HW#5: Extracting cosmological parameters!

In the following document, I lay out the process and python code I used to fit the various cosmological parameters from the WMAP data set.

In [3]:

```
from __future__ import division
import numpy as np
import matplotlib.pyplot as plt
import scipy as sp
import scipy.optimize
import camb
import corner as cp
import time
import os
```

PART A)

Plotting the raw data power spectrum along side a power spectrum generated from a reasonable choice of cosmology parameters

In [4]:

```
wmap_data = np.genfromtxt("wmap_tt_spectrum_9yr_v5.txt")
ell = wmap_data[:,0]
tt = wmap_data[:,1]
tterr = wmap_data[:,2]
```

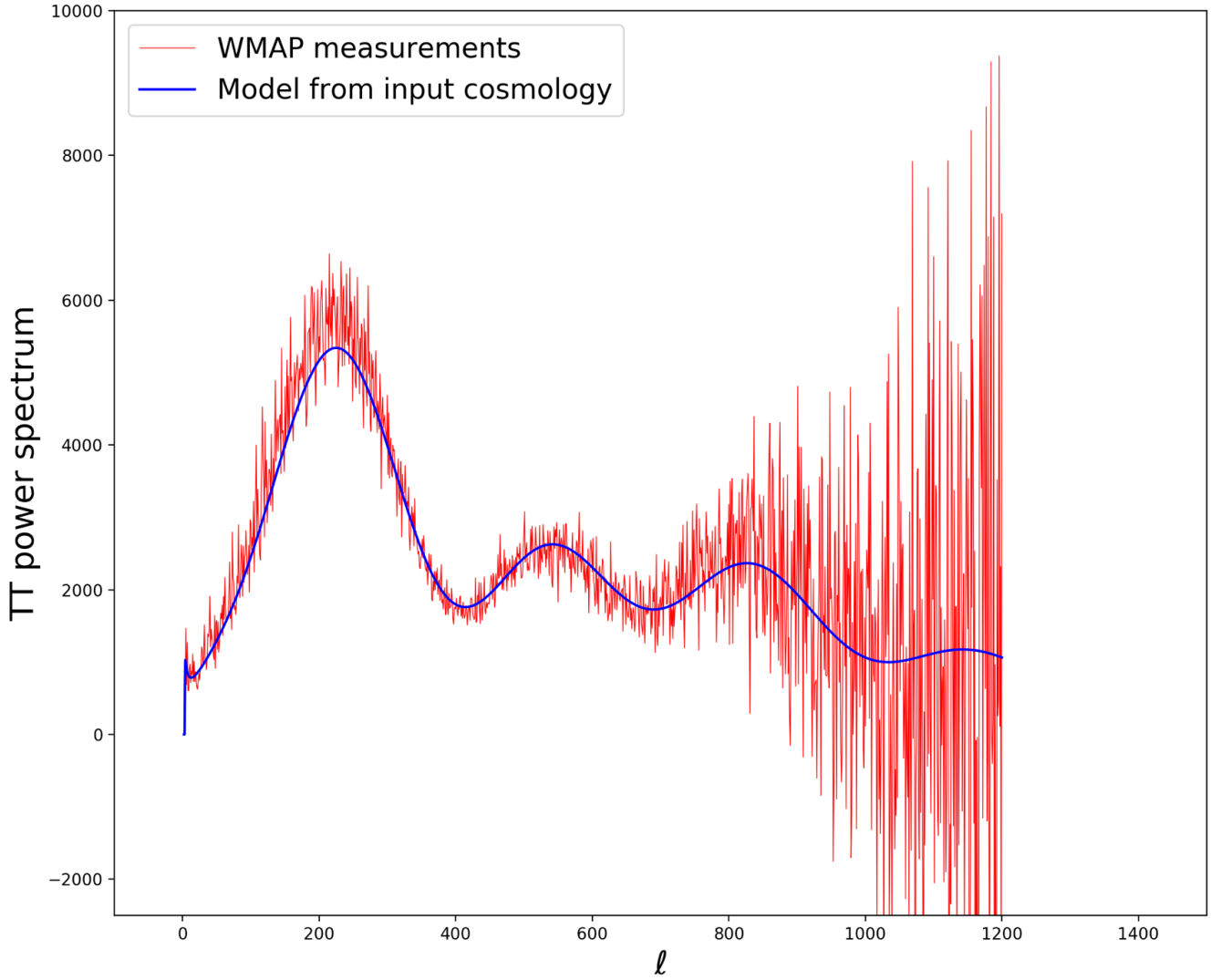
In [5]:

```
# don't need to touch these except to set max_l to something closer to our data
limit (for speed)
pars=camb.CAMBparams()
pars.set_for_lmax(1250)
print("max_l:",pars.max_l)
# I checked a before/after chi^2 and found it changed by ~0.3, so not too big of
a deal
# in exchange for the ~50% speed up I got


# (from LCDM wmap9+spt+act+snls3)
#                              Obh2    Och2  As  h0     ns    tau
test_cosmology = np.asarray([0.02237,0.1112,1.1e-9,71.2,0.9674,0.086])
# the e-9 on the As parameter isn't listed on Lambda, but it was that small
# in the class notes.  Maybe they report a scaled version?

# Also tried this, from a class demo
test_cosmology = np.asarray([0.02,0.1157,2.125e-9,70,1,0.05])


# get a model PS, truncate to ell where we have data
results=camb.get_results(pars)
powspec=results.get_cmb_power_spectra(pars,CMB_unit='muK')['total'][:,0]
powspec=powspec[:len(tt)]
ell_model = np.arange(2,len(powspec)+2)
```

```
max_l: 1400
```

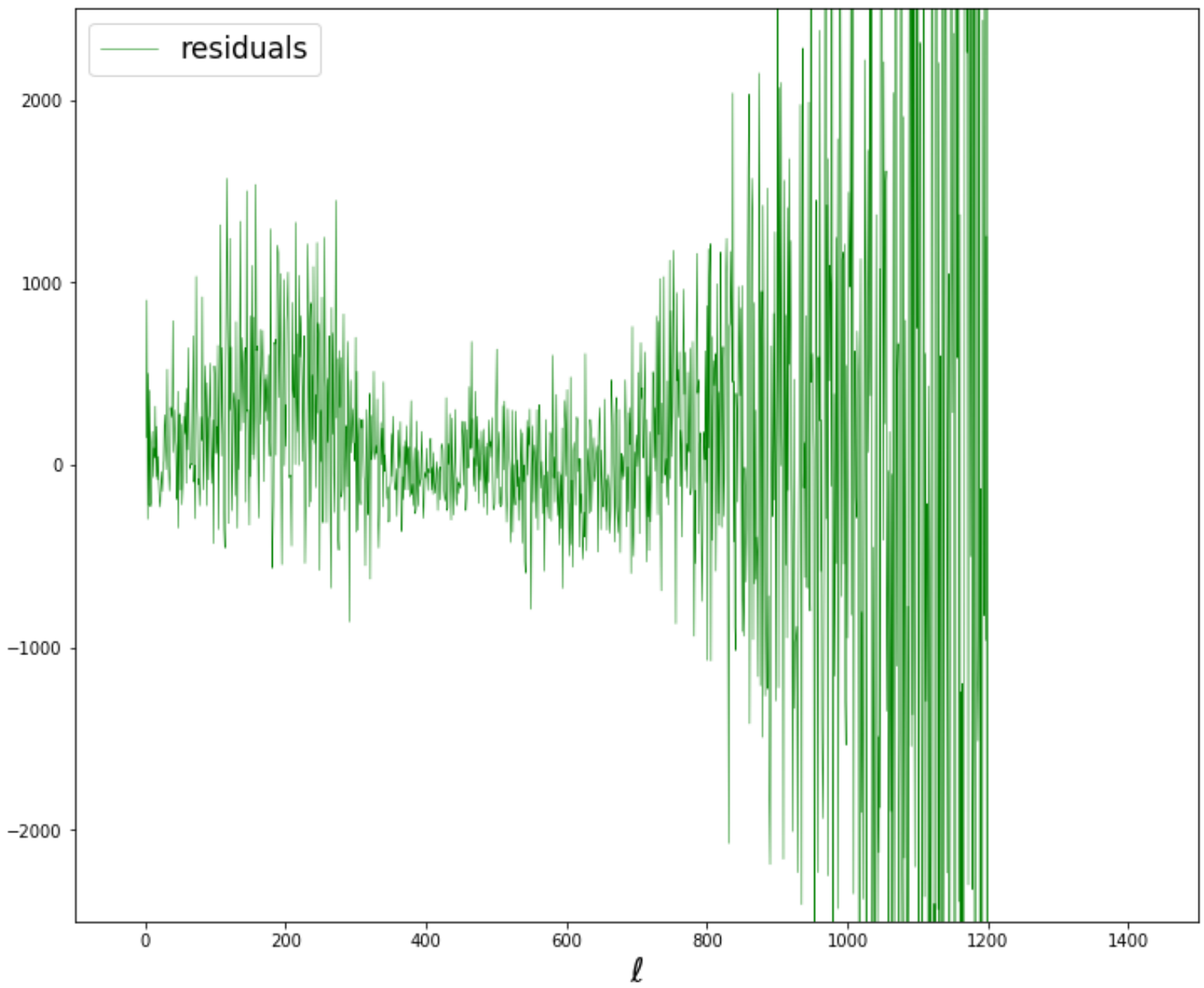
In [195]:

```
# Plot the model and the data together
plt.figure(num=1,figsize=(12,10))
plt.clf()

plt.plot(ell,tt,'r-',lw=0.5,label="WMAP measurements")
plt.plot(ell_model, powspec,'b-',label="Model from input cosmology")

plt.xlabel(r'$\ell$',fontsize=20)
plt.ylabel("TT power spectrum",fontsize=20)
plt.xlim(-100,1500)
plt.ylim(-2500,10000)
plt.legend(fontsize=17)
plt.show()
```


WMAP measurements
Model from input cosmology
TT power spectrum
10000
8000
6000
4000
2000
0
−2000
0
200
400
600
800
1000
1200
1400
ℓ

In [5]:

```
# Plot the residuals
plt.figure(num=2,figsize=(12,10))
plt.clf()
plt.plot(ell,tt-powspec,'g-',lw=0.5,label="residuals")
plt.xlabel(r'$\ell$',fontsize=20)
plt.xlim(-100,1500)
plt.ylim(-2500,2500)
plt.legend(fontsize=17)
plt.show()
```



These plots show that I haven't chosen parameters that are WAY off, but the resisduals tell me there's plenty of room for improvement

PART B)

Getting the MCMC code in shape

First we define a bunch of helper functions that will allow the MCMC function to run. These are, of course, very similar to the ones we set up in class. I have altered a few though.

In [6]:

```
def calc_chi2(wmap_data,modelPS):
    """
    Return chi^2 given a set of TT power spectrum data and a model TT power spec
trum
    """
    ell = np.asarray(wmap_data[:,0],dtype='int') #first column is ell
    delta = wmap_data[:,1] - modelPS[ell]        #next column is TT power spectr
um
    chi2 = np.sum( (delta/wmap_data[:,2])**2)    #third column is err of TT
    return chi2

def update_model(cosmology,pars):
    """
    Having chosen a new cosmology, we need camb to update the pars
    """
    pars2 = pars.copy()
    pars2.set_cosmology(ombh2=cosmology[0],omch2=cosmology[1],H0=cosmology[3],ta
u=cosmology[5])
    pars2.InitPower.set_params(As=cosmology[2],ns=cosmology[4])
    return pars2

def get_power_spectrum(cosmology,pars):
    """
    With new input cosmology, update pars and make a new power spectrum
    """
    pars2 = update_model(cosmology,pars)
    results = camb.get_results(pars2)
    modelPS = results.get_cmb_power_spectra(pars2,CMB_unit='muK')['total']
    return modelPS

def get_chi2_from_cosmology(cosmology,wmap_data,pars):
    """
    For input cosmology, we build a power spectrum, then calculate the chi^2 of
    the data against that model
    """
    modelPS=get_power_spectrum(cosmology,pars)
    modelPS=modelPS[:,0] # only want the TT power spectrum
    chi2 = calc_chi2(wmap_data,modelPS)
    return chi2


def get_step(step_size):
    """
    Take gaussian steps for each parameter with std dev according to step_size
    """
    return np.random.randn(len(step_size)) * step_size * scale_steps


def get_cov_step(mat,nset=1):
    """
    Adapted from the HW2 problem.  Given a covariance matrix, produce a random s
tep for each
```

```
tep for each
    parameter taken in a correlated manner (estimated from a previous short chai
n).
    """
    e,v=np.linalg.eigh(mat)
    e[e<0]=0 #make sure we don't have any negative eigenvalues due to roundoff
    n=len(e)
    #make gaussian random variables
    g=np.random.randn(n,nset)
    #now scale them by the square root of the eigenvalues
    rte=np.sqrt(e)
    # This loop was for the general case of requesting mulitple samples.
    # I'm only ever asking for 1 here
    for i in range(nset):
        g[:,i]=g[:,i]*rte
    #and rotate back into the original space (and make sure the shape is compata
ble)
    dat=np.dot(v,g).reshape((1,n)) * scale_steps
    return dat[0]




def MCMC(cosmology,wmap_data,pars,step_size,outfile,nstep=100,printind=0,cov_mat
=None):
    """
    Runs the MCMC chain.
    Requires as inputs:
        1) initial guess cosmology: ([Obh2, Och2, As, h0, ns, tau])
        2) wmap data set (1st col = ell, 2nd col = TT, 3rd col = err(TT))
        3) camb pars that match initial cosmology guess
        4) step size (std dev of gaussian steps for each cosmology parameter)
        5) **optional** number of steps (default 100 if not specified)
        6) **optional** index of chain parameter to print out in real time
        7) **optional** covariance matrix of chain parameters.  If "None" it wil
l
            take steps based on step_size input
    """
    if cov_mat is not None: print("--Using covariant steps--")
    print("Steps scaled by:",scale_steps)
    naccept=0
    chain = np.zeros([nstep,len(cosmology)+1])
    chi2_now = get_chi2_from_cosmology(cosmology, wmap_data, pars)

    # main loop
    for iter in range(nstep):
        if cov_mat is not None:
            new_cosmology = cosmology + get_cov_step(cov_mat)
        else:
            new_cosmology = cosmology + get_step(step_size)

        new_chi2 = get_chi2_from_cosmology(new_cosmology,wmap_data,pars)
```

```
        like = np.exp(-0.5*(new_chi2-chi2_now))

        accept = np.random.rand()<like

        # don't accept negative tau values
        if new_cosmology[5]<0:
            accept=False
            print("--> Rejected for negative tau")

        #print some output to see how the chain's doing
        print("%4i, %12.5f, %12.5f, %6s, %12.4e, %10.3e, %7.3f"%(iter,chi2_now,n
ew_chi2,accept,new_cosmology[printind],like,naccept/(iter+1)))

        if accept:
            cosmology = new_cosmology
            chi2_now = new_chi2
            naccept+=1

        chain[iter,0] = chi2_now
        chain[iter,1:]=cosmology

        # this will feed into file in real-time rather than waiting to finish ch
ain
        datastring = "%12.5e %12.5e %12.5e %12.5e %12.5e %12.5e %12.5e"%tuple(ch
ain[iter,:])
        commandstring = 'echo "%s" >> %s'%(datastring,outfile)
        os.system(commandstring)

    frac_accept=naccept/nstep
    print("Accepted %.1f percent of steps"%(frac_accept*100))
    return chain,frac_accept
```

Initially, I don't know what's a good step size or starting point for the cosmological parameters.

Let's look at one parameter at a time. This block runs a very short MCMC chain with all step sizes held to zero except for the one I'm exploring. After some trial and error, I found a combination of initial cosmological parameter value and reasonable step size that allows me to trace out the 1D χ^2 χ^2 minimum for each parameter.

In [1]:

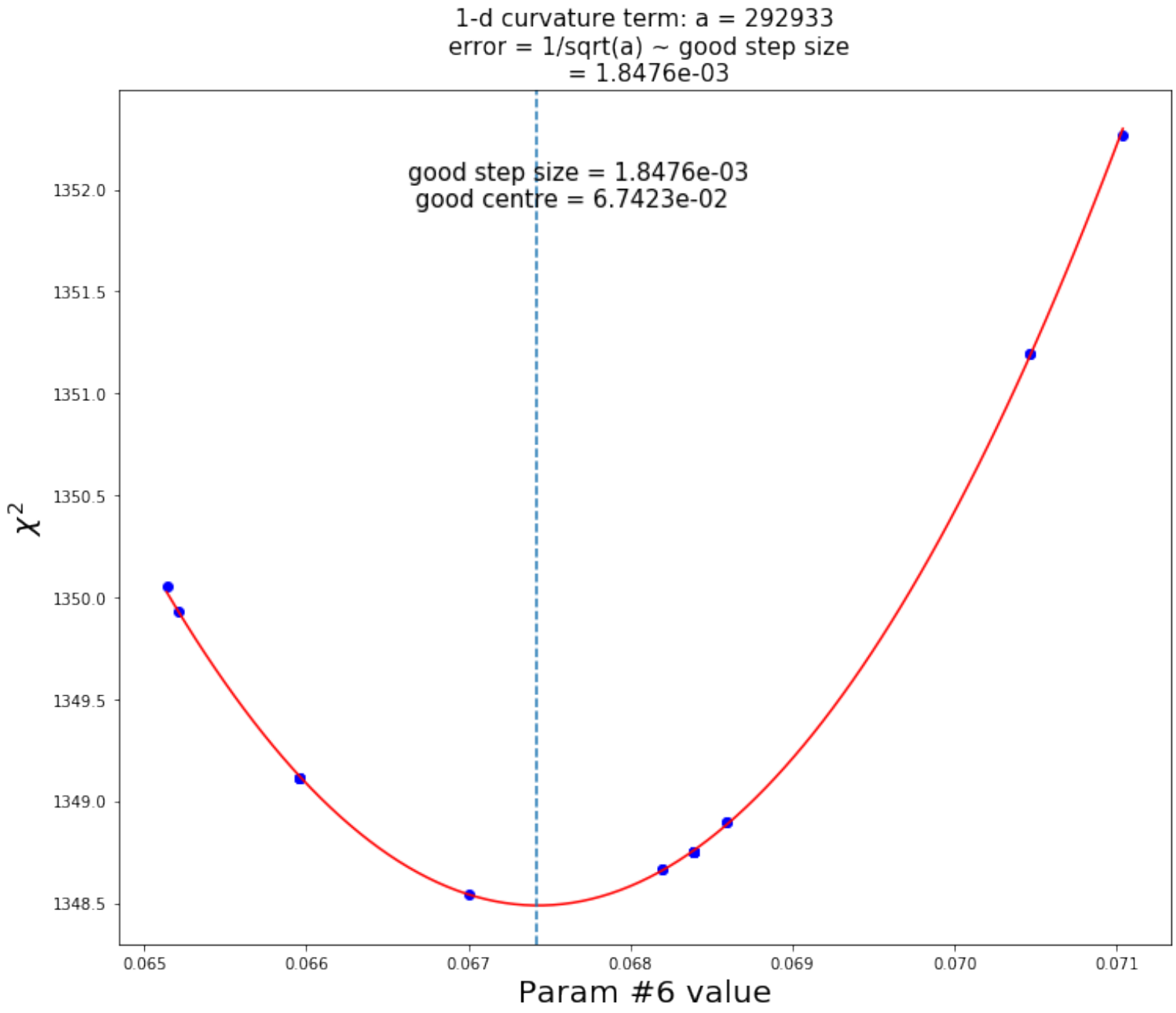
```
# which param in cosmology are we going to explore
#  (counting from 1, not zero because chi^2 takes up 0th index later)
index = 6

#cosmology0 = np.asarray([0.02237,0.1112,1.1e-9,71.2,0.9674,0.086])
cosmology0 = np.asarray([0.02,0.1157,2.125e-9,70,0.96,0.067])
pars = camb.CAMBparams()
step_size=np.asarray([0,0,0,0,0,0.003])# just looking at one at a time for now

print("   i      old X^2       new X^2   accept      value     likelihood")
chain,f_accept = MCMC(cosmology0,wmap_data,pars,step_size,nstep=20,printind=inde
x-1)

np.savetxt("testchain_param"+str(index)+".txt",chain)

#chain indices: [ chi2, Obh2, Och2, As, h0, ns, tau ]

"""
Record the results that I found to be good for initial guess and step size via 1
D search:

cosmology0 = np.asarray([2.33e-2, 0.11583, 2.09e-9,  65.8, 0.971,   0.0674])
step_size  = np.asarray([2.31e-4, 7.76e-4, 7.63e-12, 0.66, 4.92e-3, 1.84e-3])
"""
```

This next segment takes in a small test chain that varies only one cosmology parameter (I did 20 iters) and finds the 1D curvature. I use this to find the minimum χ^2 χ^2 and estimate of the error ($1/\sqrt{a}$ $1/\sqrt{a}$ where aa is the quadratic coefficient fit to the 1D χ^2 χ^2 surface). Results are the 6 figures labeled "B_param#_1d_curvature.png".

In [254]:

```
def quad(x):
    return p[0]*x**2+p[1]*x+p[2]

# fit a quadratic to the chi^2 surface
p=np.polyfit(chain[:,index],chain[:,0],2)
x=np.linspace(min(chain[:,index]),max(chain[:,index]),100)
y=quad(x)

minimum = sp.optimize.fmin(quad, np.mean(x))

plt.figure(num=3,figsize=(12,10))
plt.clf()
plt.title("1-d curvature term: a = %.0f\n error = 1/sqrt(a) ~ good step size\n =
%.4e"\
          %(p[0],1/np.sqrt(p[0])),fontsize=15)

plt.plot(chain[:,index],chain[:,0],'bo')
plt.plot(x,y,'r-')

ytext = np.max(y)-0.1*(np.max(y)-np.min(y))
plt.text(x[25],ytext,"good step size = %.4e \n good centre = %.4e"\
         %(1/np.sqrt(p[0]),minimum),fontsize=15)


plt.axvline(minimum,linestyle='--')

plt.xlabel("Param #"+str(index)+" value",fontsize=20)
plt.ylabel(r"$\chi^2$",fontsize=20)
plt.savefig("figures/B_param"+str(index)+"_1d_curvature.png")
plt.show()
```

```
Optimization terminated successfully.
         Current function value: 1348.487699
         Iterations: 8
         Function evaluations: 16
```



Now that I've got a decent idea of the apparopriate step sizes and parameter locations (see figures called *testchain_paramX.txt*), I can run an MCMC chain that steps all parameters at once. This generally requires me to scale down my 1D guesses for the step sizes since steps in 6 dimensions will cause χ^2 χ^2 to change much more dramatically than in 1 dimension.

Through some trial and error, I found a combination of initial values and steps that gave a reasonable convergence after a couple thousand iterations (see figures for "run3"). This run had an acceptance rate of ~19%. I fiddled aroudn the the step sizes a bit and settled on a scaling of about 0.7 of the 1D estimate step sizes (though the scaling ranged from 0.57-0.92).

As can be seen in the figure called *chain_overview_run3.png*, in this run $\Omega_b h^2$ $\Omega_b h^2$, $\Omega_c h^2$ $\Omega_c h^2$, H_0 H_0, and n_s n_s converged pretty well, but A_s A_s and τ τ didn't really. The correlation lenght (see figure *corr_overview_run3.png*) is bad (>2000), but I was able to use this run to build a rough covariance matrix for the parameters.

In [7]:

```
# Covariance matrix
cov_chain = np.genfromtxt("chain_ALLparam_run3.txt")

cov = np.cov(cov_chain[:,1:].T)
```

Now I used this covariance matrix to take steps in a correlated way. Since $A_s A_s$ and $\tau\tau$ are highly correlated, these steps are taken in such a way as to take long steps along the long axis of their likelihood contour and short steps along the short axis. This allowed the subsequent chain to converge much faster when taking steps this way.

The following code block below allows me to:

```
- Run a chain with a set of starting parameters, step sizes, and scaling o
f steps
- Take correlated steps by including a covariance matrix
- Load a chain that was previously calculated instead of running a new one
- Resume a chain from where it left off to extend it
```

In [25]:

```
"""
Took 35737.2 seconds to run 16000 iters
  i.e. 2.23 seconds per iter
    or 1611 iters per hour

overnight, run 9 hrs (10pm-7am) --> 14,500 iters
"""

""" RUNNING MCMC WILL ADD LINES TO BOTTOM OF FILE (or create new file if new nam
e is given)"""

name="run8"
nsteps= 20
cov_mat = cov
scale_steps = np.asarray([1.,1.,1.,1.,1.,1.])*0.5
index = 6 #print out value:  1=Obh2, 2=Och2, 3=As, 4=h0, 5=ns, 6=tau

loadchain=True; loadname=name
restart  =False; restartfrom=name

if loadchain==False:
    # should have chi2 = 1.23e+03
    cosmology0 = np.asarray([ 2.26e-02, 1.10e-01, 2.08e-09, 7.20e+01, 9.73e-01,
6.61e-02]) #good chi2
    step_size = np.asarray([4e-4, 1.26e-3, 1e-11, 0.8, 7e-3, 2e-3])#run3

    if restart:
        lastchain = np.genfromtxt("chain_ALLparam_"+restartfrom+".txt")[-1]
        cosmology0=lastchain[1:]


    outfile="chain_ALLparam_"+name+".txt"

    start = time.time()
    print("   i      old X^2       new X^2   accept      value     likelihood  f
rac accept")
    chain,f_accept = MCMC(cosmology0,wmap_data,pars,step_size,outfile,nstep=nste
ps,printind=index-1,cov_mat=cov_mat)
    end = time.time()
    print("Took %.1f seconds to run %i iters"%(end-start,nsteps))
    print("  i.e. %.2f seconds per iter"%((end-start)/nsteps))
    print("    or %.i iters per hour"%(3600/((end-start)/nsteps)))

else:
    chain = np.genfromtxt("chain_ALLparam_"+loadname+".txt")


#chain indices: [ chi2, Obh2, Och2, As, h0, ns, tau ]
```

In [27]:

```
np.asarray([2.31e-4, 7.76e-4, 7.63e-12, 0.66, 4.92e-3, 1.84e-3])/np.asarray([4e-
4, 1.26e-3, 1e-11, 0.8, 7e-3, 2e-3])
```

Out[27]:

```
array([0.5775    , 0.61587302, 0.763     , 0.825     , 0.70285714,
       0.92      ])
```

In [ ]:

```
"""
Here I'm keeping track of values that I used in certain chain runs.
Won't be very meaningful to you, the grader.

FROM 1D CURVATURE TESTS:
cosmology0 = np.asarray([2.33e-2, 0.11583, 2.09e-9,  65.8, 0.971,   0.0674])
step_size  = np.asarray([2.31e-4, 7.76e-4, 7.63e-12, 0.66, 4.92e-3, 1.84e-3])

AFTER SHORT FULL-PARAMETER MCMC
LOOKING AT SCATTER AROUND MEAN:
cosmology0 = np.asarray([2.225e-2, 0.114, 1.85e-9,  69.0,   0.97,   0.0674])
step_size  = np.asarray([5.97e-4, 5.27e-3, 3.62e-11, 2.59, 1.53e-2, 2.26e-02])
                                                                    ^might be t
oo large...
BY EYE
step_size = np.asarray([4e-4, 1.26e-3, 1e-11, 0.8, 7e-3, 2e-3])
produces acceptance rate of ~0.189


SUMMARY OF DIFFERENT CHAIN RUNS:

run2:
    15,000 iters, tau was allowed negative --not useful--

run3:
    16,000 iters , f_accept = 19%
    step_size = np.asarray([4e-4, 1.26e-3, 1e-11, 0.8, 7e-3, 2e-3])
    Obh2 - corr 750, chi2 isparabolic
    Och2 - corr 2300, chi2 is parabolic
    As   - corr 2000, chi2 looks quite random
    H0   - corr 2400, chi2 is parabolic
    ns   - corr 1400, chi2 is parabolic
    tau  - corr >2500, chi2 looks quite random

    --extended another 14,500 iters
    --extended yet another 14,500 iters

    --USEFUL IN CREATING COVARIANCE MATRIX--

run4:
      -- increase step size for Och2(a bit), As(moderately), H0(a bit), tau(a lo
```

```
t)

    step_size = np.asarray([4e-4, 1.6e-3, 3e-11, 1.0, 7e-3, 1e-2])


run5:
    -- steps are generated from covariance matrix of run3
     (SOMETHING WENT VERY WRONG HERE)
        +/- errors from cov cornerplot [5.77e-04 6.23e-03 8.90e-11 2.84e+00  1.5
6e-02  2.31e-02]

run6:
    -- steps are from run3 cov matrix, but scaled by 1/4
     - accepted close to 50%, some params veered way off expected values

run7:
    -- steps from run3 cov matrix, but scaled by 1/2

run8:
    -- cov matrix steps.  scaled by 0.5.  Chose a better starting point
    -- A VERY CONVERGENT RUN ON VALUES THAT SEEM LEGITIMATE
       (acceptance is a little high -61%- could maybe scale steps a little bigge
r)
         acceptance rate stabilizes by about 500 iters


run9:
    -- cov matrix scaled by 1/1.5.  Go with good starting point
    -- set lmax to lower value (2500 -> 1300) should run faster
"""
```

My best chain

This next block takes a look at the chain produced above. On the left side are plots of the chain values. On the right are the χ^2 χ^2 vs parameter plots.

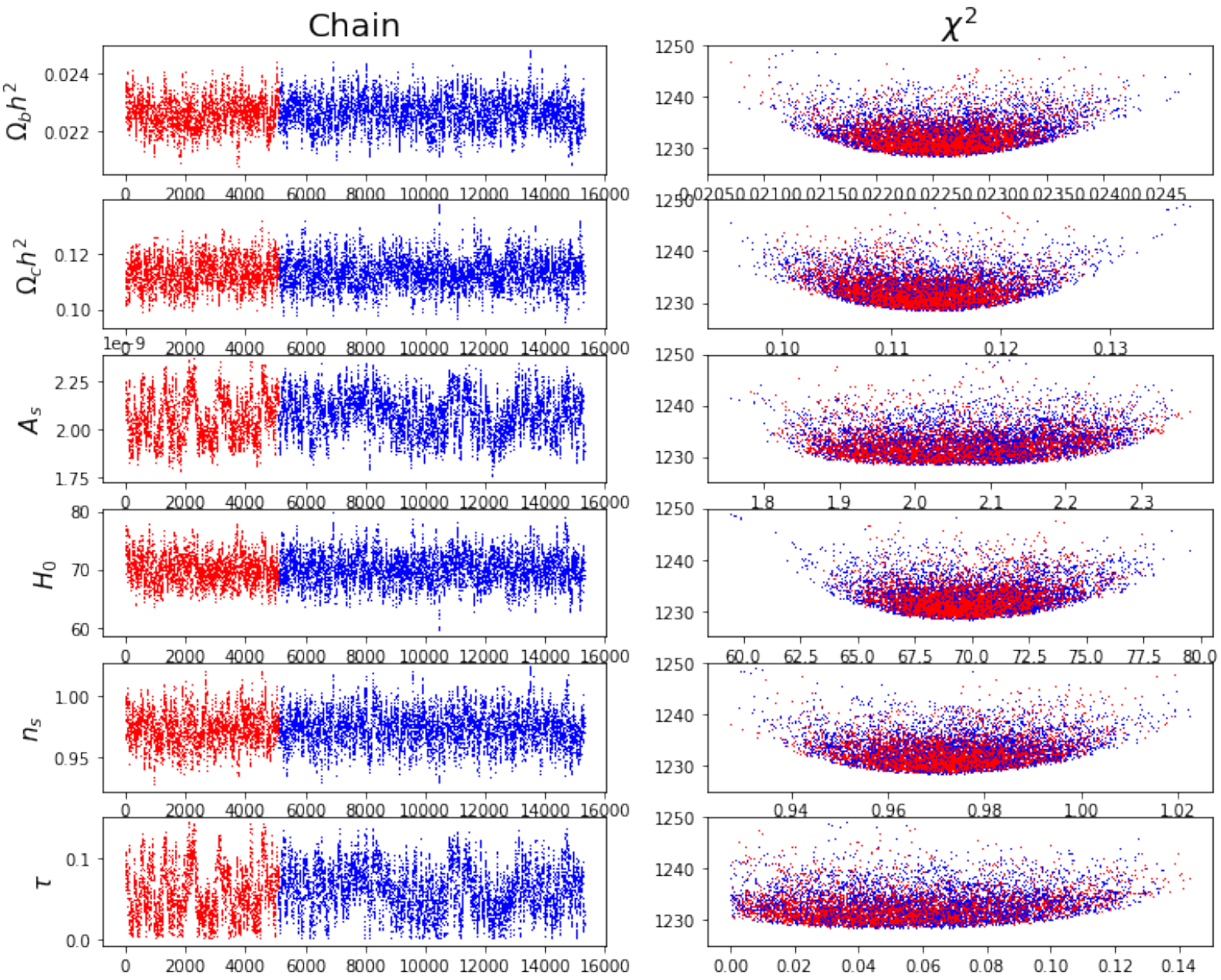
My best chain (run8) came from using the covariance matrix to pick correlated steps. Here you can see how the chain points make a fairly constant fuzz around the converged value without too much obvious evidence of correlation.

(The red points are the first third of the chain, which I was considering the burn-in at one point. It's pretty clear that it has converged long before this though.)

In [9]:

```
paramnames=[r"$\Omega_b h^2$",r"$\Omega_c h^2$", "$A_s$", "$H_0$", "$n_s$", r"$\
tau$"]
plt.figure(figsize=(12,10))
plt.clf()
burn=np.int(len(chain[:,0])/3)
for i in np.arange(6)+1:
    chainvalues = chain[:,i]
    chi2values = chain[:,0]
    plt.subplot(6,2,2*i-1)
    plt.plot(chainvalues,'b,')
    plt.plot(chainvalues[:burn],'r,')
    plt.ylabel(paramnames[i-1],fontsize=15)
    #plt.xlim(6000,7000)
    plt.subplot(6,2,2*i)
    plt.plot(chainvalues,chain[:,0],'b,')
    plt.plot(chainvalues[:burn],chi2values[:burn],'r,')
    plt.ylim(1225,1250)

plt.subplot(6,2,1); plt.title("Chain",fontsize=20)
plt.subplot(6,2,2); plt.title(r"$\chi^2$",fontsize=20)
plt.savefig('figures/chain_overview_'+name+'.png')
plt.show()
```

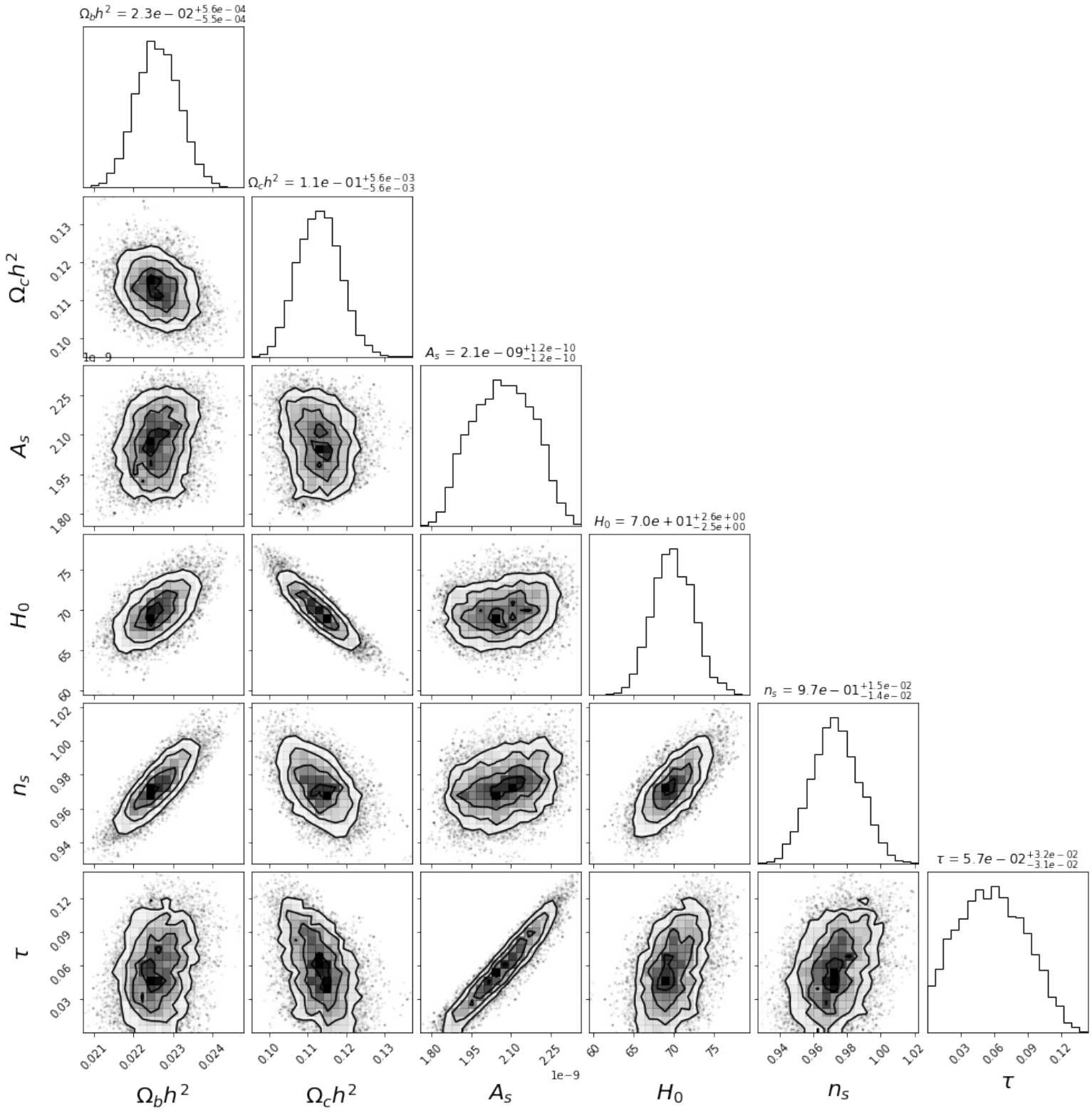
```
/Applications/anaconda/envs/forjupyter3/lib/python3.6/site-packages/
matplotlib/cbook/deprecation.py:107: MatplotlibDeprecationWarning: A
dding an axes using the same arguments as a previous axes currently
reuses the earlier instance.  In a future version, a new instance wi
ll always be created and returned.  Meanwhile, this warning can be s
uppressed, and the future behavior ensured, by passing a unique labe
l to each axes instance.
  warnings.warn(message, mplDeprecation, stacklevel=1)
```



By making a corner plot, we can see how all the parameters correlate with one another, and what their best fit values and uncertainties are (in the titles each histogram).

In [10]:

```
cornerplot = cp.corner(chain[:,1:],
                       labels=paramnames, label_kwargs={"fontsize": 20},
                       title_fmt='.1e',
                       show_titles=True, title_kwargs={"fontsize": 12} )

plt.savefig('figures/cornerplot_'+name+'.png')
```



Below we take a look at the correlation length of the chain. I first define an autocorrelation function.

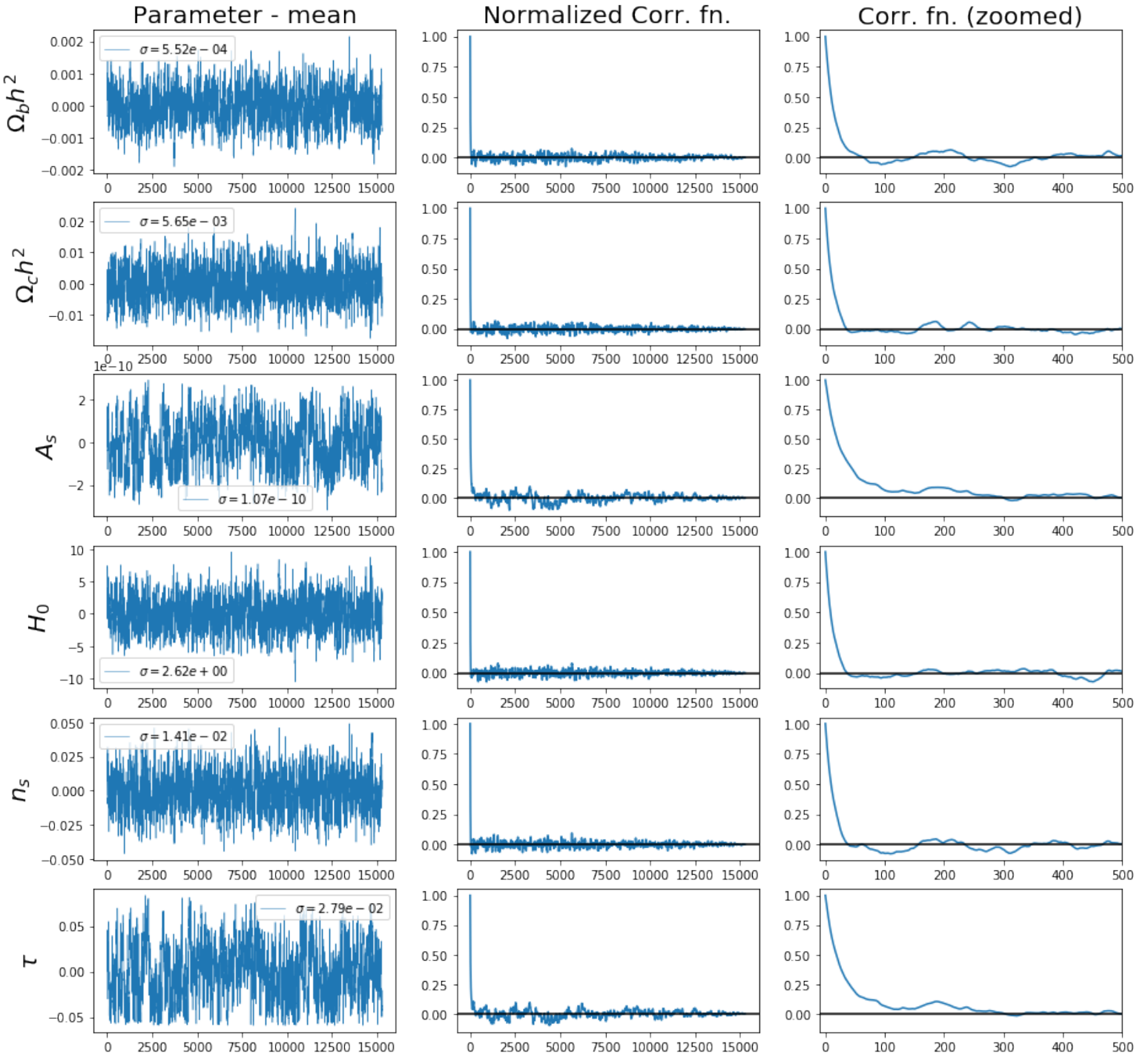
My best chain that did not take correlated steps (run3) had a correlation length of >2500 for all parameters.

Once I took correlated steps (run8) and scaled the steps by 0.5, the correlation length dropped to ~50 for most parameters (a bit longer for $A_s A_s$ and $\tau\tau$).

In [11]:

```
def autocorr(f):
    n = len(f)
    ff = np.append(f,np.zeros(n))
    C = np.zeros(n)
    # I know there's a cleverer way to do this with fft's
    #   but I never got around to it.
    for i in range(n):
        C[i] = np.sum( f*ff[i:n+i] )
    return C/C[0]


plt.figure(figsize=(15,15))

new_step_size=np.zeros(6)

for i in range(6):

    paramchain = chain[:,i+1]
    n=len(paramchain)
    burn = np.int(n/3)
    paramchain = paramchain-np.mean(paramchain[burn:])

    corr = autocorr(paramchain)
    err = np.std(paramchain[burn:])
    new_step_size[i]=err

    plt.subplot(6,3,3*i+1)
    plt.plot(paramchain,lw=0.5,label="$\sigma=%.2e$"%(err))
    plt.ylabel(paramnames[i],fontsize=20)
    plt.legend()

    plt.subplot(6,3,3*i+2)
    plt.plot(corr)
    plt.axhline(0,color='k')

    plt.subplot(6,3,3*i+3)
    plt.plot(corr)
    plt.axhline(0,color='k')
    plt.xlim(-10,500)

plt.subplot(6,3,1);plt.title("Parameter - mean",fontsize=20)
plt.subplot(6,3,2);plt.title("Normalized Corr. fn.",fontsize=20)
plt.subplot(6,3,3);plt.title("Corr. fn. (zoomed)",fontsize=20)
plt.savefig('figures/corr_overview_'+name+'.png')
plt.show()
print(new_step_size)
```

```
/Applications/anaconda/envs/forjupyter3/lib/python3.6/site-packages/
matplotlib/cbook/deprecation.py:107: MatplotlibDeprecationWarning: A
dding an axes using the same arguments as a previous axes currently
reuses the earlier instance.  In a future version, a new instance wi
ll always be created and returned.  Meanwhile, this warning can be s
uppressed, and the future behavior ensured, by passing a unique labe
l to each axes instance.
  warnings.warn(message, mplDeprecation, stacklevel=1)
```




```
[5.51880035e-04 5.65365089e-03 1.07306967e-10 2.61547843e+00
 1.41271229e-02 2.79225565e-02]
```

Best fit parameters:

(Values and uncertainties can be seen in the corner plot)

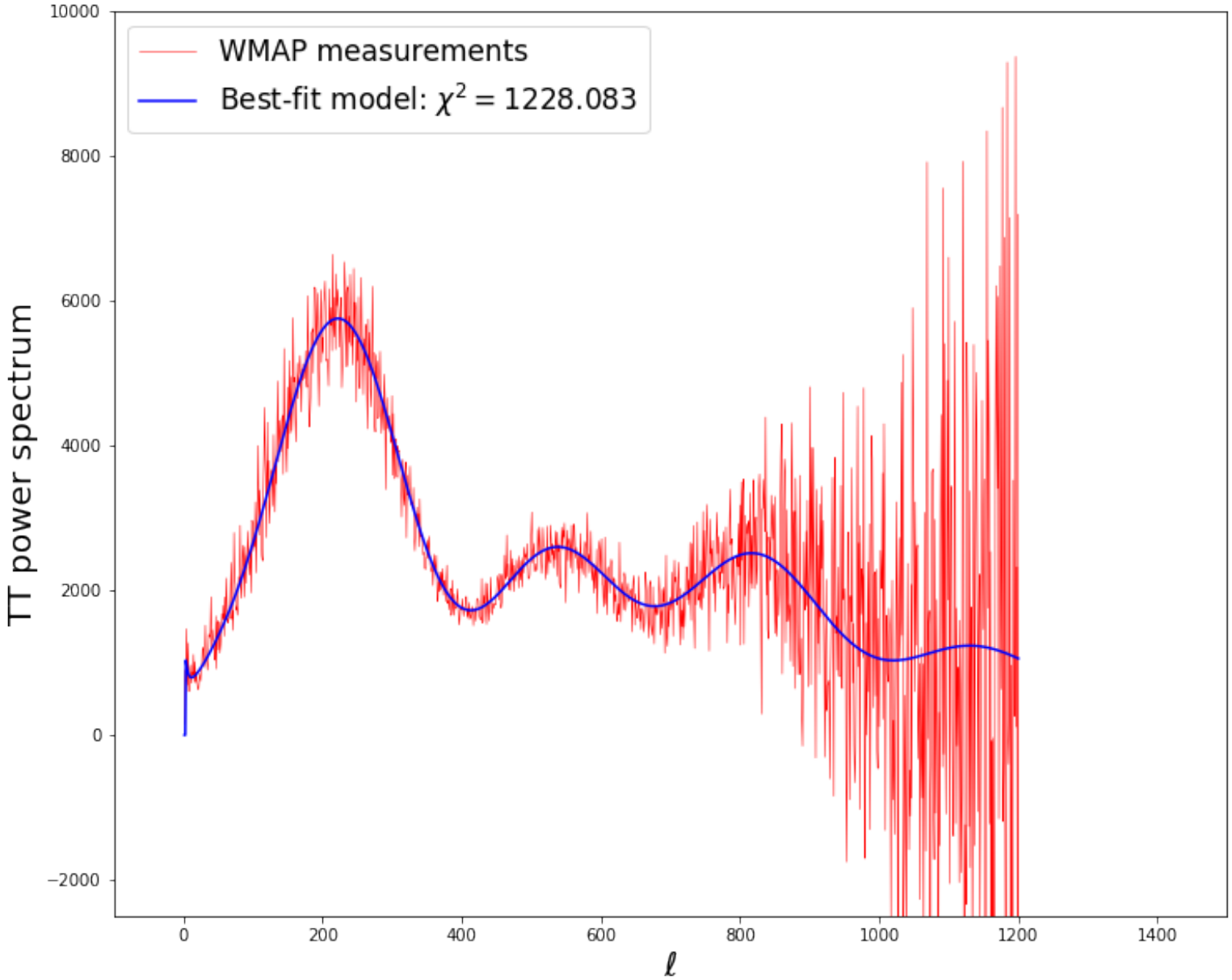
What does this cosmology look like?

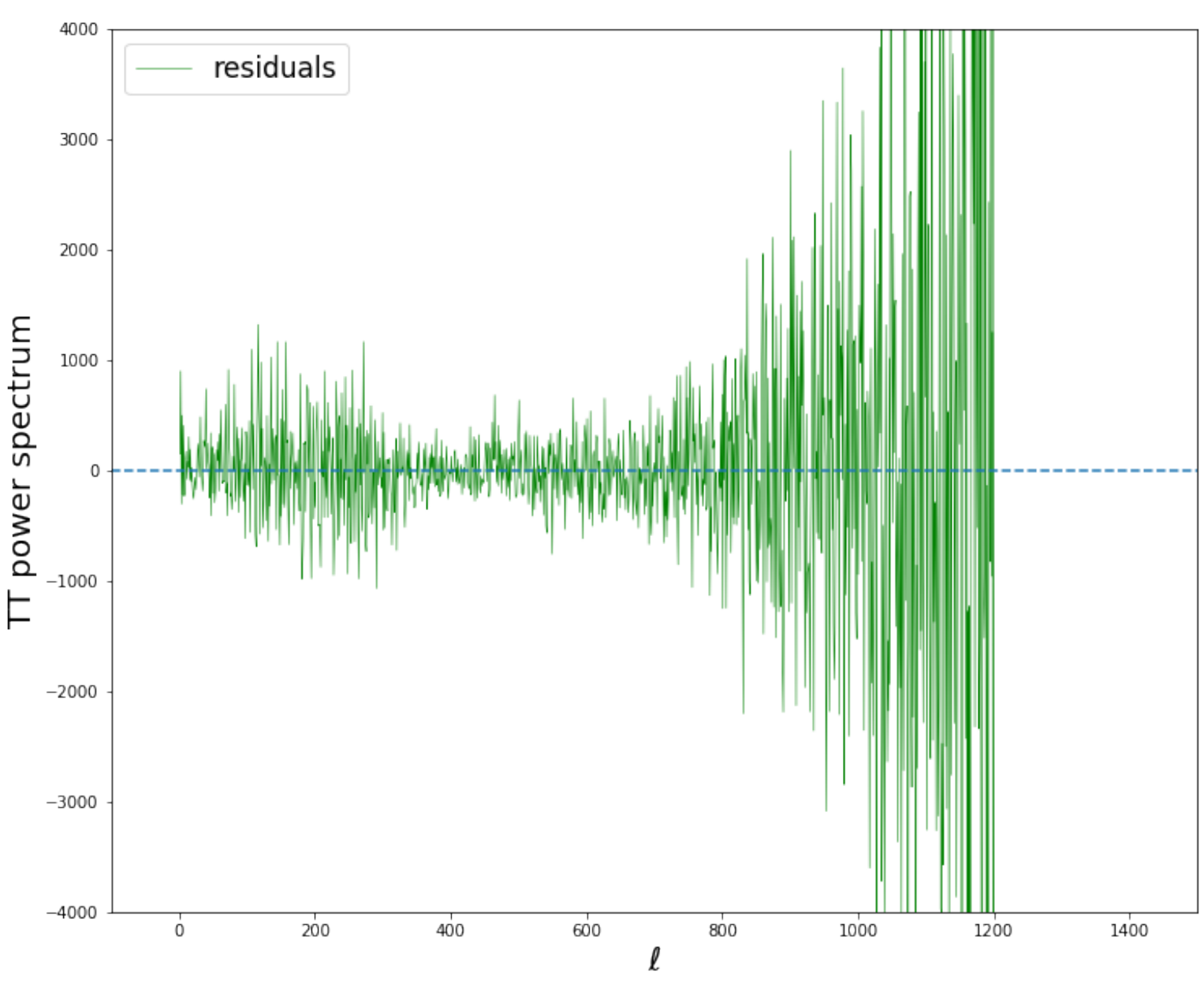
Much better than our original guess! (See plots below)

In [28]:

```
best_cosmo=np.zeros(6)
for i in range(6):
    best_cosmo[i] =  np.mean(chain[:,i+1])

pars=update_model(best_cosmo,pars)
results=camb.get_results(pars)
powspec=results.get_cmb_power_spectra(pars,CMB_unit='muK')['total'][:,0]
powspec=powspec[:len(tt)]
ell_model = np.arange(2,len(powspec)+2)

best_chi2 = get_chi2_from_cosmology(best_cosmo,wmap_data,pars)


plt.figure(figsize=(12,10))
plt.clf()

plt.plot(ell,tt,'r-',lw=0.5,label="WMAP measurements")
plt.plot(ell_model, powspec,'b-',label=r"Best-fit model: $\chi^2=%.3f$"%best_chi
2)

plt.xlabel(r'$\ell$',fontsize=20)
plt.ylabel("TT power spectrum",fontsize=20)
plt.xlim(-100,1500)
plt.ylim(-2500,10000)
plt.legend(fontsize=17)
plt.show()



plt.figure(figsize=(12,10))
plt.clf()

plt.plot(ell,tt-powspec,'g-',lw=0.5,label="residuals")
plt.axhline(0,linestyle='--')

plt.xlabel(r'$\ell$',fontsize=20)
plt.ylabel("TT power spectrum",fontsize=20)
plt.xlim(-100,1500)
plt.ylim(-4000,4000)
plt.legend(fontsize=17)
plt.show()
```

```
1228.0829573755336
```


WMAP measurements
Best-fit model: $\chi^2 = 1228.083$
TT power spectrum
ℓ
10000
8000
6000
4000
2000
0
−2000
0
200
400
600
800
1000
1200
1400



Below here I'm just doing sanity checks on the calculation of the covariance matrix and making sure that the steps it generates are what I intended

To see my answer to part C (i.e. including a point source parameter), look at the *cosmoparams_ptsrc.py* script

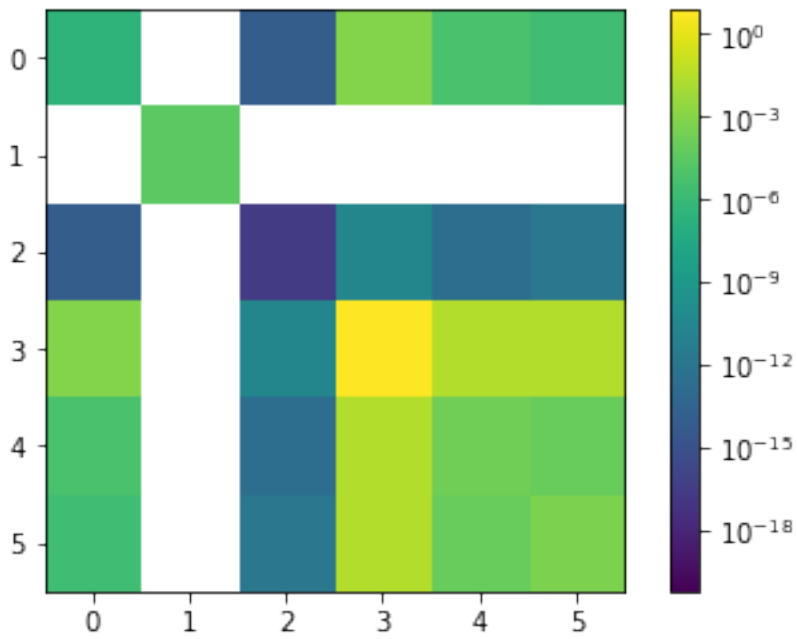
In [5]:

```
# Testing out the Covariance matrix
cov_chain = np.genfromtxt("chain_ALLparam_run3.txt")[2000:15000,1:]

# These two give the same answer: np.cov() and dot(chain.T,chain)/len(chain)  [w
ith mean subtracted]
cov = np.cov(cov_chain.T)
#meanchain=cov_chain-np.mean(cov_chain,axis=0)
#cov2 = np.dot(meanchain.T,meanchain)/len(cov_chain)


print(np.shape(cov))
print(cov)


from matplotlib.colors import LogNorm

plt.imshow(cov,norm=LogNorm())
plt.colorbar()

#divide by sqrt(diagonal) of product of diag elements... should match the np.cor
rcoef result
```

```
(6, 6)
[[ 3.00392694e-07 -9.11803722e-07  9.97925783e-15  8.85030081e-04
   6.75094940e-06  2.40806408e-06]
 [-9.11803722e-07  3.82020778e-05 -3.17265439e-14 -1.52065855e-02
  -4.52857352e-05 -7.09965884e-05]
 [ 9.97925783e-15 -3.17265439e-14  6.15383521e-21  2.82462509e-11
   2.75378797e-13  1.49079487e-12]
 [ 8.85030081e-04 -1.52065855e-02  2.82462509e-11  7.52177191e+00
   2.91787683e-02  2.87604862e-02]
 [ 6.75094940e-06 -4.52857352e-05  2.75378797e-13  2.91787683e-02
   2.10081088e-04  1.01149042e-04]
 [ 2.40806408e-06 -7.09965884e-05  1.49079487e-12  2.87604862e-02
   1.01149042e-04  4.80704038e-04]]
```

Out[5]:

```
<matplotlib.colorbar.Colorbar at 0x13177f8080>
```

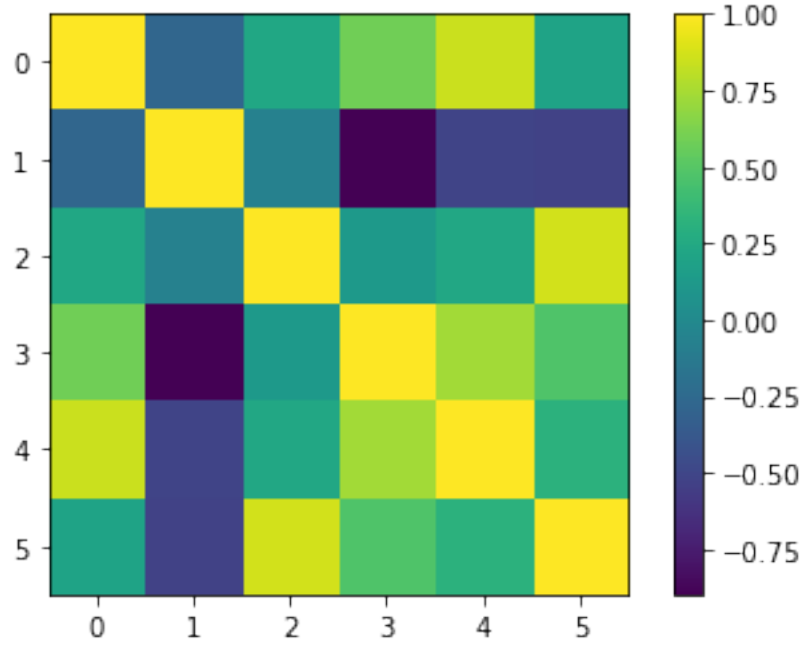


In [166]:

```
cov2=np.corrcoef(cov_chain.T)
plt.imshow(cov2)
plt.colorbar()
```

Out[166]:

```
<matplotlib.colorbar.Colorbar at 0x1320ec9940>
```



In [9]:

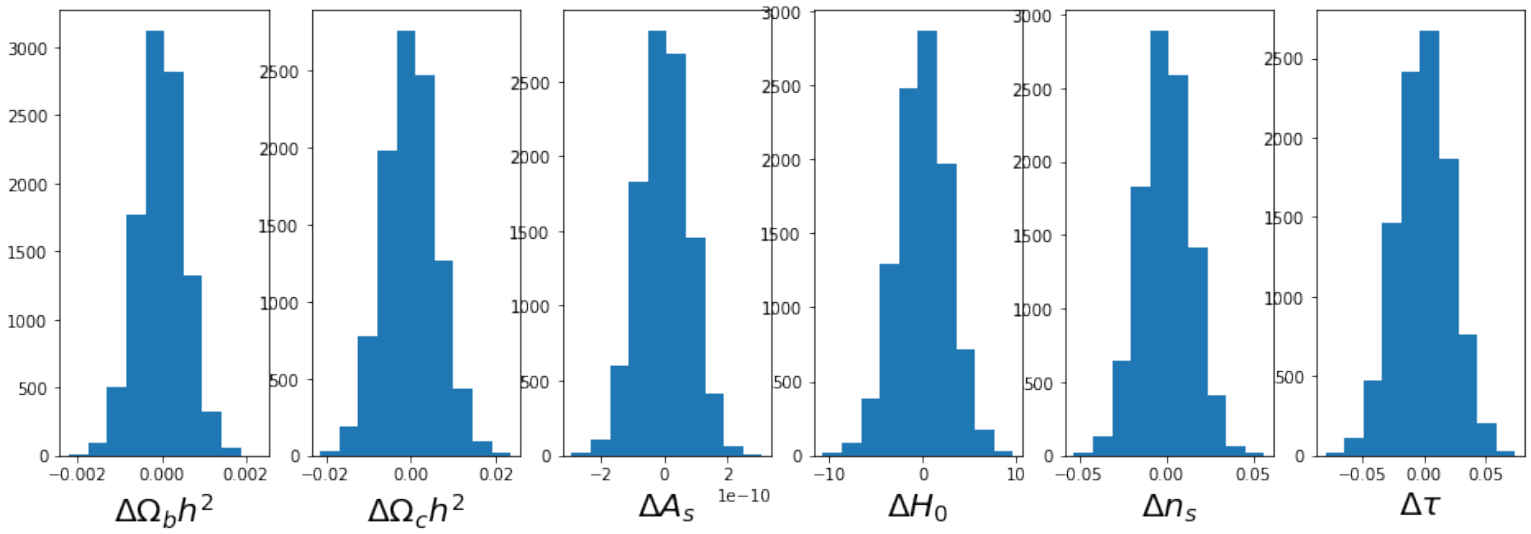
```
# this from the HW2 solutions and is a mirror of the function used in the MCMC t
o take steps
# I'm checking it here to see what the distribution of steps looks like

def check_cov_step(mat,nset=1):
    #if you work through the math, you need
    #to scale gaussian noise by the square root of the eigenvalues
    #then multiply by the eigenvectors.  Note that eigh assumes
    #input matrix is symmetric, and is more stable than eig for our purposes.
    #also, if we want to simulate many sets of data, there's no point finding
    #the eigenvalues/eigenvectors lots of times.  Let nset be the number of simu
lated datasets you want
    e,v=np.linalg.eigh(mat)
    e[e<0]=0 #make sure we don't have any negative eigenvalues due to roundoff
    n=len(e)
    #make gaussian random variables
    g=np.random.randn(n,nset)
    #now scale them by the square root of the eigenvalues
    rte=np.sqrt(e)
    for i in range(nset):
        g[:,i]=g[:,i]*rte
    #and rotate back into the original space
    dat=np.dot(v,g)
    return dat
```

```
nset=10000
#mat=np.ones([n,n])+np.eye(n) #make the noise matrix that is one everywhere but
2 along diagonal
mat= cov

dat=check_cov_step(mat,nset=nset)

mat_sim=np.dot(dat,dat.transpose())/nset
print('RMS error is ',np.std(mat-mat_sim))

print(dat)

# every instance of 'dat' is a random draw of steps for the 6 parameters
```

```
RMS error is  0.007222824263919798
[[ 2.38599074e-04 -4.20149737e-04 -7.79409758e-04 ...  9.58234857e-0
4
  -3.79015912e-04 -9.76420260e-05]
 [ 4.69698645e-03 -1.00856413e-02 -7.81111822e-03 ... -9.98351345e-0
3
  -1.03922712e-03  1.44504883e-03]
 [-7.17560548e-11 -9.84443012e-11 -7.45188942e-11 ...  9.22820799e-1
1
  -3.98463486e-11  1.08880388e-10]
 [-9.72731494e-01  3.00065356e+00  1.72343641e+00 ...  4.40726741e+0
0
   4.57412504e-02 -1.36219991e+00]
 [ 6.81930476e-03 -9.69396325e-03 -9.76287505e-03 ...  2.28690428e-0
2
  -1.64323743e-02  1.99703910e-03]
 [-2.87822829e-02 -2.10956193e-03 -4.50530667e-04 ...  3.66323000e-0
2
  -3.88211341e-03  2.40439292e-02]]
```

In [10]:

```
# Lets see what the steps look like

step_names=[r"$\Delta \Omega_b h^2$",r"$\Delta \Omega_c h^2$", "$\Delta A_s$", "
$\Delta H_0$", "$\Delta n_s$", r"$\Delta \tau$"]

plt.figure(figsize=(16,5))
for i in range(6):
    plt.subplot(1,6,i+1)
    plt.hist(dat[i,:])
    plt.xlabel(step_names[i],fontsize=20)
```

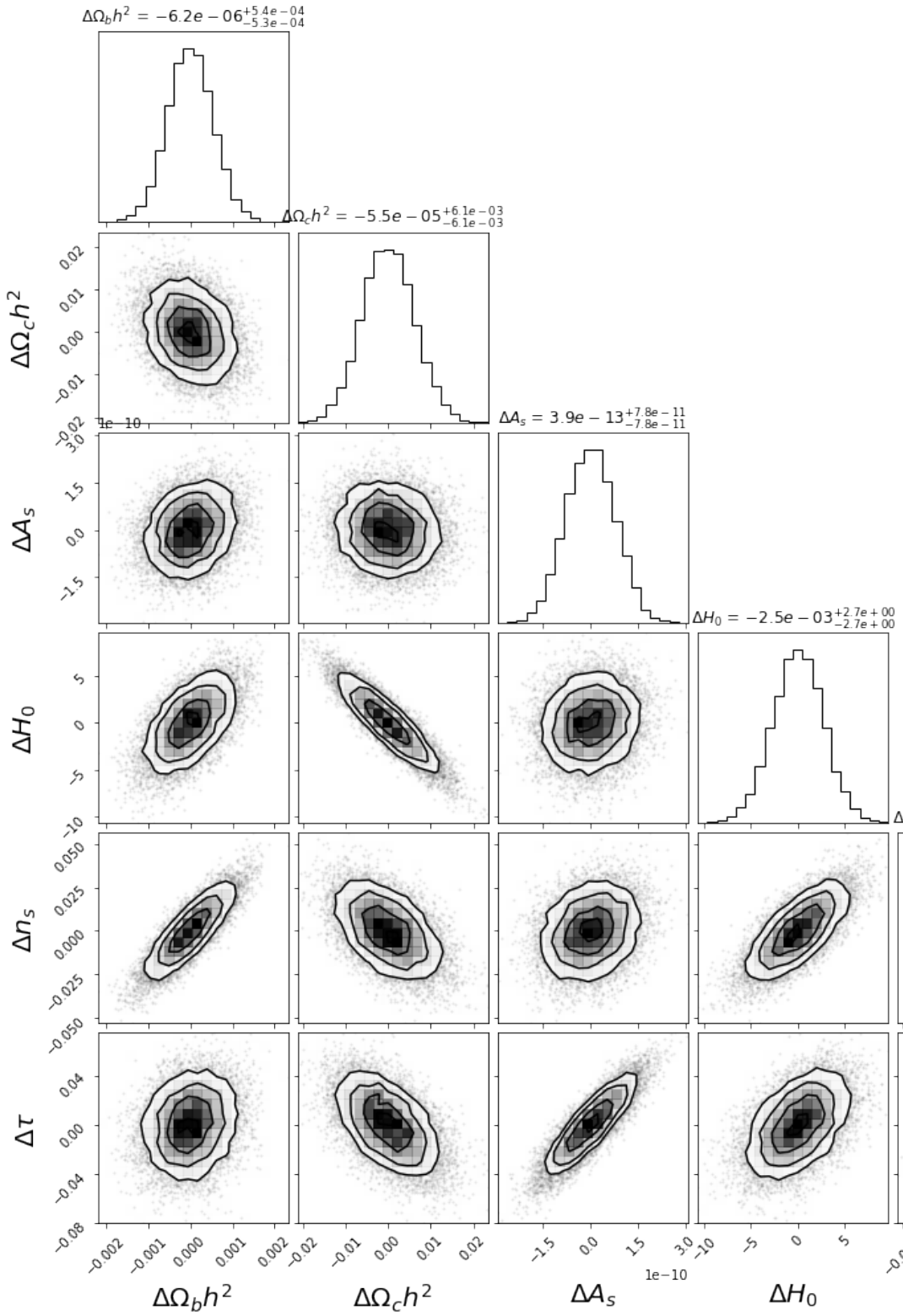


As far as I can tell, this is creating steps with the same kind of correlation as the run3 chain parameters.

See corner plot of steps below:

In [11]:

```
cornerplot = cp.corner(dat.T,
                       labels=step_names, label_kwargs={"fontsize": 20},
                       title_fmt='.1e',
                       show_titles=True, title_kwargs={"fontsize": 12} )

plt.savefig("figures/step_covariance.png")
```



In [12]:

```
# recover the typical uncertainty size for each param
cov_steps=np.zeros(6)
for i in range(6): # must be done once per variable
    q_16, q_50, q_84 = cp.quantile(dat.T[:,i], [0.16, 0.5, 0.84]) # your x is q_
50
    dx_down, dx_up = q_50-q_16, q_84-q_50
    cov_steps[i]=np.mean((dx_up,dx_down))

print(cov_steps)
```

```
[5.35189556e-04 6.07396323e-03 7.77680946e-11 2.70355105e+00
 1.43138085e-02 2.21118306e-02]
```

In [ ]: