

# **Metropolis-Hastings for bivariate densities**

Project 1 in MA8702. Written by Elling Svee.

February 06, 2026

## 1. Example densities

In this project we implement three variations of the *Metropolis-Hastings* (MH). For evaluating the algorithms we consider three bivariate target densities for  $\mathbf{x} = (x, y)^\top$ :

1. **Gaussian distribution:** The first is a bivariate Gaussian distribution with correlation. Its probability density function (PDF) is

$$\pi(\mathbf{x}) = \frac{1}{2\pi\det(\Sigma)^{\frac{1}{2}}} \exp\left(-\frac{1}{2}\mathbf{x}^\top \Sigma^{-1} \mathbf{x}\right)$$

where  $\Sigma$  has 1 on the diagonal and 0.9 on the off diagonals.

2. **Multimodal density:** The second is a multimodal density constructed as a mixture of Gaussian densities. Its PDF is

$$\pi(\mathbf{x}) = \sum_{i=1}^3 w_i \frac{1}{2\pi\det(\Sigma_i)^{\frac{1}{2}}} \exp\left(-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu}_i)^\top \Sigma_i^{-1} (\mathbf{x} - \boldsymbol{\mu}_i)\right),$$

with  $w_i = 1/3$  for  $i = 1, 2, 3$ . The means are  $\boldsymbol{\mu}_1 = (-1.5, -1.5)^\top$ ,  $\boldsymbol{\mu}_2 = (1.5, 1.5)^\top$  and  $\boldsymbol{\mu}_3 = (-2, 2)^\top$ , and the covariance matrices all have correlation 0 and variances  $\sigma_1^2 = \sigma_2^2 = 1$  and  $\sigma_3^2 = 0.8$ .

3. **Volcano density:** Lastly we consider a volcano-shaped density with PDF

$$\pi(\mathbf{x}) \propto \frac{1}{2\pi} \exp\left(-\frac{1}{2}\mathbf{x}^\top \mathbf{x}\right) (\mathbf{x}^\top \mathbf{x} + 0.25)$$

The densities are implemented in the `densities.py` file. Figure 1 visualizes the three densities on a grid covering  $[-5, 5] \times [-5, 5]$ . The grid spacing is 0.1, which gives in total  $101 \times 101$  grid cells.

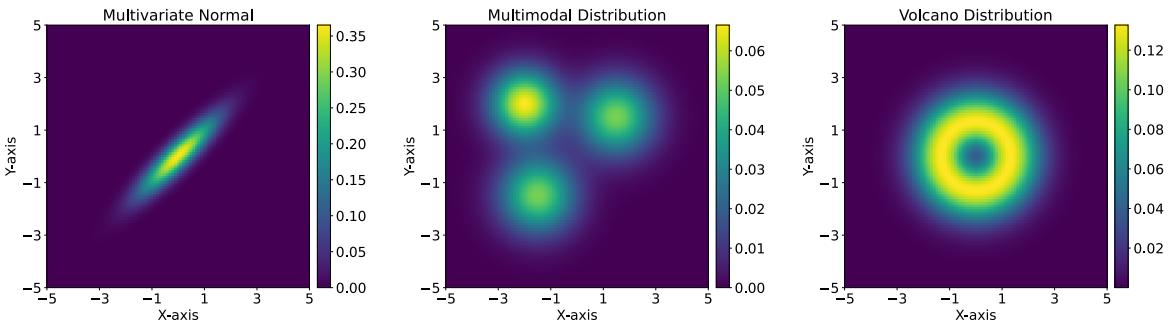


Figure 1: Bivariate densities on a  $[-5, 5] \times [-5, 5]$  domain.

## 2. Theoretical background for Metropolis-Hastings

The MH algorithm is a *Markov chain Monte Carlo* (MCMC) method for sampling from a target distribution  $\pi(\mathbf{x})$  known only up to a normalizing constant. Given a current state  $\mathbf{x}^{(t)}$ , the algorithm proceeds as follows:

1. **Propose** a candidate  $\mathbf{x}'$  from a proposal distribution  $q(\mathbf{x}' | \mathbf{x}^{(t)})$ .
2. **Compute** the acceptance probability

$$\alpha(\mathbf{x}^{(t)}, \mathbf{x}') = \min\left(1, \frac{\pi(\mathbf{x}')q(\mathbf{x}^{(t)}|\mathbf{x}')}{\pi(\mathbf{x}^{(t)})q(\mathbf{x}'|\mathbf{x}^{(t)})}\right).$$

3. **Accept** the proposal with probability  $\alpha$ : set  $\mathbf{x}^{(t+1)} = \mathbf{x}'$ . Otherwise, set  $\mathbf{x}^{(t+1)} = \mathbf{x}^{(t)}$ .

The acceptance ratio ensures the chain satisfies *detailed balance* with respect to  $\pi$ :

$$\pi(\mathbf{x})P(\mathbf{x} \rightarrow \mathbf{x}') = \pi(\mathbf{x}')P(\mathbf{x}' \rightarrow \mathbf{x}),$$

which guarantees that  $\pi$  is a stationary distribution of the chain.

### 3. Code setup

The three MCMC algorithms are implemented in Python using the JAX (Bradbury et al., 2018) library for linear algebra and *automatic differentiation* (AD). This enables us to compute the gradients of the log-densities required for the Langevin and Hamiltonian algorithms without having to specify them manually. JAX also has functionality for just-in-time (JIT) compilation, which we use to speed up the sampling process, and for running multiple chains in parallel. The implementation is overall inspired by the great Blackjax library (Cabezas et al., 2024), which provides modern and efficient implementations of many MCMC algorithms.

When developing the code, I found it most sensible to implement the three algorithms as separate *kernels*, and write a single function for that takes a specific kernel as input and runs the chains. This inference function is then used for all three algorithms, and allows for a general and reusable implementation. The code can be found in `scripts/inference.py`.

## 4. Random-walk Metropolis-Hastings

### 4.1. Theory

Random-walk MH uses a symmetric proposal centered at the current state:

$$q(\mathbf{x}'|\mathbf{x}) = \mathcal{N}(\mathbf{x}'; \mathbf{x}, \sigma^2 \mathbf{I}),$$

where  $\sigma > 0$  is the step size. Since the proposal is symmetric, i.e.,  $q(\mathbf{x}'|\mathbf{x}) = q(\mathbf{x}|\mathbf{x}')$ , the acceptance probability simplifies to

$$\alpha(\mathbf{x}, \mathbf{x}') = \min\left(1, \frac{\pi(\mathbf{x}')}{\pi(\mathbf{x})}\right).$$

The step size  $\sigma$  controls the trade-off between exploration and acceptance rate. A small  $\sigma$  yields high acceptance but slow exploration; a large  $\sigma$  proposes distant points but with low acceptance. For high-dimensional targets, optimal scaling theory (Roberts and Rosenthal, 2001) suggests tuning  $\sigma$  to achieve an acceptance rate of approximately 0.234.

Building the Random-walk MH kernel is done in `scripts/random_walk.py`. I use the tuples `RWState` for storing the intermediate states of the chain, and `RWInfo` for storing information about the acceptance rate and other diagnostics. The `build_kernel()` function takes in the log-density function and a step size, and returns a kernel function that takes in a state and

returns the next state and info. As discussed previously, this function is passed into the `inference_loop()` function that runs the chain.

## 4.2. Results

### 4.2.1. Gaussian distribution

Figure 2 shows the tuning experiment for the Gaussian distribution. Observe that  $\sigma = 1.5$  gives an acceptance rate of 0.264. This is closest to the theoretical optimal, and is therefore preferred.

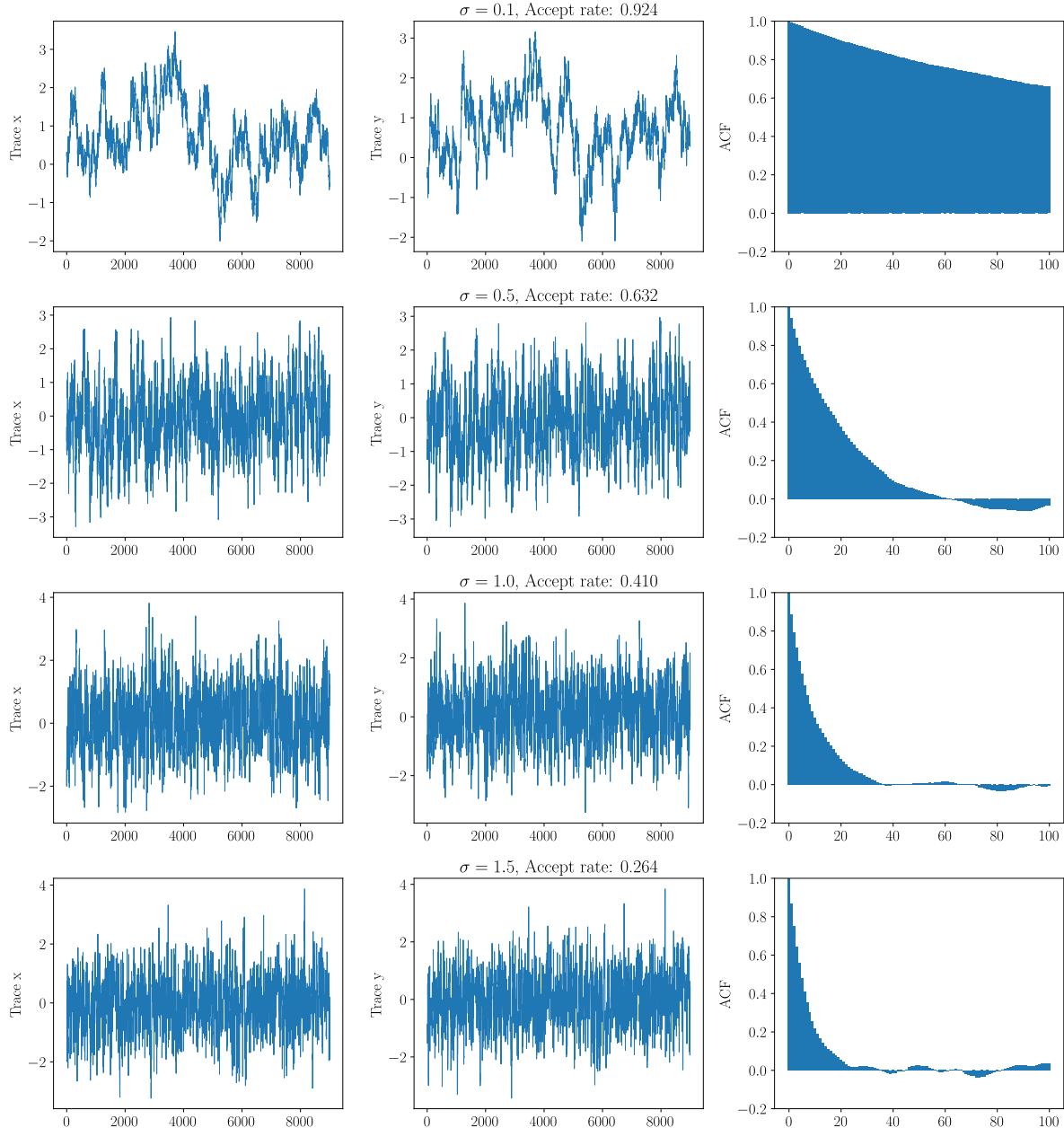


Figure 2: Tuning experiment of Random-walk MH for Gaussian distribution

### 4.2.2. Multimodal distribution

Figure 3 shows the tuning experiment for the multimodal distribution.  $\sigma = 1.5$  gives an acceptance rate of 0.503. This is closest to the theoretical optimal, and is therefore preferred.

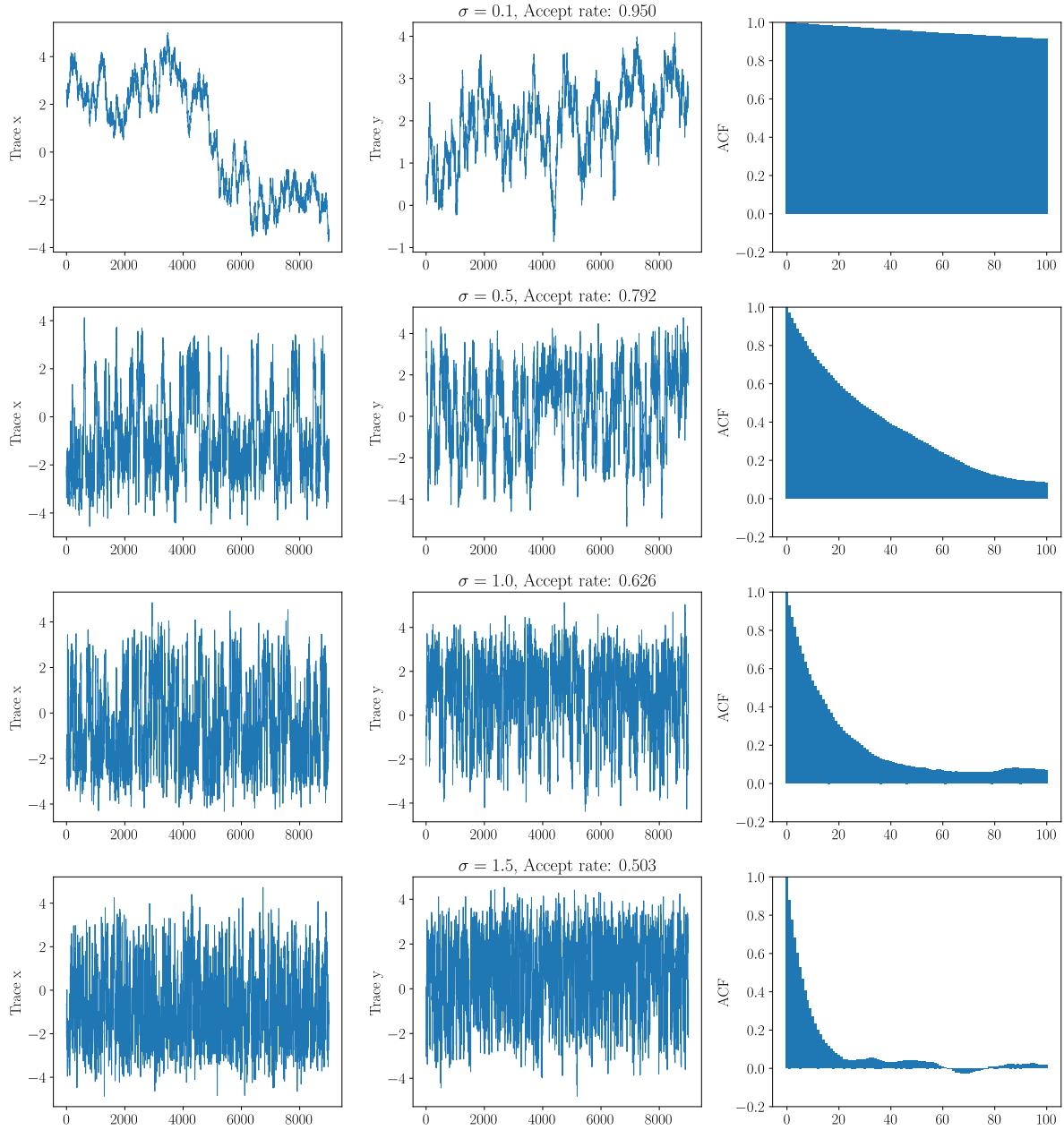


Figure 3: Tuning experiment of Random-walk MH for multimodal distribution

#### 4.2.3. Volcano distribution

Figure 4 shows the tuning experiment for the volcano distribution.  $\sigma = 1.5$  gives an acceptance rate of 0.523. This is closest to the theoretical optimal, and is therefore preferred.

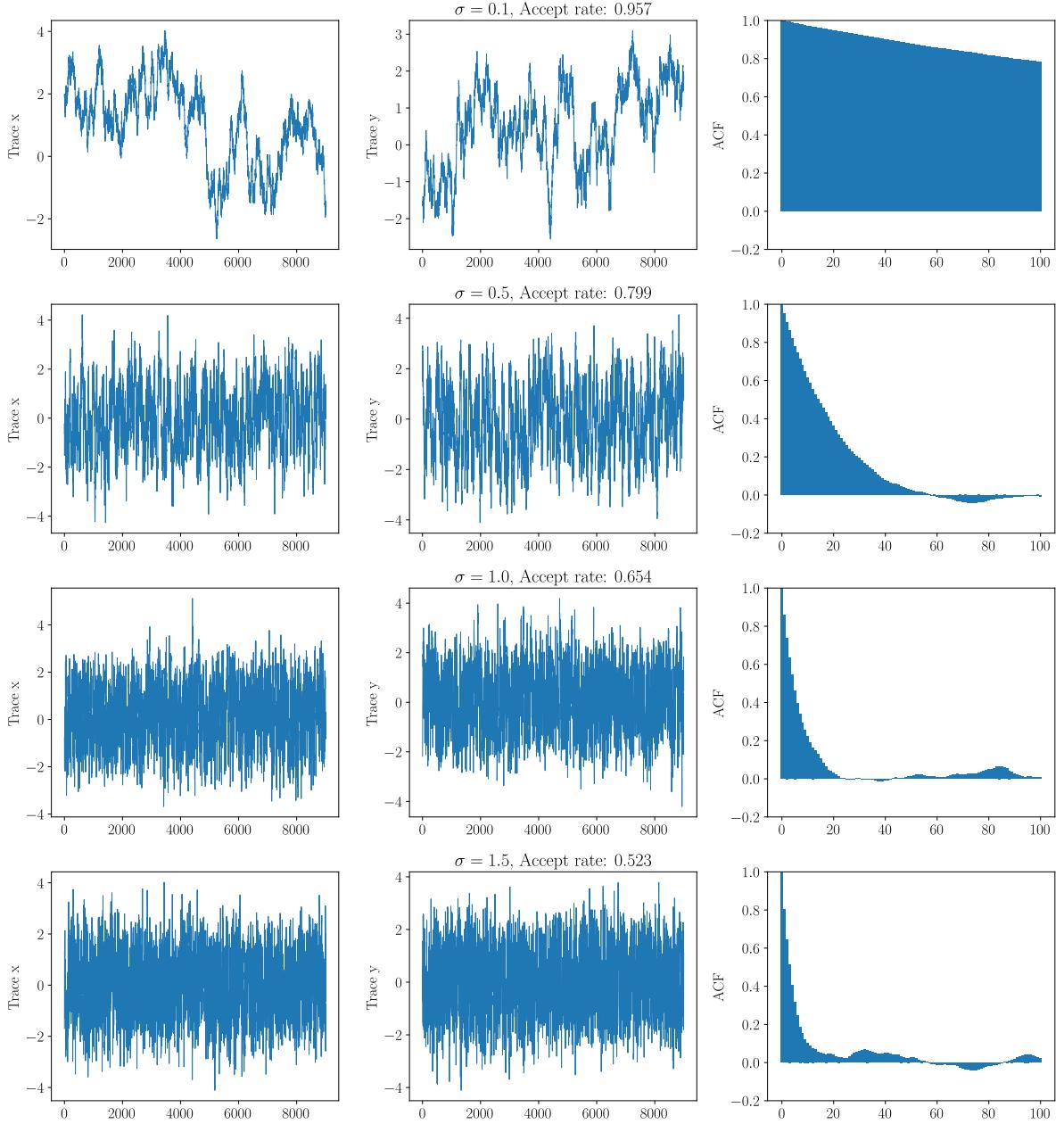


Figure 4: Tuning experiment of Random-walk MH for volcano distribution

## 5. Langevin Metropolis-Hastings

### 5.1. Theory

The *Metropolis-adjusted Langevin algorithm* (MALA) incorporates gradient information into the proposal. It is motivated by the Langevin diffusion, the stochastic differential equation

$$d\mathbf{X}_t = \nabla \log \pi(\mathbf{X}_t) dt + \sqrt{2} d\mathbf{W}_t,$$

which has  $\pi$  as its stationary distribution. Discretizing with step size  $\varepsilon$  yields the proposal

$$q(\mathbf{x}'|\mathbf{x}) = \mathcal{N}(\mathbf{x}'; \mathbf{x} + \varepsilon \nabla \log \pi(\mathbf{x}), 2\varepsilon \mathbf{I}).$$

Unlike the random-walk proposal,  $q(\mathbf{x}'|\mathbf{x}) \neq q(\mathbf{x}|\mathbf{x}')$  mean this is *not* symmetric. Therefore, we must use the full MH acceptance probability

$$\alpha(\mathbf{x}, \mathbf{x}') = \min\left(1, \frac{\pi(\mathbf{x}')q(\mathbf{x}|\mathbf{x}')}{\pi(\mathbf{x})q(\mathbf{x}'|\mathbf{x})}\right).$$

The gradient mean proposal move toward high-density regions, enabling larger step sizes and faster mixing compared to random-walk MH. The scaling limit literature indicates that the optimal acceptance probability is approximately 0.57 (Dunson and Johndrow, 2020).

For the implementation, the `build_kernel()` in `langevin_chain.py` is structured similarly as for the random-walk MH. One thing to note is the `jax.value_and_grad()`, which computes both the value and the gradient of the log-density in a single pass.

## 5.2. Results

### 5.2.1. Gaussian distribution

Figure 5 shows the tuning experiment for the Gaussian distribution.  $\sigma = 0.5$  is closest to the optimal acceptance probability, and is therefore preferred.

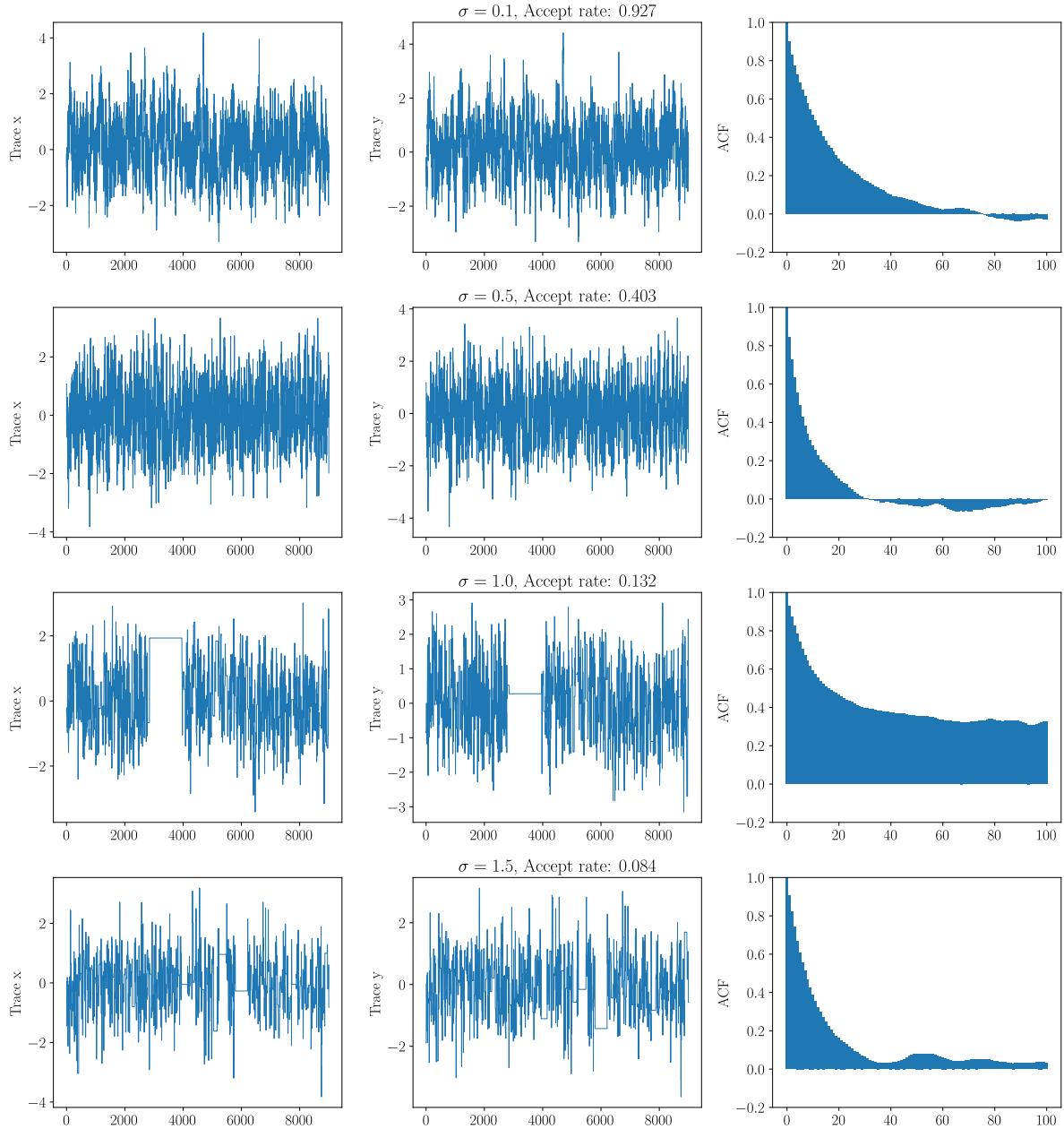


Figure 5: Tuning experiment of Langevin MH for Gaussian distribution

### 5.2.2. Multimodal distribution

Figure 6 shows the tuning experiment for the multimodal distribution.  $\sigma = 1.0$  is closest to the optimal acceptance probability, and is therefore preferred.

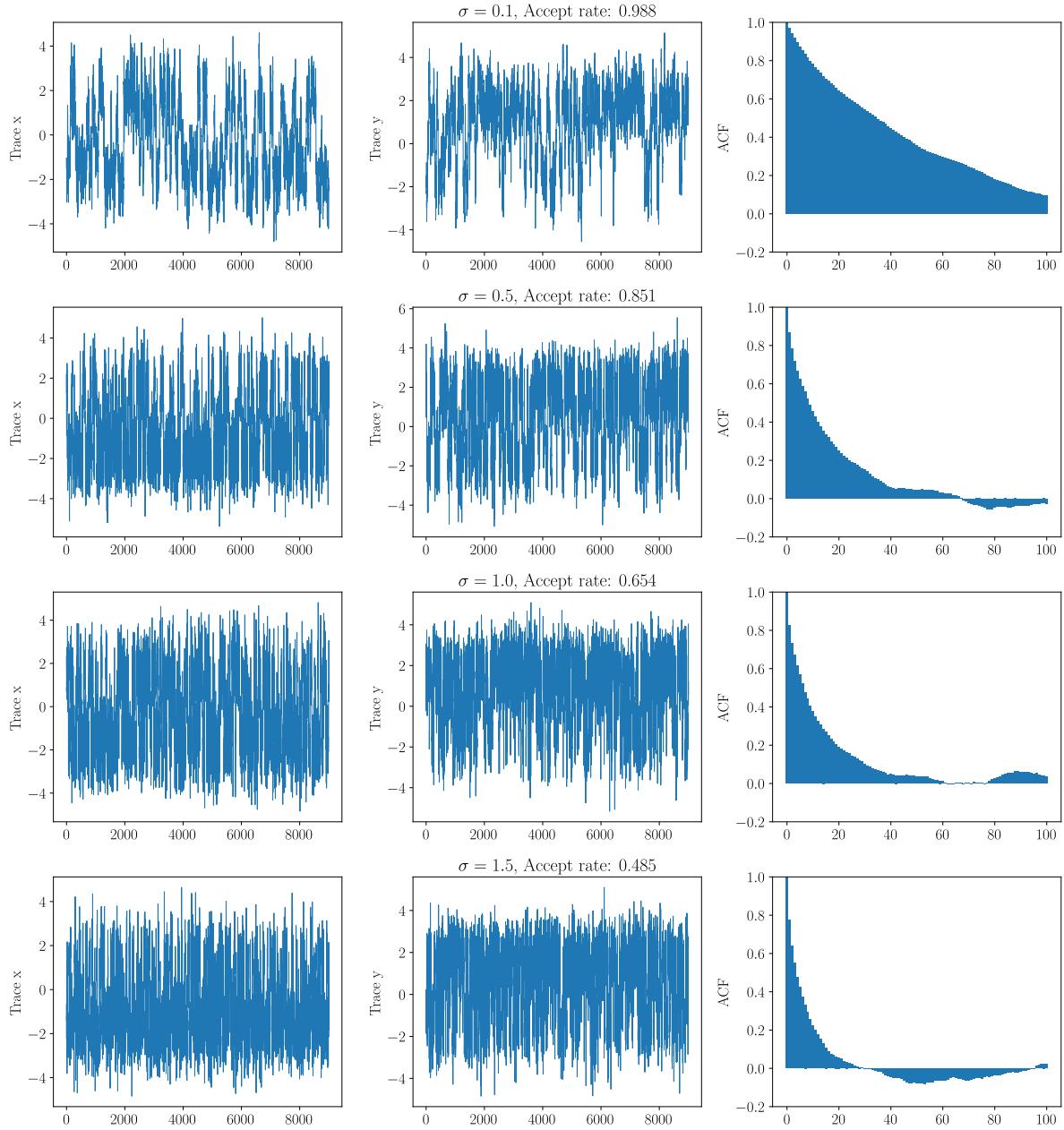


Figure 6: Tuning experiment of Langevin MH for multimodal distribution

### 5.2.3. Volcano distribution

Figure 7 shows the tuning experiment for the volcano distribution.  $\sigma = 1.5$  is closest to the optimal acceptance probability, and is therefore preferred.

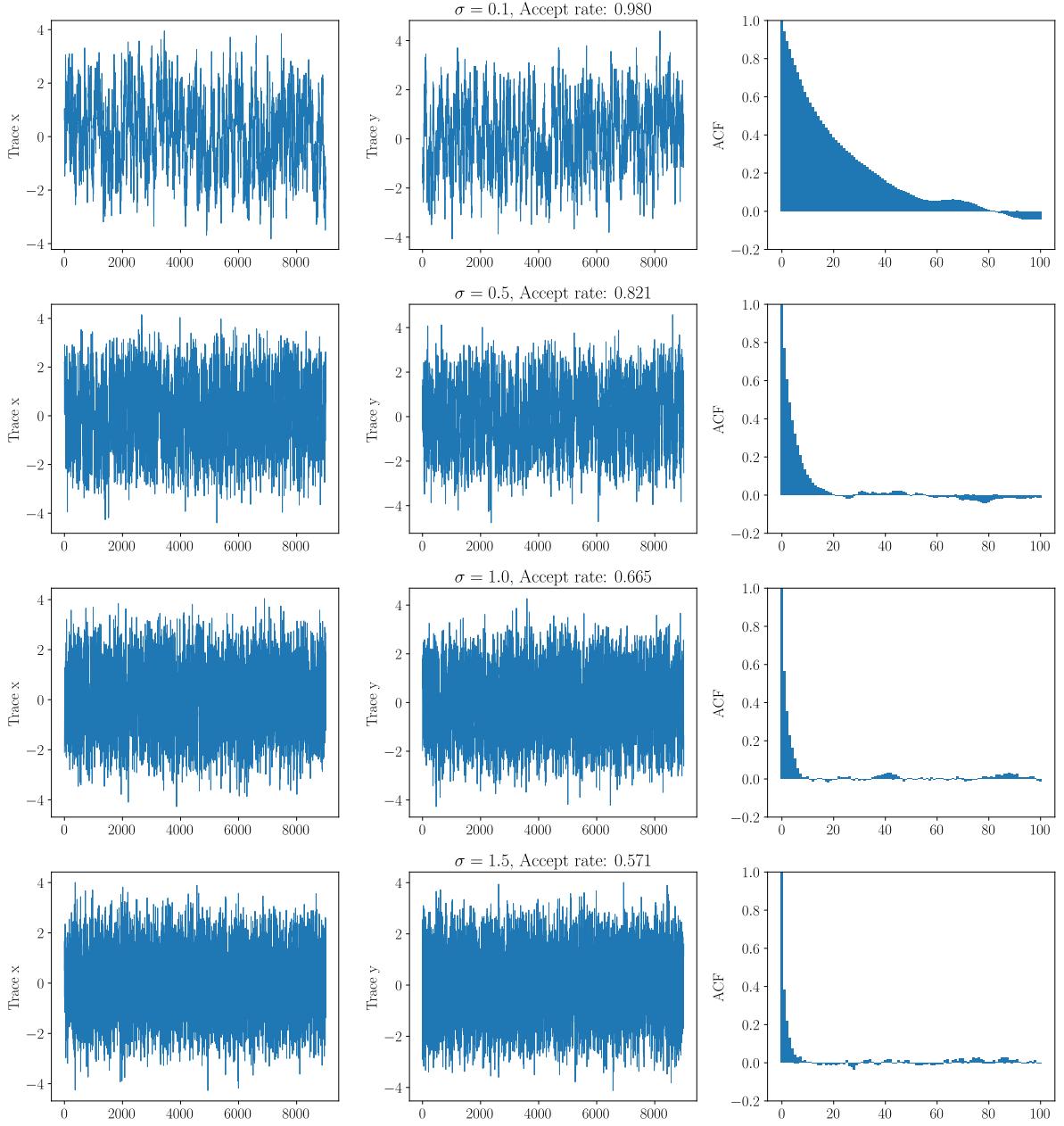


Figure 7: Tuning experiment of Langevin MH for multimodal distribution

## 6. Hamiltonian Metropolis-Hastings

### 6.1. Theory

*Hamiltonian Monte Carlo* (HMC) augments the target with auxiliary *momentum* variables  $\mathbf{p} \in \mathbb{R}^d$  and samples from the joint distribution

$$\pi(\mathbf{x}, \mathbf{p}) \propto \pi(\mathbf{x}) \exp\left(-\frac{1}{2}\mathbf{p}^\top \mathbf{p}\right).$$

This defines a Hamiltonian system with potential energy  $U(\mathbf{x}) = -\log \pi(\mathbf{x})$  and kinetic energy  $K(\mathbf{p}) = \frac{1}{2}\mathbf{p}^\top \mathbf{p}$ , giving total energy (Hamiltonian)

$$H(\mathbf{x}, \mathbf{p}) = U(\mathbf{x}) + K(\mathbf{p}).$$

Each iteration proceeds as follows:

1. **Resample momentum:** Draw  $\mathbf{p} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$  independently of  $\mathbf{x}$ .
2. **Simulate dynamics:** Integrate Hamilton's equations for  $L$  steps using the leapfrog integrator with step size  $\varepsilon$ :

$$\begin{aligned}\mathbf{p}_{t+\varepsilon/2} &= \mathbf{p}_t + \frac{\varepsilon}{2} \nabla \log \pi(\mathbf{x}_t) \\ \mathbf{x}_{t+\varepsilon} &= \mathbf{x}_t + \varepsilon \mathbf{p}_{t+\varepsilon/2} \\ \mathbf{p}_{t+\varepsilon} &= \mathbf{p}_{t+\varepsilon/2} + \frac{\varepsilon}{2} \nabla \log \pi(\mathbf{x}_{t+\varepsilon})\end{aligned}$$

3. **Accept/reject:** Accept the proposal  $(\mathbf{x}', \mathbf{p}')$  with probability  $\min(1, \exp(-\Delta H))$ , where  $\Delta H = H(\mathbf{x}', \mathbf{p}') - H(\mathbf{x}, \mathbf{p})$ .

The leapfrog integrator is *symplectic* (volume-preserving and time-reversible), which ensures the proposal mechanism is symmetric. In exact arithmetic  $\Delta H = 0$ . However, in practice, small discretization errors require the MH correction. HMC can traverse the state space rapidly by following the geometry of  $\pi$ , achieving low autocorrelation even with high acceptance rates.

As for the other algorithms, the `build_kernel()` in `scripts/hmc_chain.py` returns a kernel function that updates the state and info. Like in the MALA implementation, it also relies AD to compute the gradients. The leapfrog integrator is implemented in the `leapfrog()` function, which is called from inside the kernel.

## 6.2. Gaussian distribution

Figure 8 shows the tuning experiment for the Gaussian distribution. For Hamiltonian MC we prefer a high acceptance rate. Looking at the plots we see that  $\sigma = 0.5$  gives the smallest correlation between consecutive samples. As it still has a very high acceptance rate, this is preferred. Compared to the other algorithms, we also see that HMC can achieve much lower autocorrelation even with a high acceptance rate, which is a key advantage of this method.

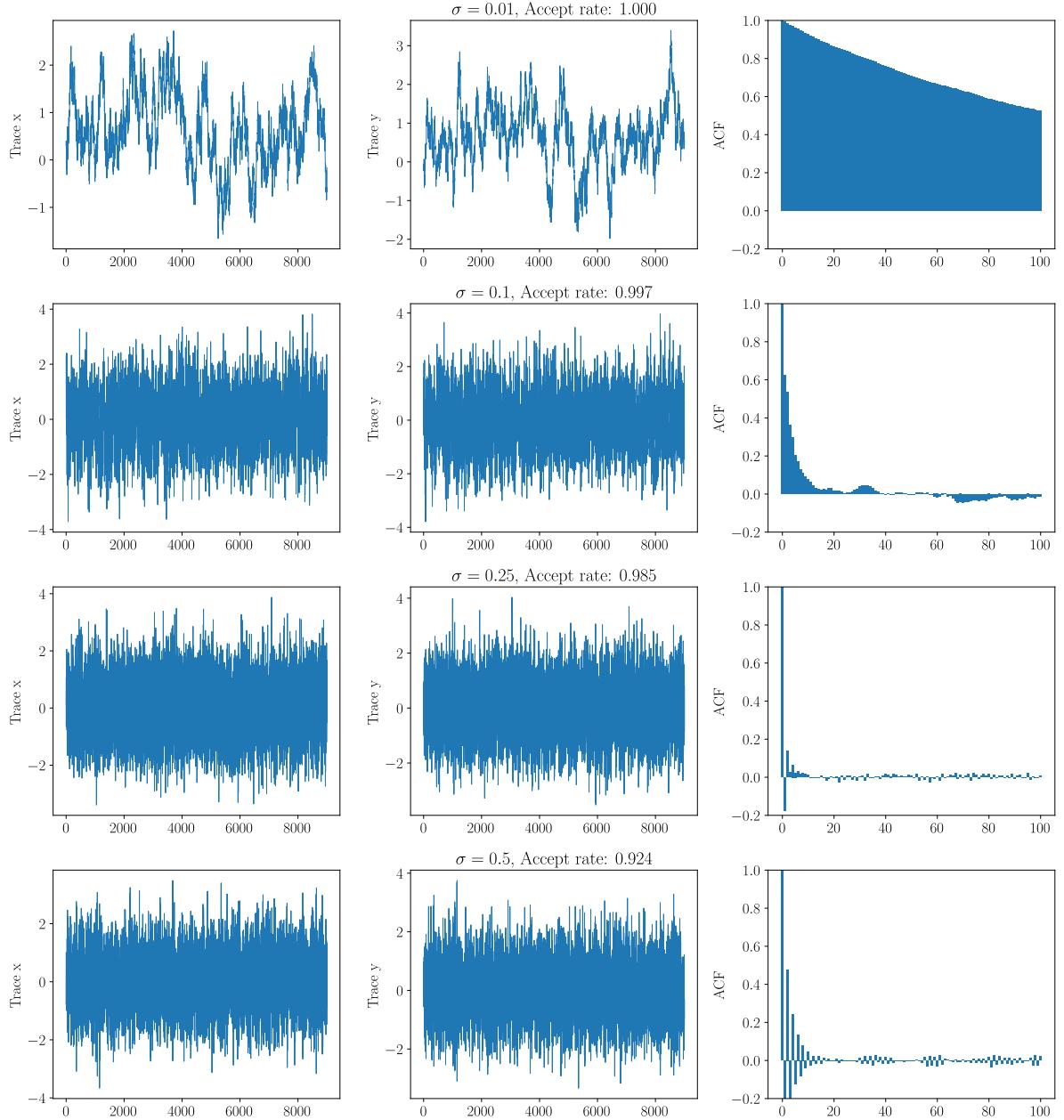


Figure 8: Tuning experiment of HMC for Gaussian distribution

### 6.3. Multimodal distribution

Figure 9 shows the tuning experiment for the multimodal distribution. Again, the  $\sigma = 0.5$  gives fast-decreasing automatic while retaining a high acceptance rate. This is therefore preferred.

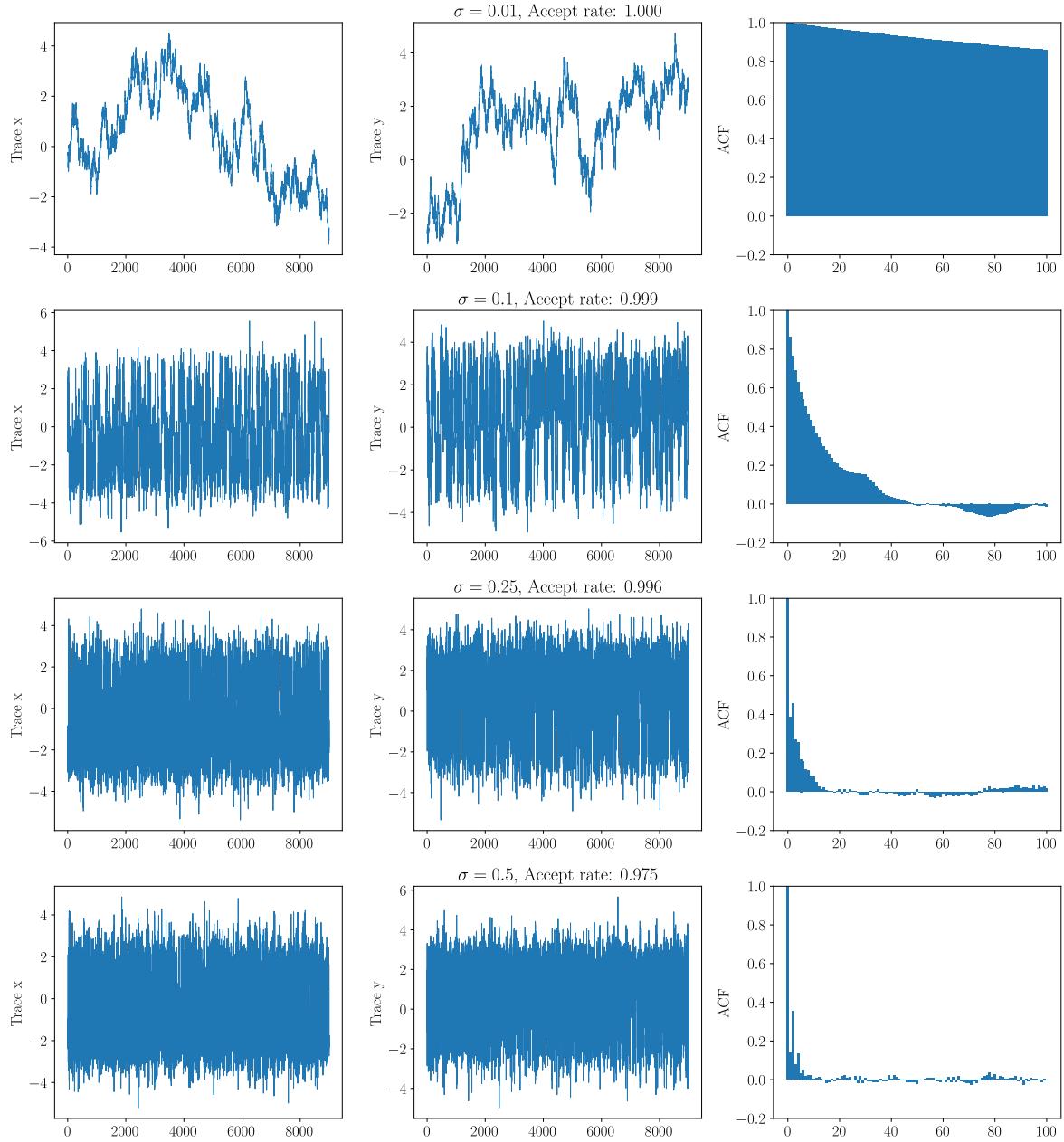


Figure 9: Tuning experiment of HMC for multimodal distribution

#### 6.4. Volcano distribution

Figure 10 shows the tuning experiment for the volcano distribution. For the same reasons as for the previous other distributions, the  $\sigma = 0.5$  is preferred.

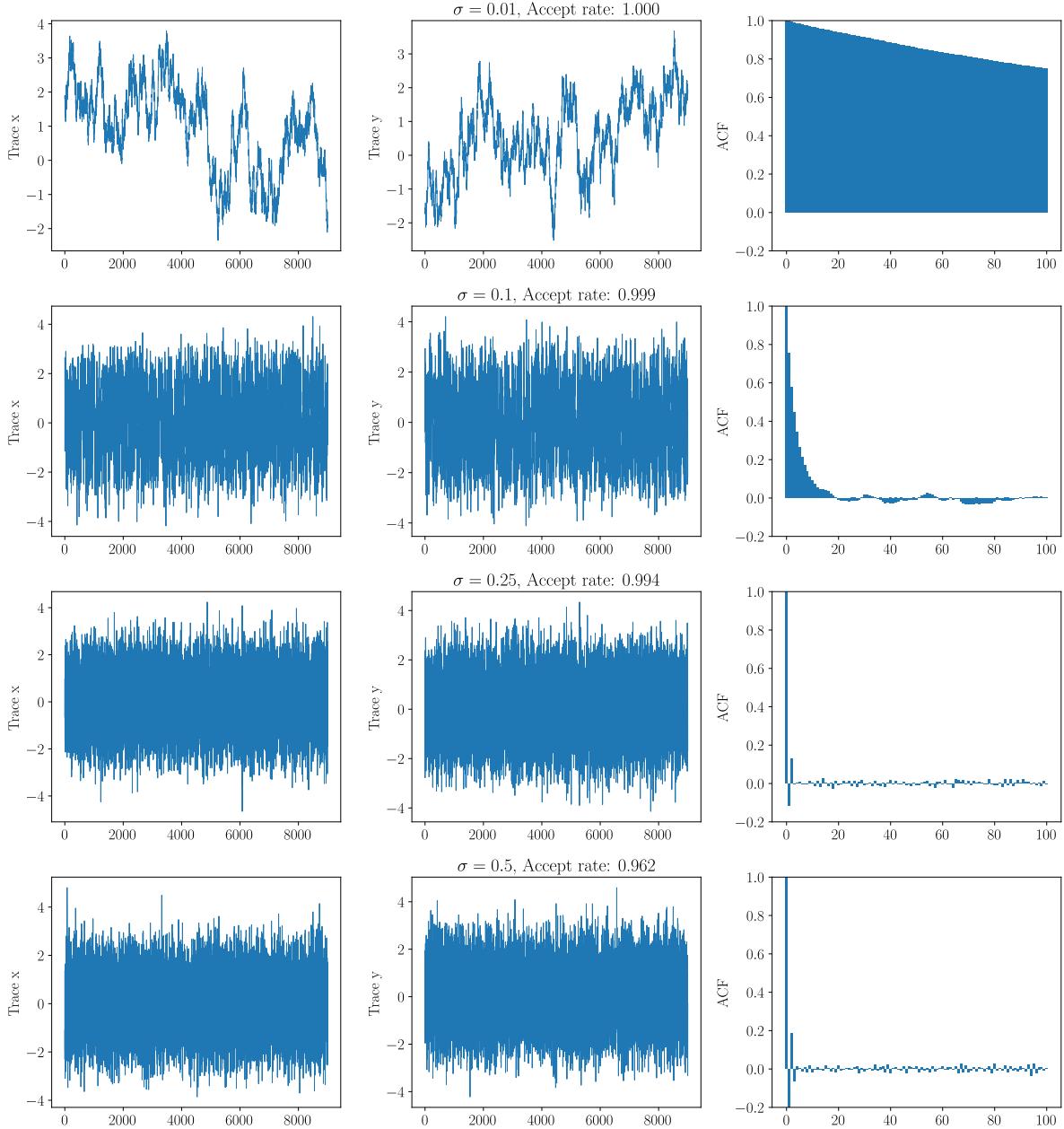


Figure 10: Tuning experiment of HMC for volcano distribution

## 7. Stan

For the last part of the project, we implement a simple model in the probabilistic programming language Stan (Carpenter et al., 2017). This language implements a variant of HMC called the *No-U-Turn Sampler* (NUTS) (Hoffman and Gelman, 2011), which automatically tunes the number of leapfrog steps and step size during sampling. In my personal opinion, developing and using a separate language for MCMC seems like bit of an overkill. However, I am open to giving it a try and see how it works in practice.

We consider an example concerning the number of failures in ten power plants. The data is presented in Table 1. Here  $y_i$  is the number of times that pump  $i$  has failed and  $t_i$  is the operation time for the pump (in 1000s of hours). Pump failures are modelled as

$$y_i | \lambda_i \sim \text{Posson}(\lambda_i t_i), \quad i = 1, \dots, 10.$$

We chose a conjugate prior for  $\lambda_i$

$$\lambda_i | \alpha, \beta \sim \text{Gamma}(\alpha, \beta), \quad i = 1, \dots, 10,$$

where  $\alpha$  and  $\beta$  are given the hyperpriors

$$\alpha \sim \text{Exp}(1.0) \quad \beta \sim \text{Gamma}(0.1, 1.0).$$

PUMP	1	2	3	4	5	6	7	8	9	10
y	5	1	5	14	3	19	1	1	4	22
t	94.3	15.7	62.9	126.0	5.24	31.4	1.05	1.05	2.1	10.5

Table 1: Number of failures and operating times for ten power plants.

In the `stan/pump.stan` file, we write a Stan program for this model. Its defines the data, the parameters and the model, which together specify the likelihood and the priors. Now, we can from R prepare the data and fit the model. In `stan/model.R` we write a script for running four chains for 10000 iterations, with a burn-in of 2000. We also utilize the `bayesplot` package (Gabry et al., 2019) to visualize the results.

Fitting the model, Listing 1 shows summary statistics for the parameters, including the mean, standard deviation, quantiles, effective sample size and Rhat statistic. All parameters exhibits good convergence, with  $\hat{R}$  values equal to 1. Effective sample sizes are large for all parameters (from around 25,000 to 46,000), suggesting low autocorrelation and efficient exploration of the posterior distribution. Monte Carlo standard errors are negligible relative to posterior standard deviations, implying high numerical precision in the estimated posterior summaries.

```
Inference for Stan model: anon_model.
4 chains, each with iter=10000; warmup=2000; thin=1;
post-warmup draws per chain=8000, total post-warmup draws=32000.
```

	mean	se_mean	sd	2.5%	25%	50%	75%	97.5%	n_eff	Rhat
alpha	0.48	0.00	0.18	0.21	0.35	0.45	0.58	0.89	24685	1
beta	0.29	0.00	0.18	0.05	0.16	0.25	0.38	0.74	25076	1
lambda[1]	0.06	0.00	0.02	0.02	0.04	0.05	0.07	0.12	44146	1
lambda[2]	0.09	0.00	0.08	0.01	0.04	0.07	0.13	0.29	46050	1
lambda[3]	0.09	0.00	0.04	0.03	0.06	0.08	0.11	0.17	43638	1
lambda[4]	0.11	0.00	0.03	0.06	0.09	0.11	0.13	0.18	44393	1
lambda[5]	0.63	0.00	0.34	0.15	0.38	0.57	0.81	1.45	45022	1
lambda[6]	0.62	0.00	0.14	0.37	0.52	0.61	0.70	0.91	43654	1
lambda[7]	1.11	0.00	0.93	0.07	0.44	0.87	1.53	3.49	46891	1
lambda[8]	3.39	0.01	1.65	0.99	2.18	3.12	4.28	7.33	39492	1
lambda[9]	9.37	0.01	2.06	5.80	7.90	9.22	10.68	13.79	35355	1
lambda[10]	0.97	0.00	0.30	0.47	0.76	0.94	1.15	1.65	41896	1

Listing 1: Output from fitting the Stan model, showing summary statistics for the parameters.

Figure 11 and Figure 12 shows the traceplots for the parameters. The chains seem to have good mixing and no signs of non-convergence. The posterior distributions for the  $\lambda_i$  are shown in Figure 13, and the 95% credible intervals for  $\lambda_i$  are shown in Figure 14. Overall, the results look reasonable given the data, and the inference seems to have worked well. We observe the

posteriors of  $\lambda_8$  and  $\lambda_9$  being relatively high compared to the others, suggesting that these have a higher failure rate.

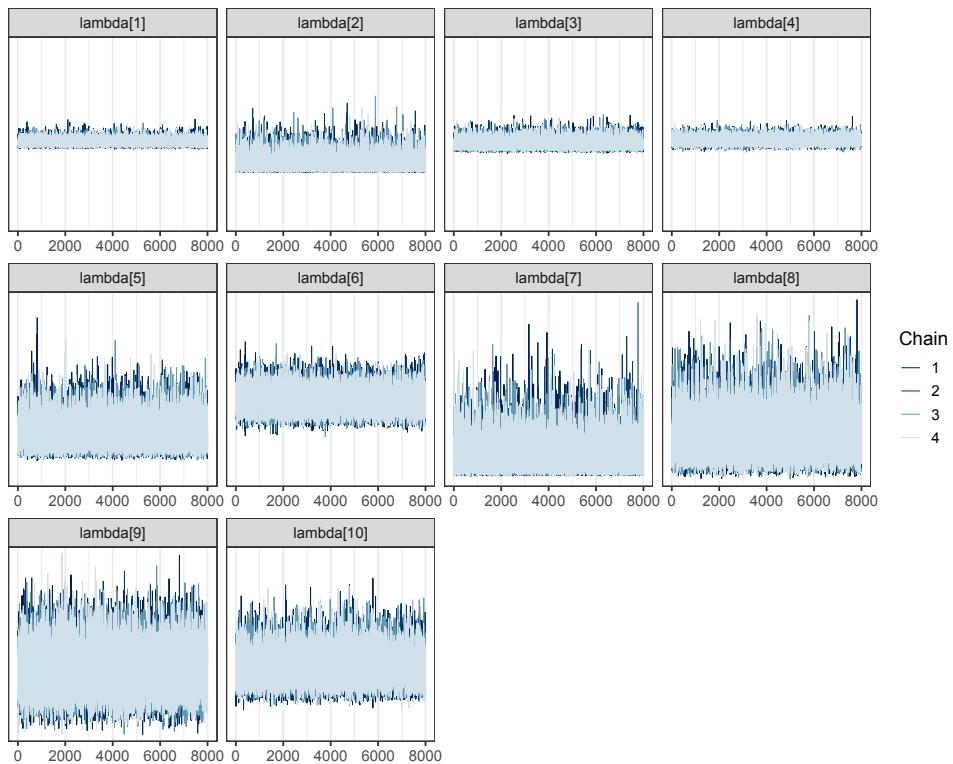


Figure 11: Traceplots for the  $\lambda_i$  parameters.

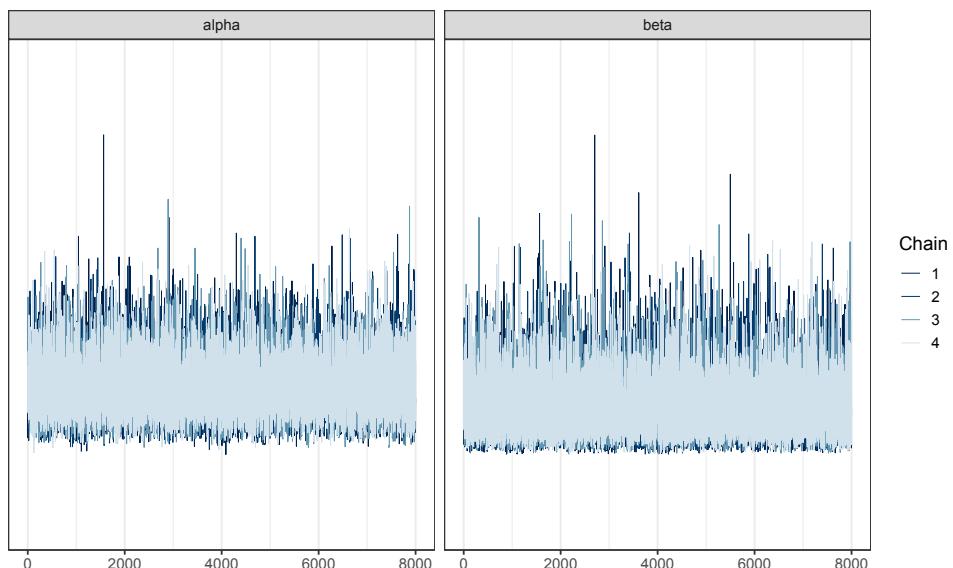


Figure 12: Traceplots for the  $\alpha$  and  $\beta$  hyperparameters.

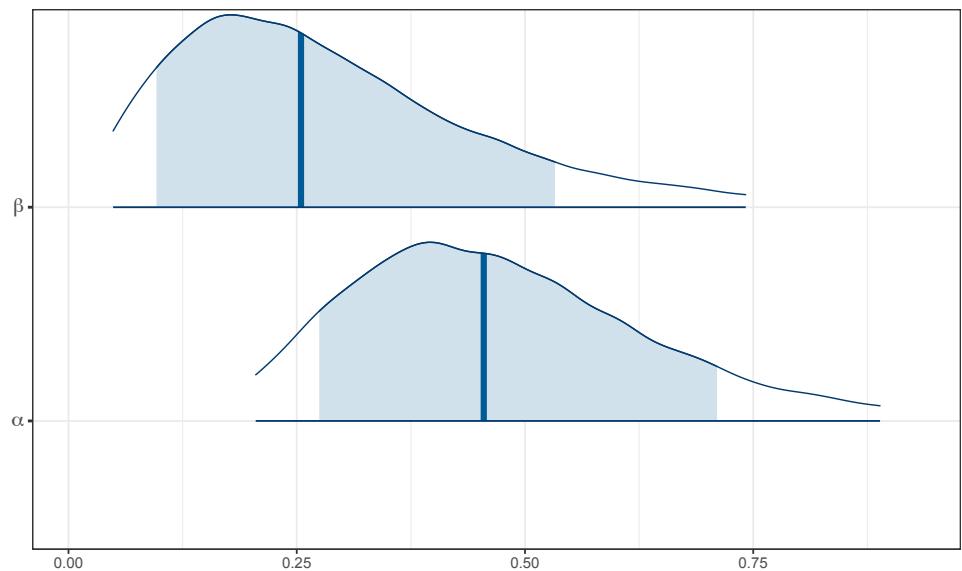


Figure 13: Posterior distributions for the  $\alpha$  and  $\beta$  hyperparameters.

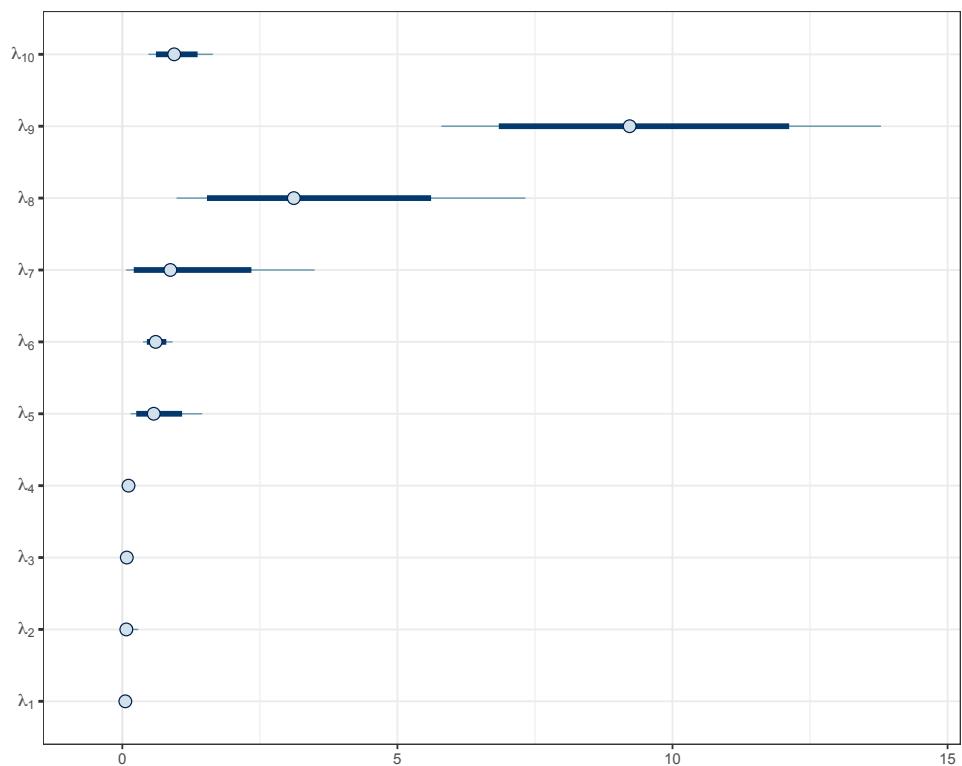


Figure 14: Posterior distributions for the  $\lambda_i$  parameters, with 95% credible intervals.

## Bibliography

- Bradbury, J., Frostig, R., Hawkins, P., Johnson, M.J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., Zhang, Q., 2018. JAX: composable transformations of Python+NumPy programs [WWW Document].. URL <http://github.com/jax-ml/jax>
- Cabezas, A., Corenflos, A., Lao, J., Louf, R., 2024. BlackJAX: Composable Bayesian inference in JAX.
- Carpenter, B., Gelman, A., Hoffman, M.D., Lee, D., Goodrich, B., Betancourt, M., Brubaker, M., Guo, J., Li, P., Riddell, A., 2017. Stan: A probabilistic programming language. *Journal of statistical software* 76.
- Dunson, D.B., Johndrow, J.E., 2020. The Hastings algorithm at fifty. *Biometrika* 107, 1–23.. <https://doi.org/10.1093/biomet/asz066>
- Gabry, J., Simpson, D., Vehtari, A., Betancourt, M., Gelman, A., 2019. Visualization in Bayesian Workflow. *Journal of the Royal Statistical Society Series A: Statistics in Society* 182, 389–402.. <https://doi.org/10.1111/rss.a.12378>
- Hoffman, M.D., Gelman, A., 2011. The No-U-Turn Sampler: Adaptively Setting Path Lengths in Hamiltonian Monte Carlo [WWW Document].. <https://doi.org/10.48550/arXiv.1111.4246>
- Roberts, G.O., Rosenthal, J.S., 2001. Optimal scaling for various Metropolis-Hastings algorithms. *Statistical Science* 16, 351–367.. <https://doi.org/10.1214/ss/1015346320>