Assignment 5

1) By running planck_likelihood.py

we get :

chisq is 15267.937150261658 for 2501 degrees of freedom.

A reasonable fit would have

$\text{chisq} \simeq n \pm \sqrt{2n}$

$\simeq 2501 \pm 70.7$

So our initial guess is not very good, because it is too far away from our expected chisq

We can then replace pars with

[69, 0.022, 0.12, 0.06, 2.1e-9, 0.95]

The output then is :

chisq is 3272.2053559202186 for 2501 degrees of freedom.

The newer parameters would give a better fit, but is still not acceptable.

(2)

we could use the code that we wrote in the last hw to run the Newton's method.

Our code already have a numerical derivative taker, and has the Newton's method (and Param errors, and cov matrix) implemented.

We just need to define a function to put into the derivative taker. But we could simply use get_spectrum which is already implemented by Jon. Just slightly modify to cut the output by len(spec) before returning.

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

```
#copied from planck_likelihood.py
def get_spectrum(pars,t,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
    tt= tt[2:]
    tt=tt[:len(t)]
    return tt
```

Use what I have last time:

```
#Take the derivative of a function. The function fun must take
# an array of parameters and a t array as input. param is the parameter values
# we want to take derivative at. param should have the same length
# as the parameter array that fun takes.
def fun_deriv(fun,param,t):
    grad=np.zeros([t.size,param.size])
    n_param = np.size(param)
    dx = param*((1e-16)**(1/3))
    for i in range(n_param):
        param_new = np.copy(param)
        #calculate the two-sided derivative of the ith parameter
        param_new[i] = param[i]-dx[i]
        fun_minus = fun(param_new,t)
        param_new[i] = param[i]+dx[i]
        fun_plus = fun(param_new,t)
        grad[:,i] = (fun_plus - fun_minus)/(2*dx[i])
    return grad


#use numerical derivation to calculate grad matrix
def calc_camb(p,t):
    y=get_spectrum(p,t)
    grad=fun_deriv(get_spectrum,p,t)
    return y,grad
```

Then we can start Newton's

Then we can start Newton's method. However, after some testing, I realize it is better if we use L.M.

Our scheme is that we start with $\Lambda = 1$. if χ^2 increase, if $\Lambda = 0$ we make it 1, or else double it.

If χ^2 decrease, $\Lambda = 0.3\Lambda$ if $\Lambda < 0.1$, $\Lambda \Rightarrow 0$, then we update params.

```
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])

#use part 1 for initial guess
p0=np.asarray([60,0.02,0.1,0.05,2.00e-9,1.0])
# p0=np.asarray( [69, 0.022, 0.12, 0.06, 2.1e-9, 0.95]) #another initial guess

numda = 1 #numda in LM method
cur_chisq = 1e9
p=p0.copy()

for j in range(30):
    pred,grad=calc_camb(p,ell)
    resid=spec-pred
    r=np.matrix(resid).transpose()
    grad=np.matrix(grad)

    #get the chisq now
    chisq = np.sum((resid/errs)**2)

    noise_inverse = np.matrix(np.diag(1/errs**2))
    lhs=grad.transpose()*noise_inverse*grad
    cov = np.linalg.inv(lhs)
    lhs += numda*np.diag(np.diag(grad.transpose()*noise_inverse*grad))
```

```
    lhs = np.matrix(lhs)
    rhs=grad.transpose()*noise_inverse*r
    dp=np.linalg.inv(lhs)*(rhs)

    if chisq > cur_chisq:
        print("LM Increase", numda)
        if numda == 0:
            numda = 1
        else:
            numda *= 2
    else:
        print("LM Decrease", numda)
        if numda < 0.1:
            numda = 0
        else:
            numda *= 0.3
        #update params
        for jj in range(p.size):
            p[jj]=p[jj]+dp[jj]

    print(p,chisq)
```

Then write answers for later uses.

```
residual = spec-pred
par_errs=np.sqrt(np.diag(np.linalg.inv(cov)))
print(par_errs)
f = open("planck_fit_params.txt", "a")
f.write(f"The best fit params are: {p}")
f.write(f"The erros are: {par_errs}")
f.write('\n')
f.close()

chisq=np.sum( (residual/errs)**2)
print("chisq is ",chisq," for ",len(spec)-len(p0)," degrees of freedom.")
np.save('cov_matrix',cov)
np.save('best_fit_pars',p)
```

The output of the code is:

```
LM Decrease 1
[6.25733972e+01 1.86266571e-02 1.05167549e-01 5.56114699e-02
 1.97900741e-09 9.85145933e-01] 15267.937150261658
LM Decrease 0.3
[6.55247486e+01 1.93508608e-02 1.10669253e-01 5.08693467e-02
 1.99957673e-09 9.90559047e-01] 5109.783080101091
LM Decrease 0.09
[6.76480289e+01 2.12369152e-02 1.14848023e-01 4.57637741e-02
 2.02161899e-09 9.80501340e-01] 3138.2975253784734
LM Decrease 0
[6.82733886e+01 2.23474230e-02 1.17394617e-01 1.03243580e-01
```

2.28030837e-09 9.72050281e-01] 2637.437286584939
LM Decrease 0
[6.81212976e+01 2.23492854e-02 1.17935472e-01 8.01990761e-02
2.19639198e-09 9.72353336e-01] 2612.9698091354394
LM Decrease 0
[6.82681629e+01 2.23674332e-02 1.17610437e-01 8.65386738e-02
2.22374377e-09 9.73184004e-01] 2576.4021153645213
LM Decrease 0
[6.81655063e+01 2.23523451e-02 1.17834685e-01 8.36010628e-02
2.21212034e-09 9.72666272e-01] 2576.1585482934697
LM Decrease 0
[6.82487103e+01 2.23646852e-02 1.17653600e-01 8.55280386e-02
2.21965561e-09 9.73076469e-01] 2576.1565846290196
LM Decrease 0
[6.82719253e+01 2.23687414e-02 1.17604201e-01 8.54883346e-02
2.21925127e-09 9.73182778e-01] 2576.1527860792903
LM Decrease 0
[6.81787373e+01 2.23565072e-02 1.17813693e-01 8.39672860e-02
2.21366152e-09 9.72715507e-01] 2576.1540503483484
LM Decrease 0
[6.82500039e+01 2.23651162e-02 1.17651471e-01 8.54920878e-02
2.21950073e-09 9.73086718e-01] 2576.1548430697057
LM Decrease 0
[6.79397233e+01 2.23239899e-02 1.18329789e-01 8.03704114e-02
2.20041982e-09 9.71553214e-01] 2576.1526985914247
LM Decrease 0
[6.82595782e+01 2.23655297e-02 1.17624950e-01 8.64880834e-02
2.22361776e-09 9.73148741e-01] 2576.2153487514925
LM Decrease 0
[6.82571195e+01 2.23630168e-02 1.17626122e-01 8.51068467e-02
2.21764932e-09 9.73137032e-01] 2576.157039121791
LM Decrease 0
[6.82479452e+01 2.23664031e-02 1.17660319e-01 8.51852222e-02
2.21821286e-09 9.73059036e-01] 2576.15334698371
LM Decrease 0
[6.82290108e+01 2.23626376e-02 1.17699258e-01 8.48881318e-02
2.21709737e-09 9.72974739e-01] 2576.1524432032593
LM Decrease 0
[6.82391443e+01 2.23634554e-02 1.17674871e-01 8.51154598e-02
2.21796632e-09 9.73030162e-01] 2576.1523752068288
LM Decrease 0
[6.82339399e+01 2.23624347e-02 1.17685485e-01 8.49892480e-02
2.21746360e-09 9.73006203e-01] 2576.152252853109
LM Decrease 0
[6.82443561e+01 2.23643526e-02 1.17663902e-01 8.51802011e-02
2.21819616e-09 9.73054046e-01] 2576.1522720560365
LM Decrease 0
[6.82461256e+01 2.23643118e-02 1.17659159e-01 8.51684848e-02
2.21811824e-09 9.73064222e-01] 2576.1522806750654
LM Decrease 0
[6.82338194e+01 2.23616151e-02 1.17683675e-01 8.49731206e-02
2.21737819e-09 9.73010921e-01] 2576.1522959331032
LM Decrease 0

[6.82380925e+01 2.23634603e-02 1.17677810e-01 8.50826600e-02
 2.21783889e-09 9.73023577e-01] 2576.1523244678583
LM Decrease 0
[6.82364747e+01 2.23636468e-02 1.17682545e-01 8.50332451e-02
 2.21764855e-09 9.73012129e-01] 2576.152247448422
LM Decrease 0
[6.82361771e+01 2.23630927e-02 1.17681720e-01 8.50403434e-02
 2.21767199e-09 9.73014807e-01] 2576.1522623103556
LM Decrease 0
[6.82390669e+01 2.23633631e-02 1.17674780e-01 8.50864914e-02
 2.21783795e-09 9.73030082e-01] 2576.152252446932
LM Decrease 0
[6.82362324e+01 2.23630851e-02 1.17681550e-01 8.50299287e-02
 2.21762522e-09 9.73014977e-01] 2576.1522449927575
LM Decrease 0
[6.82446264e+01 2.23641847e-02 1.17662689e-01 8.51777063e-02
 2.21817753e-09 9.73057039e-01] 2576.152248290773
LM Decrease 0
[6.82372853e+01 2.23630984e-02 1.17678756e-01 8.50273715e-02
 2.21759844e-09 9.73020854e-01] 2576.15228126392
LM Decrease 0
[6.82301898e+01 2.23621801e-02 1.17694536e-01 8.49434793e-02
 2.21731082e-09 9.72985657e-01] 2576.1522496814428
LM Decrease 0
[6.82427610e+01 2.23638936e-02 1.17666816e-01 8.51619696e-02
 2.21812938e-09 9.73047991e-01] 2576.152341166303
[1.18270010e+00 2.28465781e-04 2.64513848e-03 3.40380288e-02
 1.42914245e-10 6.53717065e-03]
chisq is 2576.152341166303 for 2501 degrees of freedom.

] final params

] error

we see that chisq has converged, and that it is an acceptable fit.

(3)

we could use our MCMC running in the last assignment for this part.

We simply have to write an get_chisq function, which is already implemented in the first part

is already implemented in
the first part.
we used param fit from last part
as starting position, and uses
covariance matrix to get steps
in mcmc.

However, we could also run
the chain from where we
ended last time.

```
from tracemalloc import start
import numpy as np
import camb
from matplotlib import pyplot as plt
import time

#use the covariance matrix and noise from previous calculation
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])
tt_lenth = len(spec)

# start_pos = np.load("best_fit_pars.npy")
start_pos = np.load("final_chain.npy")[-1,:]
start_pos = start_pos[1:]
cov = np.load('cov_matrix.npy')
nparam = len(start_pos)

#copied from planck_likelihood.py
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
    tt= tt[2:]
    tt=tt[:tt_lenth]
    return tt

#get chisq for the three lorentzian fit
def get_chisq(p):
    model = get_spectrum(p)
    resid=spec-model
    chisq=np.sum( (resid/errs)**2)
    return chisq


#get the step size using the covariance matrix. Scaled by step_size
def get_step(step_size):
    zero_mean = np.zeros(nparam)
    step = np.random.multivariate_normal(zero_mean,cov)*step_size
    return step
```

Then we can run the mcmc chain:

```
#mostly copied from class PPT
def run_mcmc(start_pos,nstep,step_size):
    nparam = start_pos.size
    params=np.zeros([nstep,nparam+1])
    params[0,1:] = start_pos #save initial params
    cur_chisq = get_chisq(start_pos)
    params[0,0] = cur_chisq #save initial chisq
    cur_pos = start_pos.copy()
    for i in range(1,nstep): #loop through nstep
        new_pos=cur_pos+get_step(step_size)
        new_chisq=get_chisq(new_pos)
        if new_chisq<cur_chisq:
            accept=True
        else:
            delt=new_chisq-cur_chisq
            prob=np.exp(-0.5*delt) #accept probability
            if np.random.rand()<prob:
                accept=True
            else:
                accept=False
        if accept:
            cur_pos=new_pos
            cur_chisq=new_chisq
        params[i,0]=cur_chisq #save current chisq
        params[i,1:]=cur_pos #save current params
        if i%100 ==0:
            print("finished step ",i,"/",nstep)
```

```
    return params
#I tried and this scaling converges fastest
step_size = np.array([0.4,0.4,0.4,0.4,0.4,0.4])
#using an initial guess close to best fit params
pars_start = start_pos
nstep=10000
start_time = time.time()
chain=run_mcmc(pars_start,nstep,step_size)
end_time = time.time()
print("time for running is", end_time-start_time)
```

we will also save the results:

```
np.save("chain_cont_2",chain)
old_chain = np.load("final_chain.npy")[:-1,:]
new_chain = np.vstack((old_chain,chain))
# new_chain = chain
np.save("final_chain_2",new_chain)
np.savetxt("planck_chain.txt", new_chain)
```

we can then do data analysis.
First we print parameter averages and std.

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

chain = np.load("final_chain_2.npy")

#print mean and std of params
for i in range(1,7):
    val=np.mean(chain[:,i])
    scat=np.std(chain[:,i])
    print([val,scat])
```

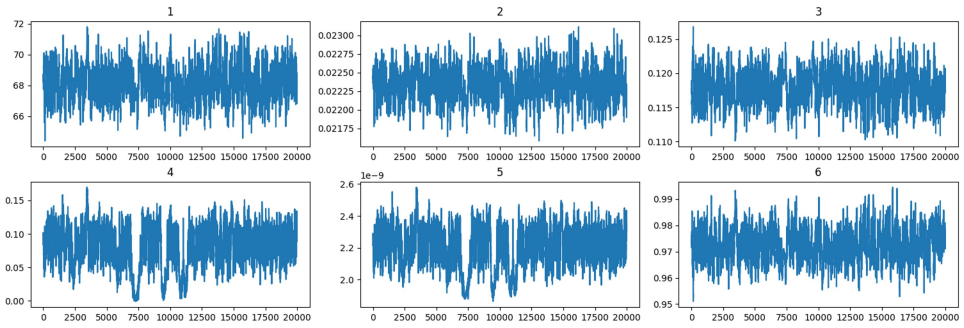
Output is:

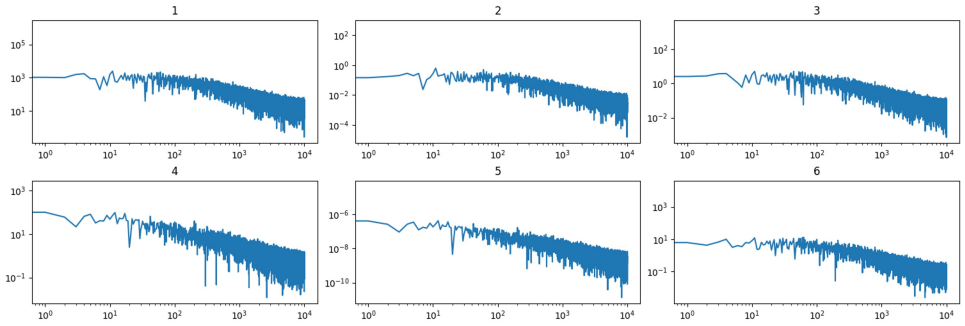
```
[68.1497683564527, 1.1105301535073184]
[0.022337481282359347, 0.00022093802427923073]
[0.11785650807964543, 0.0024921028536286317]
[0.08080604087845542, 0.030768708988365034]
[2.203797206701731e-09, 1.2782211460548948e-10]
[0.9725797685931408, 0.006146334206147058]
```

we can also check the convergence of params:

I learned how to plot these diagrams in a same figure from David Cai.

```
#I learned about this kind of plot from David Cai
fig = plt.figure(figsize = (15,5),constrained_layout = True)
axes = fig.subplots(2,3).flatten()
for i in np.arange(1,7):
    axes[i-1].plot(chain[:, i])
    axes[i-1].set_title(f"{i}")
fig.savefig("param_conv_2.png")
fig = plt.figure(figsize = (15,5),constrained_layout = True)
axes = fig.subplots(2,3).flatten()
for i in np.arange(1,7):
    axes[i-1].loglog(np.abs(np.fft.rfft(chain[:, i])))
    axes[i-1].set_title(f"{i}")
fig.savefig("param_fft_2.png")
```





We see that from the parameter

We see that from the parameter plots that they are mostly converged, because the plot looks like white noise.

From FFTs, we see that the low frequencies are flat, which is another indication of convergence.

To estimate error of Ω_Λ:

$$\Omega_\Lambda = 1 - \Omega_b - \Omega_c$$

$$\Rightarrow \sigma_{\Omega_\Lambda} = \sqrt{\sigma_{\Omega_b}^2 + \sigma_{\Omega_c}^2}$$

$$h = \frac{H_0}{100} \Rightarrow \sigma_h = \frac{\sigma_{H_0}}{100}$$

From error propagation:

$$\sigma_{h^2} = 2\sigma_h^2 \qquad \Omega_b = \frac{\text{Baryon}}{h^2}$$

$$\Rightarrow \sigma_{\Omega_b}^2 = \frac{\text{Baryon}^2}{h^4} \cdot \left[\left(\frac{\sigma_{\text{Baryon}}}{\text{Baryon}} \right)^2 + \left(\frac{\sigma_{h^2}}{h^2} \right)^2 - 2 \frac{\sigma_{h^2,\text{Baryon}}}{\text{Baryon} \cdot h^2} \right]$$

$$= \left[\frac{\sigma_{Baryon}^2}{h^4} + \frac{4\sigma_{h^2,Baryon}^2}{h^8} - 2\frac{\sigma_{h^2,Baryon} \cdot Baryon}{h^6} \right]$$

dmd is Dark matter density

similarly:

$$\sigma_{\Omega_c}^2 = \left[\frac{\sigma_{dmd}^2}{h^4} + \frac{4\sigma_{h^2,dmd}^2}{h^8} - 2\frac{\sigma_{h^2,dmd} \cdot dmd}{h^6} \right]$$

we can plug in these numbers.

```
val = np.zeros(6)
scat = np. zeros(6)
for i in range(1,7):
    val[i-1]=np.mean(chain[:,i])
    scat[i-1]=np.std(chain[:,i])

h = val[0]/100
d_h = scat[0]/100
baryon = val[1]
d_baryon = scat[1]
dmd = val[2]
d_dmd = scat[2]

h_array = chain[:,1]/100
h_sq_array = h_array**2
baryon_array = chain[:,2]
d_h2_baryon = np.cov(h_sq_array,baryon_array)[0,1]
dmd_array = chain[:,3]
d_h2_dmd = np.cov(h_sq_array,dmd_array)[0,1]

omega_b = baryon/h**2
omega_c = dmd/h**2

d_omega_b_sq = d_baryon**2/h**4 + 4*baryon**2*d_h**2/h**8 - 2*d_h2
_baryon*baryon/h**6

d_omega_c_sq = d_dmd**2/h**4 + 4*dmd**2*d_h**2/h**8 - 2*d_h2_dmd*dmd/h**6

omega_numda = 1 - omega_b - omega_c
```

```
d_omega_numda = d_omega_b_sq + d_omega_c_sq

print("Dark Energy is",omega_numda,"+-",np.sqrt(d_omega_numda))
```

Output is:

Dark Energy is 0.6981434345979791 +- 0.016338145048743247

(4)

discussed with David.

we can weight the mcmc chain by the sqrt of inverse of χ^2 of tau.

$$\text{weight} = \sqrt{\frac{\sigma_\tau^2}{\tau_{model} - \tau_{expect}}}$$

Then we can compute means and cov matrix with this weight.

```
#tau constraints
tau = 0.054
d_tau = 0.0074
#compute the weight of the mcmc chain
chi_tau = (old_chain[:,4]-tau)**2/d_tau**2
weight = 1/np.sqrt(chi_tau)
param_chain = old_chain[:,1:]
nparam = 6
imp_samp_pars = np.average(param_chain, axis = 0, weights = weight)
imp_samp_cov = np.cov(param_chain,rowvar=False,aweights=weight)
imp_samp_err = np.sqrt(np.diag(imp_samp_cov))
print()
for i in range(nparam):
    print(imp_samp_pars[i],"+-",imp_samp_err[i])
```

Out put is

67.67964329814392 +- 1.154815480888193
0.022264087197865726 +- 0.0002347735880402698 8
0.11884262869563196 +- 0.00260010110549204
0.05903776205839168 +- 0.015915250048751873
2.112596271045241e-09 +- 6.706648188925196e-11
0.9700056619871616 +- 0.006369818607787841

This is the result we get from importance sampling.

Then we can compare this with rerunning the chain with constraints.

First we need to use the above calculated cov matrix

Secondly, note that now

$$\chi^2_{total} = \chi^2_{comb} + \chi^2_{tau}$$

so in the mcmc runner we need to change it into:

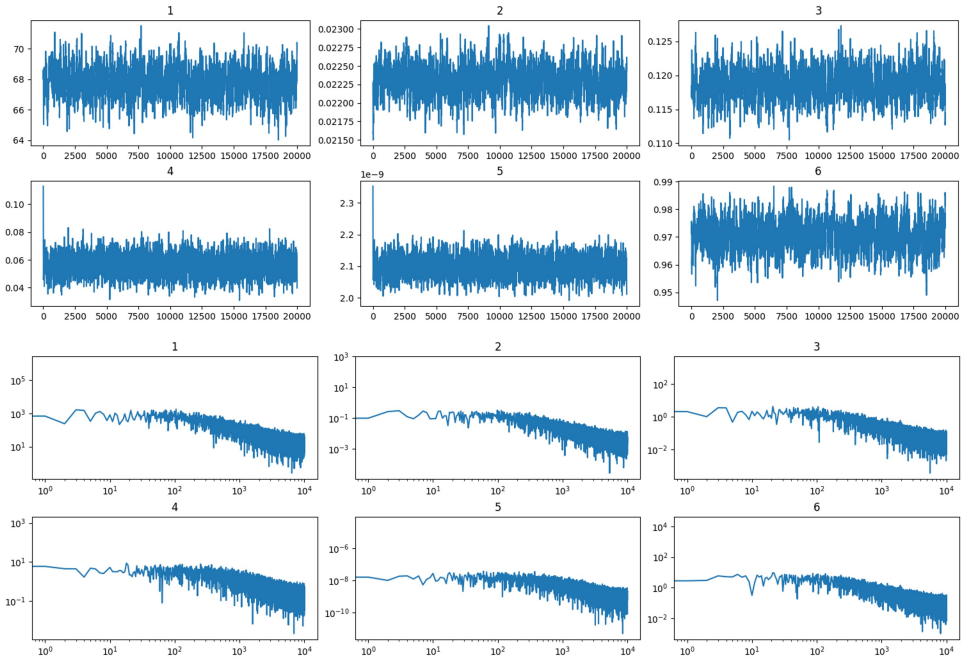
```
#adding chisq for tau into new chisq
        tau_chisq = (new_pos[3]-tau)**2/d_tau**2
        delt_chisq = new_chisq + tau_chisq - cur_chisq

        if delt_chisq <0:
            accept=True
        else:
            prob=np.exp(-0.5*delt_chisq) #accept probability
```

```
        if np.random.rand()<prob:
            accept=True
        else:
            accept=False
    if accept:
        cur_pos=new_pos
        cur_chisq=new_chisq
```

We then analyze the data

Params error

[67.77309578217003, 1.0182544545070138]
[0.022303329641149384, 0.00020379946568435965]
[0.11871556354953461, 0.002317439499480044]
[0.055833403023116526, 0.007230033242722017]
[2.0978241663468824e-09, 3.1294985844544553e-11]
[0.9704281374463423, 0.005463936038311659]



From the param plot and fft plot, we see that the chain is converged.

we see that the chain is converged.

We can compare our param & error from the chain to what we get from importance sampling at the start.

We see that the param values and their errors are very close to each other.

But importance sampling is instant, while re running a chain takes another 4 hours.