MCMC Assignment 1

Grace Fiacco

March 2022

Introduction

This assignment has you create an MCMC algorithm to explore a Student-t distribution and produce N draws from the distribution. The likelihood I used was the shifted/scaled Student-t distribution given in the problem, and I implemented a flat prior, where my sample draws could not go beyond $\sigma \pm 4$, which was an arbitrary choice. For the likelihood I used $\nu = 3, \mu = 1, \sigma = 1$ for my fixed parameters.

Since I am generating a data set from this likelihood, each proposed jump will be a point in the set, so the parameter space I am exploring will be x in the Student-t distribution. I am using a Gaussian proposal distribution for my jumps, as stated in the assignment.

Part A

For the proposal distribution, I tried out four different jump sizes: $\alpha = 0.01, \alpha = 0.1, \alpha = 1.0$, and $\alpha = 10.0$. Trying out all four of these scalings, I realized that jump size makes a huge difference in how many iterations one needs to effectively explore the parameter space. A smaller jump size means you would need more iterations, since the code will take longer to reach the farther edges of the space. Meanwhile a larger jump size would need less iterations to cover the same ground.

For the four scalings that I explored, I found that for $\alpha = 10$ I actually needed more iterations than for $\alpha = 1$. This does make sense since our mean and standard deviation of the distribution is much smaller than 1, so using a jump that is too large would miss the small details at first. Looking at this made me realize how important picking a good jump size actually is for the efficiency of the MCMC.

Figure 1 shows a posterior plot of the $\alpha = 0.1$ scaling, run at a high resolution. With enough iterations of the code, you can see that the MCMC draws enough points to almost perfectly fit the analytic curve of the distribution. This is the best check for if the code is working correctly; with the number of iterations approaching infinity, you will get out the exact distribution you are trying to simulate.

To visually inspect the efficiency of my four scaling choices, I used the trace plotting method to see each chain. I ran all four scalings for 10,000 iterations.

To do a trace plot, I plotted the iteration number against the accepted jump value at that point. These plots are shown in Figure 2.

As expected, $\alpha = 1.0$ and $\alpha = 10.0$ do the best job of exploring the parameter space with 10,000 iterations. Since our mean and standard deviation are 1, values close to that would be the best choices for a jump size. You can also clearly see that for the smaller α values, 10,000 iterations is nowhere near close enough to properly explore all of the parameter space.

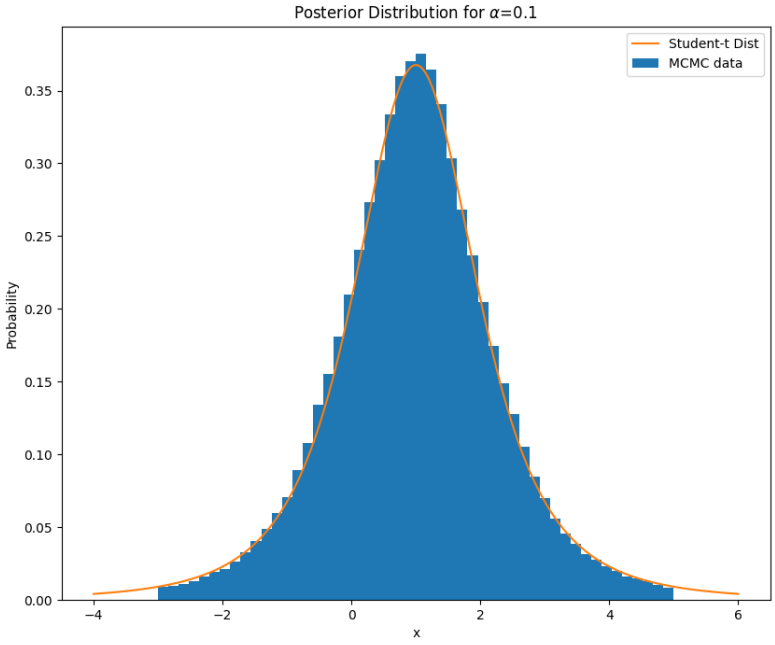

Figure 1: Posterior distribution for the simulated Student-t data set with $\alpha = 0.1$ and 1,000,000 iterations. The analytic curve is plotted over the data.

Part B

For 10,000 iterations, I computed the acceptance fraction for each α value. This tells you how many of the proposed jumps are accepted out of the total number of iterations, which helps measure the code's efficiency.

For $\alpha = 0.01$, the acceptance rate was 0.9944. For $\alpha = 0.1$, it was 0.9712. These are incredibly high acceptance rates for how poorly these chains explored the space, showing that having a high acceptance doesn't necessarily equate to a more efficient code.

For $\alpha = 1.0$, the acceptance rate was 0.7232 and for $\alpha = 10$, it was 0.1481. These are much lower rates than the smaller scaling, but explored the distribution much more efficiently, as you can see in Figure 2.

Part C

For this part, I ran all four jump sizes for 10,000 iterations and plotted the histogram against the target distribution. The graphs are shown in Figure 3. The distribution that fits the best is for $\alpha = 1.0$, which makes sense.

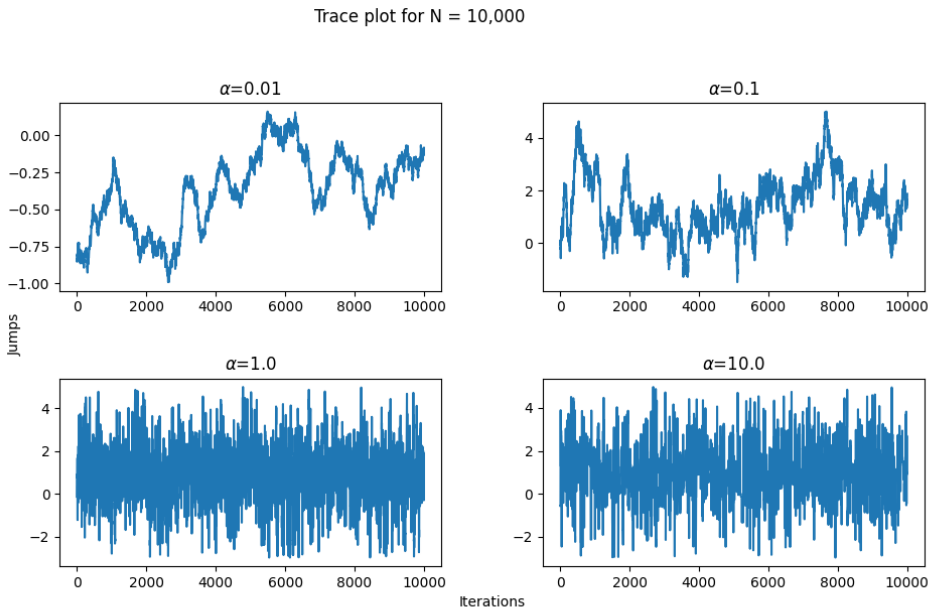

Figure 2: Trace plots for each scaling value and 10,000 iterations. $\alpha = 1$ explores this space the best while $\alpha = 0.01$ does the worst job.

Part D

To calculate the auto-correlation for each jump size, I used the Python stats function `plot_acf`. This plots the number of lag against the auto-correlation coefficient, and the lag where the auto-correlation coefficient is 0.01 tells the auto-correlation length. I could not get an auto-correlation length for $\alpha = 0.01$, but I got values for the other three jump sizes, which are in the table below. Figure 4 also shows an example graph of the auto-correlation for $\alpha = 1.0$.

α	$\rho(h)$
10.0	37
1.0	30
0.1	1330
0.01	??

Part E

I computed the Fisher information matrix for my Student-t log likelihood, first for the parameters $\nu = 3, \mu = 1, \sigma = 1$. To do this, I used the Python package `sympy` to calculate the partial derivatives and evaluate the

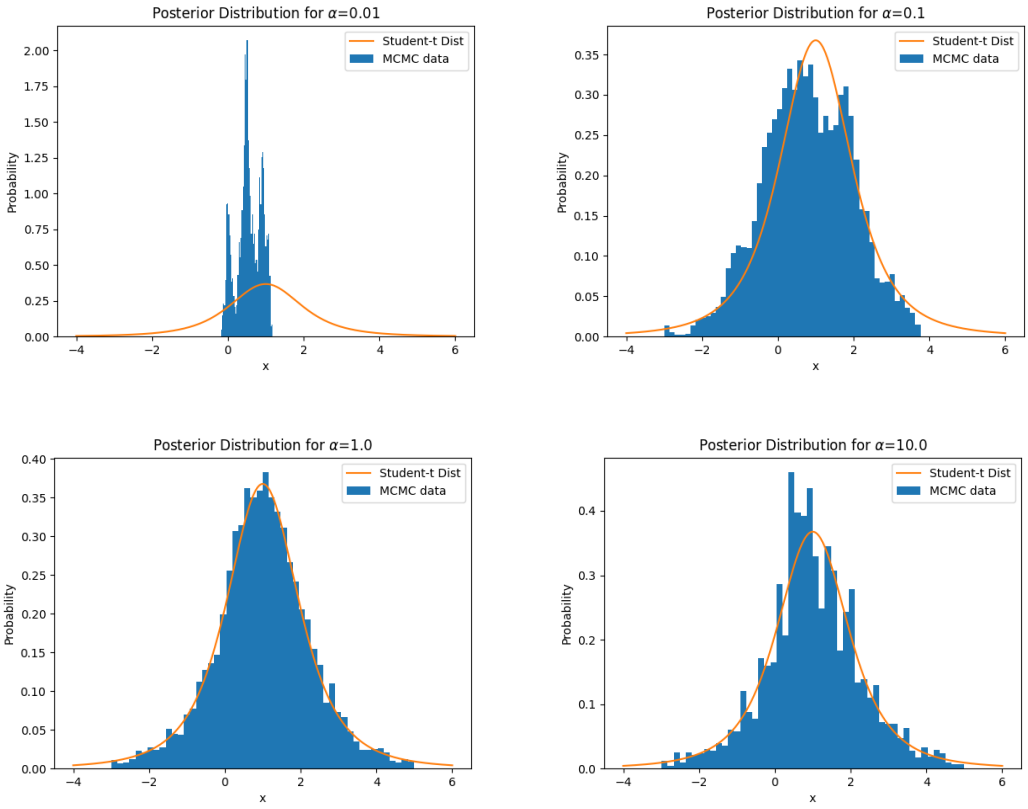

Figure 3: Histograms of the four jump sizes at 10,000 iterations.

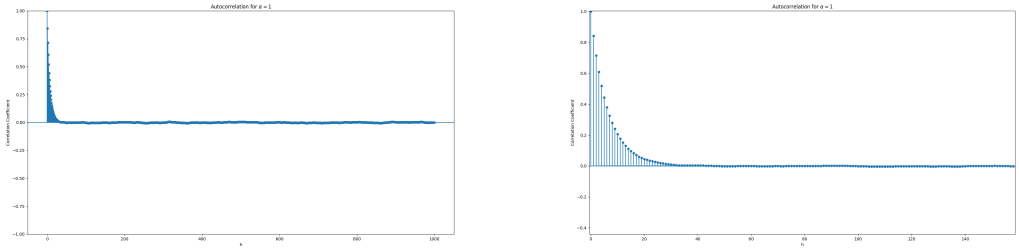

Figure 4: Auto-correlation graph for $\alpha = 1.0$. The right figure is a zoomed in version of the left, showing more detail.

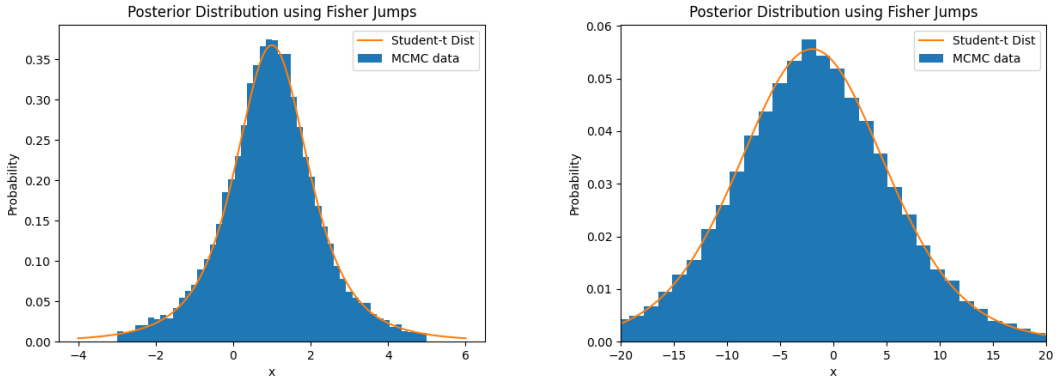

Figure 5: Posterior distributions using Fisher proposal jumps. The left figure is for $\nu = 3, \mu = 1, \sigma = 1$, and the right is for $\nu = 10, \mu = -2, \sigma = 7$. Using a Fisher proposal is a good choice, as it selects an appropriate jump size very well.

expression at the maximum. For these parameters, I got a value of $\Gamma_{xx} = 1.33$, which represents the eigenvalue for the space the MCMC explores. To relate this to the jump size for my MCMC, you create a jump that is $\frac{\delta}{\sqrt{\lambda}}$, where δ is a unit normal deviate (mean of 0, sigma of 1), and λ is the value from the Fisher matrix.

For the parameter set $\nu = 10, \mu = -2, \sigma = 7$, I calculated a Fisher eigenvalue of $\Gamma_{xx} = 0.0212$. I plotted the posterior distributions for both of these using my Fisher jumps, and they are shown in Figure 5. The Fisher jumps did a very good job at exploring the parameter space effectively, creating a spot-on distribution.