# Numerical Laboratory
# An Example of Monte Carlo Markov Chain Methods: Estimating the Mass of a Galaxy Cluster with Weak Lensing

**Abstract.** Weak Lensing is one of the main cosmological probes. From observed data of cosmic structures (such as their shear and convergence profiles), we perform statistical analyses to obtain information about them, since their growth is not explicitly defined by the cosmological model. Particularly, in this project, we are going to use Monte Carlo Markov Chains (MCMC) in order to estimate the virial mass and concentration of a galaxy cluster. By fitting the observed cosmic shear profile to a theoretical model we compute the $\chi^2$ statistic as a goodness of fit of this model. From this function we obtain a point estimate of the values of the mass and concentration. Then, we study the parameter space and its posterior distribution using the MCMC, which provides us a deeper idea of the relationship between the parameters.

Jennifer Fabà-Moreno
Pablo Vega del Castillo
Group 18

Prof. Alex Cridland

May 25, 2025

## 1 Theoretical framework

One of the main probes of Einstein's General Relativity is that light is bent in a gravitational field. When the distortion is so noticeable that the image of a distant source is perceived as an arc around the lens (called Einstein rings when they form the complete circumference) or as multiple images around it, we are working in the Strong lensing regime. If, on the other hand, we detect little distortions by the tidal gravitational field of the intervening matter inhomogeneities on the images of the source, we are working with Weak lensing.

Weak lensing is sensitive to both the geometry of the Universe and also to the growth of structure and its evolution in redshift [1]. Since both concepts are related to the evolution of dark energy and its equation-of-state parameter $\omega$, this sensitivity carries over to the cosmic shear signal. The shear and convergence profiles describe the distortions we observe on the galaxy's shapes (we use the sources as test particles to measure the effect of the lens in light propagation). As we do not usually know the intrinsic properties of the lenses, we work with the average of them, so we make use of statistical tools for their analyses [2]. With them, we have a direct way of probing the statistical properties of the large-scale dark matter distribution in the Universe.

Thus, we are going to work with this shear profile to evaluate the properties of the lens. In our case, we want to get the virial mass and concentration of the lens, $M200$ and $c$, respectively, and we are going to use the data from the lens dark matter halo ([3], file `halo5.tab`) to do so. It includes the measures needed for computing the theoretical model of the shear profile. On the other hand, it also includes the observed values of the shear, so we can make use of the chi-squared statistic as a goodness of fit:

$$\chi^2(\theta) = \sum_{i=1}^{N} \frac{(\gamma_{model}(r_{dat,i}, \theta) - \gamma_{dat,i})^2}{\sigma^2_{\gamma,i}} \tag{1.1}$$

to test the validity of the theoretical model for $\gamma$ ([3], file `Instructions`, page 4). By normalizing this factor by the number of degrees of freedom, we are able to compare the accuracy of the model when exposing it to sets of different size. In this case, we work with the reduced $\chi^2$.

Our primary goal is to estimate the posterior function for the parameters $\theta = (M200, c)$. However, as stated in [3], the computational work relies on the computation of the likelihood, so we will work with a Monte Carlo approximation of it, allowing us to work numerically. Monte Carlo Markov chains together with the Metropolis Hastings Algorithm provide us with the stationary distribution of the likelihood function, from where we can obtain the most probable (MLE) mass and concentration of our lens.

Markov chains are set of random variables $\{X_n\}$ in which $\mathbb{P}(X_{n+1} = i_{n+1}|X_0 = i_0, ..., X_{n-1} = i_{n-1}, X_n = i_n) = \mathbb{P}(X_{n+1} = i_{n+1}|X_n = i_n)$, hence Markov chains are said to not have memory. Thus, when computing the step $n$ of the chain, we do not need the $n-1$ values but only the previous one. Regarding the Metropolis Hastings algorithm, it let us build the Markov chain by comparing at each step the ratio of likelihoods of 2 given points with a random uniform variable $U \sim U(0, 1)$. In particular, we start with the initial guess for mass and concentration $X = X_0$ and compare it with a multivariate Gaussian random variable $Y$ (which we have chosen following the suggestions of [3]) with mean $X$ and covariance $C$. We update the value of $X$ if the probability of the parameters being $Y$ is greater than $U$[1].

The efficiency of this algorithm depends on the scaling of the proposal density. If its variance is too small, the Markov chain will converge slowly since all its increments will be small. Conversely, if the variance is too large, the Metropolis algorithm will reject too high a proportion of its

[1]That is why $U$ is chosen to be a uniform in (0,1), since we want it to represent a probability.

proposed moves. Thus, we are going to follow the criterion presented in [4] when choosing $C$ which is based in the value of the acceptance rate $\alpha$:

$$\alpha(X,Y) = \min\left(1, \frac{\pi(Y)}{\pi(X)}\right) \tag{1.2}$$

where $\pi$ is the stationary distribution. It states that $\alpha \sim 0.234$ for it to be optimal. When working with covariance matrices that are proportional to the diagonal: $C = \sigma^2$diag(1,...,1), we can tune the value of $\alpha$ by changing $\sigma^2$. With other types of covariance matrices, finding the optimal value of $\alpha$ would be more challenging.

Finally, we are going to generate various chains with different initial parameters to widely analyse all the possible values of the parameters. To check the convergence of this set of chains, we consider the indicator $R$, defined in the manual [3]:

$$\sqrt{R} = \sqrt{\frac{v\hat{a}r(\psi)}{W}} \tag{1.3}$$

where $W$ is the average of the variances of all the chains and $v\hat{a}r(\psi)$ is an over-estimate of the variance of in the target distribution. On the other hand, $W$ represents an underestimate of the target variance, because the finite chains have not had time to explore all the parameter space. In the limit of infinite chains (that is, the stationary distribution) we want the ratio to tend to $1$, so as to the variance of the chain to represent the target variance. Thus, this will be our convergence criterion.

## 2 Tasks

### 2.1 Exercise 1

**Have a look at the IPython notebook. Implement the $\chi^2$ calculation outlined above and complete the `Cluster` and `Grid_Search` classes. You can find more information and most of the functions you need in the notebook.**
**Test your new function for speed and aim for execution times of $\sim 0.1$ s. If you try several methods of calculating $\chi^2$ and find performance differences, document them brie y. Can you think of reasons for the difference in execution times?**
**Keep in mind that Python is not a compiled language. Once you have a fast routine, implement a loop over $M$ and $c$ to find an estimate for the minimal $\chi^2$ and make a 2d-plot of the likelihood (Hint: The mass of the cluster is above $10^{14} M_\odot$ and below $10^{16} M_\odot$, the concentration is above 2 and below 10).**

In this first part of the project we are going to work with the Python class `Cluster`, which we have to complete. In this section we are going to write only the code we have implemented but you can find all the instances of the class in [3], file `code_MCMC.ipynb`

```
def get_model_values(self, M200, c200):

    x = self.r/self.scale_radius_nfw(M200,c200)
    model = np.zeros(len(x))
    #if x<1:
    model[np.where(x<1)] = (self.scale_radius_nfw(M200,c200) * self.delta_c(c200)
   * self.rho_crit_zlens) / self.Sigma_critical(self.zlens,self.zsource) * self.
   g_less(x[np.where(x<1)])
    #if x == 1:
    model[np.where(x==1)] = (self.scale_radius_nfw(M200,c200) * self.delta_c(c200)
     * self.rho_crit_zlens) / self.Sigma_critical(self.zlens,self.zsource) *
   (10./3. - 4. * np.log(2.))
    #else:
    model[np.where(x>1)] = (self.scale_radius_nfw(M200,c200) * self.delta_c(c200)
   * self.rho_crit_zlens) / self.Sigma_critical(self.zlens,self.zsource) * self.
   g_larger(x[np.where(x>1)])

    return model

def chi_sq(self, y_model, y_data, y_err):
    chi = (y_model - y_data) / y_err

    return np.sum(chi**2)
```

Listing 1: Code incorporated into the existing Cluster class.

For the `get_model_values` function we have computed the value of the dimensionless radial distance $x = \frac{r}{r_s}$ so as to evaluate the model, that is, the gravitational shear profile. As stated in [3], we have to take into account where we are evaluating the shear profile, thus we define the value of the function depending on the value of $x$.
Once we have the model for the shear profile implemented, we can compute the $\chi^2$ value in order to compare the predictions of this theoretical model with the actual observed values.
We consider the prior range to be the one stated in the question so when we set the parameters limits we use these values. On the other hand, one of the instances of the `Cluster` class chooses uniformly one random number from this range which we assign to the variables $M_{200}, c_{200}$. We use these values to define our model and to obtain the value of the reduced $\chi^2$. In our case, as we have implemented a function that computes the value of the $\chi^2$ 1.1, there is only left to normalize it. As we are trying to fit 2 parameters, the normalization factor would be $n-2$ where $n$ is the number of data points.

```
cluster = Cluster(filename_small)
cluster.set_parameter_limits()
M200, c200 = cluster.get_random_parameter()
cluster.log_prob(M200, c200)
--> outout
np.float64(15.967531956387283)
```

Listing 2: Test of Cluster class.

We have checked that everything was working in our class, by creating a small file of the original one and computing the MLE of the $\chi^2$ of the subset, so we can continue with further experiments with the hole data set.

Now, we have to optimize our function `get_model_values`, since we are going to call it multiple times during our experiments, when calling the `log_prob` function (Appendix A).

```
1  def get_model_values(self, M200, c200):
2
3      x = self.r/self.scale_radius_nfw(M200,c200)
4      model = np.ones(len(x)) * (self.scale_radius_nfw(M200,c200) * self.delta_c(
      c200) * self.rho_crit_zlens) / self.Sigma_critical(self.zlens,self.zsource)
5      #if x<1:
6      model[np.where(x<1)] *= self.g_less(x[np.where(x<1)])
7      #if x == 1:
8      model[np.where(x==1)] *= (10./3. - 4. * np.log(2.))
9      #else:
10     model[np.where(x>1)] *= self.g_larger(x[np.where(x>1)])
11
12     return model
```

Listing 3: Alternative for `get_model_values` function.

As suggested in the Jupyter notebook given, we do not make any loop when defining the above mentioned functions, in order to reduce the computational time in Python. In order to get the required computational time for each method, we use the `%timeit` function from Python to obtain:

| Version | Computational time $ms$ (per loop) |
|---|---|
| Original | $95.60 \pm 0.69$ |
| Alternative | $87.10 \pm 6.28$ |

Table 2.1: Computational time for different versions of function `get_model_values`.

We have reduced the computational time by $8$ $ms$, since the factor defined in line $4$ has to be computed only once. Taking into account that Python is an interpreted language, thus the code is executed line by line at runtime, we have reduced the time required by making the lines of code $4, 6, 8$ of Listing $3$ simpler. We also note that the execution time depends on the computer's hardware, which impacts how efficiently the code runs.

Now that we have set the main function of the code, we can continue working with the grid search algorithm. First, we work with `Grid_Search` class in order to study the likelihood function.

```
class Grid_Search:
    def __init__(self, func, n_x, n_y, par_min, par_max, p_names):

        x_min, y_min = par_min
        x_max, y_max = par_max

        chi_sq_array = np.zeros((n_x, n_y))

        x_edges = np.linspace(x_min, x_max, n_x+1)
        y_edges = np.linspace(y_min, y_max, n_y+1)
```

```
11
12         x_centers = 0.5*(x_edges[1:]+x_edges[:-1])
13         y_centers = 0.5*(y_edges[1:]+y_edges[:-1])
14
15
16         # TODO:
17         for i in range(n_x):
18             for j in range(n_y):
19                 chi_sq_array[i,j] = func(x_centers[i],y_centers[j])
20
21         i_min, j_min = np.unravel_index(np.argmin(chi_sq_array), chi_sq_array.
    shape)
22         chi_sq_array -= np.amin(chi_sq_array)
23         print(np.shape(chi_sq_array))
24         self.chi_sq = chi_sq_array
25         self.x = x_centers
26         self.y = y_centers
27         self.ix_min = i_min
28         self.iy_min = j_min
29         self.extent = [x_edges[0], x_edges[-1], y_edges[0], y_edges[-1]]
30         self.p_names = p_names
31         self.gauss_fit_done = False
32
33         print(self.chi_sq[self.ix_min, self.iy_min])
```

Listing 4: Code incorporated into the existing Grid_Search class.

In this class we are creating a grid considering the minimum and maximum values of the masses and concentrations. For constructing the `chi_sq_array`, we compute the value for the centers of each grid cell by averaging neighboring edges. These will be the points in which we are going to compute the chi-squared value, since we want to do an homogeneous search inside the whole grid (lines 12-19 of the above code).

Next, we "normalize" the values of the chi-squared in order for the likelihood to be 1 at the minimum value. As:

$$\mathbb{P}(D|\theta) \propto e^{-\frac{\chi^2(\theta)}{2}} \tag{2.1}$$

we see that the exponent has to be 0, when evaluating at the minimum value of the chi-squared, in order for the exponential to be 1. Thus, we subtract the minimum value of the chi-squared from the original matrix (line 22). We obtain that the minimum value of the chi-sq is 627644.689 (before normalizing, that is, `np.amin(chi_sq_array) = 627644.689`).

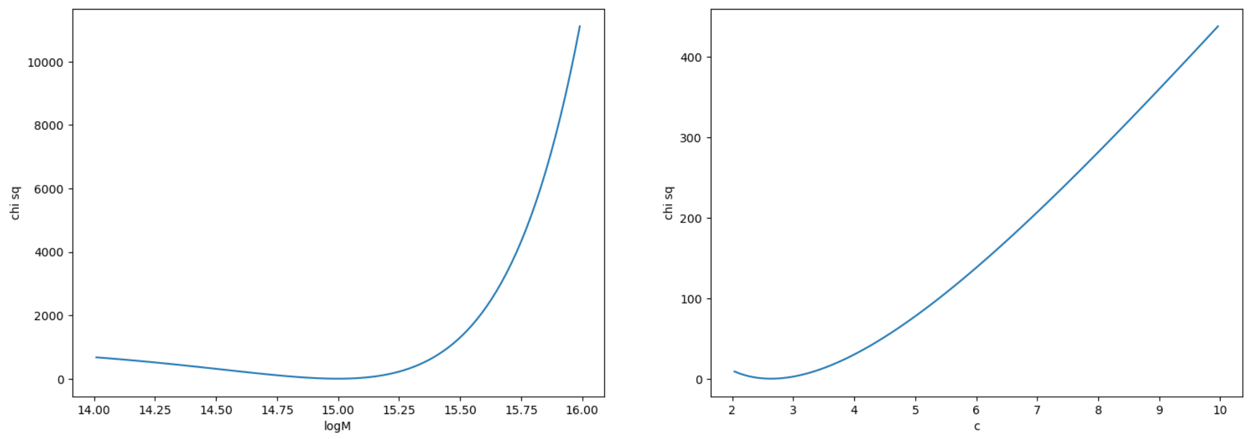


Figure 2.1: Chi-squared function for mass (left) and concentration (right).

We can interpret Figure 2.1 as how likely are the parameter values given the data we have. Considering Equation 2.1, we expect an inverse relationship between the chi-squared function and the likelihood function. Thus, the value that gives the minimum of the chi-squared corresponds to the most likely estimate (MLE) parameter. We see that the slope of the function also tells us about the confidence of this value, since a sharp slope indicates that the parameters of the chi-squared function are much less probable than the minimum ones and a flatter slope indicates that the values are quite less probable, since the value of the chi-squared is similar to the one at its minimum.

In our case, we obtain that for the logarithm of the mass (logM), the MLE corresponds to 14.995 and lower values than the MLE are more probable than higher ones. On the other hand, regarding the concentration (c), the MLE is 2.618 and values higher than it are much less likely than lower ones.

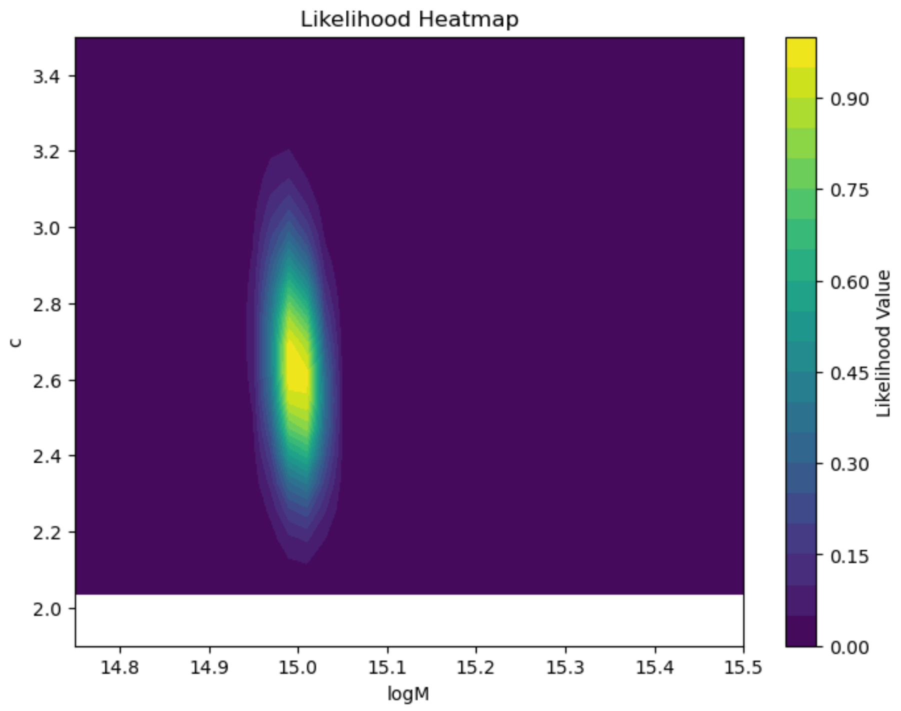


Figure 2.2: Likelihood function of $\theta = (log10M, c)$.

In Figure 2.2, we can see how the value of the MLE in the plot, the parameters of the highest likelihood value, correspond to the minimum value of the $\chi^2$.

## 2.2 Exercise 2

**In the notebook you can find a Sampler class. Implement the Metropolis-Hastings algorithm there. Write the chain to a file as outlined in the notebook. Run some short chains and observe the convergence behaviour. What happens when you change the covariance? Run some longer chains (depending on how fast your $\chi^2$ calculation is) and plot a histogram of the chain values and the two-dimensional distribution.**

In order to implement the Metropolis-Hastings algorithm we complete the `run_mcmc` function within the `Sampler` class. The steps of this algorithm can be found in [3]. For sampling the candidate points at each step we have used a multivariate Gaussian function provided in Python by `numpy.random.multivariate_normal`.

```python
    def run_mcmc(self, p0, cov, n_samples):
        """
        Runs a Monte Carlo Markov Chain.

        Parameters:
        ----------
        p0          : initial parameters
        cov         : covariance matrix of the proposal distribution
        n_samples   : number of mcmc samples
        """
        log_prob0 = self.log_prob(*p0) # initial parameters log likelihood

        samples = np.zeros((len(p0), n_samples)) # matrix for storing all sample points/candidates
        log_prob_arr = np.zeros(n_samples) # also store the values of the log-likelihood
        alpha = np.zeros(n_samples) # probability of accepting each candidate

        samples[:,0] = p0 # store initial parameters as first candidates
        log_prob_arr[0] = log_prob0 # store log likelihood of initial parameters
        accepted = 0 # counter of accpeted candidates
        x = p0

        for i in range(1, n_samples):
            y = np.random.multivariate_normal(x, cov)
            u = np.random.uniform()
            alpha[i] = np.min(np.array([1, np.exp(-self.log_prob(*y)+self.log_prob(*x))]))
            if u <= alpha[i]:
                x = y
                accepted += 1
            else:
                pass
            samples[:,i] = x
            log_prob_arr[i] = self.log_prob(*x)

        self.chain = samples
        self.alpha = alpha
        self.log_prob_arr = log_prob_arr
        self.accepted = accepted
```

Listing 5: Metropolis-Hastings algorithm implementation.

In order to test our `run_mcmc` method we start by generating short chains in which instead of using the real log likelihood we use an interpolated log likelihood given by the `get_gaussian_fit` function in the `Grid_Search` class. This is done for time saving as we will see later how working with the real likelihood takes up much more computational time. In Figure 2.3 we have plotted the likelihood and its gaussian fit using the mentioned function. For this plot the `Grid_Search` class was initialized with $(n_m, n_c) = (100, 110)$, that is the number of grid steps for the two parameters.

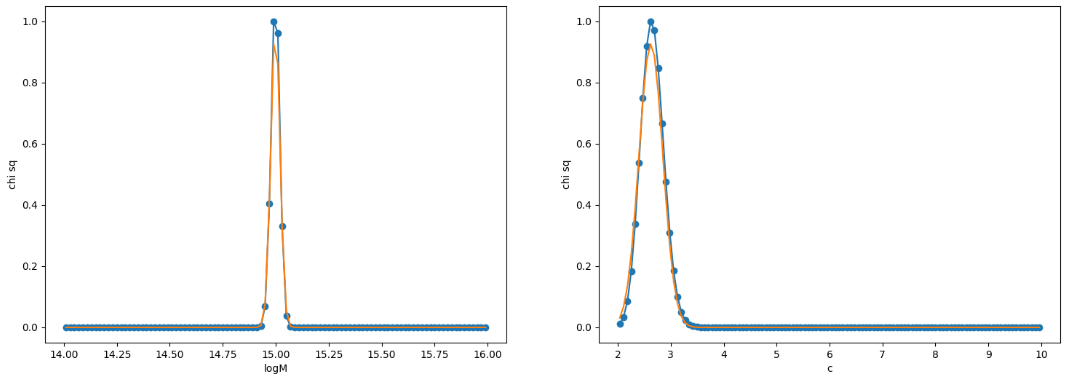


Figure 2.3: The gaussian fit functions (in orange) which best approximate the real likelihood (in blue) for the parameters $\theta = (log10M, c)$.

From Figure 2.3 we see that the gaussian fit of the real likelihood is an accurate approximation for first studying the data. With this in mind we can proceed to the next step which is obtaining the first Markov chains using this interpolated likelihood. Figure 2.4 shows a first attempt of obtaining the Markov chains for both parameters with a number of steps: $n_{samples} = 200$. As can be seen, although the number of steps is not large, both chains show a trend towards the expected values, meaning that convergence is easily achieved using this interpolated likelihood.

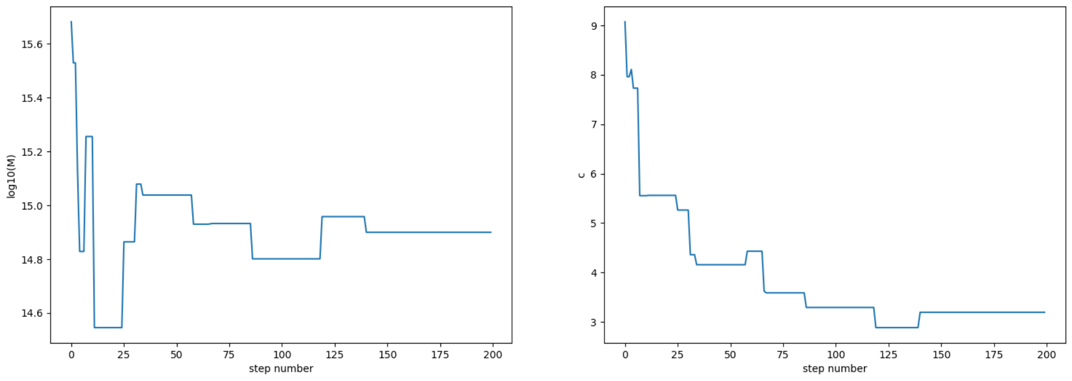


Figure 2.4: Markov chains for the parameters $\theta = (log10M, c)$ $n_{samples} = 200$ and cov=diag(1,1).

For this first case an identity matrix (2x2) was used as covariance. We can now increase the number of steps in the chains, which will help us visualize the values to which they converge. This is showed in Figure 2.5. We can see from both cases, $n_{samples} = 200$ and $n_{samples} = 50000$, that flat sections appear in the chains. As we will see soon, this is due to the choice of the covariance matrix. These flat sections mean that the algorithm is continuously rejecting new candidates so the chain does not explore other possibilities. As we are constructing the MCMC for analyzing the parameter space (since we already have a point estimator for the value of the mass and concentration) we want our chains to have a reasonable rate of acceptance so they are not stuck in a value for too many steps. We apply the criterion introduced in section 1 to properly choose our covariance matrix.

On the other hand, once the chains are obtained for both parameters, we are able to study the frequency with which the values of the chains appear, that is, represent the chains using histograms. Thus, we can approximately see their distribution and compare it with that of a Gaussian, which is our proposed distribution for the Metropolis Hasting algorithm.

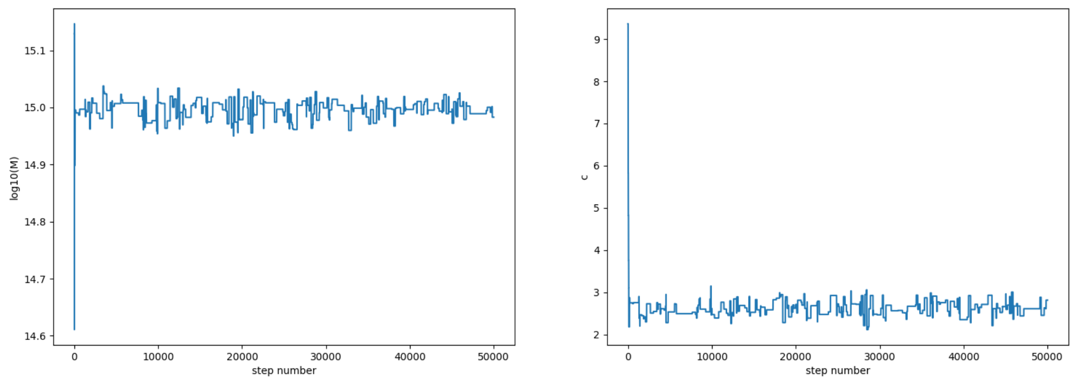


Figure 2.5: Markov chains with $n_{samples} = 50000$ and cov=diag(1,1).

In Figure 2.6, we can see how both the distribution of the masses and concentrations chosen in each step of the chain adjust relatively well to a Gaussian. So our posterior has approximately a Gaussian shape.

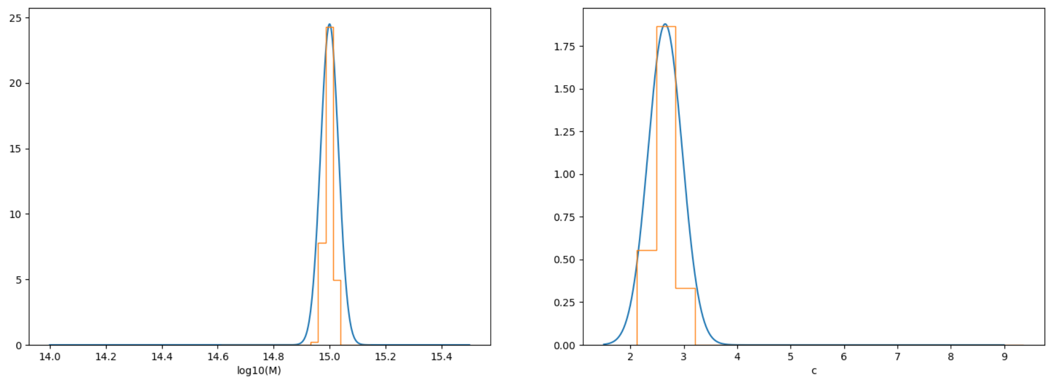


Figure 2.6: Histograms of Figure 2.5 chains.

Now we will change the covariance in order to see how the chains behave and how their convergence is affected by our choice. For this study we will calculate the number of accepted candidates and the acceptance rate for each chosen covariance. The acceptance rate is simply the number of accepted candidates divided by $n_{samples}$.

| Covariance | accepted candidates | acceptance rate |
|---|---|---|
| diag(1,1) | 256 | 0.00512 |
| 0.5 $\times$ diag(1,1) | 534 | 0.01068 |
| 0.2 $\times$ diag(1,1) | 1069 | 0.02138 |
| 0.01 $\times$ diag(1,1) | 8525 | 0.1705 |
| 0.0055 $\times$ diag(1,1) | 11691 | 0.23382 |

Table 2.2: Acceptance rates of the `run_mcmc` method for the different tested covariances for a fixed number of samples ($n_{samples}$ =50000).

According to what was stated in section 1, the choice of the covariance can not be arbitrary. If the covariance is large, there will be a lot of rejection of candidates. This is exactly what happened with the case of the diag(1,1) covariance: candidates were continuously rejected, leading to flat sections in which the chain does not properly explore the parameter space.

Table 2.3 displays the different tested covariance. It can be seen that as the covariance is reduced, the number of accepted candidates increases and so does the acceptance rate. For cov= $0.0055 \times$diag(1,1) we see that roughly one in four candidates is accepted. As described in [4] an acceptance rate of around 0.234 is an optimal choice for the Metropolis algorithm. Figure 2.7 show the chains for a similar value of this optimal acceptance rate. We see how in these chains, flat regions are not noticeable and candidates are being sampled around the values of convergence.

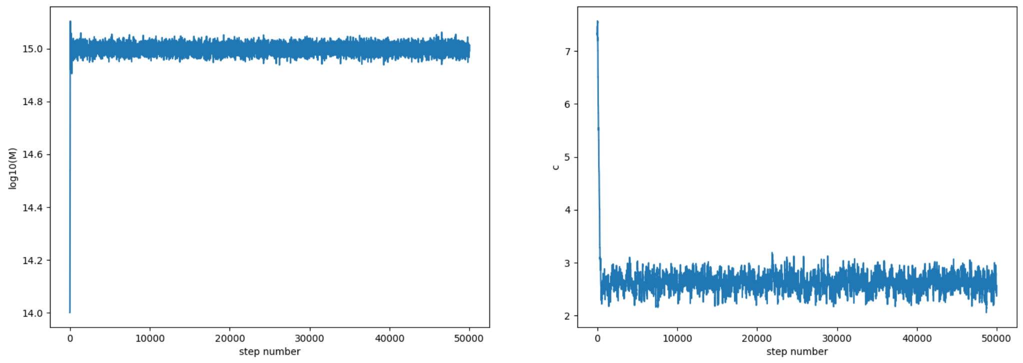


Figure 2.7: Markov chains with $n_{samples} = 50000$ and cov= $0.0055 \times$ diag(1,1). Acceptance rate $\approx$ 0.234.

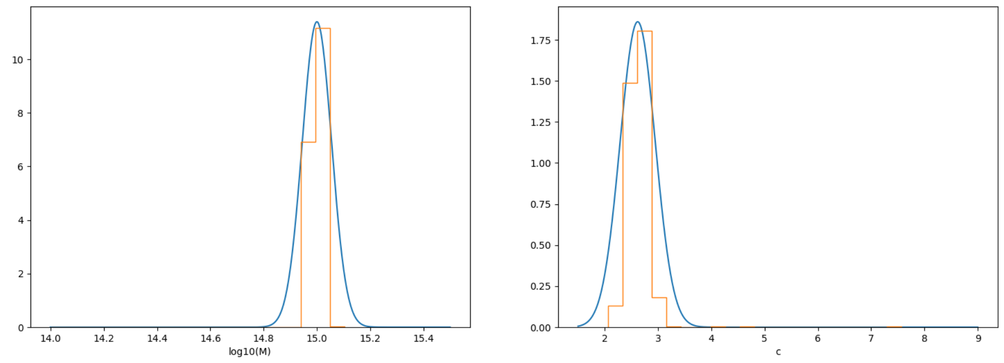


Figure 2.8: Histograms from Figure 2.7 chains.

While the chains are built, we store the likelihood of each pair of candidates as can be seen from Listing 5. Now, in Figure 2.9, we show the joint likelihood distribution of the parameters $\theta = (log10M, c)$. If we compare it with Figure 2.2 we can see how both graphs agree in the estimation of the MLE parameter. That means that the search of this value using the Markov chains produces a satisfactory outcome if compared to the search of the minimum value of $\chi^2$.

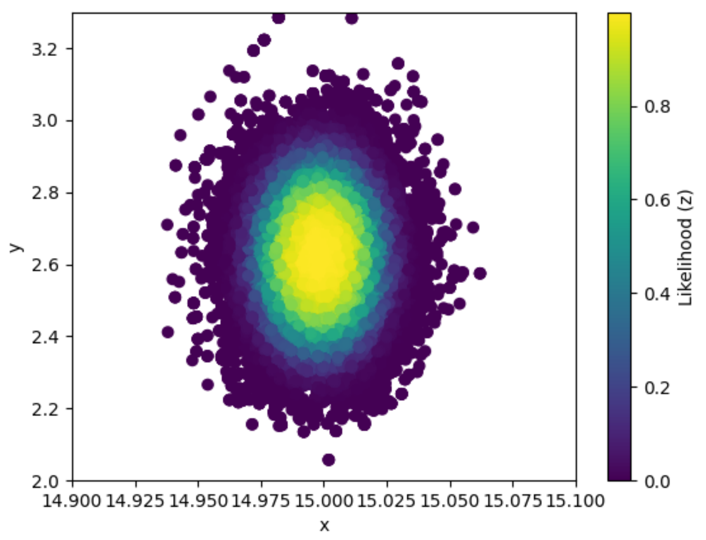


Figure 2.9: Two dimensional distribution of the likelihood. The $x$ axis corresponds to log10(M) and the $y$ axis corresponds to c.

Another thing to have in mind when setting up the Markov chains is the election of the initial values of the parameters. In Figure 2.10 we show an example of a bad choice of initial values. These values where chosen as the ones that minimize the $\chi^2$ function, meaning that the chains should actually converge towards these values. As a result, the chains are not probably going to study the parameter space properly. Furthermore it can be the case that the chosen initial parameters are not actually the global minimum but a local one. In this case the chains can get stuck in this local minimum and not be able to find the global minimum values.

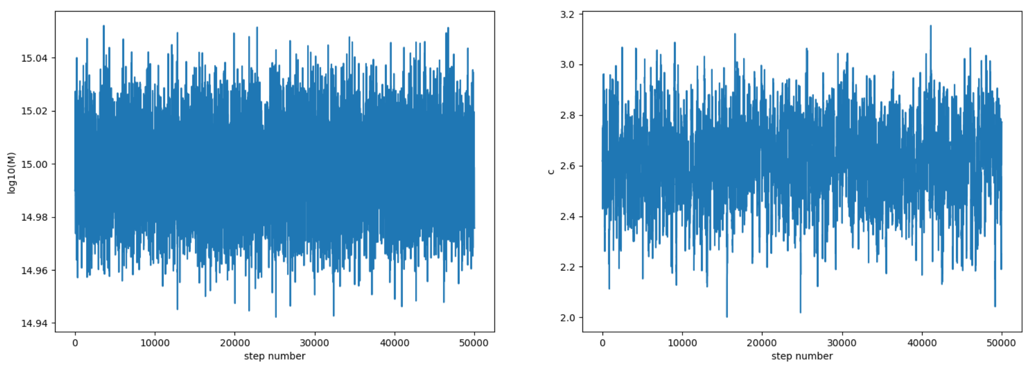


Figure 2.10: Markov chains with initial values chosen as those which minimize the value of $\chi^2$.

### 2.3 Exercise 3

**Generate at least 5 different chains with the code written in Exercise 2 and analyze the chains. Discuss your results and compare to the result of Exercise 1. Use at least a number of samples that $\mathbf{R - 1 < 0.05}$.**

In this part of the project, we are going to study the dependence of the convergence of the chains on the initial parameters. As stated, our mission when working with MCMC is to explore the parameter space as much as needed, to find a consistent distribution for the values of the mass and the concentration. To do so, we set the initial parameters to span over all the allowed range, build 5 different chains and run them simultaneously.

```python
f = plt.figure(figsize = (18,6))

ax1 = plt.subplot(121)
ax2 = plt.subplot(122)

n_samples = 5000
n_chains = 5
p0 = np.array([[14.,2.], [14.5,3.], [15.,5.], [15.5,6.], [16.,9.]])
psi_m = np.zeros((n_chains, n_samples))
psi_c = np.zeros((n_chains, n_samples))

cov_def = 0.0055 *  np.diag((1, 1))
sampler = Sampler(cluster.log_prob)
for i in tqdm(range(n_chains)):
    sampler.run_mcmc(p0[i,:], cov_def, n_samples)
    sampler.plot_chain(axes = np.array([ax1, ax2]), p_names = np.array(['log10(M)'
    , 'c']))
    psi_m[i,:] = sampler.chain[0,:]
    psi_c[i,:] = sampler.chain[1,:]

sampler.R_conv(psi_m, n_samples, n_chains)
sampler.R_conv(psi_c, n_samples, n_chains)
```

Listing 6: Code used for plotting the 5 Markov chains.

To compute the value of $R$ introduced in section 1, we add a method to the `Sampler` class.

```python
        def R_conv(self, x, n, m):
        """
        x : matrix of chains
        n : number of samples
        m : number of chains
        """
        psi_m_cad = np.mean(x, axis = 1)

        psi_m_mean = np.mean(psi_m_cad)

        B_m = n / (m - 1.) * np.sum((psi_m_cad - psi_m_mean)**2)

        s2_m = 1. / (n - 1.) * np.sum((x[0, 0:n] - psi_m_cad[:, np.newaxis])**2,
    axis = 1)

        W_m = np.mean(s2_m)


        var_m = (n - 1.) / n * W_m + 1. / n * B_m
        R_msq = np.sqrt(var_m / W_m)

        return R_msq
```

Listing 7: Convergence criterion implementation.

We will compute its value and consider that the set of chains has converged if $|R-1| < 0.05$.

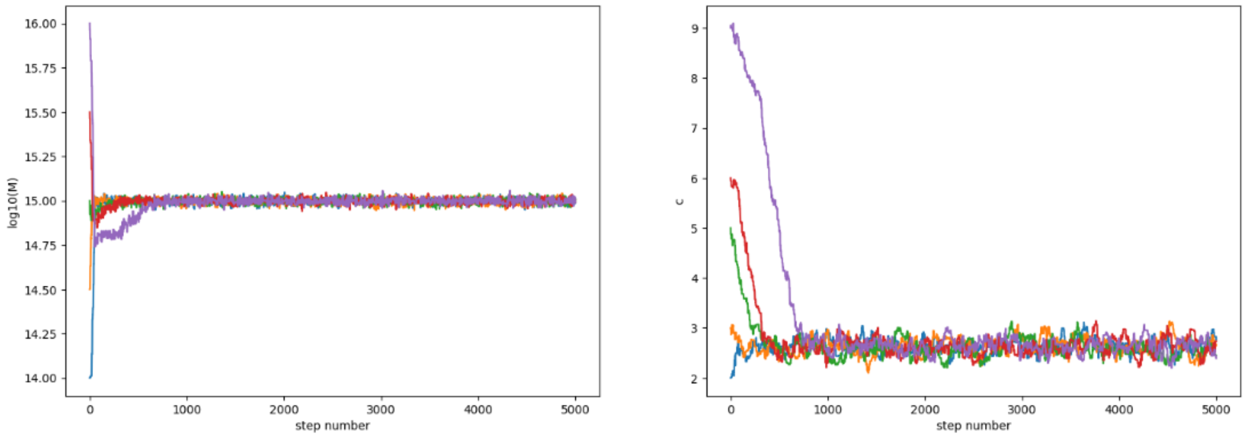


Figure 2.11: Five Markov chains using the real likelihood function with $n_{samples} = 5000$.

We can see how after different number of steps all the chains end up around the same value. The fact that they agree in the limiting value provides us with high confidence in our point estimators: after analyzing a wide range of alternative parameters the chains decide for the optimal ones. However, the chains for the concentration parameter are not as stable as the ones for the mass, as it can be seen in Figure 2.11. We note that the range for the initial concentrations is much bigger than the one for the mass, so it was expected that this set needed more steps to achieve the same level of accuracy as the mass.

| M0 range | c0 range | $n_{samples}$ | $\sqrt{R_m}$ | $\sqrt{R_c}$ | $\lvert R_m - 1\rvert$ | $\lvert R_c - 1\rvert$ |
|---|---|---|---|---|---|---|
| [14, 16] | [2, 9] | 5000 | 1.0014 | 1.2774 | 0.0028 | 0.6317 |
| [14, 16] | [2, 4] | 500 | 1.0082 | 1.1223 | 0.0165 | 0.2596 |
| [14, 16] | [2, 3] | 250 | 1.0271 | 1.1348 | 0.0549 | 0.2877 |

Table 2.3: The different Markov chains generated with the real likelihood function. The $R$ indicator of convergence is also calculated for both parameters.

From Table 2.3, we can see that the chains for the mass with $n_{samples} = 5000$ have converged but the chains for the concentration have not. To reduce the value of $R$ we consider the initial concentrations to be more concentrated (worth the redundancy), so to resemble the mass ones. With $n_{samples} = 500$ and $c0 \in [2, 4]$, the value of R has been reduced although it still does not satisfy our convergence condition. We note that the stationary value for the mass is located near the mean of the initial interval, so we try to do this for the concentration parameter. We do not obtain a satisfactory result either.

As for our machines each of these computations take a considerable amount of time (p.e., the 5 chains with $n_{samples} = 5000$ took more than 4h), we decide not to try with longer chains, since we have already studied the behavior of the chains. We see that reducing the interval for the initial concentrations has been useful, so if one runs either the second or third proposed chains in Table 2.3 (see Appendix B for the plotted chains) for a larger number of steps, the set should converge. We remark the fact that the covariance matrix is diagonal and has the same variance for each parameter. This could lead us to think that the chains should have the same behavior. However, we note that the magnitudes of the variables differ by one order of magnitude, which means that the variances do not affect in the same way each of them. We could have tried to change the concentration variance, leading to a matrix not proportional to the diagonal, but this would have changed our criterion for the selection of the covariance.

## 3 Conclusions

In this project, we have developed a set of steps in order to find out the mass and concentration of a galaxy cluster which acts as a gravitational lens. We have compared the given measured gravitational shear data with the one obtained from a theoretical model, which depends on the virial mass ($M_{200}$) and the concentration ($c_{200}$). Once this model was computed and optimized, we searched for the values of these mentioned parameters which best adjusted the model to the data. To do so, we focused on determining the $\chi^2$ value associated to the parameters (Equation 1.1) and from it the likelihood.

First, in order to alleviate the required computational time, we computed an interpolated version of the $\chi^2$ using the `Grid_Search` class (subsection 2.1). We observed how this $\chi^2$ function had a minimum for both parameters which corresponds to the maximum value of the likelihood as could be inferred from Equation 2.1. This allowed us to obtain a first point estimate for the mass and concentration: $\log(M_{200}) = 14.995$, $c = 2.618$.

Our next step, subsection 2.2, was to implement in Python the `run_mcmc` method that performs the Metropolis-Hastings algorithm. To check the reliability of the method, we then tested it with the interpolated likelihood function. Figure 2.4 and Figure 2.5 showed that our Markov chains converged towards the expected values obtained previously from the minimum of the $\chi^2$. Once this test was done, we proceeded to study the dependence of the chains with the covariance. After building several chains with different covariances and following the criterion presented in [4] we conclude that our best choice of covariance was $0.0055 \times diag(1, 1)$. This choice of covariance resulted in an acceptance rate of almost 1 in 4 candidates, meaning that the chains test enough candidates to avoid getting stuck in flat regions but not excessively, which helps prevent slow convergence. This covariance was used to plot the chains in Figure 2.7 from which we obtained the two dimensional distribution of the joint likelihood, which agrees with the one obtained from the $\chi^2$.

Finally, in subsection 2.3, we have built the Markov chains with the real likelihood while studying the parameter space. Due to lack of computational resources we ended up generating chains of a maximum of $5000$ steps. We realized that although the mass was converging to the expected value satisfying the convergence criterion, this was not happening with the concentration. After changing the initial range for the parameters of this magnitude, we noted that more steps in the chain were still needed in all the cases (in order to achieve convergent chains), so we do not try to change the form of the covariance matrix in the Metropolis Hasting algorithm as a possible method for increasing the convergence.

We can conclude that our fiducial analysis is consistent with the initial point estimators and that it provides us a tool for exploring the whole range of possible parameters. Thus, we have a sense of all the possible candidates and their probability for mass and concentration of the galaxy cluster. We note that the computational time in this kind of analyses is crucial for the quality of the results and we encourage the reader to build longer Markov chains for the concentration as future work, since with a higher number of samples it will surely converge.

## 4 Author's note

We are aware of the maximum extension of the report (12 text pages). Hence when considering only the written part of this report (not taking into account neither codes nor figures), it is about 9 pages.

## Bibliography

[1] Lucas Frozza Secco et al. "Dark Energy Survey Year 3 results: Cosmology from cosmic shear and robustness to modeling uncertainty". In: *Physical Review D* 105.2 (2022), p. 023515 (cit. on p. 2).

[2] J. Cohn. *Gravitational Lensing*. https://w.astro.berkeley.edu/~jcohn/lens.html. Accessed: Nov, 2024 (cit. on p. 2).

[3] Stella Seitz and Jochen Weller. *Manual*. https://www.dropbox.com/scl/fo/8efzzhmxjjhgns03h7oj1/AKqmpoHq3T-zlWE4uQ2B9hI?rlkey=v494swbp4cbsta8mp0293na4n&st=g0vyxymw&dl=0. Accessed: Nov, 2024 (cit. on pp. 2–4, 8).

[4] Andrew Gelman, Walter R Gilks, and Gareth O Roberts. "Weak convergence and optimal scaling of random walk Metropolis algorithms". In: *The annals of applied probability* 7.1 (1997), pp. 110–120 (cit. on pp. 3, 11, 15).

## A Codes from the Manual

```python
def log_prob(self, logM200, c200, normalize=False):
    """
    Calculates the log of the likelihood for a given set of parameters

    Parameters:
    ----------
    logM200    : log10 of the cluster mass
    c200       : cluster concentration

    Returns:
    ----------
    log-likelihood of the data
    """

    # check if parameters are outside of the prior range
    if self.outside_param_range([logM200, c200]):
        return float('inf')
    M200 = 10**logM200
    y_model = self.get_model_values(M200, c200)
    y_data = self.shear
    y_err = self.shear_err
    log_prob = self.chi_sq(y_model, y_data, y_err)
    if normalize:
        ndof = len(y_data)-2
        log_prob /= ndof
    return log_prob
```

Listing 8: Function that computes the chi-square with our data.

## B Real likelihood chains

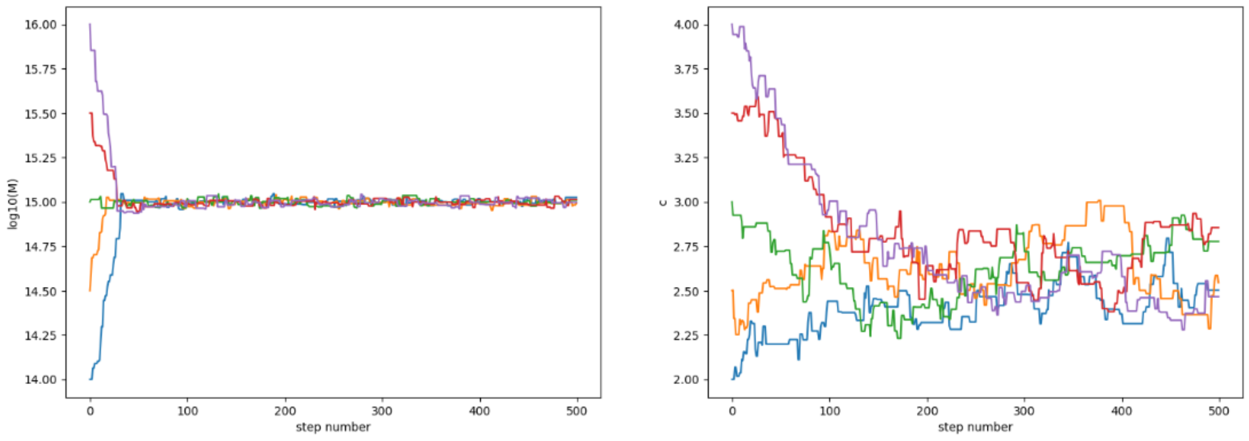
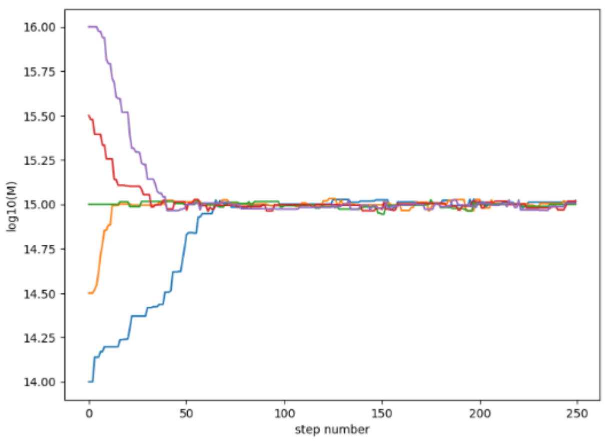


Figure B.1: Markov chains with $n_{samples} = 500$, M0 range$= [14, 16]$ and c0 range$= [2, 4]$



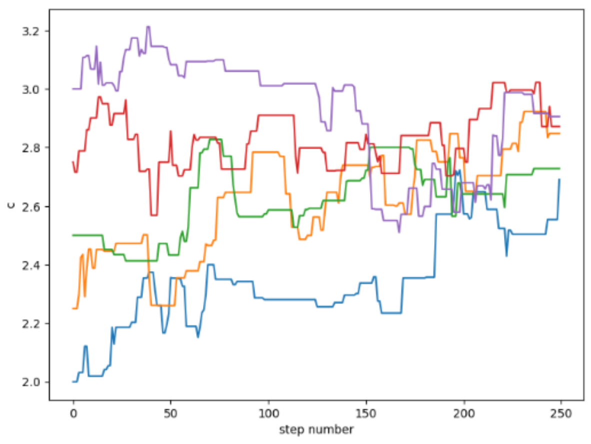


Figure B.2: Markov chains with $n_{samples} = 250$, M0 range= $[14, 16]$ and c0 range= $[2, 3]$