Probabilistic Programming Languages

Guillaume Baudart

MPRI 2023-202

MCMC Metropolis-Hastings

Probabilistic Programming Languages

Markov Chain Monte Carlo (MCMC)

Main idea
- Create a Markov chain that converge to the posterior distribution
- Iterate the process until convergence
- Generate samples to approximate the distribution

Pros
- Faster convergence
- Better results for high-dimensional models
- Advanced state-of-the-art optimizations (e.g., HMC, NUTS).

Cons
- Convergences?
- Traps: multimodal, funnel
- Samples correlation

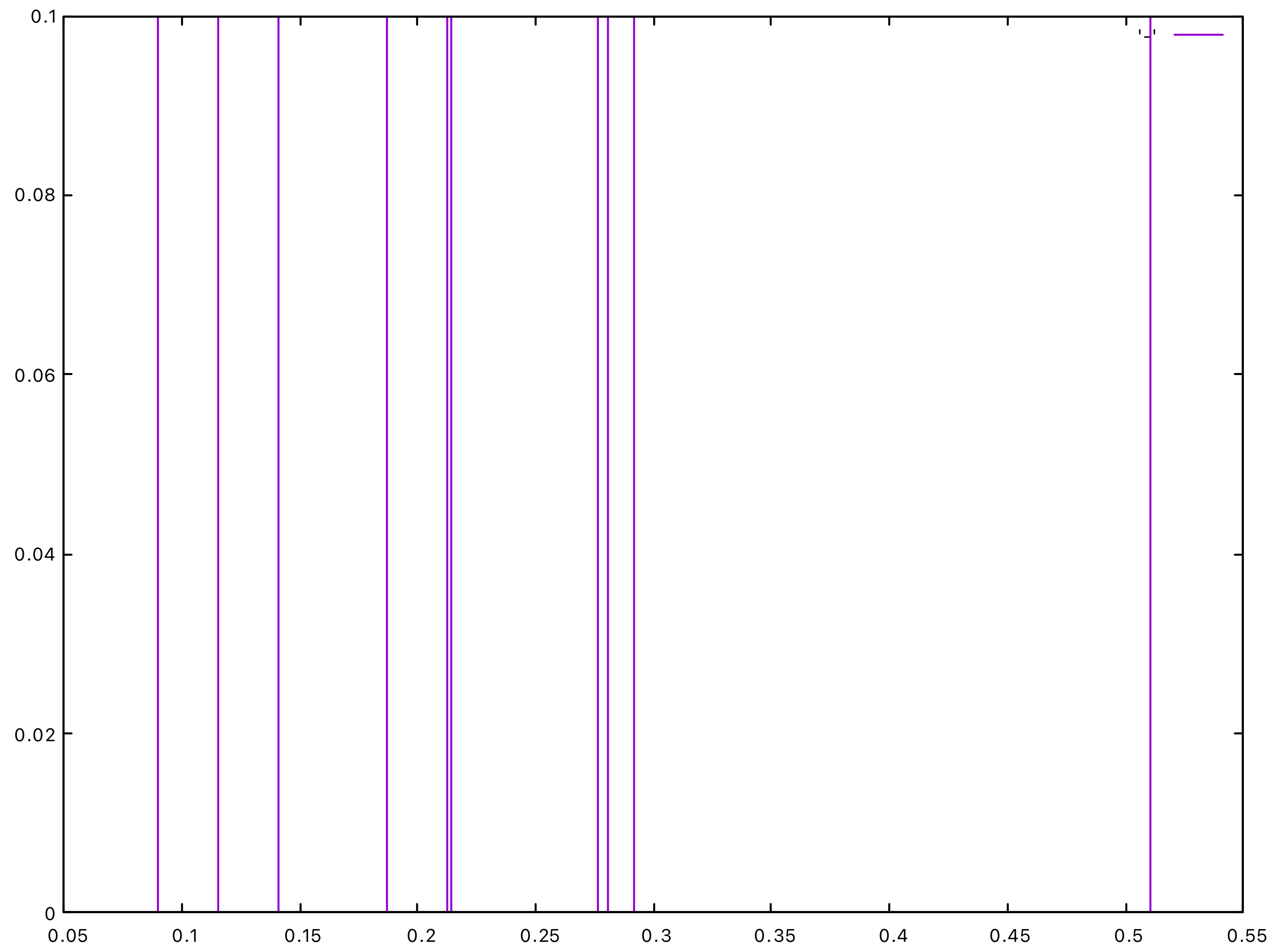
Reminder: rejection sampling

coin.ml

```
let coin prob data =
  let z = sample prob (uniform ~a:0. ~b:1.) in
  List.iter (observe prob (bernoulli ~p:z)) data;
  z

let _ =
  let d = infer coin [ 1; 1; 0; 0; 0; 0; 0; 0; 0; 0 ] in
  plot d
```

10 particles



Executing the model generates one sample

- `sample`: draw from a distribution
- `assume`/`observe`: hard conditioning, reject invalid samples
- Terminates with n valid samples

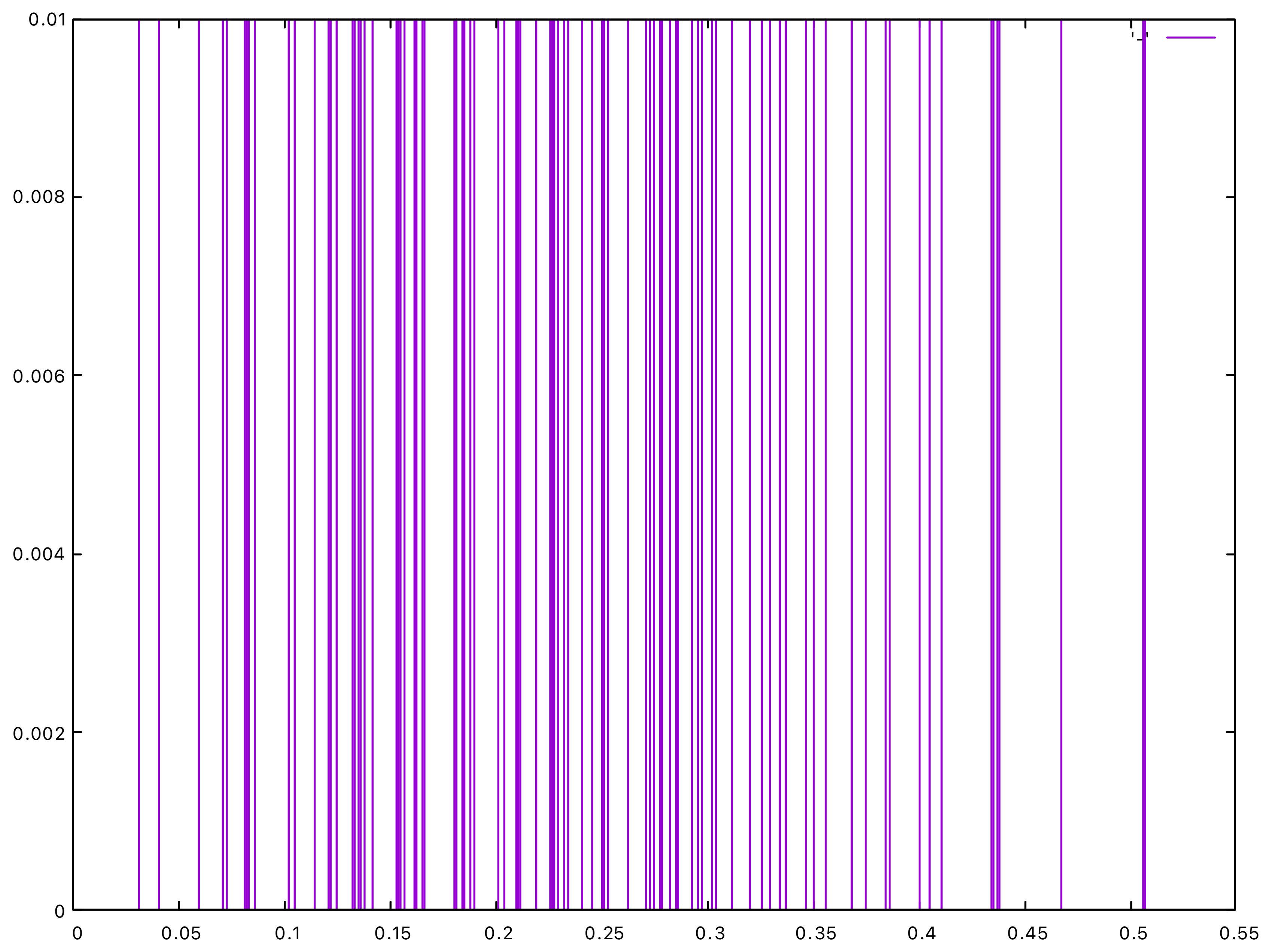
Reminder: rejection sampling

coin.ml

```
let coin prob data =
  let z = sample prob (uniform ~a:0. ~b:1.) in
  List.iter (observe prob (bernoulli ~p:z)) data;
  z

let _ =
  let d = infer coin [ 1; 1; 0; 0; 0; 0; 0; 0; 0; 0 ] in
  plot d
```

100 particles



Executing the model generates one sample

- `sample`: draw from a distribution
- `assume`/`observe`: hard conditioning, reject invalid samples
- Terminates with n valid samples

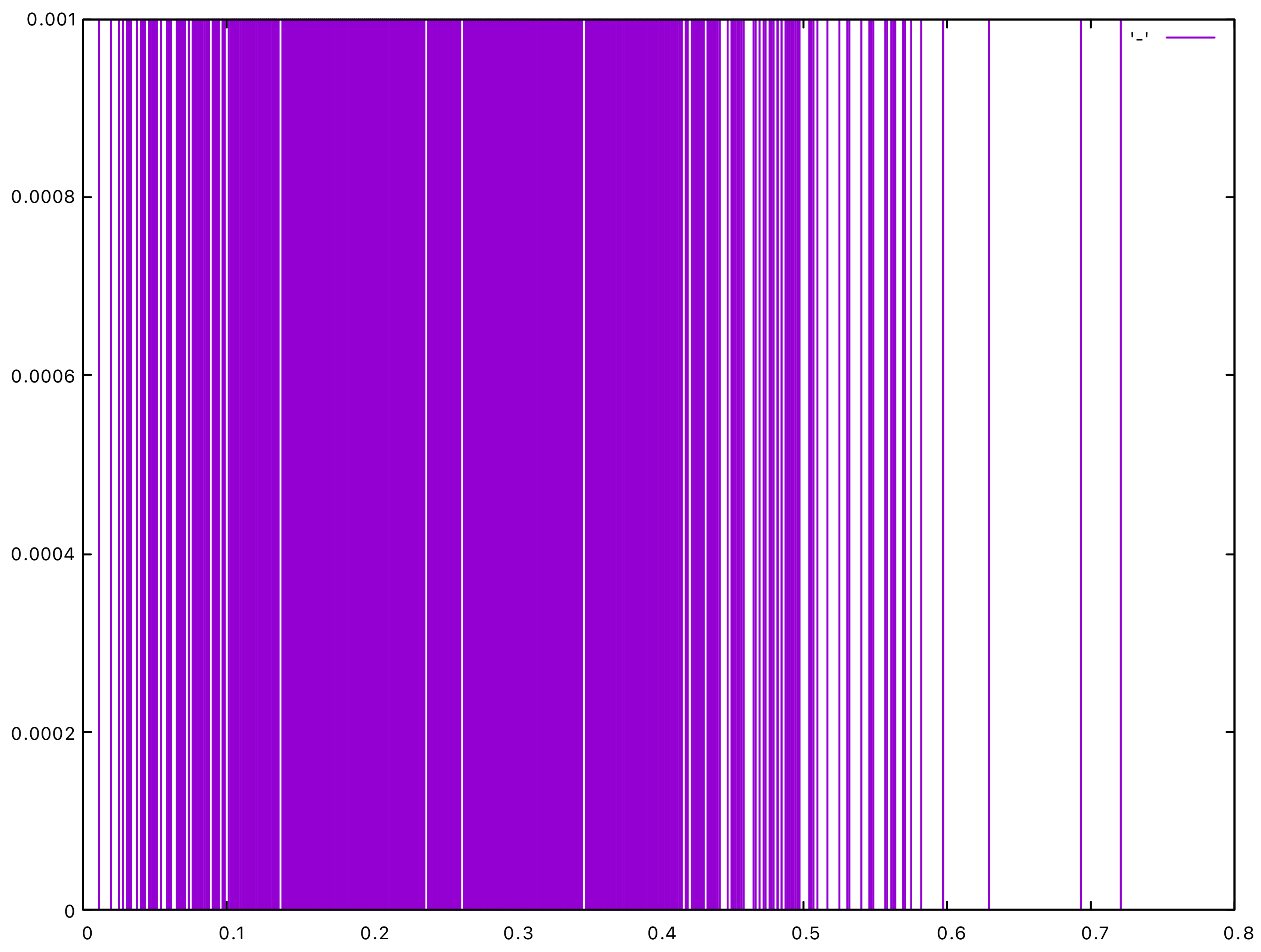
Reminder: rejection sampling

coin.ml

```
let coin prob data =
  let z = sample prob (uniform ~a:0. ~b:1.) in
  List.iter (observe prob (bernoulli ~p:z)) data;
  z

let _ =
  let d = infer coin [ 1; 1; 0; 0; 0; 0; 0; 0; 0; 0 ] in
  plot d
```

1000 particles



Executing the model generates one sample

- `sample`: draw from a distribution
- `assume`/`observe`: hard conditioning, reject invalid samples
- Terminates with n valid samples

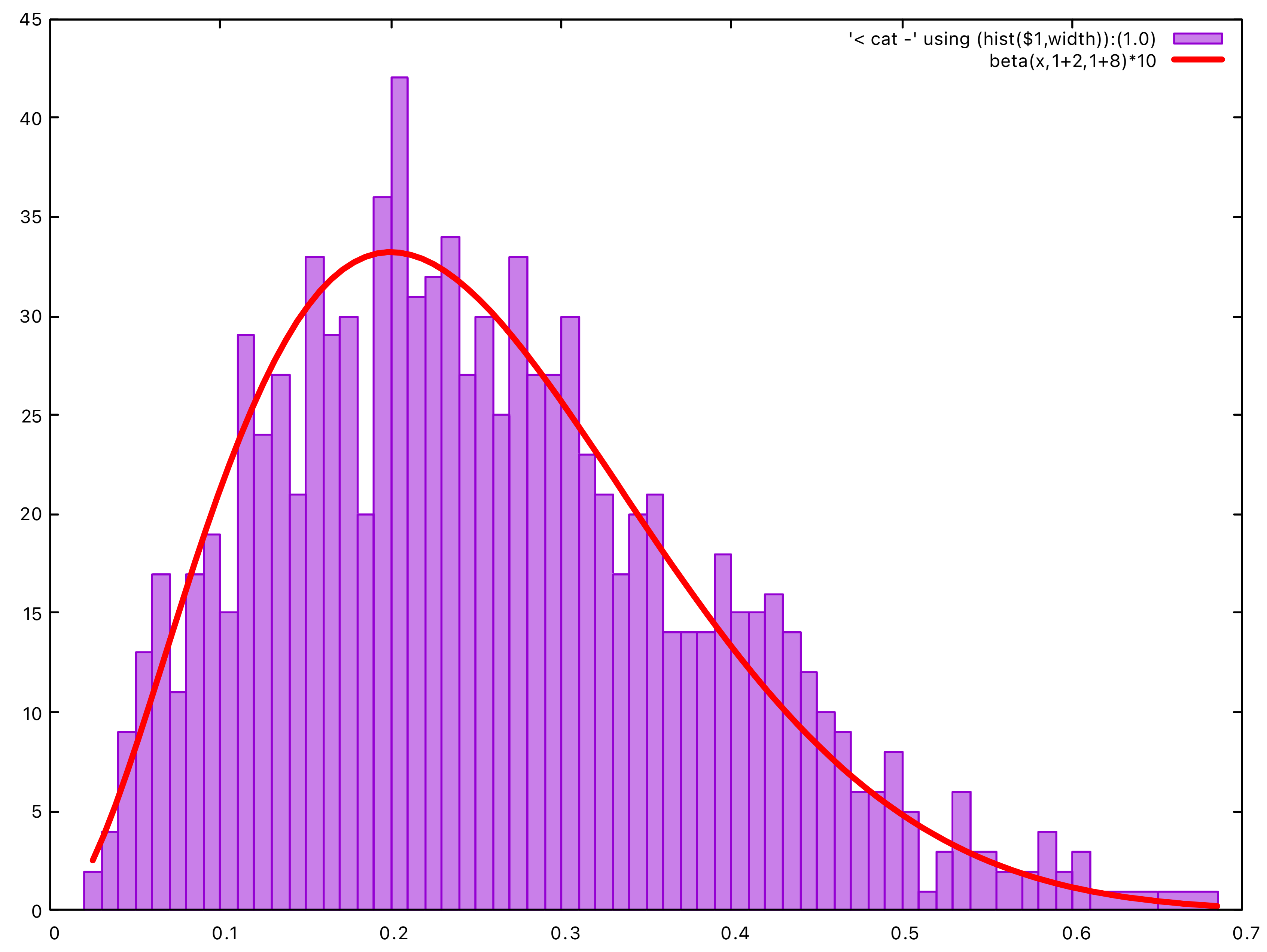
Reminder: rejection sampling

coin.ml

```
let coin prob data =
  let z = sample prob (uniform ~a:0. ~b:1.) in
  List.iter (observe prob (bernoulli ~p:z)) data;
  z

let _ =
  let d = infer coin [ 1; 1; 0; 0; 0; 0; 0; 0; 0; 0 ] in
  plot d
```

1000 particles



Executing the model generates one sample

- `sample`: draw from a distribution
- `assume`/`observe`: hard conditioning, reject invalid samples
- Terminates with n valid samples

Weighted rejection sampling

Adapt rejection sampling to soft conditioning

- Execute the sampler to get a pair (v_i, w_i)
- Suppose w_{max} is known
- Accept the sample with probability w_i / w_{max} or retry

But $w_{\max}$ is not known...

Execution trace

Consider a program execution with

- $X_i = x_0, \ldots, x_n$: set of random variables sampled at step i, i.e., the trace
- $Y_i = y_0, \ldots, y_m$: set of random variables observed at step i.

Remarks

- Sets X_i and Y_i depend on the execution path
- We can only control X_i

```
let bimodal y =
  let z = sample (bernoulli ~p:0.5) in
  let mu =
    if z then sample (gaussian ~mu:-1. ~sigma:1.)
    else sample (gaussian ~mu:1. ~sigma:1.)
  in
  let () = observe (gaussian ~mu ~sigma:1.) y in
  z
```

Multi-sites Metropolis Hastings

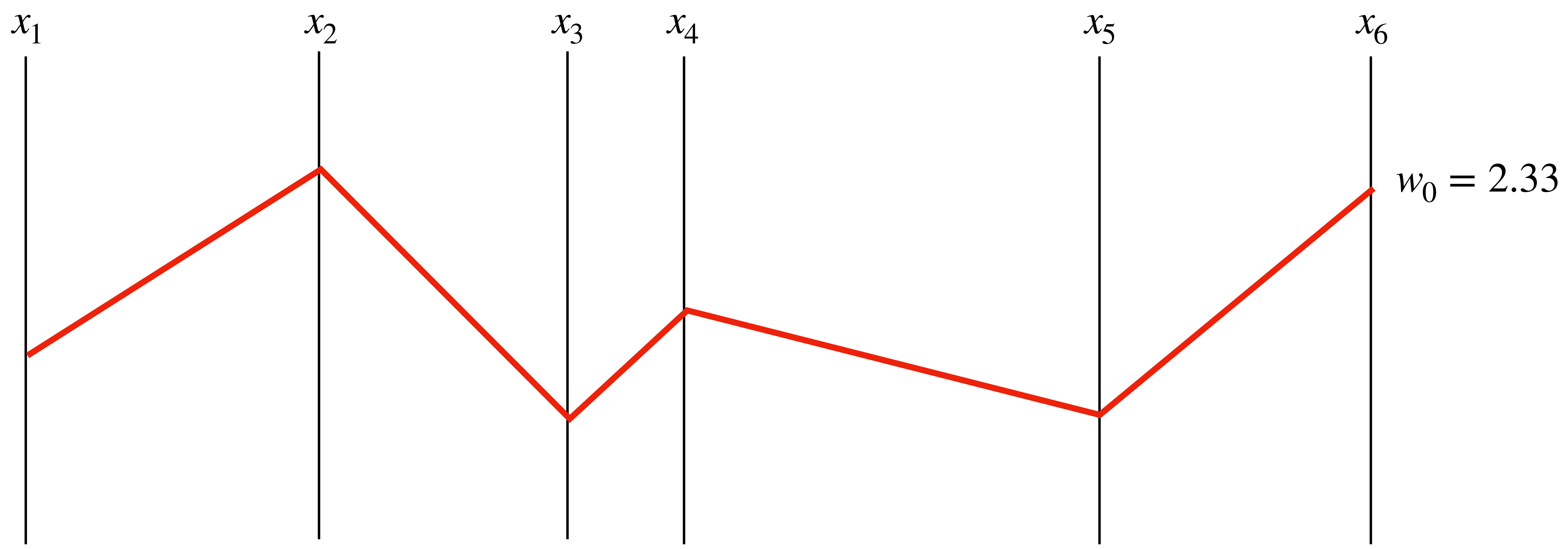
Markov chain on execution traces

- Execute the sampler to get a trace and associated score (X_i, y_i)
- If $w_i \geq w_{i-1}$ accept the trace (and the associated output)
- Else accept the trace with probability w_i / w_{i-1}
- Otherwise return the previous trace X_{i-1}

Multi-sites Metropolis Hastings

Markov chain on execution traces

- Execute the sampler to get a trace and associated score (X_i, y_i)
- If $w_i \geq w_{i-1}$ accept the trace (and the associated output)
- Else accept the trace with probability w_i / w_{i-1}
- Otherwise return the previous trace X_{i-1}



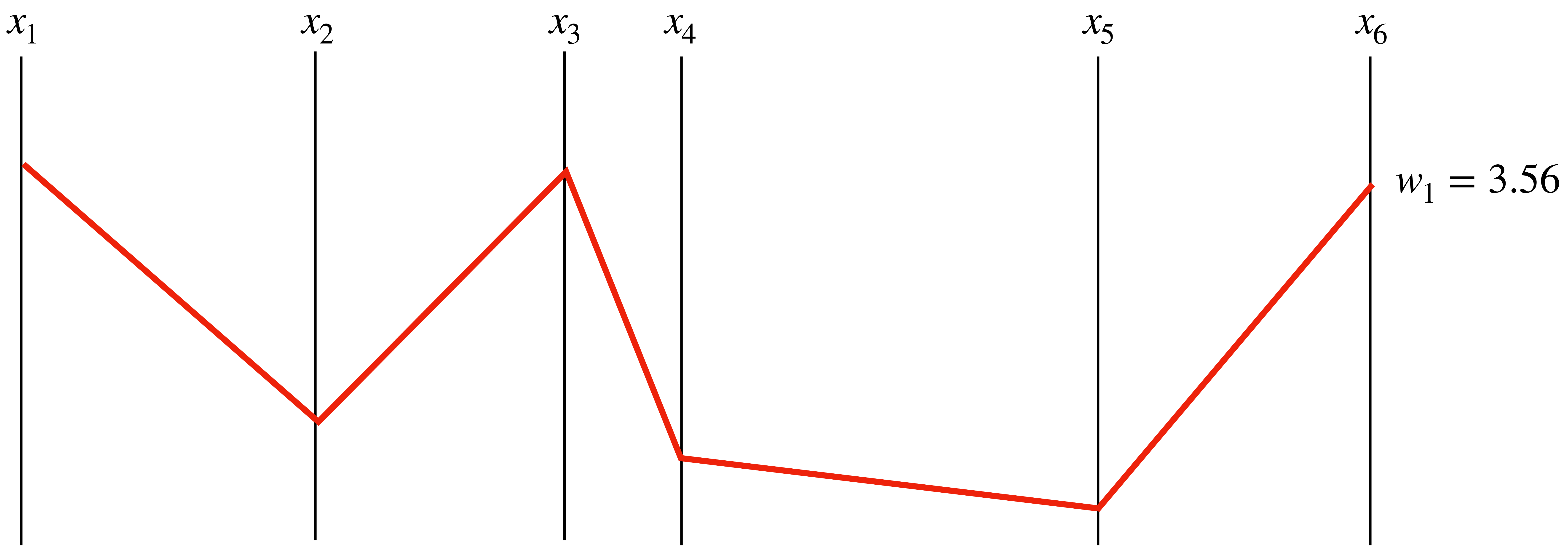
outputs

v_0

Multi-sites Metropolis Hastings

Markov chain on execution traces

- Execute the sampler to get a trace and associated score (X_i, y_i)
- If $w_i \geq w_{i-1}$ accept the trace (and the associated output)
- Else accept the trace with probability w_i / w_{i-1}
- Otherwise return the previous trace X_{i-1}



outputs

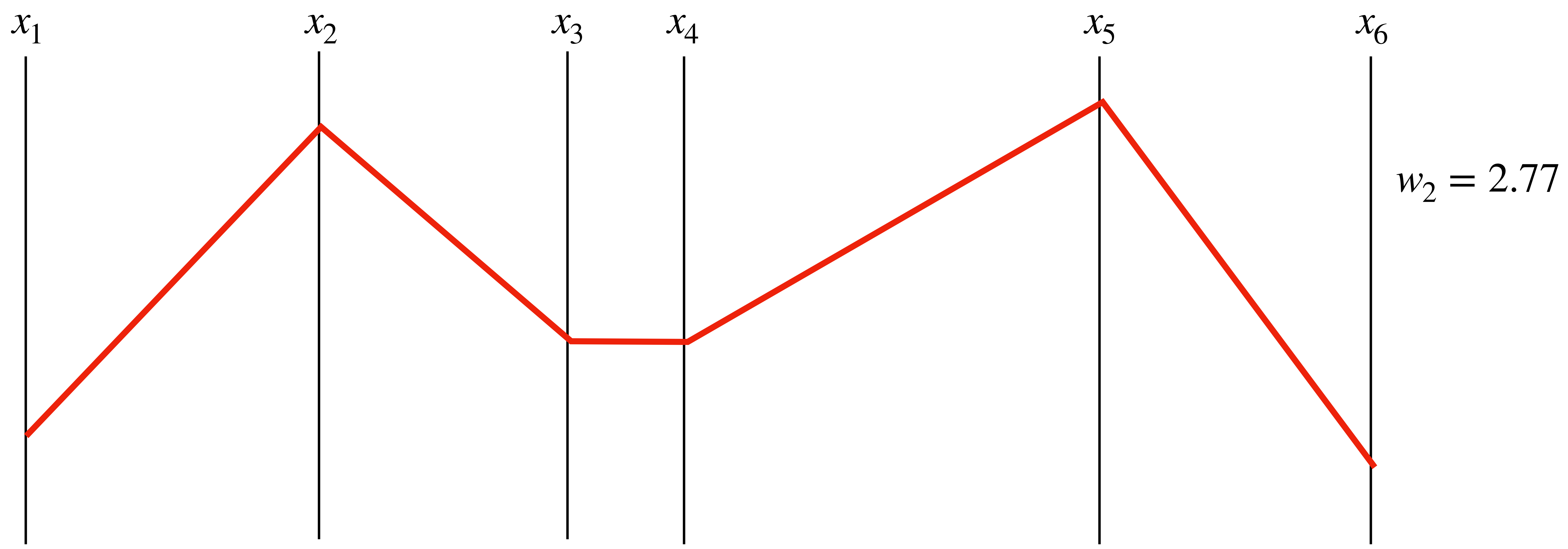
v_0

v_1

Multi-sites Metropolis Hastings

Markov chain on execution traces

- Execute the sampler to get a trace and associated score (X_i, y_i)
- If $w_i \geq w_{i-1}$ accept the trace (and the associated output)
- Else accept the trace with probability w_i / w_{i-1}
- Otherwise return the previous trace X_{i-1}



outputs

v_0

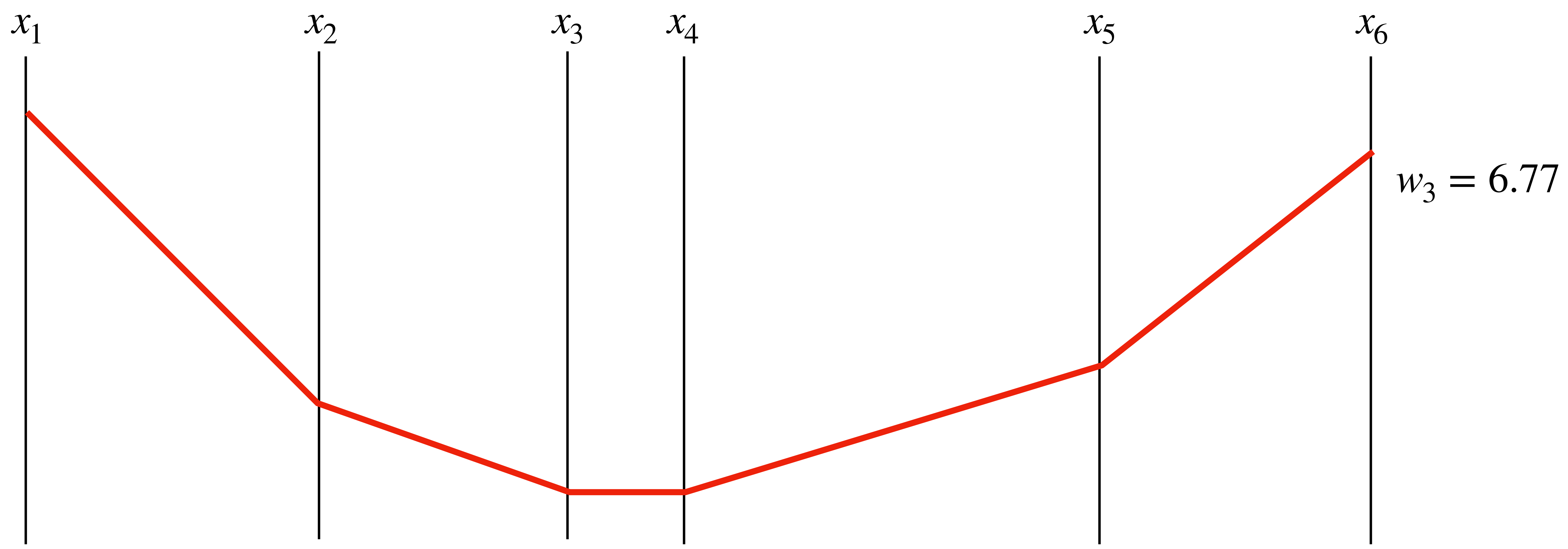
v_1

v_1

Multi-sites Metropolis Hastings

Markov chain on execution traces

- Execute the sampler to get a trace and associated score (X_i, y_i)
- If $w_i \geq w_{i-1}$ accept the trace (and the associated output)
- Else accept the trace with probability w_i / w_{i-1}
- Otherwise return the previous trace X_{i-1}



outputs
v_0
v_1
v_1
v_3
...

General Metropolis Hastings

More generally

- $X_i = x_0, \dots x_n$: set of random variables sampled at step i, i.e., the trace
- $Y_i = y_0, \dots, y_m$: set of random variables observed at step i
- Propose a new trace from a proposal distribution $q(X_i \mid X_{i-1})$
- Accept the trace X_i with probability α, where

$$\alpha = \text{min}\left(1, \frac{p(X_i, Y_i)}{p(X_{i-1}, Y_{i-1})} \frac{q(X_{i-1} \mid X_i)}{q(X_i \mid X_{i-1})}\right)$$

- Otherwise return the previous trace X_{i-1}

Multi-sites Metropolis Hastings: acceptation

- Draw proposal from priors $q(X_i \mid X_{i-1}) = p(X_i)$
- Resample all variables in X_i at each iteration

$$
\begin{aligned}
\frac{p(X_i, Y_i)}{p(X_{i-1}, Y_{i-1})} \frac{q(X_{i-1} \mid X_i)}{q(X_i \mid X_{i-1})} &= \frac{p(Y_i|X_i)\, p(X_i)}{p(Y_{i-1} \mid X_{i-1})\, p(X_{i-1})} \frac{q(X_{i-1} \mid X_i)}{q(X_i \mid X_{i-1})} \\
&= \frac{p(Y_i|X_i)\, p(X_i)}{p(Y_{i-1} \mid X_{i-1})\, p(X_{i-1})} \frac{p(X_{i-1})}{p(X_i)} \\
&= \frac{p(Y_i|X_i)}{p(Y_{i-1} \mid X_{i-1})} \\
&= \frac{w_i}{w_{i-1}}
\end{aligned}
$$

Markov chain on execution traces

- Execute the sampler to get a trace and associated score (X_i, y_i)
- If $w_i \geq w_{i-1}$ accept the trace (and the associated output)
- Else accept the trace with probability w_i / w_{i-1}
- Otherwise return the previous trace X_{i-1}

Single-site Metropolis Hastings

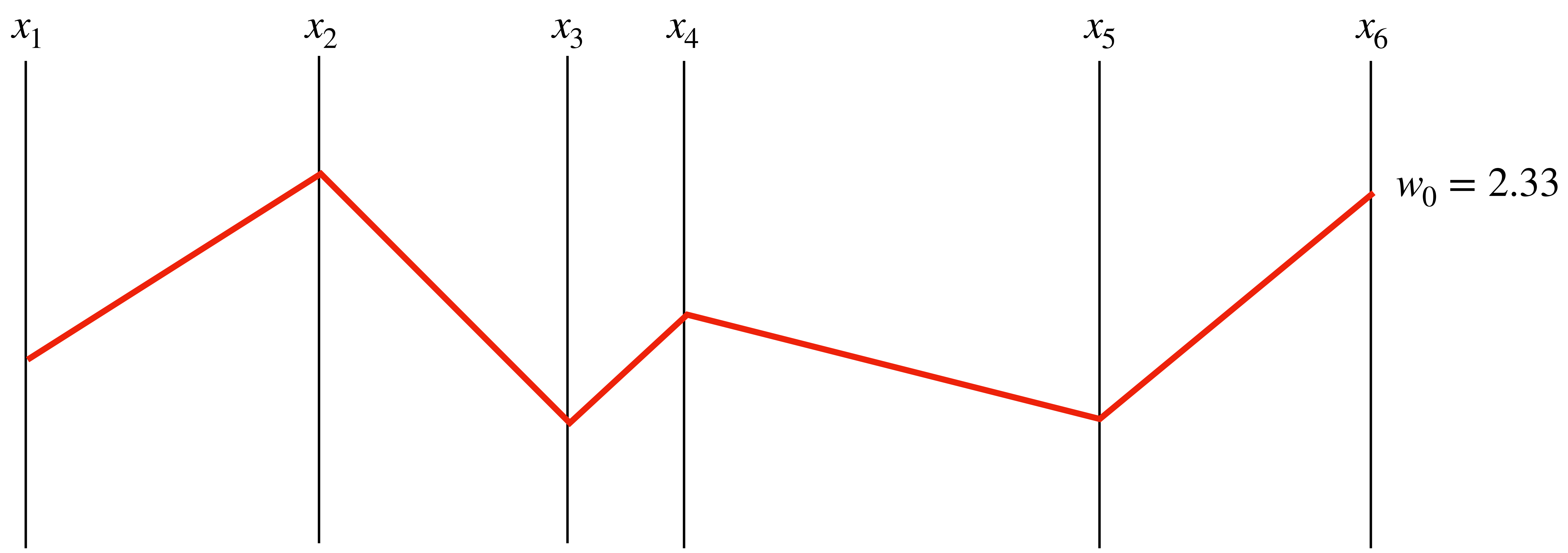
Reuse most of the previous trace (i.e., sampled values)

- Choose one random variable to resample to obtain a new execution
- Accept the trace with probability α
- Otherwise use the previous trace

Single-site Metropolis Hastings

Reuse most of the previous trace (i.e., sampled values)

- Choose one random variable to resample to obtain a new execution
- Accept the trace with probability α
- Otherwise use the previous trace



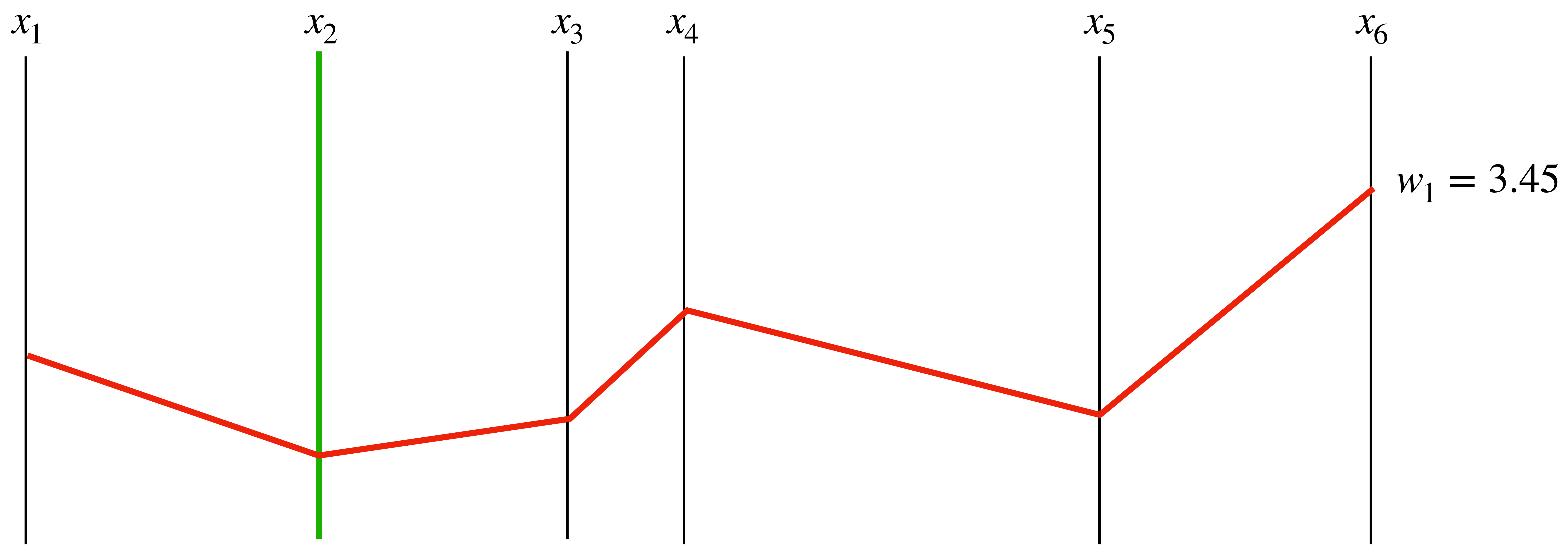
samples

v_0

Single-site Metropolis Hastings

Reuse most of the previous trace (i.e., sampled values)

- Choose one random variable to resample to obtain a new execution
- Accept the trace with probability α
- Otherwise use the previous trace



samples

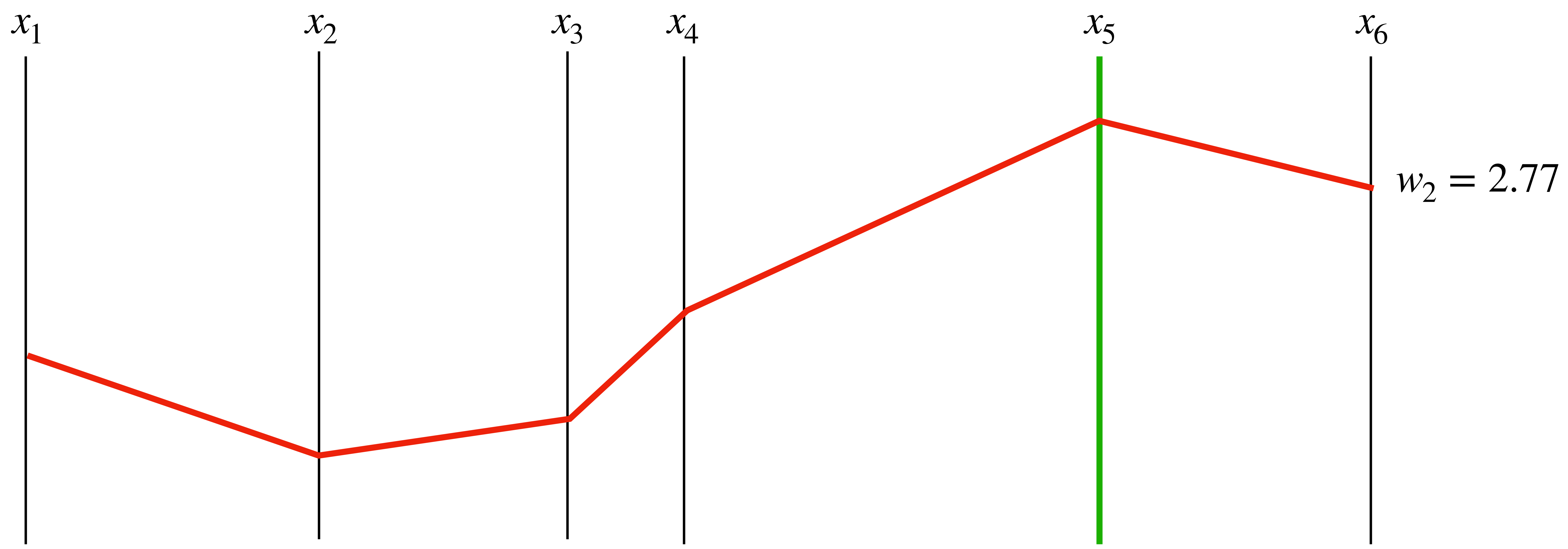
v_0

v_1

Single-site Metropolis Hastings

Reuse most of the previous trace (i.e., sampled values)

- Choose one random variable to resample to obtain a new execution
- Accept the trace with probability α
- Otherwise use the previous trace



samples

v_0

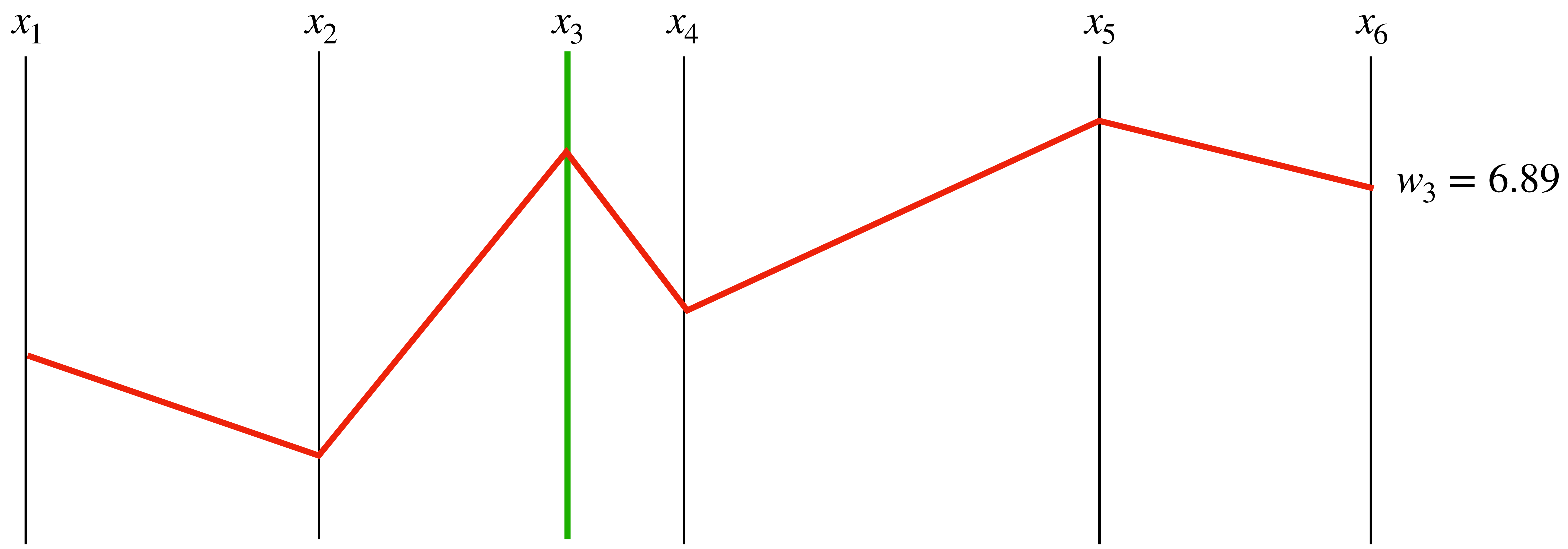
v_1

v_1

Single-site Metropolis Hastings

Reuse most of the previous trace (i.e., sampled values)

- Choose one random variable to resample to obtain a new execution
- Accept the trace with probability α
- Otherwise use the previous trace



samples

v_0

v_1

v_1

v_3

Single-site Metropolis Hastings: acceptation

Track the likelihood of all random variables during execution

- x = `sample`$(d) \to w(x) = (x, \text{pdf(d)(x)})$
- `observe`$(d, y) \to w(y) = (y, \text{pdf(d)(y)})$

score (as in importance sampling)

```
let bimodal y =
  let z = sample (bernoulli ~p:0.5) in
  let mu =
    if z then sample (gaussian ~mu:-1. ~sigma:1.)
    else sample (gaussian ~mu:1. ~sigma:1.)
  in
  let () = observe (gaussian ~mu ~sigma:1.) y in
  z
```

$w(z) = (0,\ 0.5)$

$w(\mu) = (1.2,\ 0.40)$

$w(y) = (2.0,\ 0.27)$

Single-site Metropolis Hastings: acceptation

- Pick one variable x_0 at random in the trace
- Resample only this variable and all its dependencies $X = X^{\text{sample}} \cup X^{\text{reuse}}$

$$\frac{p(X_i, Y_i)}{p(X_{i-1}, Y_{i-1})} \frac{q(X_{i-1} \mid X_i)}{q(X_i \mid X_{i-1})} = \frac{p(X_i, Y_i)}{p(X_{i-1}, Y_{i-1})} \frac{q(X_{i-1} \mid X_i, x_0)\, q(x_0 \mid X_i)}{q(X_i \mid X_{i-1}, x_0)\, q(x_0 \mid X_{i-1})}$$

Single-site Metropolis Hastings: acceptation

- Pick one variable x_0 at random in the trace
- Resample only this variable and all its dependencies $X = X^{\text{sample}} \cup X^{\text{reuse}}$

$$\frac{p(X_i, Y_i)}{p(X_{i-1}, Y_{i-1})} \frac{q(X_{i-1} \mid X_i)}{q(X_i \mid X_{i-1})} = \frac{p(X_i, Y_i)}{p(X_{i-1}, Y_{i-1})} \frac{q(X_{i-1} \mid X_i, x_0)\, q(x_0 \mid X_i)}{q(X_i \mid X_{i-1}, x_0)\, q(x_0 \mid X_{i-1})}$$

$$p(X, Y) = \prod_{x \in X} w(x) \prod_{y \in Y} w(y)$$

(not necessarily independent)

$$q(x_0 \mid X) = \frac{1}{|X|}$$

1/(n choices)

$$q(X \mid X', x_0) = \prod_{x \in X^{\text{sample}}} w(x)$$

resampled from the prior

Single-site Metropolis Hastings: acceptation

- Pick one variable x_0 at random in the trace
- Resample only this variable and all its dependencies $X = X^{\text{sample}} \cup X^{\text{reuse}}$

$$p(X, Y) = \prod_{x \in X} w(x) \prod_{y \in Y} w(y)$$

$$q(x_0 \mid X) = \frac{1}{|X|}$$

$$q(X \mid X', x_0) = \prod_{x \in X^{\text{sample}}} w(x)$$

$$\frac{p(X_i, Y_i)}{p(X_{i-1}, Y_{i-1})} \frac{q(X_{i-1} \mid X_i)}{q(X_i \mid X_{i-1})} = \frac{p(X_i, Y_i)}{p(X_{i-1}, Y_{i-1})} \frac{q(X_{i-1} \mid X_i, x_0)\, q(x_0 \mid X_i)}{q(X_i \mid X_{i-1}, x_0)\, q(x_0 \mid X_{i-1})}$$

$$= \frac{q(x_0 \mid X_i)}{q(x_0 \mid X_{i-1})} \frac{p(X_i, Y_i)}{q(X_i \mid X_{i-1}, x_0)} \frac{q(X_{i-1} \mid X_i, x_0)}{p(X_{i-1}, Y_{i-1})}$$

$$= \frac{|X_{i-1}|}{|X_i|} \frac{\prod_{x \in X_i} w(x)\ \prod_{y \in Y_i} w(y)}{\prod_{x \in X_i^{\text{sample}}} w(x)} \frac{\prod_{x \in X_{i-1}^{\text{sample}}} w(x)}{\prod_{x \in X_{i-1}} w(x)\ \prod_{y \in Y_{i-1}} w(y)}$$

$$= \underbrace{\frac{|X_{i-1}|}{|X_i|}}_{\text{choice } x_0} \underbrace{\cancel{\frac{\prod_{x \in X_i^{\text{reuse}}} w(x)}{\prod_{x \in X_{i-1}^{\text{reuse}}} w(x)}}}_{\text{reused}} \underbrace{\frac{\prod_{y \in Y_i} w(y)}{\prod_{y \in Y_{i-1}} w(y)}}_{\text{scores}}$$

Single-site Metropolis Hastings

infer.ml

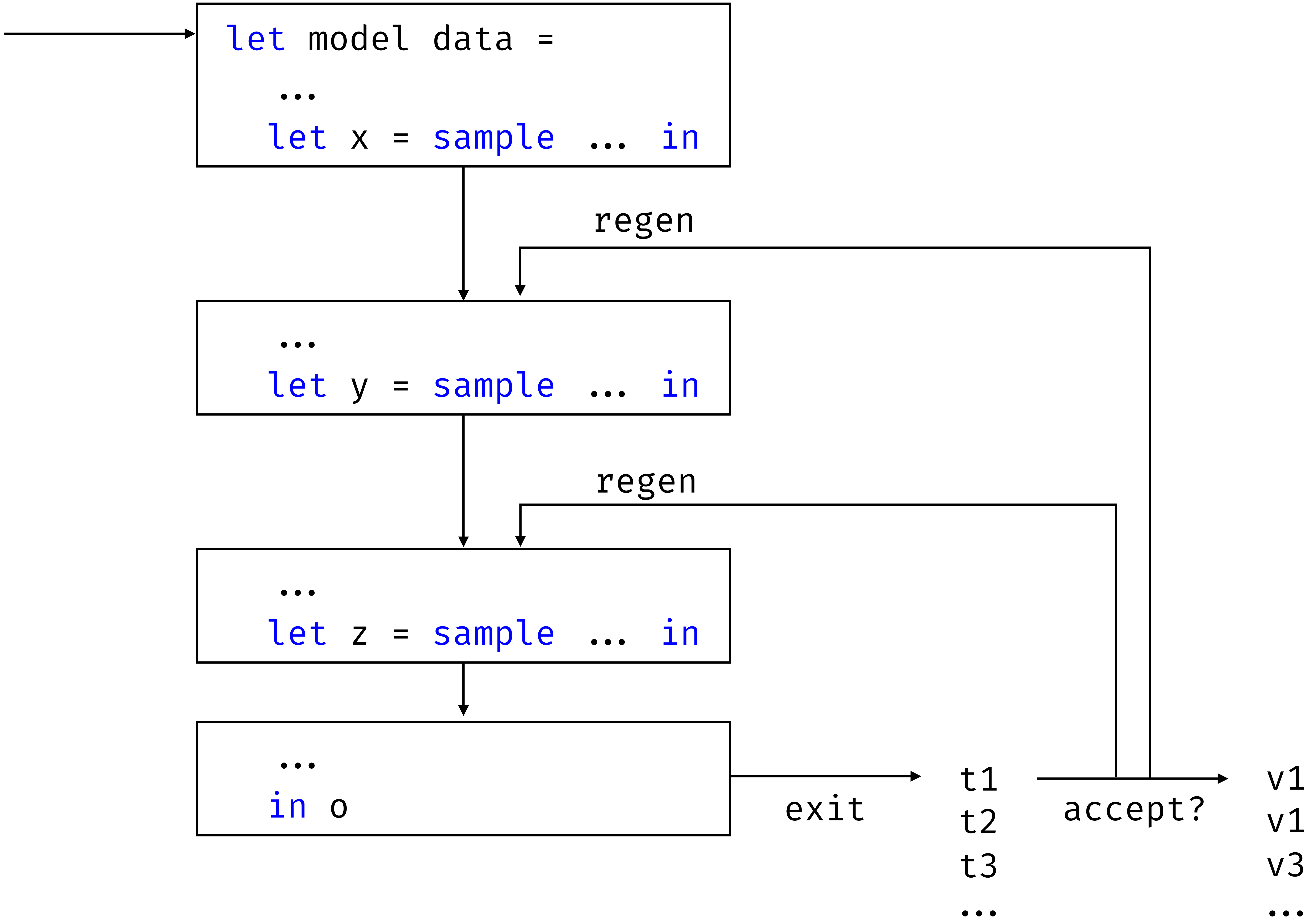
```
module Metropolis_hasting: sig
  type 'a prob
  and 'a next = 'a prob → 'a prob
  and ('a, 'b) model = 'a → ('b → 'b next) → 'b next

  val sample : 'a Distribution.t → ('a → 'b next) → 'b next
  val factor : float → (unit → 'b next) → 'b next
  val infer : ('a, 'b) model → 'a → 'b Distribution.t
end = struct ... end
```

Inference algorithm

- Generate an execution trace
- Compare the score to the previous trace
- Accept (keep the new trace) or reject (keep the previous trace)
- Draw a sample site at random in the execution path
- Restart execution from this site to generate a new trace
- Stop after n iterations

Single-site Metropolis Hastings



Single-site Metropolis Hastings

```
module Metropolis_hasting = struct
  type 'a prob = {
    score : float;
    trace : 'a sample_site list;
    value : 'a option;
  }

  and 'a sample_site =
    | Sample : {
        k : 'b → 'a next;
        score : float;
        dist : 'b Distribution.t;
      }
        → 'a sample_site

  and 'a next = 'a prob → 'a prob
  and ('a, 'b) model = 'a → ('b → 'b next) → 'b next
```

Single-site Metropolis Hastings

```
let sample dist k prob =
  let value = Distribution.draw dist in
  let sample_site = Sample { k; score = prob.score; dist } in
  k value { prob with trace = sample_site :: prob.trace }

let factor s k prob = k () { prob with score = prob.score +. s }

let observe d x k prob = factor (Distribution.logpdf d x) k prob

let exit v prob = { prob with value = Some v }
```

Single-site Metropolis Hastings

```
let mh prob prob' =
  let fw = -.log (prob.trace ▷ List.length ▷ Float.of_int) in
  let bw = -.log (prob'.trace ▷ List.length ▷ Float.of_int) in
  min 1. (exp (prob'.score -. prob.score +. bw -. fw))
```

Single-site Metropolis Hastings

```
let rec gen n values prob =
  if n = 0 then values
  else
    let regen_from = Random.int (List.length prob.trace) in
    let (Sample regen) = List.nth prob.trace regen_from in
    let prob' =
      sample regen.dist regen.k
        {
          prob with
          trace = Utils.slice prob.trace regen_from;
          score = regen.score;
        }
    in
    let next_prob = if Random.float 1. < mh prob prob' then prob' else prob in
    gen (n - 1) (Option.get next_prob.value :: values) next_prob
```

Single-site Metropolis Hastings

```
  let infer ?(n = 1000) m data =
    let prob = (m data) exit { score = 0.; trace = []; value = None } in
    let values = gen n [] prob ▷ Array.of_list in
    Distribution.uniform_support ~values
end
```

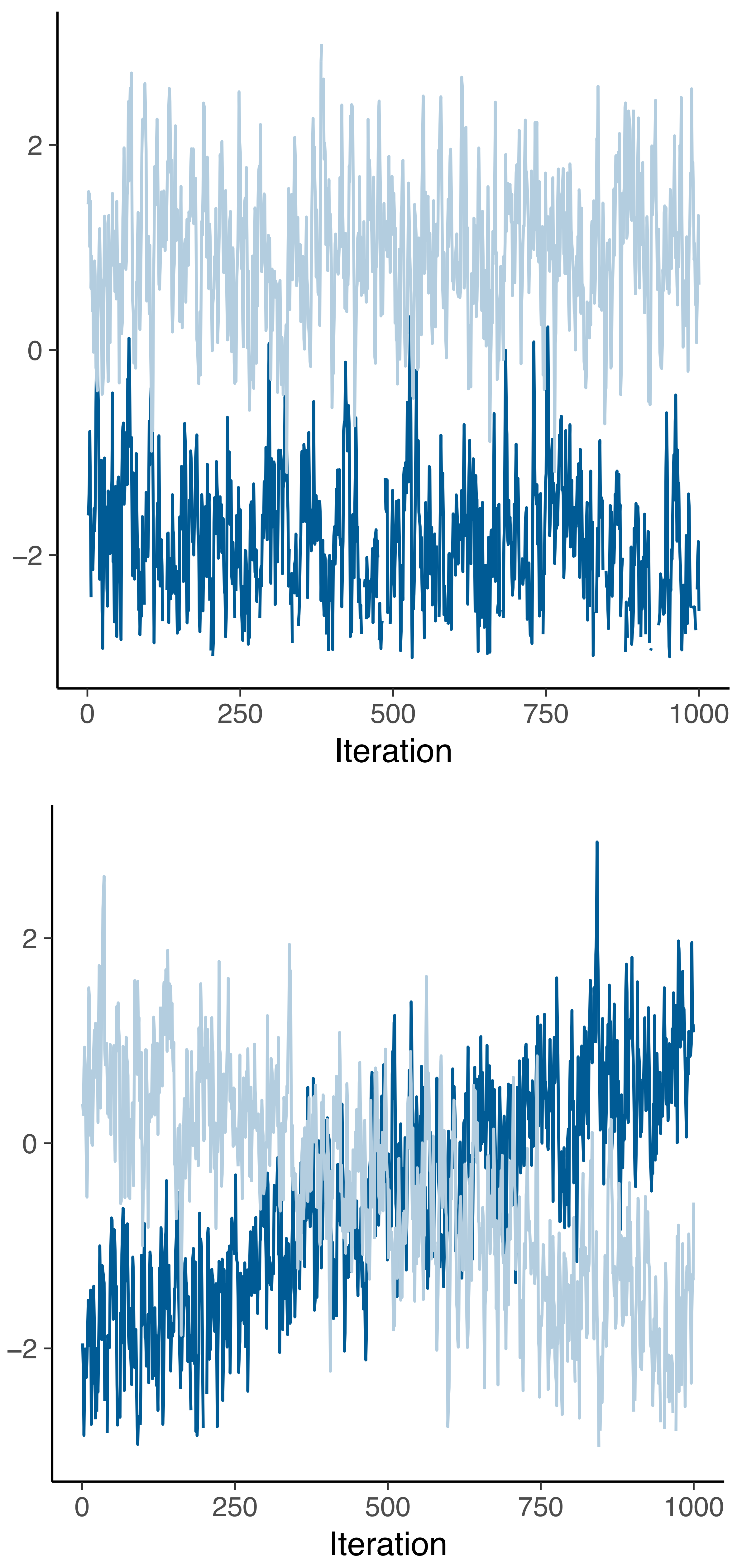
Limitations

Convergence

- Theoretical conditions are complex
- Must be checked experimentally
- Diagnostic tools: trace plot, R-hat (multi-chains)
- Solution: warmup, change initial conditions, reparameterization, ...

Sample correlation

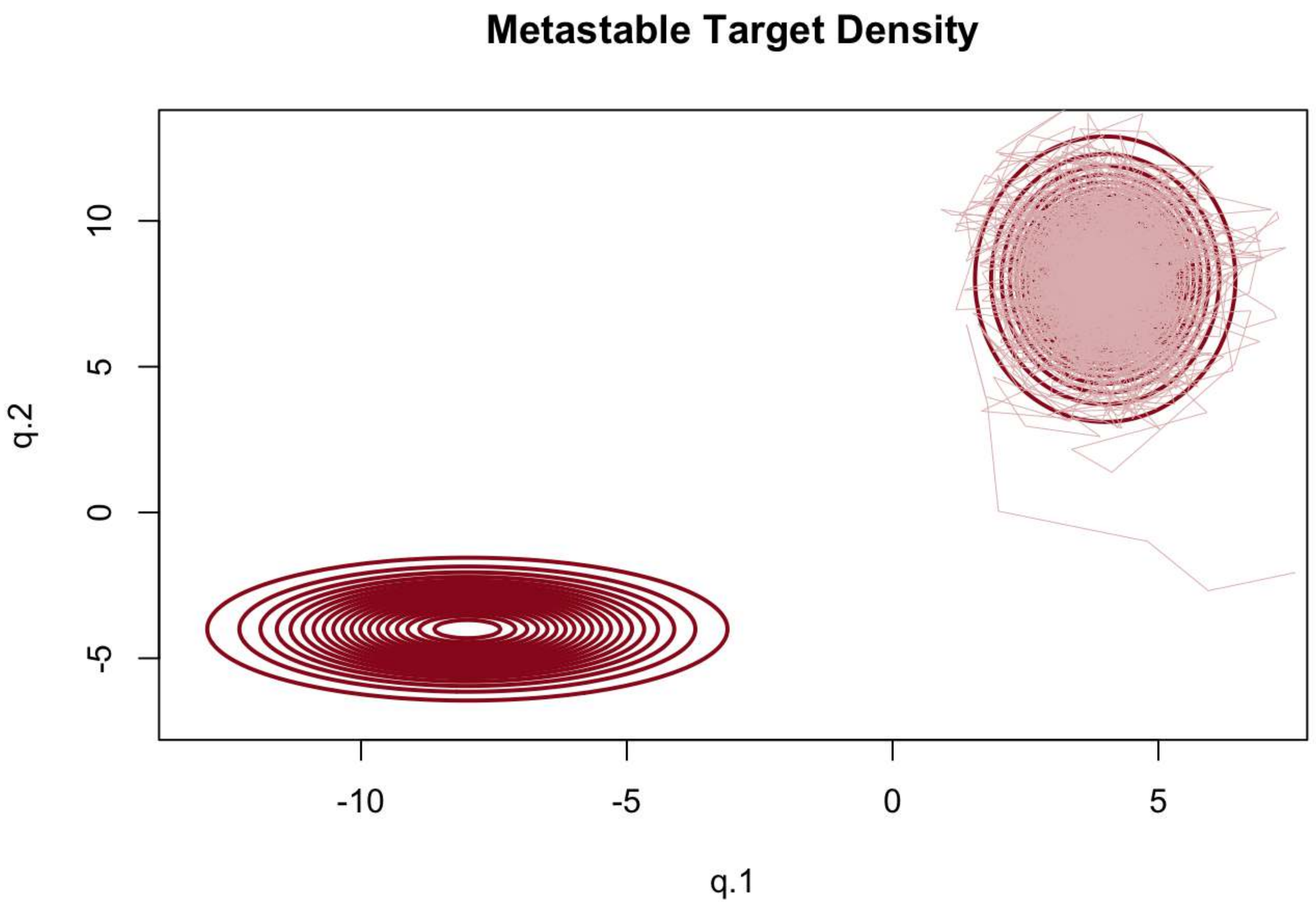
- Next sample depends on the previous one
- Diagnostic tools ESS (effective sample size)
- Solution: thinning, ...



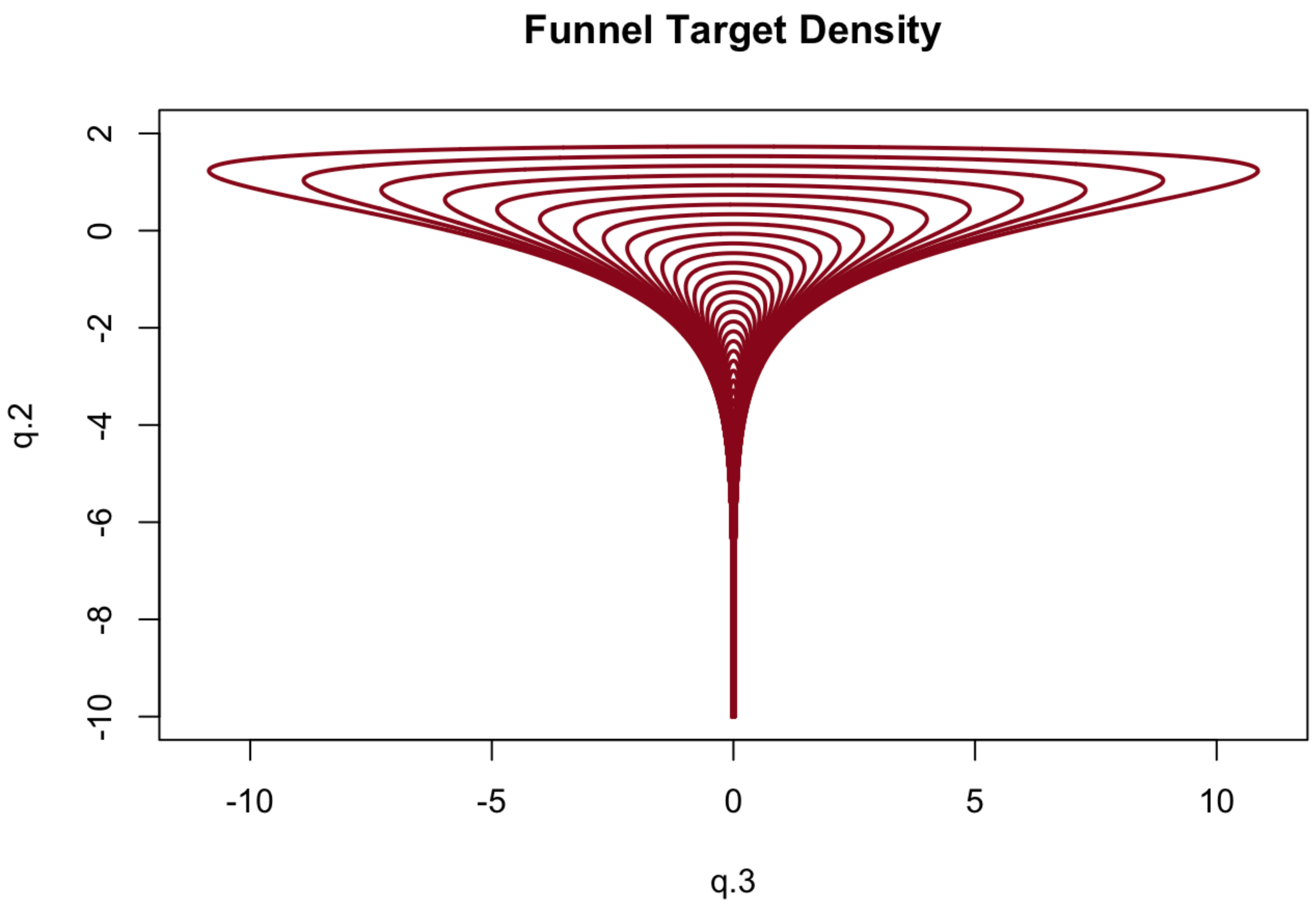
Vehtari et al. 2021

Limitations

Pathological models



Multimodal distribution



Neal's funnel

https://betanalpha.github.io/assets/case_studies/markov_chain_monte_carlo.html

Advanced inference

Probabilistic Programming Languages

Stan: A probabilistic programming language

"A Stan program imperatively defines a log probability function over parameters conditioned on specified data and constants"

Brief History:

- Named after Stanisław Marcin Ulam (Monte Carlo method)
- Initial release: August 30, 2012
- Today: version 2.33 (50+ iterations after 1.0.0)

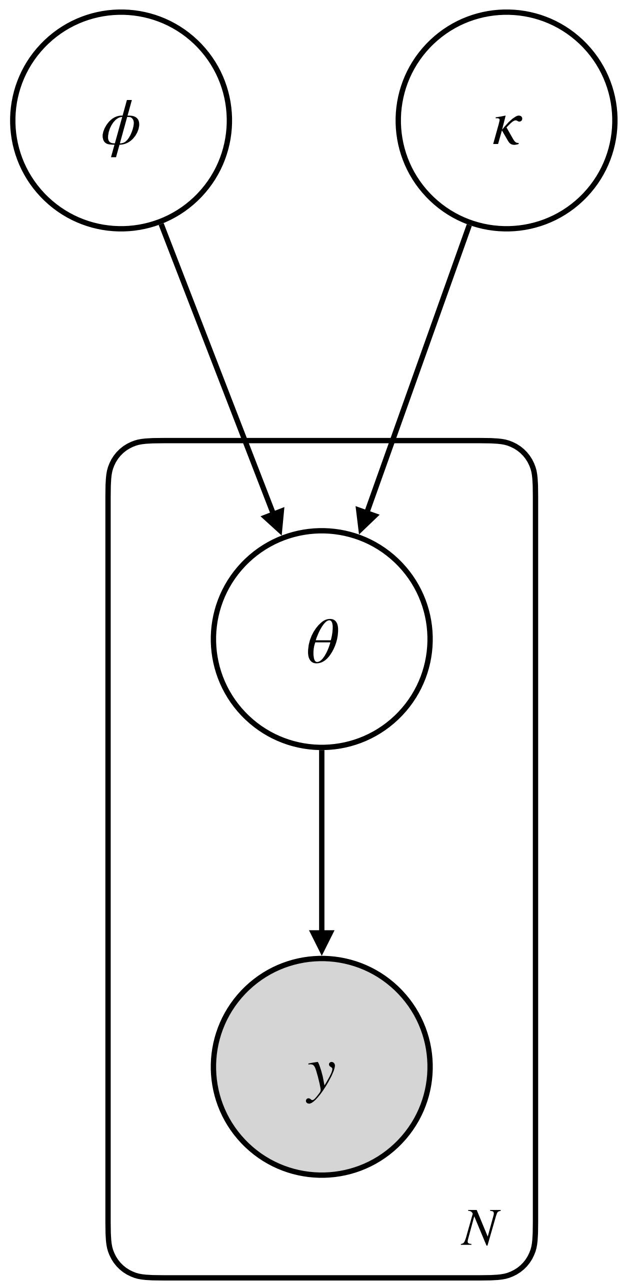
In a nutshell:

- Small imperative language to describe probabilistic models
- Bayesian inference on continuous latent variables
- No U-Turn Sampler (NUTS): an optimized Hamiltonian Monte Carlo (HMC) inference
- Interface with popular programming language: R, Python, etc…

Carpenter et al. 2017

Stan: reminders

```
data {
  int<lower=0> N;              // players
  int<lower=0> K[N];           // initial trials
  int<lower=0> y[N];           // initial successes
}
parameters {
  real<lower=0, upper=1> phi;          // population chance of success
  real<lower=1> kappa;                 // population concentration
  vector<lower=0, upper=1>[N] theta;   // chance of success
}
model {
  kappa ~ pareto(1, 1.5);                          // hyperprior
  theta ~ beta(phi * kappa, (1 - phi) * kappa);    // prior
  y ~ binomial(K, theta);                          // likelihood
}
```



Carpenter 2016

Stan semantics

Parameters and Data blocks

- Declared in their respective blocks (before the model)
- Precisely identified before inference

Model block

- A small imperative deterministic language with a global accumulator `target`
- Two constructs to update this accumulator: `~` and `target +=`

The language is deterministic: statements update the environment

- $[\![s]\!] : \Gamma \to \Gamma$
- The model block defines an un-normalized log-density function

model is deterministic

Probabilistic semantics: turn the log-density into a measure over parameters

- $\{\!|p|\!\} : \mathcal{D} \to \Sigma_X \to [0, \infty)$, where X is the parameters domain
- $\{\!|p|\!\}_D = \lambda U. \int_U \exp([\![\texttt{model}(p)]\!]_{D,\theta}(\texttt{target}))\, d\theta.$
- D is the initial environment containing the (observed) data.

function of parameter and data

Gorinova et al. 2019

Stan semantics

Statements: $[\![s]\!] : \Gamma \to \Gamma$

$$
\begin{aligned}
[\![\texttt{skip}]\!]_\gamma &= \gamma \\
[\![x \texttt{ = } e]\!]_\gamma &= \gamma + [x \leftarrow [\![e]\!]_\gamma] \\
[\![x\texttt{[}e_1,\ldots,e_n\texttt{] = }e]\!]_\gamma &= \gamma + \left[x\texttt{[}[\![e_1]\!]_\gamma,\ldots,[\![e_n]\!]_\gamma\texttt{]} \leftarrow [\![e]\!]_\gamma\right] \\
[\![\texttt{if(}e\texttt{) } s_1 \texttt{ else } s_2]\!]_\gamma &= \textit{if } [\![e]\!]_\gamma = 0 \textit{ then } [\![s_1]\!]_\gamma \textit{ else } [\![s_2]\!]_\gamma \\
[\![s_1 \texttt{ ; } s_2]\!]_\gamma &= [\![s_2]\!]_{[\![s_1]\!]_\gamma} \\
[\![\texttt{for(}x \texttt{ in } e_1\texttt{:}e_2\texttt{)\{}s\texttt{\}}]\!]_\gamma &= \textit{let } n_1 = [\![e_1]\!]_\gamma \textit{ in let } n_2 = [\![e_2]\!]_\gamma \textit{ in} \\
&\quad\ \textit{if } n_1 > n_2 \textit{ then } \gamma \textit{ else } [\![\texttt{for(}x \texttt{ in } n_1+1\texttt{:}n_2\texttt{)\{}s\texttt{\}}]\!]_{[\![s]\!]_{\gamma+[x\leftarrow n_1]}} \\
[\![\texttt{while(}e\texttt{)\{}s\texttt{\}}]\!]_\gamma &= \textit{if } [\![e]\!]_\gamma = 0 \textit{ then } \gamma \textit{ else } [\![\texttt{while(}e\texttt{)\{}s\texttt{\}}]\!]_{[\![s]\!]_\gamma} \\
[\![\texttt{target += } e]\!]_\gamma &= \gamma + [\texttt{target} \leftarrow \gamma(\texttt{target}) + [\![e]\!]_\gamma] \\
[\![e_1 \sim e_2]\!] &= \textit{let } \mu = [\![e_2]\!]_\gamma \textit{ in } [\![\texttt{target += } \mathrm{pdf}(\mu)(e_1)]\!]_\gamma
\end{aligned}
$$

imperative, deterministic

Gorinova et al. 2019

Density semantics beyond Stan

Remember: sampler semantics

- Expressions are interpreted as weighted samplers in log space
- Given an environment γ, $\{\!|e|\!\}_\gamma = v, w$
- $\{\!|e|\!\} : \Gamma \to V \times \mathbb{R}$
- Parameters are now inputs

$$
\begin{aligned}
\{\!|c|\!\}_\gamma &= c, 0 \\
\{\!|x|\!\}_\gamma &= \gamma(x), 0 \\
\{\!|\texttt{sample}(e)|\!\}_\gamma &= ??? \\
\{\!|\texttt{factor}(e)|\!\}_\gamma &= (\,), -[\![e]\!]_\gamma \\
\{\!|\texttt{observe}(e_1, e_2)|\!\}_\gamma &= (\,), -\text{logpdf}([\![e_1]\!]_\gamma)([\![e_2]\!]_\gamma) \\
\{\!|\texttt{let}\ x = e_1\ \texttt{in}\ e_2|\!\}_\gamma &= \textit{let}\ v_1, w_1 = \{\!|e_1|\!\}_\gamma\ \textit{in} \\
&\quad\ \textit{let}\ v_2, w_2 = \{\!|e_2|\!\}_{\gamma+[x \leftarrow v_1]}\ \textit{in} \\
&\quad\ v_2, w_1 + w_2
\end{aligned}
$$

$\texttt{let}\ x = \texttt{sample}(e)\ \texttt{in} \ldots \to \texttt{observe}(e, x)\ ;\ \ldots$

```
sample mu (* where mu is defined on [a, b] *)
≡
let x = sample (uniform (a, b)) in
let () = observe (mu, x) in
x
```

all parameters must be know statically

Hamiltonian Monte-Carlo (HMC)

Preferred inference algorithm for Stan

Analogy: Particle in an energy field

- Program define a density of the form exp$(-U(X))$
- On continuous spaces U can be interpreted as an energy
- Low energy wells correspond to high probability regions
- HMC simulate the trajectory of a particle in this energy field

Hamiltonian Dynamics

- M: mass matrix
- P: momentum

$$K(P) = \frac{1}{2} P^T M^{-1} P$$

hamiltonian

potential energy

$$\underbrace{H(X,P)}_{\text{hamiltonian}} = \underbrace{K(P)}_{\text{kinetic energy}} + \underbrace{U(X)}_{\text{potential energy}}$$

kinetic energy

Hamiltonian Monte-Carlo (HMC)

Energy conservation

$$\frac{dH}{dt} = (\nabla_P H)^T \frac{dP}{dt} + (\nabla_X H)^T \frac{dX}{dt}$$

Hamiltonian dynamics

$$\begin{cases} \dfrac{dX}{dt} = \nabla_P H(X, P) = M^{-1} P \\ \dfrac{dP}{dt} = -\nabla_X H(X, P) = -\nabla_X U(X) \end{cases}$$

hamiltonian

potential energy

$$H(X, P) = K(P) + U(X)$$

kinetic energy

Hamiltonian Monte-Carlo (HMC)

Generate samples (X, P) from the density $\exp(-H(X, P))$

- At each iteration
- Sample an initial momentum $P_0 \sim \mathcal{N}(0, M)$
- Solve the Hamiltonian dynamics (discretized)
- Perform a Metropolis Hastings update with probability α

momentum can then be marginalized

$$\alpha = \min\left(1, \frac{\exp(-H(X_i, P_i))}{\exp(-H(X_{i-1}, P_{i-1}))}\right)$$

If the hamiltonian is preserved: accept with probability 1.

- Problem: numerical approximations
- Solution: leapfrog integrator and reject using MH acceptance probability

Hamiltonian Monte-Carlo (HMC)

```
let u x = let _, u = model data x in u
let k p =  0.5 * transpose p * inv m * p

let h x p = u x +. k p

let rec gen n values x =
  if n = 0 then values
  else
   let p = Distribution.draw mv_normal(0, m) in
   let x', p' = leapfrog (grad u) x p in
   let next_x = if Random.float 1. < exp(h x p -. h x' p') then x' else x in
   let next_value, _ = model data next_x in
   gen (n - 1) (next_value :: values) next_x
```

autodiff magic!

Warning: pseudo-code

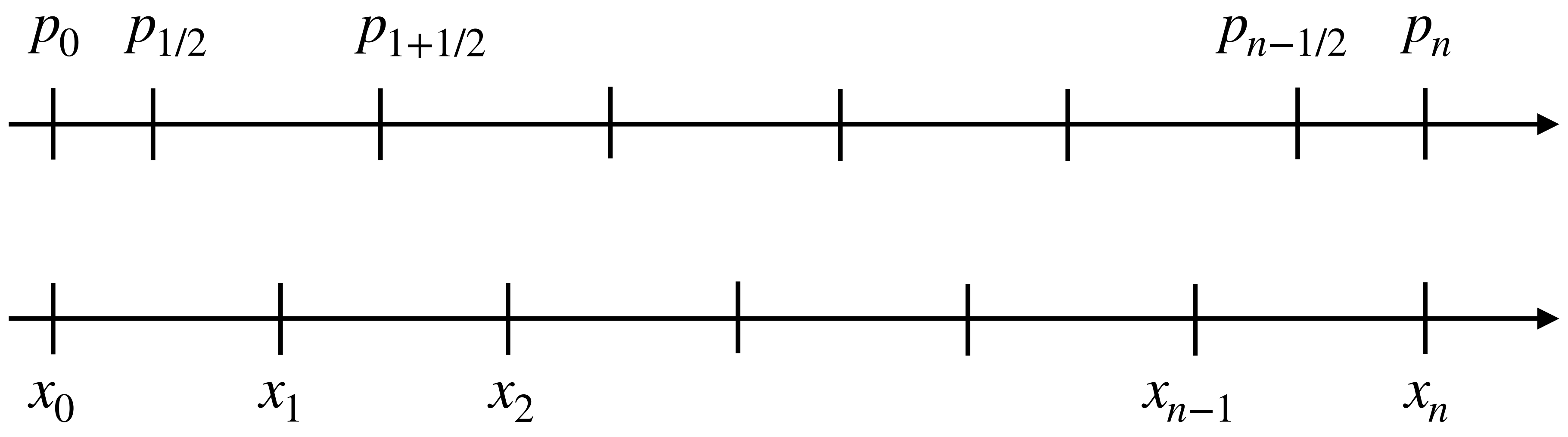
Leapfrog integration

```
let leapfrog u_grad x0 p0 =
  let p = p0 - 0.5 * step_size * u_grad x0 in
  let rec loop n x p =
    if n = 0 then x, p
    else
      let x' = x + step_size * p in
      let p' = p - step_size * u_grad x' in
      loop (n-1) x' p'
  in
  let xt, pt = loop (path_len - 1) x0 p in
  let x' = xt + step_size * pt in
  let p' = pt - 0.5 * step_size * u_grad x' in
  x', p'
```

first half-step for the momentum

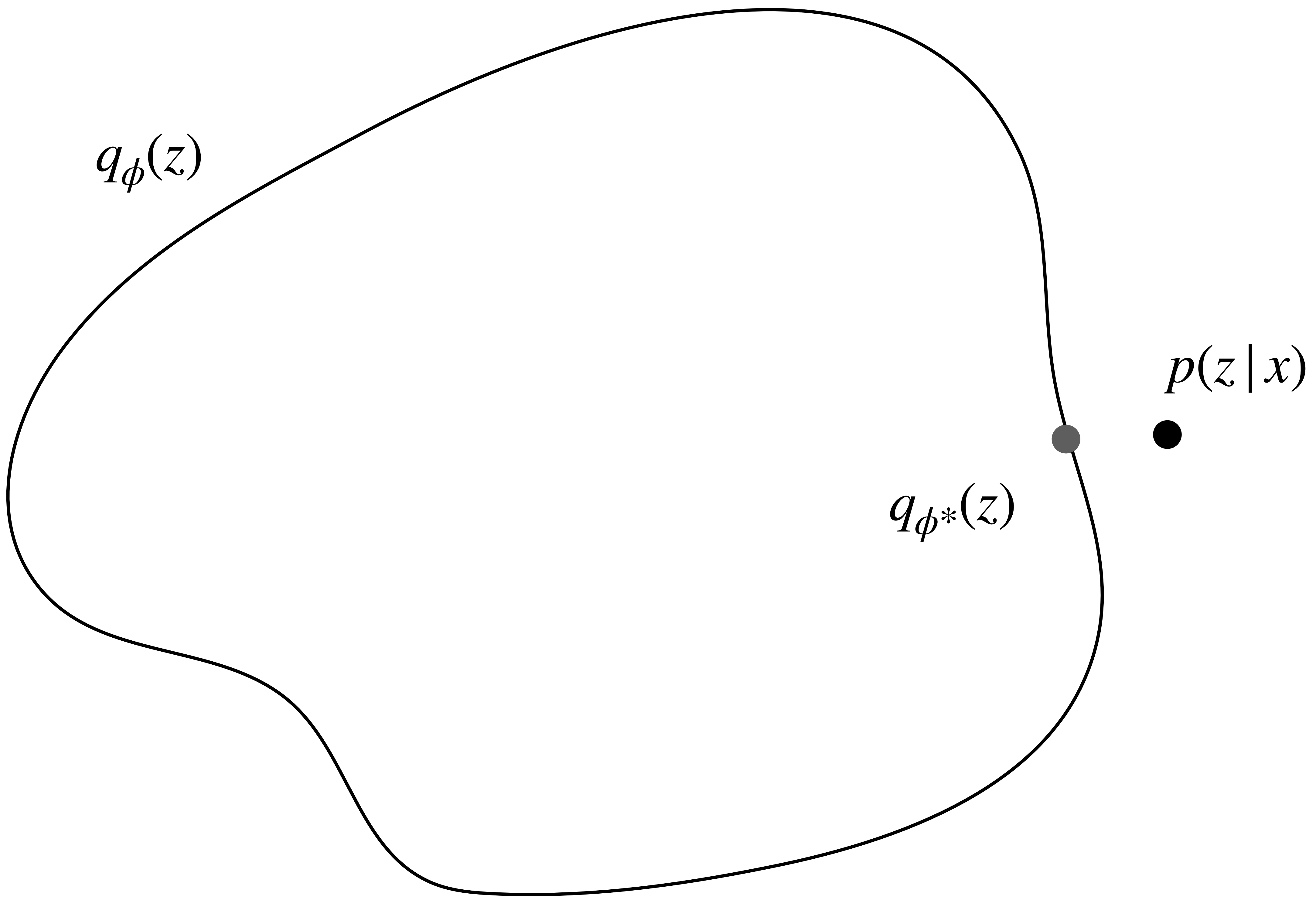
last half-step for the momenum

Warning: pseudo-code



Stochastic Variational Inference (SVI)

$$p(z \mid x) = \frac{p(x \mid z)p(z)}{p(x)} = \frac{p(x \mid z)p(z)}{\int_z p(x \mid z)p(z)dz}$$



Variational family

- Parameterized by a parameter ϕ
- Find the closest member to the posterior $q_{\phi^*}(z)$
- Optimization problem

Metrics: Kullback-Leibler divergence

$$KL(q(x) \parallel p(x)) = -\int q(x) \log \frac{p(x)}{q(x)}$$

- $KL(q \parallel p) \geq 0$ positive
- $KL(q \parallel p) = 0 \iff |x| \neq 0 \implies p(x) = q(x)$, equal almost everywhere
- $KL(q \parallel p) \neq KL(p \parallel q)$ asymetric
- No triangular inequality

Blei et al. 2017

Stochastic Variational Inference (SVI)

$$
\begin{aligned}
KL(q_\phi(z) \parallel p(z|x)) &= -\int q_\phi(z) \log \frac{p(z|x)}{q_\phi(z)}\ dz \\
&= -\int q_\phi(z) \log \frac{p(x,z)}{p(x)q_\phi(z)}\ dz \\
&= -\int q_\phi(z) \log \frac{p(x,z)}{q_\phi(z)}\ dz + \int q_\phi(z) \log p(x)\ dz \\
&= -\int q_\phi(z) \log \frac{p(x,z)}{q_\phi(z)}\ dz + \log p(x)
\end{aligned}
$$

Stochastic Variational Inference (SVI)

$$
\begin{aligned}
KL(q_\phi(z) \parallel p(z|x)) &= -\int q_\phi(z) \log \frac{p(z|x)}{q_\phi(z)}\, dz \\
&= -\int q_\phi(z) \log \frac{p(x,z)}{p(x)q_\phi(z)}\, dz \\
&= -\int q_\phi(z) \log \frac{p(x,z)}{q_\phi(z)}\, dz + \int q_\phi(z) \log p(x)\, dz \\
&= -\int q_\phi(z) \log \frac{p(x,z)}{q_\phi(z)}\, dz + \log p(x)
\end{aligned}
$$

$$
\underbrace{\log p(x)}_{\text{constant}} = \underbrace{KL(q_\phi(z) \parallel p(x \mid z))}_{\text{minimize}} + \underbrace{\int q_\phi(z)\, \log \frac{p(x,z)}{q_\phi(z)}\, dz}_{\text{maximize ELBO}}
$$

Stochastic Variational Inference (SVI)

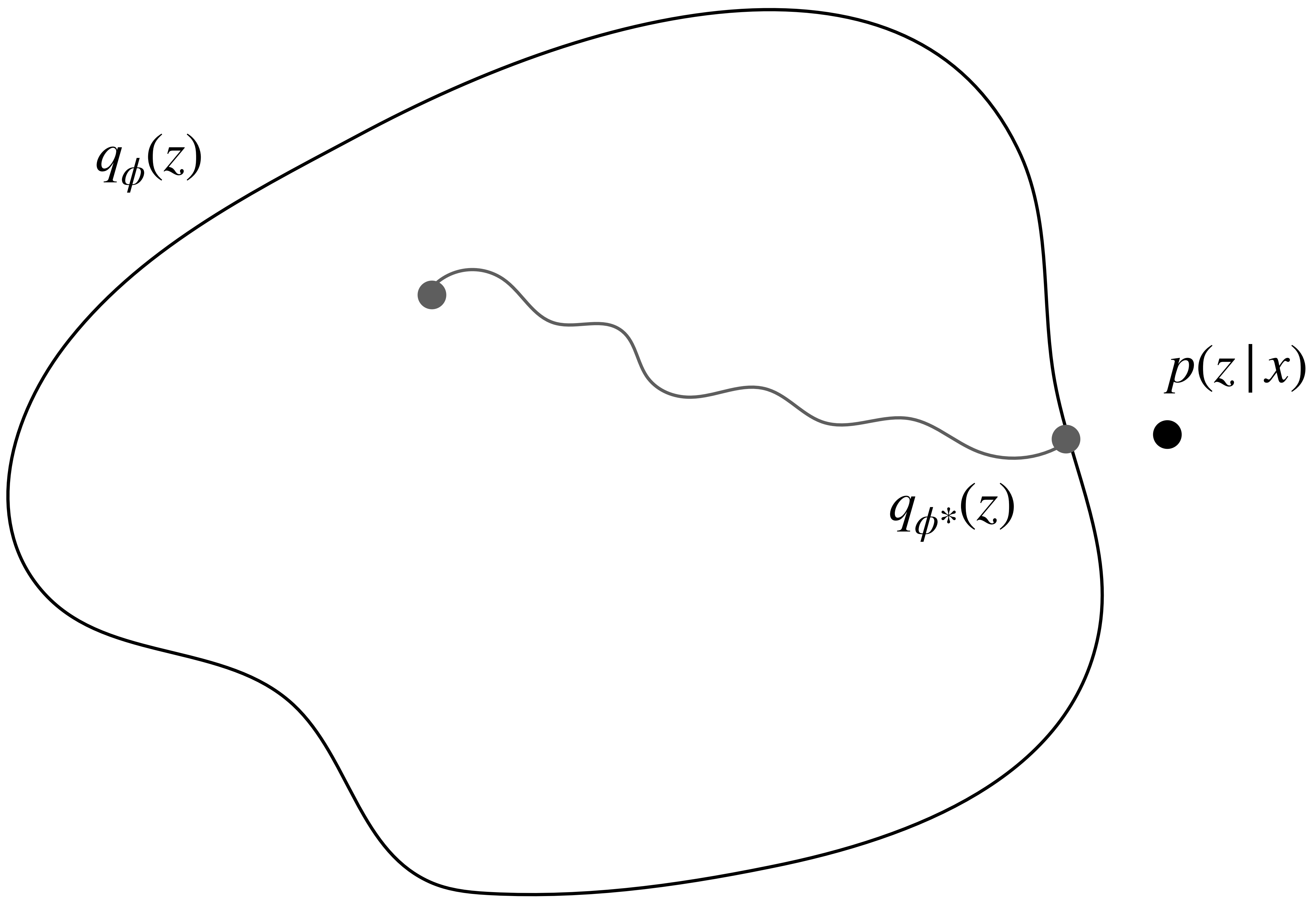
How to solve the optimisation problem?

Program your own guide

- Pyro (first versions)
- Sample the same variables in the guide and the model

```
def model():
  pyro.sample("z_1", ... )

def guide():
  pyro.sample("z_1", ... )
```



Approximate gradient ascent.

$$\nabla_\phi KL(q_\phi(z) \parallel p(z \mid x)) \longrightarrow \nabla_\phi \mathcal{L} = \nabla_\phi \int q_\phi(z) \log \frac{p(x,z)}{q_\phi(z)}\, dz$$

autodiff magic!

Lee, Yu, Rival, Yang POPL 2020

Stochastic Variational Inference (SVI)

How to solve the optimisation problem?

Black-box variational inference

- Variational families with tractable solution
- Mean-field approximation $q_\phi(z) = \prod_{i=1}^{n} \mathcal{N}(z_i|\mu_i, \sigma_i)$ where $\phi = \{\mu_i, \sigma_i\}_{i \in [1,n]}$
- Full-rank approximation $q_\phi(z) = \mathcal{N}(z|\mu, \Sigma)$ where $\phi = (\mu, \Sigma)$

Assumptions

- Independences between random variables
- Only use Gaussians distributions

Blei et al. 2017

References

An Introduction to Probabilistic Programming
Jan-Willem van de Meent, Brooks Paige, Hongseok Yang, Frank Wood
https://arxiv.org/abs/1809.10756

Lightweight Implementations of Probabilistic Programming Languages Via Transformational Compilation
David Wingate Andreas Stuhlmüller Noah D. Goodman
AISTATS 2011

Markov Chain Monte Carlo in Practice
Michael Bettancourt
https://betanalpha.github.io/assets/case_studies/markov_chain_monte_carlo.html

Variational Inference: A Review for Statisticians
David M. Blei, Alp Kucukelbir, Jon D. McAuliffe
https://arxiv.org/abs/1601.00670