

# Fisher, Hamilton Monte Carlo, and JAX

## Fisher matrix

For a distribution  $p(X|\theta)$  of a RV  $X$ , given a parameter  $\theta$ , the Fisher information tells us how much information  $X$  can give us about  $\theta$ .

The Fisher information is defined as

$$I(\theta) = \mathbb{E}_{p(X|\theta)} \left[ \left( \frac{\partial \log p(X|\theta)}{\partial \theta} \right)^2 \middle| \theta \right] \quad (1)$$

$$=^* - \mathbb{E}_{p(X|\theta)} \left[ \frac{\partial^2 \log p(X|\theta)}{\partial \theta^2} \middle| \theta \right]. \quad (2)$$

The second equality applies in most cases of relevance.

### Example: 1d Gaussian with known variance

Let

$$\log p(X|\mu) = -\frac{1}{2} \frac{(X - \mu)^2}{\sigma^2} + \text{const}$$

then

$$I(\mu) = \frac{1}{\sigma^2}$$

Larger  $\sigma$  means less information gained on  $\mu$  by data  $X$ .

## Cramer-Rao bound

The variance of an estimator  $\hat{\theta}$  is bounded from below by the inverse of the Fisher information for that parameter:

$$\text{Var}[\hat{\theta}] \geq \frac{1}{I(\theta)}.$$

For  $n$  iid Gaussian  $\vec{X} \sim \mathcal{N}(\mu, \sigma^2 I_n)$ , with known variance  $\sigma^2$  and unknown mean  $\mu$

$$\log p(\vec{X}|\mu) = -\frac{1}{2} \sum_i \frac{(X_i - \mu)^2}{\sigma^2} + \text{const}$$

The Fisher informaton on  $\mu$  is

$$I(\mu) = \frac{n}{\sigma^2}.$$

The Cramer-Rao bound is therefore

$$\text{Var}[\hat{\mu}] \geq \frac{\sigma^2}{n}$$

That means that the variance of the sample mean is the lowest possible variance of an estimator for the mean.

## Multivariate case

For multiple parameters  $\vec{\theta}$ , the generalisation is the Fisher information matrix

$$I_{ij}(\vec{\theta}) = \mathbb{E}_{p(X|\vec{\theta})} \left[ \left( \frac{\partial \log p(X|\vec{\theta})}{\partial \theta_i} \right) \left( \frac{\partial \log p(X|\vec{\theta})}{\partial \theta_j} \right) \middle| \theta \right] \quad (3)$$

$$= -\mathbb{E}_{p(X|\vec{\theta})} \left[ \frac{\partial^2 \log p(X|\vec{\theta})}{\partial \theta_i \partial \theta_j} \middle| \theta \right] \quad (4)$$

We can get an estimate of the covariance of the parameters  $\vec{\theta}$  by taking the inverse of the Fisher information matrix:

$$\text{Cov}[\vec{\theta}] = I(\vec{\theta})^{-1}.$$

This allows us to make predictions on what the constraints on the parameters are going to be before we take data.

This is only exact in the case of Gaussian distributed parameters but even if not this is a useful tool to get an idea on what an experimental setup (as encoded in the likelihood) is expected to produce.

For the case of a Gaussian likelihood where the covariance  $\Sigma$  does not depend on the parameters, this reduces to

$$I_{ij}(\vec{\theta}) = \sum_{kl} \frac{\partial \mu_k(\vec{\theta})}{\partial \theta_i} (\Sigma^{-1})_{kl} \frac{\partial \mu_l(\vec{\theta})}{\partial \theta_j}$$

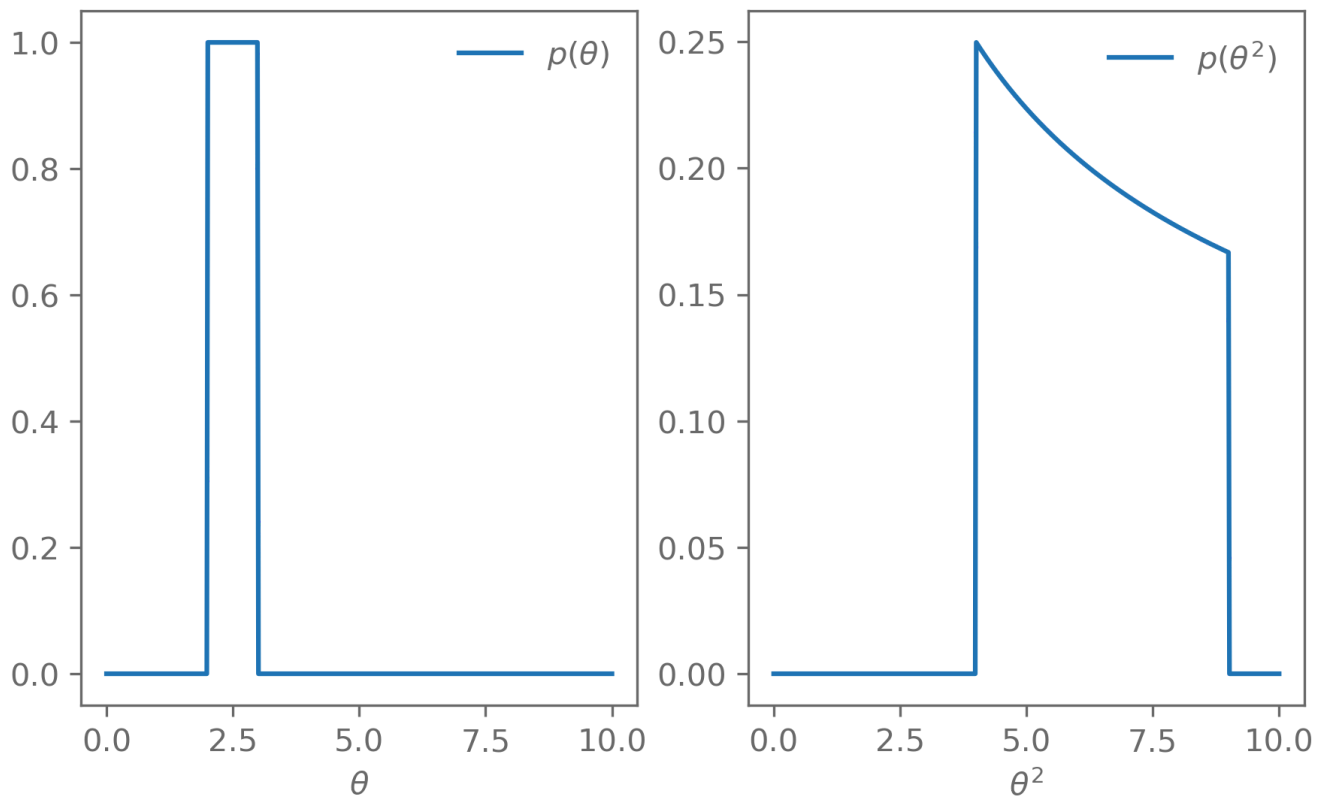
## Jeffreys prior

Another place where the Fisher information appears is the Jeffreys prior.

We have talked about uninformative priors a couple of times so far. We choose uninformative priors when we do not want our posteriors to be influenced by our choice of priors.

But what does uninformative mean?

A flat prior in  $\theta$  will not be flat for  $\theta^2$  for example:



Jeffreys priors address this, as they are constructed such that they are invariant to reparameterisation. This is achieved by setting the prior to

$$p(\theta) \propto \sqrt{\det I(\theta)}.$$

This definition gives us the desired reparameterisation invariance, because under a reparameterisation  $\theta \rightarrow \rho$ , the Fisher information transforms as

$$I(\rho) = I(\theta) \left( \frac{d\theta}{d\rho} \right)^2 ,$$

while the change of variables transforms the distribution  $p(\theta)$  as

$$p(\rho) = p(\theta) \left| \frac{d\theta}{d\rho} \right| .$$

By choosing  $p(\theta) \propto \sqrt{\det I(\theta)}$ , we also have  $p(\rho) \propto \sqrt{\det I(\rho)}$ , the invariance to reparameterisation we are looking for.

## Example: Jeffreys prior for mean and variance

As we have seen before, the Fisher information of the mean of Gaussian is

$$I(\mu) = \frac{1}{\sigma^2}$$

The Jeffreys prior for  $\mu$  is therefore

$$p(\mu) \propto \frac{1}{\sigma} \propto \text{const} ,$$

i.e., it does not depend on  $\mu$ . So for the mean (or location parameters in general), a flat prior is appropriate.

Note that the constant prior is not a probability distribution, since it cannot be normalised. It is an improper prior. In this case the posterior is still well-defined though.

In practice there are usually physical limits on the parameter, which addresses this.

Now, the case where  $\mu$  is fixed but  $\sigma$  is a parameter:

$$I(\sigma) = \frac{2}{\sigma^2}$$

This gives the Jeffreys prior for the standard deviation (and scale parameters in general)

$$p(\sigma) \propto \frac{1}{\sigma}$$

Now the prior depends on  $\sigma$  and is not a flat prior. This is again an improper prior, and depending on the likelihood, it might lead to an ill-defined posterior.

## JAX

[JAX](#) brings autodifferentiation to numpy and allows it to run fast on CPUs, GPUs, etc.

I will go briefly over the main features here but I suggest you have a look at the [tutorial](#) (or others, in case you find something better) to get a feel for the features.

## Autodifferentiation

The code we run on computers ultimately is just a sequence of elementary operations, such as addition and multiplication.

If we can track all the operations a function executes internally, we can use the chain rule to compute the derivatives of the function.

Autodifferentiation frameworks do the hard work for us by tracking all the operations from our code down to the elementary operations and giving us access to derivatives.

```
from jax import grad

# Define our function
def f(x):
    return x**2 - 1

# jax.grad takes a function and returns another function, which is the gradient
df_dx = grad(f)
d2f_dx2 = grad(df_dx)

x = 1.0
print(f"f({x}) = {f(x)}")
print(f"df({x})/dx = {df_dx(x)}")
print(f"d^2f({x})/dx^2 = {d2f_dx2(x)}")
```

```
f(1.0) = 0.0
df(1.0)/dx = 2.0
d^2f(1.0)/dx^2 = 2.0
```

## Auto-vectorisation

JAX also gives a convenient utility to vectorise our functions.

```
from jax import vmap

# Define a vectorised version of our derivative df/dx
batched_df_dx = vmap(df_dx)

batched_df_dx(jnp.array([1.0, 2.0]))
```

```
Array([2., 4.], dtype=float32)
```

## Just-in-time compilation

Finally, JAX allows compiling functions to run fast on CPUs, as well as GPUs. This puts some [mild restrictions](#) on how your code needs to be structured (see also

[https://jax.readthedocs.io/en/latest/notebooks/Common\\_Gotchas\\_in\\_JAX.html](https://jax.readthedocs.io/en/latest/notebooks/Common_Gotchas_in_JAX.html)).

```
from jax import jit

jitted_batched_df_dx = jit(batched_df_dx)

# For complex functions, this can be MUCH faster
jitted_batched_df_dx(jnp.array([1.0, 2.0]))
```

```
Array([2., 4.], dtype=float32)
```

## Random numbers

The way that JAX handles random numbers is a bit different to how they are being handled in other frameworks, such as numpy.

For details, have a look at <https://jax.readthedocs.io/en/latest/jax-101/05-random-numbers.html>

The core is that instead of setting a seed once and each subsequent call to a function that generates random numbers advances the seed automatically, each random function needs to be given an explicit key.

New keys are generated by splitting an existing key:

```
key = jax.random.PRNGKey(42)

key, sub_key = jax.random.split(key)
```

## Example: gradients of a log probability

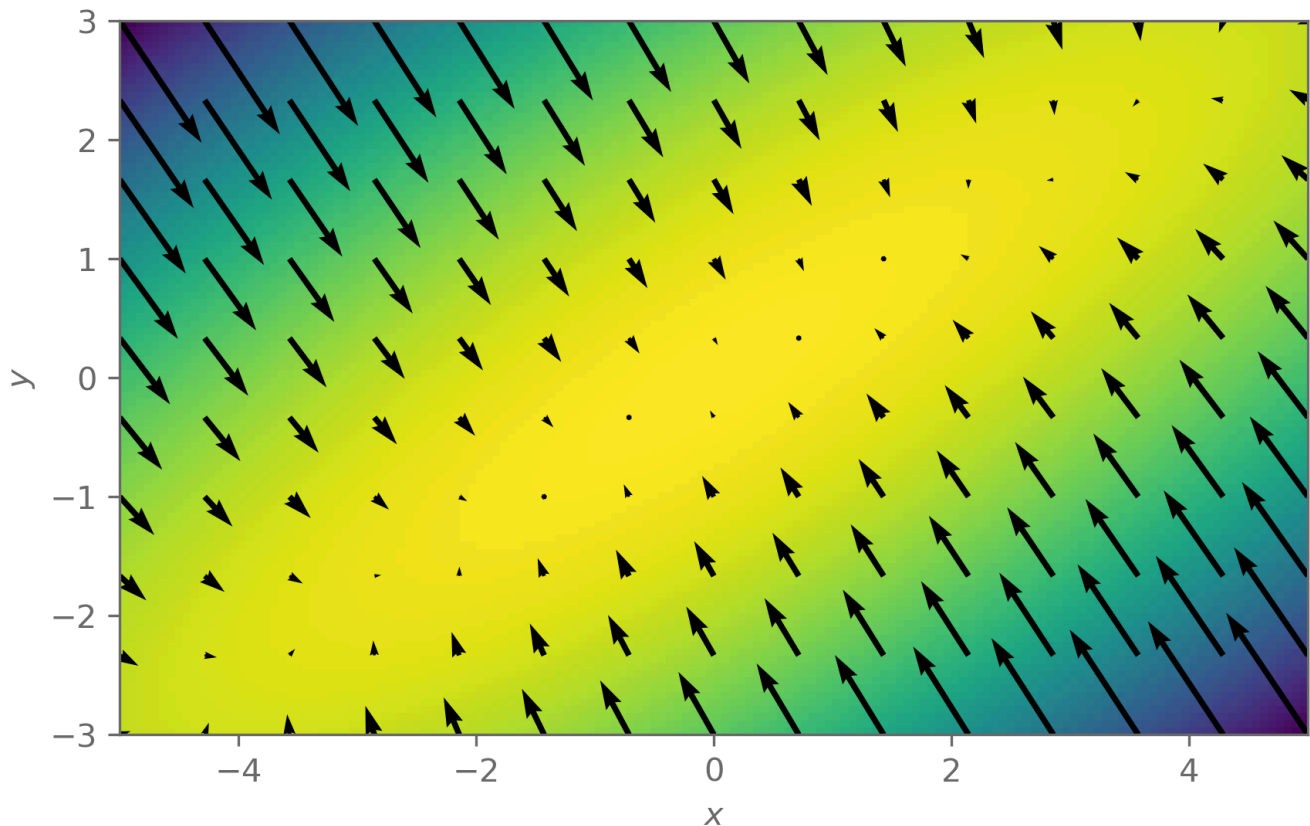
```
# Define variances and correlation
sigma_x = 1.5
sigma_y = 1
rho = 0.85

# Define mean and covariance
mean = jnp.array([0.0, 0.0])
cov = jnp.array([[sigma_x**2, sigma_x*sigma_y*rho],
                 [sigma_x*sigma_y*rho, sigma_y**2]])
cov_inv = jnp.linalg.inv(cov)

# Create distribution object
def log_pdf(x):
    r = x - mean
    return -0.5 * r @ cov_inv @ r

x_grid = jnp.linspace(-5, 5, 15)
y_grid = jnp.linspace(-3, 3, 10)
xx, yy = jnp.meshgrid(x_grid, y_grid)
coords = jnp.dstack((xx, yy)).reshape(-1, 2)
```

```
# Compute the gradients
grads = vmap(grad(log_pdf))(coords)
```



## Example: Fisher information of the Jovian moon data

We can use the Fisher information matrix to predict what kind of constraints we can expect from our data. For complex and computationally expensive likelihoods this is a useful to get a rough estimate of the constraints without having to run an expensive MCMC.

Let us go back to the Jovian moon example.

```
moon = "Io"
t, distance, distance_alt, distance_err = np.loadtxt(
    f"./data/jovian_moons/{moon}.dat", unpack=True)

# Use the offset between the distance estimates as an extra error
cov = np.diag(distance_err**2 + (distance-distance_alt)**2)
inv_cov = np.linalg.inv(cov)
```

```
def model(theta, t):
    semimajor, period, phi = theta
    return semimajor * jnp.sin(2*np.pi/period * t + phi)

def log_likelihood(theta):
    mu = model(theta, t=t)
    r = distance - mu
    return -0.5 * r @ inv_cov @ r
```

```
def log_prior(theta):
    # Use flat priors for now
    return 0

def log_posterior(theta):
    return log_likelihood(theta) + log_prior(theta)
```

```
theta_MAP = scipy.optimize.minimize(
    fun=lambda x: -log_posterior(x),
    jac=grad(lambda x: -log_posterior(x)), # We can use JAX's autodifferentiation
                                           # to use gradient-based optimisation
                                           # methods.

    method="L-BFGS-B",
    x0=(1.0, 40.0, 1.0),
).x

semimajor_MAP, period_MAP, phase_MAP = theta_MAP
print("MAP results")
print(f"{semimajor_MAP=:.3f}, {period_MAP=:.3f}, {phase_MAP=:.3f}")
```

MAP results

semimajor\_MAP=0.419, period\_MAP=42.323, phase\_MAP=2.912

Because we use a Gaussian likelihood with parameter-independent covariance

$$I_{ij}(\vec{\theta}) = \sum_{kl} \frac{\partial \mu_k(\vec{\theta})}{\partial \theta_i} (\Sigma^{-1})_{kl} \frac{\partial \mu_l(\vec{\theta})}{\partial \theta_j}$$

```
jacobian = jacfwd(model)(theta_MAP, t)

fisher_I = np.einsum("ip,ij,js", jacobian, inv_cov, jacobian)

fisher_cov = np.linalg.inv(fisher_I)

print(f"Expected uncertainty on a: {np.sqrt(cov[0,0]):.4f}")
print(f"Expected uncertainty on T: {np.sqrt(cov[1,1]):.3f}")
```

Expected uncertainty on a: 0.0087

Expected uncertainty on T: 0.010

```
n_param = 3
n_walker = 10
n_step = 8000

# The starting point for each walker
theta_init = np.array([0.5, 40.0, 3.0]) \
    + 0.1*np.random.normal(size=(n_walker, n_param))

sampler = emcee.EnsembleSampler(
    nwalkers=n_walker, ndim=n_param,
    log_prob_fn=jit(vmap(log_posterior)),
    vectorize=True
)

state = sampler.run_mcmc(theta_init, nsteps=n_step, progress=True)
```



```

print("Auto-correlation time:")
for name, value in zip(["a", "T", "phi"], autocorr_time := sampler.get_autocorr_time()):
    print(f"{name} = {value:.1f}")

chain = sampler.get_chain(
    discard=int(5*max(autocorr_time)),
    thin=int(max(autocorr_time)/4),
    flat=True
)

```

100% |██████████| 8000/8000 [00:02<00:00, 2731.44it/s]

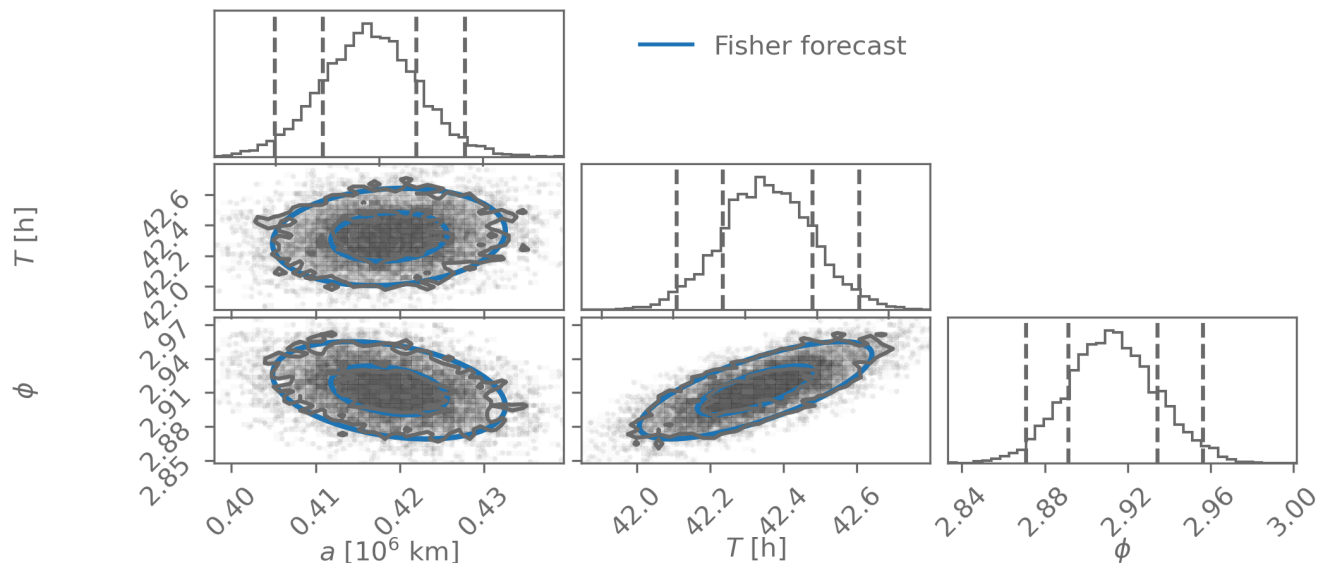
Auto-correlation time:

a = 51.4

T = 38.9

phi = 57.6

How does the Fisher estimate of the parameter covariance compare to the posteriors we get from a MCMC?



The Fisher information matches the posterior well. One reason is that we did not include any prior information.

Instead of the Fisher information (which is the expectation over the Hessian of the log likelihood), we might just want to get a Gaussian approximation of the posterior.

Expand the log-posterior around the posterior mode  $\vec{\theta}^*$ :

$$\log p(\vec{\theta}|\vec{d}) = \log p(\vec{\theta}^*|\vec{d}) + \frac{1}{2}(\vec{\theta} - \vec{\theta}^*)_i \left( \frac{\partial^2}{\partial \theta_i \partial \theta_j} \log p(\vec{\theta}|\vec{d}) \right)_{\vec{\theta}=\vec{\theta}^*} (\vec{\theta} - \vec{\theta}^*)_j.$$

The term  $\propto (\vec{\theta} - \vec{\theta}^*)$  is zero at the mode.

From this expansion we see that the posterior can be approximated by

$$p(\vec{\theta}|\vec{d}) \approx \mathcal{N}(\vec{\theta}^*, I(\vec{\theta}^*)^{-1}),$$

where the *observed* information  $I(\vec{\theta})$  is given by

$$I(\vec{\theta}) = -\frac{\partial^2}{\partial \theta_i \partial \theta_j} \log p(\vec{\theta}|\vec{d}).$$

```
# JAX makes this very easy
posterior_observed_information = -hessian(log_posterior)(theta_MAP)
posterior_covariance_estimate = np.linalg.inv(posterior_observed_information)
```

## Exercises

Read the JAX tutorials.

Use JAX to confirm the Fisher information for the case of a Gaussian with either fixed mean or fixed variance.

Implement one of the likelihoods you have used so far (for example from the quadratic model fit exercise) in JAX and compute the Fisher information matrix.

## Hamiltonian Monte Carlo

Hamiltonian Monte Carlo is an MCMC algorithm that scales well to high dimensions.

The downside is that it requires the gradient of the log probability. But with JAX we can get that easily now.

Hamilton Monte Carlo casts the sampling problem as a physics problem of exploring a potential using Hamiltonian dynamics.

We define a Hamiltonian

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

with potential  $U(\vec{x})$  at position  $\vec{x}$  and kinetic energy  $K(\vec{p}) = \frac{1}{2} \vec{p}^T M^{-1} \vec{p}$  of momentum  $\vec{p}$  with mass matrix  $M$ .

From this we can define a probability

$$p_H(\vec{x}, \vec{p}) = \frac{1}{Z} e^{-H(\vec{x}, \vec{p})} = \frac{1}{Z} e^{-U(\vec{x})} e^{-K(\vec{p})}$$

The positions and momenta are independent, such that the marginal distribution of  $\vec{x}$  is

$$p(\vec{x}) \propto e^{-U(\vec{x})}$$

This is the distribution we want to sample from. To do this, we define  $U(\vec{x}) = -\log p(\vec{x})$ .

Hamiltonian dynamics are defined by the equations of motion

$$\frac{dx_i}{dt} = \frac{\partial H}{\partial p_i} \quad (5)$$

$$\frac{dp_i}{dt} = -\frac{\partial H}{\partial x_i} \quad (6)$$

In our case we have

$$\frac{dx_i}{dt} = \frac{\partial K}{\partial p_i} = (M^{-1}\vec{p})_i \quad (7)$$

$$\frac{dp_i}{dt} = -\frac{\partial U}{\partial x_i} \quad (8)$$

Hamiltonian Monte Carlo works as follows:

Start at  $\vec{x}_0$ .

1. Sample  $\vec{p}_0$  from a Gaussian with mean 0 and covariance  $M$ :  $\vec{p}_0 \sim \mathcal{N}(0, M)$
2. Evolve  $\vec{x}$  and  $\vec{p}$  with the dynamics defined by the Hamiltonian using the leap frog algorithm
3. Accept the new position using a Metropolis-Hastings acceptance criterion based on the integration error of the Hamiltonian

The leap frog algorithm is a method to integrate the equation of motion to evolve the system a small amount  $\epsilon$  from time  $t$  to time  $t + \epsilon$ .

1. Evolve the momentum a half step:  $\vec{p}_{t+\epsilon/2} = \vec{p}_t - \frac{\epsilon}{2} \nabla U(\vec{x}_t)$
2. Evolve the position by a full step:  $\vec{x}_{t+\epsilon} = \vec{x}_t + \epsilon M^{-1} \vec{p}_{t+\epsilon/2}$
3. Evolve the momentum another half step:  $\vec{p}_{t+\epsilon} = \vec{p}_{t+\epsilon/2} - \frac{\epsilon}{2} \nabla U(\vec{x}_{t+\epsilon})$

Repeat this for  $L$  steps to move from the state  $\vec{x}_{\text{old}}, \vec{p}_{\text{old}}$  to  $\vec{x}_{\text{new}}, \vec{p}_{\text{new}}$ .

Then accept the new state  $\vec{x}_{\text{new}}, \vec{p}_{\text{new}}$  with probability

$$a = \min \left( 1, \frac{e^{-H(\vec{x}_{\text{new}}, \vec{p}_{\text{new}})}}{e^{-H(\vec{x}_{\text{old}}, \vec{p}_{\text{old}})}} \right).$$

If the equations of motions are solved exactly, the Hamiltonian is be conserved, and the acceptance probability is always be 1.

In practice, we get an integration error, which depends on the step size  $\epsilon$  and number of steps  $L$ . These are the main tuning parameters.

For the integration to be stable,  $\epsilon$  needs to be smaller than the smallest length scale in the distribution we want to sample.

The number of steps should be chosen such that  $L\epsilon$  is long enough to cover the longest length scale.

In addition, the mass matrix  $M$  can be chosen to align the momenta better with the shape of the distribution.

```
def sample_hmc(x0, log_pdf, n, L, epsilon, mass_matrix, key):

    mass_matrix_inv = jnp.linalg.inv(mass_matrix)

    # Define the potential and Hamiltonian
    def U(x):
        return -log_pdf(x)

    def H(x, p):
        return U(x) + 0.5 * p @ mass_matrix_inv @ p

    # Use JAX to get the gradient of U
    grad_x_U = grad(U)

    # The leap frog integrator. We speed this up using just-in-time compilation
    @jit
    def leap_frog(x, p):
        t = [x]
        for _ in range(L):
            p_prime = p - epsilon/2 * grad_x_U(x)
            x = x + epsilon * mass_matrix_inv @ p_prime
            p = p_prime - epsilon/2 * grad_x_U(x)
            t.append(x)
        return x, p, jnp.array(t)

    samples = [x0]
    trajectories = []
    n_accpeted = 0
    for _ in range(n):
        key, subkey = jax.random.split(key)
        # Sample a new momentum
        p0 = jax.random.multivariate_normal(
            subkey, mean=jnp.zeros_like(x0), cov=mass_matrix
        )
        # Integrate Hamiltonian dynamics
        x, p, t = leap_frog(x0, p0)

        # Metropolis-Hastings step
        a = min(1, jnp.exp(H(x0, p0) - H(x, p)))

        key, subkey = jax.random.split(key)
        u = jax.random.uniform(subkey)
        if a >= u:
            x0 = x
            n_accpeted += 1
```

```

    samples.append(x0)
    trajectories.append(t)

    acceptance_rate = n_accepted/n
    return jnp.array(samples), trajectories, acceptance_rate

```

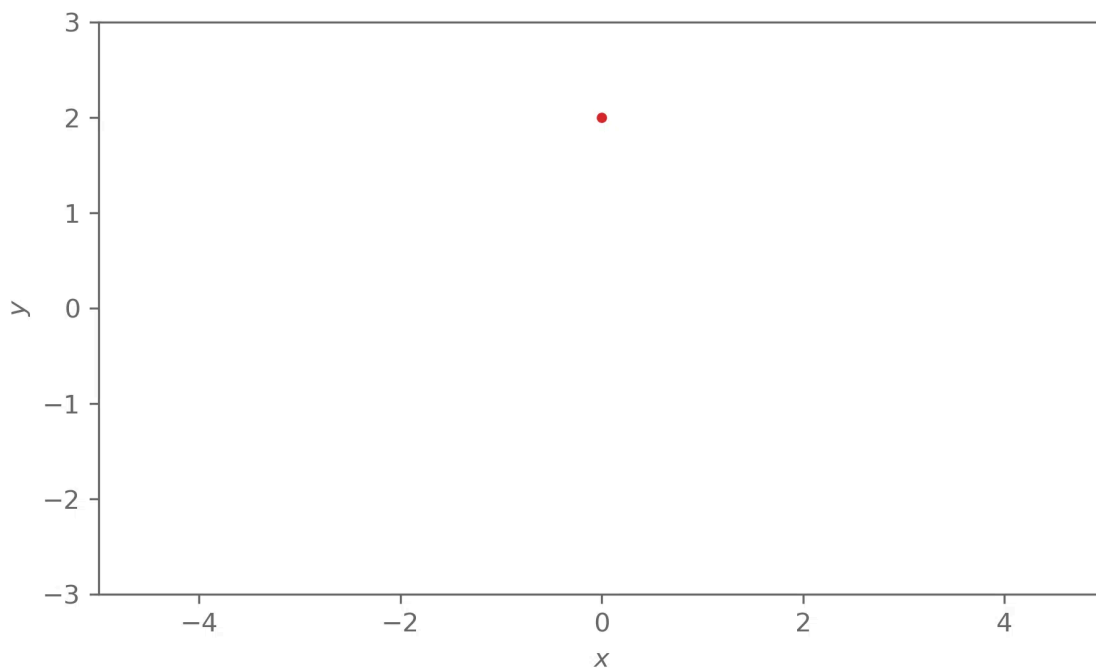
```

# Setup our PRNG keys
key, subkey = jax.random.split(jax.random.PRNGKey(42))

samples, trajectories, acceptance_rate = sample_hmc(
    x0=jnp.array([0.0, 2.0]),
    log_pdf=log_pdf,
    n=30,
    L=20, epsilon=0.2, mass_matrix=jnp.eye(2),
    key=subkey
)
print(f"Acceptance rate: {acceptance_rate:.3f}")

```

Acceptance rate: 0.967



The tuning steps are important but can be tedious. Mature implementations, such as the no-U-turn algorithm (NUTS) with automatic tuning are available.

Here is an implementation in `tensorflow_probability`. This works without the need for `tensorflow` thanks to the JAX backend.

```

import tensorflow_probability.substrates.jax as tfp

num_results = 2000
num_burnin_steps = 1000

```

```

# Set the adaptation strategy...
adaptive_hmc = tfp.mcmc.DualAveragingStepSizeAdaptation(
    # ... for the sampler, which is NUTS here
    tfp.mcmc.NoUTurnSampler(
        target_log_prob_fn=jax.jit(jax.vmap(log_pdf)), # The target distribution
        step_size=1.),
    num_adaptation_steps=int(num_burnin_steps * 0.8) # How many steps to use for tuning
)

```

```

def run_chain(seed):
    # Run the chain (with burn-in).
    samples, info = tfp.mcmc.sample_chain(
        num_results=num_results,
        num_burnin_steps=num_burnin_steps,
        current_state=jnp.array([0.0, 0.0]),
        kernel=adaptive_hmc,
        trace_fn=lambda _, pkr: (pkr.inner_results.target_log_prob, pkr.inner_results
        seed=seed
    )

    log_prob, is_accepted = info
    print(f"Acceptance rate: {is_accepted.sum()/is_accepted.size}")
    return samples, log_prob

samples, log_prob = run_chain(seed=jax.random.PRNGKey(42))

```

Acceptance rate: 0.8647500276565552

## Exercises

Implement HMC yourself.

Experiment with the step size, number of steps, and mass matrix.

Try an implementation of NUTS, for example the one in tensorflow probability shown earlier.