In [1]:

```
import numpy as np
import matplotlib.pyplot as plt
import camb
import time
```

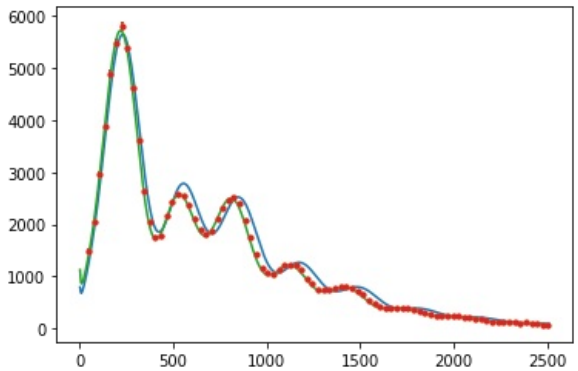
In [2]:

```
def get_spectrum(pars,lmax=3000):
    #print('pars are ',pars)
    H0=pars[0]
    ombh2=pars[1]
    omch2=pars[2]
    tau=pars[3]
    As=pars[4]
    ns=pars[5]
    pars=camb.CAMBparams()
    pars.set_cosmology(H0=H0,ombh2=ombh2,omch2=omch2,mnu=0.06,omk=0,tau=tau)
    pars.InitPower.set_params(As=As,ns=ns,r=0)
    pars.set_for_lmax(lmax,lens_potential_accuracy=0)
    results=camb.get_results(pars)
    powers=results.get_cmb_power_spectra(pars,CMB_unit='muK')
    cmb=powers['total']
    tt=cmb[:,0]    #you could return the full power spectrum here if you wanted to do say EE

    return tt[2:]
```

In [3]:

```
# use 2 sets of parameters:
pars_set = [np.asarray([60,0.02,0.1,0.05,2.00e-9,1.0]), np.asarray([69,0.022,0.12,0.06,2.1e-9,0.95])]

# plot both together
plt.figure()

# save chi squared
chisqs = np.empty(2)

# loop through and print chi^2 for both
for i, pars in enumerate(pars_set):
    #pars=np.asarray([60,0.02,0.1,0.05,2.00e-9,1.0])
    planck=np.loadtxt('COM_PowerSpect_CMB-TT-full_R3.01.txt',skiprows=1)
    ell=planck[:,0]
    spec=planck[:,1]
    errs=0.5*(planck[:,2]+planck[:,3]);
    model=get_spectrum(pars)
    model=model[:len(spec)]
    resid=spec-model
    chisq=np.sum( (resid/errs)**2)
    chisqs[i]=chisq
    print("chisq is ",chisq," for ",len(resid)-len(pars)," degrees of freedom.")
    #read in a binned version of the Planck PS for plotting purposes
    planck_binned=np.loadtxt('COM_PowerSpect_CMB-TT-binned_R3.01.txt',skiprows=1)
    errs_binned=0.5*(planck_binned[:,2]+planck_binned[:,3]);
    plt.plot(ell,model)
    plt.errorbar(planck_binned[:,0],planck_binned[:,1],errs_binned,fmt='.')
```

```
chisq is  15264.64618098867  for  2501  degrees of freedom.
chisq is  3272.572592433381  for  2501  degrees of freedom.
```



Problem 1

Based on the χ^2 value, are the parameters dialed into the test script an acceptable fit? What do you get for χ^2 for parameters [69, 0.022, 0.12, 0.06, 2.1e-9, 0.95], which are closer to their currently accepted values? Would you consider these values an acceptable fit? Note - the mean and variance of χ^2 are n and $2n$ where n is the number of degrees of freedom.

We want $\chi^2 = n$ for the best fit, where $\chi^2 < n$ suggests an overfit and $\chi^2 > n$ suggests an underfit.

In [4]:

```
# compare ideal chi^2 (number of degrees of freedom) to results:
dof = len(resid)-len(pars)

# get percent differences
pdiffs = (chisqs-dof)/dof*100
for i in range(2):
    print('set {} of parameters is {}% off from ideal chi squared'.format(i+1,pdiffs[i]))
```

```
set 1 of parameters is 510.34171055532465% off from ideal chi squared
set 2 of parameters is 30.85056347194646% off from ideal chi squared
```

The first set is 500% off from the ideal, so I would not consider this an acceptable set of parameters. The second is 30% off, which while not ideal is considerably better and I would consider it acceptable, though somewhat of an underfit.

Problem 2

Use Newton's method or LM to find the best-fit parameters and their errors, using numerical derivatives. Report in planck_fit_params.txt. Write your own fitter/numerical-derivative-taker. Keep track of the curvature matrix at the best-fit values for the next problem.

Newton's method and the numerical derivative taker use my code from the previous assignment. The function to get the parameter errors is an updated version of my function from the previous assignment. The main differences are the function inputs and I use the errors from the planck data file to make N^{-1} to find the parameters and their errors.

In [5]:

```
def num_deriv(fun,x,p,dp):
    """
    Parameters:
    fun = input function of the form form fun(parameters)
    x = x-values
    p = parameters
    dp = step size of parameters

    Returns:
    y = fun(x)
    derivs = numerical derivatives with respect to each parameter
    """

    # Get function for x-values
    y = fun(p)

    # Initialize matrix of derivatives
    derivs = np.empty([len(x), len(p)])

    # Loop through parameters and
    for i in range(len(p)):

        # Compute numerical derivatives using:
        # d(fun)/dp = (1/2dp)*(fun(p+dp) - fun(p-dp))
        pp, pm = p.copy(), p.copy()
        pp[i] = p[i] + dp[i]
        pm[i] = p[i] - dp[i]

        f_right = fun(pp)[:len(x)]
        f_left = fun(pm)[:len(x)]

        # Fill columns with derivatives
        derivs[:,i] = (f_right-f_left)/(2*dp[i])

    return y, derivs

def newton(fun, x, d, guess_params, dp, errs, niter):
    """
    Parameters:
    fun = = input function of the form form fun(parameters)
    x = x-values
    d = data
    guess_params = starting guess for parameters
```

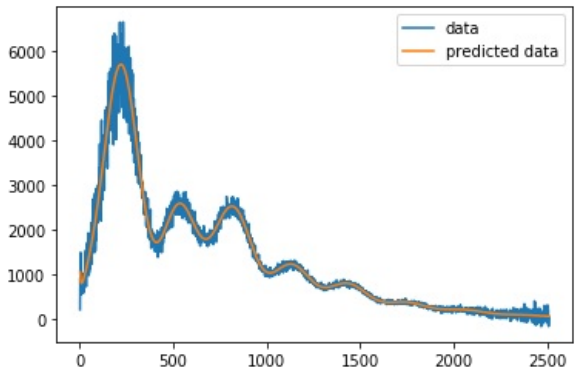
```
    dp = step size of parameters

    errs = data errors
    niter = number of Newton's method steps

    Returns:
    p = best fit parameters
    p_errors = error on parameters
    cov = covariance matrix
    """

    # Go through Newton 5 times
    niter = 5

    # Initialize fit parameters
    p=guess_params.copy()
    for j in range(niter):

        # Get the prediction and numerical derivatives
        pred, grad = num_deriv(fun,x,p,dp)

        # Spectrum and results from get_spectrum should be the same length
        pred = pred[:len(d)]

        # Get residuals, put them in a matrix
        r = d-pred
        rmat = np.matrix(r).T

        # Make an inverse noise matrix
        Ninv = np.diag(1/errs**2)

        # Newton (which includes Ninv)
        lhs = grad.T@(Ninv@grad)
        rhs = grad.T@(Ninv@rmat)
        dp = np.linalg.inv(lhs)*(rhs)
        for jj in range(p.size):
            p[jj]=p[jj]+dp[jj]

    # Get the errors on the fit parameters
    p_errors, cov = get_perr(p,pred,d,grad,errs)

    # Print parameters with their errors
    for i in range(len(p)):
        print('fitted p[{}] = {}, with error = {}'.format(i, p[i], p_errors[i]))

    # Save parameters, errors
    np.savetxt('planck_fit_params.txt', np.asarray([p, p_errors]))

    return p, p_errors, cov

def get_perr(fitp,pred,d,grad,errs):
    """
    Parameters:
    fitp = best fit parameters
    pred = predicted data
    d = data
    grad = gradient of model, evaluated at fit parameters
    errs = data errors

    Returns:
    perr = error on fit parameters
    cov = covariance matrix"""

    # Rename the gradient
    A = grad

    # Make an inverse noise matrix
    Ninv = np.diag(1/errs**2)

    # Get the covariance matrix
    lhs=A.T@(Ninv@A) #/sigma**2
    cov=np.linalg.inv(lhs)
    perr = np.sqrt(np.diag(cov))

    return perr, cov
```

Get the best fit parameters with Newton and print them, with their errors.

In [6]:

```
# Use Newton to get the best fit parameters

# set some initial things
guess_params = np.asarray([69,0.022,0.12,0.06,2.1e-9,0.95])
fun = get_spectrum

# take 10% of the parameters for the Newton step
dp = 0.01*guess_params

# load data
planck=np.loadtxt('COM_PowerSpect_CMB-TT-full_R3.01.txt',skiprows=1)
ell=planck[:,0]
spec=planck[:,1]
errs=0.5*(planck[:,2]+planck[:,3])

# name data and x-values
x = ell
d = spec

# start Newton
t1 = time.time()
p, perr, curve = newton(fun, x, d, guess_params, dp, errs, niter=5)
t2 = time.time()
print('took {} minutes to do newton'.format((t2-t1)/60))

# plot the results
plt.figure()
plt.plot(ell, planck[:,1], label='data')
plt.plot(ell, get_spectrum(p)[:len(ell)], label='predicted data')
plt.legend()
plt.savefig('problem2_newton.png')
```

```
fitted p[0] = 68.24100461012601, with error = 1.1887801931628588
fitted p[1] = 0.02235672513956849, with error = 0.0002300537231115929
fitted p[2] = 0.11765098434790795, with error = 0.0026572451014653226
fitted p[3] = 0.08507415696758514, with error = 0.03424381807375773
fitted p[4] = 2.2176902201165065e-09, with error = 1.435598324963632e-10
fitted p[5] = 0.9730694901435594, with error = 0.006587519305426668
took 4.852361567815145 minutes to do newton
```



Problem 3

Estimate the parameter values and uncertainties using an MCMC sampler you write yourself. Draw trial steps from the curvature matrix. Save the chain (including χ^2 for each sample in the first column) in planckchain.txt. Explain why you think chains are converged. What is your estimate on the mean value of the dark energy $\Omega{\lambda}anditsuncertainty?Weassumedaspatiallyflatuniverseso\Omega_b + \Omegac + \Omega{\lambda} = 1 . Rememberchainisreporting\Omega_{b/c}h^2$.

I will use roughly the same MCMC code that I used in my last assignment - but I fixed my mistake so that I generate a new trial step for each MCMC step.

In [7]:

```
def calc_chisq(d,pred,errs):
    """
    Parameters:
    d = data
    pred = predicted data
    whoops you don't belong here pred_bestfit = predicted data from original best fit parameters

    Returns:
    chisq = chi squared
    """

    # Make the inverse noise matrix
    ninv = 1/errs**2
```

```
    # Get residuals
    r = (d-pred)

    # Get chi^2
    chisq = np.sum(r**2*(ninv))

    return chisq

def run_chain(modelfun,pars,data,curve,errs,scale=np.ones(len(pars)),chifun=calc_chisq,nstep=20000,tau_prior=Fals
e,tau=[None]):
    """
    Parameters:
    modelfun = function to model data to
    pars = starting guess parameters
    data = data
    curve = curvature or covariance matrix
    errs = data errors
    scale = scaling on covariance matrix to get trial steps
    chifun = function to calculate chi squared
    nstep = number of steps in the chain
    tau_prior = adjust function if we have a prior for the tau parameter
    tau = tau prior, of the form [tau, tau error]

    Returns:
    chain = chain parameters
    chisq = chain chi squared
    accept_rate = MCMC acceptance rate
    """

    # Initialize chain (parameters), chi squared
    npar = len(pars)
    chain = np.zeros([nstep,npar])
    chisq = np.zeros(nstep)
    chain[0,:] = pars

    # First predicted data from parameters
    pred = fun(pars)
    pred = pred[:len(data)]

    # First chi squared
    chi_cur = chifun(data, pred, errs)
    chisq[0] = chi_cur

    # Append the number of accepted steps
    accept_steps = []

    # Loop through nsteps
    for i in range(1,nstep):

        # Get a new trial step each time
        trial_step1 = scale*np.random.multivariate_normal(len(curve)*[0],curve)

        # Step the parameters
        pp = pars + trial_step1

        # Option for a tau prior - don't go outside tau error range
        if tau_prior:
            tau_min = tau[0] - tau[1]
            tau_max = tau[0] + tau[1]
            trial_step1[3] = 0.0

            # Keep the parameters if tau is within the range we want
            if pp[3] >= tau_min and pp[3] <= tau_max:
                pp = pp

            # If tau is outside the range we want, try another trial step
            else:
                while pp[3] <= tau_min or pp[3] >= tau_max:
                    # Generate new trial step
                    trial_step1 = scale*np.random.multivariate_normal(len(curve)*[0],curve)
                    # Step the parameters
                    pp = pars + trial_step1


        # Get new predicted data
        new_pred = modelfun(pp)
        new_pred = new_pred[:len(data)]

        # Compute chi squared
        new_chisq=chifun(data, new_pred, errs)

        # Compute probability of accepting the step
```

```
        accept_prob=np.exp(-0.5*(new_chisq-chi_cur))


        # Accept or reject the step in parameter space
        if np.random.rand(1)<accept_prob:
            accept_steps.append(1)
            pars = pp
            chi_cur = new_chisq

        # Append new parameters and chi squared (or same, if rejected)
        chain[i,:] = pars
        chisq[i] = chi_cur

    # Acceptance rate
    accept_rate = np.sum(accept_steps)/nstep
    print('acceptance rate = ', accept_rate)

    return chain, chisq, accept_rate
```

Run MCMC for 10,000 steps

In [71]:

```
# run MCMC for 10,000 steps

# using parameters from Newton as the starting guess
pguess_new = p

# scale steps by unity
scale_step = np.ones(6)

# run mcmc and print the acceptance rate
t1 = time.time()
chain, chisq, accept_rate = run_chain(get_spectrum,pguess_new,d,curve,errs,scale=scale_step,chifun=calc_chisq,nst
ep=10000)
t2 = time.time()
print('took {} hours to do MCMC'.format((t1-t2)/(60*60))
```

```
acceptance rate =  0.1997

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-71-01024a270619> in <module>
     11 chain, chisq, accept_rate = run_chain(get_spectrum, pguess_new, d, curve, scale=scale_step,
chifun=calc_chisq, nstep=10000,T=1)
     12 t2 = time.time
---> 13 print('took {} minutes to do MCMC'.format((t2-t1)/60))

TypeError: unsupported operand type(s) for -: 'builtin_function_or_method' and 'builtin_function_or_
method'
```

(This ran successfully, I just messed up the time.time() call but I don't want to run this cell again). The acceptance rate is ~20%, which is close to the 25% we want for a converged chain.

Save the chains and chi squareds (I'm commenting these out for now so I don't overwrite anything). Then plot the chains (subtracting the mean). They should look like white noise when they're converged.

np.savetxt('planck_chain.txt', chain) np.savetxt('planck_chisq.txt', chisq)

In [8]:

```
# load the saved chains, chi squareds
chain = np.loadtxt('planck_chain.txt')
chisq = np.loadtxt('planck_chisq.txt')
```

In [9]:

```
# plot chains
fig, axs = plt.subplots(2,3,figsize=(12,6),tight_layout=True)
axs = axs.ravel()
for i in range(chain.shape[1]):
    axs[i].plot(chain[:,i] - np.mean(chain[:,i]))
    axs[i].set_title('parameter p[{}]'.format(i))
fig.savefig('problem3_chains.png')

# plot the power spectra
fig, axs = plt.subplots(2,3,figsize=(12,6),tight_layout=True)
axs = axs.ravel()
for i in range(chain.shape[1]):
    pspec = np.abs(np.fft.rfft(chain[:,i]))**2
    axs[i].loglog(pspec)
    axs[i].set_title('parameter p[{}]'.format(i))
fig.savefig('problem3_chains_spectra.png')
```

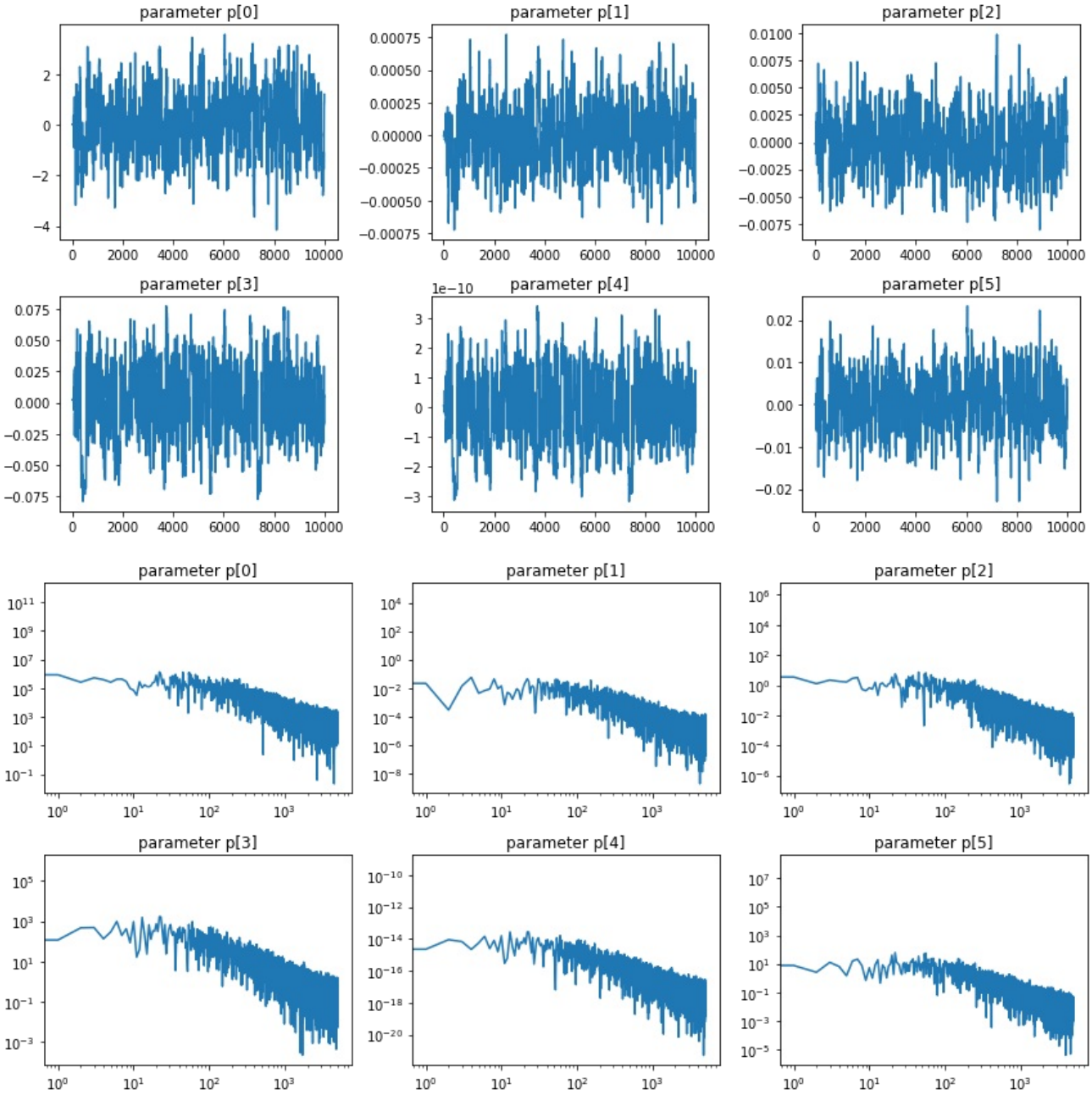


Fig: The chains look like white noise. Their power spectra look flat along the x-axis until various knees, meaning there is a white noise portion of the chains. This white noise supports the chain's convergence.

The mean value of the dark energy is:

$$\Omega_\Lambda = 1 - (\Omega_b + \Omega_c) = 1 - \frac{100}{H_0}(p[1] + p[2]).$$

Propagating the uncertainty gives:

$$\sigma_{\Omega_\Lambda} = \frac{100}{H_0}\sqrt{\sigma^2_{p[1]} + \sigma^2_{p[2]} + \frac{p[1]^2}{H_0}\sigma^2_{H_0} + \frac{p[2]^2}{H_0}\sigma^2_{H_0}}.$$

In [10]:

```
# get the parameters and errors
p_mcmc = np.mean(chain,axis=0)
err_mcmc = np.std(chain,axis=0)

# print results for all parameters
for i in range(6):
    print('p[{}] = {} +/- {}'.format(i, p_mcmc[i], err_mcmc[i]))

# get the dark energy value, remember to scale by planck constant
H0 = p_mcmc[0]
h = H0/100
Ob = p_mcmc[1]/h**2
Oc = p_mcmc[2]/h**2
Ode = 1 - (Ob + Oc)

# rename some of the parameter errors
p1err = err_mcmc[1]
p2err = err_mcmc[2]
H0err = err_mcmc[0]

# propagate the uncertainty to get error on the dark energy value
err_Ode = (100/H0)*np.sqrt(p1err**2 + p2err**2 + (H0err**2/H0)*(p_mcmc[1]**2 + p_mcmc[2]**2))

# print dark energy results
print('mean value of dark energy = {} +/- {}'.format(Ode,err_Ode))
```

```
p[0] = 68.21463458878 +/- 1.1424736011712058
p[1] = 0.022361091160759308 +/- 0.00022501965675380102
p[2] = 0.11779393646314654 +/- 0.0025621353606026832
p[3] = 0.08267441889537765 +/- 0.030324949015656372
p[4] = 2.2115406980030477e-09 +/- 1.2690223444465436e-10
p[5] = 0.973007019618461 +/- 0.0065071144071300584
mean value of dark energy = 0.6988009711588477 +/- 0.024603717192960495
```

Problem 4

Polarization data give a better constraint on reionization, with optical depth $\tau = 0.0540 \pm 0.0074$. Run a new chain where you include this constraint and save to "planck_chain_taupriot.txt". Compare those results to what you get from importance sampling the chain from Problem 3. Re-estimate the parameter covariance matrix (possibly via importance sampling) before running the new chain.

I've done this problem in 2 ways. I think the first way was not what we were supposed to do, however, the chains look like they've converged. I just set a range of τ's and if the MCMC steps put τ outside that range then the step isn't accepted. I use a while loop to generate a new random step until it's within the acceptable range.

For the second method, I don't set an acceptable τ range but I multiply $\delta\chi^2$ by the likelihood ratio for the τ parameter only (I add $exp(-\frac{1}{2}\frac{(\tau_{chain}-0.0540)^2}{0.0074^2})$ to the acceptance probability). The chains don't look converged to me (the power spectra aren't flat at low k's).

For both methods, I compute the new covariance matrix using the new τ parameter and my function "get_perr".

For both methods, I do the importance sampling by weighting each of the parameters by $exp(-\frac{1}{2}\delta\chi^2)$ where $\delta\chi^2$ is for τ only to get the average.

In [12]:

```
# constraint
tau, tauerr = 0.0540, 0.0074

# parameter guess from before but with updated tau
pguess_priortau = p
pguess_priortau[3] = tau

# new prediction and gradient for this tau
dp = 0.01*pguess_priortau
y, grad = num_deriv(get_spectrum,x,pguess_priortau,dp)

# new covariance matrix and parameter errors
perr, curve = get_perr(pguess_priortau,y,d,grad,errs)

# wait i think this actually doesn't matter bcs I just need CURVE
# ah... but I still need to use this number... somewhere...
# set parameter error from tau manually
perr[3] = tauerr
```

In [70]:

```
# MCMC for tau prior - full chain (10,000)

# Scale steps by unity
scale_step = np.ones(6) # got acceptance ~6%

# run mcmc and print the acceptance rate
t1 = time.time()
chain_tau, chisq_tau, accept_rate_tau = run_chain(get_spectrum,pguess_priortau,d,curve,errs,scale=scale_step,
                                    chifun=calc_chisq,nstep=10000,tau_prior=True, tau=[0.0540, 0.0074])
t2 = time.time()
print('took {} hours to do MCMC'.format((t2-t1)/(60*60)))
```

```
acceptance rate =  0.3287
took 12.800641700294282 hours to do MCMC
```

Save the chains, chi squared.

load the saved chains, chi squareds np.savetxt('planck_chain_tauprior.txt', chain_tau) np.savetxt('planck_chisq_tauprior.txt', chisq_tau)

In [13]:

```
# load the saved chains, chi squareds
chain_tau = np.loadtxt('planck_chain_tauprior.txt')
chisq_tau = np.loadtxt('planck_chisq_tauprior.txt')
```

In [14]:

```
# plot the rest of the chains
fig, axs = plt.subplots(2,3,figsize=(12,6),tight_layout=True)
axs = axs.ravel()
for i in range(chain_tau.shape[1]):
    axs[i].plot(chain_tau[:,i] - np.mean(chain_tau[:,i]))
    axs[i].set_title('parameter p[{}]'.format(i))
fig.savefig('problem4_tauprior_chains.png')

# plot the rest of the chains' power spectra
fig, axs = plt.subplots(2,3,figsize=(12,6),tight_layout=True)
axs = axs.ravel()
for i in range(chain_tau.shape[1]):
    pspec = np.abs(np.fft.rfft(chain_tau[:,i]))**2
    axs[i].loglog(pspec)
    axs[i].set_title('parameter p[{}]'.format(i))
fig.savefig('problem4_tauprior_chains_pspec.png')
```

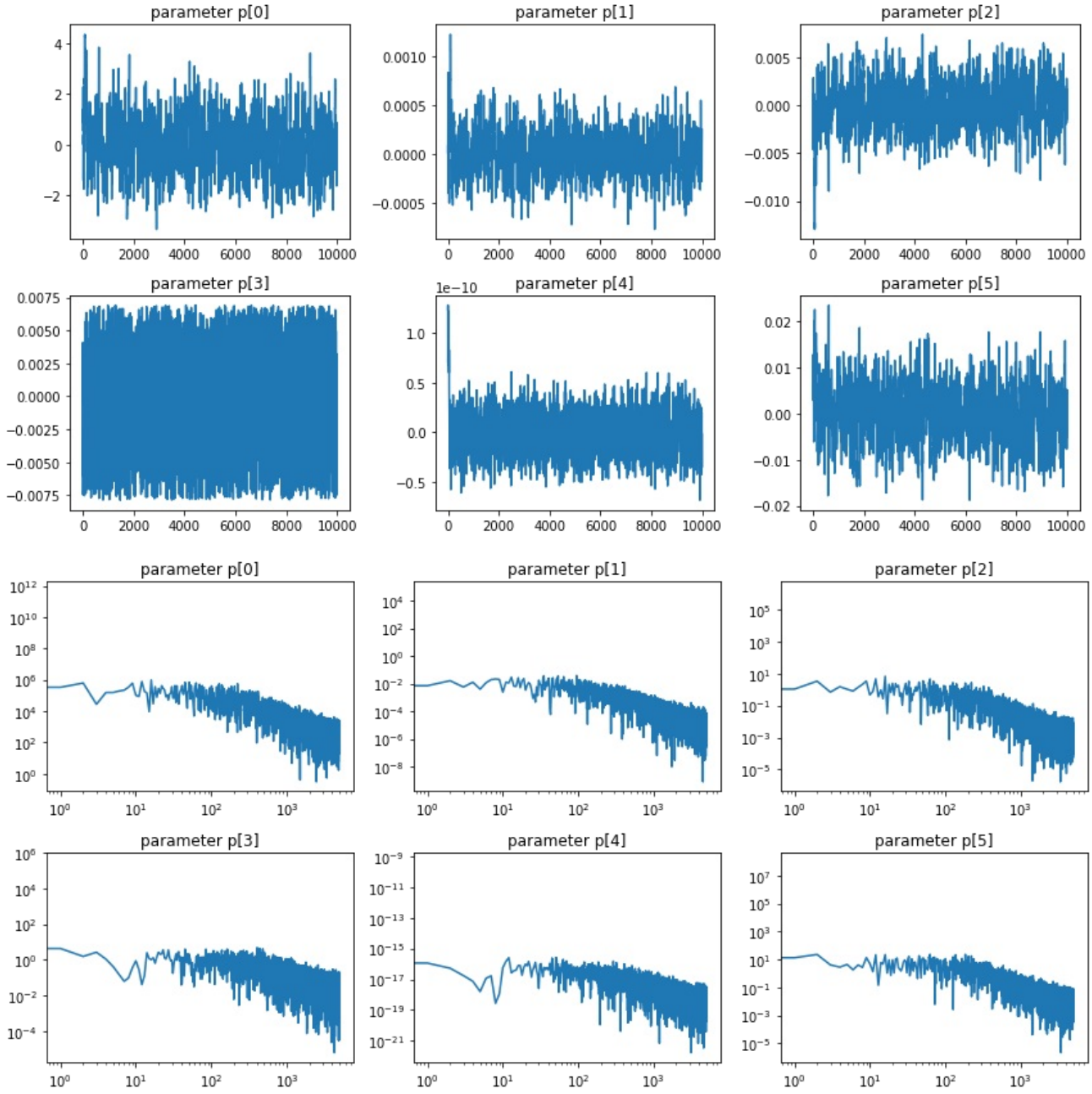


Fig: These look converged.

The importance sampling gives the weighted average of each parameter as the sum over each chain parameter times the weight divided by the sum of the weights: $\frac{\sum p_{chain,i} w_i}{\sum w_i}$. The weights are $\exp(-\frac{\delta\chi^2}{2})$ for the optical depth parameter τ. $\delta\chi^2$ is then the difference between χ^2 and the χ^2 for $\tau = 0.0540 \pm 0.0074$.

In [15]:

```
# delta chi^2 for tau prior
delta_chisq_tau = (chain[:,3] - tau)**2/tauerr**2

# weighted average of parameters for problems 3 and 4
isamp_pars, isamp_pars_tau = np.empty(6), np.empty(6)

# loop through parameters and weight them
for i in range(6):

    # the weights are exp(-0.5*delta chi^2)
    #weights = delta_chisq_tau
    weights = np.exp(-0.5*delta_chisq_tau)

    # parameters from problem 3
    pars = chain[:,i]

    # parameters from problem 4
    pars_tau = chain_tau[:,i]

    # weight both
    avg_pars = np.sum(pars*weights)/np.sum(weights)
    avg_pars_tau = np.sum(pars_tau*weights/np.sum(weights))

    # save em
    isamp_pars[i] = avg_pars
    isamp_pars_tau[i] = avg_pars_tau

    # take error = standard deviation?
    errs = np.std(pars)
    errs_tau = np.std(pars_tau)

    # print results from first chain
    print('(no prior) weighted average p[{}] = {} +/- {}'.format(i, avg_pars, errs))

    # print results from second chain
    print('(prior) weighted average p[{}] = {} +/- {}'.format(i, avg_pars_tau, errs_tau))
```

```
(no prior) weighted average p[0] = 67.68352529552803 +/- 1.1424736011712047
(prior) weighted average p[0] = 67.72710671615766 +/- 1.0634591023468924
(no prior) weighted average p[1] = 0.022305115831573536 +/- 0.00022501965675380113
(prior) weighted average p[1] = 0.022294714394807673 +/- 0.00022434779382999318
(no prior) weighted average p[2] = 0.11895182155885554 +/- 0.0025621353606026832
(prior) weighted average p[2] = 0.11888641081848736 +/- 0.0024470310417736424
(no prior) weighted average p[3] = 0.055803636692468316 +/- 0.030324949015656563
(prior) weighted average p[3] = 0.05422738458026921 +/- 0.0042385996168326735
(no prior) weighted average p[4] = 2.099392933700957e-09 +/- 1.2690223444465368e-10
(prior) weighted average p[4] = 2.092237767652276e-09 +/- 2.1949130405659904e-11
(no prior) weighted average p[5] = 0.9698138962106301 +/- 0.006507114407130041
(prior) weighted average p[5] = 0.9701944284395112 +/- 0.0059185199000608875
```

These are very similar, but the importance sampling with τ prior has smaller error bars for τ and A_s (p[3] and p[4]).

Problem 4 again

Second method of constraining τ.

In [16]:

```
# code to run a mcmc
def run_chain_v2(modelfun,pars,data,curve,errs,scale=np.ones(len(pars)),chifun=calc_chisq,nstep=20000,tau_prior=F
alse,Tau=[None]):
    """
    Parameters:
    modelfun = function to model data to
    pars = starting guess parameters
    data = data
    curve = curvature or covariance matrix
    errs = data errors
    scale = scaling on covariance matrix to get trial steps
    chifun = function to calculate chi squared
    nstep = number of steps in the chain
    tau_prior = adjust function if we have a prior for the tau parameter
    Tau = tau prior, of the form [tau, tau error]

    Returns:
    chain = chain parameters
    chisq = chain chi squared
    accept_rate = MCMC acceptance rate
    """
```

```
    # Initialize chain (parameters), chi squared
    npar=len(pars)
    chain=np.zeros([nstep,npar])
    chisq=np.zeros(nstep)
    chain[0,:]=pars

    # First predicted data from parameters
    pred = fun(pars)
    pred = pred[:len(data)]

    # First chi squared
    chi_cur=chifun(data, pred, errs)
    chisq[0]=chi_cur

    # Append the number of accepted steps
    accept_steps = []

    # Loop through nsteps
    for i in range(1,nstep):

        # Get a new trial step each time
        trial_step1 = scale*np.random.multivariate_normal(len(curve)*[0],curve)

        # Step the parameters
        pp = pars + trial_step1

        # Get new predicted data
        new_pred = modelfun(pp)
        new_pred = new_pred[:len(data)]

        # Compute chi squared
        new_chisq=chifun(data, new_pred, errs)

        # Compute probability of accepting the step
        accept_prob=np.exp(-0.5*(new_chisq-chi_cur))

        # Constrain tau using chi^2
        if tau_prior:

            # delta chi^2 for tau prior
            tau, tauerr = Tau[0], Tau[1]
            delta_chisq_tau = (pp[3] - tau)**2/tauerr**2

            # add the delta chi^2 for tau prior
            accept_prob += np.exp(-0.5*delta_chisq_tau)

        # Accept or reject the step in parameter space
        if np.random.rand(1)<accept_prob:
            accept_steps.append(1)
            pars=pp
            chi_cur=new_chisq

        # Append new parameters and chi squared (or same, if rejected)
        chain[i,:]=pars
        chisq[i]=chi_cur

    # Acceptance rate
    accept_rate = np.sum(accept_steps)/nstep
    print('acceptance rate = ', accept_rate)

    return chain,chisq,accept_rate
```

Set initial parameters, guess, get the covariance matrix again.

In [19]:

```
# constraint
tau, tauerr = 0.0540, 0.0074

# parameter guess from before but with updated tau
pguess_priortau = p
pguess_priortau[3] = tau

# new prediction and gradient for this tau
dp = 0.01*pguess_priortau
y, grad = num_deriv(get_spectrum,x,pguess_priortau,dp)

# new covariance matrix and parameter errors
errs=0.5*(planck[:,2]+planck[:,3])
perr, curve = get_perr(pguess_priortau,y,d,grad,errs)

# wait i think this actually doesn't matter bcs I just need CURVE
# ah... but I still need to use this number... somewhere...
# set parameter error from tau manually
perr[3] = tauerr
```

In [24]:

```
# MCMC for tau prior - full chain (5,000 this time)

# Scale steps by unity
scale_step = np.ones(6) # got acceptance ~6%

# run mcmc and print the acceptance rate
t1 = time.time()
chain_tau, chisq_tau, accept_rate_tau = run_chain_v2(get_spectrum,pguess_priortau,d,curve,errs,scale=scale_step,
                                       chifun=calc_chisq,nstep=5000,tau_prior=True, Tau=[0.0540, 0.0074])
t2 = time.time()
print('took {} hours to do MCMC'.format((t2-t1)/(60*60)))
```

```
acceptance rate =  0.3418
took 5.776441129843394 hours to do MCMC
```

The acceptance rate should be slightly lower, but I think this isn't too unreasonable.

load the saved chains, chi squareds np.savetxt('planck_chain_taupriorv2.txt', chain_tau) np.savetxt('planck_chisq_taupriorv2.txt', chisq_tau)

In [20]:

```
# load the saved chains, chi squareds
chain_tauv2 = np.loadtxt('planck_chain_taupriorv2.txt')
chisq_tauv2 = np.loadtxt('planck_chisq_taupriorv2.txt')
```

Plot the chains and their power spectra.

In [27]:

```
# plot the rest of the chains
fig, axs = plt.subplots(2,3,figsize=(12,6),tight_layout=True)
axs = axs.ravel()
for i in range(chain_tauv2.shape[1]):
    axs[i].plot(chain_tauv2[:,i] - np.mean(chain_tauv2[:,i]))
    axs[i].set_title('parameter p[{}]'.format(i))
fig.savefig('problem4_taupriorv2_chains.png')

# plot the rest of the chains' power spectra
fig, axs = plt.subplots(2,3,figsize=(12,6),tight_layout=True)
axs = axs.ravel()
for i in range(chain_tauv2.shape[1]):
    pspec = np.abs(np.fft.rfft(chain_tauv2[:,i]))**2
    axs[i].loglog(pspec)
    axs[i].set_title('parameter p[{}]'.format(i))
fig.savefig('problem4_taupriorv2_chains_pspec.png')
```

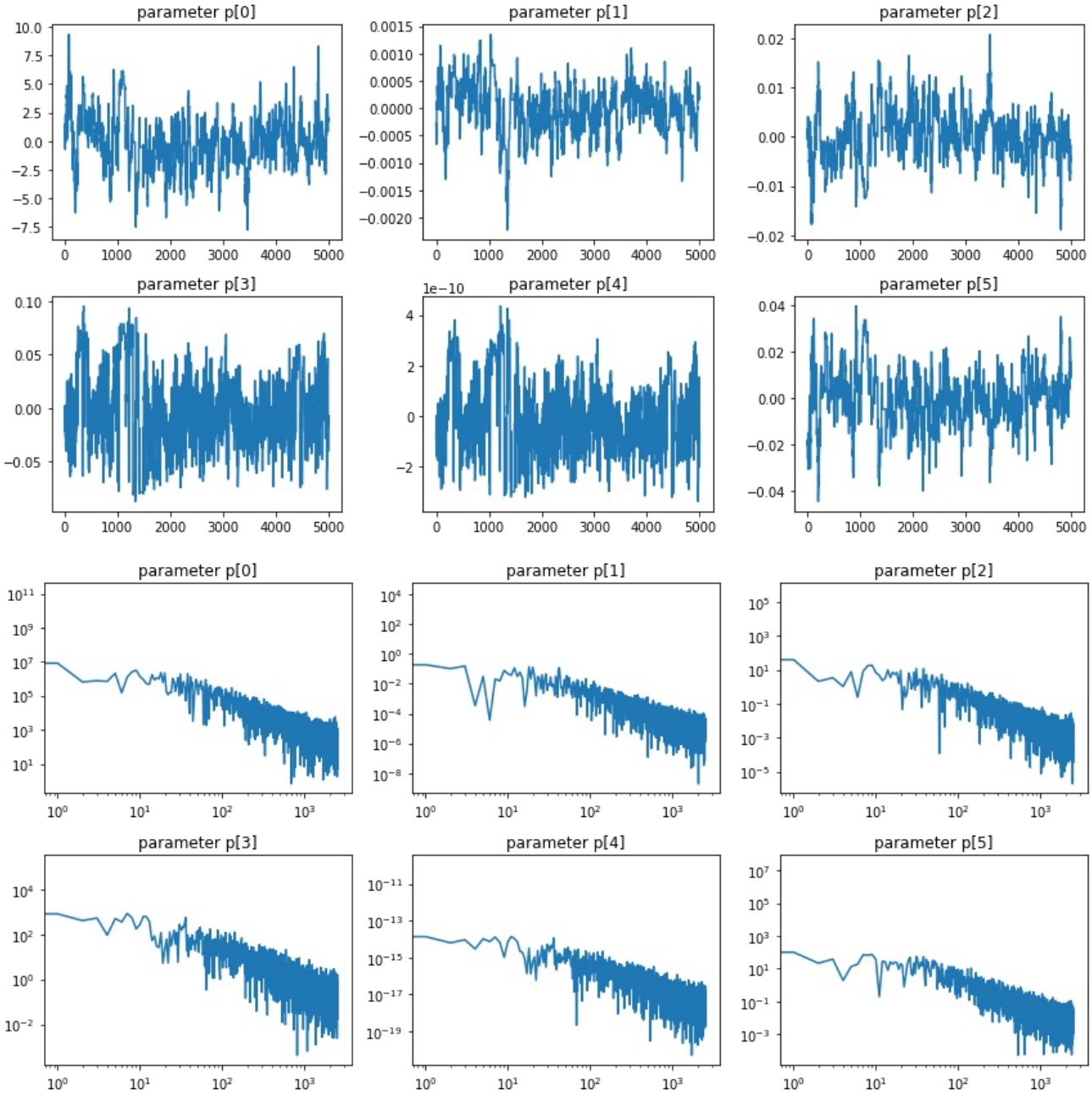


Fig: It's clear from the slope in the power spectra at low k that these are not converged.

Do the importance sampling (same code as above, but for the second method of constraining τ).

In [26]:

```
# delta chi^2 for tau prior
delta_chisq_tau = (chain_tauv2[:,3] - tau)**2/tauerr**2

# weighted average of parameters for problems 3 and 4
isamp_pars, isamp_pars_tau = np.empty(6), np.empty(6)

# loop through parameters and weight them
for i in range(6):

    # the weights are exp(-0.5*delta chi^2)
    #weights = delta_chisq_tau
    weights = np.exp(-0.5*delta_chisq_tau)

    # parameters from problem 4
    pars_tau_v2 = chain_tauv2[:,i]

    # weight the parameters
    avg_pars_tau_v2 = np.sum(pars_tau_v2*weights/np.sum(weights))

    # save them
    isamp_pars_tau[i] = avg_pars_tau_v2

    # take error = standard deviation?
    errs_tau = np.std(pars_tau_v2)

    # print results from second chain
    print('(prior) weighted average p[{}] = {} +/- {}'.format(i, avg_pars_tau_v2, errs_tau))
```

```
(prior) weighted average p[0] = 67.55720408200544 +/- 2.3584786704092946
(prior) weighted average p[1] = 0.02224182632980831 +/- 0.00045274547328389743
(prior) weighted average p[2] = 0.11954061752647414 +/- 0.005340887821420898
(prior) weighted average p[3] = 0.05453834092362572 +/- 0.03872645339574951
(prior) weighted average p[4] = 2.1011193101202165e-09 +/- 1.6179579155960012e-10
(prior) weighted average p[5] = 0.9666631643312057 +/- 0.012171483379893067
```

These error bars are bigger than my first method, so I think this method is what I should have done from the start but my chains should be converged.