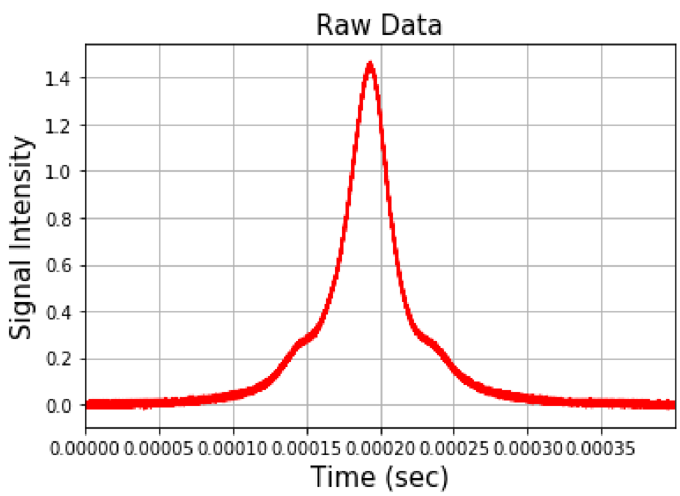
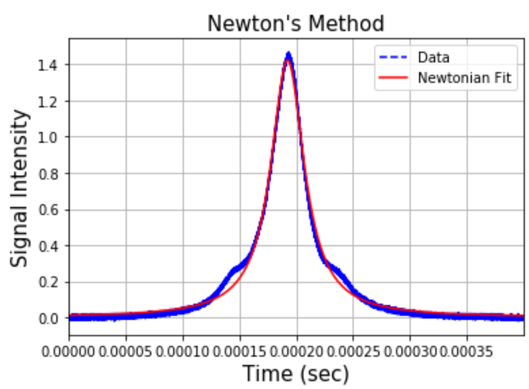
The solutions to PSET 4: Yacine Benkirane

We first load in the data…
=> Defining the Lorentzian functions returns the derivates of these functions w/ respect to varying gradient parameters.

We are finding the best parameters via Newton's Method (consisting of linearized Chi Squared Surfaces) and finding a step size to optimize a min value of Chi2. We may continue this process as long as the difference between our measured and aimed Chi2 is above an arbitrary lower bound. Initial guesses are presented in the code.






```
parameters:  [1.42281068e+00 1.92358649e-04 1.79236908e-05]
a =  1.422810680632242      t0 =  0.0001923586493756531       w =  1.792369079401261e-05
Noise:  [4.25479046e-04 5.35834556e-09 7.58809724e-09] a =  1.422810680632242  +- 0.000425479045871343 t0 =  0.0001923586493756531  +-
5.35834556331808e-09 w =  1.792369079401261e-05  +- 7.588097244722406e-09
```

(a = 1.42 +- 4.25e-4) (t0 = 1.92e-4 +- 5.36e-9) (w = 1.79e-5 +- 7.59e-9)

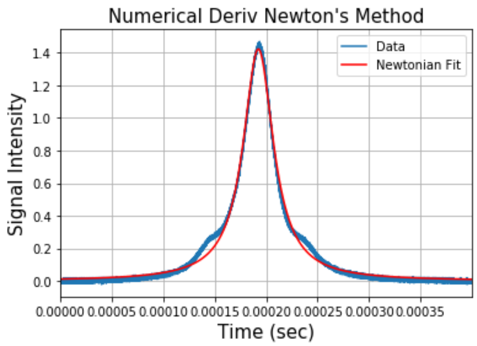
We utilize Jon's in-class notes for non-linear square fitting (and gradients derivations)...
We may then estimate the error in the data (noise) using gaussian stats: extrapolating the standard deviation from the curv_mat:

```

rms_err = np.sqrt(np.sum((d-pred)**2)/len(d))
p_err = np.sqrt(np.diag(curv_mat * rms_err**2))
print('Noise: ', p_err, 'a = ', p[0] , ' +-', p_err[0], 't0 = ', p[1], ' +-', p_err[1], 'w = ', p[2], ' +-', p_err[2],)
```

We apply these formulas to model our data and fit Lorentzian with respect to each parameter (using the gradient theory mentioned above from the notes):

```
def lorentz(p, t):
    a, t0 , w = p
    return a / (1 + ((t - t0) / w)**2)

def cacul_lorentz(p,t):
    a, t0, w = p
    y = lorentz(p, t)
    gradient = np.zeros([len(t), len(p)])
    gradient[:,0] = 1.0 / (1 + (t-t0)**2 / w**2)
    gradient[:,1] = (2 * (t-t0) / (t**2 - 2 * t0 * t + t0**2 + w**2) ) * y
    gradient[:,2] = (2 / w - 2 * w / (t**2 - 2 * t0 * t + t0**2 + w**2)) * y
    return y , gradient

def cacul_lortenzN(p, t):
    a, t0, w = p
    y = lorentz(p, t)
    gradient = np.zeros([len(t), len(p)])

    llor = lambda p: lorentz(p, t)
    gradient = Ndiff(llor, p).transpose()

    return y , gradient
```

As a result, this is our model for Newton's Method yielding best-fit parameters…




```
parameters:  [1.42281068e+00 1.92358649e-04 1.79236908e-05]
a =  1.422810680632242      t0 =  0.0001923586493756531      w =  1.792369079401261e-05
                Equiv Diff Fit: [1.42281068e+00 1.92358649e-04 1.79236908e-05]                    N diff Fit: [1.42281068e+00 1.92358649e-04
1.79236908e-05]
                  Uncertainty): [4.25479046e-04 5.35834556e-09 7.58809724e-09]

                    Difference:  [-4.52526905e-12 -1.49077799e-18  5.84215564e-17]
```

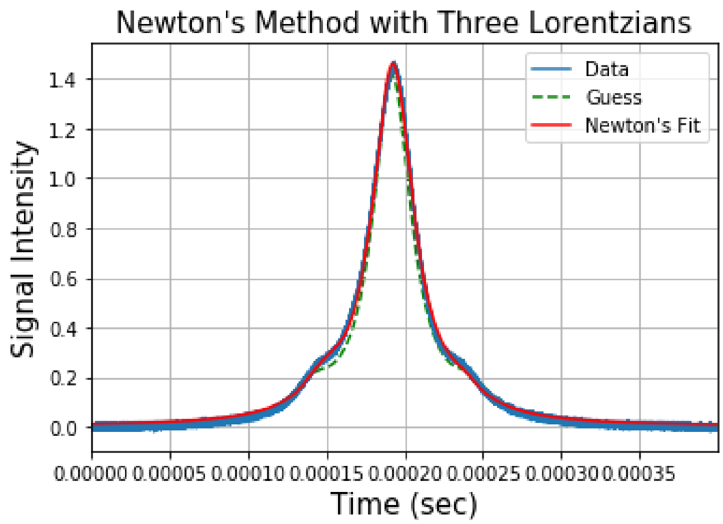
(a = 1.42   +-   4.25e-4)   (t0 = 1.92e-4   +-   5.36e-9)   (w = 1.79e-5   +-   7.59e-9)

Notice that the difference in the parameters is extremely small, and negligible!
The parameters seem to agree… The modelling techniques are statistically equivalent (within noise error).

We repeat using the Three Lorentzian technique:

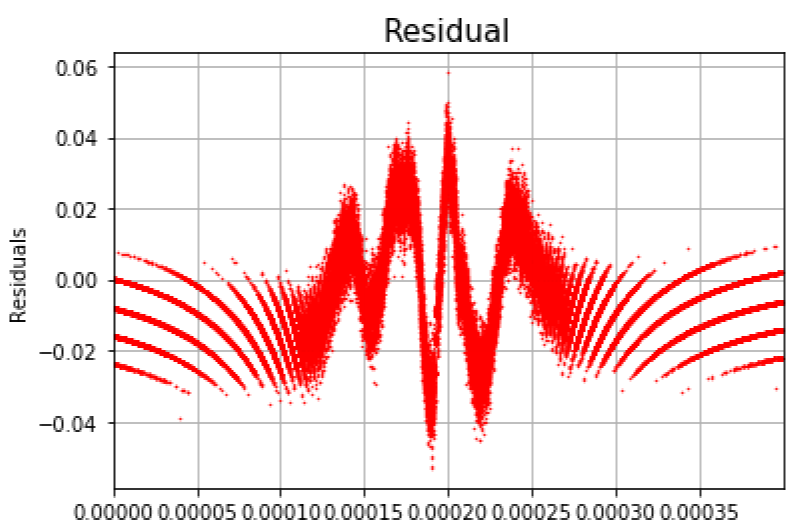
```python
def ThreeLorentz(p, t):
    a, b, c, t0, w, dt = p
    return a/(1 + (t-t0)**2 / w**2) + b/(1 + (t-t0+dt)**2 / w**2) + c/(1 + (t-t0-dt)**2 / w**2)

def cacul_ThreeLorentz(p, t):
    y = ThreeLorentz(p, t)
    gradient = np.zeros([len(t), len(p)])
    TriLorentz = lambda p: ThreeLorentz(p, t)
    gradient = Ndf(TriLorentz, p).transpose()
    return y , gradient

steps = 20
```

```
#Initial Guess for Parameters
a = 1.4
b = 0.1
c = 0.1
w = 0.000015
t0 = 0.000192
dx = 5e-5

p0 = np.array([a, b, c, t0, w, dx])
p = p0.copy()

for i in range(steps):
    pred, gradient = cacul_ThreeLorentz(p,t)
    r = d - pred
    r = np.matrix(r).transpose()
    gradient = np.matrix(gradient)

    lhs = gradient.transpose() @ gradient
    rhs = gradient.transpose() @ r
    curv_mat = np.linalg.inv(lhs)
    dp = curv_mat@(rhs)
    for j in range(len(p)):
        p[j] += dp[j]

plt.plot(t, d, label = 'Data')
plt.plot(t, cacul_ThreeLorentz(p0, t)[0], 'g--', label = "Guess", zorder = -1)
plt.plot(t, pred, 'r-', label = 'Newton\'s Fit')
plt.autoscale(enable=True, axis='x', tight=True)
plt.xlabel('Time (sec)', fontsize=15)
plt.ylabel('Signal Intensity', fontsize=15)
plt.title('Newton\'s Method with Three Lorentzians', fontsize=15)
plt.grid()
plt.legend()
plt.savefig("NewtonMethodThreeLor.png")
plt.show()
```




```
Best Parameter Fits:
         [1.44299240e+00 1.03910783e-01 6.47325292e-02 1.92578522e-04
 1.60651094e-05 4.45671634e-05]
 Error RMS:                   0.01457644476006902
 Unc:
                               [2.66428694e-04 2.54116769e-04 2.48823333e-04 3.15440252e-09
 5.64926768e-09 3.80268482e-08]
```

With a residual:



(a = 1.44 +- 2.66e-4) (b = 1.04e-1 +- 2.54e-4) (c = 6.47e-2 +- 2.49e-4)
(t0 = 1.92e-4 +- 3.15e-9) (w = 1.61e-5 +- 5.65e-9) (dx = 4.46e-5 +- 3.80e-8)

Here, we note that the earlier Error RMS was roughly 0.0146, which would fall within the residual's magnitude range. It's not fully understood why the residual is in this shape, but other class members seem to agree with very similar results so this must be some data artifact with deeper meaning relating to the three Lorentzians potentially…

As mentioned by Prof Sievers, we may form the correlated noise using "Cholesky decomposition of the covariance Matrix" $A_m^T N^{-1} A_m.$ => covariance about the minimum value of Chi-Squared…

```
for i in range(5):
    p_rand = p + np.linalg.cholesky(curv_mat * rms_err**2)@np.random.randn(len(p))
    p_rand = p_rand.tolist()[0]
    plt.plot(t, cacul_ThreeLorentz(p_rand, t)[0] - cacul_ThreeLorentz(p, t)[0], lw = 1, label = f"{i+1}")
    plt.legend()
    plt.autoscale(enable=True, axis='x', tight=True)
    plt.xlabel('Time (t)', fontsize=15)
    plt.ylabel('Residuals')
    plt.title('Alternate Fit Residuals', fontsize=15)
    plt.grid()
plt.savefig("AlternateFitRes.png")
plt.show()
```

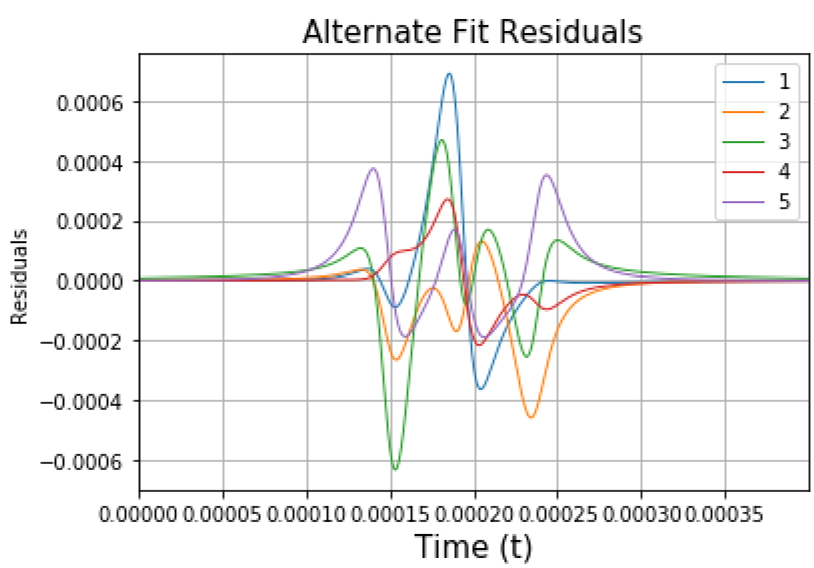
Which utilizes cacul_ThreeLorentz and Ndiff:

```
def Ndf(f, x):
    diffs = []
    eps = 1e-16 # Precision of the Machine
    dx = np.sqrt(eps)
    for i in range(len(x)):
        iter1 = np.zeros(len(x))
        iter1[i] += 1

        m1 = x.copy()
        m2  = x.copy()
        p1  = x.copy()
        p2 = x.copy()

        m2 -= 2 *dx * iter1
        m1  -= dx * iter1
        p1  += dx * iter1
        p2 += 2*dx * iter1
        diffs.append((f(m2) + 8 * f(p1) - 8 * f(m1) - f(p2))/(12 * dx))
    return np.array(diffs)
```

We may iterate through alternative **realization** fits quite easily as such:

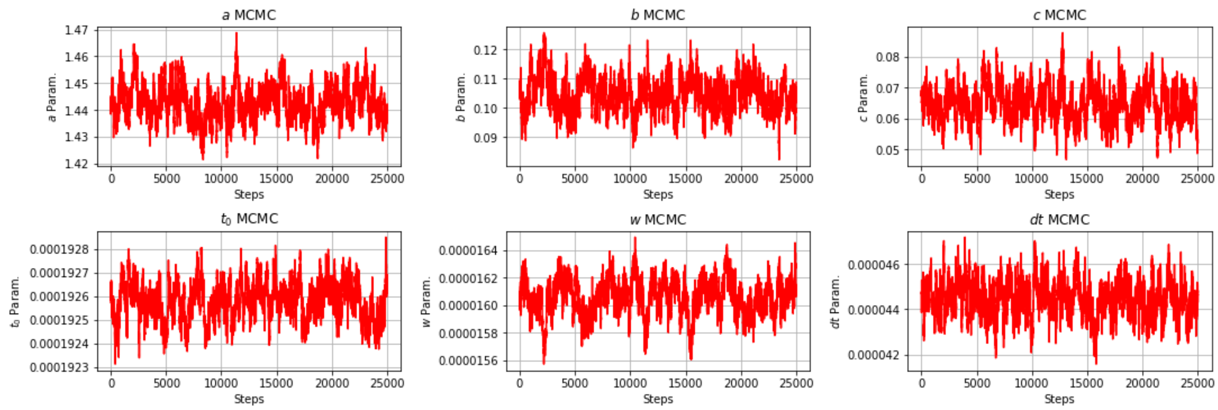
```python

for i in range(5):
    p_rand = p + np.linalg.cholesky(curv_mat * rms_err**2)@np.random.randn(len(p))
    p_rand = p_rand.tolist()[0]
    plt.plot(t, cacul_ThreeLorentz(p_rand, t)[0] - cacul_ThreeLorentz(p, t)[0], lw = 1, label = f"{i+1}")
    plt.legend()
    plt.autoscale(enable=True, axis='x', tight=True)
    plt.xlabel('Time (t)', fontsize=15)
    plt.ylabel('Residuals')
    plt.title('Alternate Fit Residuals', fontsize=15)
    plt.grid()
plt.savefig("AlternateFitRes.png")
plt.show()

```

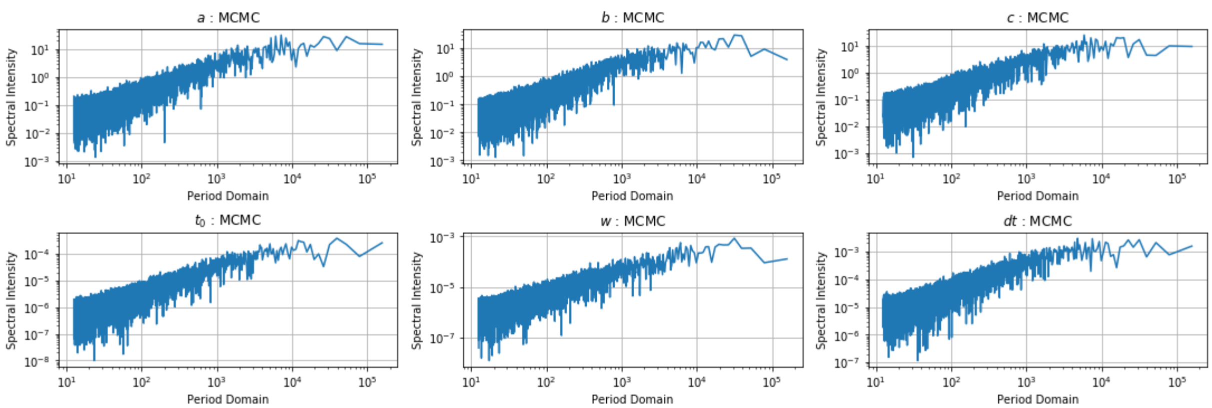
Resulting in



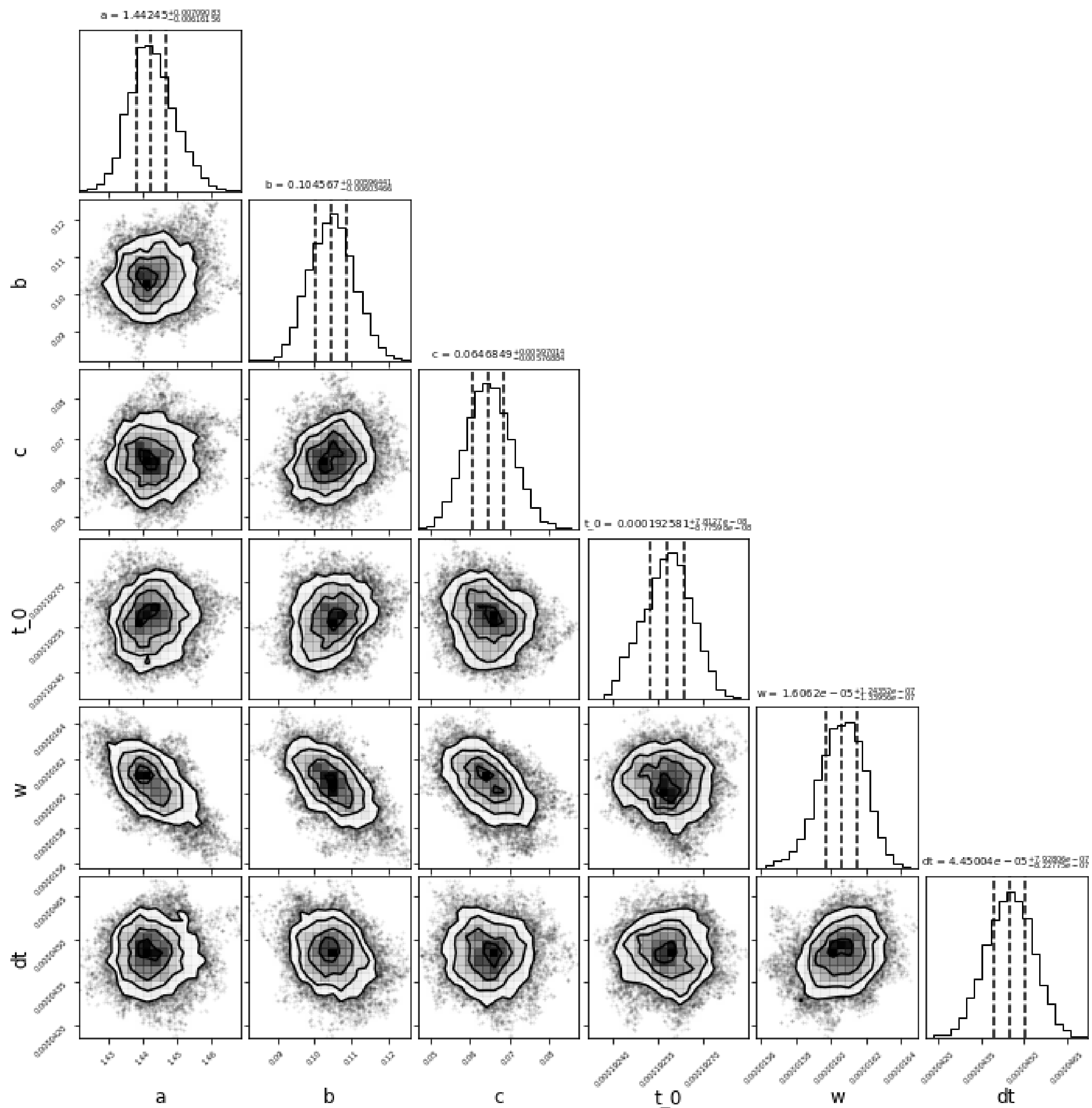
Subsequently, we may generate more realizations of the model and calculate the reduced chi-squared for each iteration. We generate here an average value of Chi2 of 317.48

I tried running the simulation at a few different step values. Anything above 2000 steps seemed to converge quite nicely with lower parameter uncertainties at higher step counts. I conceded at 25,000 steps with a scaling factor of 4. The MCMC converged quite nicely this way. All uncertainties values presented as such seemed to fit the MCMC rules and the logic of the simulation as a whole within agreeable time spans.

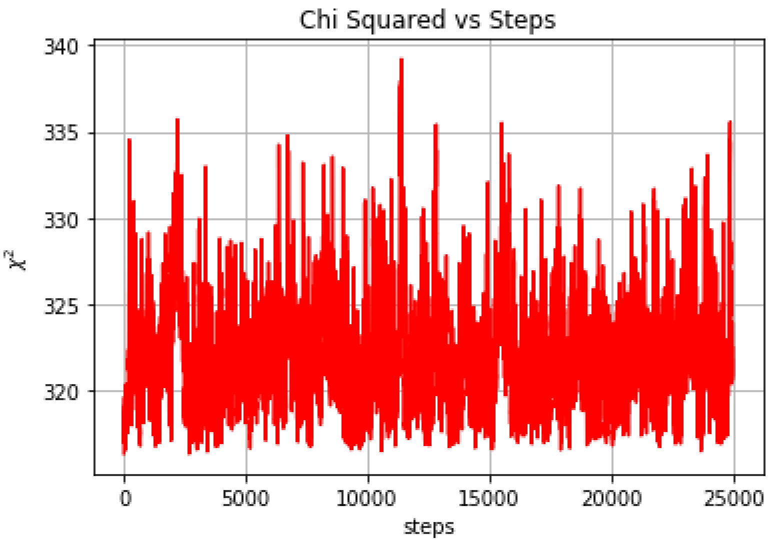




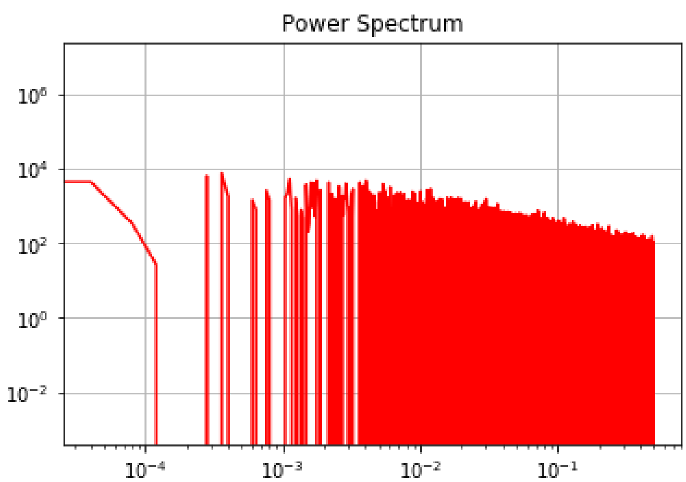
From the previous plot, it does seem like the routes have converged in this fitting problem. To further understand the underlying mechanics of the MCMC, let's plot the multidimensional cross-sections in the corner.corner library imported into spyder.



It seems that there is some correlation between the b and c params. After some discussions with classmates, this would seem to result from an offset in the eigenvectors of the hessian at Chi Squared's minimum, which can be plotted on a histogram for extra proof. Tried to discuss the noise around the contours after class but remained quite confused as to their origin.



The Minimum was found to be at 316.38 from this plot.




```
             Fit using MCMC: [1.44277858e+00 1.05171871e-01 6.51354150e-02 1.92573132e-04
1.60492627e-05 4.43552066e-05]
     Uncertainty using MCMC: [6.68507960e-03 6.28288436e-03 6.19027574e-03 8.78042607e-08
1.26101651e-07 8.76834756e-07]
```

From the plots above (as described in Prof Siever's notes to further pass an FFT to get the "power spectrum"), we can see that the time series indicates that the MCMC has converged indeed…!

```python
#From the MCMC Fit: w = 1.60*10e-5 and dt = 4.437*10e-5 and e_err = 1.418*10e-7
w = 1.60*10e-5
dt = 4.437*10e-5
w_err = 1.418*10e-7
width = dt/w*9
err_width = width*(w_err/w)
print(width, err_width)
```

$$\Delta s = \frac{dx}{w}\ 9\ \text{GHz}$$

OUTPUT:

```
24.958125000000003 0.2211913828125
```

Therefore, the width of the cavity resonance is:

$width\colon\ (24.96\ \pm\ 0.22)\ Ghz$

We can likely attribute the high error to the limited number of steps used in the Markov chain (roughly 25,000) or an iffy trial distribution resulting in slower convergence and higher uncertainty! Noisy signal?

I tried not to fill the PDF with too much code for ease of marking. All 300 lines of code are available on GitHub for further inspection.

Thank you for reviewing my solutions. Hope I did not omit anything, please do let me know if I can improve on anything. I'm being quite cautious as I did poorly on Assignment 3 and would like my grade not to suffer.

Cheers,
Yacine