Phys 512 Problem Set 4

Jan Viatteau (260857910)

Problem 1

(a) The model we want to use is the Lorentzian :

$$f(t) = \frac{a}{1 + (t - t_0)^2/w^2}$$

In order to use Newton's method, we should determine analytically the derivatives of the above function with respect to each of the parameters that we want to estimate, here a, w and t_0. After some chain rules and scribbling, we get :

$$\frac{\partial f}{\partial a} = \frac{1}{1 + (t - t_0)^2/w^2}$$
$$\frac{\partial f}{\partial w} = \frac{2aw(t - t_0)^2}{(w^2 + (t - t_0)^2)^2}$$
$$\frac{\partial f}{\partial t_0} = \frac{2aw^2(t - t_0)}{(w^2 + (t - t_0)^2)^2}$$

The gradient for each t is then calculated by a method `calc_lorentz`, which is then used to determine the direction of the next step in parameter space following the $dm = (A'^T A')^{-1}(A'^T r)$ with $r = d - A(m)$ seen in class. The initial guesses for the parameters are $a = 1.4$, $w = 10^{-5}$, $t_0 = 2 \times 10^{-4}$ based on a look at the plot of data vs t. Since Newton's method converges pretty fast, we are satisfied after taking 10 steps. The resulting best-fit parameters are then printed as :
`a = 1.4228106805885952, w = 1.7923690795113296e-05,`
`t_0 = 0.00019235864937525854`
and the resulting model is plotted over the data below.

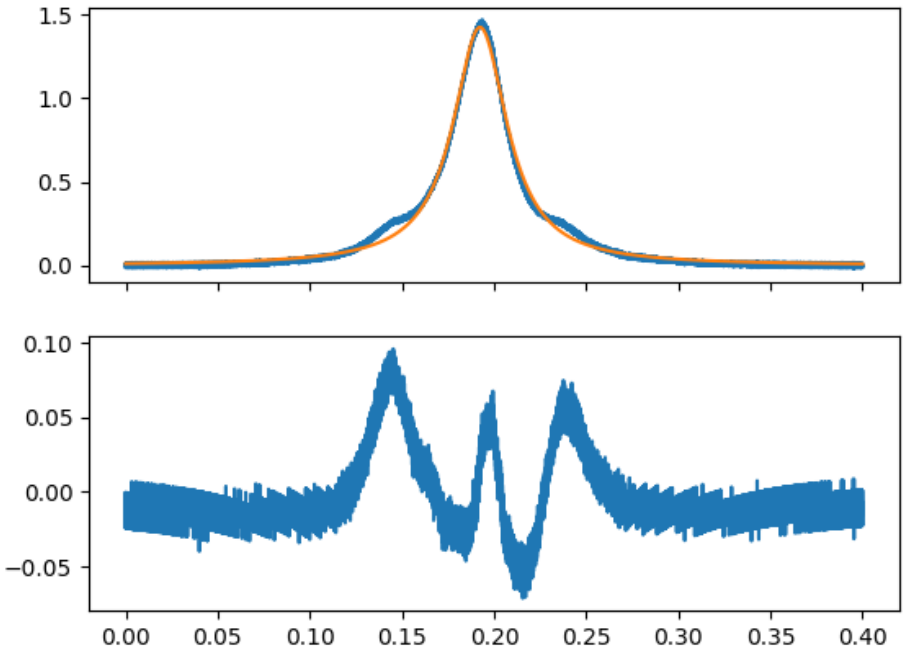

Figure 1: Resulting model with the best-fit parameters given by applying Newton's method with 10 steps, and the residuals below.

(b) An estimate of the noise in the data can be, for example, the mean of the absolute value of the difference between data and model. Using that, we can construct our N matrix (without actually constructing it, as it should be a square matrix with the number of measurements as its size, too huge of a memory requirement) as a diagonal matrix with value σ^2 on its diagonal (where σ is our estimate of the noise). We invert that by simply taking the inverse of each value, and calculate the covariance matrix $A^T N^{-1} A$ (where A is now the matrix of gradients that we got while doing (a)). Then, the numbers on the diagonal are the squared errors on each of our parameters. We end up getting :
`Errors are :  0.00032632195910660774 for a,`
`5.819705544011838e-09 for w,`
`4.109593271640055e-09 for t_0.`

(c) Here, the process is exactly the same as for (a), only we have to take a numerical estimate of the derivatives in a helper function, `num_derivs`. That is easily done following the logic of the `ndiff` method we made for problem set 1 (ie. estimating the 2-sided derivative with a somewhat optimal dx), with an added `for` loop that iterates through the parameters. We should also make `num_derivs` take in a function as an argument, so that we will be able to reuse it with no changes for the sum of 3 Lorentzians that's coming up. Then store the derivatives in an array that you output, and the rest is the same as for (a) : same method, same number of steps, same initial guesses. The best-fit parameters that we get are then :

```
a = 1.4228106805863208,
w = 1.792369079517064e-05,
t_0 = 0.00019235864937434372
```

which are definitely not statistically different from those we had the analytic way (ie. the values differ by less than the error on them). We can also check visually that the plots look the same.

(d) To do the sum of three Lorentzians fit, we estimate that it's too much of a pain to get analytic derivatives so we just define the function as it should be and do it the numerical way. For the initial guesses, we take the best-fit values gotten in (c) for a, w and t_0 and rough guesses based on looking at the plot for each of b, c and dt (which end up being 0.2, 0.2 and 5×10^{-5} respectively). Then we follow the same logic as in (b) to get errors. The values we end up getting are :

```
Triple Lorentzian numerical fit :
a = 1.4429924037493902 +/- 0.0002304188603819499
w = 1.6065108400845862e-05 +/- 4.885725819036616e-09
t_0 = 0.0001925785216042846 +/- 2.7280616381352314e-09
b = 0.10391079728050076 +/- 0.00021977086482970475
c = 0.06473257774450818 +/- 0.00021519286999396924
dt = 4.4567152614536e-05 +/- 3.2887274311376356e-08
```

Interestingly, the errors on a, w and t_0 have went down, but not by much (whereas it seems the model follows the data much better, as is shown in the plot below).

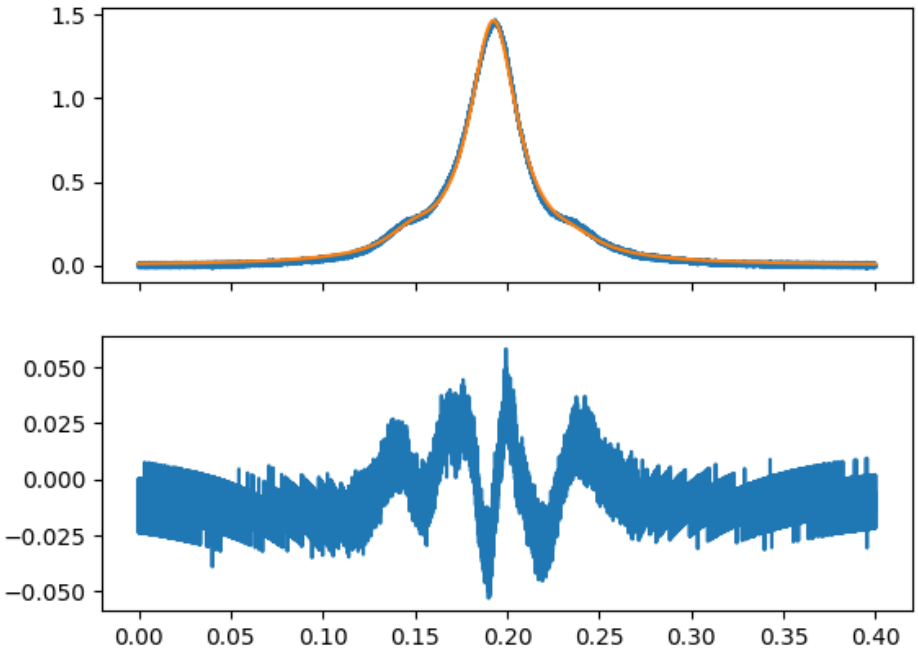

Figure 2: Data and model for the sum of three Lorentzians of same width, but different amplitudes and centers. The residuals are shown as well.

(e) As can be seen in the residuals above, it seems that indeed there is some correlation between the data points and errors, since the residuals have a definite shape and don't just look like random noise (ie. nearby data points will have similar errors in a predictable way, so there must be correlation between them). So indeed, hard to trust our previously determined error bars and that our model gives a complete description of the data.

(f) Here, we basically use the covariance matrix and best-fit parameters we got earlier in (d), and feed them into the numpy method `np.random.multivariate_normal`. This will get us a new set of parameters chosen randomly around each best-fit parameter. Using the full covariance matrix will take into account the correlation between the parameter errors when giving the new set of parameters. Since the

errors are relatively small, we can't see any difference when plotting the model with the new parameters.

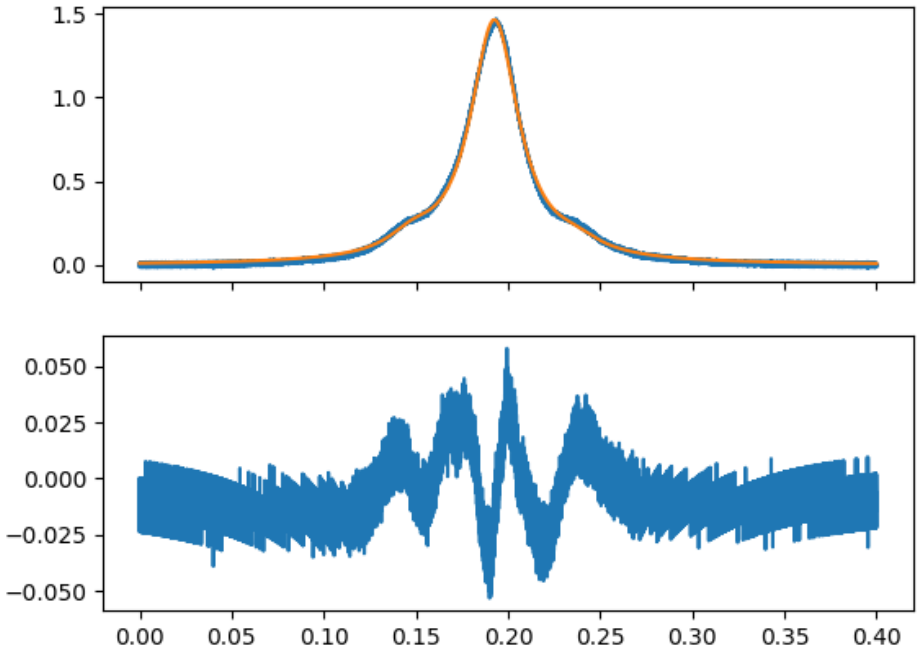

Figure 3: Model and data (and their residuals) for parameters slightly offset from the best-fit set.

The difference between chi-squared values are on the order of 10 to 100 :

```
Best-fit chi squared is :  133698.31386768597
Random offset parameters chi squared is :  133651.41091334508
```

This is pretty reasonable, considering the size of chi squared here. However, the offset parameters yield a chi squared value inferior to that of the best-fit, which seems counter-intuitive to say the least.

(g) For the MCMC, we use the offset parameters from (f) as a starting point, then use the same numpy method as above to get each next step. We still use the covariance matrix we had in (d), and take 10000 steps.

When the trial step gives a lower chi squared value, we automatically accept, and sometimes accept the step in the other case. After the chain has generated, we use the chain's mean for each parameter as the estimate and the standard deviation as the error on each estimate. At the end, we truncate the first 100 points on the chain to avoid getting non-converged points that are always present at the beginning. The results are :

```
a = 1.4363244532325605 +/- 0.0004432020410227242
w = 1.6280913507238715e-05 +/- 7.784577048094217e-09
t_0 = 0.0001926871398947602 +/- 4.179243596356281e-09
b = 0.10528788581509253 +/- 0.0006591356850227451
c = 0.05573817769284255 +/- 0.0003119835059426063
dt = 4.378228680699387e-05 +/- 8.157888509148109e-08
```

Interestingly, the errors are pretty much the same as before (they're slightly worse), but each parameter's estimate is different from the ones we had from Newton's method. Perhaps that is due to some features of chi squared in parameter space that we couldn't probe with Newton's method.

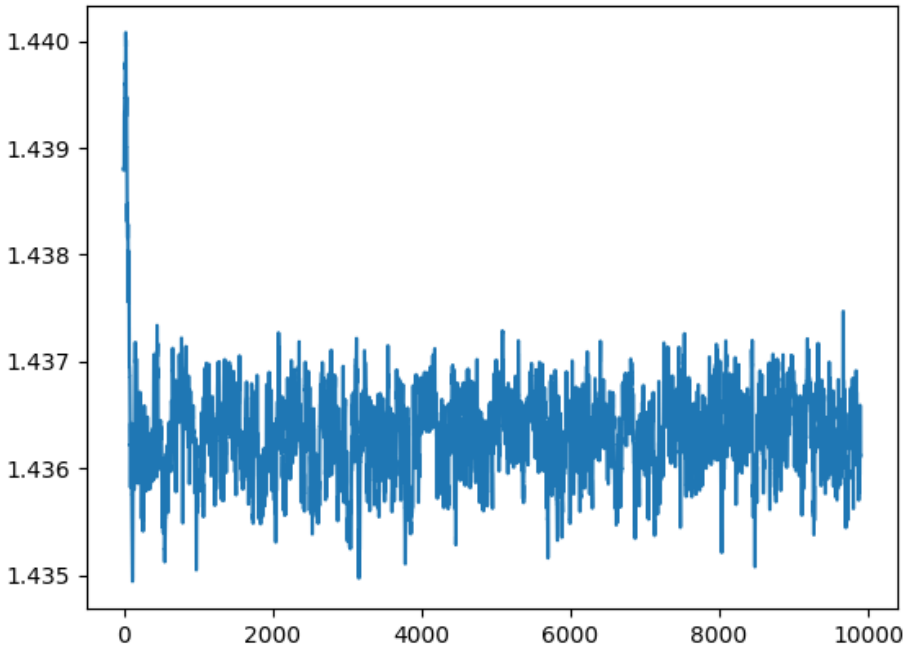

Figure 4: MCMC chain for a. As can be seen, after going down from the value we started with (the value close to the one obtained from Newton's method), the chain is oscillating around about 1.4365, looking like white noise. This is evidence that the chain has converged.

(h) Using our estimates from the MCMC we had, we can convert the non-dimensional values we got into GHz values. Since the sidebands are

separated from the main band by $dt = 9$ Ghz, it means that :

$$\begin{aligned}
\frac{w}{w(\text{Ghz})} &= \frac{dt}{9\ \text{Ghz}} \\
w(\text{GHz}) &= w\frac{9\ \text{GHz}}{dt} \\
&= 9\ \text{GHz} \times \frac{w}{dt} \\
&\approx 9 \times \frac{1.62809 \times 10^{-5}}{4.37822 \times 10^{-5}}\ \text{GHz} \\
w &\approx 3.34675\ \text{GHz}
\end{aligned}$$