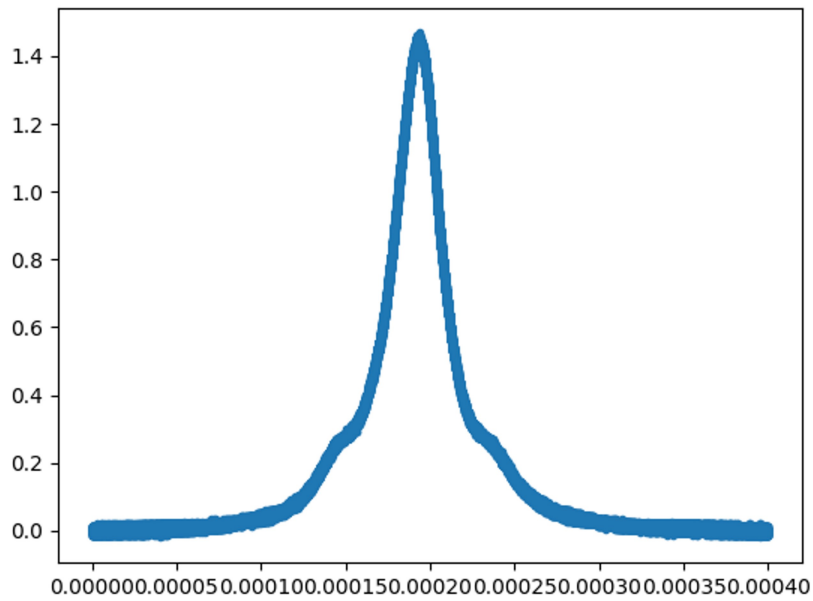
Assignment 4

Thursday, October 13, 2022 9:27 AM

(a)

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

#copied from newton.py in class repo. changed to be same as problem param
def calc_lorentz(p,t):
    y=p[0]/(1.0+((t-p[2])**2)/(p[1]**2))
    grad=np.zeros([t.size,p.size])
    #now differentiate w.r.t. all the parameters
    grad[:,0]=1.0/(1.0+((t-p[2])**2)/(p[1]**2))
    grad[:,1]=(2*p[0]*p[1]*((t-p[2])**2))/(((t-p[2])**2+p[1]**2)**2)
    grad[:,2]=(2*p[0]*p[1]**2*(t-p[2]))/(((t-p[2])**2+p[1]**2)**2)
    return y,grad

stuff=np.load('sidebands.npz')
t=stuff['time']
x=stuff['signal']

plt.clf()
plt.plot(t,x,'.')
plt.savefig('raw_data.png')
plt.show()
```



From the raw data graph we can estimate the inputting parameters

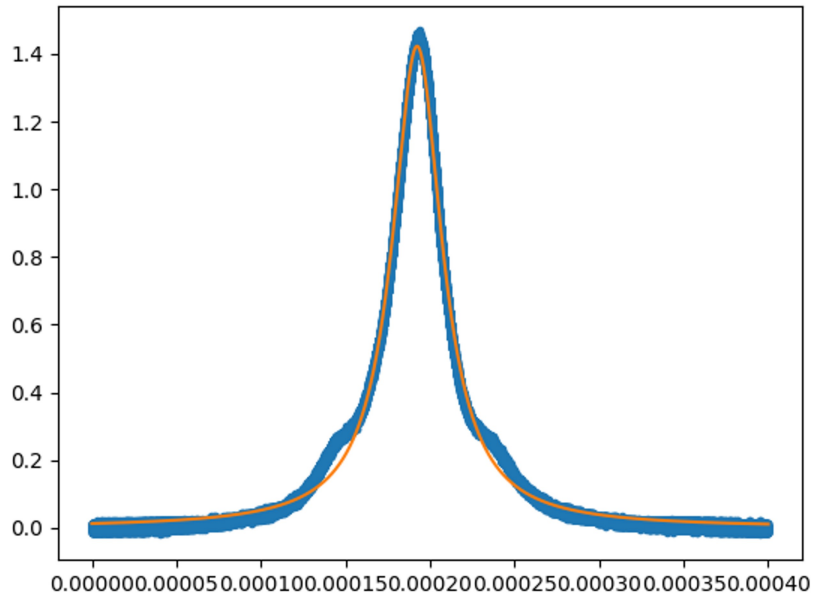
From the raw data graph we can estimate the lorentzian parameters as an initial guess.

I guess that $a \sim 1.4$, $w \sim 1e\text{-}4$ to $\sim 2e\text{-}4$

```
#from the raw data graph guess initial values
p0=np.array([1.4,0.0001,0.0002]) #starting guess, close but not exact

#copied from newton.py
p=p0.copy()
for j in range(8):
    pred,grad=calc_lorentz(p,t)
    r=x-pred
    err=(r**2).sum()
    r=np.matrix(r).transpose()
    grad=np.matrix(grad)

    lhs=grad.transpose()*grad
    rhs=grad.transpose()*r
    dp=np.linalg.inv(lhs)*(rhs)
    for jj in range(p.size):
        p[jj]=p[jj]+dp[jj]
    print(p,err)

plt.clf()
plt.plot(t,x,'.')
plt.plot(t,pred)
plt.savefig('lorentz_fit.png')
plt.show()
```

Output is:

```
[7.67854126e-01 4.88906673e-05 1.92452510e-04] 40887.001248174354
[1.11953102e+00 3.68896456e-06 1.91249789e-04] 2976.0580171036113
[8.11966718e-01 1.39080200e-05 1.91716013e-04] 8460.532481612092
[1.40547777e+00 2.06385805e-05 1.92933480e-04] 3703.038874325205
[1.41075840e+00 1.80445528e-05 1.92217837e-04] 194.09875543606978
[1.42212998e+00 1.79391465e-05 1.92360156e-04] 64.62209859758832
[1.42278167e+00 1.79244093e-05 1.92358126e-04] 63.67527817329559
[1.42280787e+00 1.79237617e-05 1.92358641e-04] 63.67267343648696
```



we see that in this fit, error is converged, and the fit parameters are:

$a \simeq 1.4228$

$w \simeq 1.7924\,e\text{-}5$

$t_0 \simeq 1.9236\,e\text{-}4$

(b) We again assume that the params are independent with uniform variance.

Note in linear case

param variance = $(A^T N^{-1} A)^{-1}$

If we follow the same derivation for non-linear fit, it is easy

for non-linear fit, it is easy to see that we simply need to replace

A with $\frac{dA}{dm}$

(Note A is $\frac{dA(m)}{dm}$ of Am, so it matches the linear case)

So we can calculate error bars:

```
#same as in linear case as discussed above
N=np.mean((x-pred)**2)
par_errs=np.sqrt(N*np.diag(np.linalg.inv(lhs)))
print(N, par_errs)
```

output is:

```
0.0006367267343648696 [4.25470571e-04 7.58840594e-09 5.35856269e-09]
```

the array is error for a, w, to, respectively

(c)

```
#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 = 1e-6
    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
        fun_minus = fun(param_new,t)
```

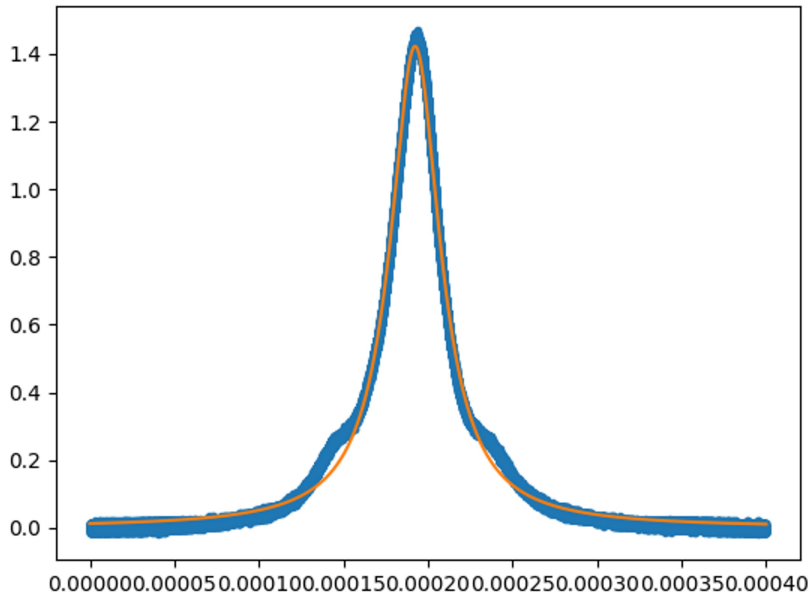
```
            param_new[i] = param[i]+dx
            fun_plus = fun(param_new,t)
            grad[:,i] = (fun_plus - fun_minus)/(2*dx)
        return grad

    def lorentz(p,t):
        y=p[0]/(1.0+((t-p[2])**2)/(p[1]**2))
        return y

    def calc_lorentz_2(p,t):
        y=lorentz(p,t)
        grad = fun_deriv(lorentz,p,t)
        return y,grad
```

Then we just need to replace calc-lorentz in last part with calc-lorentz_2

Output is

```
[7.67879725e-01 4.88882099e-05 1.92452749e-04] 40887.001248174354
[1.11965623e+00 3.68261076e-06 1.91249307e-04] 2975.808349430125
[7.57517434e-01 1.40509156e-05 1.91745237e-04] 8467.919039053904
[1.40674856e+00 2.10195058e-05 1.93001766e-04] 4144.491266326777
[1.40821795e+00 1.80698228e-05 1.92194509e-04] 236.39962607540022
[1.42201489e+00 1.79413972e-05 1.92359683e-04] 65.0179366557752
[1.42279425e+00 1.79241008e-05 1.92356890e-04] 63.6760540022498
[1.42282290e+00 1.79233831e-05 1.92357509e-04] 63.67273354488321
0.000636727335448832 [4.25806871e-04 7.59420519e-09 5.36680478e-09]
```

last line is error estimate.



Our result is very similar to analytical derivative. There is no statistical significant difference.

(d)

```
#p[0] = a, p[1] = w, p[2] = t0, p[3] = b
#p[4] = c, p[5] = dt
def tri_lorentz(p,t):
    y=p[0]/(1.0+((t-p[2])**2)/(p[1]**2)) + \
    p[3]/(1.0+((t-p[2]+p[5])**2)/(p[1]**2)) + \
    p[4]/(1.0+((t-p[2]-p[5])**2)/(p[1]**2))
    return y

#use numerical derivation to calculate grad matrix
def calc_lorentz_3(p,t):
    y=tri_lorentz(p,t)
    grad=fun_deriv(tri_lorentz,p,t)
    return y,grad
```

Again, we simply replace calc_lorentz in (a) with calc_lorentz_3.

Also from raw data I guessed the side peaks should be about 5e-5 apart from main peak, with height of about 0.2.

Therefore:

```
p0=np.array([1.42,1.79e-5,1.92e-4,0.2,0.2,5e-5])
```

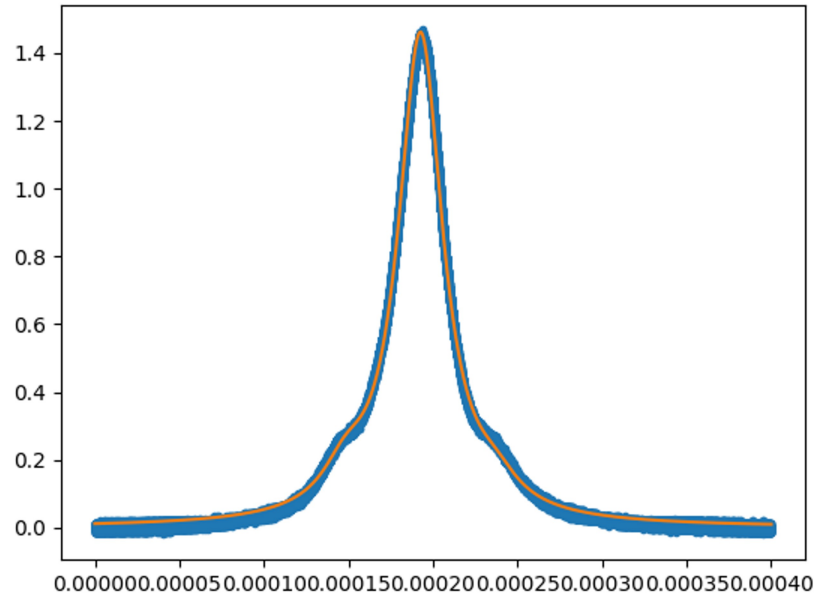
The output is:

```
[1.44264259e+00 1.61041997e-05 1.92568124e-04 1.03030081e-01
 6.92488065e-02 4.73235274e-05] 492.9359079698378
[1.44329289e+00 1.60506671e-05 1.92563762e-04 1.02521904e-01
 6.57836176e-02 4.41604298e-05] 22.896092954849124
[1.44288585e+00 1.60770900e-05 1.92579238e-04 1.03652809e-01
 6.41719534e-02 4.46815443e-05] 21.302834396811548
[1.44299650e+00 1.60626673e-05 1.92576654e-04 1.03923372e-01
 6.48796223e-02 4.45295471e-05] 21.250424793603642
[1.44296503e+00 1.60663132e-05 1.92577610e-04 1.03872918e-01
 6.46998628e-02 4.45714330e-05] 21.24765954346797
0.00021247659543467971 [2.66743493e-04 5.65611156e-09 3.16235707e-09 2.54315724e-04
 2.49006300e-04 3.80882993e-08]
```

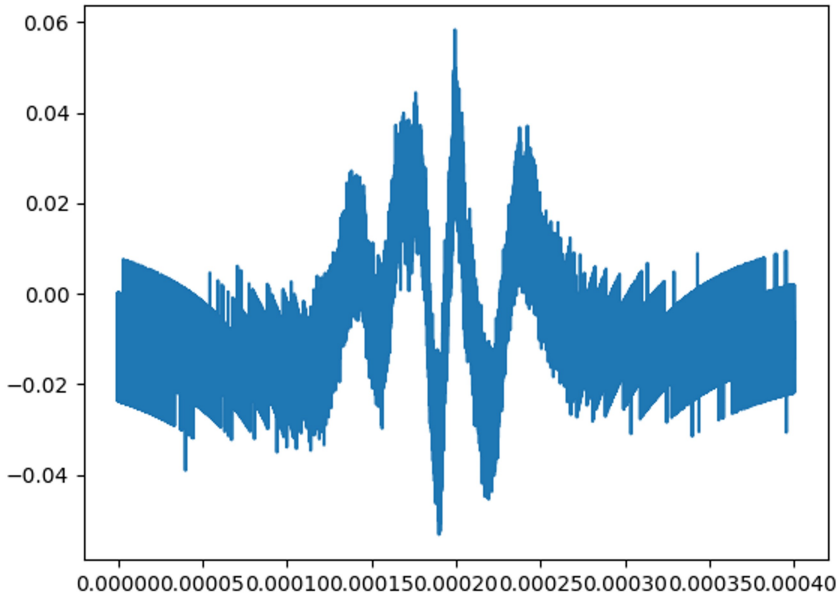
Again, the last line is error analysis.

please see above code to see which parameter is which in the array

please see above code to see which parameter is which in the array.



(e) we can simply plot t against x-pred



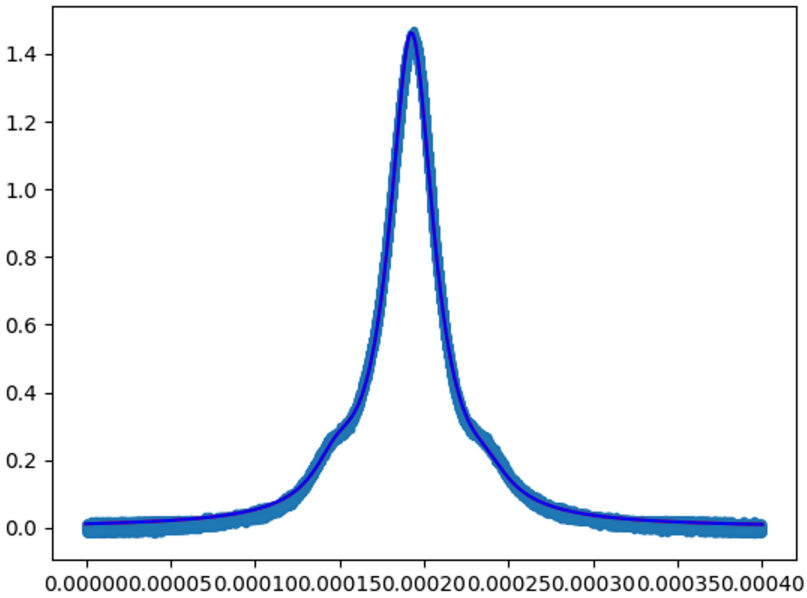
Obviously this is not white noise.
Our assumption of independent param
with uniform variance might be wrong.

(f)

Because we assumed uncorrelated noise,
we can write:

```
cov = N*np.linalg.inv(lhs)
param_with_err = np.random.multivariate_normal(p,cov)
x_with_err = tri_lorentz(param_with_err,t)
chisq = np.sum((x-pred)**2/N)
chisq_with_error = np.sum((x-x_with_err)**2/N)
print(chisq,chisq_with_error,np.abs(chisq-chisq_with_error))

plt.clf()
plt.plot(t,x,'.')
plt.plot(t,pred,c='r')
plt.plot(t,x_with_err,c='b')
plt.savefig('param_with_error.png')
plt.show()
```

A typical graph looks like:



we can see that the blue curve and the red curve are almost identical.

The typical chisq error is; from past 10 runs;

2.374036479071947
1.6866415007825708
7.676121223674272
1.6853407385060564
6.115824623295339
4.489754540176364
2.729472425489803
0.002052759999060072
0.3956101325165946
0.6570756523287855

So we can see that the chisq difference is very small.

small.

I am not sure if it is reasonable or not. Naively thinking I think it makes sense since our error bar (even though might be wrong) is very small.

(g)

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

#use the covariance matrix and noise from previous calculation
nparam = 6
cov = np.load('cov_matrix.npy')
N = np.load('sigma_squared.npy')
stuff=np.load('sidebands.npz')
t=stuff['time']
x=stuff['signal']

#get chisq for the three lorentzian fit
def get_chisq(t,x,p):
    y=p[0]/(1.0+((t-p[2])**2)/(p[1]**2)) + \
    p[3]/(1.0+((t-p[2]+p[5])**2)/(p[1]**2)) + \
    p[4]/(1.0+((t-p[2]-p[5])**2)/(p[1]**2))
    chisq = np.sum((x-y)**2/N)
    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

#mostly copied from class PPT
def run_mcmc(t,x,start_pos,nstep,step_size):
    nparam = start_pos.size
    params=np.zeros([nstep,nparam+1])
```

```
    params[0,0:-1] = start_pos #save initial params
    cur_chisq = get_chisq(t,x,start_pos)
    params[0,-1] = 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(t,x,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:-1]=cur_pos #save current params
        params[i,-1]=cur_chisq #save current chisq
    return params

#I tried and this scaling converges fastest
step_size = np.array([5,5,5,5,5,5])
#using an initial guess close to best fit params
pars_start = np.array([1.4,1.7e-5,2e-4,0.1,0.1,5e-5])
nstep=30000
chain=run_mcmc(t,x,pars_start,nstep,step_size)

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

#plot chisq
plt.plot(chain[:,-1])
plt.savefig('chisq_converge.png')
plt.show()

#plot param
plt.clf()
plt.plot(chain[:,0])
plt.savefig('first_param_conv.png')
plt.show()

plt.clf()
plt.plot(chain[:,1])
plt.savefig('second_param_conv.png')
plt.show()

#plot param fft
plt.clf()
plt.loglog(np.abs(np.fft.rfft(chain[:,0])))
plt.savefig('fft_first_param.png')
plt.show()
```

The output is:

val scat

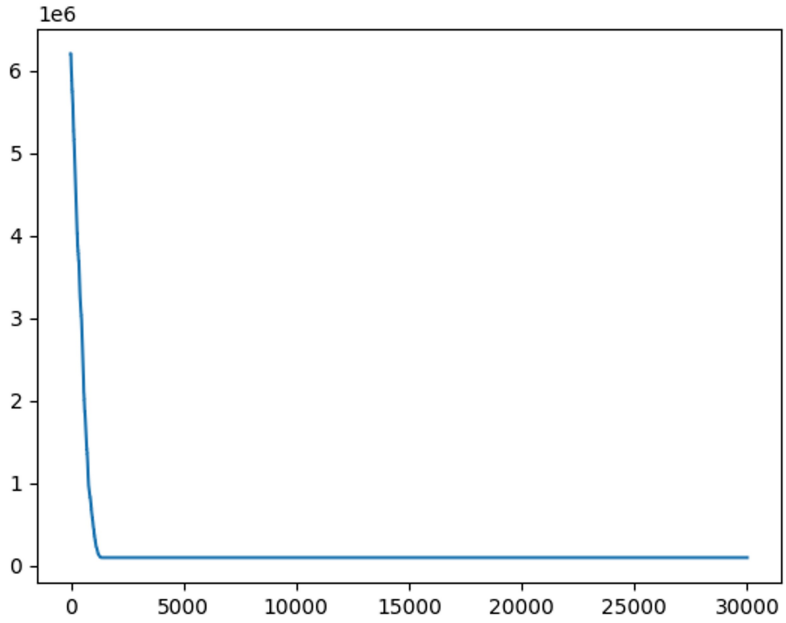
[1.4404597055616424, 0.012093211356032188]
[1.610501136284167e-05, 2.0031882876156906e-07]
[0.00019274340600460252, 9.038782215643899e-07]
[0.10389013100525335, 0.0021054510565209628]
[0.06512461147049076, 0.0026022519525476483]
[4.46491717689545e-05, 7.647628315639181e-07]

we can see that our mean for the parameters are very similar to the best fit by using Newton's method.

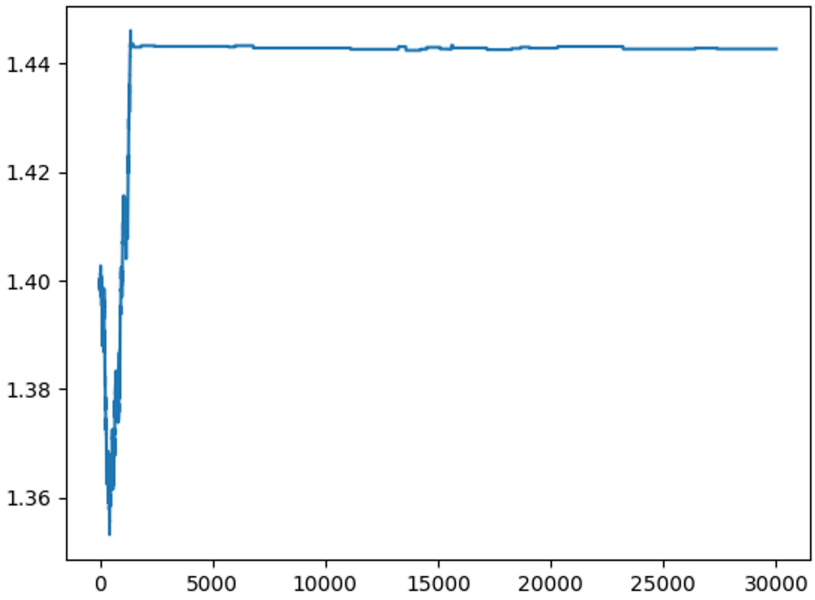
Our error bars indeed changes, sometimes by a factor of 10.

To show that our chain is converged.

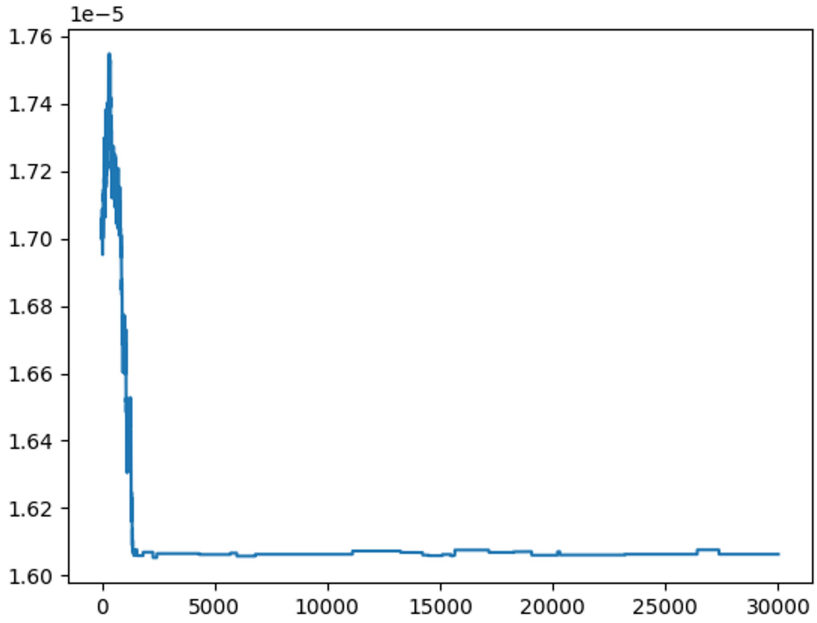
1. Chisq plot. we can see our chisq is converged



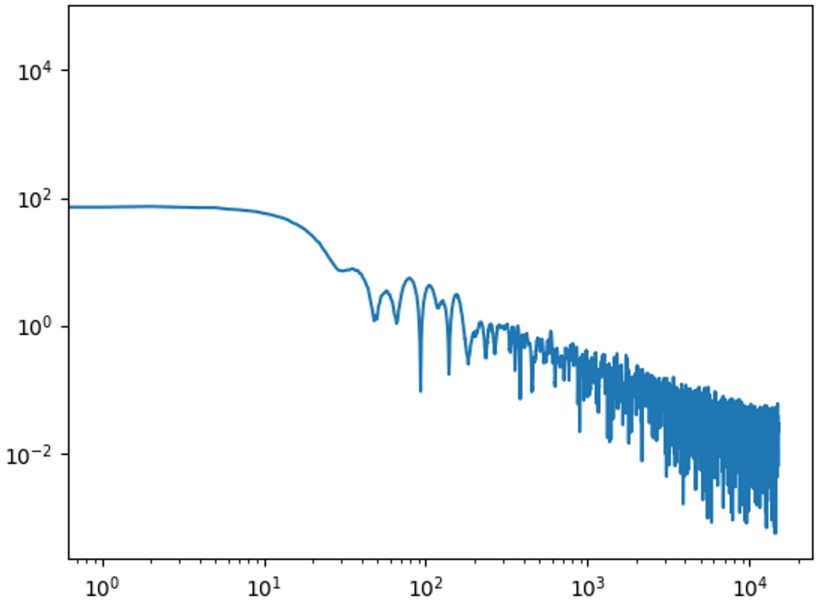
2. first parameter plot. Our first parameter looks converged.



3. Second parameter plot also looks converged.



4. FFT of first parameter. We can see that near k=0 it is flat, indicating it is converged,



(g)

(g)

So our dt with

4.4649×10^{-5} maps to 9 GHz.

Then our w with

1.6105×10^{-5} maps to

around 3.2463 GHz.