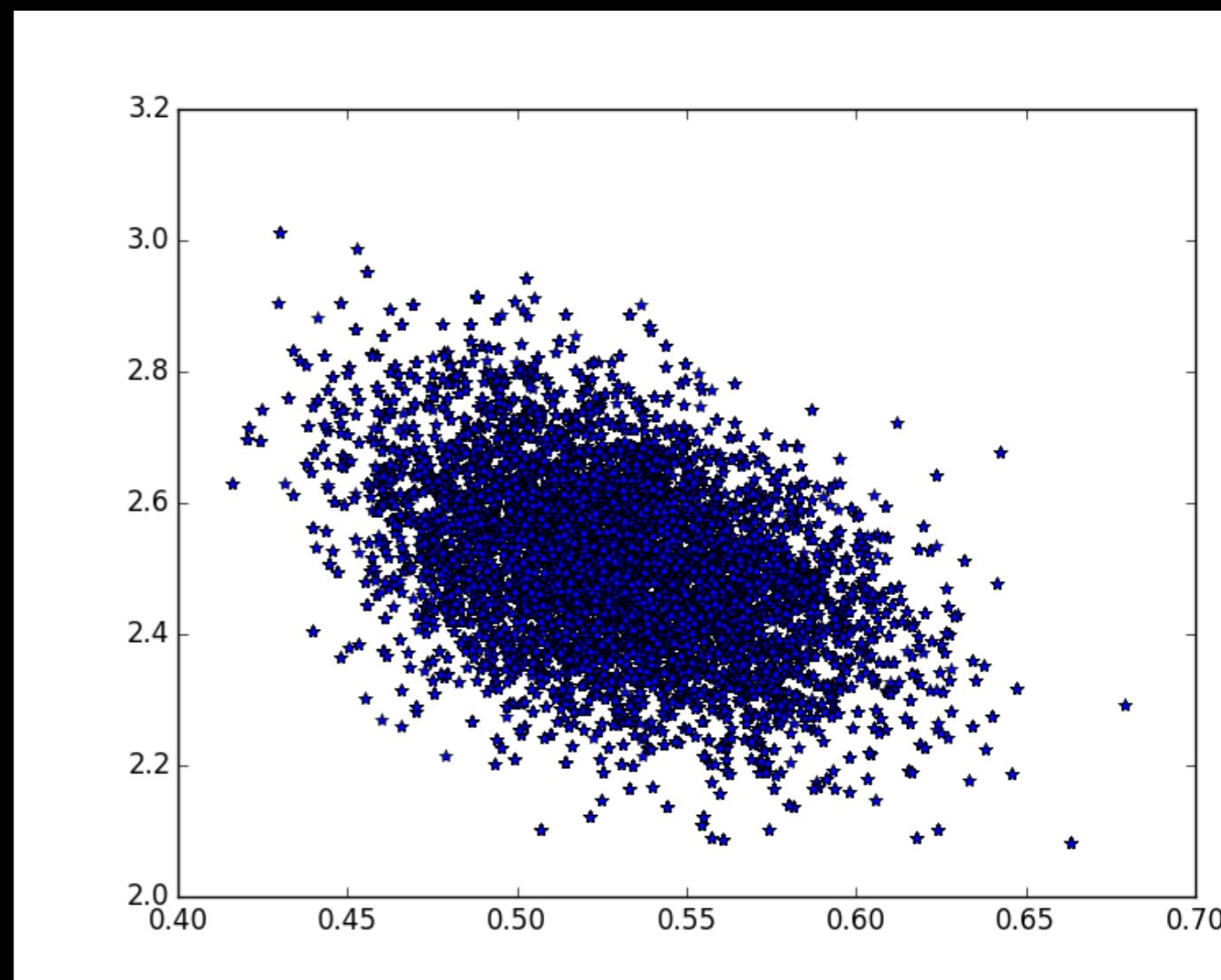
What Chain Looks Like



- 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.

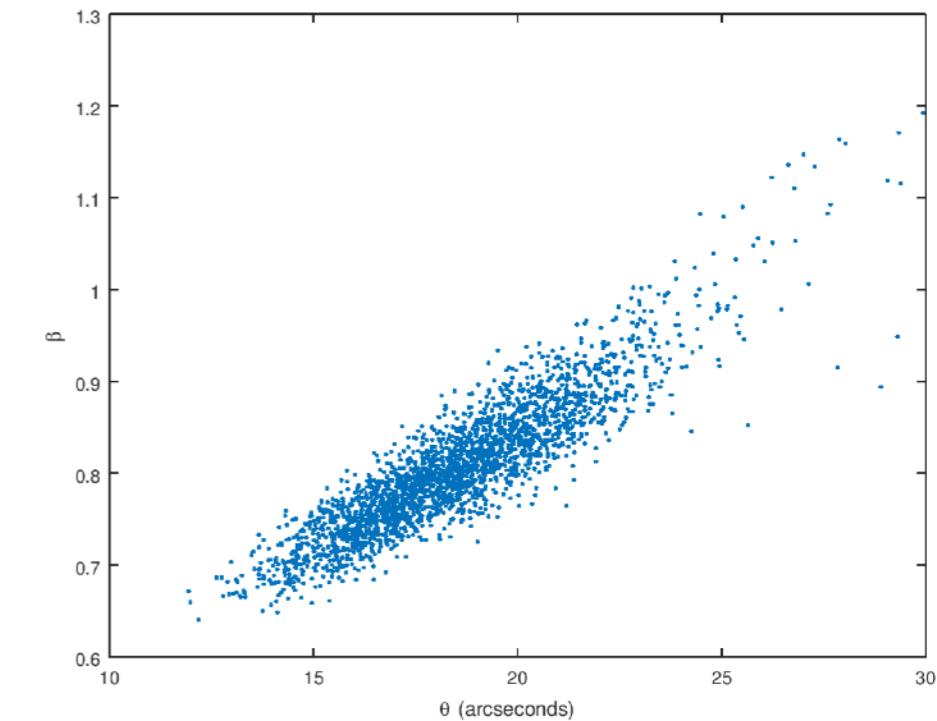
Covariances

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



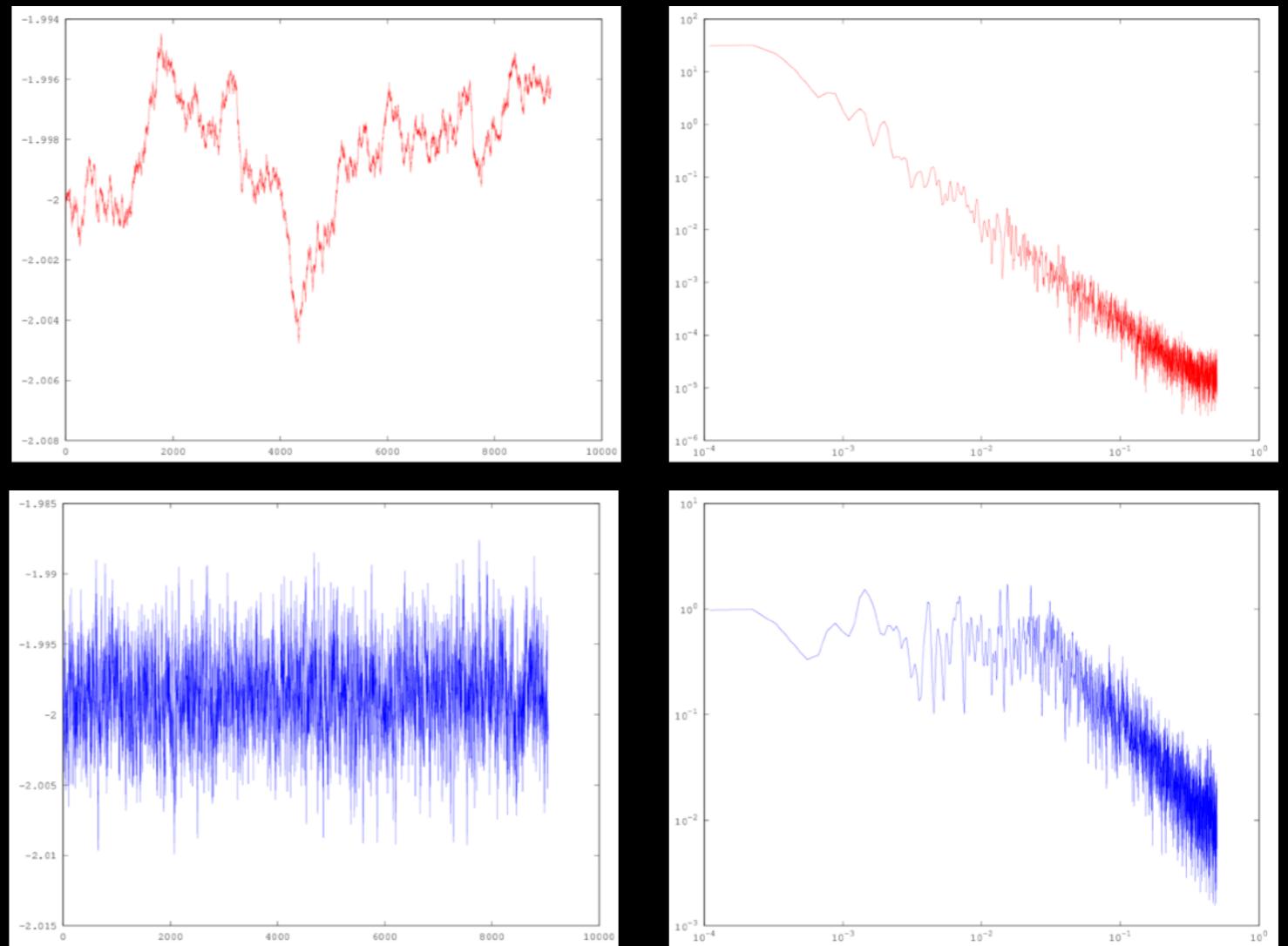
You Gotta Know When to Fold ‘em

- 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.



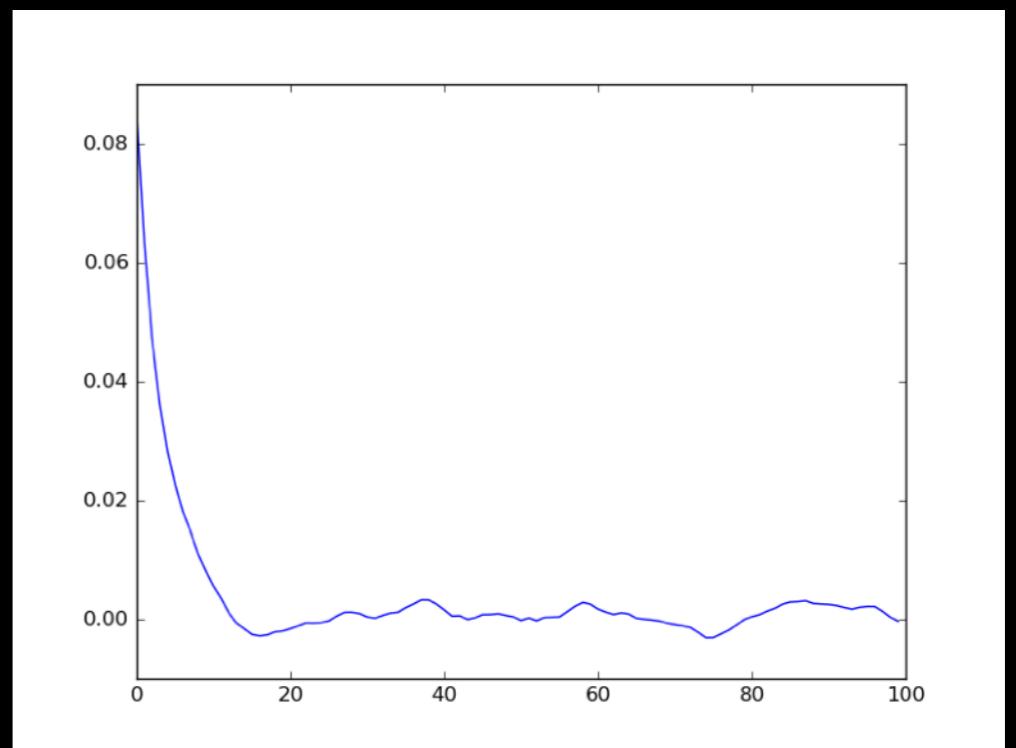
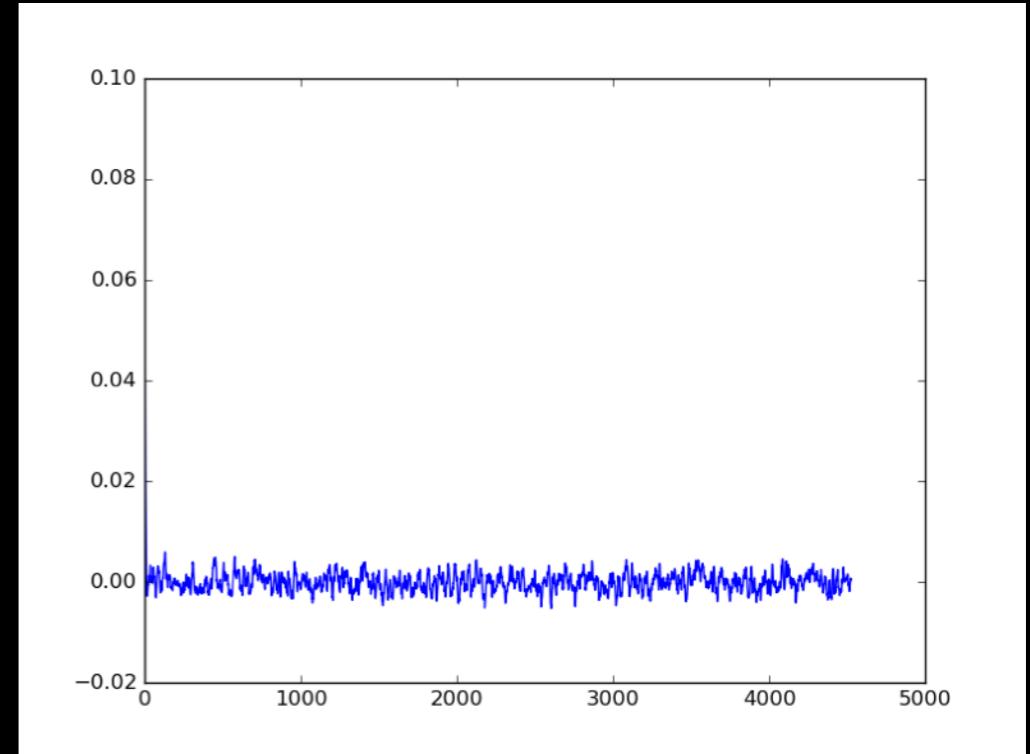
Single-Chain Convergence

- 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 $\sigma/\sqrt(n_{\text{eff}})$
- 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 and even Planck simple enough can get OK results just from raw power spectrum.

CMB Chains ctd.

- Planck code I've posted does a reasonable job reproducing parameters.
- How should we generate our trial steps?

```
  pars=camb.CAMBparams()
Click to close this tab; Option-click to close all 98]
  tabs except this one }, 4.57e-02, 2.0138e-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, 7.34e-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
```

CMB Chains ctd.

- Planck code I've posted does a reasonable job reproducing parameters.
- How should we generate our trial steps?
 - Standard trick is to run shorter chain to estimate covariance, then sample from that.
 - Highly recommended - run initial Newton's method. If that converges, use curvature.

```
98]
Click to close this tab; Option-click to close all tabs except this one
pars=camb.CAMBparams()
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, 7.34e-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
```

Aside: Correlated Samples

- We have a positive-definite matrix N . We want to draw samples such that $\langle n_i n_j \rangle = N_{ij}$.
- Simplest (but not only!) way is with Cholesky. Decompose $N=LL^T$, where L is a triangular matrix.
- Obviously, $N=LIL^T$. Add expectation: $\langle N \rangle = \langle LIL^T \rangle$.
- We'll be OK if we create data where $\langle dd^T \rangle = I$, but that's easy. Just use unit Gaussian random variables.
- We have $N = \langle nn^T \rangle = \langle Ldd^TL^T \rangle$, so $n = Ld$.
- In words, we can take Cholesky of a matrix, multiply by unit Gaussians, and get data with covariance of our matrix.

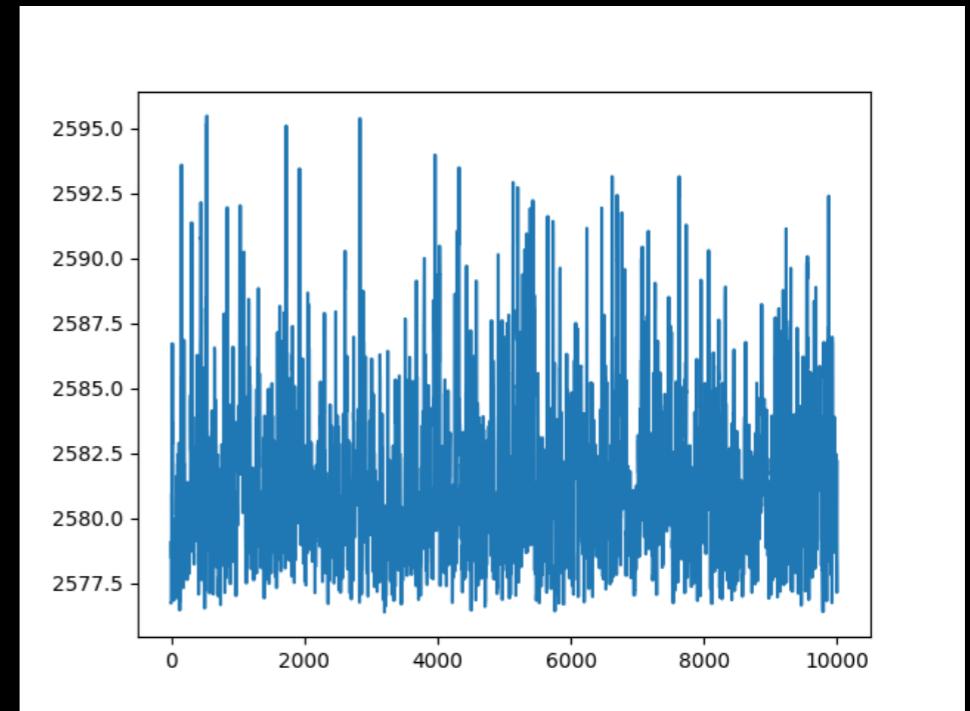
Code

- `fit_planck_newton.py` does everything! Pieces:
 - read in data, decide on parameters we want to fit.
 - do a Newton's method to find our maximum-likelihood parameters and approximate curvature near peak.
 - Run a Markov-chain using the inverse curvature at the peak for our sample steps.

Output

- I've posted chain output of a simple version of Planck (keeping τ fixed): `planck_no_tau.txt`.
- Means, standard deviations etc. can be calculated from the chains, as can covariances.
- Are the chains converged?
- Can you estimate means/errors on the parameters? Are any of the parameters correlated?

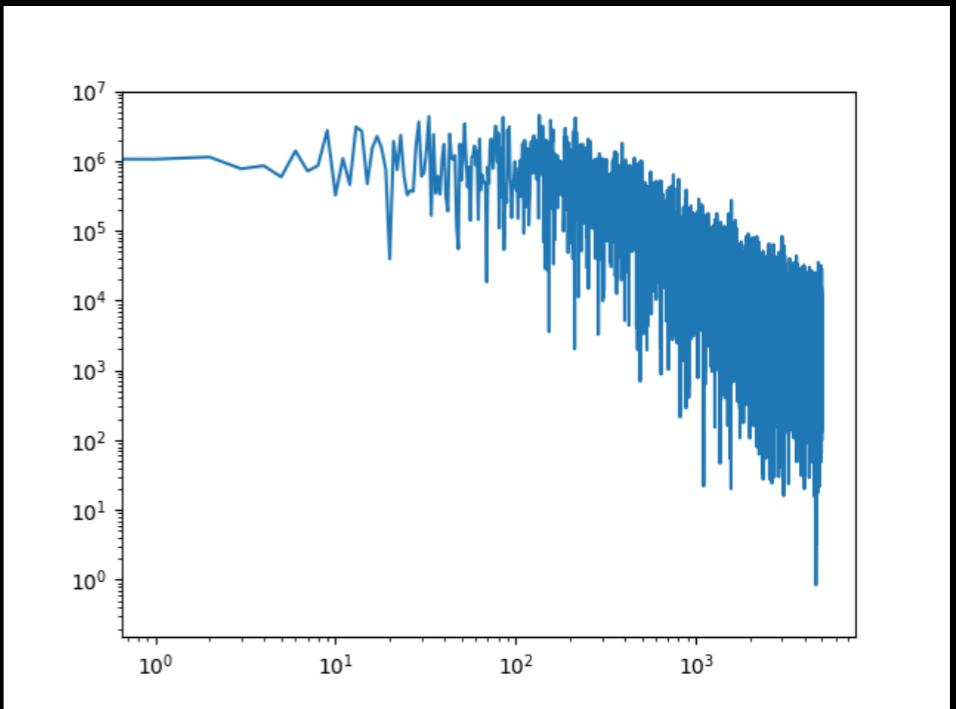
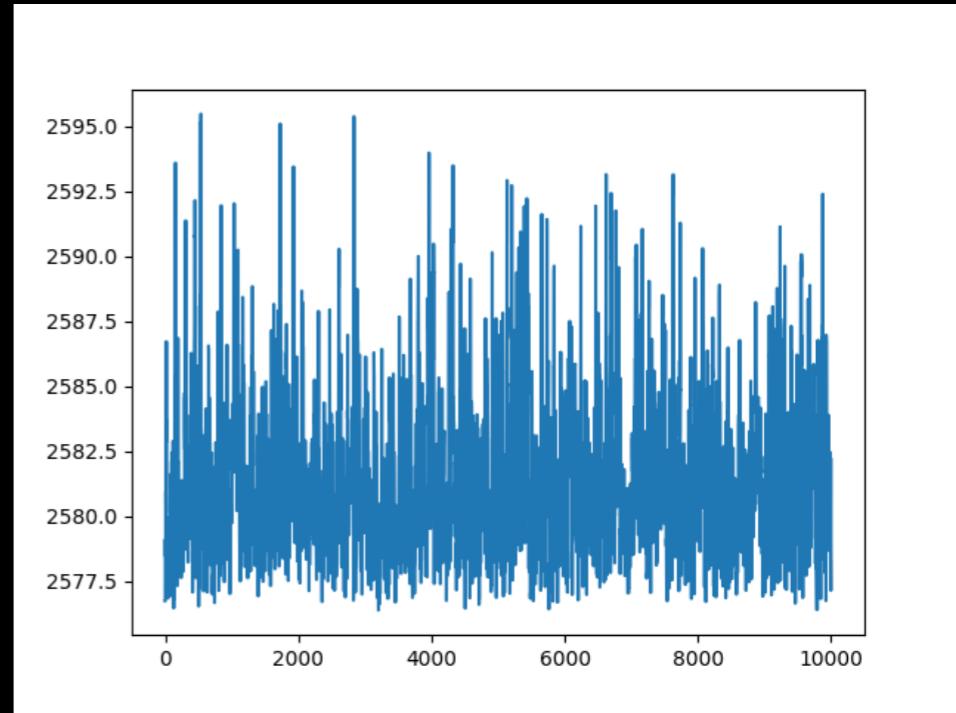
Convergence



- Right: χ^2 of our chain samples.
Does this look converged?

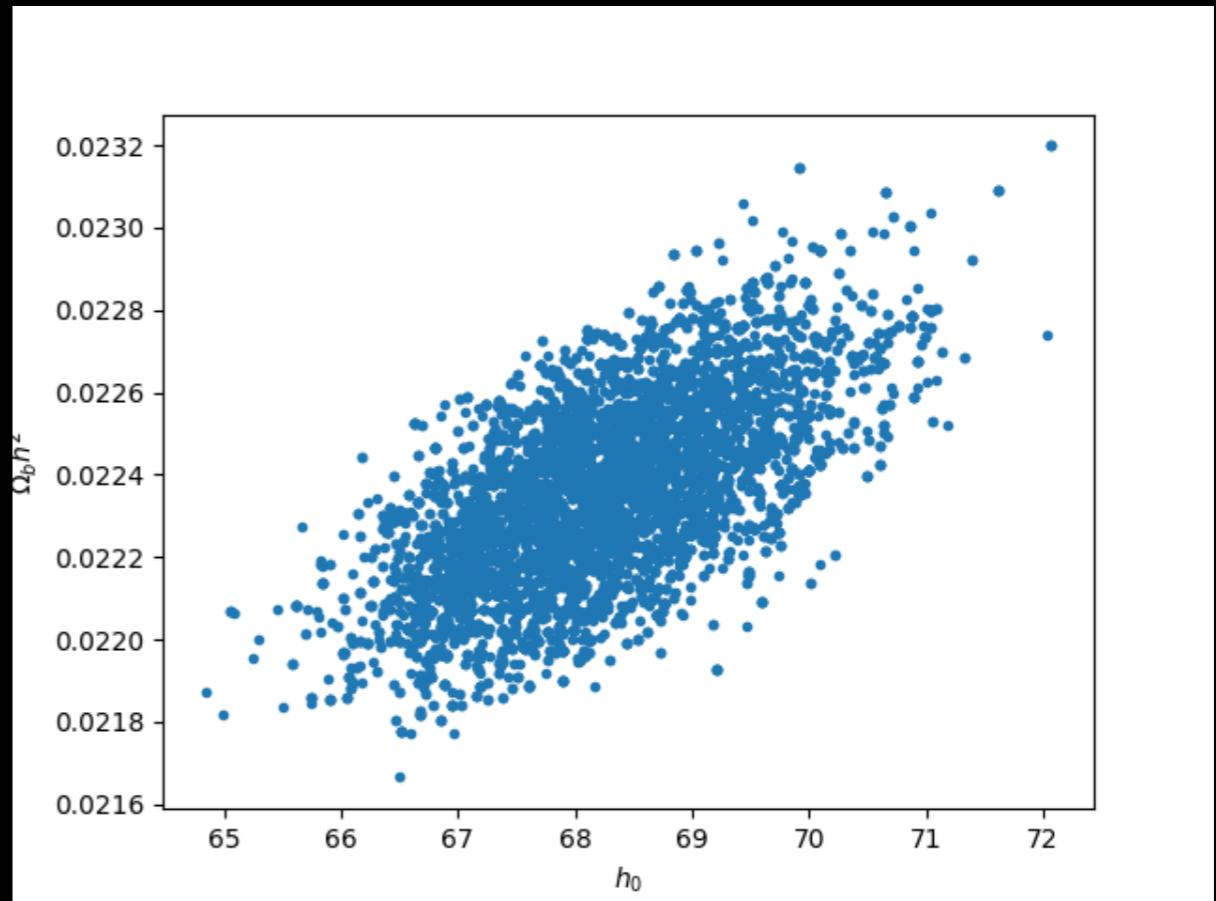
Convergence

- Right: χ^2 of our chain samples.
Does this look converged?
- Bottom right: power spectrum
of our χ^2 . How do you feel
about your convergence
choice now?



Param Scatter plot

- right: $h_0/\Omega_m h^2$ from the chain.
- How converged does this look?



Exercise

- Can you run a chain with τ floating?
- What would you expect the results to look like?
- How would you improve on TT-only?
- Can you tweak the Newton's method part to find our best universe with no dark matter?

Final Thoughts

- MCMC is an extremely powerful tool for mapping out complicated likelihood surfaces.
- Critically, we can get useful results even if we don't perfectly understand our problem.
- As presented, we won't do well at mapping out error bars at large- σ . How could we fix that (hint - think about changing the effective temperature. Then look up importance sampling.)

Final Thoughts