```
from functools import partial

import numpy as np
import scipy.stats

import matplotlib.pyplot as plt

from matplotlib.animation import ArtistAnimation
from IPython.display import HTML
```

Sampling from distributions

In Bayesian data analyses we often need to sample from probability distributions that cannot be sampled from directly.

Luckily the rise of Monte Carlo algorithms and powerful computers have made this possible.

We first have a look at rejection sampling. Conceptionally easy but very inefficient in high dimensions.

Then we go through three common sampling approaches:

- Metropolis-Hastings
- Slice sampling
- Nested sampling

Hamilton Monte Carlo and variational inference we leave until later.

The standard reference for this topic is chapter 29 in Information Theory, Inference, and Learning Algorithms.

Once we have samples from a distribution we can compute expectations under this distribution.

If we want to evaltuate the expectation of $f(x)$ with respect to the distribution p:

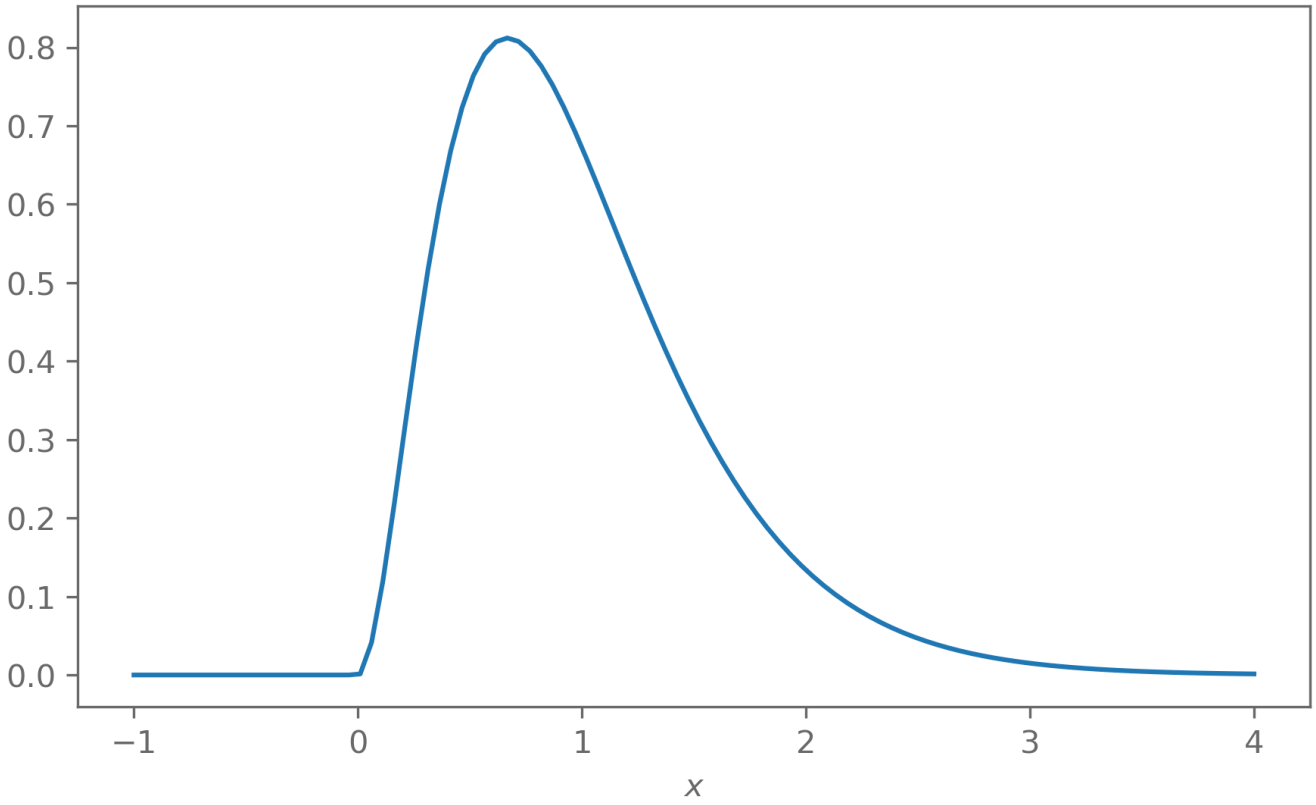
$$\Phi = \mathrm{E}[f(\vec{x})] = \int f(\vec{x}) p(\vec{x}) \mathrm{d}x^n$$

If we have N samples $\vec{x}_i \sim p$, we can approximate Φ as

$$\hat{\Phi} = \frac{1}{N} \sum_i f(\vec{x}_i)$$

If we have N samples x_{ij} from an d-dimensional distribution ($i = 1, \ldots, N$, $j = 1, \ldots, d$), we get samples from the marginal distributions by taking just dropping the columns of the matrix x_{ij} that correspond the the dimensions we want to marginalise over.

Our target distribution we want to sample from:



Rejection sampling

The basic idea is to generate points (x, y) that sample the area under $p(x)$ uniformly.

While we cannot sample from $p(x)$, we assume can find a distribution $q(x)$ that we can sample from and for which

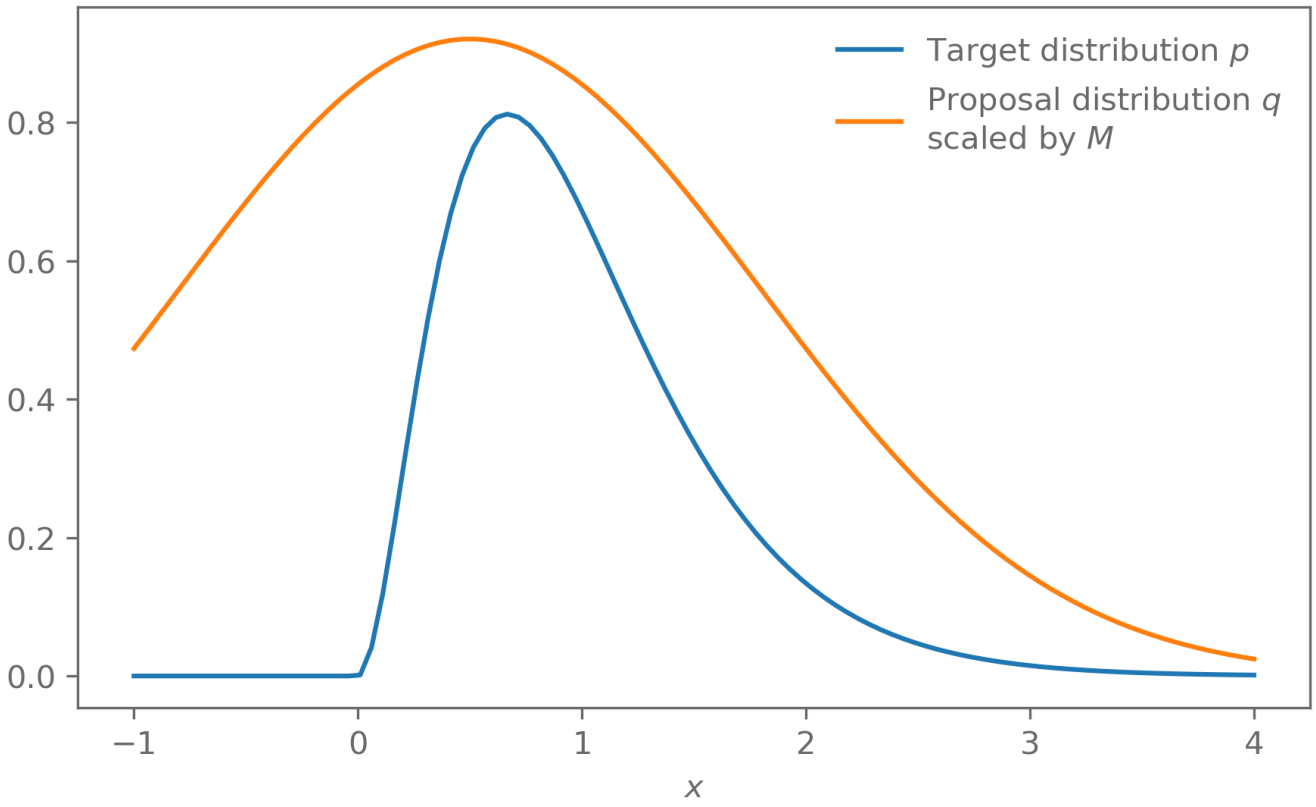
$$Mq(x) > p(x) \; \forall x$$

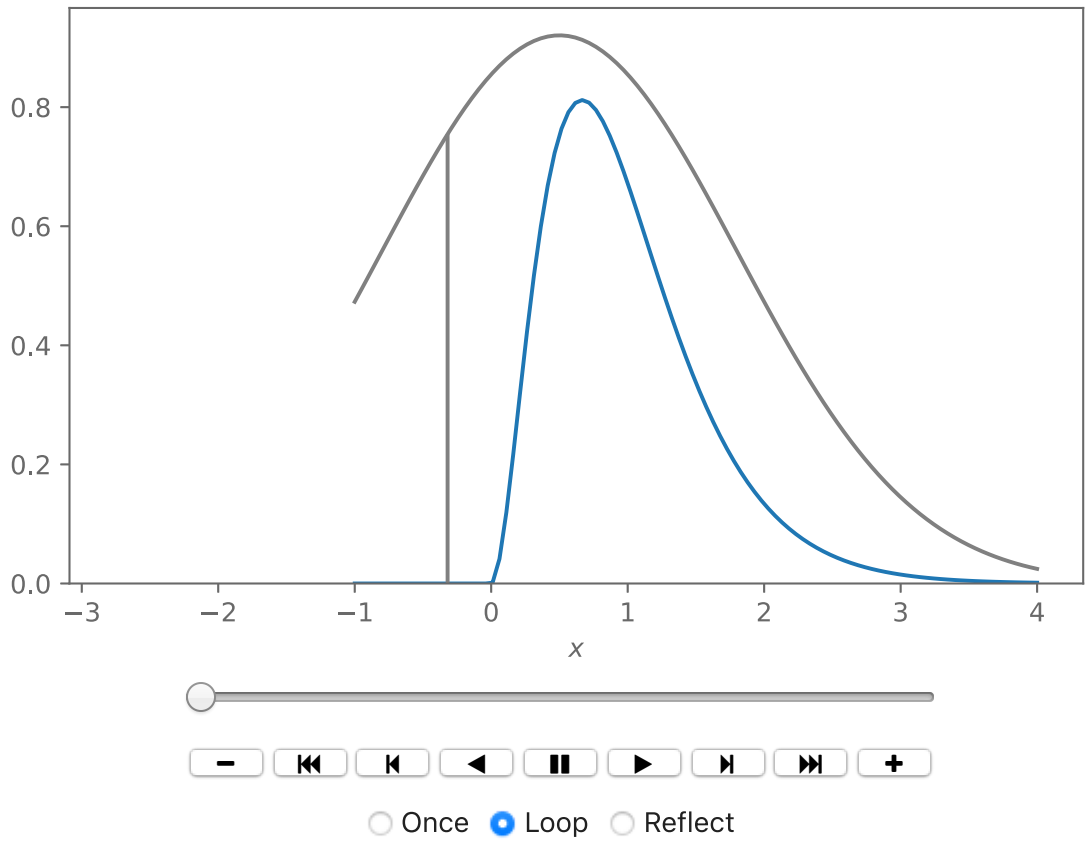
for some constant M.

We then sample x_i from $q(x)$ and $u_i | x_i \sim \mathcal{U}(0, Mq(x_i))$. The points (x_i, u_i) sample the are under the curve $Mq(x)$ uniformly.

From this sample of points, we remove those where $u_i > p(x_i)$, which leaves us with points that uniformly sample the area under $p(x)$ and thus $x_i \sim p$.

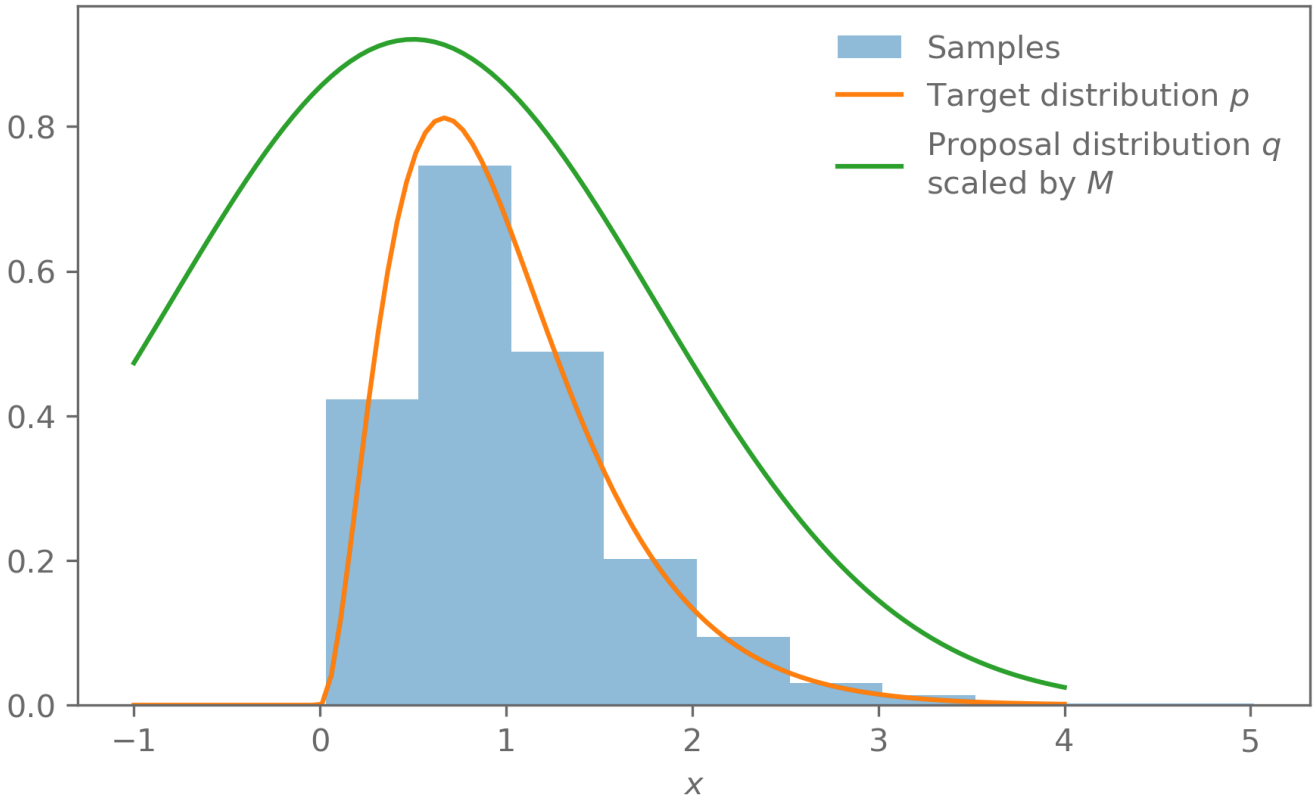
```
# We use a Gaussian as our proposal distribution q and set M to 3
proposal_distr = scipy.stats.norm(loc=0.5, scale=1.3)
M = 3
```






```python
def sample(n):
    samples_generated = 0
    rejections = 0
    while samples_generated < n:
        x = proposal_distr.rvs(size=1)
        u = np.random.uniform(size=1)

        f = target_distr.pdf(x)
        g = proposal_distr.pdf(x)

        if u < f/(M*g):
            samples_generated += 1
            yield x
        else:
            rejections += 1

    acceptance_rate = samples_generated/(samples_generated+rejections)
    print(f"Acceptance rate: {acceptance_rate}")
```

```
Acceptance rate: 0.3359086328518643
```



Challenges with rejection sampling

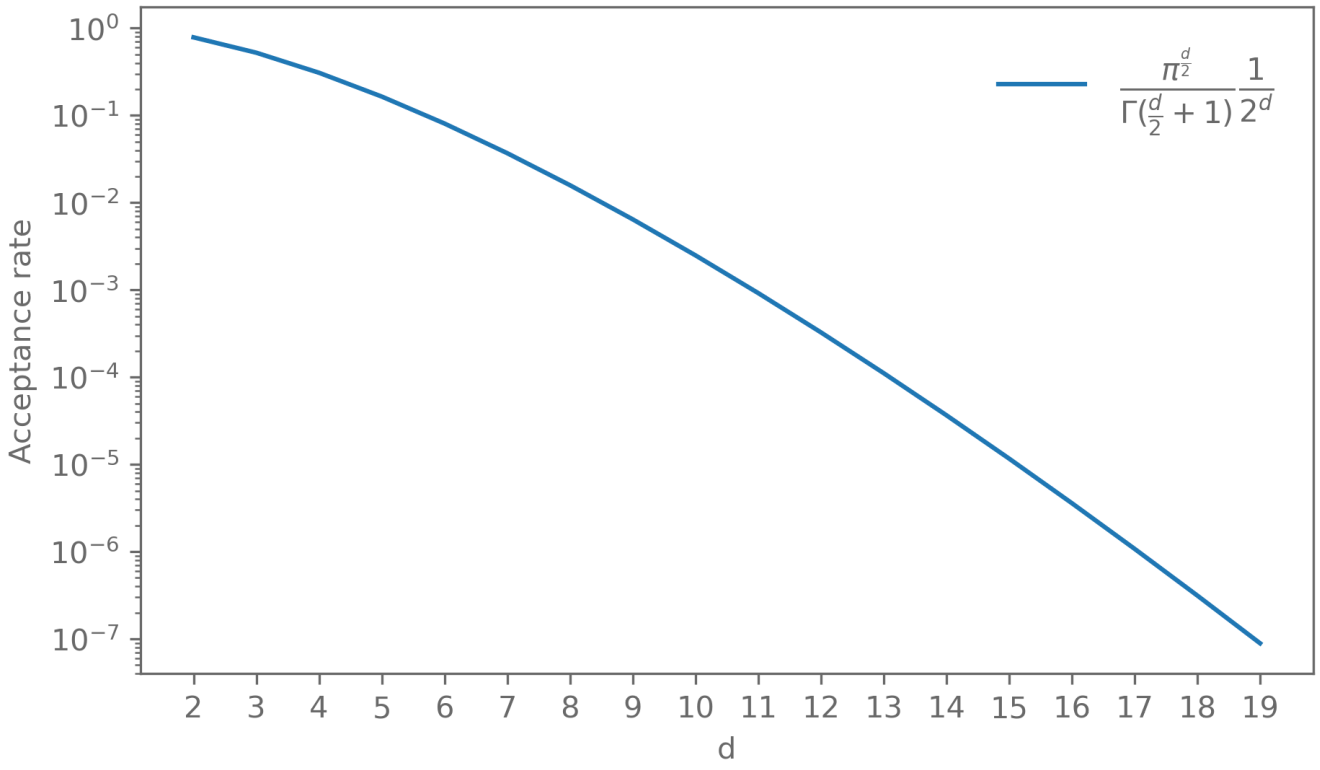
Finding a good proposal and M.

Curse of dimensionality: assume we want to sample uniformly from a disc of radius 1 and use uniform distribution on the square around the disc as the proposal distribution.

The acceptance rate in this case is $\frac{\text{area of disc}}{\text{area of square}} = \frac{\pi}{2^2} \approx 0.79$. Pretty good!

In d dimensions, the acceptance rate is

$$\frac{\text{volume of unit ball}}{\text{volume of hypercube}} = \frac{\pi^{\frac{d}{2}}}{\Gamma(\frac{d}{2}+1)} \frac{1}{2^d}$$



Exercise

Implement your own rejection sampling routine and test it on with different target and proposal distributions.

Markov chains Monte Carlo

Many of the Monte Carlo methods in use are build around the concept of Markov chains. Using such Markov chains to sample from a distribution is called Markov chain Monte Carlo (MCMC).

A Markov chain is a sequence of RVs $X_0, \ldots, X_t$ where the distribution of X_t only depends on X_{t-1}.

$$\Pr(X_t = x_t | X_0 = x_0, \ldots X_{t-1} = x_{t-1}) = \Pr(X_t = x_t | X_{t-1} = x_{t-1})$$

Knowing the states $X_0, \ldots, X_{t-2}$ in addition to X_{t-1} does not give provide more information.

The probability to transition from state y to x is given by the transition probability $q(x|y)$.

The transition probability respects detailed balance if

$$q(x|y)p(y) = q(y|x)p(x)$$

If q satisfies detailed balance, then p is a stationary distribution of the Markov chain. A stationary distribution is unchanged under the transition function:

$$p(x) = \sum_y q(x|y)p(y)$$

To show this

$$\sum_y q(x|y)p(y) = \sum_y q(y|x)p(x) \quad \text{(detailed balance)} \tag{1}$$

$$= p(x) \sum_y q(y|x) \tag{2}$$

$$= p(x) \tag{3}$$

This is the distribution we care about in MCMC: we can sample from $p(x)$ by creating a Markov chain using the transition probabilities $q(x|y)$, provided they satisfy detailed balance. We skipped over a lot of mathematical details and conditions here but this is the basic idea on how to sample from some distribution $p(x)$.

Metropolis-Hastings

Metropolis-Hastings is a classical MCMC algorithm. It works as follows: Given a distribution $p(x)$ we want to sample from, a proposal distribution $q(x|y)$, and a starting point $x_{t=0}$

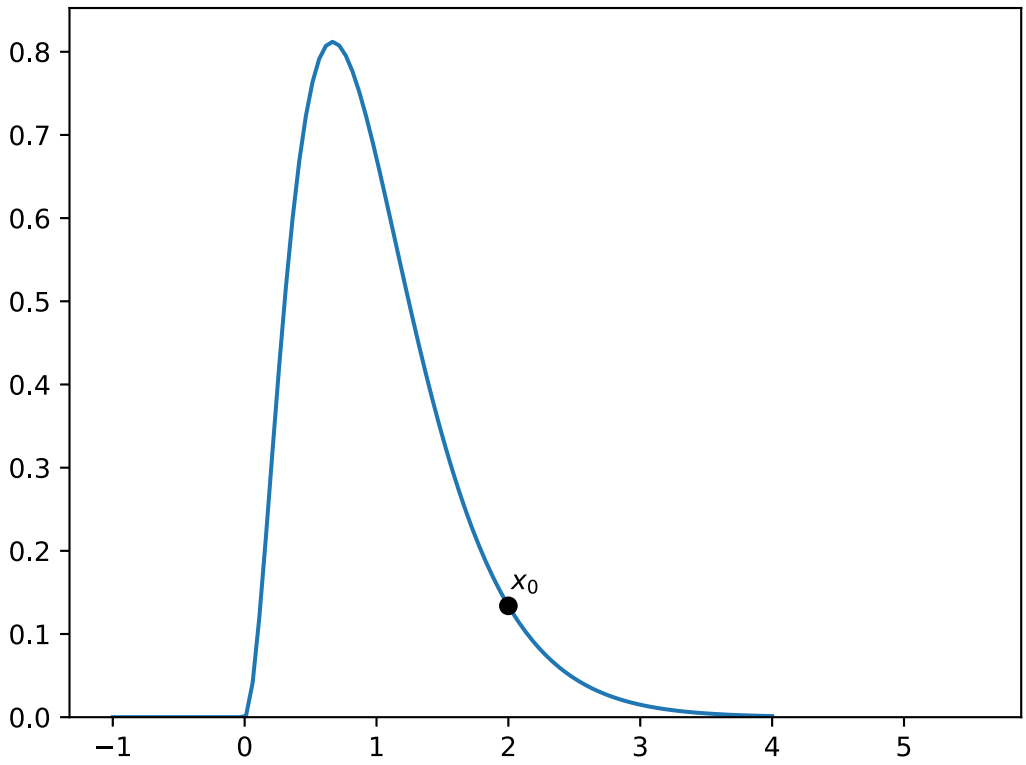
1. Sample a proposal x' from q: $x' \sim q(\cdot|x_t)$
2. Compute the quantity

$$a = \frac{p(x')q(x_t|x')}{p(x_t)q(x'|x_t)}$$

3. If $a \geq 1$, accept x'. If $a < 1$, accept x' with probability a:
   - If accepted: $x_{t+1} = x'$
   - If rejected: $x_{t+1} = x_t$

In the case where q is symmetric ($q(x|y) = q(y|x)$), $a = \frac{p(x')}{p(x_t)}$: if the proposed point has a higher probability than the previous point, accept it. Else, accept it with probability a.

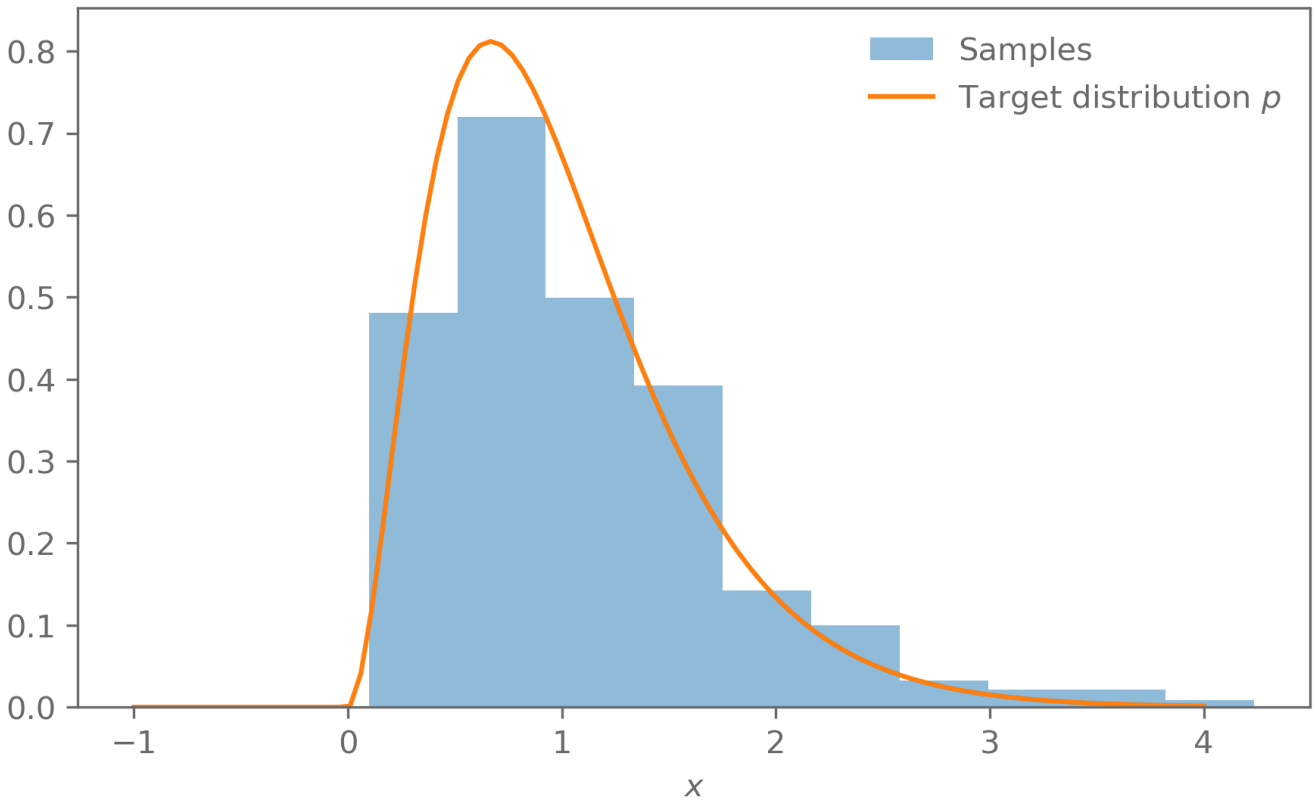
```
# We use a normal distribution with variance 1 as the proposal
proposal_distr = partial(scipy.stats.norm, scale=1)

def sample_transition(x0):
    return proposal_distr(loc=x0).rvs(size=1)

def transition_prob(x, y):
    # Q(x; y)
    return proposal_distr(loc=y).pdf(x)
```



Once Loop Reflect

```python
def sample(n, x0, target_distr, sample_transition, transition_prob):
    x0 = np.array([x0])
    for i in range(n):
        # Sample proposal
        x1 = sample_transition(x0)
        # Compute probabilities of the old and proposed states
        p0 = target_distr.pdf(x0)
        p1 = target_distr.pdf(x1)

        # Compute the transition probabilities
        q01 = transition_prob(x0, x1)
        q10 = transition_prob(x1, x0)

        a = p1/p0 * q01/q10

        u = np.random.uniform(size=1)
        if a >= u:
            # accept, proposed state becomes new state
            x0 = x1
            yield x1
        else:
```

```
            # reject, stay with current state
            yield x0
```



Challenges with Metropolis-Hastings

Metropolis-Hastings still requires a well-tuned proposal distribution to work well.

If the proposal is too broad, the acceptance rate goes down, because proposed points are likely in a low-probability part of the target distribution.

If the proposal is too narrow, Metropolis-Hasting becomes a random walk, which takes a long time to explore the full volume of the target distribution.

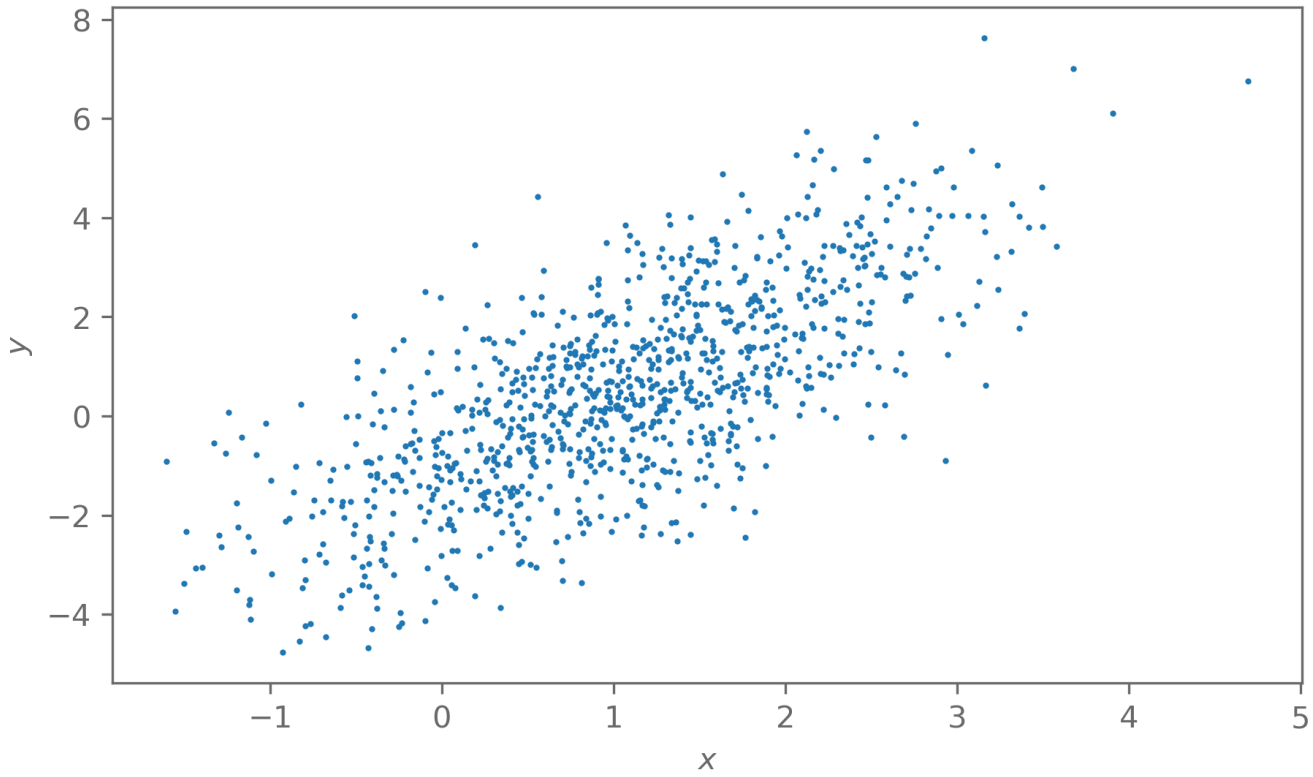
Having the proposal be as close to the target distribution is optimal but for that you need to know the target distribution first!

Exercise

- Implement Metropolis-Hastings for n dimensional distributions
  - Sample from a 2D Gaussian (code on the next slide)
  - Plot the samples in the chain. How do the samples depend on the starting position?
- Show that Metropolis-Hastings satisfies detailed balance
  - Hint: $T(x|y) = q(x|y)\min(1, a)$

```
# Define variances and correlation
```

```
sigma_x = 1
sigma_y = 2
rho = 0.7

# Define mean and covariance
mean = np.array([1, 0.5])
cov = np.array([[sigma_x**2, sigma_x*sigma_y*rho],
                [sigma_x*sigma_y*rho, sigma_y**2]])

# Create distribution object
bivariate_normal = scipy.stats.multivariate_normal(mean=mean, cov=cov)

# Sample 1000 points. Do this with your MCMC implementation instead!
samples = bivariate_normal.rvs(size=1000)
```



Slice sampling

Slice sampling is an other MCMC method and similar in that regard to Metropolis-Hastings.

Because it also samples the volume under the target distribution uniformly, it has some similarities to rejection sampling.

The advantage over MH is that is much less reliant on tuning the proposal.

Slice sampling proceeds as follows:

1. Sample u uniformly between 0 and $p(x_t)$: $y \sim \mathcal{U}(0, p(x_t))$
2. Find an interval $L < x_t < R$ such that $p(L) < y$ and $p(R) < y$

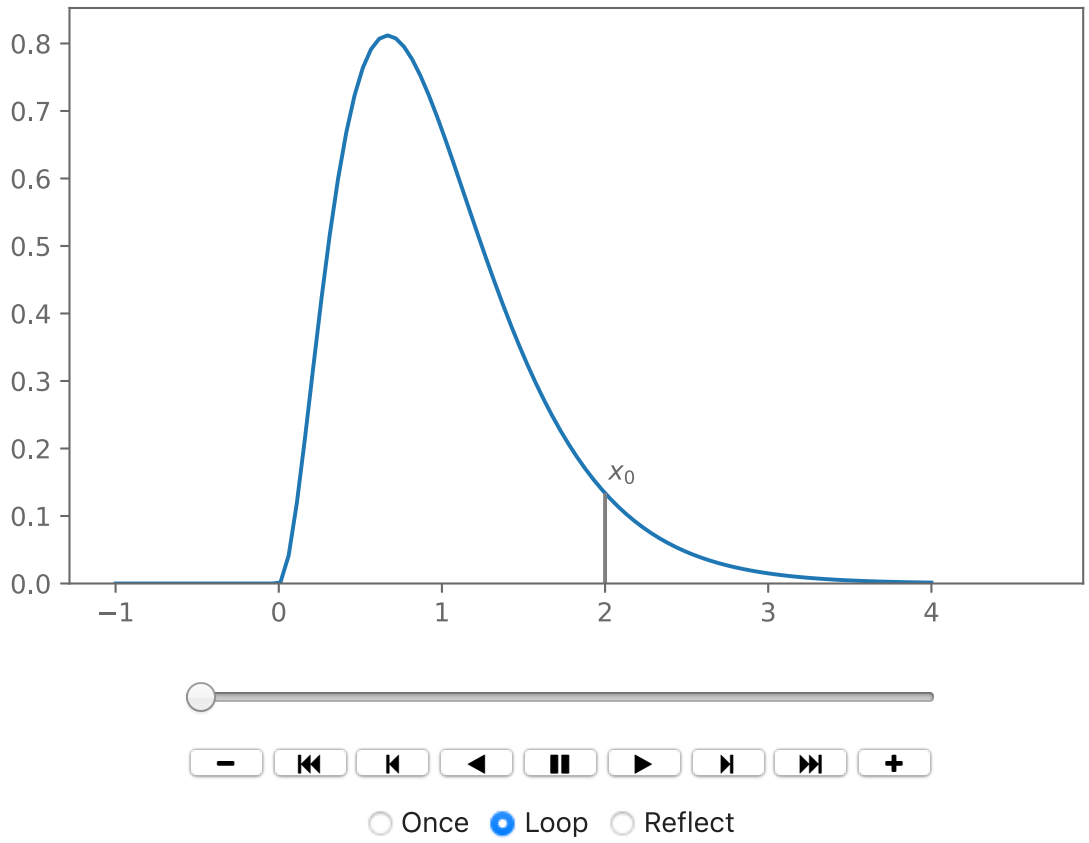
3. Draw x' uniformly from the interval $[L, R]$: $x' \sim \mathcal{U}(L, R)$
    - If $p(x') \leq u$, shrink the interval and return to 3.
    - If $p(x') > u$, the point (x', u) lies under the curve $p(x)$, so accept x': $x_{t+1} = c'$

- Finding the interval for step 2. uses a stepping out procedure:

  Given a step size w (this is the tuning parameter of slice sampling)

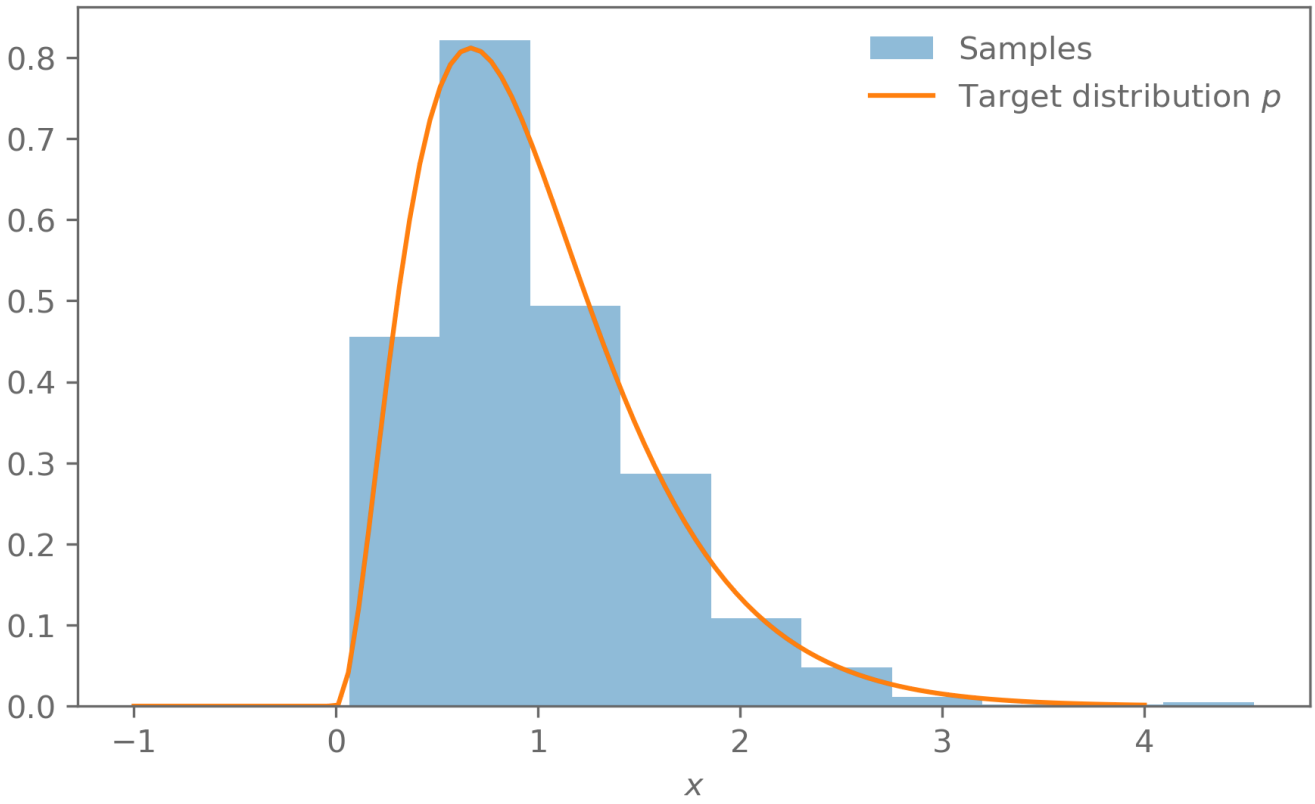
  1. Draw $r \sim \mathcal{U}(0, 1)$
  2. Set $L = x_t - rw$, $R = x_t + (1 - r)w$
  3. While $p(L) > u$: $L = L - w$
  4. While $p(R) > u$: $R = R + w$
- Shrinking the interval in step 3:

  1. While $p(x') \leq u$
     - If $x' > x_t$, $R = x'$
     - Else $L = x'$




```
def sample_slice_sampling(n, x0, target_distr, step):
    p0 = target_distr.pdf(x0)

    for _ in range(n):
        u = np.random.uniform(0, p0)
```

```
        # Define the initial interval
        w = np.random.uniform(0, 1)
        x_l, x_r = x0 - w*step, x0 + (1-w)*step

        def step_out(x, left=True):
            p = target_distr.pdf(x)
            while p > u:
                if left:
                    x -= step
                else:
                    x += step
                p = target_distr.pdf(x)
            return x

        # Step out until p(x_l) < u and p(x_r) < u
        x_l, x_r = step_out(x_l, left=True), step_out(x_r, left=False)
        while True:
            x1 = np.random.uniform(low=x_l, high=x_r)
            p1 = target_distr.pdf(x1)
            if p1 > u:
                # Accept the point x1
                break
            else:
                # Shrink the inverval
                if x1 >= x0:
                    x_r = x1
                else:
                    x_l = x1

        x0 = x1
        p0 = p1
        yield x0
```



Practical considerations for MCMC methods

The implementations shown here are the most barebones and simplest version of these methods. Implementing them yourselves is important to understand how these methods work and what some of the pitfalls are.

In a real-world application, with many parameters and complicated likelihoods, you probably want to use established implementations that use more sophisticated methods and are well-tested, instead of your own implementation.

Examples are emcee, zeus, and dynesty.

Because the state of a Markov chains depends on the previous state, the samples generated in MCMC are not independent. This has a few implications:

- The chain will take some time to move from the starting position to the bulk of the target distribution. This burn-in phase needs to be removed from the chain.
- If we use n samples from the chain to estimate a quantity of the distribution, for example the mean, then the variance of this estimate will not decrease as $\frac{1}{n}$, because the samples are correlated.

```
import emcee

emcee.autocorr.integrated_time(samples)
```

```
array([1.40694924])
```

Nested sampling

https://arxiv.org/abs/2205.15570

Nested sampling takes a very different approach to sampling than the MCMC methods covered so far. The main advantage is its ability to estimate the evidence. Remember Bayes' theorem

$$p(\theta|d) = \frac{p(d|\theta)p(\theta)}{p(d)}$$

To make the notation clearer (and consistent with some of the literature on nested sampling), write this as

$$p(\theta|d) = \frac{L(\theta)\pi(\theta)}{Z} \ ,$$

where $L(\theta) = p(d|\theta)$ is the likelihood, $\pi(\theta) = p(\theta)$ the prior, and $Z = p(d)$ the evidence or marginal likelihood.

Evaluting the evidence

$$Z = \int L(\theta)\pi(\theta)\mathrm{d}\theta$$

by naive integration is usually intractable for high-dimensional problems. To see this, imagine discretising the integral into 50 intervals: $Z = \sum_i^{50} L(\theta_i)\pi(\theta_i)\Delta\theta$. In 10 dimensions (which is not much as far as real-world applications are concerned), this would require $50^10 \approx 10^17$ evaluations of the likelihood.

The idea behind nested sampling is to rewrite the integral so that instead of integrating over θ, the integral is over levels of the likelihood. This somewhat like doing Lebesgue integration instead of Riemann integration.

$$Z = \int X(L)\mathrm{d}L = \int L(X)\mathrm{d}X \ ,$$

where $X(L^*)$ is the volume of the likelihood (weighted by the prior) above some likelihood level L^*:

$$X(L^*) = \int_{L(\theta)>L^*} \pi(\theta)\mathrm{d}\theta$$

The nested sampling algorithm works like this:

1. Sample n_{live} live points from the prior
2. At each iteration i, find the point with the lowest likelihood. This now becomes a dead point. We record its likelihood L_i^* and remove the dead point from our live points.

3. Sample a new point from the prior, with the constraint that $L(\theta) > L_i^*$
4. Estimate the volume X_i of the likelihood above L_i^*.
5. Estimate $Z = \sum_i L_i^* \Delta X_i$ and iterate from 2. until some convergence criterion on Z is reached.

How do we estimate the volumes X_i? The idea is similar to the Monte Carlo estimation of π at the beginning of the course: we sample an outer volume (the square) and count how many point end up inside the smaller inner volume (the quadrant of the circle). The ratio of the volumes is then approximated by the ratio of the points inside the inner volume over all the points.

In nested sampling, the inner volume is X_i, the volume of the likelihood where $L(\theta) > L_i^*$, has the n_{live} live points. The outer volume, the volume of the likelihood where $L(\theta) > L_{i-1}^*$ has $n_{\text{live}} + 1$ points. The n_{live} live points plus the recent dead point.

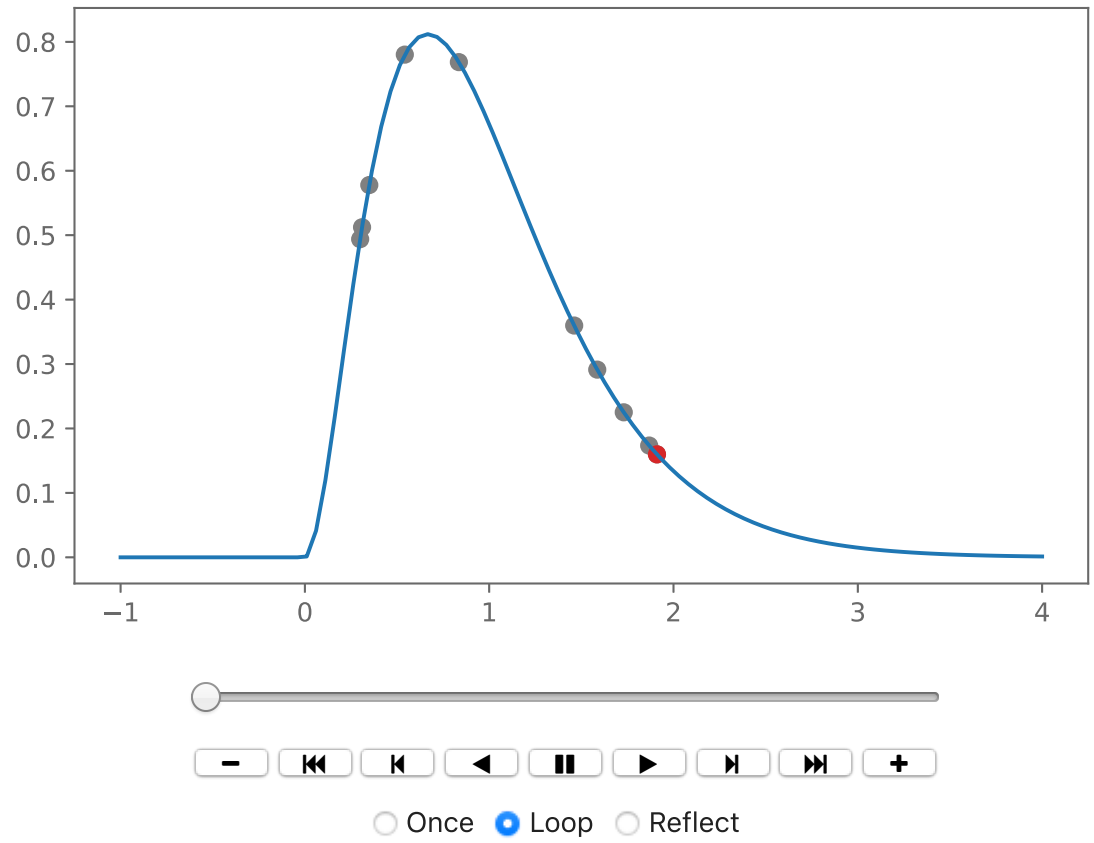
At each iteration, the volume $X(L)$ therefore decreases by a factor of approximately $t_i \approx \frac{n_{\text{live}}}{n_{\text{live}}+1}$.

The volume after i iterations is then $X_i = t_i X_{i-1} = t_i \dots t_1 X_0$, $X_0 = 1$.

The dead points sample the posterior, when weighted properly:

$$p_i = \frac{w_i L_i^*}{Z}, \quad w_i = \frac{1}{2}(X_{i-1} - X_{i+1})$$

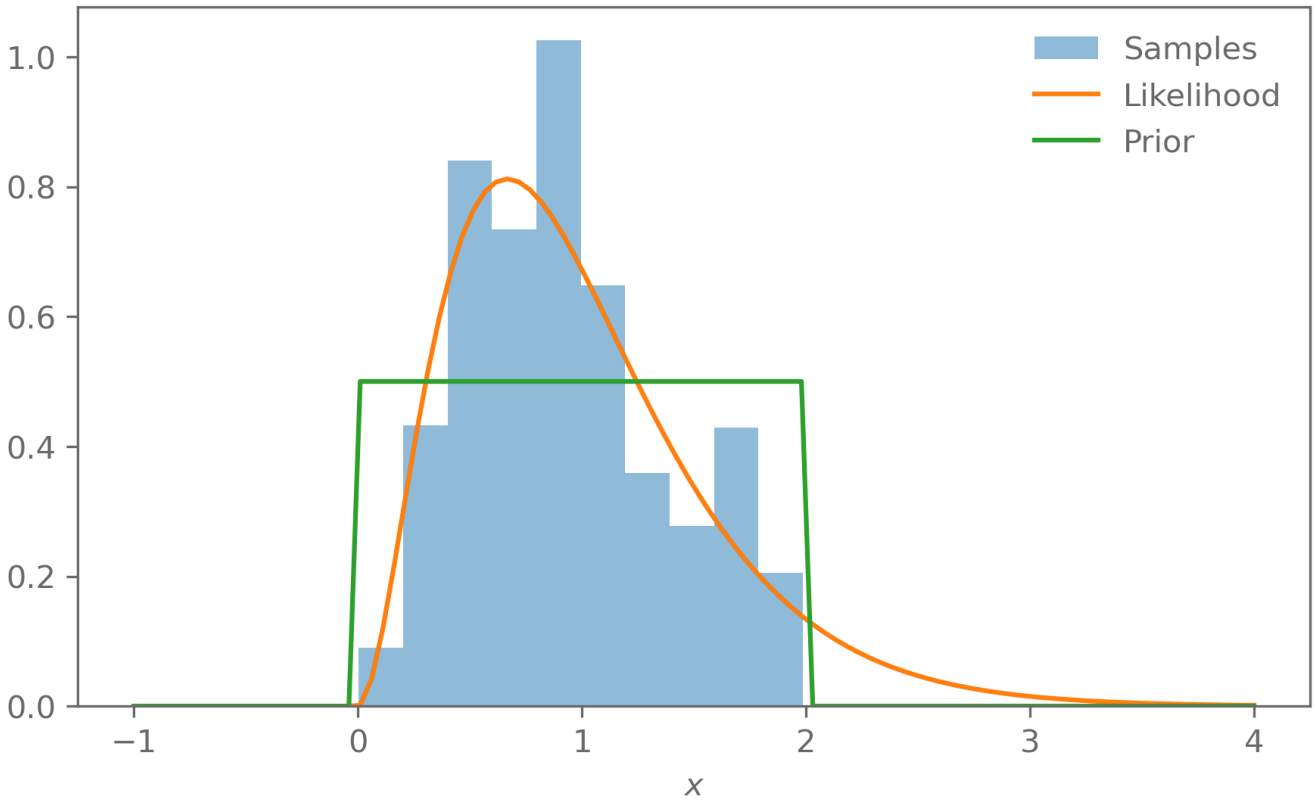
```
# We need to define separate likelihood and prior for nested sampling
log_likelihood = target_distr.logpdf
prior = scipy.stats.uniform(0., 2.)
```




```
from scipy.special import logsumexp
import tqdm

def sample(log_likelihood, prior, n_live, tol=0.01, n_max_iter=100000):
    live_points = prior.rvs(n_live)
    log_L = log_likelihood(live_points)
    if not np.all(np.isfinite(log_L)):
        raise ValueError("Non-finite log likelihood for some points.")

    log_tol = np.log(tol)

    log_X = [0,]

    dead_points = []
    dead_points_log_L = []

    n_eval = live_points.shape[0]
    drain_live_points = False
    i = 0
    progress = tqdm.tqdm()
    while i < n_max_iter:
        idx = np.argmin(log_L)
        log_L_star = log_L[idx]

        dead_points.append(live_points[idx])
        dead_points_log_L.append(log_L_star)
```

```
        log_t = -1/n_live
        log_X.append(log_X[-1] + log_t)

        if i > 4:
            X = np.exp(np.array(log_X))
            w = 0.5*(X[:-2]- X[2:])
            log_Z = logsumexp(np.array(dead_points_log_L[:-1]), b=w)
            log_mean_L = logsumexp(log_L, b=1/n_live)
            log_Delta_Z = log_mean_L + log_X[-1]
            if log_Delta_Z - log_Z < log_tol:
                drain_live_points = True
                live_points = np.delete(live_points, idx)
                log_L = np.delete(log_L, idx)
                if len(log_L) == 0:
                    break

            progress.set_postfix({"log_Z": log_Z, "n_eval": n_eval, "iter": i})

        while not drain_live_points:
            new_point = prior.rvs(1)
            log_L_new = log_likelihood(new_point)
            n_eval += 1
            if np.isfinite(log_L_new) and log_L_new > log_L_star:
                live_points[idx] = new_point
                log_L[idx] = log_L_new
                break

        i += 1

    dead_points = np.array(dead_points)
    dead_points_log_L = np.array(dead_points_log_L)
    n_sample = 100
    t_sample = scipy.stats.beta(n_live, 1).rvs((n_sample, len(dead_points_log_L)))
    log_X_sample = np.insert(np.cumsum(np.log(t_sample), axis=1), 0, 0, axis=1)
    X_sample = np.exp(log_X_sample)
    w_sample = 0.5*(X_sample[:, :-2]- X_sample[:, 2:])
    log_Z = scipy.special.logsumexp(dead_points_log_L[:-1], b=w_sample, axis=1)

    return log_Z, dead_points, w*np.exp(dead_points_log_L)[:-1]


log_Z, dead_points, weights = sample(
    log_likelihood=log_likelihood, prior=prior,
    n_live=100, tol=0.01, n_max_iter=10000)

import scipy.integrate
log_Z_exact = np.log(scipy.integrate.quad(lambda x: target_distr.pdf(x)*prior.pdf(x),

print(f"Exact log Z: {log_Z_exact:.2f}")
print(f"Nested sampling estimate of log Z: {np.mean(log_Z):.2f}±{np.std(log_Z):.2f}")
```

```
0it [00:02, ?it/s, log_Z=-.783, n_eval=14976, iter=616]
Exact log Z: -0.76
Nested sampling estimate of log Z: -0.77±0.04
```



Challenges with nested sampling

The big challenge in implementing nested sampling in practice is sampling from the prior with a likelihood constraint.

If all you care about is posterior samples, then nested sampling can be quite inefficient. Its strength is really the estimation of the evidence, which is important for model comparison.

The evidence is sensitive to the prior volume:

$$Z = \int L(\theta)\pi(\theta)\mathrm{d}\theta$$

Let us assume we have a uniform prior over some volume V: $\pi(\theta) \propto \frac{1}{V}$.

If the likelihood is much more constraining than the prior, the posterior does not change when we change the size of the prior.

But the evidence scales with the prior volume: $Z \propto \frac{1}{V}$. When comparing models, some care must be taken as not to be affected by prior volumes.