PHYS 512 - PS3

Damien Pinto (260687121)

October 2019

Note: Many of my files start with "10000_mcmc...", but actually refer to the 7930 step Markov Chain-Monte Carlo (MCMC) simulation I mention repeatedly. I simply had to stop early to actually have a hope of writing this write-up.

1)

Code for this question: *python3 wmap_camb_example.py*
First, I modified line 37 such that the get_params() function provided by Prof. Sievers only output a fit (*cmb*) that is the same length as our data (+2 for the monopole and dipole terms), which . We then use *wmap[:,1]* as our x_i, *cmb* (with the two first terms excluded) as our μ_i, and *wmap[:,2]* as σ_i in

$$\chi^2 = \Sigma_{i=1}^{N} \left(\frac{x_i - \mu_i}{\sigma_i} \right)^2 \tag{1}$$

, which returns a value of $\chi^2 = 1588.4366....$

2)

Code for this question: *python3 levenberg_marquadt.py*
As indicated by the question, I implemented a Levenberg-Marquardt fitting algorithm. I constructed the matrix of derivatives ($A' = Ap$) of the data using a two-sided derivative. The function *(*get_Ap) gets a function and a set of parameters to input into that function. One by one, for each parameter individually, the function augments the parameter it's on by a 200th of its initial value, evaluates the function there, subtracts one 200th of its value, and computes the derivative there. It then takes the difference and divides by one 100th of the parameter's initial value to construct the two-sided derivative. I acknowledge that the factor of one 200th seems a little arbitrary. I tried to find the optimal δp given our parameters and what we saw in class, that the optimal increment with respect to a parameter x is:

$$\delta p^x_{optimal} = \sqrt{\frac{\epsilon f(x)}{f''(x)}}. \tag{2}$$

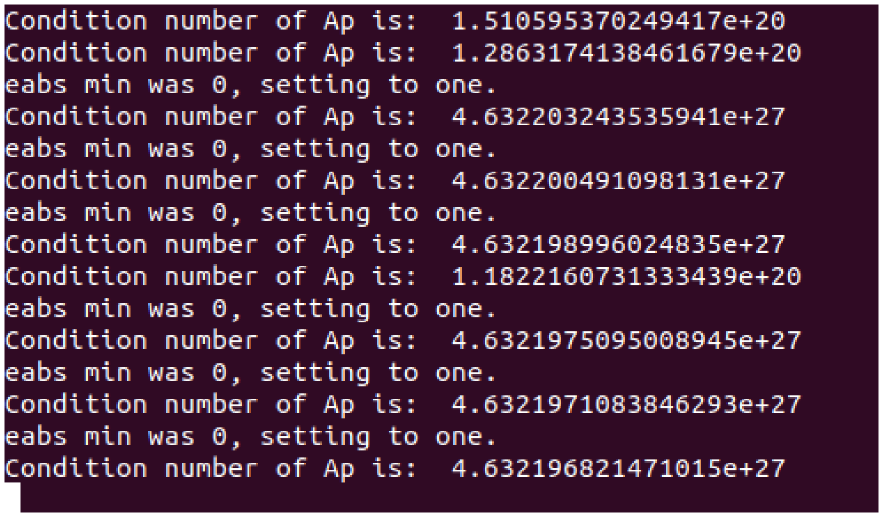
where $\epsilon = 10^{-12}$ is the machine precision for doubles. I then substituted :

$$\begin{aligned} f''(x) &\approx \frac{f'(p + \delta p^x) - f'(x - \delta p)}{2\delta p^x} \\ &\approx \frac{(f(x + 2\delta p^x) - f(x)) - (f(x) - f(x - 2\delta p^x))}{2(\delta p^x)^2} \\ &\approx \frac{f(x + 2\delta p^x) - f(x - \delta p^x)}{2(\delta p^x)^2} \end{aligned}$$

which gives:

$$\delta p^x_{optimal} = \sqrt{\sqrt{\frac{2\epsilon f(x)}{f(x+2\delta p^x) - f(x-\delta p^x)}}},$$

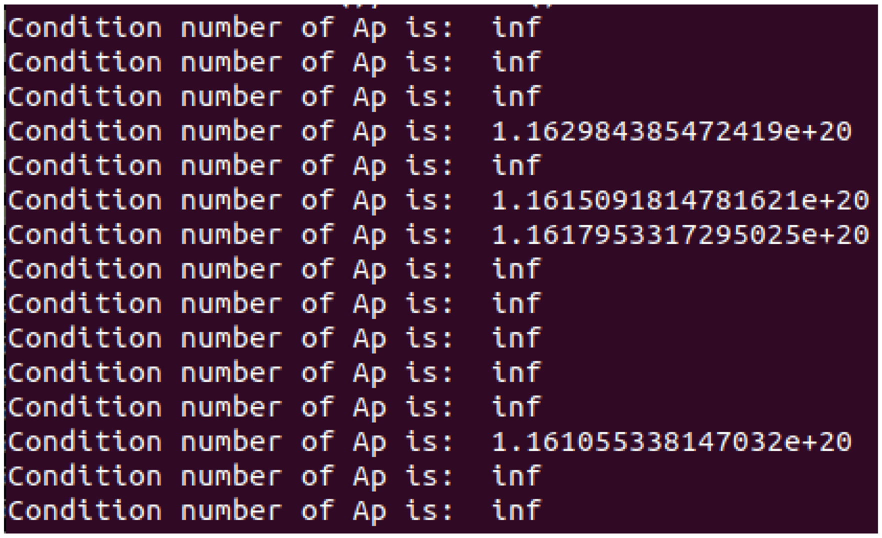
and then tried to compute the right-hand side numerically for various δp^x to see which one produced results closest to δp^x. This would have to be done for every parameter, without guarenty of finding the right δp^x for each one. Furthermore it always failed because it produced infinite values... so I abandoned that and tried just checking the condition number (ratio of the largest and lowest eigenvalues) of my Ap matrices for various fractions (done with the code in: *get_ders.py*). This also didn't produce the greatest results. Here are the condition numbers for a fraction number n from 2 to 10 (as in I would add and subtract p/n to the parameters to compute the derivatives):



```
Condition number of Ap is:  1.510595370249417e+20
Condition number of Ap is:  1.2863174138461679e+20
eabs min was 0, setting to one.
Condition number of Ap is:  4.632203243535941e+27
eabs min was 0, setting to one.
Condition number of Ap is:  4.632200491098131e+27
eabs min was 0, setting to one.
Condition number of Ap is:  4.632198996024835e+27
Condition number of Ap is:  1.1822160731333439e+20
eabs min was 0, setting to one.
Condition number of Ap is:  4.6321975095008945e+27
eabs min was 0, setting to one.
Condition number of Ap is:  4.6321971083846293e+27
eabs min was 0, setting to one.
Condition number of Ap is:  4.632196821471015e+27
```

Figure 1: *Condition numbers for the gradient matrix Ap when trying out a denominator of 2 (top) to 10 (bottom) in steps of 1.*

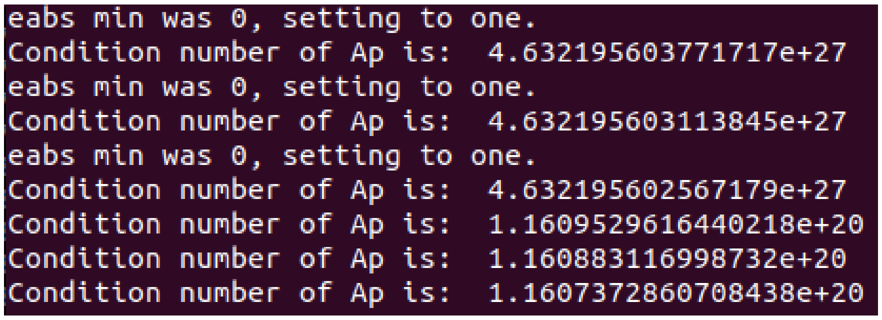
Then I tried denominators of 10 to 150 in steps of 10:



```
Condition number of Ap is:  inf
Condition number of Ap is:  inf
Condition number of Ap is:  inf
Condition number of Ap is:  1.162984385472419e+20
Condition number of Ap is:  inf
Condition number of Ap is:  1.1615091814781621e+20
Condition number of Ap is:  1.1617953317295025e+20
Condition number of Ap is:  inf
Condition number of Ap is:  inf
Condition number of Ap is:  inf
Condition number of Ap is:  inf
Condition number of Ap is:  inf
Condition number of Ap is:  1.161055338147032e+20
Condition number of Ap is:  inf
Condition number of Ap is:  inf
```

Figure 2: *Condition numbers for the gradient matrix Ap when trying out a denominator of 10 (top) to 150 (bottom) in steps of 10.*

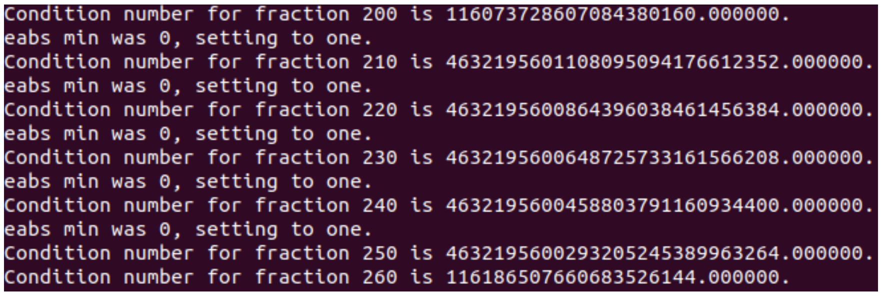
Then from 150 to 200:



```
eabs min was 0, setting to one.
Condition number of Ap is:  4.632195603771717e+27
eabs min was 0, setting to one.
Condition number of Ap is:  4.632195603113845e+27
eabs min was 0, setting to one.
Condition number of Ap is:  4.632195602567179e+27
Condition number of Ap is:  1.1609529616440218e+20
Condition number of Ap is:  1.160883116998732e+20
Condition number of Ap is:  1.1607372860708438e+20
```

Figure 3: *Condition numbers for the gradient matrix Ap when trying out a denominator of 150 (top) to 200 (bottom) in steps of 10.*

And finally from 200 to 260:



```
Condition number for fraction 200 is 116073728607084380160.000000.
eabs min was 0, setting to one.
Condition number for fraction 210 is 4632195601108095094176612352.000000.
eabs min was 0, setting to one.
Condition number for fraction 220 is 4632195600864396038461456384.000000.
eabs min was 0, setting to one.
Condition number for fraction 230 is 4632195600648725733161566208.000000.
eabs min was 0, setting to one.
Condition number for fraction 240 is 4632195600458803791160934400.000000.
eabs min was 0, setting to one.
Condition number for fraction 250 is 4632195600293205245389963264.000000.
Condition number for fraction 260 is 116186507660683526144.000000.
```

Figure 4: *Condition numbers for the gradient matrix Ap when trying out a denominator of 200 (top) to 260 (bottom) in steps of 10.*

This took a bit of time, and didn't seem to be returning great results, so I just used $n = 200$ for my denominator in my get_Ap function as it produced the best condition number and seemed potentially reasonable.

I used this along with my lev_marq function to fit all the parameters defining the power spectrum output by CAMB while holding $\tau = 0.05$. The results can be seen in the first column of Table 1. The fact that this is producing a much better fit than what CAMB output, also that the χ^2 value decreases steadily during the process, indicate to me that it seems to be working somewhat. Another interesting fact is that when I made a mistake and failed to allow my loop to recognize a successful step, which made the function keep increasing λ (and thus prioritize gradient-based descent over curvature-based descent), the χ^2 value asymptotically approached the 1588.4366 that the fit made using Prof. Sievers' guess parameters produced. This means that relying purely on gradient decent and taking a large step in parameter space will produce a fit with exactly the same probability of producing the wmap data as the said fit with Prof. Sievers' guesses. If Prof. Sievers used a form of gradient descent to get his guesses, or something equivalent, this would be another indication that my derivatives seem to be working.

Table 1: Table of Fit Parameters

Parameter	Fixed τ	Variable τ
H_0	69 ± 2	69 ± 2
$\Omega_b h^2$	$(2.25 \pm 0.05) \cdot 10^{-2}$	$(2.23 \pm 0.05) \cdot 10^{-2}$
$\Omega_c h^2$	$(1.14 \pm 0.05) \cdot 10^{-1}$	$(1.15 \pm 0.05) \cdot 10^{-1}$
A_s	$(2.04 \pm 0.04) \cdot 10^{-9}$	$(1.87 \pm 0.09) \cdot 10^{-9}$
n_s	$(9.7 \pm 0.1) \cdot 10^{-1}$	$(9.7 \pm 0.1) \cdot 10^{-1}$
τ	0.05	$(4.1 \pm 0.2) \cdot 10^{-1}$
χ^2	1227.92	1227.80

This produced the fit:

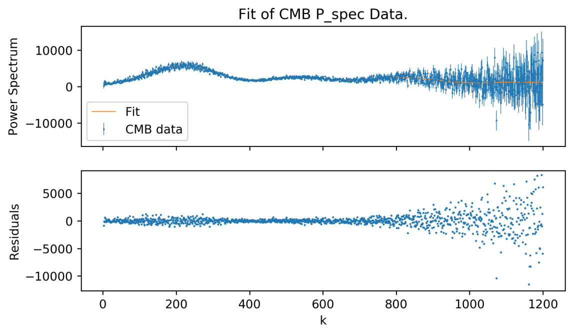


Figure 5: *Fit of power spectrum of WMAP CMB data produced using Levenberg-Marquardt method and holding the optical depth τ to 0.05.*

and the covariance matrix (actual values in "final_fit_hold_tau_flat_cov.txt"):

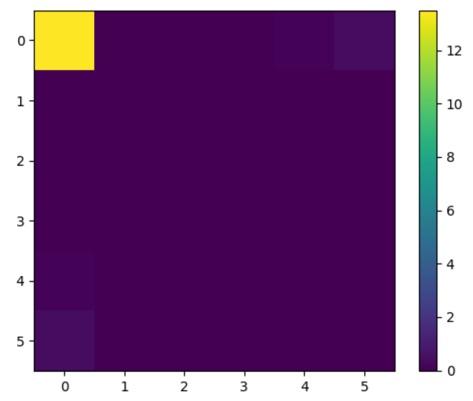


Figure 6: *Covariance matrix of the best fit parameters of the Levenberg-Marquardt fit produced by keeping τ at 0.05.*

As could be expected, the first cell overpowers the others given the magnitude of H0, although we can see some slight correlation between it and τ it seems. If we replace the first column and the first row with the minimum value this might allow us to get a better view of the correlations between parameters:

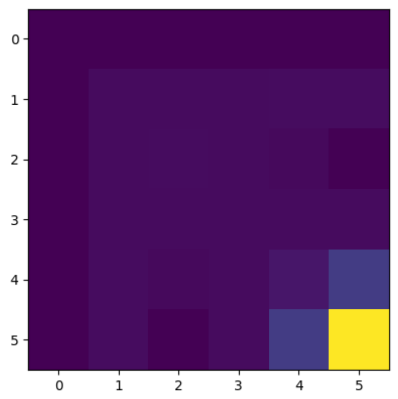


Figure 7: *Same as Figure 6 but the top row and first column have been replaced with the minimum value f the true matrix.*

We can see that the off-diagonal terms are of similar magnitude to the diagonal terms, and so we might expect the parameters of our fit to be correlated to a degree. This correlation might lead us to expect the values of our best fit parameters and their errors to shift when we float τ.

I then performed the task of fitting the data while floating τ. Since we were sort of "pinning" τ and forcing the fitting algorithm to adhere to that, I expected parameters to shift and errors to change, but as we can see in Table 1, the parameters did not shift that much, nor their errors. The freedom to float τ seems to have allowed the fitting function to obtain a "slightly" better fit.

This fit produces the curve:

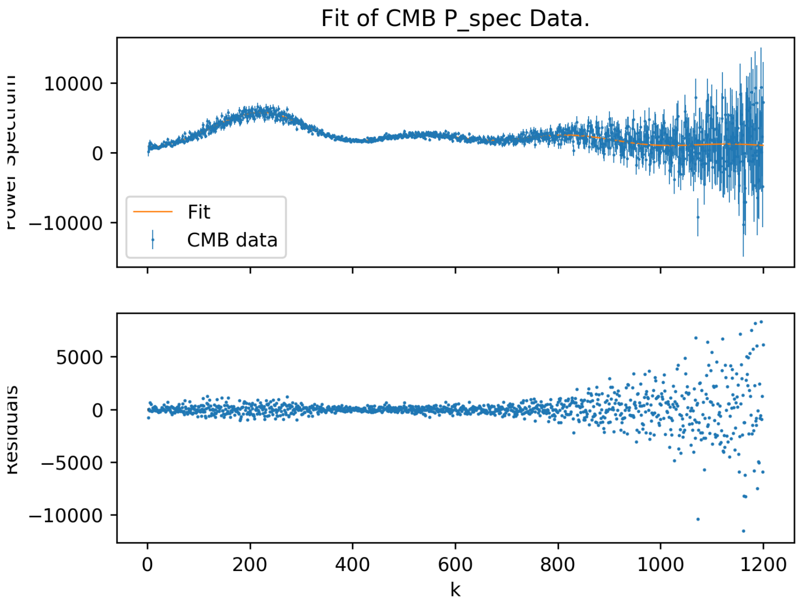


Figure 8: *Fit of power spectrum of WMAP CMB data produced using Levenberg-Marquardt method and floating the optical depth τ.*

and the covariance matrix (with full values in "final_fit_with_tau_flat_cov_mat.txt"):

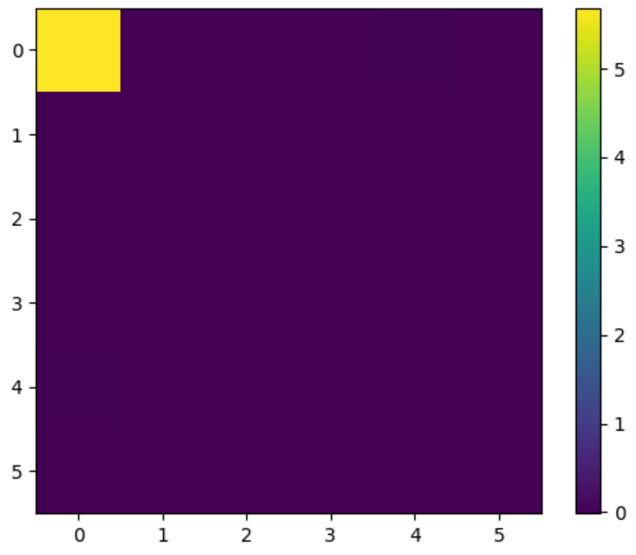


Figure 9: *Covariance matrix of the best fit parameters of the levenberg-marquardt fit produced by fitting all parameters of the CAMB power spectrum.*

3)

Code for this question: *python3 mcmc.py*
Note: For this problem, I did the best I could with the time and resources I had. My computer takes $\approx 30s$ to make one fresh run off the *get_spectrum* function Prof. Sievers gave us, and $\approx 7-10s$ for each run after that in the same program.

As a first example of non-converged chains (and as tests for my program) I ran 550 step chains with simply random steps who's probability of acceptance was discriminated by their change of the χ^2 of the fit and whether or not they prescribed a negative optical depth. This produced chains such as the one for the Hubble constant that can be seen on the left side of Figure 10 (the chains for the rest of the parameters can be seen in the directory "MCMC Simple"). This code corresponds to the commented-out lines 175-218 in *mcmc.py*.

These Fourier-space curves with similar profiles to what can be found on the left side of Figure 11. This Fourier-space spectrum indicates some large-scale structure given that the curve for the smallest inverse-step numbers do not have the same flat profile that a random noise distribution would. This indicates that data on the larger scales of the chain are still correlated somewhat.

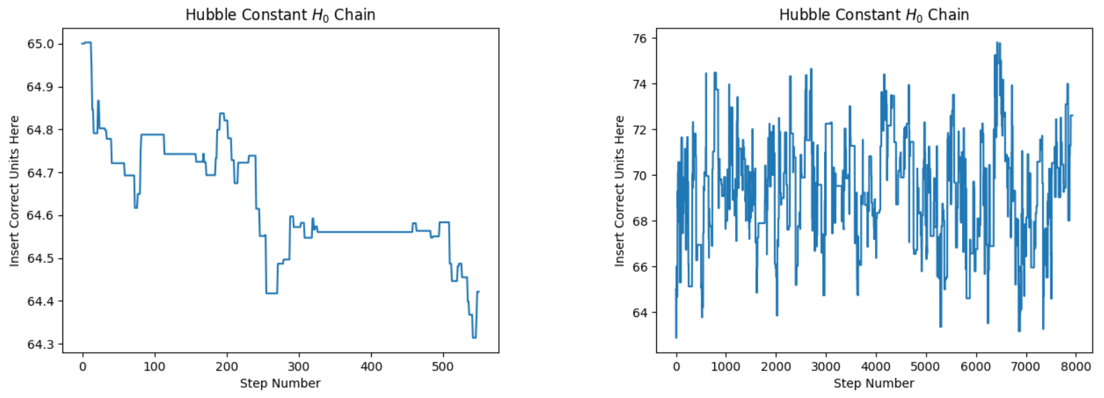


Figure 10: *Left: 550 step H_0 chain produced by an MCMC program taking random steps.*
Right: 7930 step H_0 chain produced by an MCMC program who's steps were informed by the covariance matrix in Figure 6.
In both y-axes the units should be $\frac{km}{Mpc \cdot s}$ since this is the Hubble constant, I just didn't take the time to include variable axes names in my plotting program.

This gave a fit that can be seen in Figure 12 who's parameters can be seen on the left side of Table 2.

Parameter	Random	Covariance-Based
H_0	64.6 ± 0.1	69 ± 2
$\Omega_b h^2$	$(2.17 \pm 0.04) \cdot 10^{-2}$	$(2.24 \pm 0.06) \cdot 10^{-2}$
$\Omega_c h^2$	$(1.22 \pm 0.04) \cdot 10^{-1}$	$(1.14 \pm 0.05) \cdot 10^{-1}$
A_s	$(2.03 \pm 0.01) \cdot 10^{-9}$	$(2.04 \pm 0.04) \cdot 10^{-9}$
n_s	$(9.51 \pm 0.05) \cdot 10^{-1}$	$(9.7 \pm 0.1) \cdot 10^{-1}$
τ	$(3.9 \pm 0.6) \cdot 10^{-2}$	$(5.0 \pm 0.1) \cdot 10^{-2}$
χ^2	1232.25	1227.96

Table 2: *Fit parameters determined by a 550 random-step MCMC chain (left) and a 7930 covariance-based-step MCMC (right)*

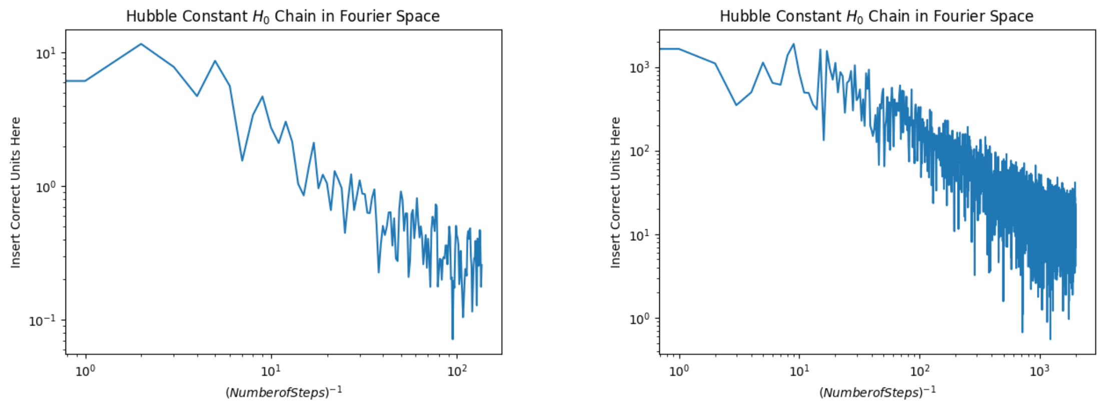


Figure 11: *Left: Power spectrum of the 550 step H_0 chain on the left of Figure 10.*
Right: Power spectrum of the 7930 step H_0 chain on the right side of Figure 10.
In both figures, the DC term has been removed as it represents simply a constant offset that both chains had, presumably in the vicinity of their respective average values.

Once this was done, I then wrote a program that performed the same MCMC process but with sampling in parameter space informed by the covariance matrix provided by are all-inclusive Levenberg-Marquardt fit and displayed in Figure 9. This produced parameter chains of similar profiles as the one for the Hubble constant H_0 on the right side of Figure 10 (the rest can be seen in the directory "MCMC Cov") and Fourier-space curves of similarprofiles to the one displayed on the right of Figure 11. The fit produced can be seen in Figure 13 and its parameters on the right side of Table 2.

We can see, by the comparison of the Fourier-space curves in Figure 11 that the 7930 covariance-based-step process seems to have converged very slightly more than the 550 random-step process, as the latter has a slope even at the smallest values of inverse-number-of-steps (and thus large-scale correlations), whereas the former has a profile that maintains a marginally flatter shape until around $N^{-1} \approx 10^1$ where N is a number of steps in the chain. This effect, despite the covariance-based-step process being an order of magnitude longer (and informed in its parameter-space sampling), is quite small, which could indicate that a larger number of steps is required for definite convergence. Another indicator of this is the chain for τ produced by the convariance-informed MCMC (seen in Figure 14). The chain seems to have undergone a significant shift in the region of parameter space it explores that *appears* to end just over 1000 steps before the end of the chain. This indicates that the MCMC process might not have converged just yet, even though the τ value it ended is compatible with the range given by the Planck measurement. The value I'm basing a good amount of my fit on could be considered a "burn-in" region for a longer fit (and taking a look at Soud Kharusi's work it seems this is very likely the case). An (admittedly terrible) corner plot is provided in "corner_cov_mcmc_2.png" and the highly non-elliptic shape of the heatmaps in parameter space also points towards non-convergence.

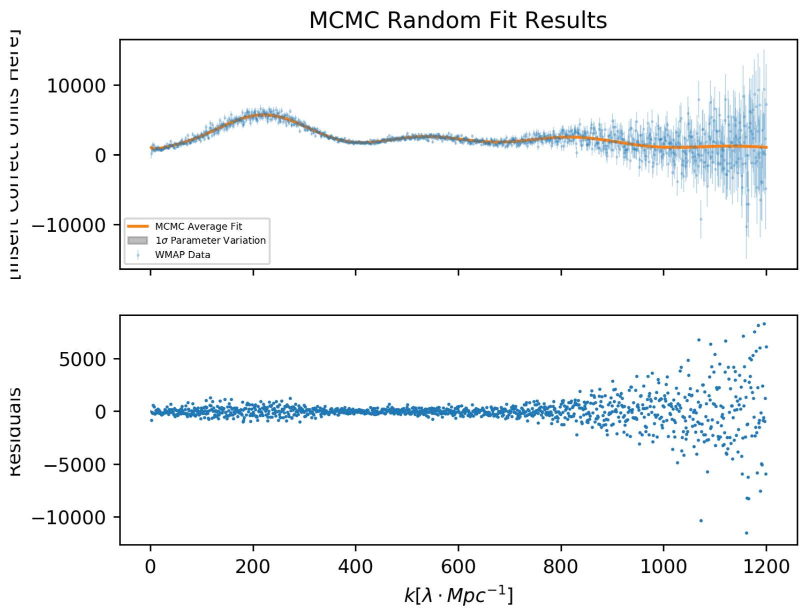


Figure 12: *Best fit produced by a 550 random-step MCMC fitting process, with an imperceptible grey area around the fit delineated by the fits that are produced with* 1σ *deviations in the parameters.*

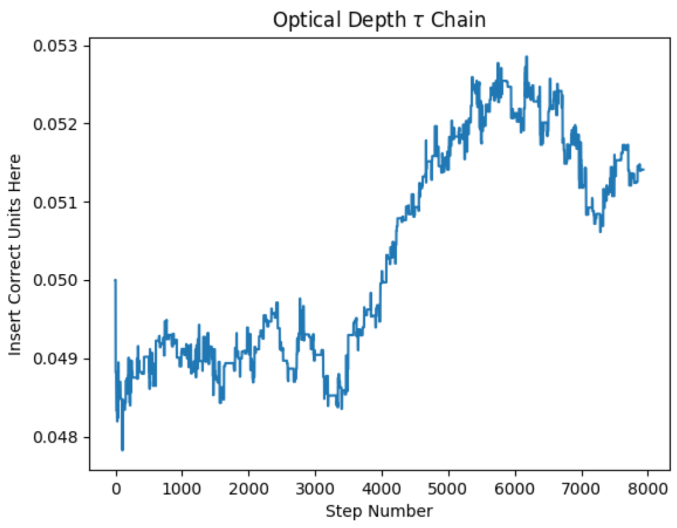


Figure 14: *Chain for the Optical Depth* τ *produced by a 7930 step covariance-based-step MCMC process.*

4)

I first selected all the steps in the 7930 step chain that were compatible with the Planck measurement and found that that was all of them, which was not very interesting. I then combined this with all the steps in my 550 step chain that were compatible with the Planck measurement and... still got the same results (same

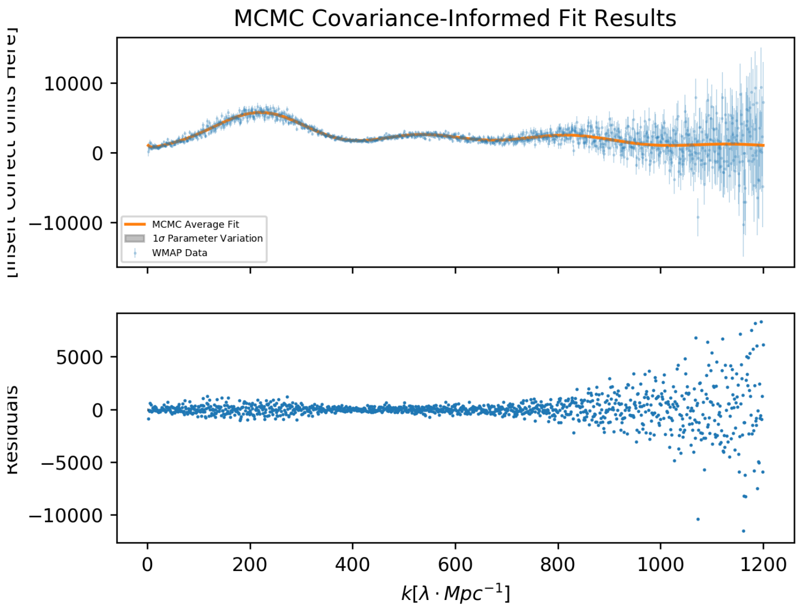


Figure 13: *Fit produced by average of parameter chains produced by a 7930 covariance-based-step MCMC fitting program with an imperceptible grey region delimited by the fits produced by a* 1σ *variation in the fit parameters.*

fit parameters).

I then tried to run a chain where the probability that the nth step was accepted – which was so far $P_{accept}(n) = e^{-\frac{\Delta\chi^2}{2}}$, $\Delta\chi^2 = \chi^2_n - \chi^2_{best}$ – was changed to $P_{accept}(n) = exp(-\frac{\Delta\chi^2}{2}) \cdot exp(-\frac{(\tau_n - 0.0544)^2}{2(0.0073)^2})$. In other words, a step where τ_n was outside the Planck range was discounted in probability the further outside the range it was, and this scaled but the standard deviation of the Planck measurement. This, however, lead to chains with dismal acceptance rates (less than 5%). This probablity distribution, although slightly, any τ_n values that were not 0.0544, even though they might still be within the range. So I then also tried $P_{accept}(n) = exp(-\frac{\Delta\chi^2}{2}) \cdot exp(-\frac{(\tau_n - 0.0544)^2}{2(0.0073)^2} + \frac{1}{2})$, which doesn't "punish" any τ_n values inside the Planck-compatible range, and even "rewards" them the closer to 0.0544 they are, and even that produced fits with dismal acceptance rates. Remnants of these attempts can be found commented out on line 129 and in the *tau_fctr* argument of my *mcmc* function in my *mcmc.py* file. Results/chains of these trials can be found in the "Restricted Tau" directory.

At this point, I severely ran out of time and had to abandon the idea of finishing this question to a satisfactory degree.

Apology

I apologize if this submission ends up being difficult to correct. This assignment really took a lot from me, which I know might sound like a blowing out of proportion of a single assignment, but really with the resources I had at hand, and the fact that I got stuck at question 2 on a dumb coding bug for a few days despite showing my code to multiple people, and the time it took to run even small MCMC chains to see if they were viable, I just really ran out of time and often got severely discouraged.

If you require any clarification on my code or submitted files, don't hesitate to contact me, I'll be happy to help out as best I can!