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 $\boldsymbol{x} = (x, y)^\top$:

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

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

   where $\boldsymbol{\Sigma}$ 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

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

   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

$$\pi(\boldsymbol{x}) \propto \frac{1}{2\pi} \exp\left(-\frac{1}{2}\boldsymbol{x}^\top \boldsymbol{x}\right)(\boldsymbol{x}^\top \boldsymbol{x} + 0.25) \tag{3}$$

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 101×101 grid cells.

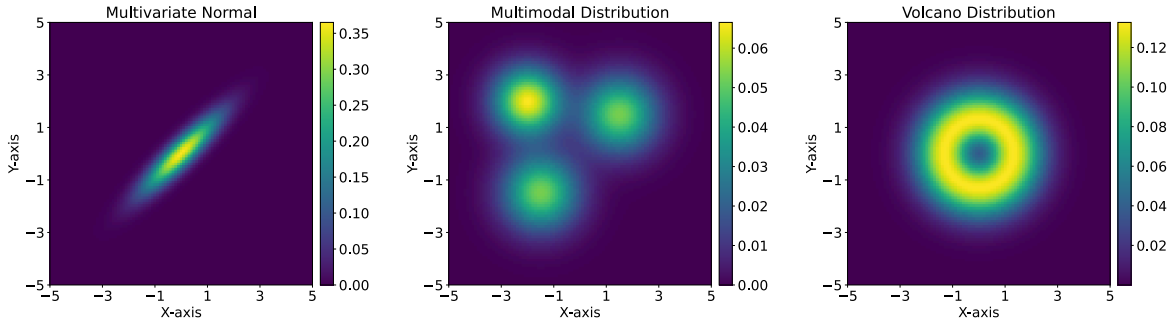

Figure 1: Bivariate densities on a $[-5, 5] \times [-5, 5]$ domain.

The densities are implemented in the `densities.py` file, which contains the following code.

```
import jax
import jax.numpy as jnp
import jax.scipy.stats as stats


def logdensity_mvn(x):
    cov = jnp.array([[1.0, 0.8], [0.8, 1.0]])
    return stats.multivariate_normal.logpdf(x, mean=jnp.zeros(2), cov=cov)
```

```
def logdensity_multimodal(x):
    w = jnp.ones((3,)) / 3.0
    means = jnp.array([[-1.5, -1.5], [1.5, 1.5], [-2.0, 2.0]])

    # All three covariance matrices with 0 on the off-diagonal elements.
    # The first two have 1.0 on the diagonal, the last has 0.8.
    covs = jnp.array([jnp.eye(2), jnp.eye(2), 0.8 * jnp.eye(2)])

    # Compute log density for each component
    log_components = jax.vmap(
        lambda mean, cov: stats.multivariate_normal.logpdf(x, mean=mean, cov=cov),
        in_axes=(0, 0),
    )(means, covs)

    # Log-sum-exp trick: log(sum w_i * exp(log_p_i)) = logsumexp(log(w_i) +
log_p_i)
    log_w = jnp.log(w)
    return jax.scipy.special.logsumexp(log_w + log_components)


def logdensity_volcano(x):
    xtx = jnp.sum(x**2)
    norm_const = 1.0 / (2 * jnp.pi)
    return jnp.log(norm_const) + jnp.log(xtx + 0.25) - 0.5 * xtx
```

2. Theoretical background for Metropolis-Hastings

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

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

$$\alpha\big(\boldsymbol{x}^{(t)}, \boldsymbol{x}'\big) = \min\left(1, \frac{\pi(\boldsymbol{x}')q\big(\boldsymbol{x}^{(t)}|\boldsymbol{x}'\big)}{\pi\big(\boldsymbol{x}^{(t)}\big)q\big(\boldsymbol{x}'|\boldsymbol{x}^{(t)}\big)}\right). \tag{4}$$

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

The acceptance ratio ensures the chain satisfies *detailed balance* with respect to π:

$$\pi(\boldsymbol{x})P(\boldsymbol{x} \to \boldsymbol{x}') = \pi(\boldsymbol{x}')P(\boldsymbol{x}' \to \boldsymbol{x}), \tag{5}$$

which guarantees that π 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 is organized as follows:

```
import jax


def inference_loop(rng_key, kernel, initial_state, num_samples):
    @jax.jit
    def one_step(state, rng_key):
        new_state, info = kernel(rng_key, state)
        return new_state, (new_state, info)

    keys = jax.random.split(rng_key, num_samples)
    _, (states, infos) = jax.lax.scan(one_step, initial_state, keys)

    return states, infos
```

4. Random-walk Metropolis-Hastings

4.1. Theory

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

$$q(\boldsymbol{x}'|\boldsymbol{x}) = \mathcal{N}(\boldsymbol{x}'; \boldsymbol{x}, \sigma^2 \boldsymbol{I}), \tag{6}$$

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

$$\alpha(\boldsymbol{x}, \boldsymbol{x}') = \min\left(1, \frac{\pi(\boldsymbol{x}')}{\pi(\boldsymbol{x})}\right). \tag{7}$$

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

4.2. Implementation

Building the Random-walk MH kernel is done in `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.

```
import jax
from jax import Array
```

```
import jax.numpy as jnp
from typing import NamedTuple, Callable


class RWState(NamedTuple):
    position: Array
    logdensity: Array


class RWInfo(NamedTuple):
    acceptance_rate: Array
    is_accepted: Array
    proposal: RWState


def init(position: Array, logdensity_fn: Callable) -> RWState:
    return RWState(position, logdensity_fn(position))


def build_kernel(logdensity_fn: Callable, step_size: float) -> Callable:
    """Build a Random Walk Rosenbluth-Metropolis-Hastings kernel

    Returns
    -------
    A kernel that takes a rng_key and a Pytree that contains the current state
    of the chain and that returns a new state of the chain along with
    information about the transition.
    """

    def kernel(
        rng_key: Array,
        state: RWState,
    ) -> tuple[RWState, RWInfo]:
        # Generate proposal: x' = x + step_size * N(0, I)
        key_proposal, key_accept = jax.random.split(rng_key)
        proposal = state.position + step_size * jax.random.normal(
            key_proposal, shape=state.position.shape
        )

        # Compute log probability at proposal
        proposal_logdensity = logdensity_fn(proposal)

        # Compute acceptance ratio (symmetric proposal cancels out)
        log_ratio = proposal_logdensity - state.logdensity
        acceptance_prob = jnp.minimum(1.0, jnp.exp(log_ratio))

        # Accept or reject
        u = jax.random.uniform(key_accept)
        accepted = u < acceptance_prob

        # Update state (use lax.cond for cleaner scalar handling)
        new_position = jax.lax.select(accepted, proposal, state.position)
        new_logdensity = jax.lax.select(accepted, proposal_logdensity,
state.logdensity)
        new_state = RWState(new_position, new_logdensity)
```

```
        # Store info
        info = RWInfo(acceptance_prob, accepted, RWState(proposal,
proposal_logdensity))

        return new_state, info

    return kernel
```

4.3. Results

4.3.1. 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.

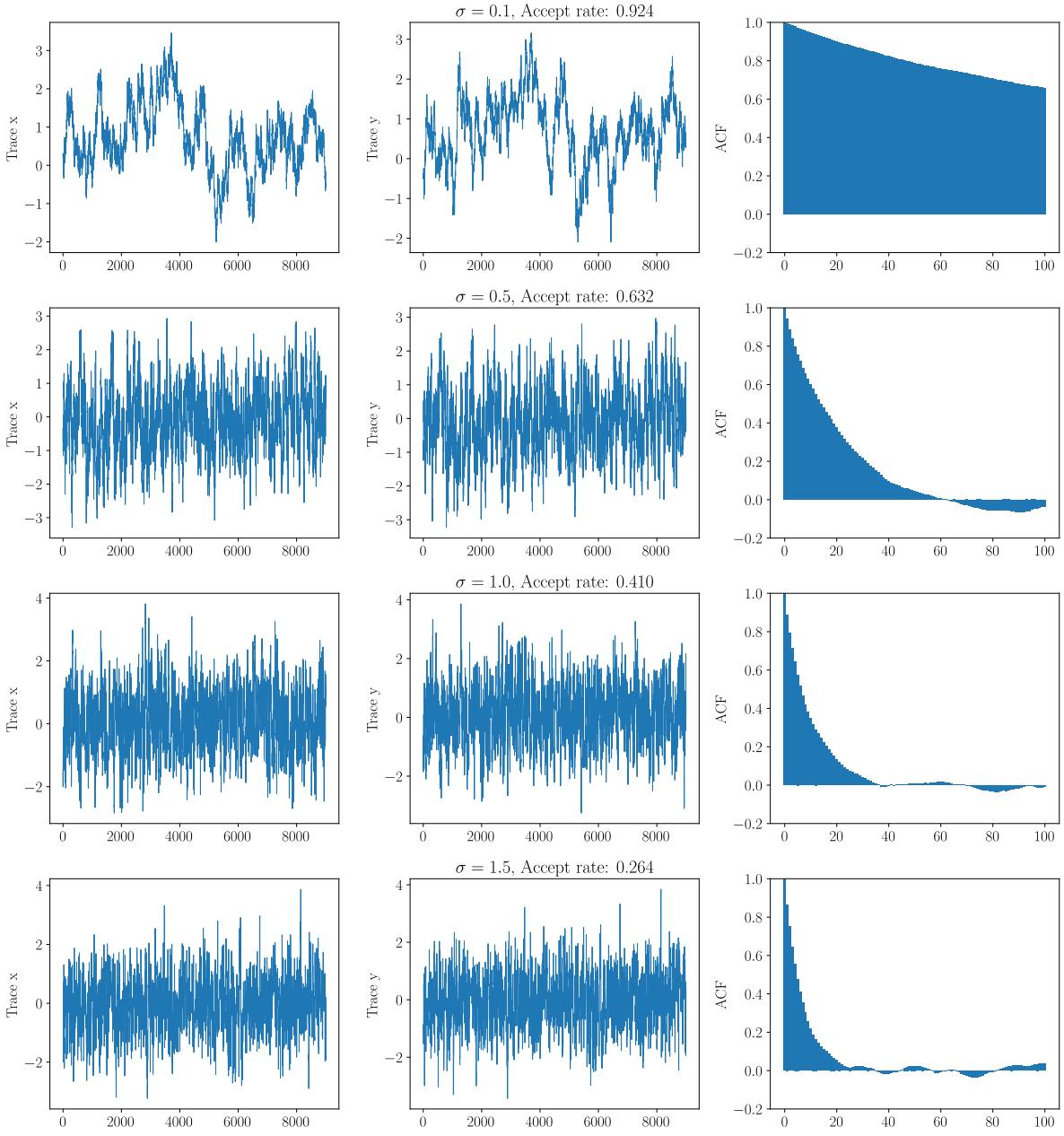

Figure 2: Tuning experiment of Random-walk MH for Gaussian distribution

4.3.2. 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.

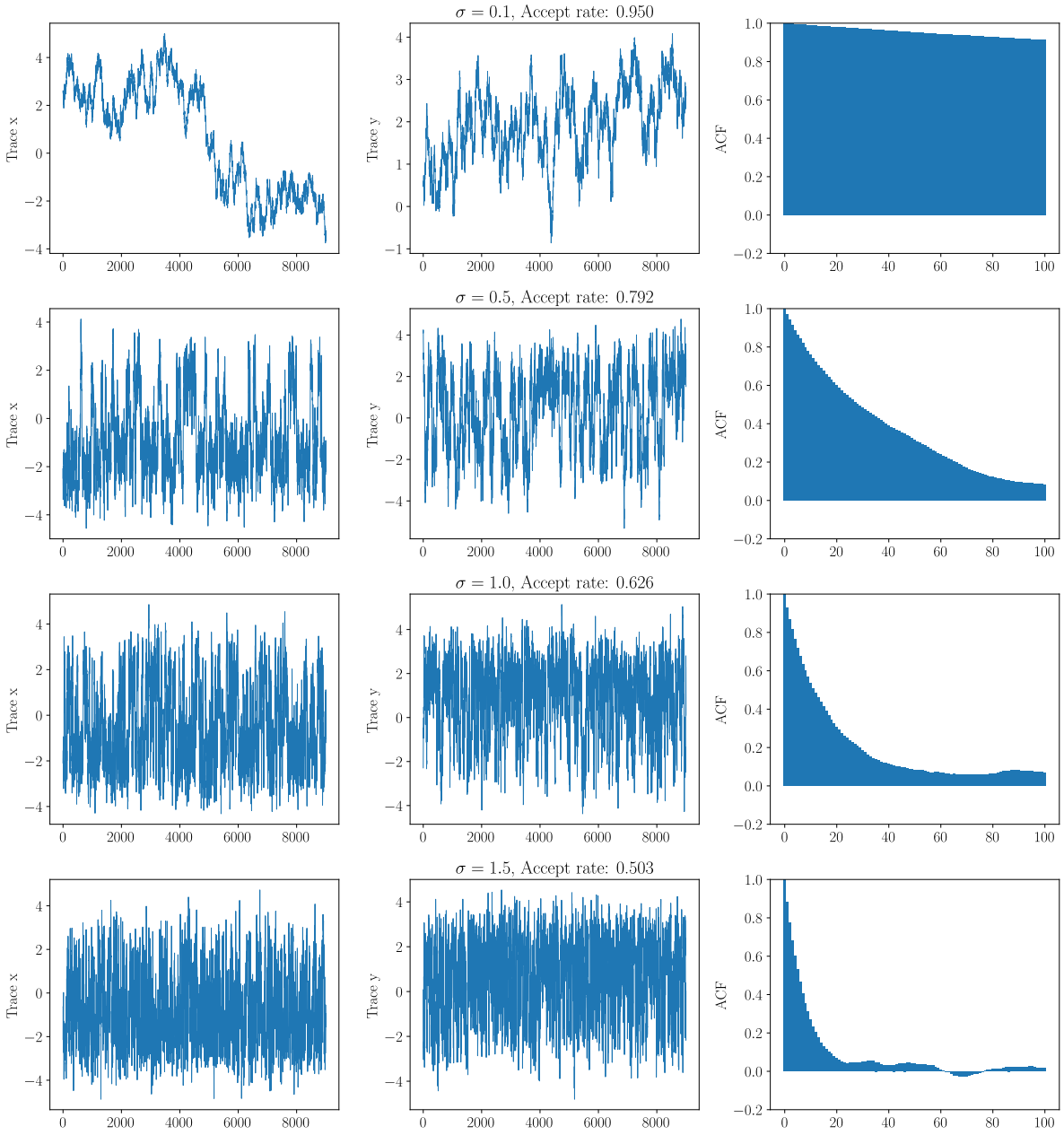

Figure 3: Tuning experiment of Random-walk MH for multimodal distribution

4.3.3. 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.

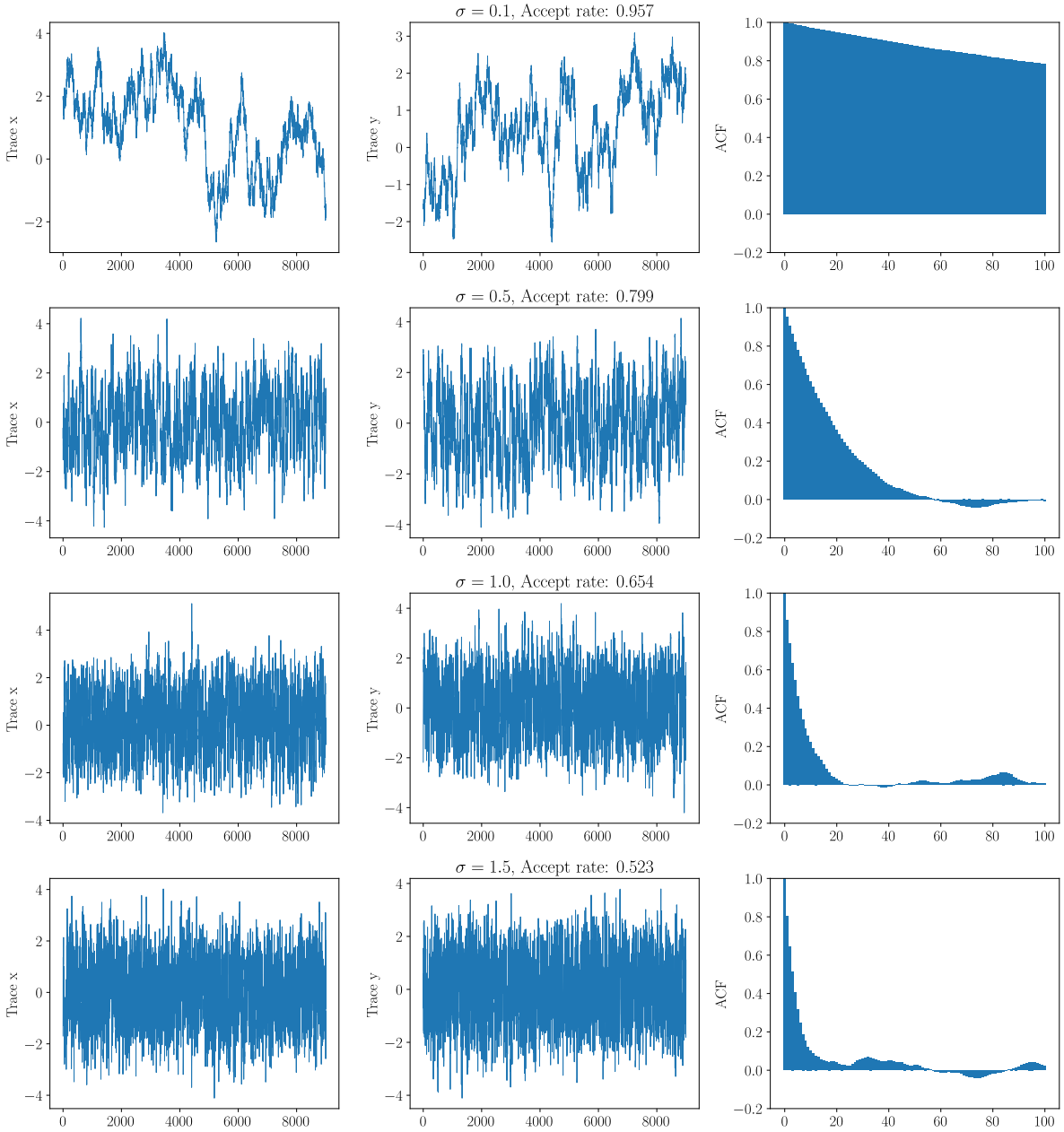

Figure 4: Tuning experiment of Random-walk MH for volcano distribution

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

$$\mathrm{d}\boldsymbol{X}_t = \nabla \log \pi(\boldsymbol{X}_t)\,\mathrm{d}t + \sqrt{2}\,\mathrm{d}\boldsymbol{W}_t, \tag{8}$$

which has π as its stationary distribution. Discretizing with step size ε yields the proposal

$$q(\boldsymbol{x}'|\boldsymbol{x}) = \mathcal{N}(\boldsymbol{x}'; \boldsymbol{x} + \varepsilon \nabla \log \pi(\boldsymbol{x}), 2\varepsilon \boldsymbol{I}). \tag{9}$$

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

$$\alpha(\boldsymbol{x}, \boldsymbol{x}') = \min\left(1, \frac{\pi(\boldsymbol{x}')q(\boldsymbol{x}|\boldsymbol{x}')}{\pi(\boldsymbol{x})q(\boldsymbol{x}'|\boldsymbol{x})}\right). \tag{10}$$

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).

5.2. Implementation

The `build_kernel()` 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.

```
from typing import Callable, NamedTuple

import jax
import jax.numpy as jnp
from jax import Array


class LangevinState(NamedTuple):
    position: Array
    logdensity: Array
    logdensity_grad: Array


class LangevinInfo(NamedTuple):
    acceptance_rate: Array
    is_accepted: Array
    proposal: LangevinState


def init(position: Array, logdensity_fn: Callable) -> LangevinState:
    logdensity, logdensity_grad = jax.value_and_grad(logdensity_fn)(position)
    return LangevinState(position, logdensity, logdensity_grad)


def build_kernel(logdensity_fn: Callable, step_size: float) -> Callable:
    def kernel(
        rng_key: Array,
        state: LangevinState,
    ) -> tuple[LangevinState, LangevinInfo]:
        # Generate proposal: x' = x + step_size * N(0, I)
        key_proposal, key_accept = jax.random.split(rng_key)

        proposal = (
            state.position
            + step_size * state.logdensity_grad
            + jnp.sqrt(2 * step_size)
            * jax.random.normal(key_proposal, shape=state.position.shape)
        )

        proposal_logdensity, proposal_logdensity_grad = jax.value_and_grad(
            logdensity_fn
```

```
        )(proposal)

        # Compute acceptance ratio (symmetric proposal cancels out)
        log_ratio = proposal_logdensity - state.logdensity

        # Compute the proposal densities q(x'|x) and q(x|x')
        def log_proposal_density(from_pos, to_pos, from_grad):
            diff = to_pos - from_pos - step_size * from_grad
            return -0.5 * jnp.sum(diff**2) / (2 * step_size)

        log_q_forward = log_proposal_density(
            state.position, proposal, state.logdensity_grad
        )
        log_q_backward = log_proposal_density(
            proposal, state.position, proposal_logdensity_grad
        )
        log_ratio += log_q_backward - log_q_forward
        acceptance_prob = jnp.minimum(1.0, jnp.exp(log_ratio))

        # Accept or reject
        u = jax.random.uniform(key_accept)
        accepted = u < acceptance_prob

        # Update state (use lax.cond for cleaner scalar handling)
        new_position = jax.lax.select(accepted, proposal, state.position)
        new_logdensity = jax.lax.select(accepted, proposal_logdensity,
state.logdensity)
        new_logdensity_grad = jax.lax.select(
            accepted, proposal_logdensity_grad, state.logdensity_grad
        )
        new_state = LangevinState(new_position, new_logdensity,
new_logdensity_grad)

        # Store info
        info = LangevinInfo(
            acceptance_prob,
            accepted,
            LangevinState(proposal, proposal_logdensity,
proposal_logdensity_grad),
        )

        return new_state, info

    return kernel
```

5.3. 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.

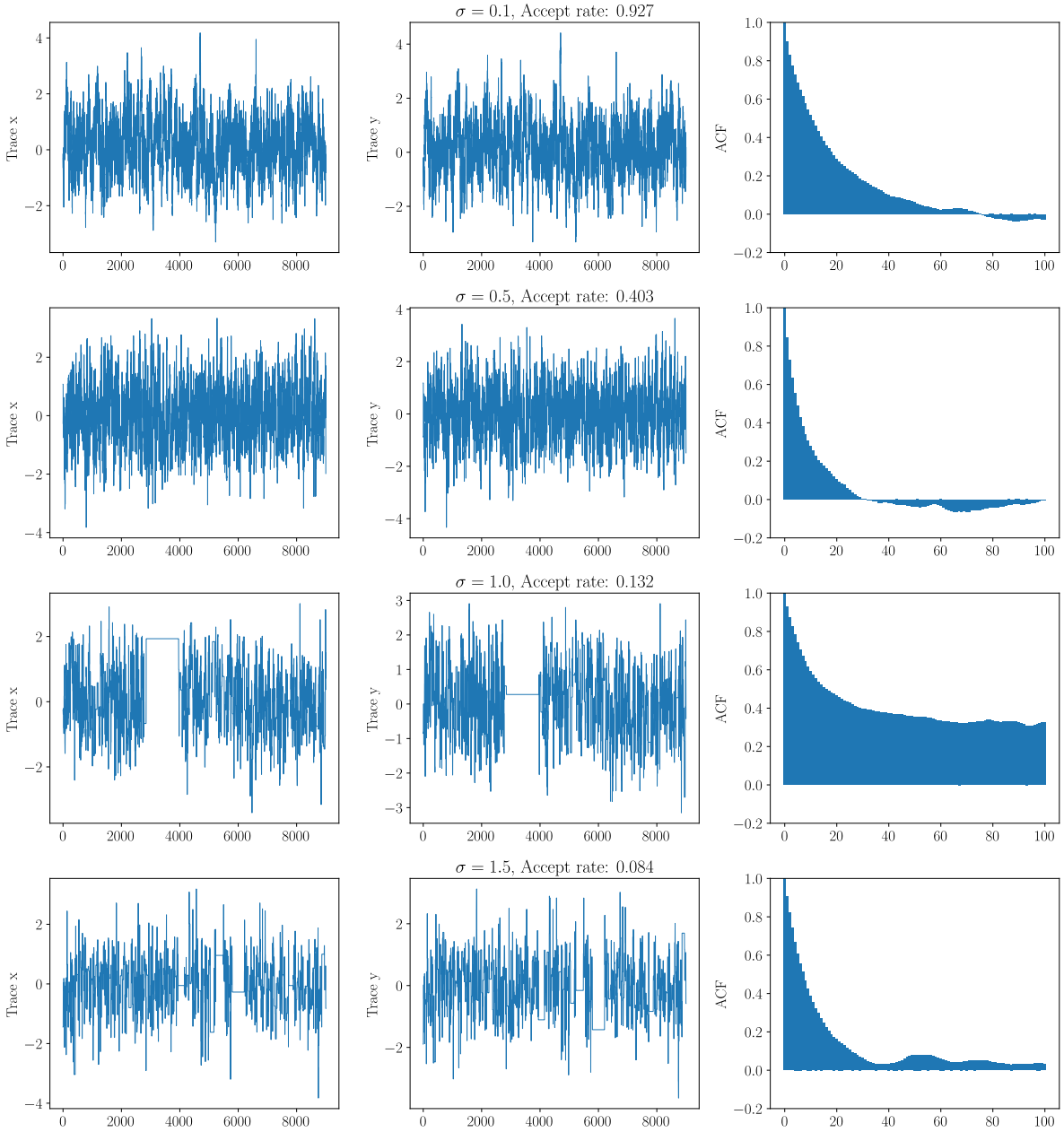

Figure 5: Tuning experiment of Langevin MH for Gaussian distribution

5.4. 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.

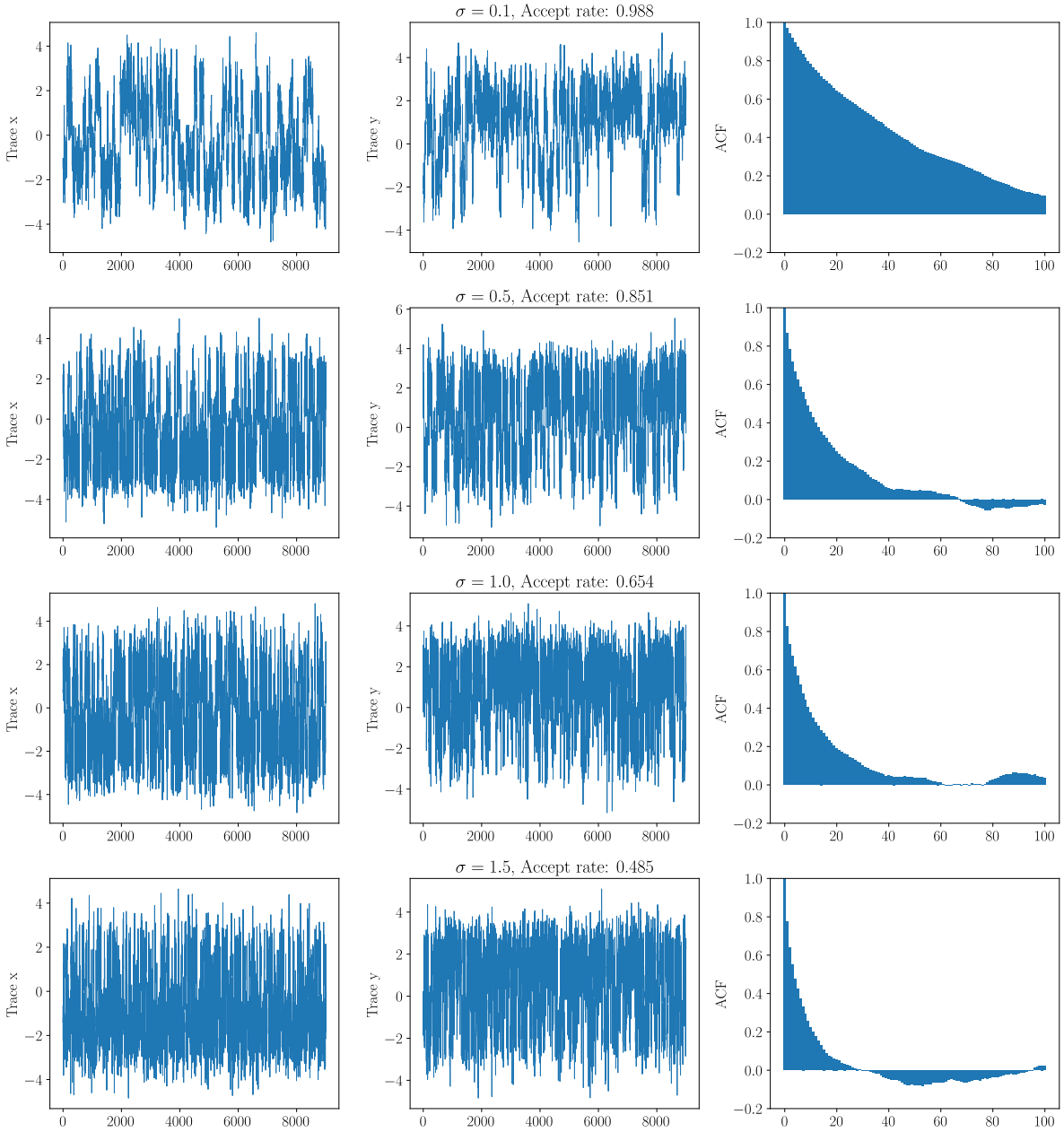

Figure 6: Tuning experiment of Langevin MH for multimodal distribution

5.5. 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.

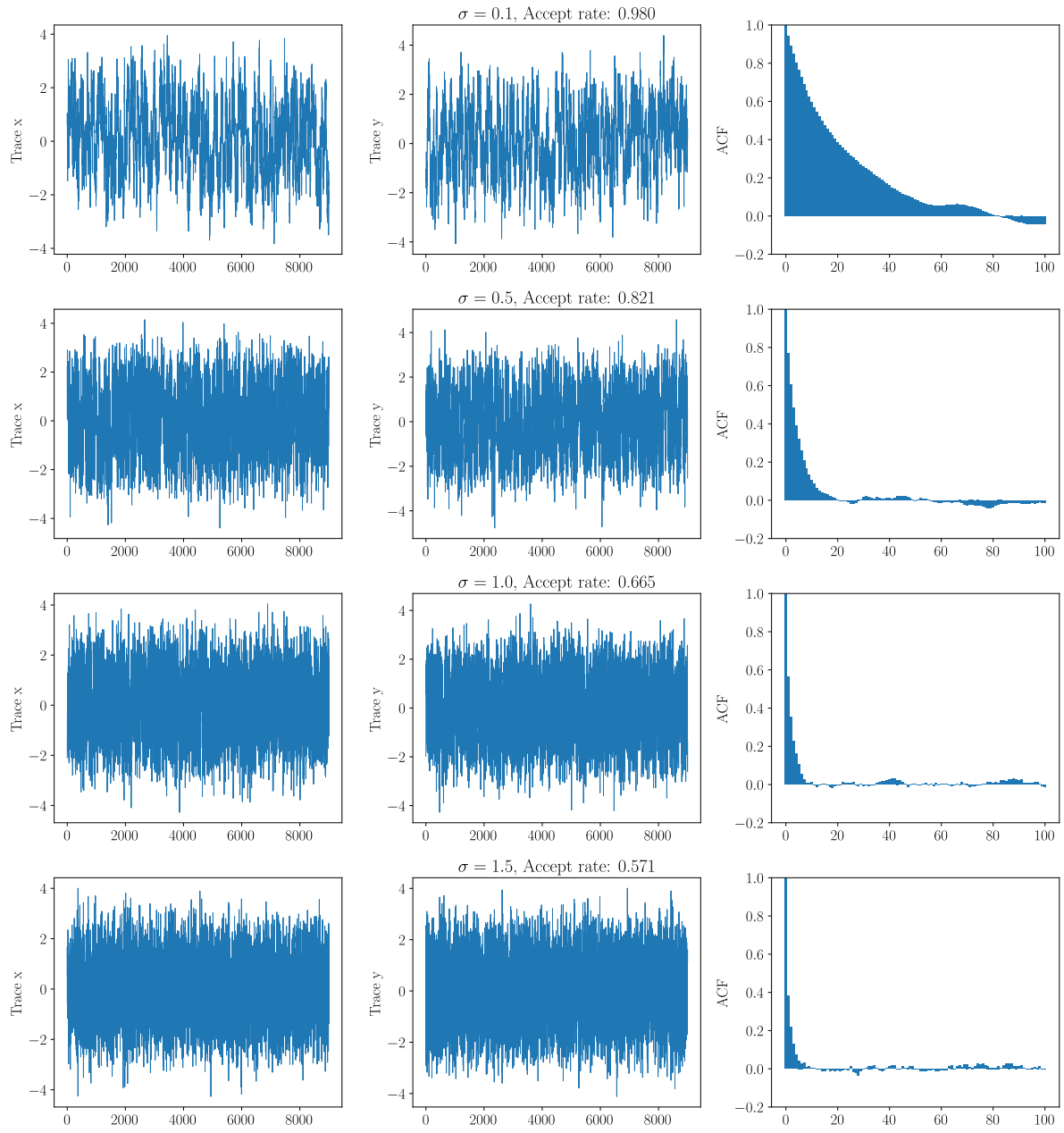

Figure 7: Tuning experiment of Langevin MH for multimodal distribution

6. Hamiltonian Metropolis-Hastings

6.1. Theory

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

$$\pi(\boldsymbol{x}, \boldsymbol{p}) \propto \pi(\boldsymbol{x}) \exp\left(-\frac{1}{2}\boldsymbol{p}^\top \boldsymbol{p}\right). \tag{11}$$

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

$$H(\boldsymbol{x}, \boldsymbol{p}) = U(\boldsymbol{x}) + K(\boldsymbol{p}). \tag{12}$$

Each iteration proceeds as follows:

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

$$\begin{aligned} \boldsymbol{p}_{t+\varepsilon/2} &= \boldsymbol{p}_t + \frac{\varepsilon}{2} \nabla \log \pi(\boldsymbol{x}_t) \\ \boldsymbol{x}_{t+\varepsilon} &= \boldsymbol{x}_t + \varepsilon \boldsymbol{p}_{t+\varepsilon/2} \\ \boldsymbol{p}_{t+\varepsilon} &= \boldsymbol{p}_{t+\varepsilon/2} + \frac{\varepsilon}{2} \nabla \log \pi(\boldsymbol{x}_{t+\varepsilon}) \end{aligned} \tag{13}$$

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

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 π, achieving low autocorrelation even with high acceptance rates.

6.2. Implementation

As for the other algorithms, the `build_kernel()` 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 inside the kernel.

```
from typing import Callable, NamedTuple

import jax
import jax.numpy as jnp
from jax import Array


class HMCState(NamedTuple):
    position: Array
    logdensity: Array
    logdensity_grad: Array


class HMCInfo(NamedTuple):
    acceptance_rate: Array
    is_accepted: Array
    proposal: HMCState


def init(position: Array, logdensity_fn: Callable) -> HMCState:
    logdensity, logdensity_grad = jax.value_and_grad(logdensity_fn)(position)
    return HMCState(position, logdensity, logdensity_grad)


def hamiltonian(logdensity, momentum):
    kinetic = 0.5 * jnp.sum(momentum**2)
```

```
    potential = -logdensity
    return potential + kinetic


def leapfrog(
    position: Array,
    momentum: Array,
    logdensity_fn: Callable,
    step_size: float,
    num_steps: int,
):
    def body_fn(_, state):
        x, p, logp, grad = state

        # Half step momentum
        p = p + 0.5 * step_size * grad

        # Full step position
        x = x + step_size * p

        # Refresh gradient
        logp, grad = jax.value_and_grad(logdensity_fn)(x)

        # Half step momentum
        p = p + 0.5 * step_size * grad

        return x, p, logp, grad

    logp0, grad0 = jax.value_and_grad(logdensity_fn)(position)

    position, momentum, logp, grad = jax.lax.fori_loop(
        0,
        num_steps,
        body_fn,
        (position, momentum, logp0, grad0),
    )

    return position, momentum, logp, grad


def build_kernel(
    logdensity_fn: Callable,
    step_size: float,
    num_steps: int = 10,
) -> Callable:
    def kernel(
        rng_key: Array,
        state: HMCState,
    ) -> tuple[HMCState, HMCInfo]:
        key_momentum, key_accept = jax.random.split(rng_key)

        # Sample momentum
        momentum0 = jax.random.normal(key_momentum, shape=state.position.shape)

        # Current Hamiltonian
```

```
        H = hamiltonian(state.logdensity, momentum0)

        # Propose new state via leapfrog integrator
        q_prop, p_prop, logp_prop, grad_prop = leapfrog(
            state.position,
            momentum0,
            logdensity_fn,
            step_size,
            num_steps,
        )

        # Proposed Hamiltonian
        H_prop = hamiltonian(logp_prop, p_prop)

        # Acceptance probability
        log_accept_ratio = H - H_prop
        acceptance_prob = jnp.minimum(1.0, jnp.exp(log_accept_ratio))

        # Accept or reject
        u = jax.random.uniform(key_accept)
        accepted = u < acceptance_prob

        new_state = HMCState(
            position=jax.lax.select(accepted, q_prop, state.position),
            logdensity=jax.lax.select(accepted, logp_prop, state.logdensity),
            logdensity_grad=jax.lax.select(accepted, grad_prop,
state.logdensity_grad),
        )

        info = HMCInfo(
            acceptance_rate=acceptance_prob,
            is_accepted=accepted,
            proposal=HMCState(q_prop, logp_prop, grad_prop),
        )

        return new_state, info

    return kernel
```

6.3. 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.

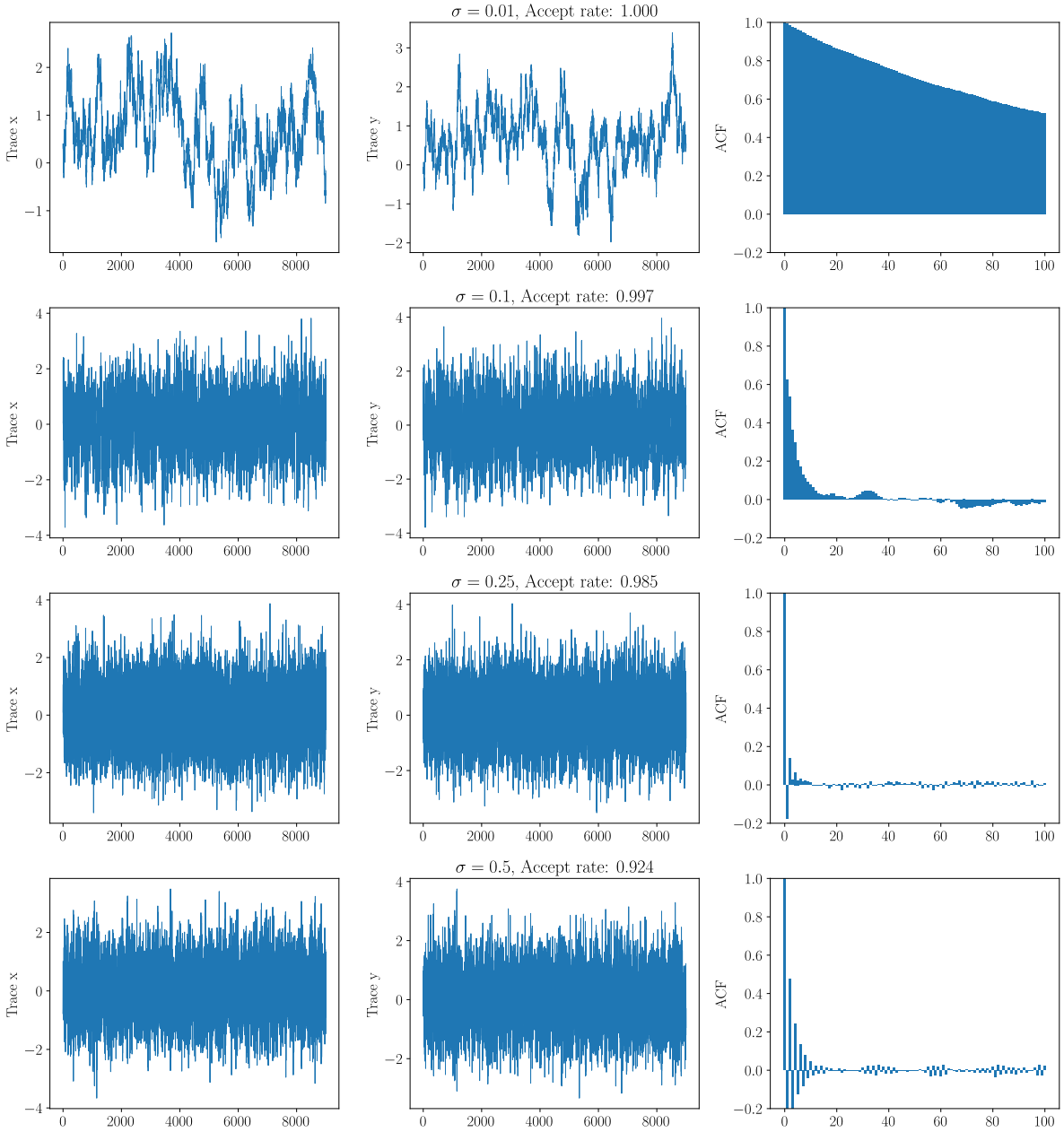

Figure 8: Tuning experiment of HMC for Gaussian distribution

6.4. 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.

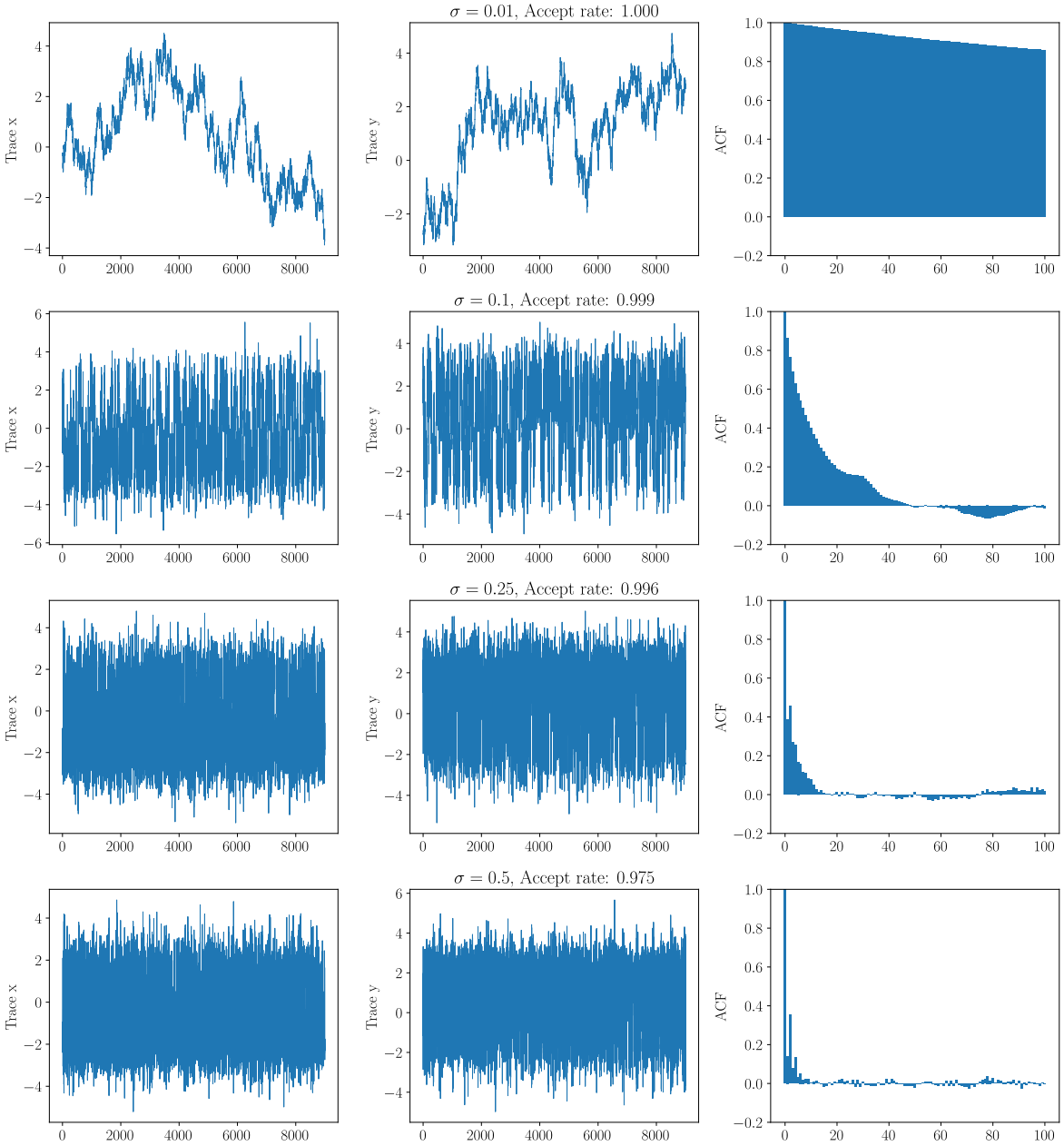

Figure 9: Tuning experiment of HMC for multimodal distribution

6.5. 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.

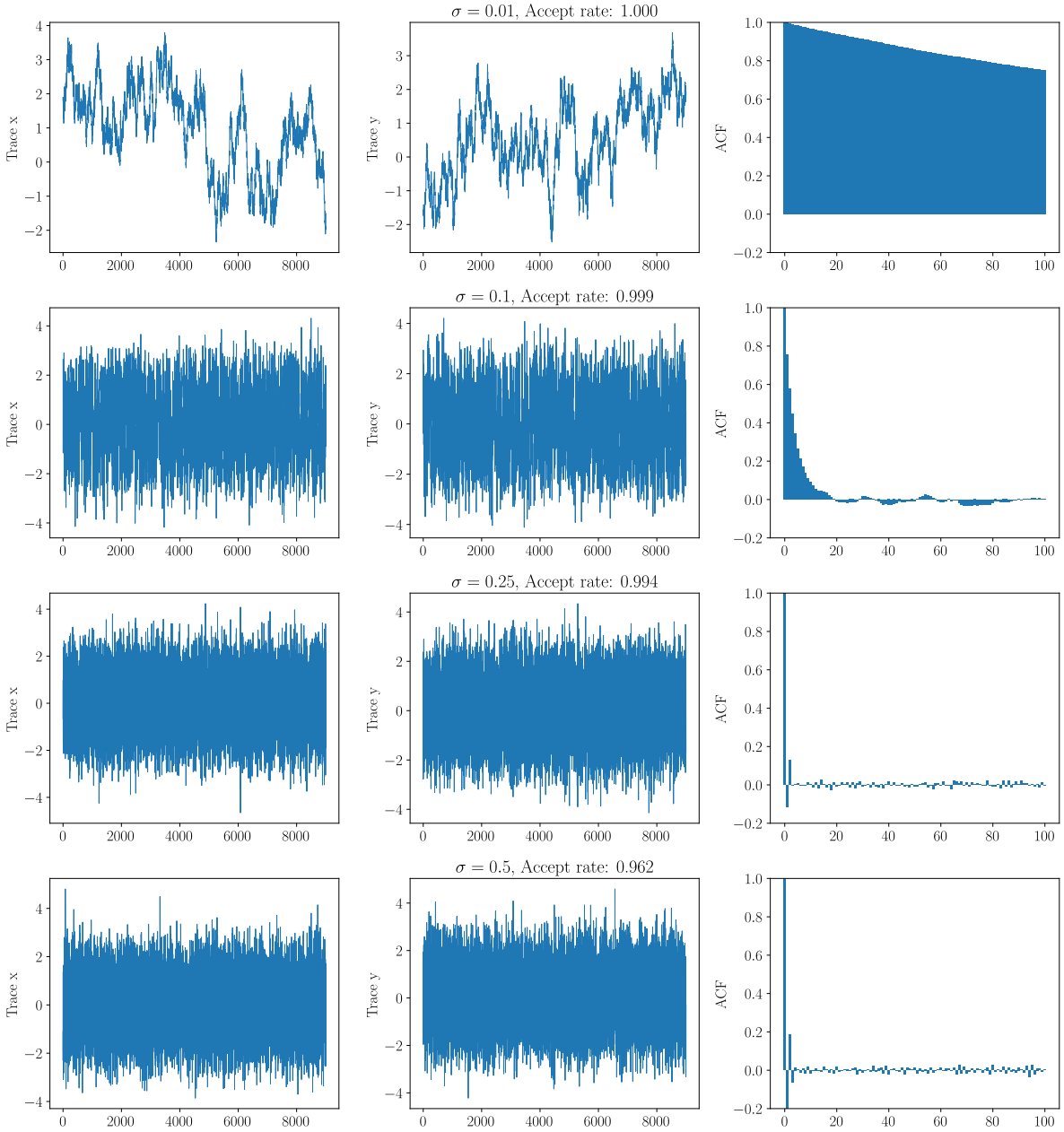

Figure 10: Tuning experiment of HMC for volcano distribution

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

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

We chose a conjugate prior for λ_i

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

where α and β 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 `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. The code is as follows.

```
data {
  int<lower=1> N;           // number of pumps
  int<lower=0> y[N];        // failures
  vector<lower=0>[N] t;     // operating times
}

parameters {
  real<lower=0> alpha;
  real<lower=0> beta;
  vector<lower=0>[N] lambda;
}

model {
  // hyperpriors
  alpha ~ exponential(1.0);
  beta  ~ gamma(0.1, 1.0);

  // prior for individual failure rates
  lambda ~ gamma(alpha, beta);

  // likelihood
  y ~ poisson(lambda .* t);
}
```

Now, we can from R prepare the data and fit the model. We run four chains for 10000 iterations, with a burn-in of 2000. We further use the bayesplot package (Gabry et al., 2019) to visualize the results.

```
library(rstan)
library(bayesplot)
library(ggplot2)

# Allow parallel chains
rstan_options(auto_write = TRUE)
options(mc.cores = parallel::detectCores())
```

```r
y <- c(5, 1, 5, 14, 3, 19, 1, 4, 22, 10)
t <- c(94.3, 15.7, 62.9, 126.0, 5.24, 31.4, 1.05, 1.05, 2.12, 10.5)

stan_data <- list(
  N = length(y),
  y = y,
  t = t
)

fit <- stan(
  file = "pump.stan",  # make sure "pump.stan" is in your working dir
  data = stan_data,
  chains = 4,
  iter = 10000,
  warmup = 2000,
  seed = 123
)

# Print summary of parameters including ESS and R-hat
print(fit, pars = c("alpha", "beta", "lambda"))

# Convert to array for bayesplot
posterior_array <- as.array(fit)

# Traceplots
trace_svg_file <- "traceplots.svg"
svg(trace_svg_file, width = 10, height = 6)
mcmc_trace(posterior_array, pars = c("alpha", "beta"))
dev.off()

# Traceplots for individual lambda_i
trace_lambda_svg <- "trace_lambda.svg"
svg(trace_lambda_svg, width = 10, height = 8)
mcmc_trace(posterior_array, pars = paste0("lambda[", 1:10, "]"))
dev.off()

area_svg <- "posterior_area.svg"
svg(area_svg, width = 10, height = 6)
mcmc_areas(
  posterior_array,
  pars = c("alpha", "beta"),
  prob = 0.8,
  prob_outer = 0.95
)
dev.off()

# Individual lambda_i intervals
interval_svg <- "lambda_intervals.svg"
svg(interval_svg, width = 10, height = 8)
mcmc_intervals(
  posterior_array,
  pars = paste0("lambda[", 1:10, "]"),
  prob = 0.8,
  prob_outer = 0.95
```

```
)
dev.off()
```

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 λ_i are shown in Figure 13, and the 95% credible intervals for λ_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 λ_8 and λ_9 being relatively high compared to the others, suggesting that these have a higher failure rate.

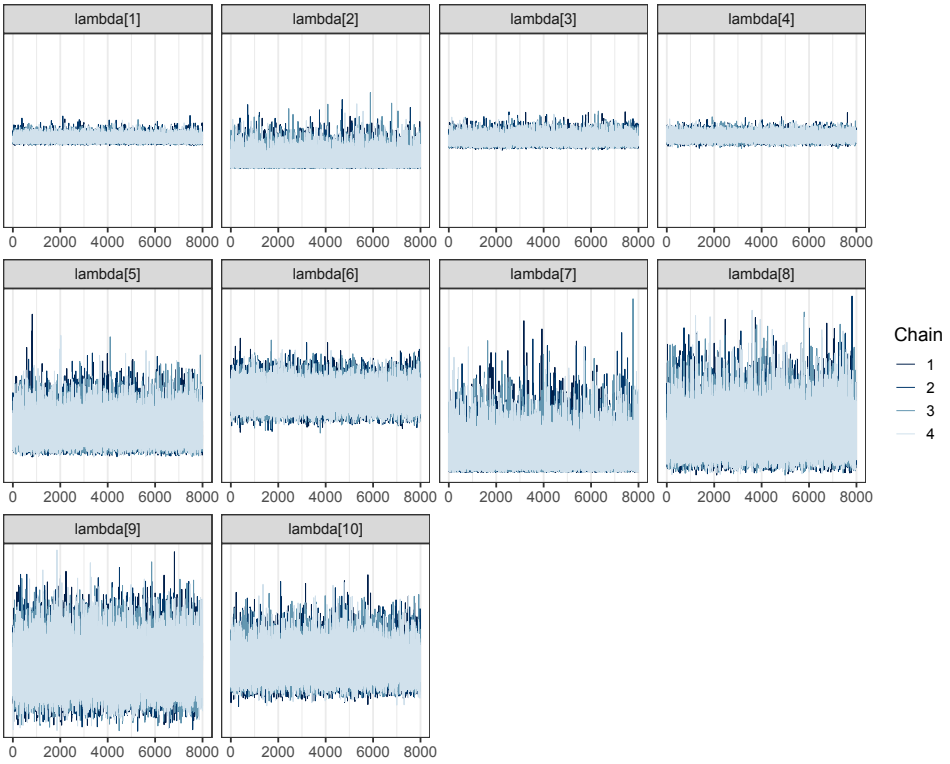

Figure 11: Traceplots for the λ_i parameters.

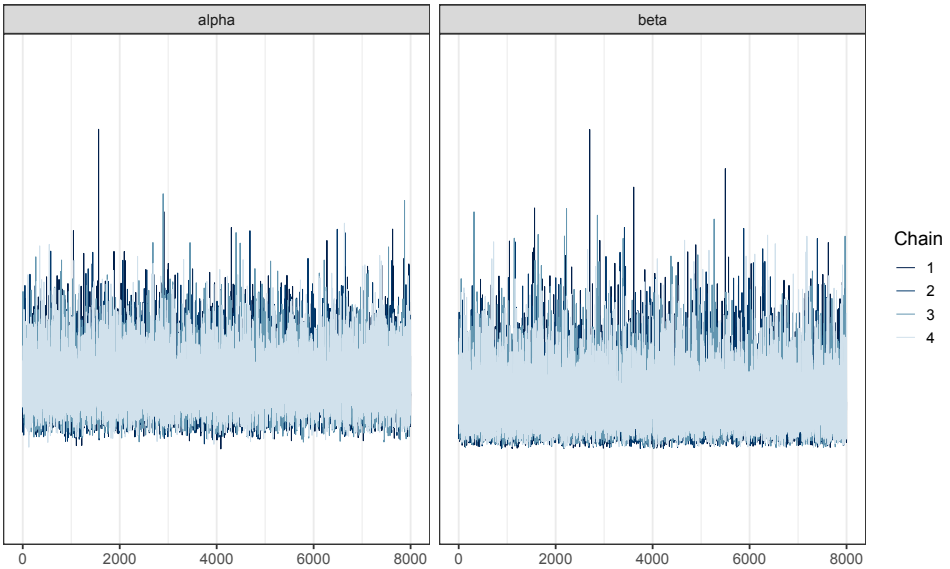

Figure 12: Traceplots for the α and β hyperparameters.

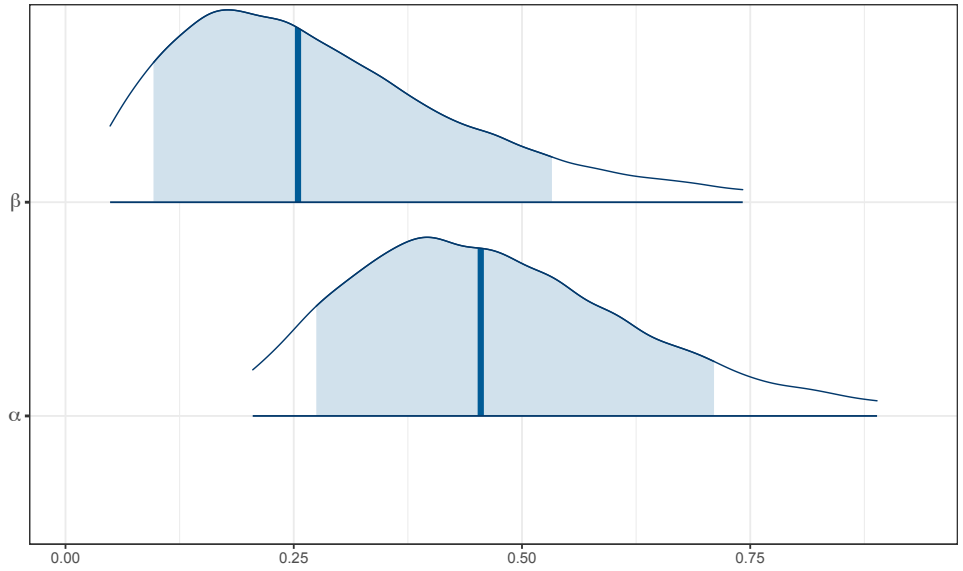

Figure 13: Posterior distributions for the α and β hyperparameters.

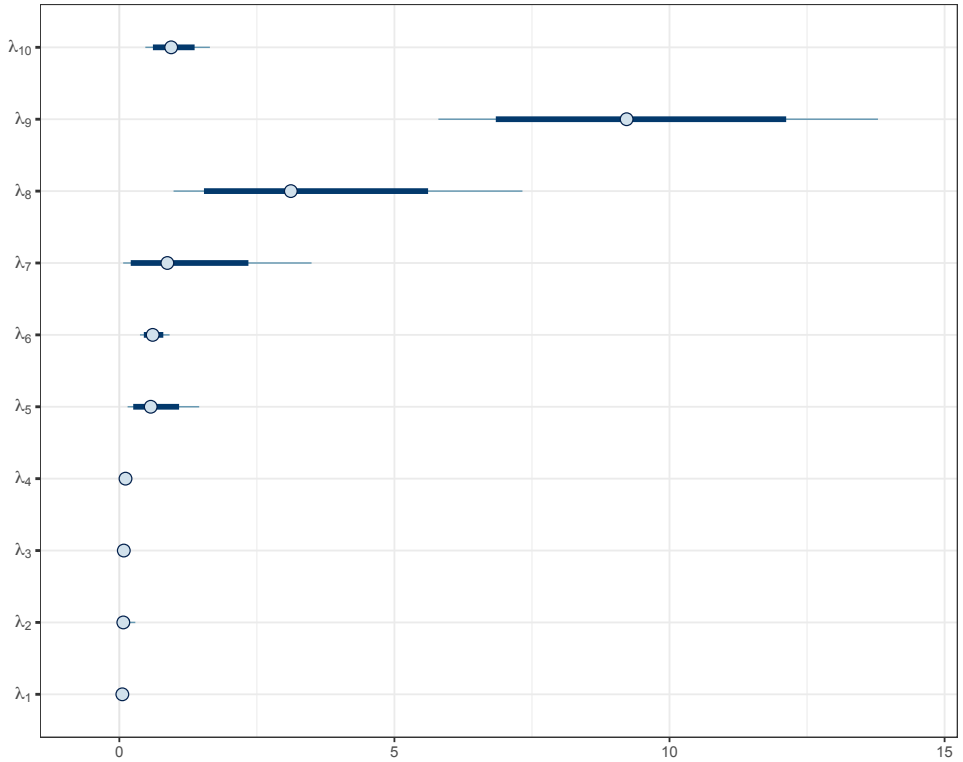

Figure 14: Posterior distributions for the λ_i parameters, with 95% credible intervals.

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/rssa.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