Draft version December 11, 2018
Typeset using LaTeX default style in AASTeX62

Using Markov Chain Monte Carlo to test properties of Nebular Emission

Christopher Agostino

1. INTRODUCTION

Astrophysical Nebulae provide unique opportunities to study an array of topics, ranging from atomic physics to star formation to the accretion and evolution of black holes. Astrophysical nebulae are typically denser than the rest of the interstellar medium in galaxies and are being actively ionized by newly formed massive stars, white dwarfs (planetary nebulae), or black hole accretion disks. These nebulae are typically hot ($\sim$ 10000 K) and the majority of their emission comes either in the form of radiative de-excitation of collisionally excited forbidden lines, like [OIII]λ5007 or [NII]λ6583, or in the form of Hydrogen Balmer recombination lines, where these lines will be emitted as hydrogen electrons recombine with protons and they de-excite to the ground state.

As part of my research here at IU, I use nebular emission lines to test whether external galaxies have their nebulae primarily heated by star-formation or black hole accretion in the form of an active galactic nucleus. In doing this, I also am looking at the reliability of these selection methods for finding or removing galaxies with active galactic nuclei, and I wanted to use this computational physics project to explore an aspect of my research subfield that I have not yet had the chance to investigate. For many years, this field of research has relied on empirical relationships to derive physical properties of ionizing sources (Baldwin et al. 1981; Veilleux & Osterbrock 1987; Kauffmann et al. 2003) but advances in computations, have increased our ability to compute properties of nebulae from first principles. The software `CLOUDY` has the ability to compute emission line spectra using initialized properties and these computed spectra can then be used to compare with real, observed data (Ferland et al. 2017).

Oftentimes, it is possible and useful to fit a real emission line spectrum using software like `CLOUDY` and being able to do so provides a method for determining the underlying physical properties that are producing the observed emission line spectrum. However, there are many properties that go into computing the emission line spectrum in `CLOUDY`, including the number density of the cloud and the temperature of the ionizing source and there are even more parameters that can be tweaked than one can feasibly explore in their own lifetime. Fortunately, though, our computers allow us to explore high-dimensional parameter spaces using advanced methods. One of these techniques, Markov-Chain Monte-Carlo (MCMC), is the driver of this final project. Specifically, I wanted to explore nebular emission line properties using MCMC and `CLOUDY` so that I could one day possible use this software with my own research.

There are already a number of MCMC programs written in Python which can accomplish what I would like to, but I have always wanted to write my own so that I can better understand what is going on with it. For that reason, I have written an implementation of an MCMC using the popular Metropolis-Hastings algorithm. I describe this algorithm in detail in Section 2.1 and then test it on several simple cases in Section 4. Lastly, I show preliminary results from my Cloudy tests in Section 3. Unfortuantely, due to the extensive calculations performed by `CLOUDY` and thereby its excessive runtime ($\sim$ 40 s/model), I was unable to run all of the tests I had hoped to do.

2. MARKOV CHAIN MONTE CARLO

In Bayesian statistics, it is often useful to estimate to posterior distribution. This task, however, is often intractable due to the high-dimensional integral that is involved in calculating the marginal likelihood from each of the relevant parameters. With regular Monte Carlo integration, samples need to be drawn from the posterior distribution which is problematic if the posterior distribution is not easily computed, as is the case with high-dimensional data.

Markov Chain Monte Carlo methods allow us to draw samples from a distribution such that each sample depends only on the state of the previous one. These samples form a so-called Markov chain. Not all samples are used as an acceptance criteria is created by comparing the successive states with respect to some target distribution to ensure the distribution of samples mimics the posterior distribution of interest. This target distribution must be proportional to the posterior distribution, meaning that the full calculation of the likelihoood is unnecessary and can be seen as just some normalizing constant. Thus, the posterior distribution is proportional to the likelihood distribution multiplied by the prior distribution. After a number of samples, the Markov chain will converge to a stationary distribution and these samples can be used as correlated draws from the posterior distribution.

2.1. *Metropolis-Hastings Algorithm*

There are a number of algorithms which implement a version of MCMC but in this project, I will focus on implementing the Metropolis-Hastings algorithm. This algorithm works by generating a sequence of sample values in a way that the distribution of values more closely approximates the target distribution as the number of samples used increases. Each sample depends only on the previous one and at each step, the algorithm chooses the next sample. The future values are chosen by comparing the current and candidate sample values with respect to the target distribution. These sequences form a so-called 'chain' and many chains are initiated at different positions in the parameter space so the underlying probability distribution can be approximated without using methods that might require much more computing time as fine grids do in high dimensions.

The exact algorithm is written succinctly in the following 5 steps, as was shown in my project outline:

1. A state, s_0, is selected at $n = 0$.
2. Randomly pick a candidate state s_{n+1} according to the proposal distribution $p(s_{n+1}|s_n)$
3. Calculate the acceptance probability using the so-called Metropolis choice:

$$A(s_{n+1}, s_n) = \min\left(1, \frac{P(s_{n+1})p(s_n|s_{n+1})}{P(s_n)p(s_{n+1}|s_n)}\right) \tag{1}$$

   where P(s) corresponds to the desired distribution.
4. Now that a candidate state's acceptance probability is calcualted, determine whether or not to accept or reject it by generating a uniform random number $\chi \in [0, 1]$ and accept state s_{n+1} if $\chi \leq A(s_{n+1}, s_n)$ or reject it if $\chi > A(s_{n+1}, s_n)$. If the new state is rejected, copy the old state forward, $s_{n+1} = s_n$.
5. Set $n = n + 1$ and repeat steps 2-5 for a set number of iterations.

In my implementation, the desired distribution is characterized by χ^2 values as I am trying to determine fitting parameters. My Metropolis Hastings code is implemented in the code shown in the `run_chain` method shown in Section 6. The overall MCMC implementation is contained within `run_mcmc`, where the many different chains are generated and stored into attributes which can be used for further analysis. In addition, the final output parameters are computed as being the mean of all of the samples weighted by their likelihood. My particular implementation of the Metropolis-Hastings algorithm seeks to maximize the negative likelihood, which is why the `chisq` function in Section 6 has a negative sign in front of the sum of the square of the residuals. My implementation of step 4 is slightly different but the principle is still the same. I also implemented a functionality such that if the randomly chosen new state is outside of the limits specified by the user, the method will simply continue onto the next random state and not even consider that point.

3. `CLOUDY` INTERFACE

The python package, `pyCloudy`, can be used to interface directly with `CLOUDY` to create and run models to synthesize spectra from nebulae, given certain conditions for the nebulae like temperature, density, and composition. One can also adjust the source of heating for nebulae in Cloudy, which can allow one to test the response of nebulae to different sources of heating like hot stars or black hole accretion disks. In section 7, I show the code which is used to compute `CLOUDY` models. To set this up, I actually followed one of the tutorials from the `pyCloudy` website for initializing a model. I was able to reproduce the graph from their first tutorial and show it here in Figure 1. I left out most of the lines that were used in that tutorial in favor of saving time and use this model as a basis for creating a model spectrum that I can use for fitting. The specific parameters (number density, effective temperature, number of ionizing photons per second, etc.) are initialized in the code shown in Section 8.

4. MCMC TESTS

4.1. *Quality Tests*

I first began testing my MCMC method by testing how it would fare in computing fits to polynomials, which are quite simple compared to emission line spectra. I created linear, quadratic, cubic, quartic, and quintic polynomials using the parameters shown in Section 8. I also tested one sinusoidal model and one model which combined a sinusoid

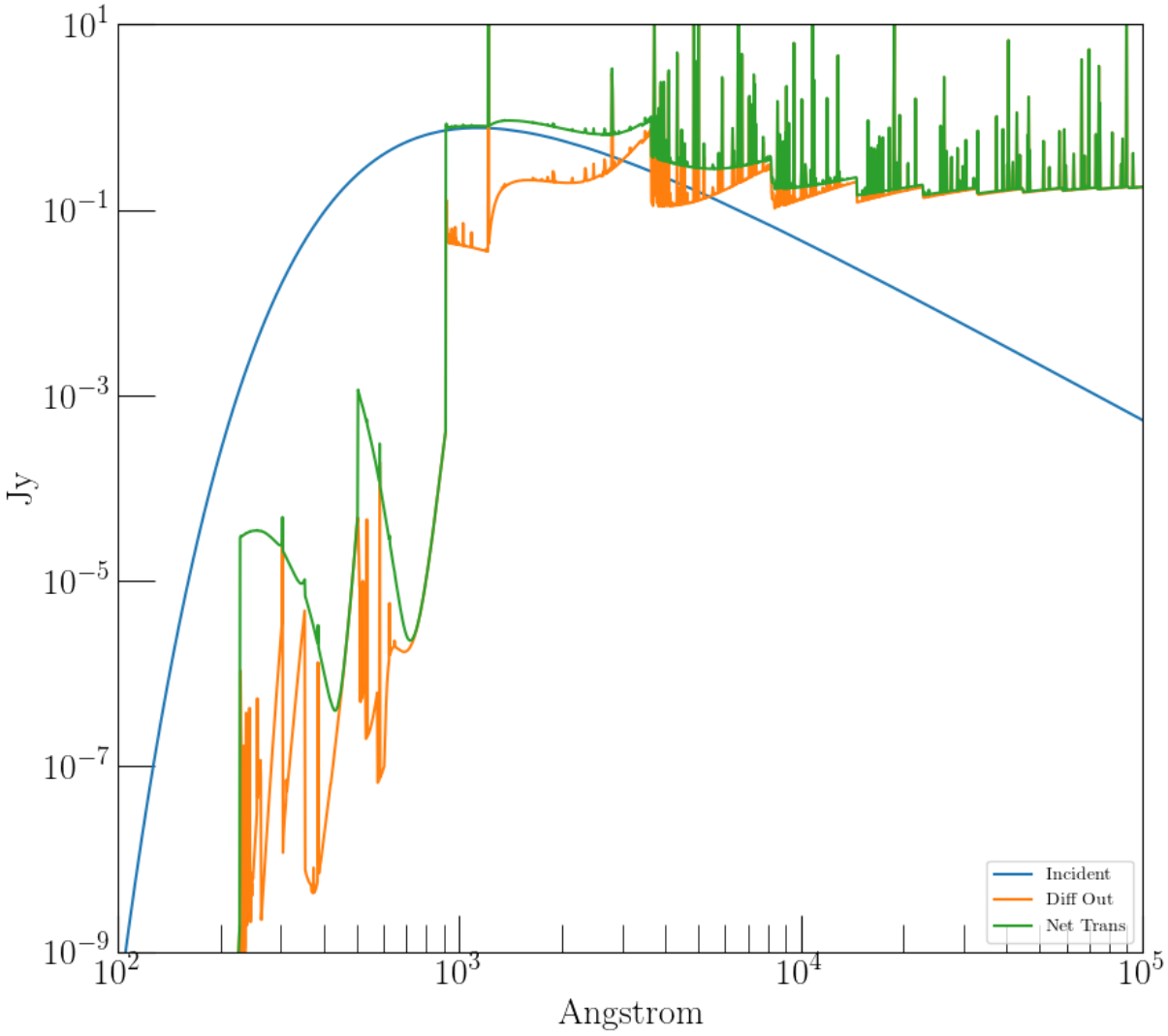

Figure 1. Output of Cloudy shows spectra of various parts involved in the production of nebular emission. The blackbody spectrum of the ionizing star is shown in blue while the net emission out is shown in green and the differential out is shown in orange.

and a polynomial. I then used my MCMC method to determine the fit parameters, using 10000 for the number of chains and 100 for the number of iterations in each chain. The resultant fits for these seven models are shown in Figures 2, 4, 6, 8, 10, 12, and 14. Their fit parameters are tabulated alongside their input parameters in Tables 1 – 7. I decided to add the sinusoid+polynomial fit because I figured it would be considerably more difficult for the MCMC to determine than the other fits and found this to be true. The fit shown in Figure 14 is by far the most different from its original data among the set of tests I ran.

For the most part, the MCMC method is able to find relatively good solutions with fairly accurate results for the highest order terms. Nevertheless, I wanted to see how these parameters spread against each other in terms of their likeliness to investigate whether or not parameters might be correlated with each other. For that reason, I use the `corner` package to plot the likelihood as a function of the individual parameters plotted against each other. These

so-called 'corner' or 'triangle' plots, which are quite popular among MCMC users, are shown in Figures 3, 5, 7, 9, 11, 13, and 15. The likelihoods of the highest order parameters are usually well concentrated to a specific region of parameter space with respect to the other parameters. The lower-order parameters actually spread in seemingly nonsensical ways against each other because when they change a lot there is not much of an effect on the overall fit, contrary to what I see with the higher order terms. The parameter spaces for the polynomial+sinusoidal model are not well-constrained, suggesting that it was a difficult problem for the MCMC to solve and would require more samples or chains if a more accurate solution is desired. It should be noted that the upper and lower limits on the parameters in the corner plots are not the 1-σ level errors but are computed as the 14% and 86% quantiles. This was suggested in the tutorial for the corner package. I do not quote these values as errors on my values in Tables 1–7 but I leave them on the corner plots because I think they are instructive regarding the majority of the distribution.

Source	a_1	a_2
Original	3.7	2.2
Fit	3.71	1.95

Table 1. Linear Fit parameters versus input.

Source	a_1	a_2	a_3
Original	-1.1	2.3	-3.5
Fit	-1.09	1.84	3.21

Table 2. Quadratic Fit parameters versus input.

Source	a_1	a_2	a_3	a_4
Original	-2.1	2.3	4.5	-10
Fit	-2.10	2.38	1.89	-8.01

Table 3. Cubic Fit parameters versus input.

Source	a_1	a_2	a_3	a_4	a_5
Original	4.2	1.1	-4.3	-2.1	14
Fit	4.17	1.04	0.36	0.08	1.25

Table 4. Quartic Fit parameters versus input.

Source	a_1	a_2	a_3	a_4	a_5	a_6
Original	-0.8	-6.2	1.4	3.2	2.1	4.8
Fit	-0.79	-6.10	-0.15	-2.17	0.79	2.95

Table 5. Quintic Fit parameters versus input.

Source	a_1	a_2
Original	1.1	0.7
Fit	1.14	0.70

Table 6. Sinusoidal Fit parameters versus input.

Source	a_1	a_2	a_3	a_4
Original	2	4	2	2.2
Fit	1.54	4.04	2.00	1.99

Table 7. Linear+Sinusoidal Fit parameters versus input.

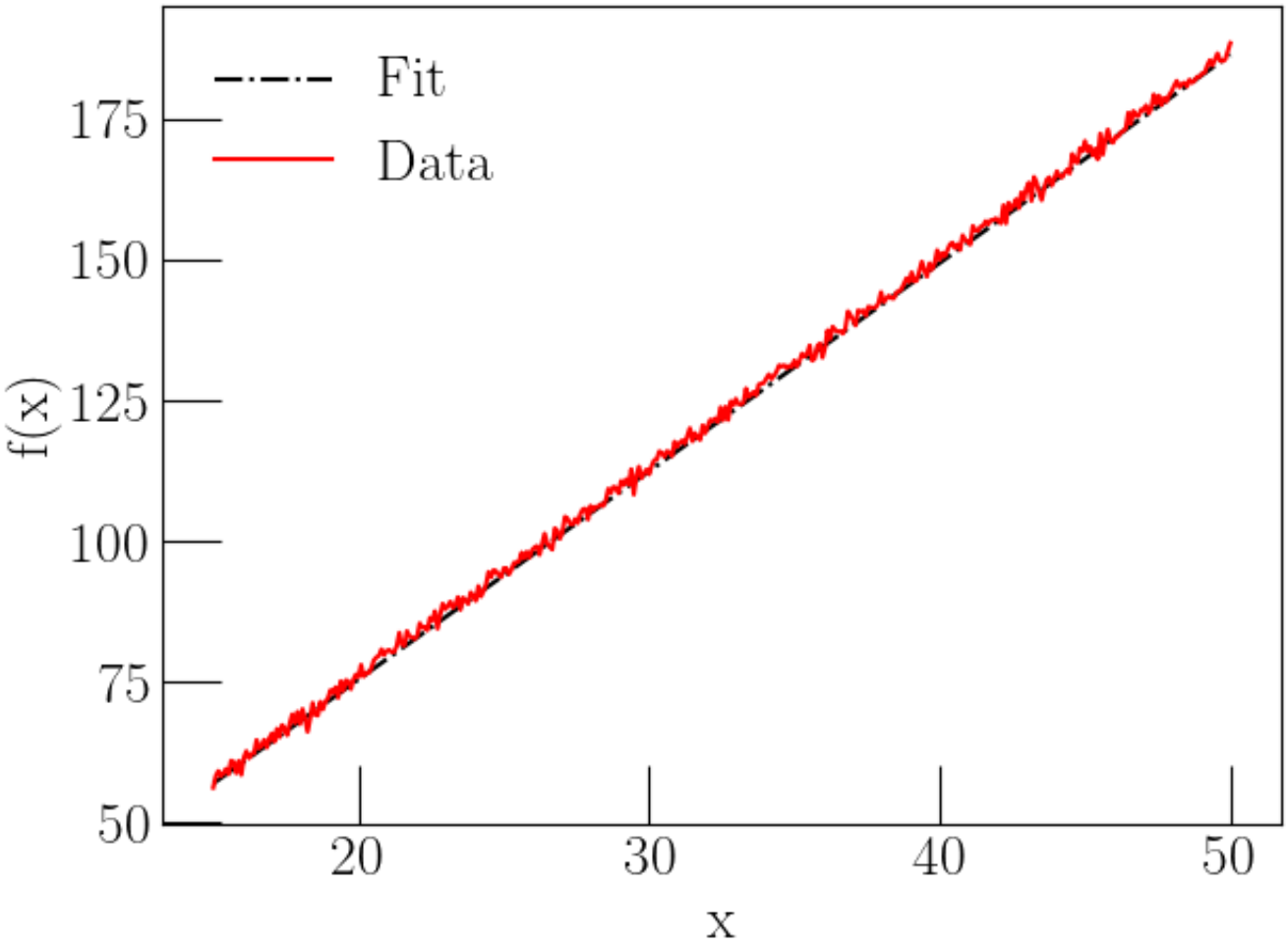

Figure 2. Linear MCMC fit of data with Gaussian Noise.

4.2. *Algorithm Reliability Testing*

After running these initial polynomial tests, I decided to run some tests concerning the number of chains used by the MCMC method, keeping the number of iterations in each chain fixed at 100. This code is shown in Section 8. I first tested the runtime versus number of chains and found that the runtime scaled roughly linearly with the number of chains used by MCMC. This relationship is shown in Figure 16. I used an array of chain numbers that logarithmically spanned the range $10^2 - 10^5$. After this, I wanted to test convergence. Using a linear model with the input parameters shown in Table 1, I computed the fit slope and y-intercept with the different numbers of chains as was used for the runtime. These experiments are shown in Figures 17 and 18 for the slope (m) and y-intercept (b) values, respectively. The MCMC essentially converges for the slope at $\sim 10^{3.7}$ whereas it never approaches it for the y-intercept. Another test of convergence I did was to look at the minimum χ^2 value among the samples and I show this versus number of chains in Figure 19. This test shows that the minimum χ^2 value does not really change much after log(N)≈3, suggesting that this may be an adequate number of chains to use for linear fitting.

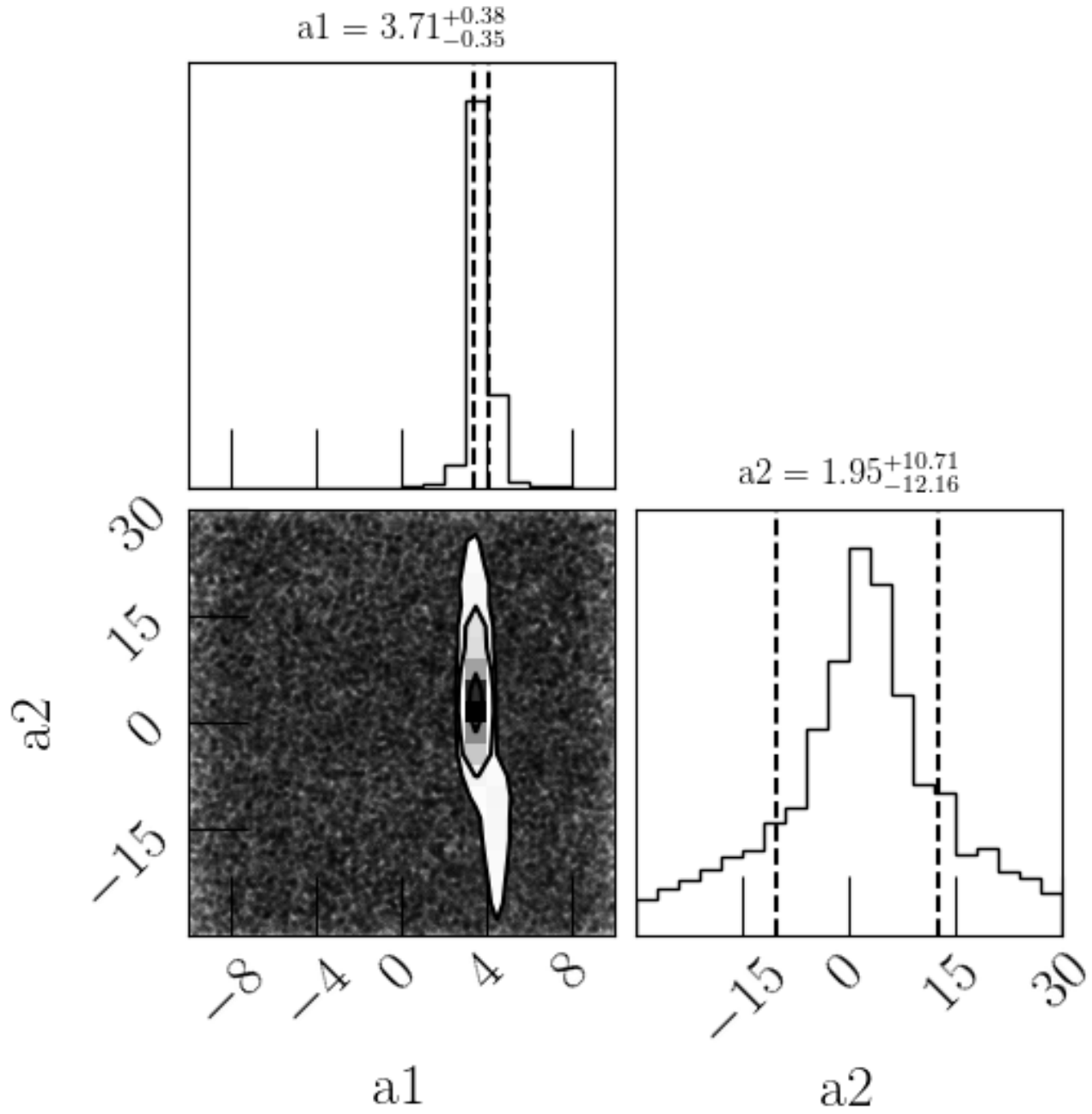

Figure 3. Corner plot of MCMC samples and 14%, 86% quantiles for linear fit.

I then repeated these same experiments with the number of iterations used in each chain, keeping the number of chains fixed at 10^3 because of these previous experiments. I again find that the runtime scales roughly linearly with the number of iterations in each chain, as shown in Figure 20. There does not seem to be an obvious trend of convergence for either the slope value or the y-intercept value, as shown in Figures 21 and 22, respectively. The minimum χ^2 values actually converge here at about log(N)$\approx$ 1.5. Despite this, I decided to keep the number of iterations in each chain at 100.

4.3. *CLOUDY Testing*

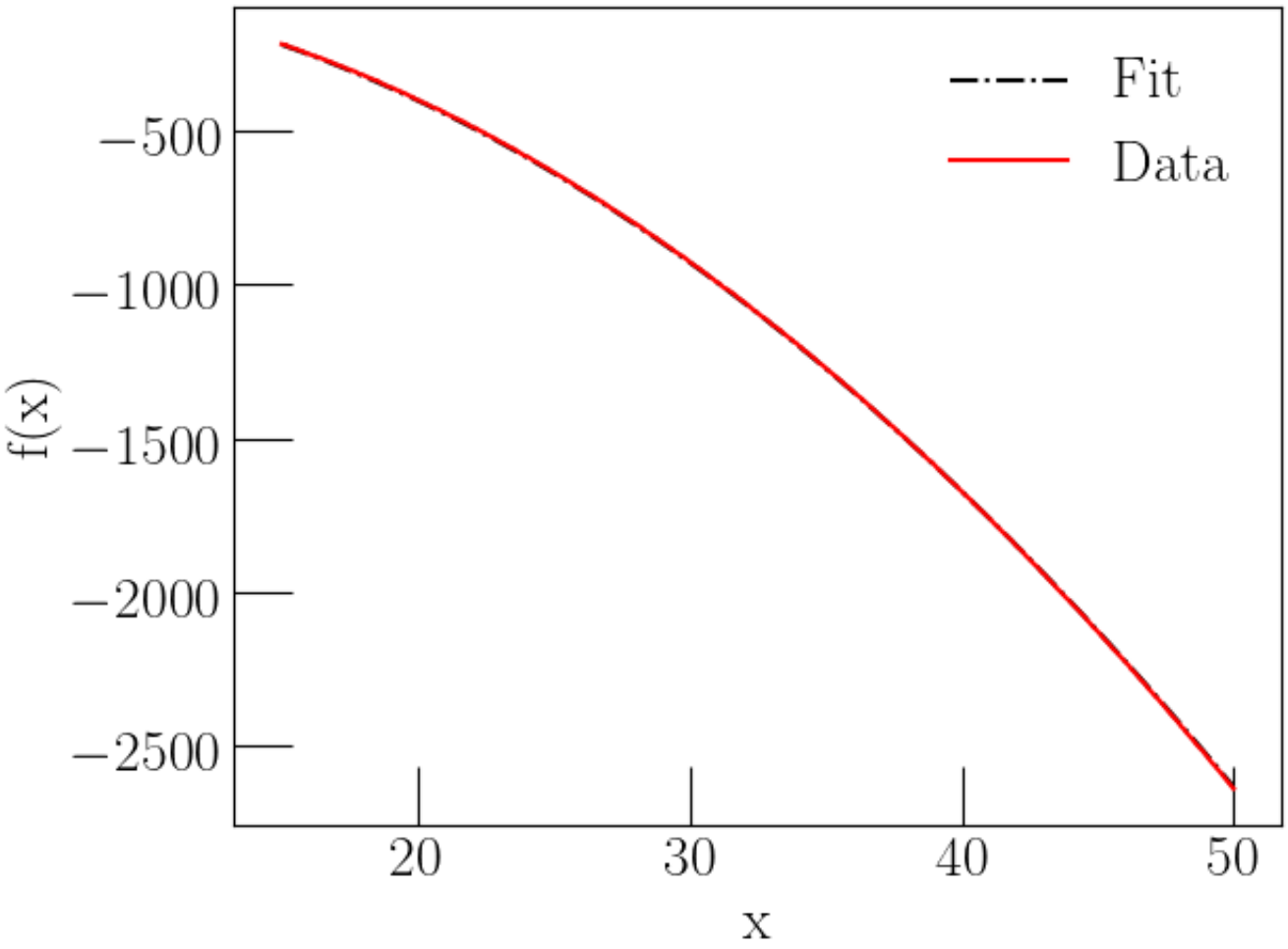

Figure 4. Quadratic MCMC fit of data with Gaussian Noise.

With these various tests done, I then attempted to use my MCMC to fit an emission-line spectrum produced by `CLOUDY`. Unfortunately, I severely underestimated the amount of time required to compute each individual `CLOUDY` model, which on average was ~ 40 seconds. I had originally planned to use 10000 chains and 100 iterations in each chain to fit an emission line spectrum, but because I was pressed for time and did not have $10000\times\ 100\times\ 40$ seconds left, I went for a very rudimentary MCMC run, using only 10 chains and 10 iterations. I'm currently computing a more ambitious version with 100 chains and 10 iterations to get a better approximation of the distribution but as of this time, I have been unable to compute exactly what it was that I wanted to do.

In either case, I do have a fit done, even if it is quite bad. This fit is shown in Figure 24 along with the transmission emission-line spectrum from Figure 1. In figure 24, I have restricted the region of the spectrum only to the optical regime ($\sim 4000 - 8000\mathring{A}$) and only used this region to perform the fitting. The fit assigned is definitely not the best but certainly is not absolutely atrocious, so it seems promising that this MCMC method might be able to produce some promising results in the future for me in my research in fitting emission-line spectra. When it is finished running, I do plan to submit the final result alongside this report, though I do not expect it to factor into the grading.

5. CONCLUSIONS AND FUTURE WORK

While I was not able to accomplish my ultimate goal, I gained a lot of valuable experience in doing this project. My original plan in my outline was to test distributions of nebular parameters and how they correspond to different observables, but in starting the project I realized that that would be better accomplished by some sort of gridding attack or Monte Carlo simulation and would not necessarily warrant a full MCMC method. This realization was

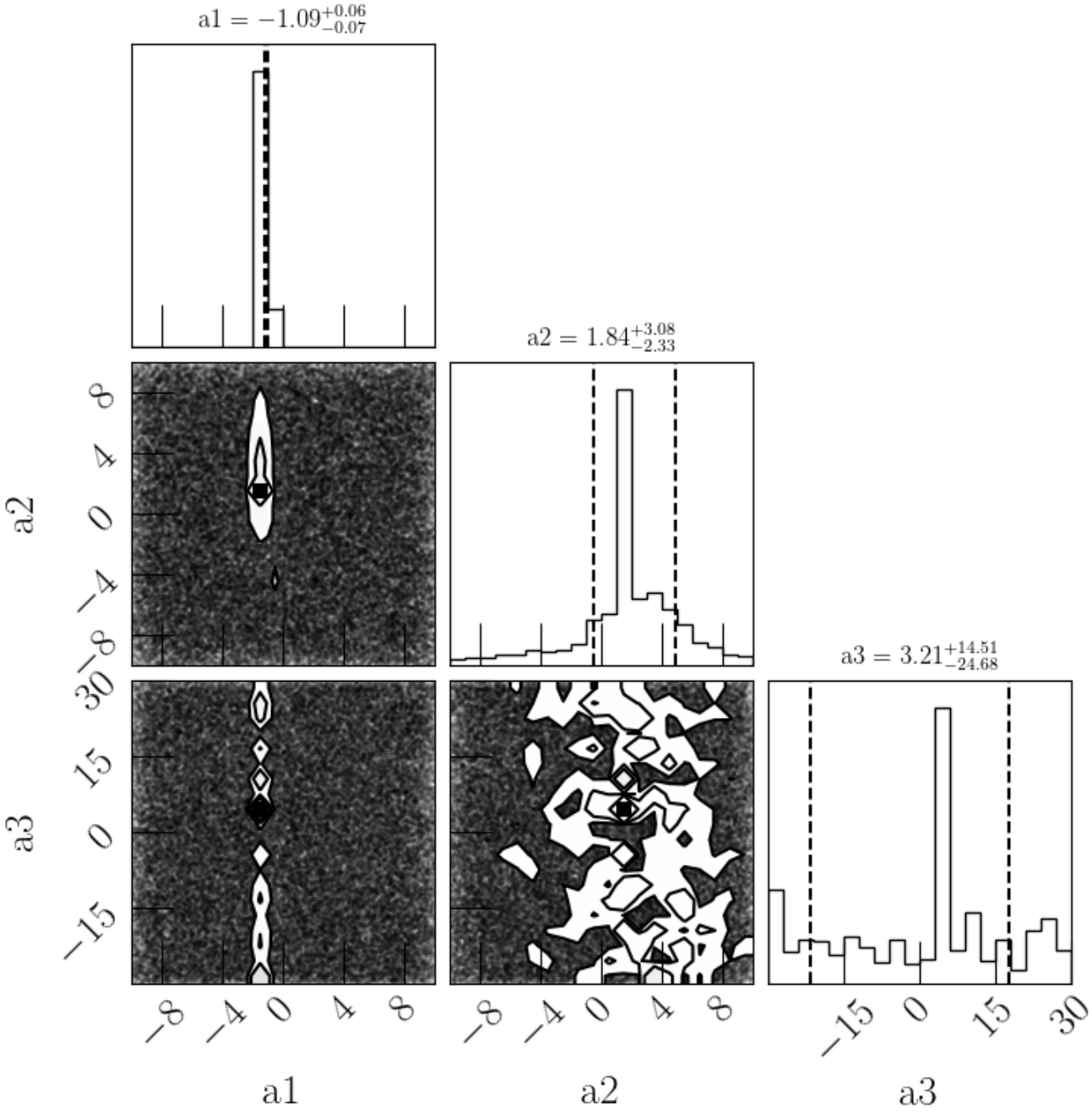

Figure 5. Corner plot of MCMC samples and 14%, 86% quantiles for linear fit.

what prompted me to switch to using MCMC to perform curve fitting rather than for simple sampling and modelling, though I may reconsider this when I take another look at what exactly I want to accomplish in my future research endeavors. There are certainly a number of ways in which I could improve my MCMC code. First, I would like to add a parallelization feature, so that it can run different chains on different cores to save a moderate amount of time. I also would like to add a more careful method to determine the initial position of each of the chains so that it will do a better job at parameterizing the whole distribution. I am also certain that I could get the runtime of `CLOUDY` models down substantially but I ran out of time before I could tinker enough with the specifics of the software. Realistically,

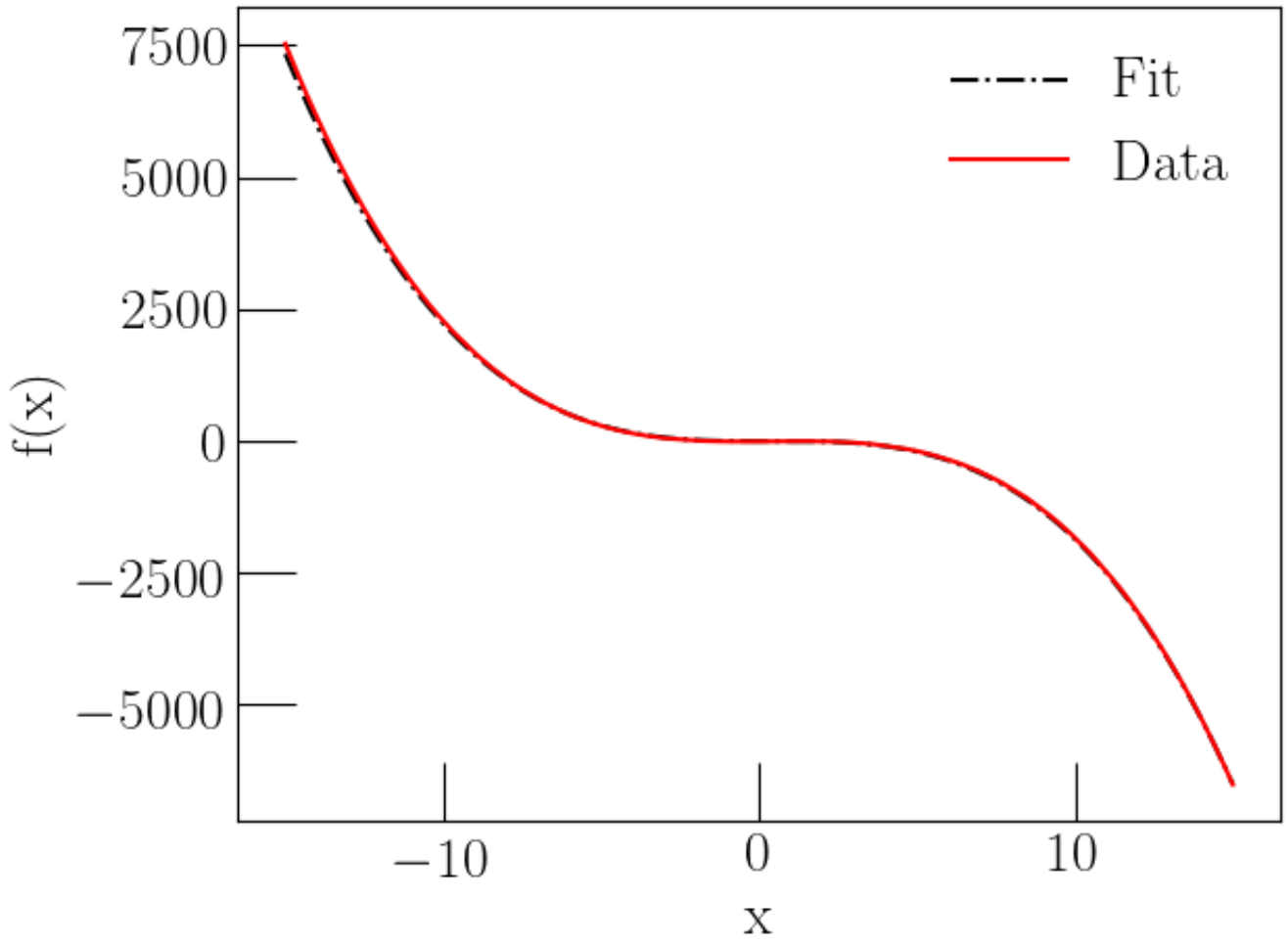

Figure 6. Cubic MCMC fit of data with Gaussian Noise.

I will likely use a well-tested MCMC package in Python with optimized sampling, like `emcee`, if I eventually end up pursuing this in a serious capacity.

REFERENCES

Baldwin, J. A., Phillips, M. M., & Terlevich, R. 1981, PASP, 93, 5

Ferland, G. J., Chatzikos, M., Guzmán, F., et al. 2017, RMxAA, 53, 385

Kauffmann, G., Heckman, T. M., Tremonti, C., et al. 2003, MNRAS, 346, 1055

Veilleux, S., & Osterbrock, D. E. 1987, ApJS, 63, 295

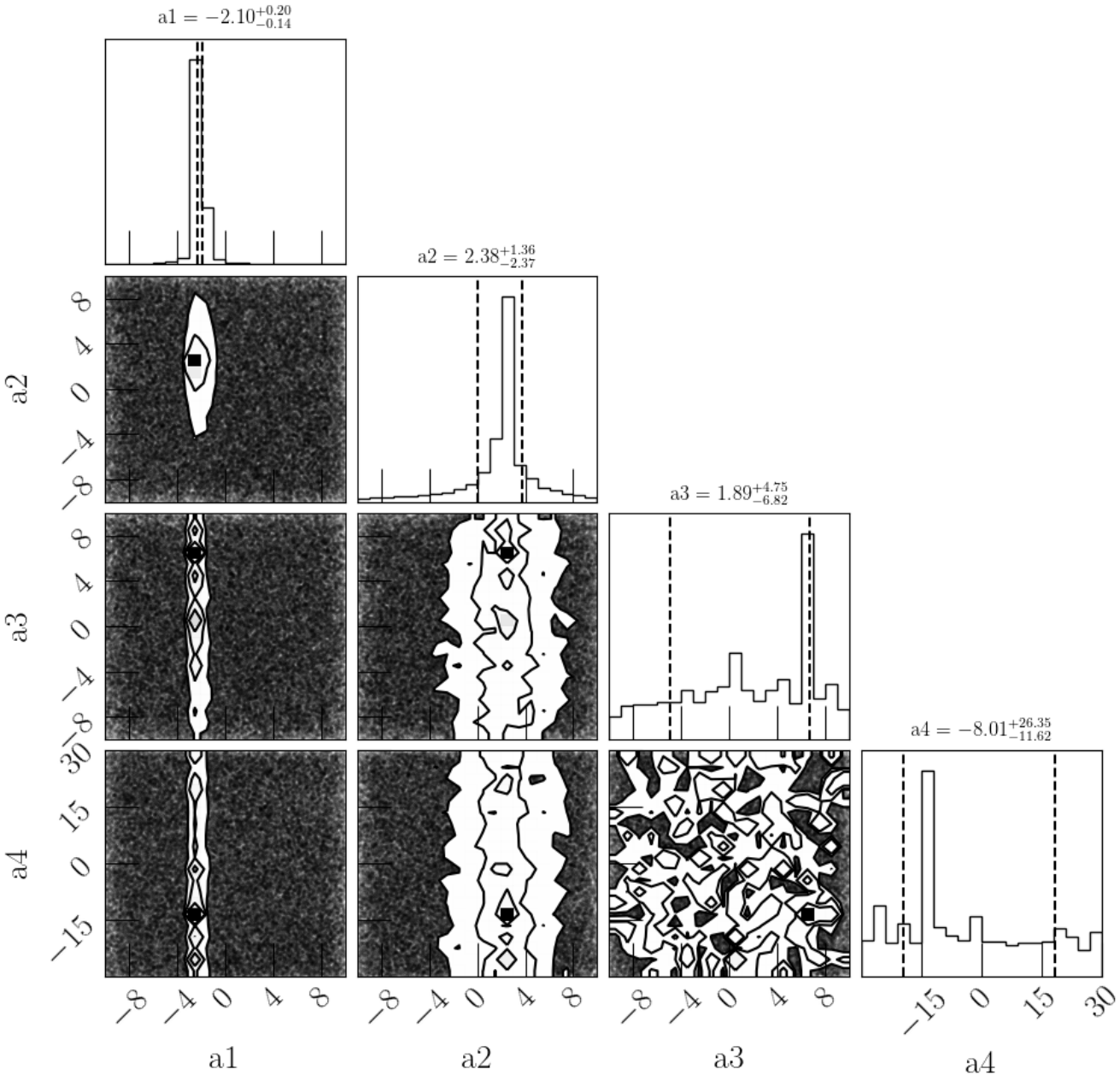

Figure 7. Corner plot of MCMC samples and 14%, 86% quantiles for linear fit.

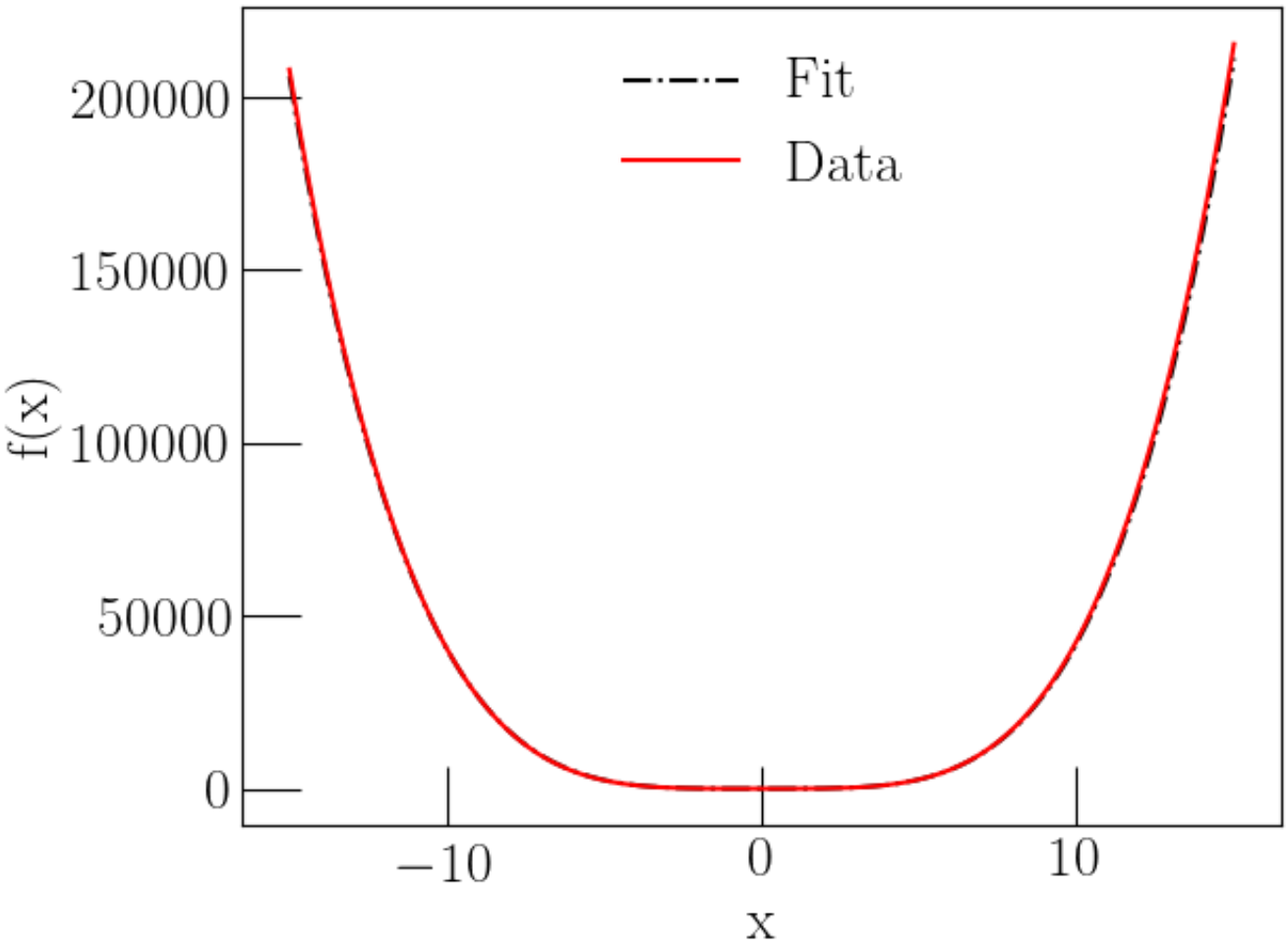

Figure 8. Quartic MCMC fit of data with Gaussian Noise.

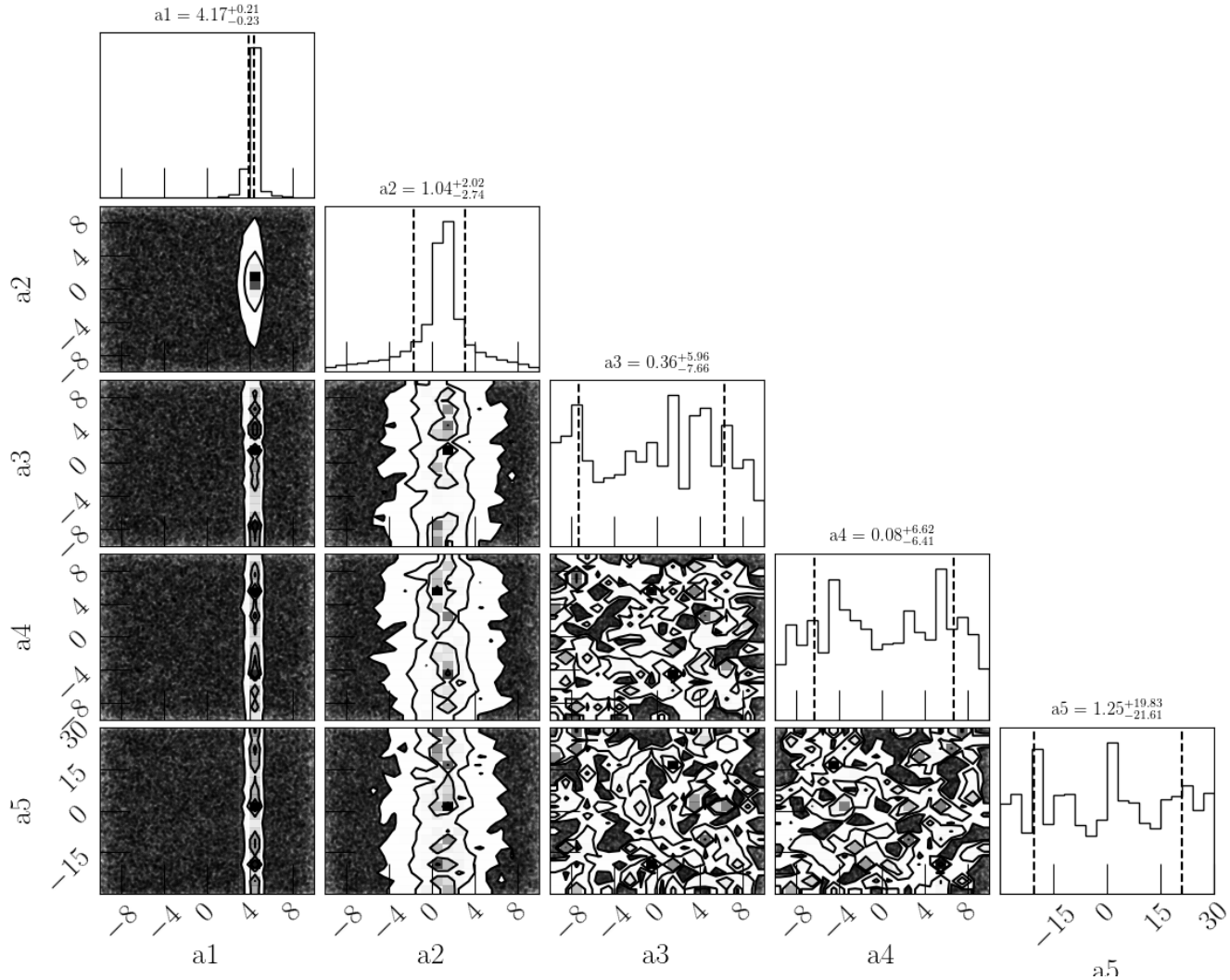

Figure 9. Corner plot of MCMC samples and 14%, 86% quantiles for linear fit.

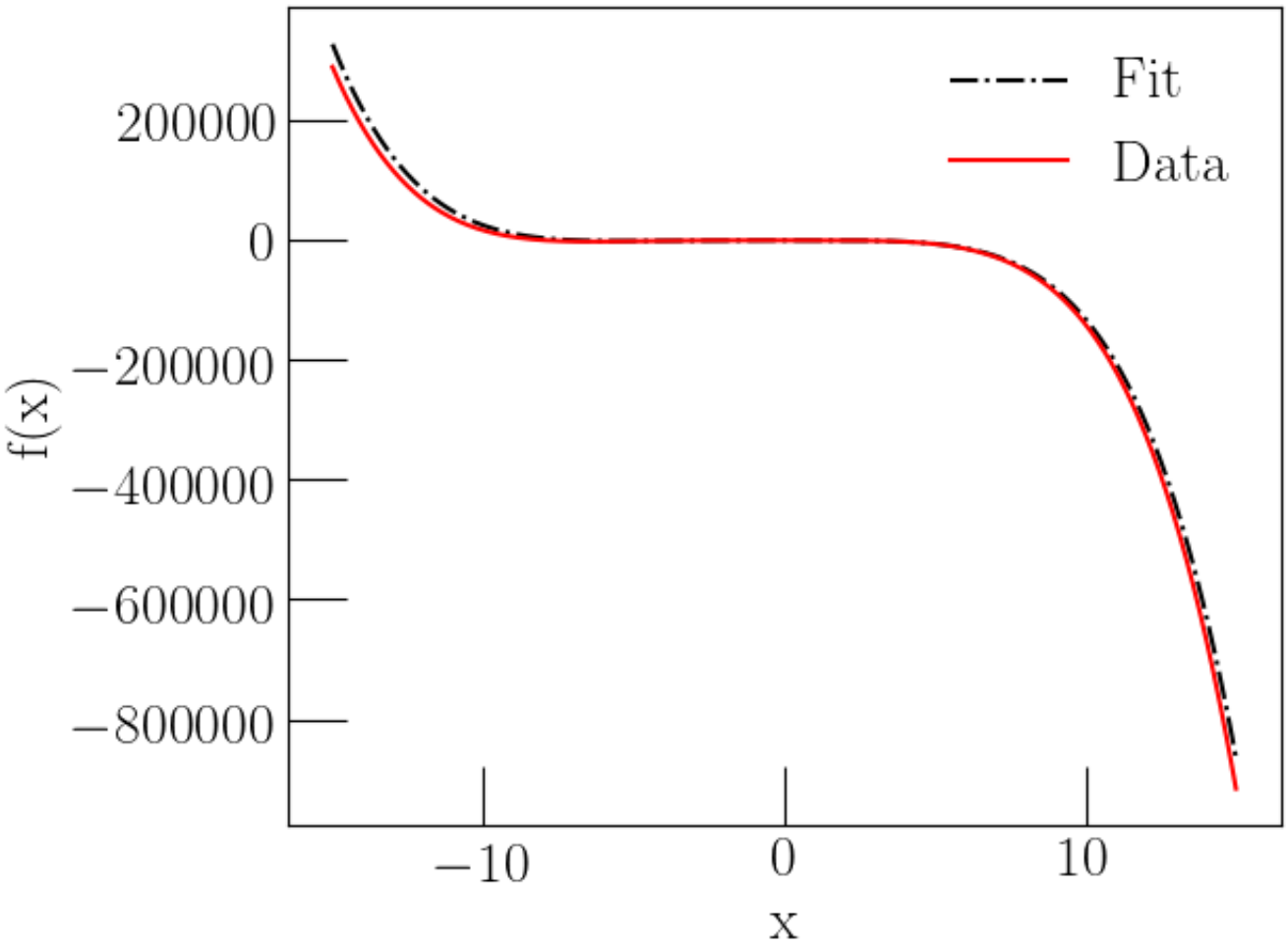

Figure 10. Quintic MCMC fit of data with Gaussian Noise.

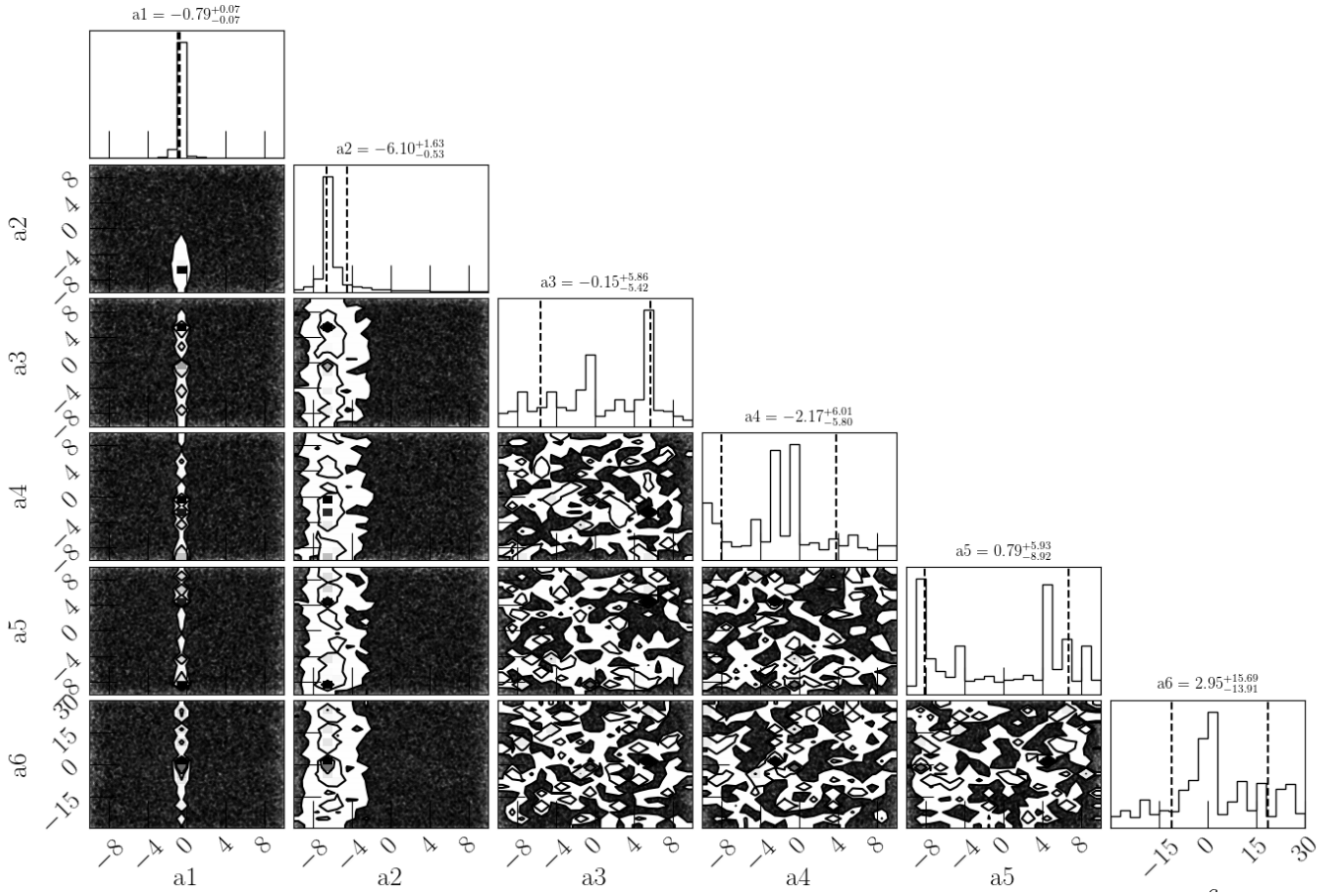

Figure 11. Corner plot of MCMC samples and 14%, 86% quantiles for linear fit.

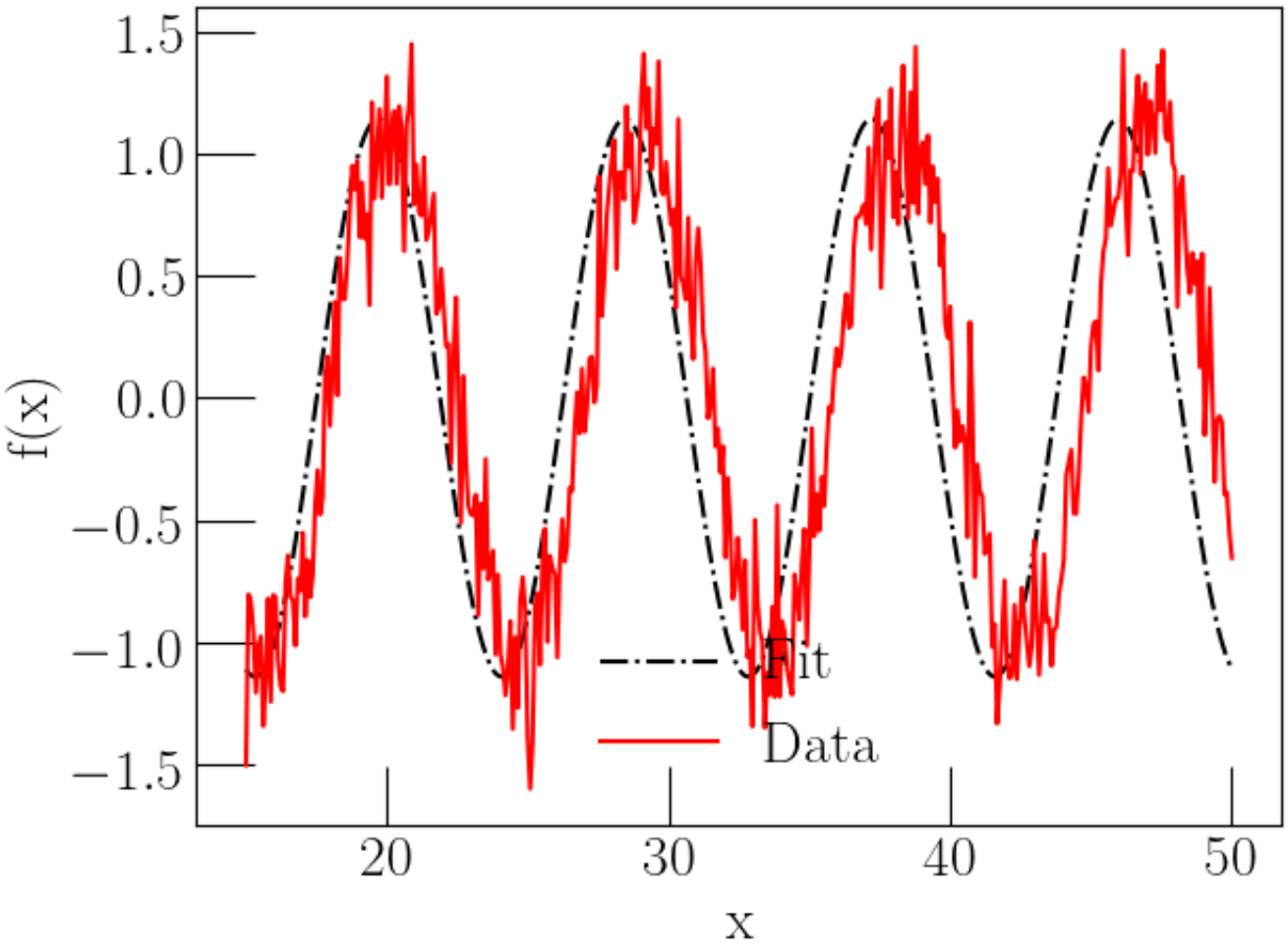

Figure 12. Sinusoidal MCMC fit of data with Gaussian Noise.

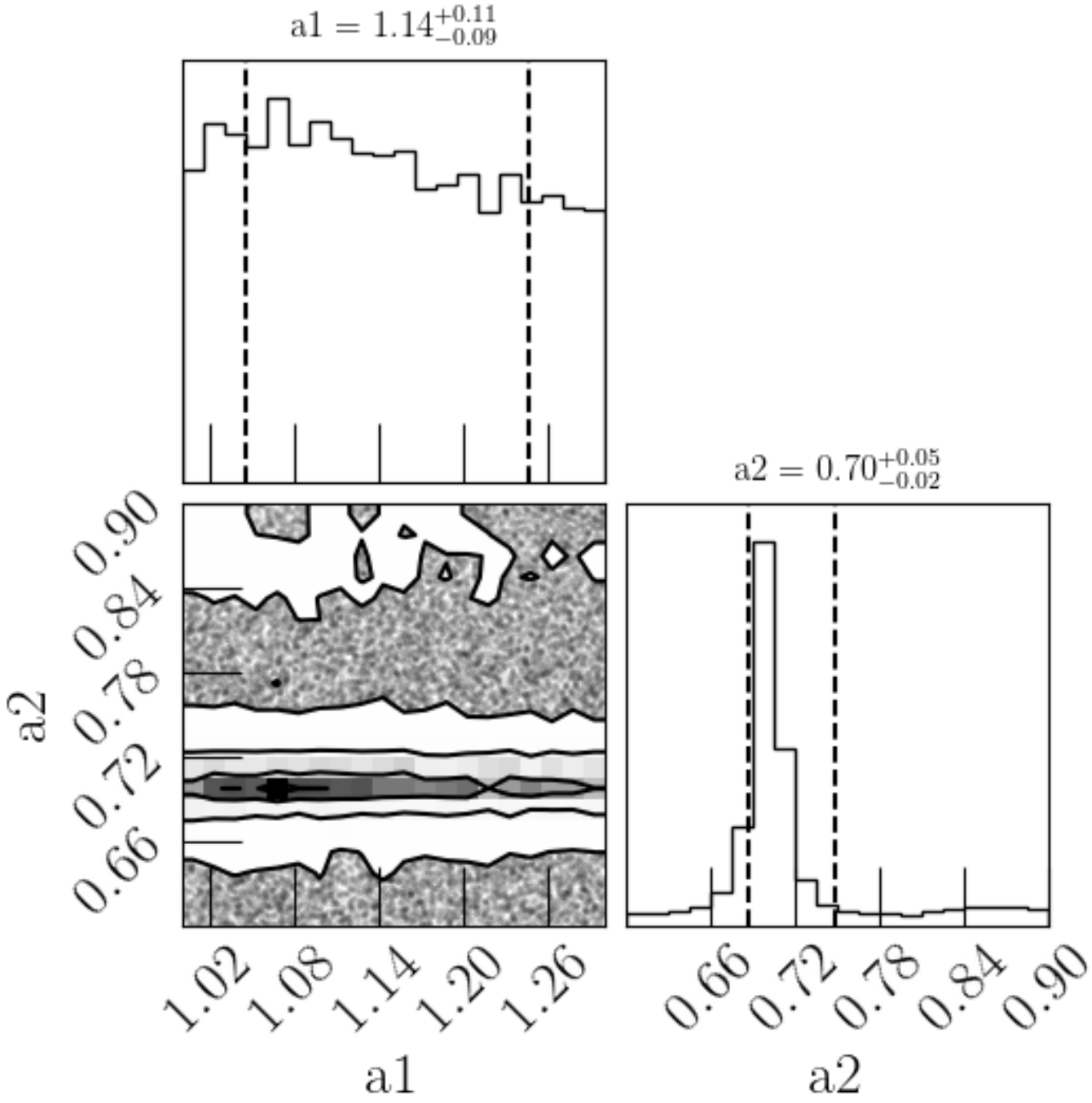

Figure 13. Corner plot of MCMC samples and 14%, 86% quantiles for linear fit.

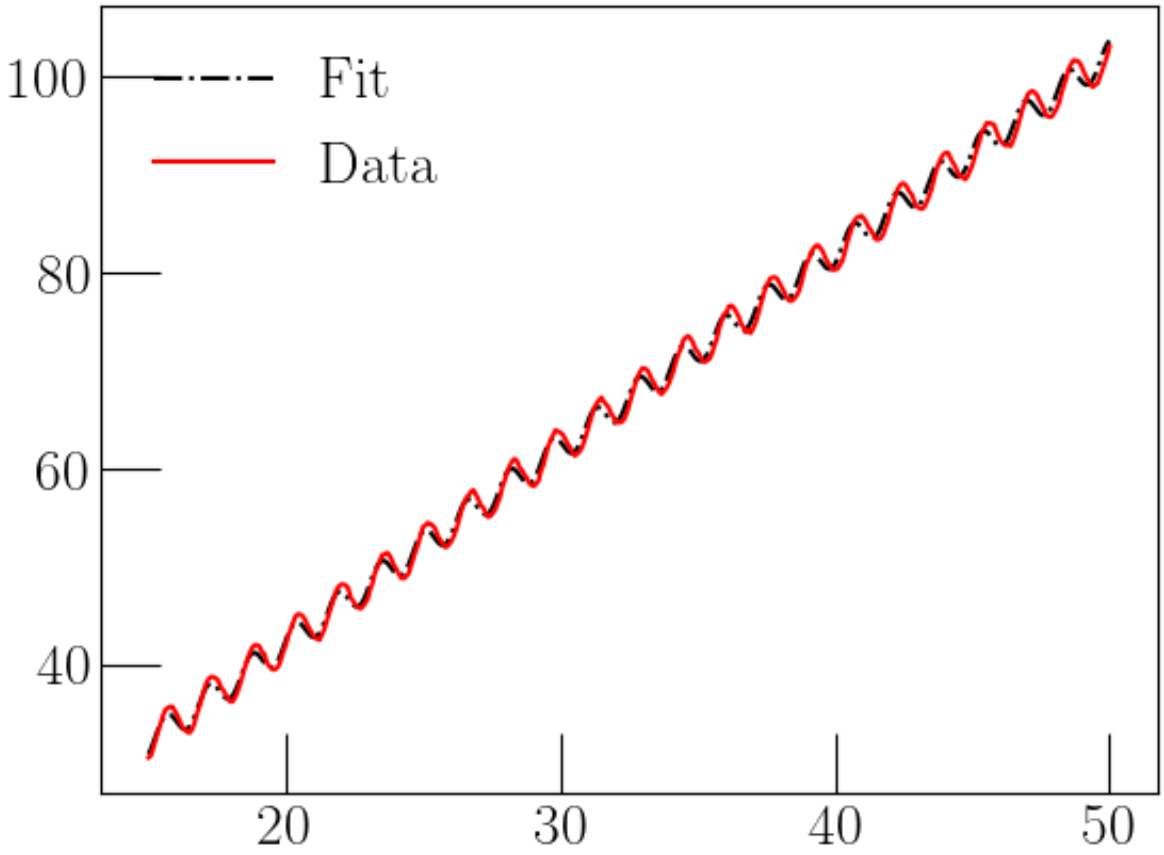

Figure 14. MCMC fit combining sinusoid and first-order polynomial of data with Gaussian Noise.

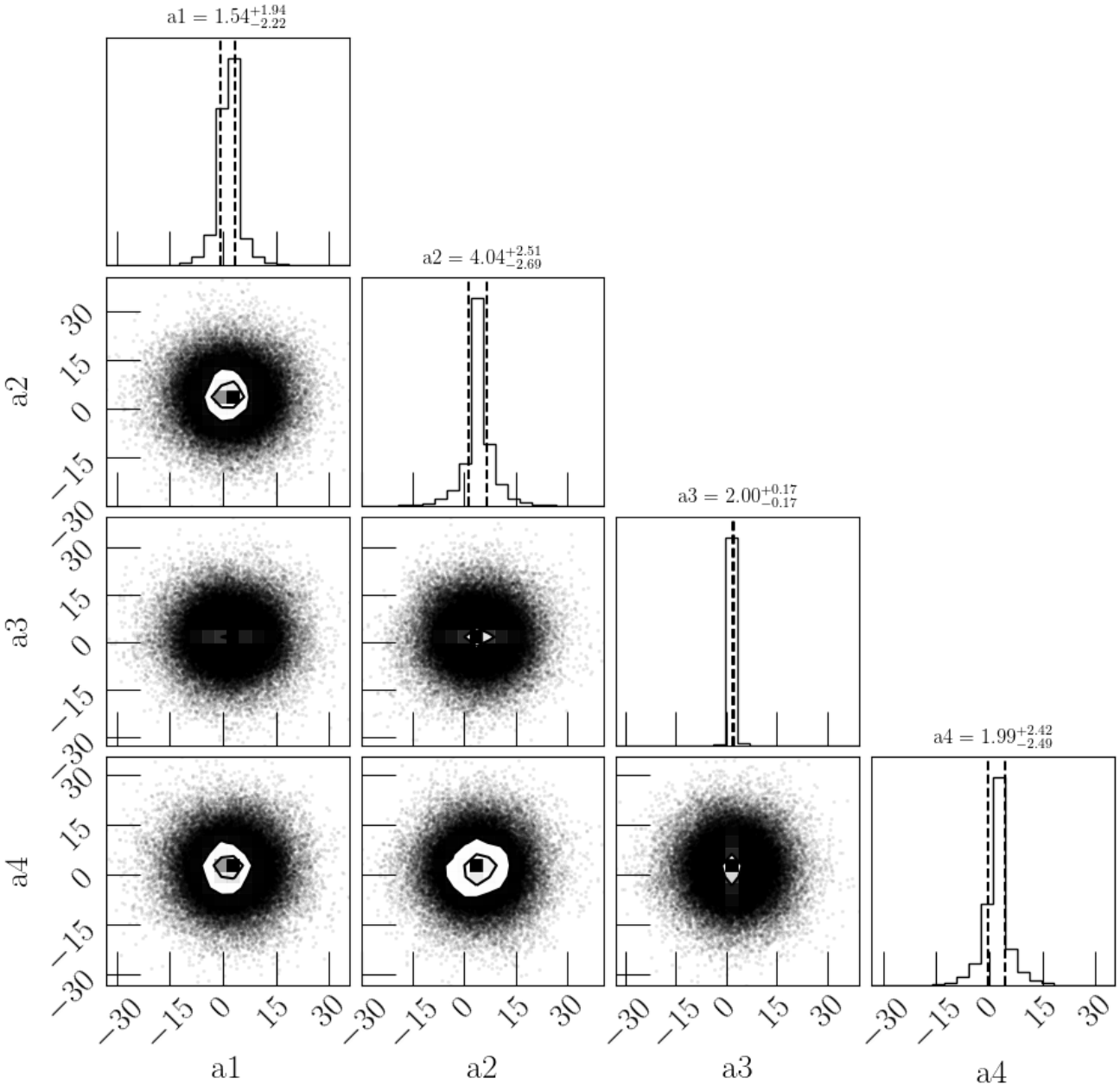

Figure 15. Corner plot of MCMC samples and 14%, 86% quantiles for linear fit.

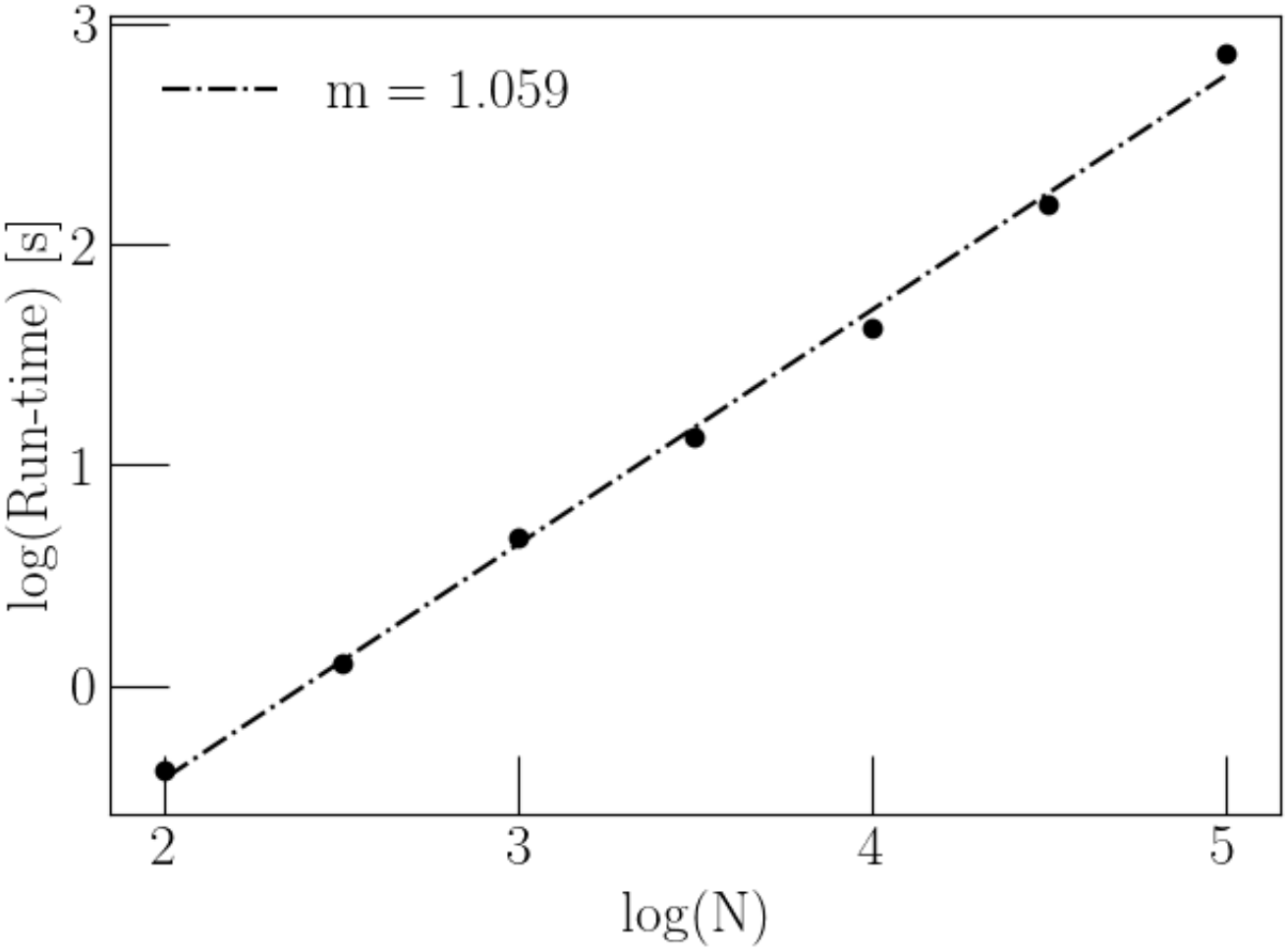

Figure 16. Log of runtime versus number of chains.

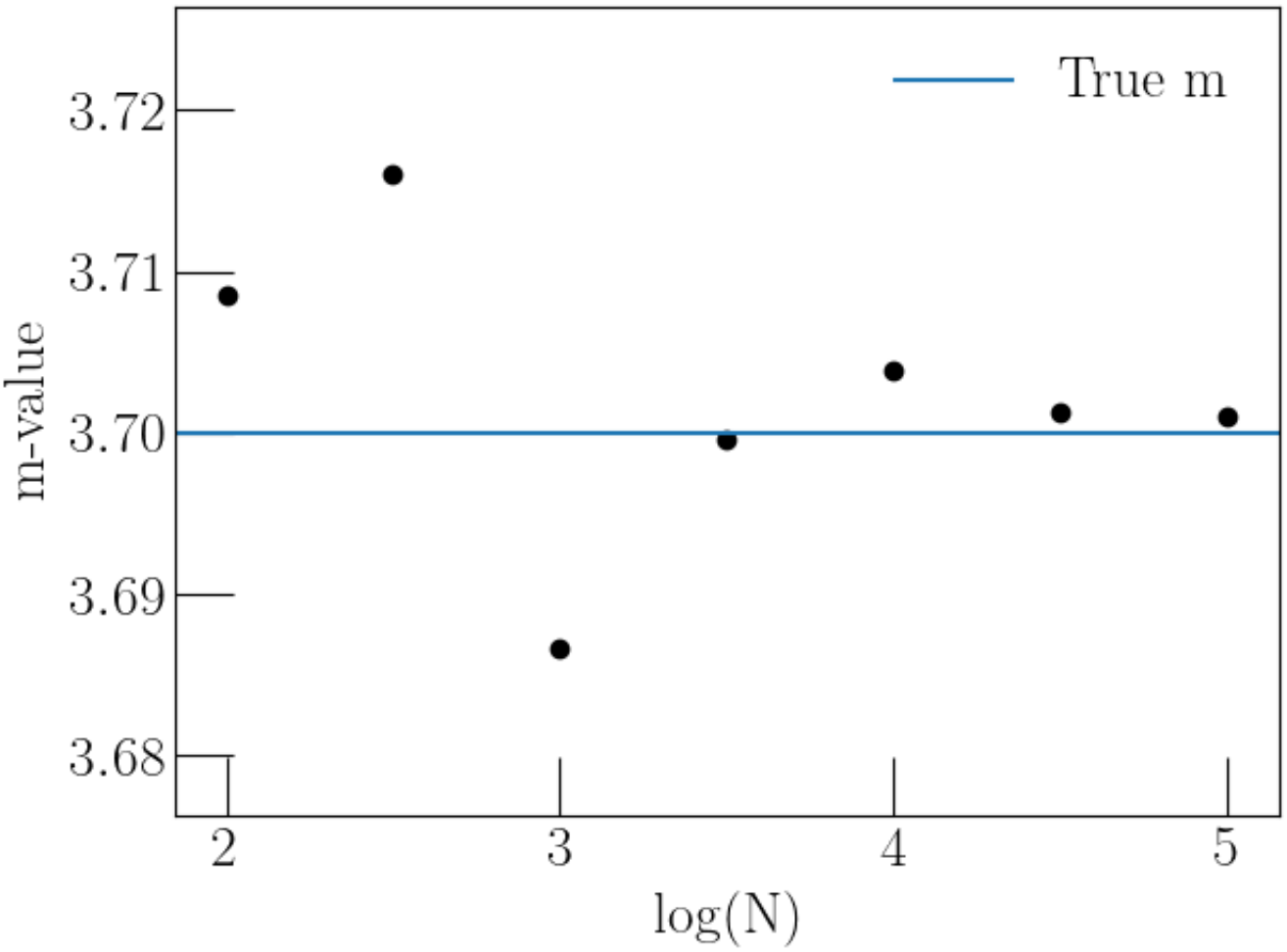

Figure 17. Fit slope-value versus number of chains.

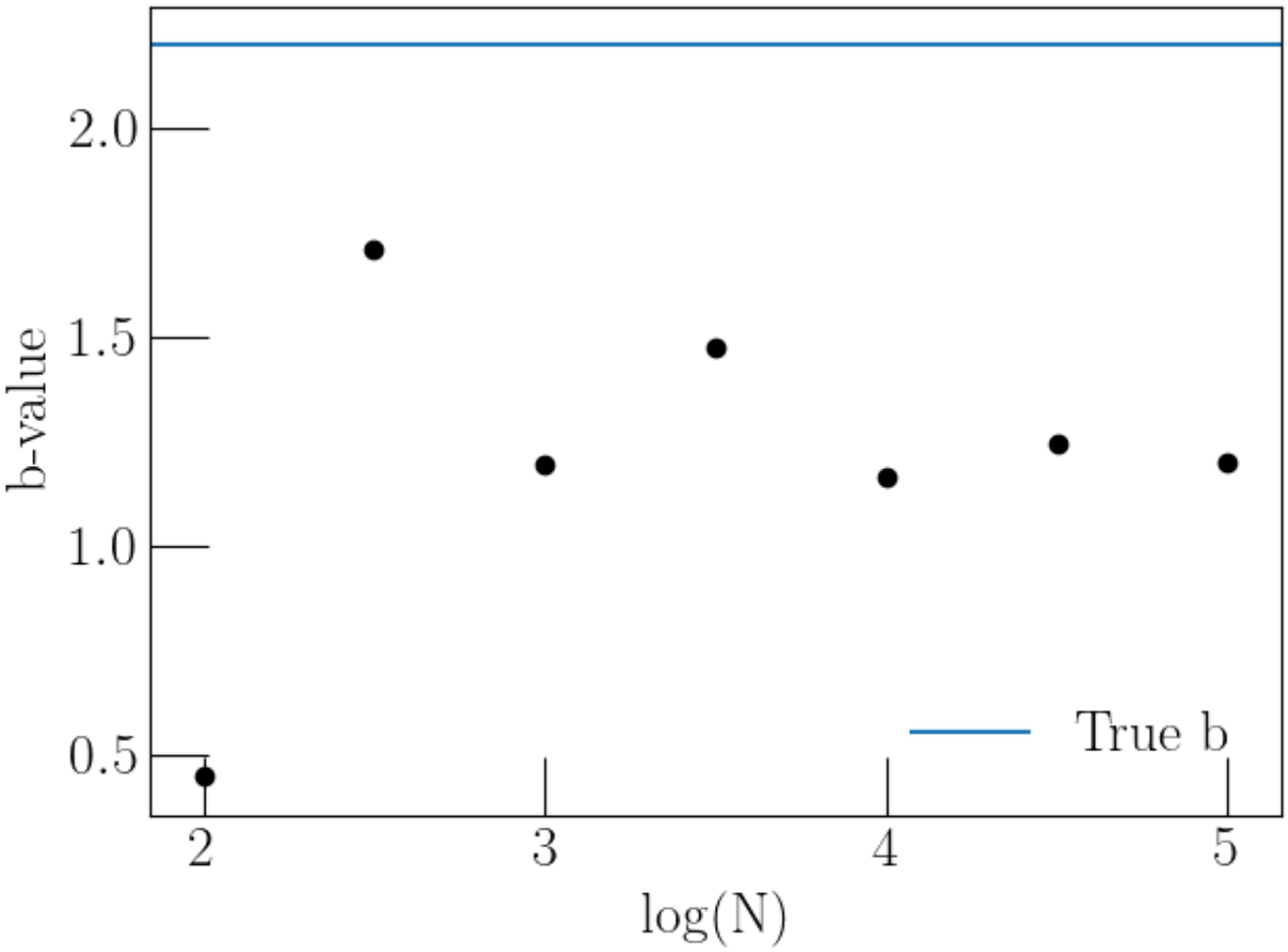

Figure 18. Fit y-intercept versus number of chains.

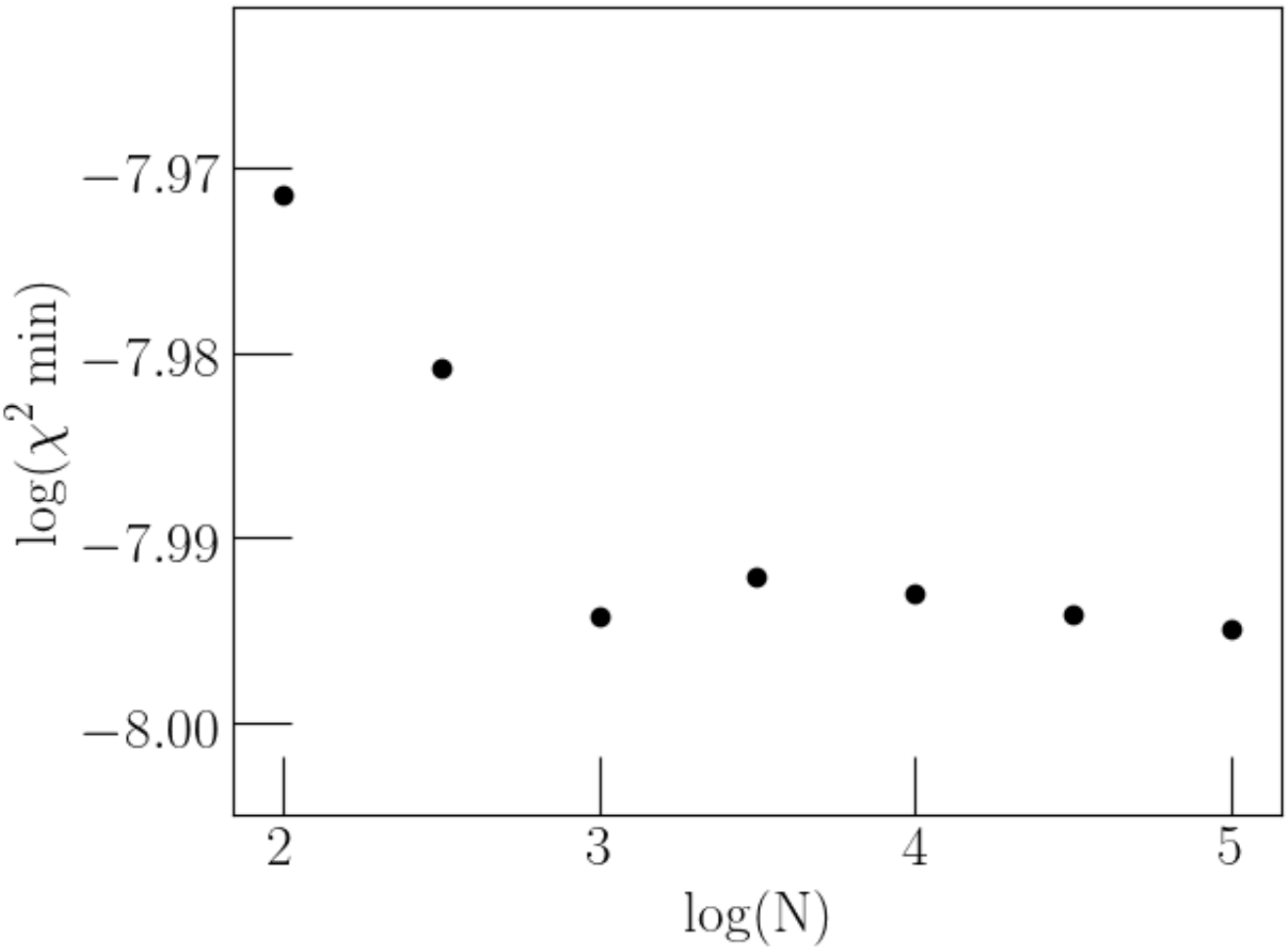

Figure 19. Minimum χ^2 value for distribution versus number of chains.

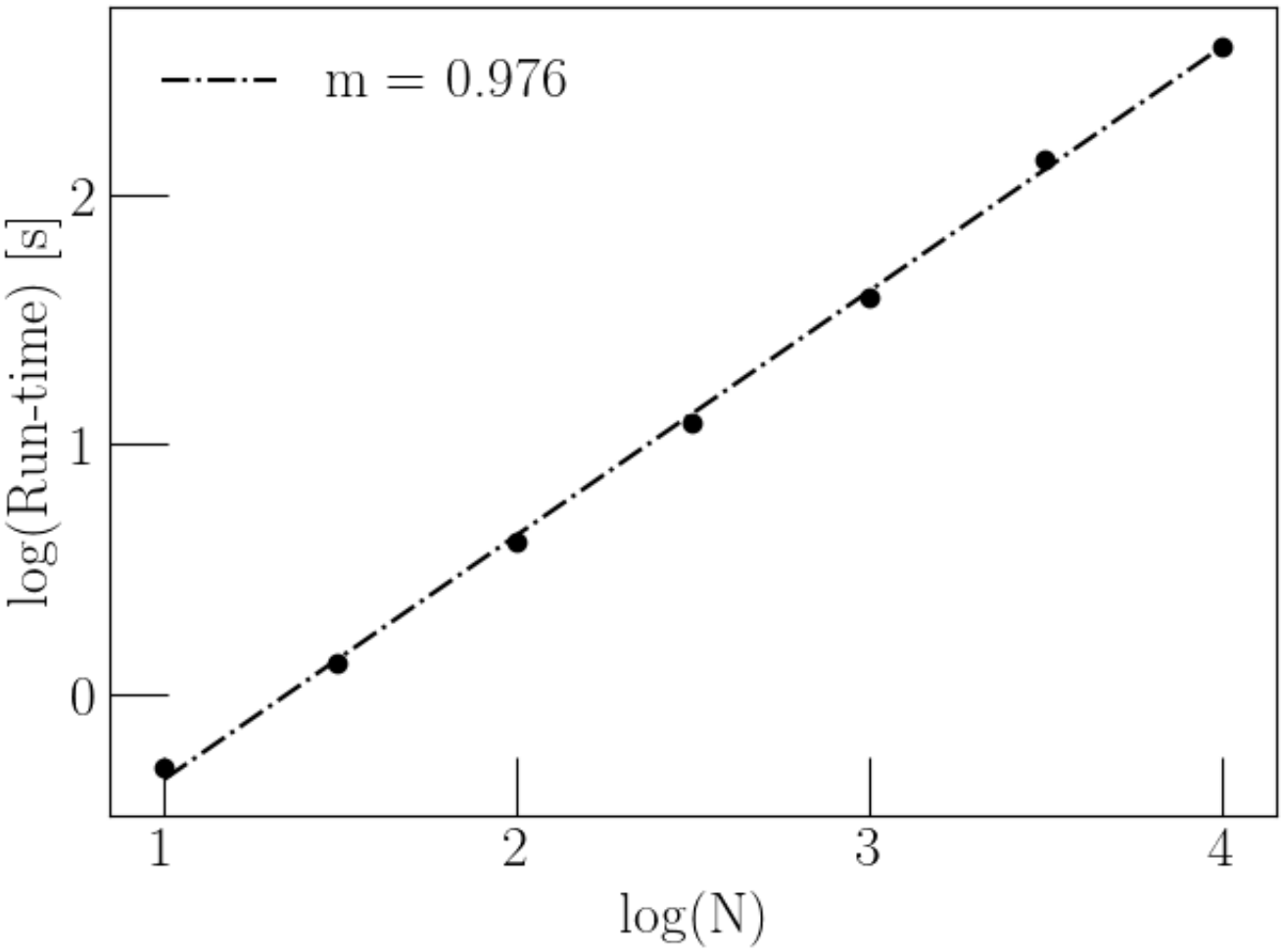

Figure 20. runtime versus number of iterations in each chain.

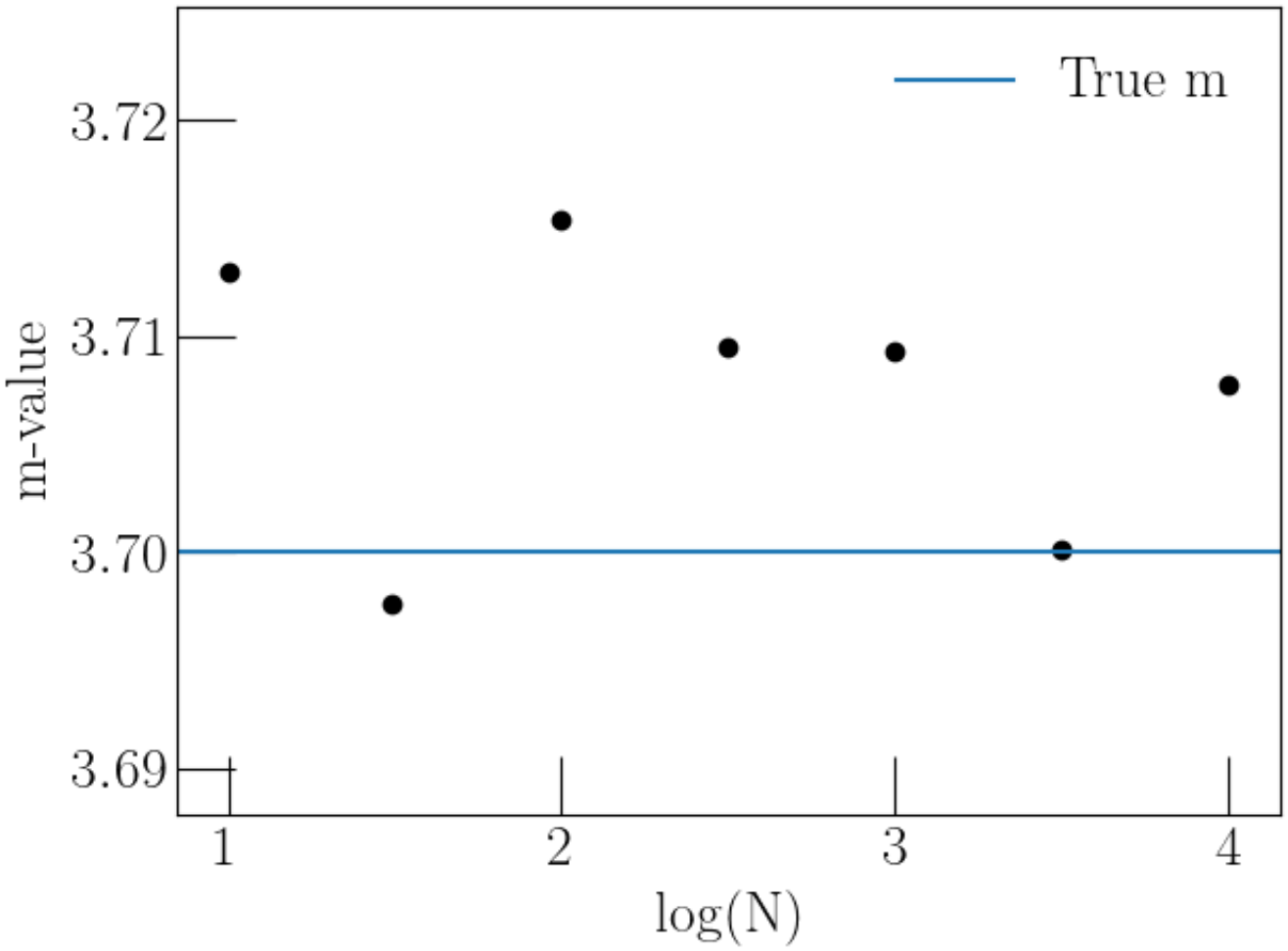

Figure 21. Fit slope-value versus number of iterations in each chain.

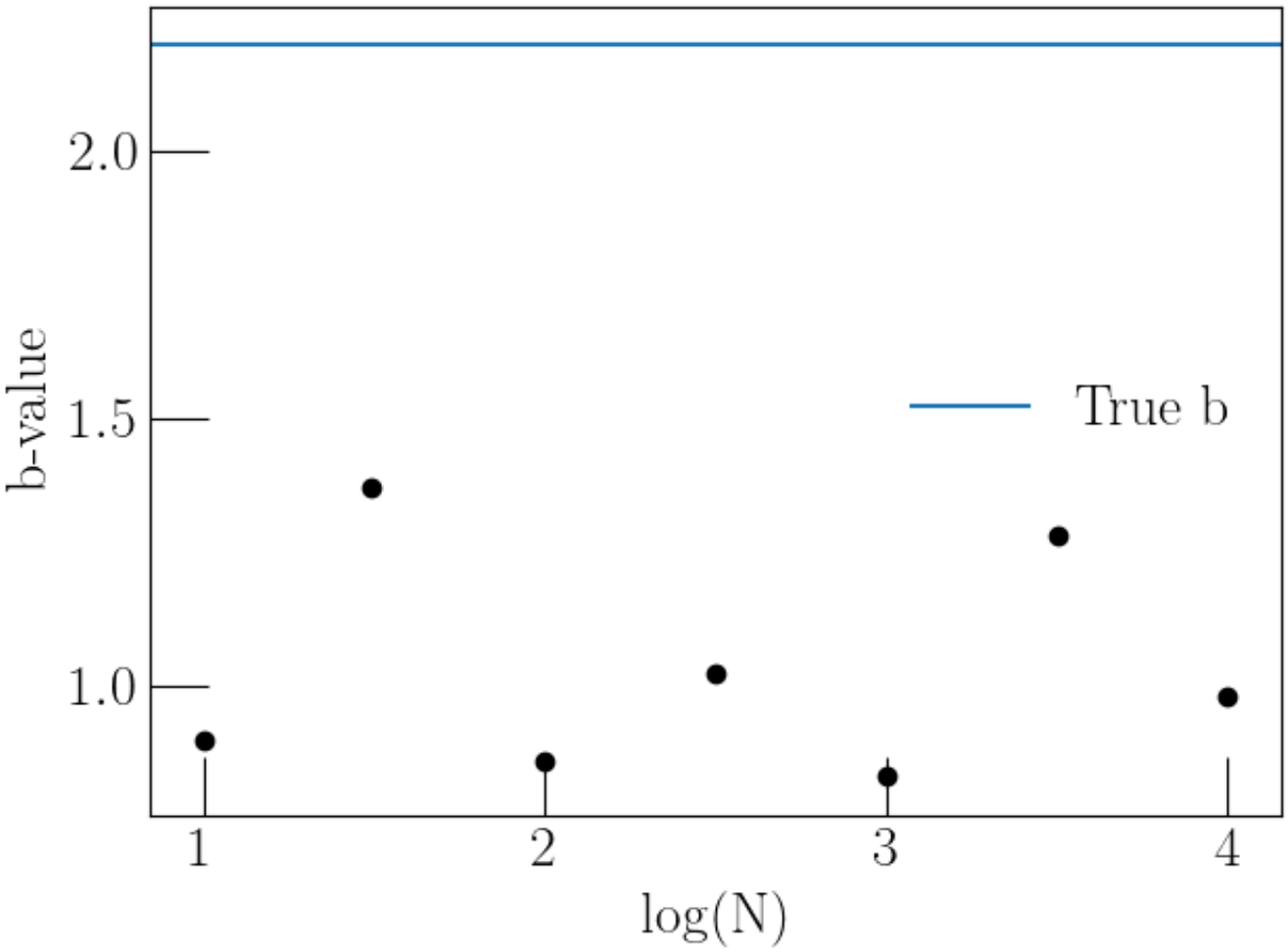

Figure 22. Fit y-intercept versus number of iterations in each chain.

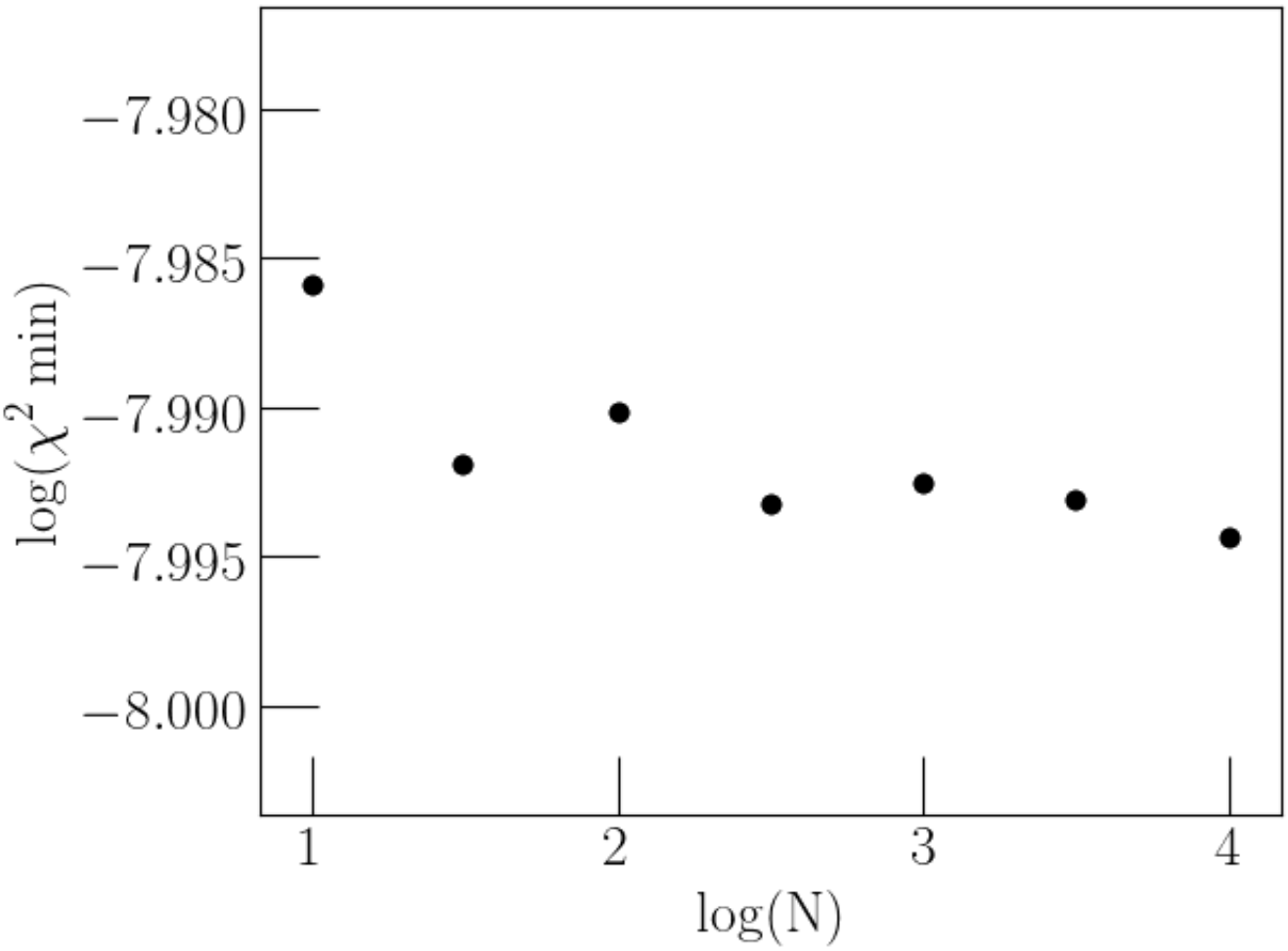

Figure 23. Minimum χ^2 value for distribution versus number of iterations in each chain.

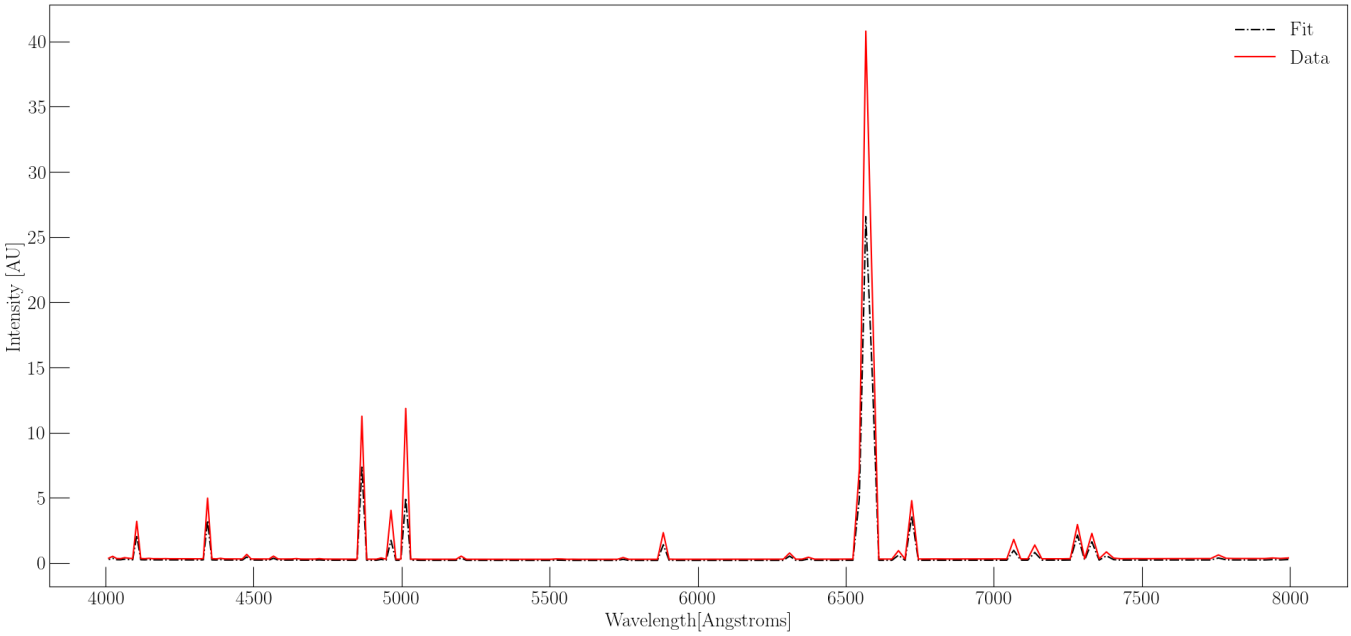

Figure 24. Cloudy Model and MCMC fit with only 10 chains and 10 iterations per chain.

```
import numpy as np
import matplotlib.pyplot as plt
import corner
import time
class MCMC:
    def __init__(self, model, xreal, yreal, initpars,
    parlims, n_iter=100, n_chains=1000, sample_rate=10):
        '''
        model: function to compute data based on parameters
        xreal: real x-data, used for computing new model data
        yreal: real y-data, used for comparing against model data
        initpars: input parameters where the mcmc chains begin
        parlims: limits on parameter space, often can be used to improve quality of result if
            known limitations can be applied
        n_iter: autoset to 100
        n_chains: autoset to 1000
        sample_rate: autoset to happen every 10 iterations in each chain.
        '''
        self.xreal = xreal
        self.model = model
        self.n_iter = n_iter
        self.initpars = initpars
        self.yreal = yreal
        self.dofs = len(self.yreal )-len(self.initpars)
        self.parlims = parlims
        self.n_pars = np.shape(parlims[0])
        self.n_chains = n_chains
        self.sample_rate = sample_rate
    def run_mcmc(self):
        '''
        This function creates chains that carry out the Metropolis-Hastings Algorithm
        to build up a likelihood distribution.
        '''
        self.start_time = time.time()
        all_samples = np.array([])
        all_probs = np.array([])
        all_finpars = np.array([])
        chains = []
        chainprobs = []
        chainpars = []
        self.initchainpar = self.initpars
        for i in range(self.n_chains):
            if i%100 ==0:
                print('n_chain: ', i)
            probs = []
            par_fin, samples, probs = self.run_chain()
            chains.append(samples)
            chainprobs.append(probs)
            all_samples = np.append(all_samples, samples)
            all_probs = np.append(all_probs, probs)
            all_finpars = np.append(all_finpars,par_fin)
```

```
            self.initchainpar = [np.random.uniform(self.parlims[i][0], self.parlims[i][1])
                    for i in range(len(self.parlims))]
            chainpars.append(par_fin)
        self.end_time = time.time()
        self.time_total = self.end_time-self.start_time
        self.all_samples = all_samples.reshape((int(np.size(all_samples)/len(self.initpars)),
                len(self.initpars))).transpose()
        self.all_probs = all_probs
        self.all_finpars = all_finpars
        self.chains = chains
        self.chainprobs = chainprobs
        self.chainpars = chainpars
        self.finalpars = np.array([np.average(self.all_samples[i], weights = -1/self.all_probs)
                for i in range(len(self.all_samples))])
    def run_chain(self):
        '''
        This function carries out the Metropolis-Hastings Algorithm for a chain.
        '''
        probs = []
        samples = []
        par = self.initchainpar
        self.ymod = self.model(self.xreal,par)
        p = self.chisq(self.ymod,self.yreal)
        for i in range(self.n_iter):
            parn = par + np.random.normal(size=len(par))
            parlimbools = [parn[i]>self.parlims[i][0] and parn[i]<self.parlims[i][1]
                    for i in range(len(self.parlims))]
            if not all(parlimbools):
                continue
            self.ymod = self.model(self.xreal, parn)
            pn = self.chisq(self.ymod, self.yreal)
            if pn >= p:
                p = pn
                par = parn
            else:
                u = np.random.rand()
                if u < pn/p:
                    p = pn
                    par = parn
            if i % self.sample_rate == 0:
                samples.append(par)
                probs.append(p)
        return par,samples, probs
    def chisq(self, ymod, yreal):
        '''
        Compares the model distribution to the real distribution
        to compute sum of square of residuals.
        '''
        return -np.sum((ymod-yreal)**2)

    def plotfit(self):
        plt.plot(self.xreal,self.model(self.xreal, self.finalpars),'k-.',label='Fit')
```

```
        plt.plot(self.xreal,self.yreal,'r-', label='Data')
        plt.legend(frameon=False, fontsize=20)
    def plotmcmc_corner(self):
        npars = len(self.parlims)
        parnames = ['a'+str(i+1) for i in range(npars)]

        figure = corner.corner(self.all_samples.transpose(), labels=parnames,
                               weights=-1/(self.all_probs),
                               quantiles = [0.16, 0.84],
                               show_titles=True,
                               title_kwargs={"fontsize": 12})
```

7. CLOUDY CLASS

```
import numpy as np
import pyCloudy as pc
class PCloudy:

    def __init__(self, pars ):
        '''
        Program for setting up parameters for pyCloudy to send to CLOUDY to compute
        properties of nebulae.
        '''
        self.numdensity = pars[0]
        self.Teff = pars[1]
        self.qH = pars[2]
        self.r_min = pars[3]
        self.dist = pars[4]
        self.dir_ = './models/'
        self.model_name = 'model_test'
        self.full_model_name = '{0}{1}'.format(self.dir_, self.model_name)
        self.options = ('no molecules',
                        'no level2 lines',
                        'no fine opacities',
                        'atom h-like levels small',
                        'element limit off -8')
        self.emis_tab = ['H  1  4861.33A', #hbeta
                         'H  1  6562.81A', #halpha
                         'N  2  6583.45A', #nii
                         'O  3  5006.84A' ] #oiii
        self.abund = {'He' : -0.92, 'C' : 6.85 - 12, 'N' : -4.0, 'O' : -3.40, 'Ne' : -4.00,
         'S' : -5.35, 'Ar' : -5.80, 'Fe' : -7.4, 'Cl' : -7.00}
        c_input = pc.CloudyInput(self.full_model_name)
        c_input.set_BB(Teff = self.Teff, lumi_unit = 'q(H)', lumi_value = self.qH)
        c_input.set_cste_density(self.numdensity)
        c_input.set_radius(r_in=np.log10(self.r_min))
        c_input.set_abund(ab_dict = self.abund, nograins = True)
        c_input.set_other(self.options)
        c_input.set_iterate()
        c_input.set_sphere() #
        c_input.set_emis_tab(self.emis_tab ) #
        c_input.set_distance(dist=self.dist, unit='kpc', linear=True)
        c_input.print_input(to_file = True, verbose = False)
```

```
        self.c_input = c_input

    def cloudy(self):
        '''
        Computes a Cloudy Model for the parameters specified in the initialization
        '''
        pc.log_.message('Running {0}'.format(self.model_name ), calling = 'test1')
        pc.config.cloudy_exe = '~/cloudy/source/sys_gcc/cloudy.exe'
        pc.log_.timer('Starting Cloudy', quiet = True, calling = 'test1')
        self.c_input.run_cloudy()
        self.Mod = pc.CloudyModel(self.full_model_name)
        pc.log_.timer('Cloudy ended after seconds:', calling = 'test1')
```

8. MCMC TESTS

```
import numpy as np
import matplotlib.pyplot as plt
from MCMC import MCMC
from scipy.stats import sem
import pyCloudy as pc
from PCloudy import PCloudy
plt.rc('font',family='serif')
plt.rc('text',usetex=True)
def poly(x, pars):
    return np.sum(np.array([ pars[i]*x**(len(pars)-1- i) for i in range(len(pars))]), axis=0)

def polyperiodic(x, pars):
    return pars[0]*np.cos(pars[1]*x)+np.sum(np.array([ pars[i]*x**(len(pars)-1- i)
        for i in range(2,len(pars))]), axis=0)
def per(x, pars):
    return pars[0]*np.sin(x*pars[1])

x_vals = np.linspace(-15,15, 400)

period = [1.1,0.7 ]

linperiod = [2, 4, 2, 2.2]
linpars = [3.7, 2.2]
quadpars = [-1.1,2.3,-3.5]
cubicpars = [-2.1, 2.3, 4.5, -10]
quarticpars = [4.2,1.1, -4.3, -2.1, 14]
quinticpars = [-0.8, -6.2, 1.4, 3.2, 2.1, 4.8]

linper_true = polyperiodic(x_vals, linperiod)
per_true = per(x_vals, period)
lin_true = poly(x_vals, linpars)
quad_true = poly(x_vals,quadpars)
cubic_true = poly(x_vals,cubicpars)
quartic_true = poly(x_vals,quarticpars)
quintic_true = poly(x_vals,quinticpars)

linper_comp = linper_true+np.random.normal(scale=0.1,size=len(linper_true))
per_comp = per_true+np.random.normal(scale=0.2,size=len(per_true))
```

```
lin_comp = lin_true + np.random.normal(size=len(lin_true))
quad_comp = quad_true+np.random.normal(size=len(x_vals))
cubic_comp = cubic_true+np.random.normal(size=len(x_vals))
quartic_comp = quartic_true+np.random.normal(size=len(x_vals))
quintic_comp = quintic_true+np.random.normal(size=len(x_vals))


def test_n_chains(plot_n=False, plot_chi=False):
    '''
    This function contains tests of the MCMC on the number of
    chains used overall.
    '''
    n_chain_arr= np.int64(10**(np.arange(2,5.5,0.5)))
    runtimes = []
    runs = []
    for n_chain in n_chain_arr:
        lm = MCMC(poly, x_vals, lin_comp, [2,2], [[-10,10], [-30,30]],
            n_iter=100, n_chains = n_chain )
        lm.run_mcmc()
        runs.append(lm)
        runtimes.append(lm.time_total)
    runtimes = np.array(runtimes)

    bs = []
    ms = []
    chi_min = []
    for mc in runs:
        finpars = mc.finalpars
        bs.append([finpars[1]])
        ms.append([finpars[0]])
        chi_min.append(np.min(mc.all_probs))

    ms = np.array(ms)
    mdiffs = np.abs(ms - linpars[0])
    bs = np.array(bs)
    bdiffs = np.abs(bs - linpars[1])
    chi_min =np.array(chi_min)
    if plot_chi:

        plt.scatter(np.log10(n_chain_arr), ms, color='k' )
        plt.axhline(y=linpars[0],label='True m')
        plt.xlabel('log(N)')
        plt.ylabel('m-value')
        plt.legend(frameon=False, fontsize=20)
        plt.tight_layout()


        plt.scatter(np.log10(n_chain_arr), bs, color='k' )
        plt.axhline(y=linpars[1],label='True b')
        plt.xlabel('log(N)')
        plt.ylabel('b-value')
        plt.legend(frameon=False, fontsize=20)
```

```
        plt.tight_layout()

        plt.scatter(np.log10(n_chain_arr), -np.log10(-chi_min), color='k' )
        plt.xlabel('log(N)')
        plt.ylabel(r'log($\chi^2$ min)')
        plt.legend(frameon=False, fontsize=20)
        plt.tight_layout()


    if plot_n:
        plt.scatter(np.log10(n_chain_arr), np.log10(runtimes), color='k' )
        m, b = np.polyfit(np.log10(n_chain_arr), np.log10(runtimes), 1)
        plt.plot(np.log10(n_chain_arr),m*np.log10(n_chain_arr)+b,'k-.', label='m = '+str(m)[0:5])
        plt.xlabel('log(N)')
        plt.ylabel('log(runtime) [s]')
        plt.legend(frameon=False, fontsize=20)
        plt.tight_layout()

def test_n_iter(plot_n=False, plot_chi=False):
    '''
    This function contains tests of the MCMC on the number of
    iterations used in each chain.
    '''
    n_iter_arr= np.int64(10**(np.arange(1,4.5,0.5)))
    runtimes = []
    runs = []
    for n_iter in n_iter_arr:
        lm = MCMC(poly, x_vals, lin_comp, [2,2], [[-10,10],[-30,30]],
            n_iter=n_iter, n_chains = 1000 )
        lm.run_mcmc()
        runs.append(lm)
        runtimes.append(lm.time_total)
    runtimes = np.array(runtimes)

    bs = []
    ms = []
    chi_min = []
    for mc in runs:
        finpars = mc.finalpars
        bs.append([finpars[1]])
        ms.append([finpars[0]])
        chi_min.append(np.min(mc.all_probs))

    ms = np.array(ms)
    mdiffs = np.abs(ms - linpars[0])
    bs = np.array(bs)
    bdiffs = np.abs(bs - linpars[1])
    chi_min =np.array(chi_min)
    if plot_chi:

        plt.scatter(np.log10(n_iter_arr), ms, color='k' )
        plt.axhline(y=linpars[0],label='True m')
```

```
        plt.xlabel('log(N)')
        plt.ylabel('m-value')
        plt.legend(frameon=False, fontsize=20)
        plt.tight_layout()


        plt.scatter(np.log10(n_iter_arr), bs, color='k' )
        plt.axhline(y=linpars[1],label='True b')
        plt.xlabel('log(N)')
        plt.ylabel('b-value')
        plt.legend(frameon=False, fontsize=20)
        plt.tight_layout()

        plt.scatter(np.log10(n_iter_arr), -np.log10(-chi_min), color='k' )
        plt.xlabel('log(N)')
        plt.ylabel(r'log($\chi^2$ min)')
        plt.legend(frameon=False, fontsize=20)
        plt.tight_layout()

    if plot_n:
        plt.scatter(np.log10(n_iter_arr), np.log10(runtimes), color='k' )
        m, b = np.polyfit(np.log10(n_iter_arr), np.log10(runtimes), 1)
        plt.plot(np.log10(n_iter_arr),m*np.log10(n_iter_arr)+b,'k-.', label='m = '+str(m)[0:5])
        plt.xlabel('log(N)')
        plt.ylabel('log(runtime) [s]')
        plt.legend(frameon=False, fontsize=20)
        plt.tight_layout()

def test_mods():
    '''
    This function contains calls which are used in testing my
    MCMC method on polynomials.
    '''
    n_iter=100
    n_chains=10000

    linperlims = [[1.,3.],[3,5],[1.,3.],[1.,3.]]
    linpermcmc= MCMC(polyperiodic,x_vals, linper_comp, [2.5,3,2,2],
                linperlims, n_iter=n_iter, n_chains=n_chains)
    linpermcmc.run_mcmc()

    perlims = [[1.0,1.3],[0.6,0.9]]
    permcmc= MCMC(per, x_vals, per_comp, [1,1],
                perlims, n_iter=n_iter, n_chains=n_chains)
    permcmc.run_mcmc()

    linlims = [[-10,10],[-30,30]]
    linmcmc = MCMC(poly,x_vals, lin_comp, [2,2],
                linlims, n_iter=n_iter, n_chains=n_chains)
    linmcmc.run_mcmc()
```

```
    quadlims = [[-10,10],[-10,10],[-30,30]]
    quadmcmc = MCMC(poly, x_vals, quad_comp, [1,1,1],
                quadlims, n_iter=n_iter, n_chains=n_chains)
    quadmcmc.run_mcmc()

    cubiclims = [[-10,10],[-10,10],[-10,10],[-30,30]]
    cubicmcmc = MCMC(poly, x_vals, cubic_comp, [1,1,1,1],
                cubiclims, n_iter=n_iter, n_chains=n_chains)
    cubicmcmc.run_mcmc()

    quarticlims = [[-10,10],[-10,10],[-10,10],[-10,10],[-30,30]]
    quarticmcmc = MCMC(poly, x_vals, quartic_comp, [1,1,1,1,1],
                quarticlims, n_iter=n_iter, n_chains=n_chains)
    quarticmcmc.run_mcmc()

    quinticlims = [[-10,10],[-10,10],[-10,10],[-10,10],[-10,10],[-30,30]]
    quinticmcmc = MCMC(poly, x_vals, quintic_comp, [1,1,1,1,1,1],
                quinticlims, n_iter=n_iter, n_chains=n_chains)
    quinticmcmc.run_mcmc()


def test_cloudy():
    '''
    This function contains function calls to run the MCMC on Cloudy models.
    '''
    dens = 2 #log cm^-3
    Teff = 45000. #K
    qH = 47. #s-1, ionizing phot /second
    r_min = 5e17 #cm,
    dist = 1.26 #kpc, how far is the region from us
    cloudypars = np.array([dens, Teff, qH, r_min, dist])
    pclo = PCloudy(cloudypars)
    pclo.cloudy()
    Mod = pclo.Mod
    wl = Mod.get_cont_x(unit='Ang')

    intens = Mod.get_cont_y(cont='ntrans',unit='Jy')
    opt=np.where((wl>4000) & (wl<8000))[0]
    wl_opt = wl[opt]
    intens_opt = intens[opt]
    initpars = cloudypars + np.random.normal(size=5)
    cloudylims = [[1,3],[40000,50000],[46,49],[1e17,1e18],[1,2]]
    cloudyMCMC = MCMC(cloudy_model, wl_opt, intens_opt, initpars,
                cloudylims, n_iter=10, n_chains=100, sample_rate=1)
    cloudyMCMC.run_mcmc()
def cloudy_model(wl_ran, pars):
    '''
    This cloudy_model function is used for computing cloudy models and
    returns spectrum intensity in the optical region.
    '''
    pclo = PCloudy(pars)
    pclo.cloudy()
```

```
    Mod = pclo.Mod
    wl = Mod.get_cont_x(unit='Ang')
    intens = Mod.get_cont_y(cont='ntrans',unit='Jy')
    opt=np.where((wl>4000) & (wl<8000))[0]
    intens_opt = intens[opt]
    return intens_opt
```