Metropolis-Hastings for bivariate densities

Prosjekt 1 in MA8702. Written by Elling Svee.

January 24, 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 $\boldsymbol{x} = (x, y)^{\mathrm{T}}$:

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

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

   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

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

   with $w_i = 1/3$ for $i = 1, 2, 3$. The means are $\boldsymbol{\mu}_1 = (-1.5, -1.5)^{\mathrm{T}}$, $\boldsymbol{\mu}_2 = (1.5, 1.5)^{\mathrm{T}}$ and $\boldsymbol{\mu}_3 = (-2, 2)^{\mathrm{T}}$, 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

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

Figure 1 visualize 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.

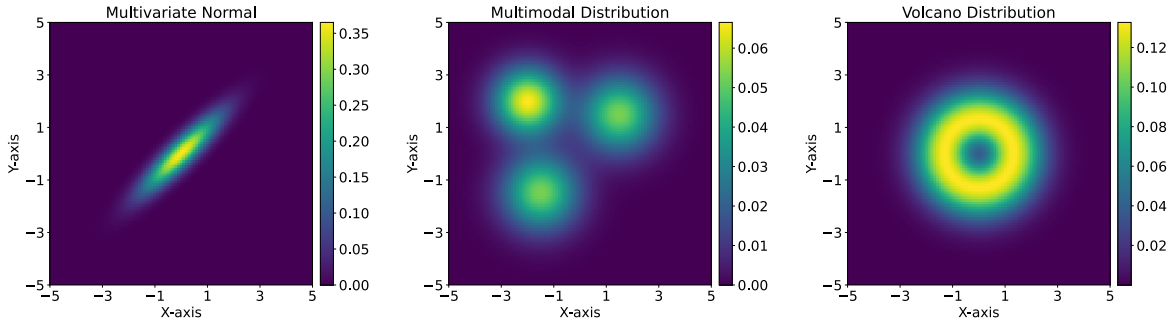

Figure 1: Bivariate densities

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 $q(\boldsymbol{x}'|\boldsymbol{x}^{(t)})$.
2. **Compute** the acceptance probability

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

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). Outside of the main implementation-scripts, we use the following files:

`densities.py` Implementation of the three densities.

```
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
```

`inderence.py` For running the chains.

```python
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
```

`autocorr.py` Computing the autocorrelations.

```python
import jax.numpy as jnp


def autocorr(x, max_lag=100, normalize=True):
    x = jnp.asarray(x)
    x = x - jnp.mean(x)

    n = x.shape[0]

    if max_lag is None:
        max_lag = n - 1
    max_lag = jnp.minimum(max_lag, n - 1)

    # FFT with zero-padding
    f = jnp.fft.fft(x, n=2 * n)
    ac = jnp.fft.ifft(f * jnp.conj(f)).real

    # keep only requested lags
    ac = ac[: max_lag + 1]

    if normalize:
        ac = ac / ac[0]

    return ac
```

`tuning_experiment.py` Running the algorithms for different σ-values, and plotting results.

```python
from os import PathLike
from typing import Callable

import jax
import jax.numpy as jnp
import matplotlib.pyplot as plt

from scripts.autocorr import autocorr
from scripts.inference import inference_loop


def run_tuning_experiment(
```

```
    init_fn: Callable,
    build_kernel_fn: Callable,
    logdensity_fn: Callable,
    filename: PathLike[str],
    sigma_values=[0.1, 0.5, 1.0, 1.5],
) -> None:
    """Run RWMH tuning experiment with different proposal stddev (sigma) values
    and plot the results.
    """
    # Run with different sigma values
    num_steps = 10000
    burnin = 1000

    key = jax.random.key(42)
    initial_pos = jnp.array([0.0, 0.0])

    fig, axes = plt.subplots(len(sigma_values), 3, figsize=(15, 4 *
len(sigma_values)))
    fig_scatter, axes_scatter = plt.subplots(
        1, len(sigma_values), figsize=(5 * len(sigma_values), 5)
    )

    initial_state = init_fn(initial_pos, logdensity_fn)
    for i, sigma in enumerate(sigma_values):
        kernel = build_kernel_fn(logdensity_fn, sigma)

        # Run chain
        key, subkey = jax.random.split(key)
        samples, infos = inference_loop(subkey, kernel, initial_state, num_steps)

        # Remove burnin
        positions = samples.position[burnin:]
        accept = infos.is_accepted[burnin:]

        # Acceptance rate
        acc_rate = float(jnp.mean(accept))

        row_title = rf"$\sigma = {sigma}$, Accept rate: {acc_rate:.3f}"

        # Trace plots
        axes[i, 0].plot(positions[:, 0], linewidth=0.5)
        axes[i, 0].set_ylabel("Trace x")

        axes[i, 1].plot(positions[:, 1], linewidth=0.5)
        axes[i, 1].set_ylabel("Trace y")
        axes[i, 1].set_title(row_title)

        # Autocorrelation
        acf = autocorr(positions[:, 0])
        axes[i, 2].bar(range(len(acf)), acf, width=1.0)
        axes[i, 2].set_ylabel("ACF")
        axes[i, 2].set_ylim([-0.2, 1.0])

        # 2D Scatter plot
        axes_scatter[i].scatter(positions[:, 0], positions[:, 1], alpha=0.5)
```

```
        axes_scatter[i].set_xlabel("x")
        axes_scatter[i].set_ylabel("y")
        axes_scatter[i].set_title(row_title)
        axes_scatter[i].axis("equal")
        axes_scatter[i].grid()

    # Save the two figure separately
    fig.tight_layout()
    fig.savefig(f"{filename}.svg")
    plt.close(fig)

    fig_scatter.tight_layout()
    fig_scatter.savefig(f"{filename}_scatter.svg")
    plt.close(fig_scatter)
```

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

`random_walk.py` Building the Random-walk MH kernel

```
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
```

`rwmh_chain.py` Running the tuning experiment for the Random-walk MH.

```
import os
from pathlib import Path
from scripts.tuning_experiment import run_tuning_experiment
```

```
from scripts.random_walk import init, build_kernel
from scripts.densities import logdensity_multimodal, logdensity_mvn,
logdensity_volcano


OUTPUT_DIR = Path("output/rwmh/")
os.makedirs(OUTPUT_DIR, exist_ok=True)


print("Running tuning experiment for Random Walk Metropolis-Hastings...")
print("- Multivariate Normal distribution...")
run_tuning_experiment(init, build_kernel, logdensity_mvn, OUTPUT_DIR /
"tuning_mvn")
print("- Multimodal distribution...")
run_tuning_experiment(
    init, build_kernel, logdensity_multimodal, OUTPUT_DIR / "tuning_multimodal"
)
print("- Volcano distribution...")
run_tuning_experiment(
    init, build_kernel, logdensity_volcano, OUTPUT_DIR / "tuning_volcano"
)
```

4.3. 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.

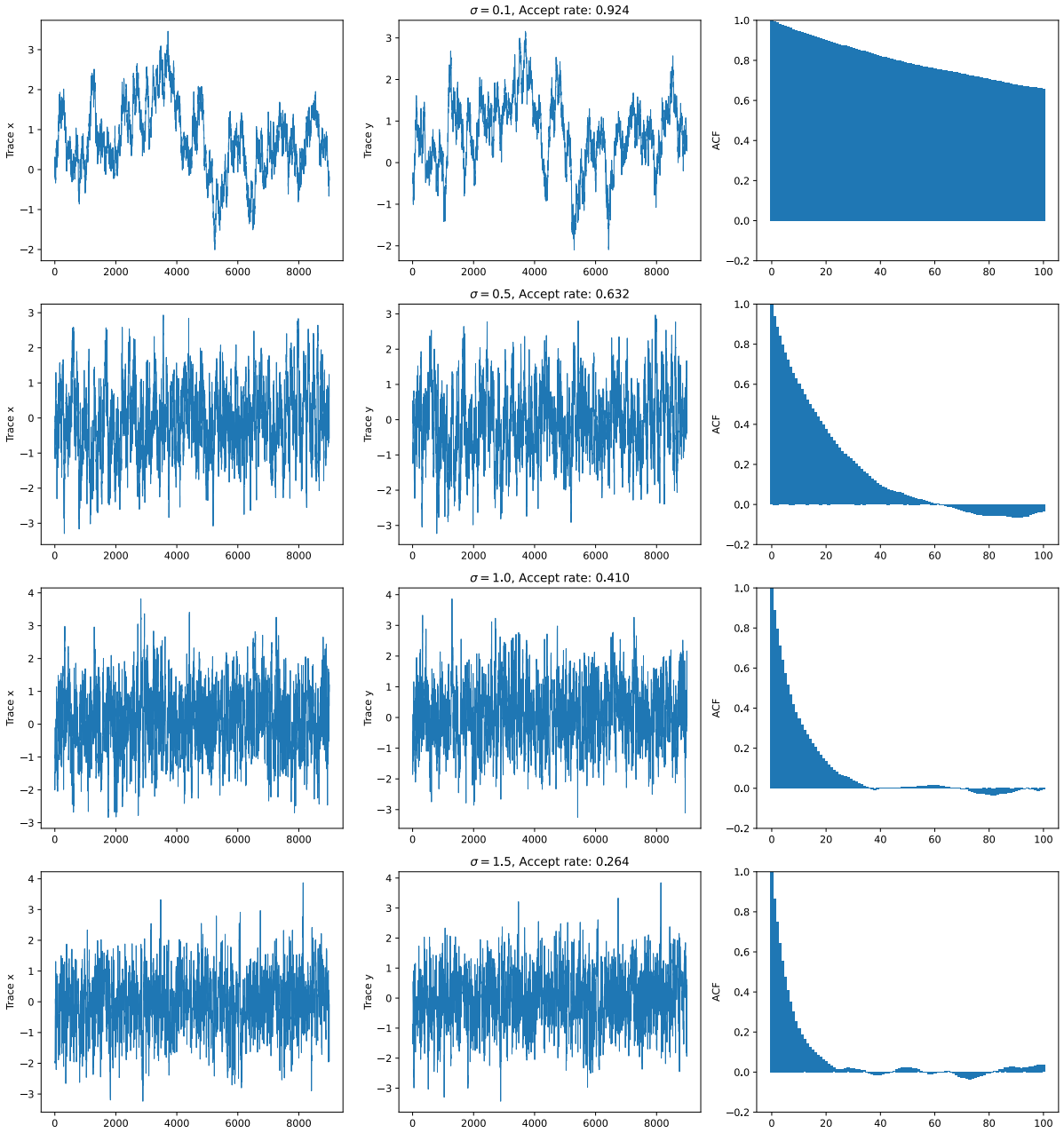

Figure 2: Tuning experiment of Random-walk MH for Gaussian distribution

4.4. 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.

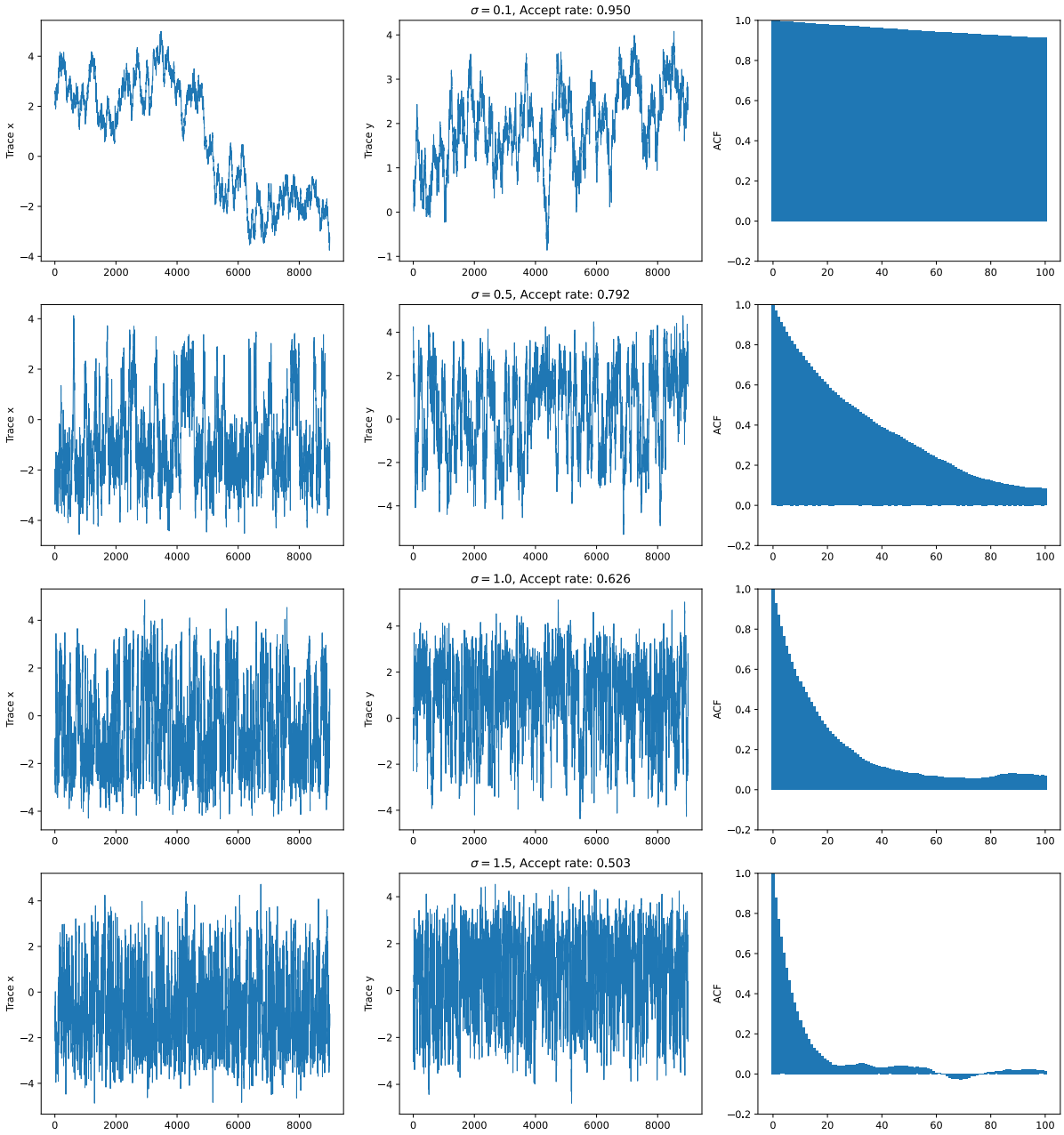

Figure 3: Tuning experiment of Random-walk MH for multimodal distribution

4.5. 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.

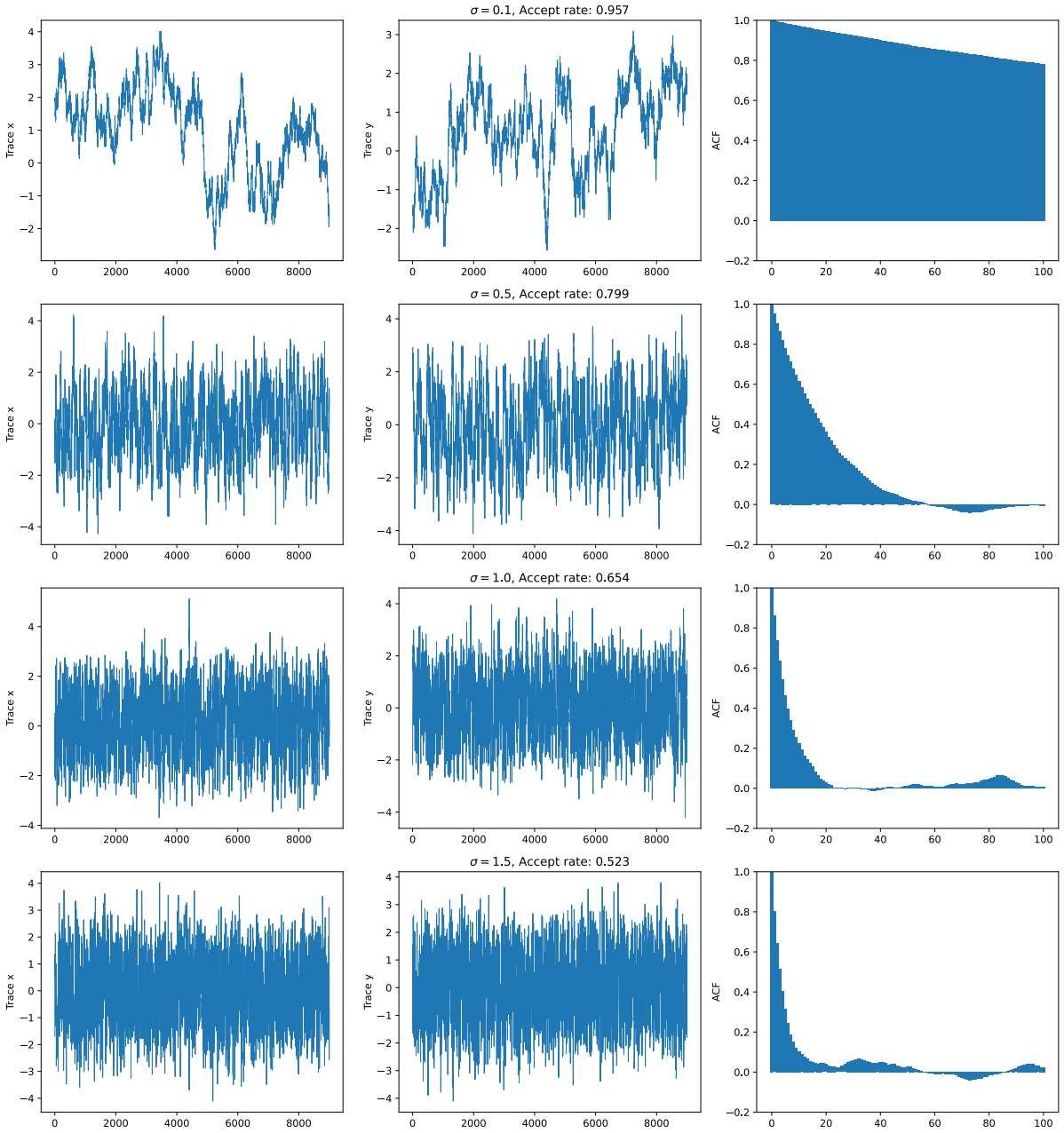

Figure 4: Tuning experiment of Random-walk MH for volcano distribution

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

`langevin.py` Building the Angevin MH kernel

```
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
```

`langevin_chain.py` Running the tuning experiment for the Angevin MH.

```
import os
from pathlib import Path
from scripts.tuning_experiment import run_tuning_experiment
from scripts.langevin import init, build_kernel
from scripts.densities import logdensity_multimodal, logdensity_mvn,
logdensity_volcano


OUTPUT_DIR = Path("output/langevin/")
os.makedirs(OUTPUT_DIR, exist_ok=True)
```

```
print("Running tuning experiment for Langevin Monte Carlo...")
print("- Multivariate Normal distribution...")
run_tuning_experiment(init, build_kernel, logdensity_mvn, OUTPUT_DIR /
"tuning_mvn")
print("- Multimodal distribution...")
run_tuning_experiment(
    init, build_kernel, logdensity_multimodal, OUTPUT_DIR / "tuning_multimodal"
)
print("- Volcano distribution...")
run_tuning_experiment(
    init, build_kernel, logdensity_volcano, OUTPUT_DIR / "tuning_volcano"
)
```

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.

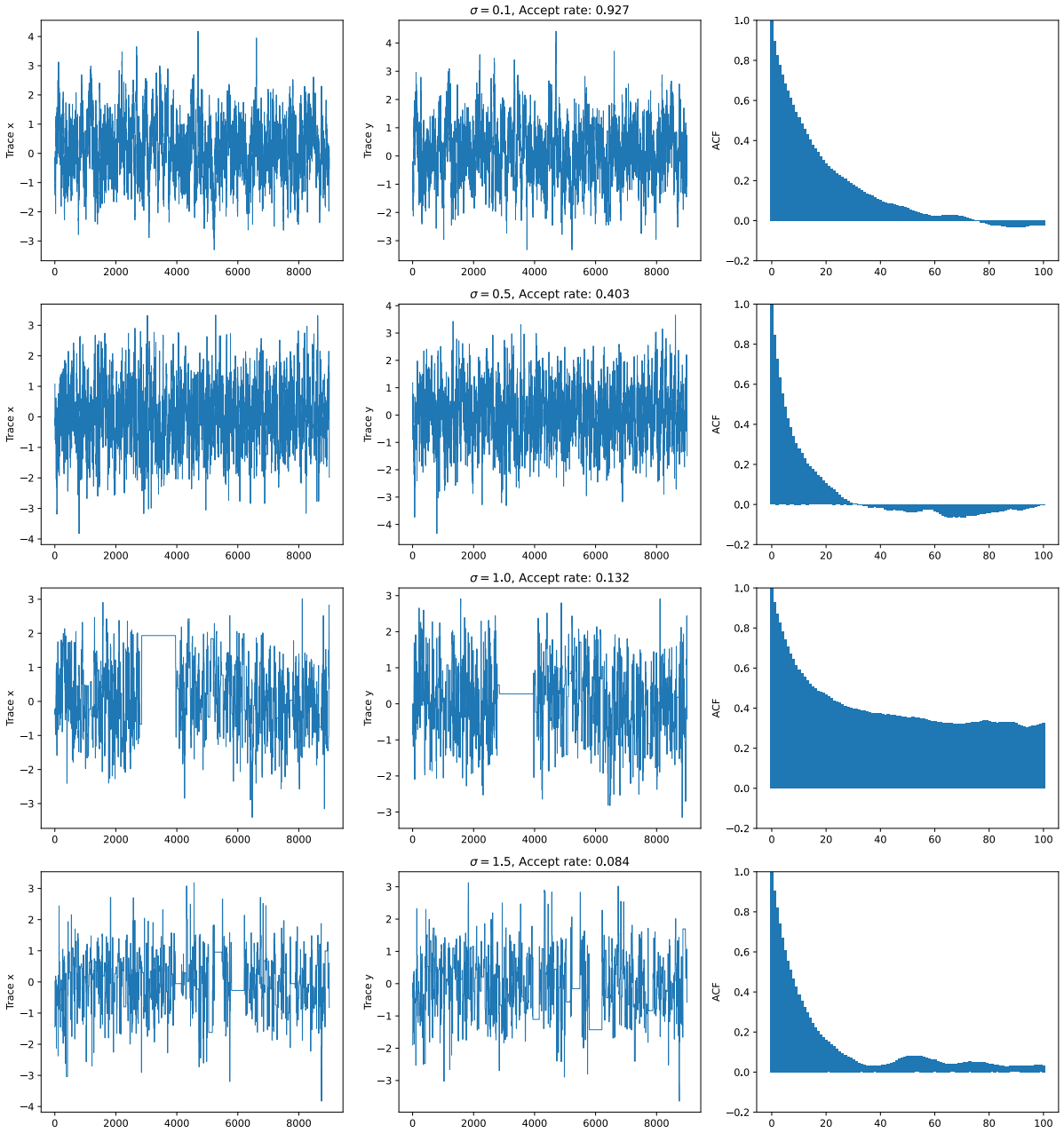

Figure 5: Tuning experiment of Langevin MH for Gaussian distribution

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.

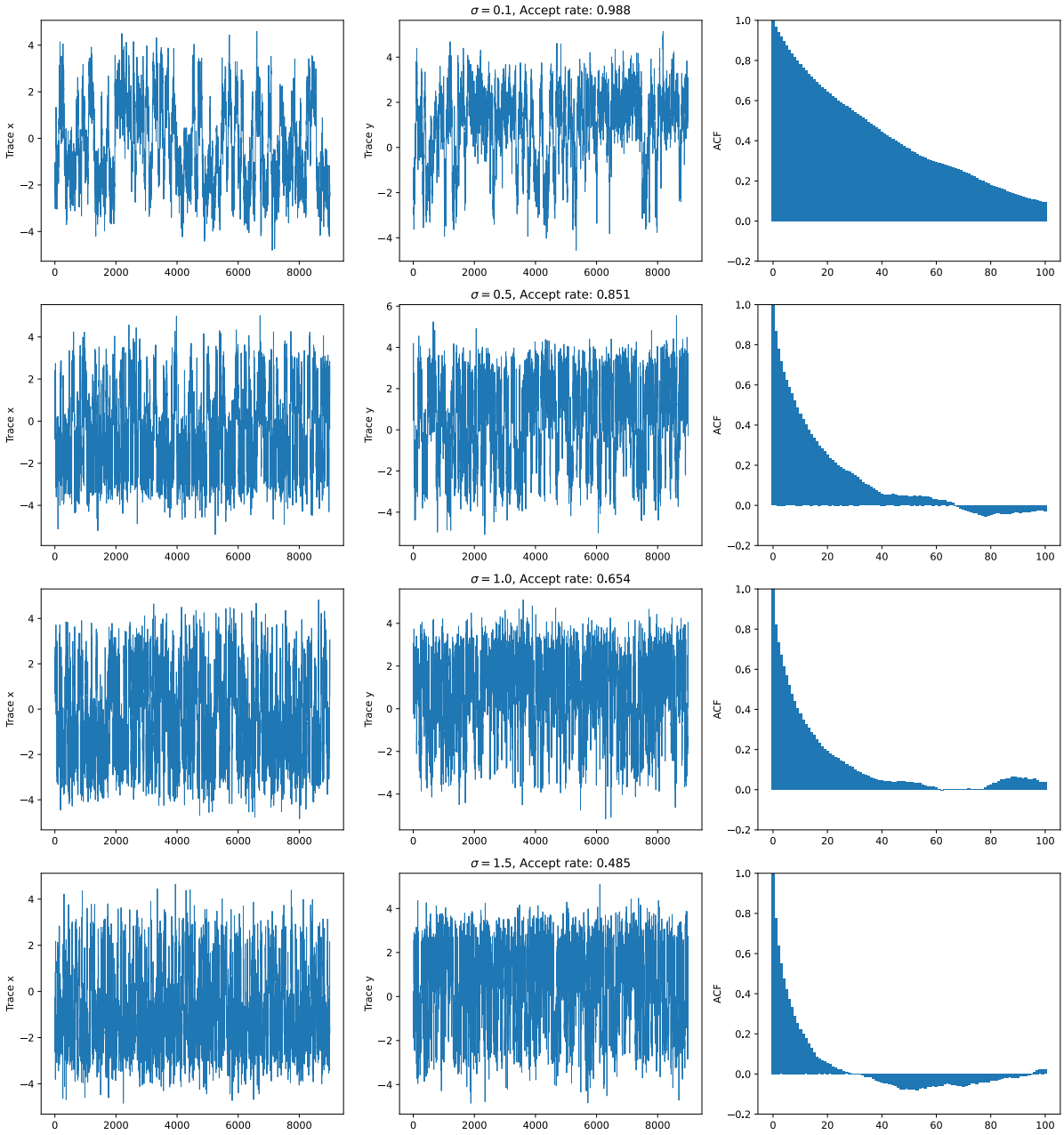

Figure 6: Tuning experiment of Langevin MH for multimodal distribution

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.

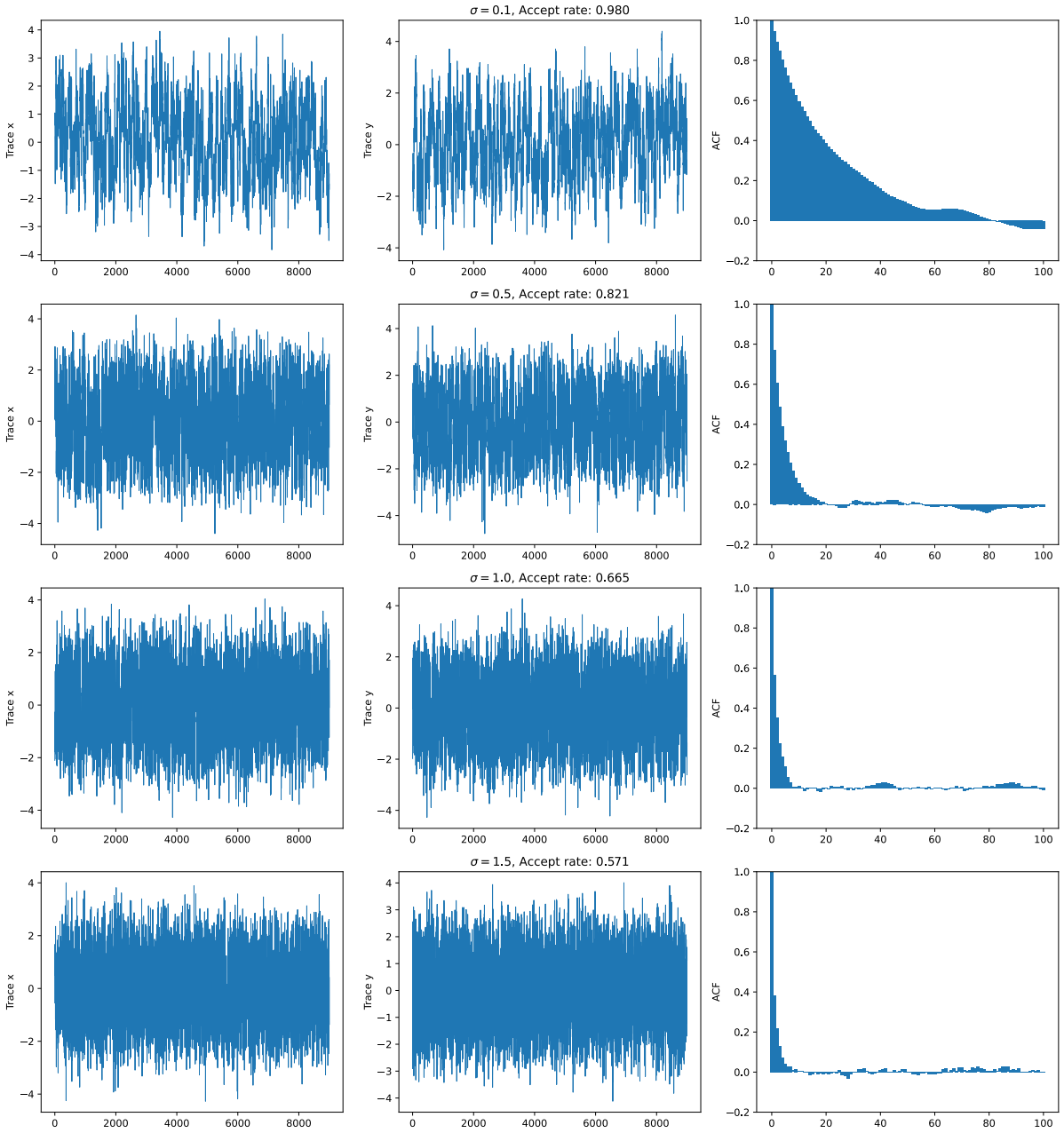

Figure 7: Tuning experiment of Langevin MH for multimodal distribution

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

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

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

`hamiltonian.py` Building the HMH 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
```

`hmc_chain.py` Running the tuning experiment for the HMC.

```
import os
from pathlib import Path

from scripts.densities import logdensity_multimodal, logdensity_mvn,
logdensity_volcano
from scripts.hamiltonian import build_kernel, init
from scripts.tuning_experiment import run_tuning_experiment

OUTPUT_DIR = Path("output/hmc/")
os.makedirs(OUTPUT_DIR, exist_ok=True)

print("Running tuning experiment for Hamiltonian Monte Carlo...")

step_sizes = [0.01, 0.1, 0.25, 0.5]

print("- Multivariate Normal distribution...")
run_tuning_experiment(
    init,
```

```
    build_kernel,
    logdensity_mvn,
    OUTPUT_DIR / "tuning_mvn",
    sigma_values=step_sizes,
)
print("- Multimodal distribution...")
run_tuning_experiment(
    init,
    build_kernel,
    logdensity_multimodal,
    OUTPUT_DIR / "tuning_multimodal",
    sigma_values=step_sizes,
)
print("- Volcano distribution...")
run_tuning_experiment(
    init,
    build_kernel,
    logdensity_volcano,
    OUTPUT_DIR / "tuning_volcano",
    sigma_values=step_sizes,
)
```

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.

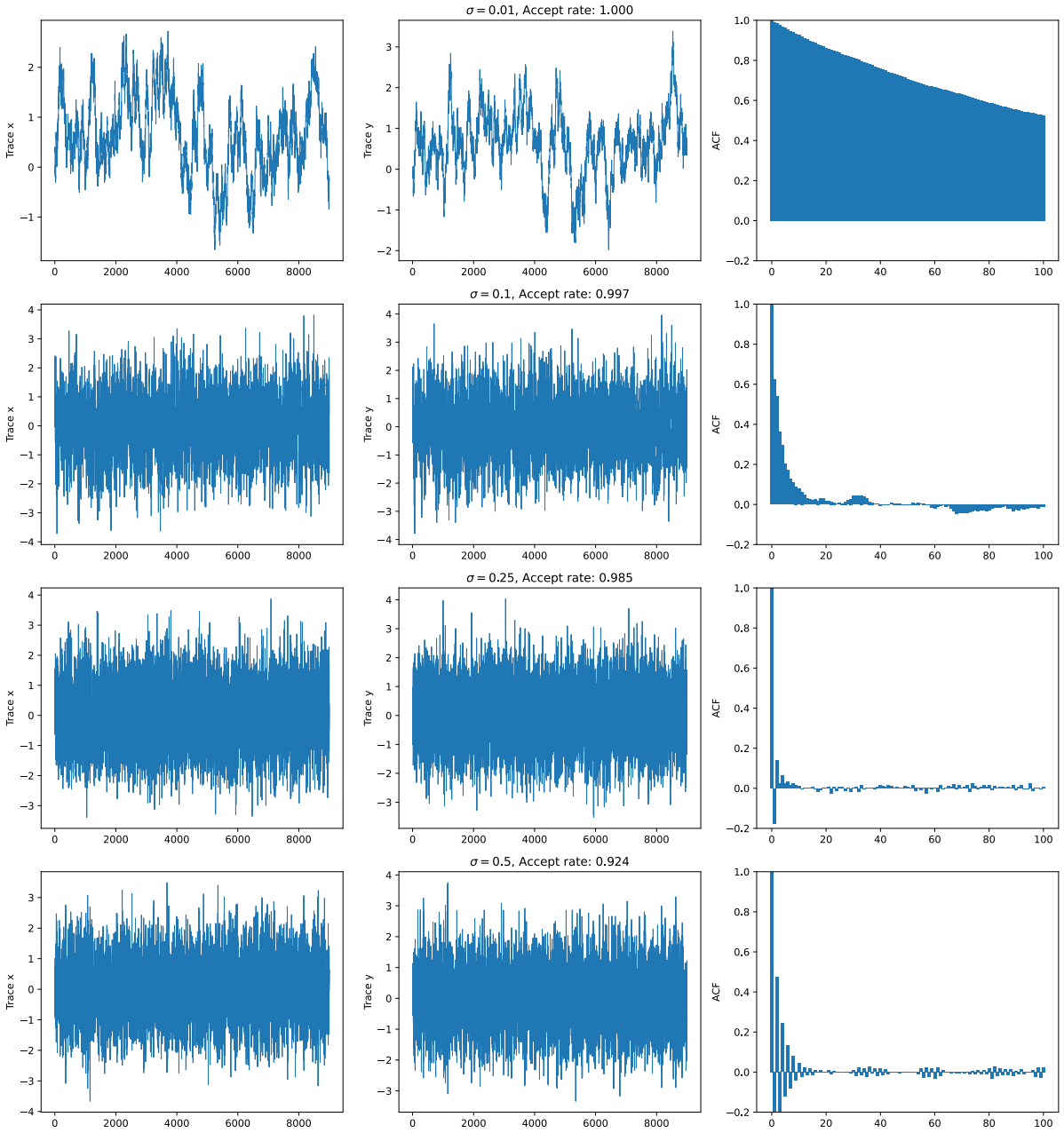

Figure 8: Tuning experiment of HMC for Gaussian distribution

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.

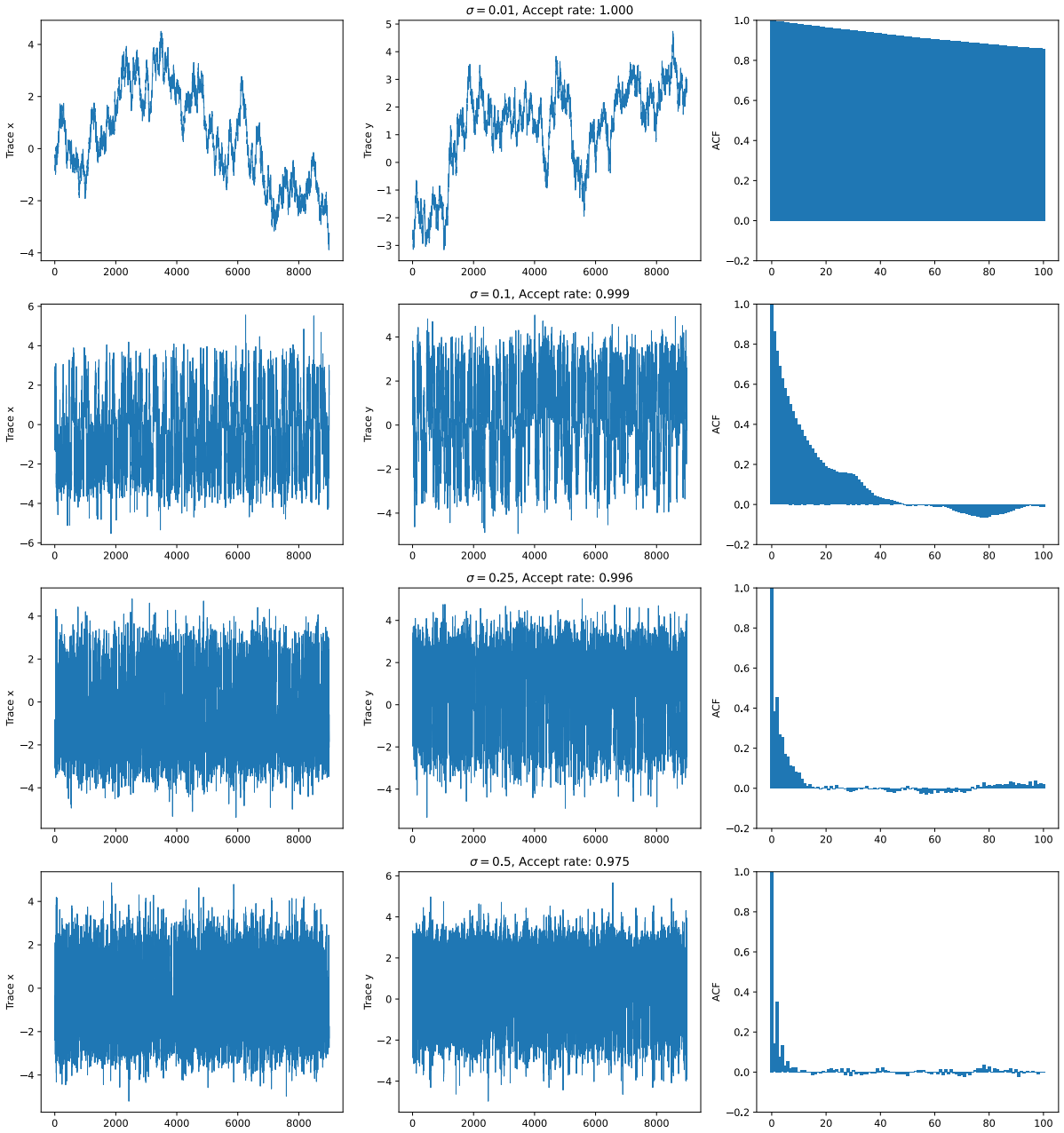

Figure 9: Tuning experiment of HMC for multimodal distribution

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.

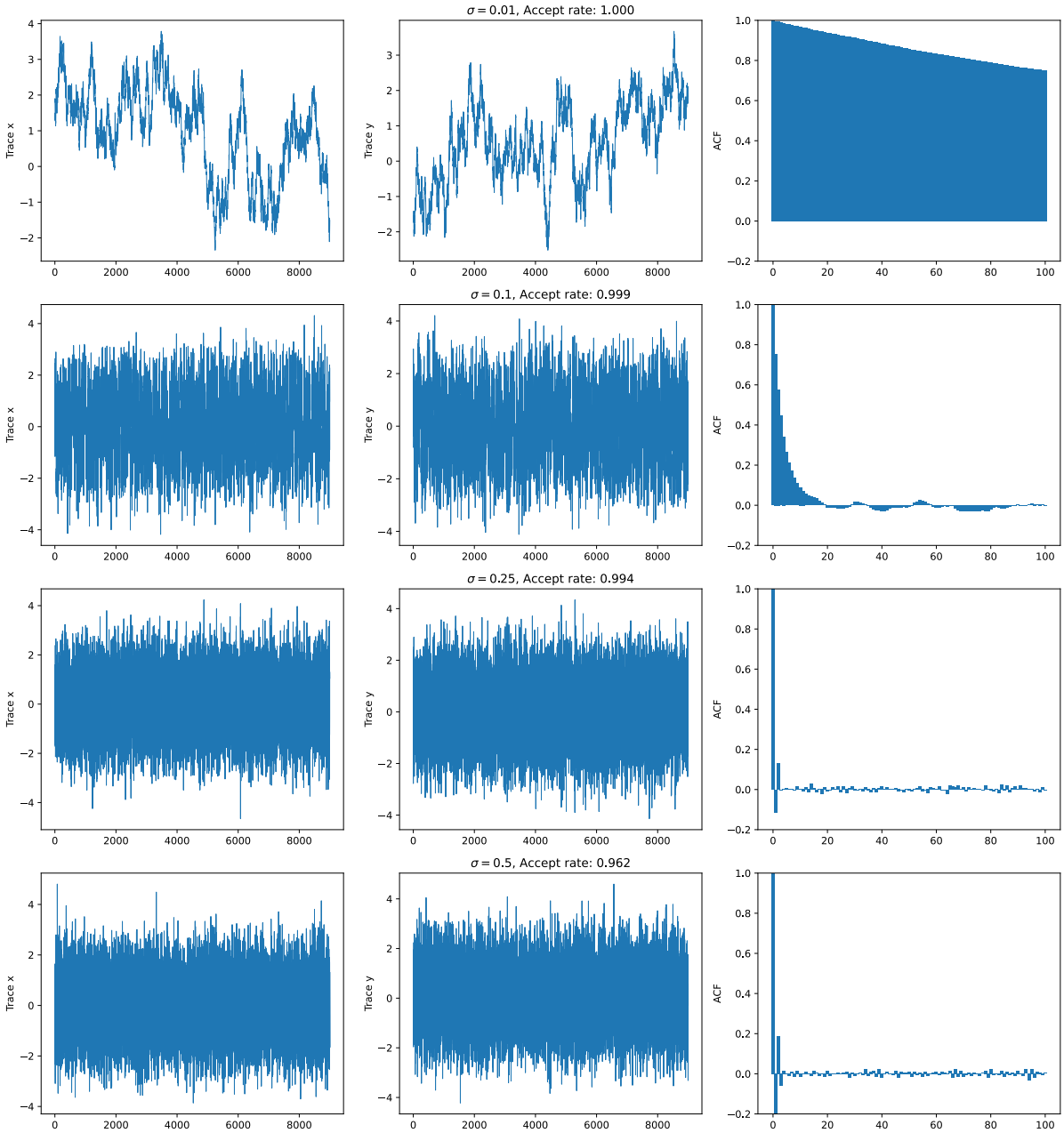

Figure 10: Tuning experiment of HMC for multimodal distribution

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

Dunson, D.B., Johndrow, J.E., 2020. The Hastings algorithm at fifty. Biometrika 107, 1–23.. https://doi.org/10.1093/biomet/asz066

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