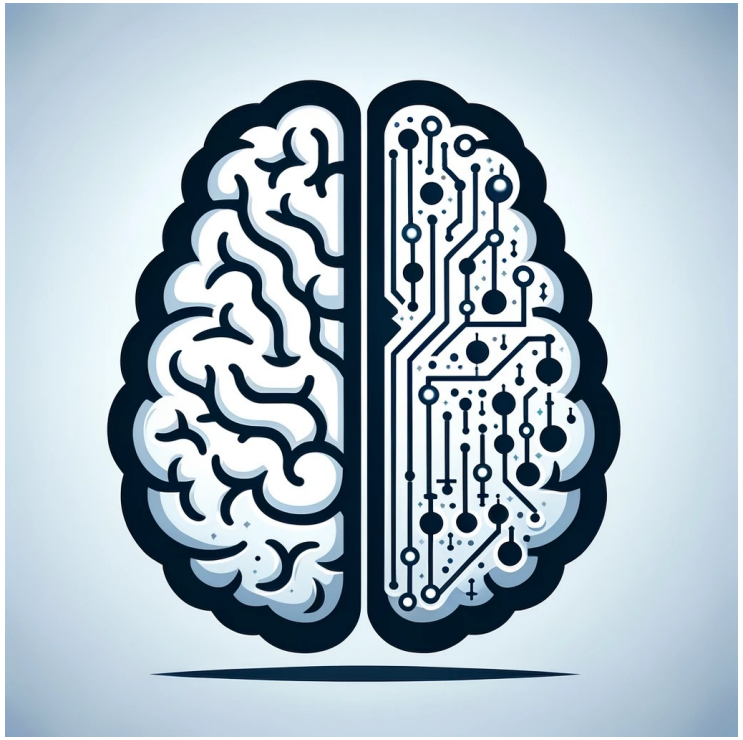
EN 601.473/601.673: Cognitive Artificial Intelligence (CogAI)



Lecture 11: MCMC in Gen, RJMCMC, SMC

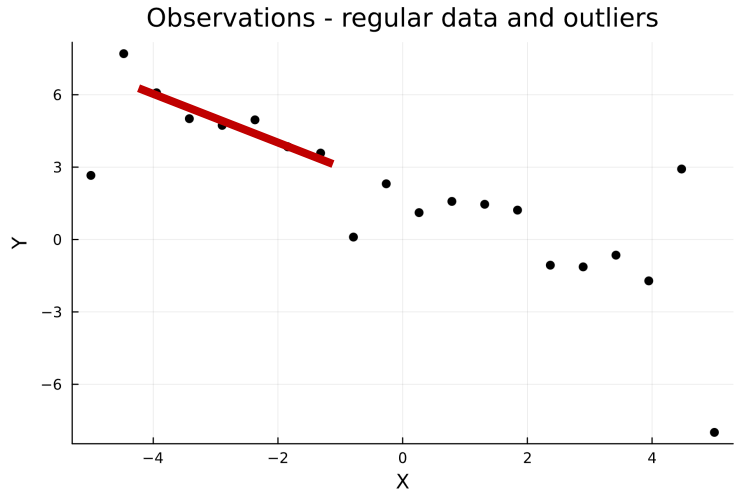
Tianmin Shu

Problem set 2 release

- Release: next lecture, Feb 29
- Due by March 26, end of day
- Q1: importance sampling vs MCMC in Gen
- Q2: SMC (particle filtering) in Gen
- Start as early as possible (also, go through the notebooks for the lectures)

Use heuristics to make smarter proposals

- Data-driven proposals q(H | data)



- An example: RANSAC (Random Sample Consensus)
- Repeatedly choose a small random subset of the points, say, of size 3.
- Conduct least-squares linear regression to find a line of best fit for those points.
- Count how many points (from the entire set) are near the line we found.
- After a suitable number of iterations (say, 10), return the line that had the highest score.

RANSAC in Julia

```
import StatsBase

struct RANSACParams
    """the number of random subsets to try"""
    iters::Int

    """the number of points to use to construct a hypothesis"""
    subset_size::Int

    """the error threshold below which a datum is considered an inlier"""
    eps::Float64

    function RANSACParams(iters, subset_size, eps)
        if iters < 1
            error("iters < 1")
        end
        new(iters, subset_size, eps)
    end
end
```

```
function ransac(xs::Vector{Float64}, ys::Vector{Float64}, params::RANSACParams)
    best_num_inliers::Int = -1
    best_slope::Float64 = NaN
    best_intercept::Float64 = NaN
    for i=1:params.iters
        # select a random subset of points
        rand_ind = StatsBase.sample(1:length(xs), params.subset_size, replace=false)
        subset_xs = xs[rand_ind]
        subset_ys = ys[rand_ind]

        # estimate slope and intercept using least squares
        A = hcat(subset_xs, ones(length(subset_xs)))
        slope, intercept = A \ subset_ys # use backslash operator for least sq soln

        ypred = intercept .+ slope * xs

        # count the number of inliers for this (slope, intercept) hypothesis
        inliers = abs.(ys - ypred) .< params.eps
        num_inliers = sum(inliers)

        if num_inliers > best_num_inliers
            best_slope, best_intercept = slope, intercept
            best_num_inliers = num_inliers
        end
    end

    # return the hypothesis that resulted in the most inliers
    (best_slope, best_intercept)
end;
```

RANSAC proposal in Gen

- Wrap the ransac Julia function in a Gen proposal

```
@gen function ransac_proposal(prev_trace, xs, ys)
    (slope_guess, intercept_guess) = ransac(xs, ys, RANSACParams(10, 3, 1.))
    slope ~ normal(slope_guess, 0.1)
    intercept ~ normal(intercept_guess, 1.0)
end;
```

One iteration of RANSAC-based update

```
function ransac_update(tr)
    # Use RANSAC to (potentially) jump to a better line
    # from wherever we are
    (tr, _) = mh(tr, ransac_proposal, (xs, ys))

    # Spend a while refining the parameters, using Gaussian drift
    # to tune the slope and intercept, and resimulation for the noise
    # and outliers.
    for j=1:20
        (tr, _) = mh(tr, select(:prob_outlier))
        (tr, _) = mh(tr, select(:noise))
        (tr, _) = mh(tr, line_proposal, ())
        # Reclassify outliers
        for i=1:length(get_args(tr)[1])
            (tr, _) = mh(tr, select(:data => i => :is_outlier))
        end
    end
    tr
end
```

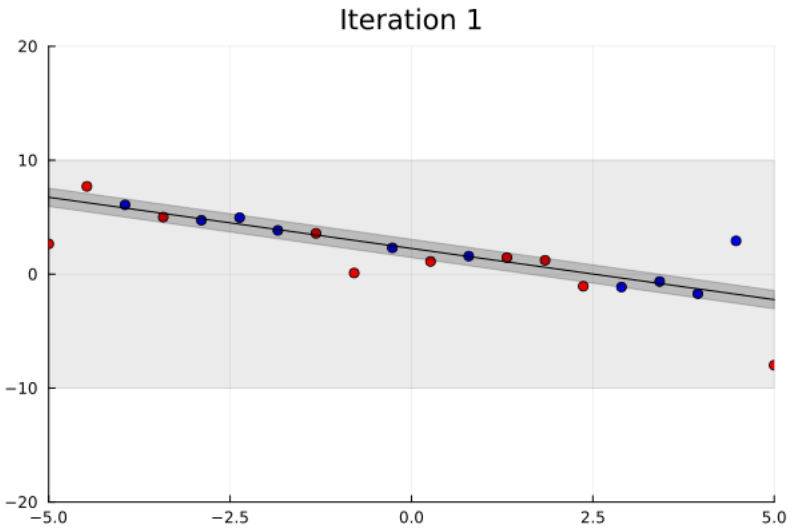
RANSAC-based MCMC inference

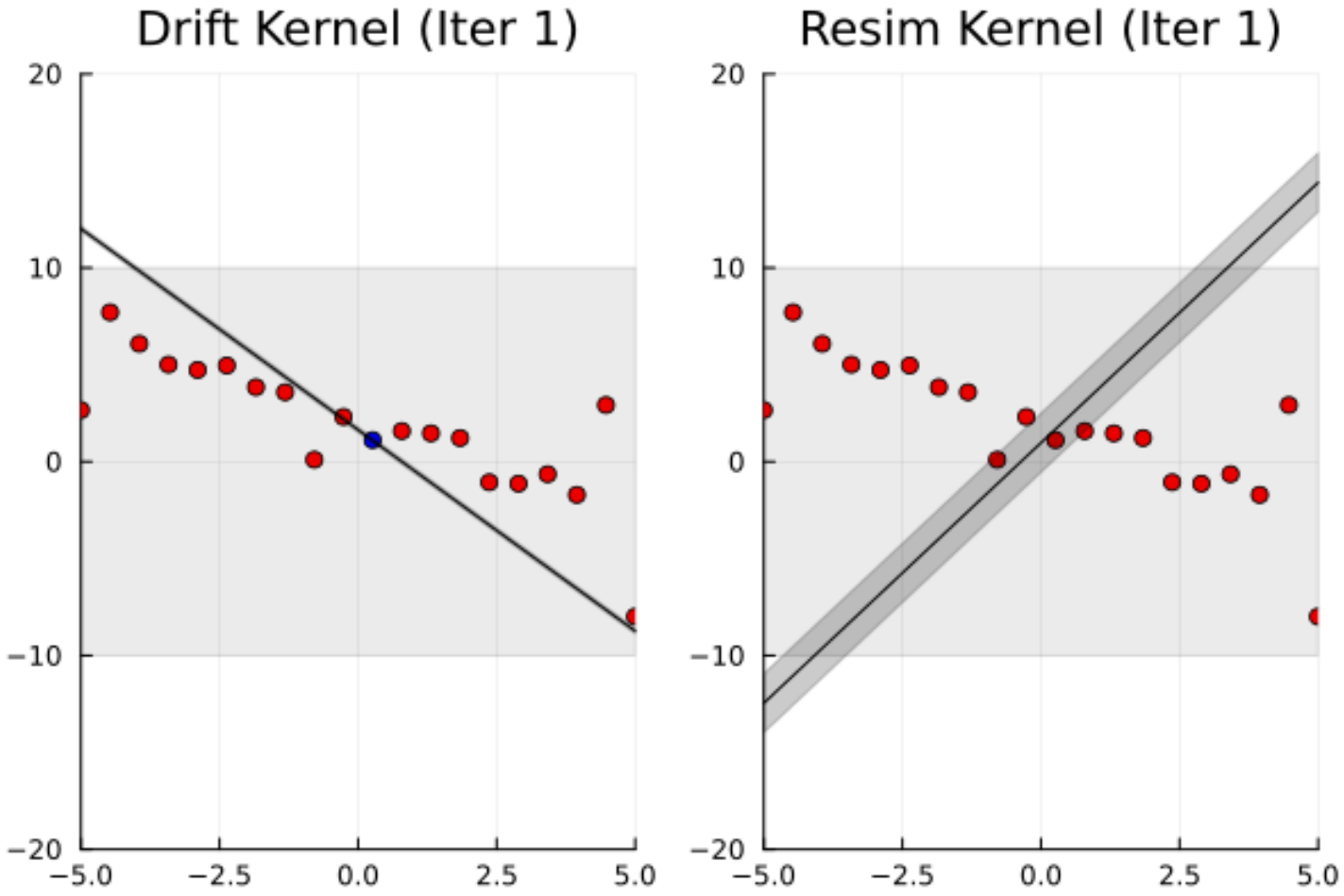
- Sample the initial proposal of slope and intercept using RANSAC
- Conduct RANSAC-based MH update for 5 iterations

```
function ransac_inference(xs, ys, observations)
    (slope, intercept) = ransac(xs, ys, RANSACParams(10, 3, 1.))
    slope_intercept_init = choicemap()
    slope_intercept_init[:slope] = slope
    slope_intercept_init[:intercept] = intercept
    (tr, _) = generate(regression_with_outliers, (xs,), merge(observations, slope_intercept_init))
    for iter=1:5
        tr = ransac_update(tr)
    end
    tr
end
```

RANSAC-based MCMC inference

RANSAC Kernel





Finds a reasonable proposal much earlier thanks to the heuristics (data-driven proposal)

Block resimulation MH

```
  1.343702 seconds (11.47 M allocations: 676.832 MiB, 4.73% gc time, 29.51% compilation time)
  0.802108 seconds (11.06 M allocations: 648.812 MiB, 5.46% gc time)
  0.729552 seconds (11.06 M allocations: 648.812 MiB, 5.89% gc time)
  0.787784 seconds (11.06 M allocations: 648.812 MiB, 5.40% gc time)
  0.955552 seconds (11.06 M allocations: 648.812 MiB, 4.78% gc time)
  0.891746 seconds (11.06 M allocations: 648.812 MiB, 5.89% gc time)
  0.920517 seconds (11.06 M allocations: 648.812 MiB, 6.03% gc time)
  0.909784 seconds (11.06 M allocations: 648.812 MiB, 5.95% gc time)
  0.839699 seconds (11.06 M allocations: 648.812 MiB, 5.54% gc time)
  0.713001 seconds (11.06 M allocations: 648.812 MiB, 5.84% gc time)
Log probability: -50.78536994535881
```

Gaussian drift MH

```
  1.051416 seconds (12.01 M allocations: 699.933 MiB, 5.93% gc time, 1.19% compilation time)
  1.161278 seconds (12.00 M allocations: 699.043 MiB, 5.36% gc time)
  1.032212 seconds (12.00 M allocations: 699.043 MiB, 5.30% gc time)
  0.732046 seconds (12.00 M allocations: 699.043 MiB, 6.20% gc time)
  0.967897 seconds (12.00 M allocations: 699.043 MiB, 5.54% gc time)
  0.947385 seconds (12.00 M allocations: 699.043 MiB, 5.59% gc time)
  1.106322 seconds (12.00 M allocations: 699.043 MiB, 5.77% gc time)
  1.220370 seconds (12.00 M allocations: 699.043 MiB, 4.90% gc time)
  0.815143 seconds (12.00 M allocations: 699.043 MiB, 5.54% gc time)
  0.778490 seconds (12.00 M allocations: 699.043 MiB, 6.39% gc time)
Log probability: -44.24115015595737
```

RANSAC proposal MH

```
  0.930152 seconds (3.70 M allocations: 230.803 MiB, 3.76% gc time, 76.41% compilation time)
  0.195143 seconds (2.41 M allocations: 144.268 MiB, 5.27% gc time)
  0.179629 seconds (2.41 M allocations: 144.268 MiB, 9.60% gc time)
  0.160974 seconds (2.41 M allocations: 144.268 MiB, 5.74% gc time)
  0.169705 seconds (2.41 M allocations: 144.268 MiB, 7.40% gc time)
  0.187931 seconds (2.41 M allocations: 144.268 MiB, 5.79% gc time)
  0.193349 seconds (2.41 M allocations: 144.268 MiB, 8.71% gc time)
  0.203889 seconds (2.41 M allocations: 144.268 MiB, 5.47% gc time)
  0.194845 seconds (2.41 M allocations: 144.268 MiB, 6.39% gc time)
  0.179412 seconds (2.41 M allocations: 144.268 MiB, 7.83% gc time)
Log probability: -42.80176245244315
```

- Better fitness
- Faster inference

No data-driven initial proposal?

```
function ransac_inference(xs, ys, observations)
    (slope, intercept) = ransac(xs, ys, RANSACParams(10, 3, 1.))
    slope_intercept_init = choicemap()
    slope_intercept_init[:slope] = slope
    slope_intercept_init[:intercept] = intercept
    (tr, _) = generate(regression_with_outliers, (xs,), merge(observations, slope_intercept_init))
    for iter=1:5
        tr = ransac_update(tr)
    end
    tr
end
```

Log probability: -42.80176245244315

```
function ransac_inference_no_init(xs, ys, observations)
    (tr, _) = generate(regression_with_outliers, (xs,), observations)
    for iter=1:5
        tr = ransac_update(tr)
    end
    tr
end
```

Log probability: -43.87146997454299

Summary: MCMC in Gen

- Define a generative model
- Generate a trace by running the generative model

```
(trace::U, weight) = generate(gen_fn::GenerativeFunction{T,U},
args::Tuple, constraints::ChoiceMap)
```

- Update *selected* variables in the trace (a sweep: update all variables)

```
(new_trace, accepted) = mh(trace, selection::Selection; ..)
```

```
(new_trace, accepted) = mh(trace,
proposal::GenerativeFunction, proposal_args::Tuple;
selection::Selection; ..)
```

Try it out yourself

- Go over the jupyter notebook “MCMC in Gen.ipynb”

Problem set 2 release

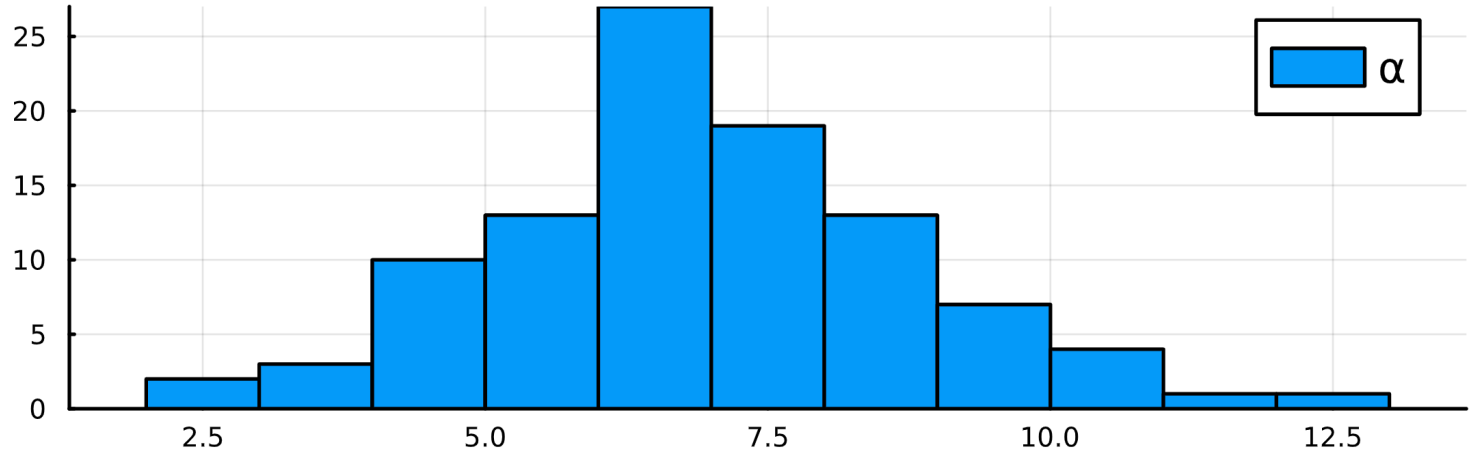
- Due by March 26, end of day

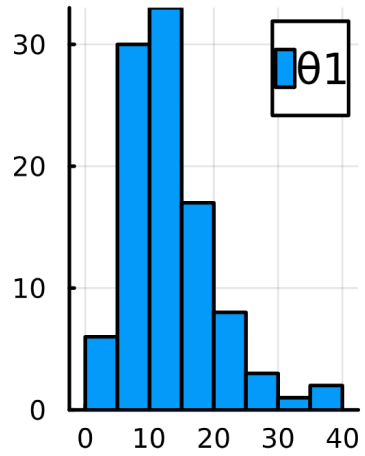
- Q1: importance sampling vs MCMC in Gen
  - Implement both
  - MCMC requires a random walk MH update
  - Hint: uniformly sample from a range centered around the previous value
  - Evaluation: average log probability after several runs

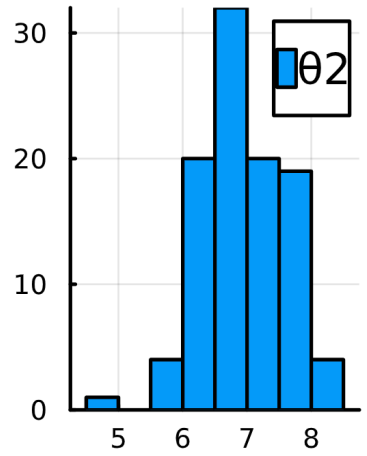
A successful inference result based on importance sampling

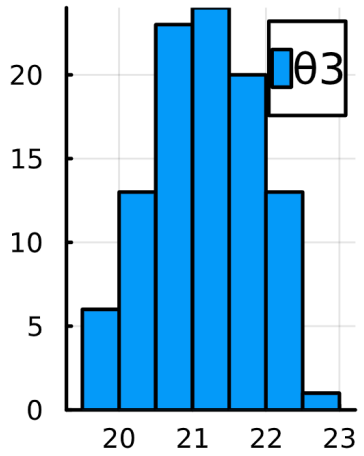
```
Average log probability: 88.14669239667002
```

Out[7]:







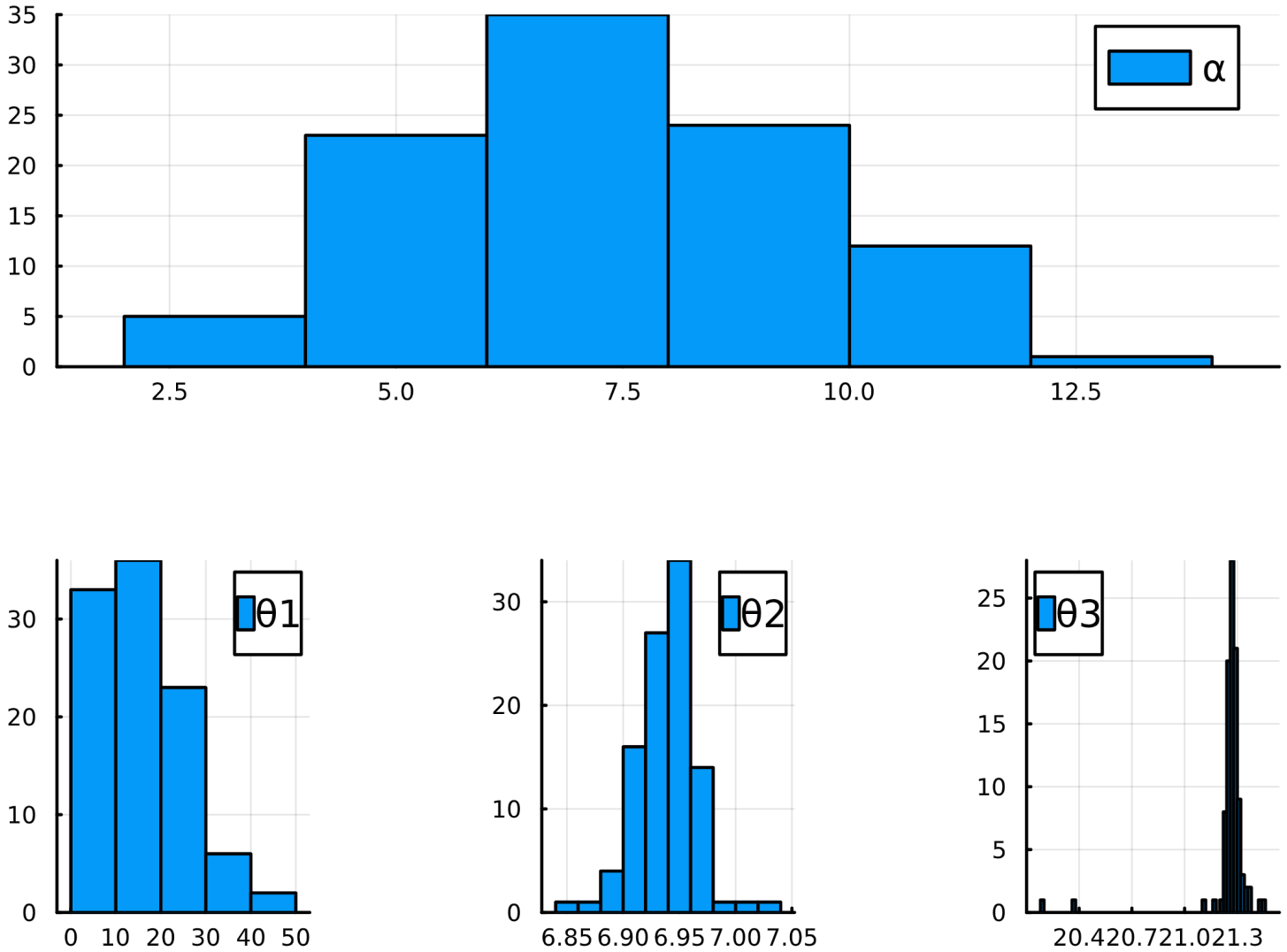


A successful inference result based on MCMC

You won’t get the exact same result, but the average log prob for MCMC should be at least around 100

```
Average log probability: 101.37401213465027
```

Out[10]:



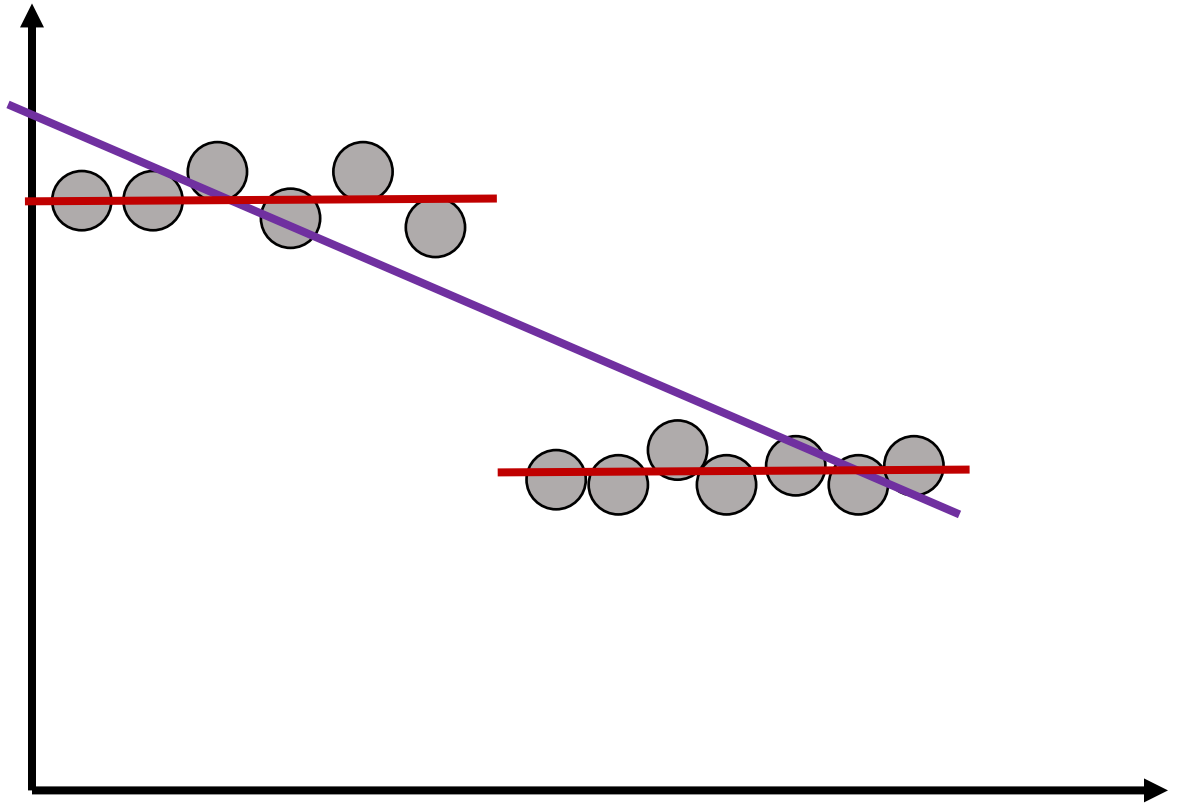
Summary

- MCMC provides an iterative update of hypotheses
- Start from an initial hypothesis (from a prior or heuristics-based/data-driven proposal)
- Update the hypothesis via Gibbs sampling or Metropolis-Hastings
- Smart proposals will help you converge to the true posterior distribution faster (mix well) – customized proposals in Gen

- Currently, we assume a fixed set parameters for each hypothesis
- How can we generalize this to hypotheses with different parameters / dimensions?

An example of hypotheses with different dimensions

- An unknown number of lines



An example of hypotheses with different dimensions

- Consider a generative model of two pieces of observed data, y1 and y2

```
@gen function model()
    if ({:z} ~ bernoulli(0.5))
        m1 = ({:m1} ~ gamma(1, 1))
        m2 = ({:m2} ~ gamma(1, 1))
    else
        m = ({:m} ~ gamma(1, 1))
        (m1, m2) = (m, m)
    end
    {:y1} ~ normal(m1, 0.1)
    {:y2} ~ normal(m2, 0.1)
end
```

Two models:

1) Same normal dist. with a mean of m
2) Two normal dist., with two means, m1, m2

Selection MH

- Selection kernel (select one of the model types):

```
select_mh_structure_kernel(trace) = mh(trace, select(:z))[1]
```

- Mean proposal kernel (proposal mean(s) given model type):

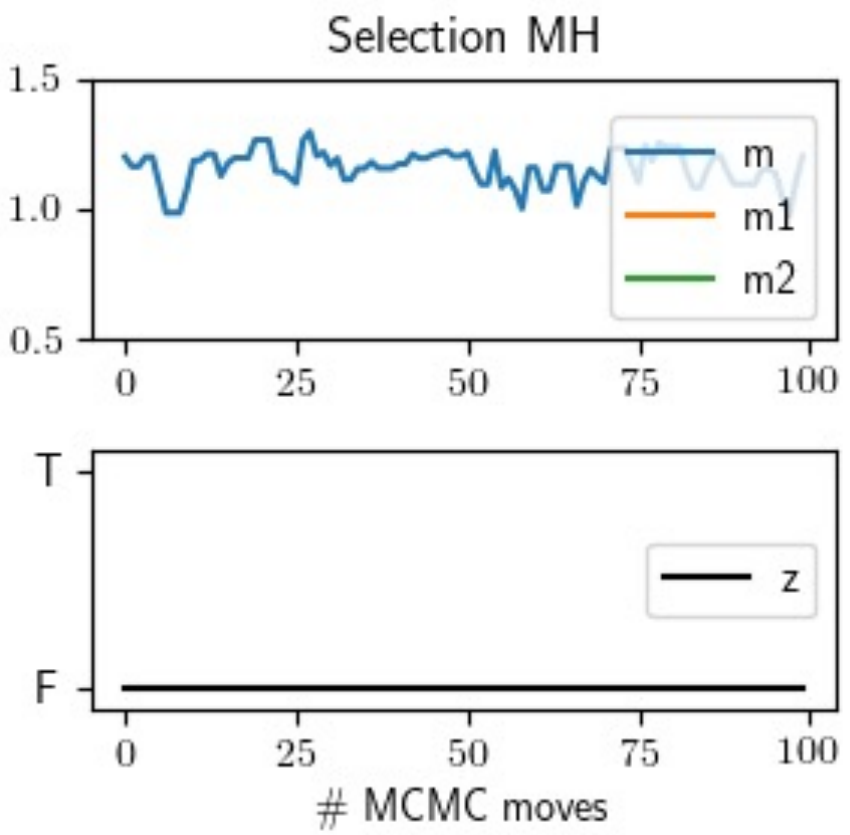
```
@gen function fixed_structure_proposal(trace)
    if trace[:z]
        {:m1} ~ normal(trace[:m1], 0.1)
        {:m2} ~ normal(trace[:m2], 0.1)
    else
        {:m} ~ normal(trace[:m], 0.1)
    end
end

fixed_structure_kernel(trace) = mh(trace, fixed_structure_proposal, ())[1]
```

- MH updates using the two kernels:

```
(y1, y2) = (1.0, 1.3)
trace, = generate(model, (), choicemap((:y1, y1), (:y2, y2), (:z, false), (:m, 1.2)))
for iter=1:100
    trace = select_mh_structure_kernel(trace)
    trace = fixed_structure_kernel(trace)
end
```

Problem: inefficient proposal sampling

- When switching from the model with a single mean to the model with two means, the values of the new addresses :m1 and :m2 will be proposed from the prior distribution (again).
- This is wasteful, sin ate value for :m, we expect the values fo alue. The same is true when proposing osite direction.
- This means it will ta accurate estimate of the posterior probab ctures.



Reversible Jump MCMC (RJMCMC)

- Standard MH: a fixed number of variables
- Different hypotheses have different dimensions
- Reversible jump MCMC is just MH, extended to moves between different hypotheses with varying dimensions, which is common for Bayesian model selection
- Key: to ensure that MH converges to the correct stationary distribution, the proposal distributions (jumps between hypotheses) must satisfy detailed balance:

$$p(h|D)q(h'|h)a(h'|h) = p(h'|D)q(h|h')a(h|h')$$

Reversible proposals

Geometric mean & ratio

(m1, m2) ⟶ (m, u)

```
function merge_means(m1, m2)
    m = sqrt(m1 * m2)
    u = m1 / (m1 + m2)
    (m, u)
end
```

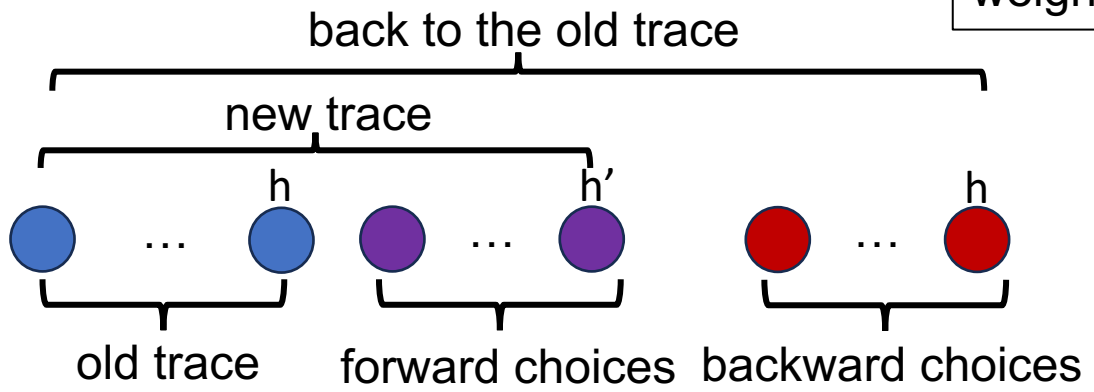
(m, u) ⟶ (m1, m2)

```
function split_mean(m, u)
    m1 = m * sqrt((u / (1 - u)))
    m2 = m * sqrt(((1 - u) / u))
    (m1, m2)
end
```

RJMCMC in Gen

Inputs:
old_trace, forward_choices
Outputs:
new_trace, backward_choices,
weight = LL(new_trace) – LL(old_trace)

- Involution functions: allow reversed jumps h ←h’



- One way to write involution function: write a Julia function f()

h→h’ (choices that lead to h’ from h)

trace of h

```
forward_choices = get_choices(simulate(proposal_randomness_func, (old_trace,)))
new_trace, backward_choices, = f(old_trace, forward_choices, ...)
```

trace of h’ h’→h (choices that lead to h from h’)

trace of h’

```
back_to_the_first_trace_maybe, backward_backward_choices, = f(new_trace, backward_choices, ...)
```

trace of h h→h’ (choices that lead to h’ from h) h’→h (choices that lead to h from h’)

```
backward_backward_choices == forward_choices
```

```
back_to_the_first_trace_maybe == old_trace
```

RJMCMC in Gen

- A simpler way to write involution function:
- Write a customized forward proposal function that only sample auxiliary variables that are not part of the generative model (e.g., u)
- Then write a transform function that implement a reversible kernel based on the model variables and the auxiliary variables. This kernel can change model variables in forward and backward directions based on the auxiliary variables

Split/merge kernel

Split a variable or merge two variables

```
@gen function split_merge_proposal(trace)
    if trace[:z]
        # currently two segments, switch to one
    else
        # currently one segment, switch to two
        {:u} ~ uniform_continuous(0, 1)
    end
end
```

Only propose auxiliary variables (variable not part of the generative model)

Split/merge involution function

A trace transform function:

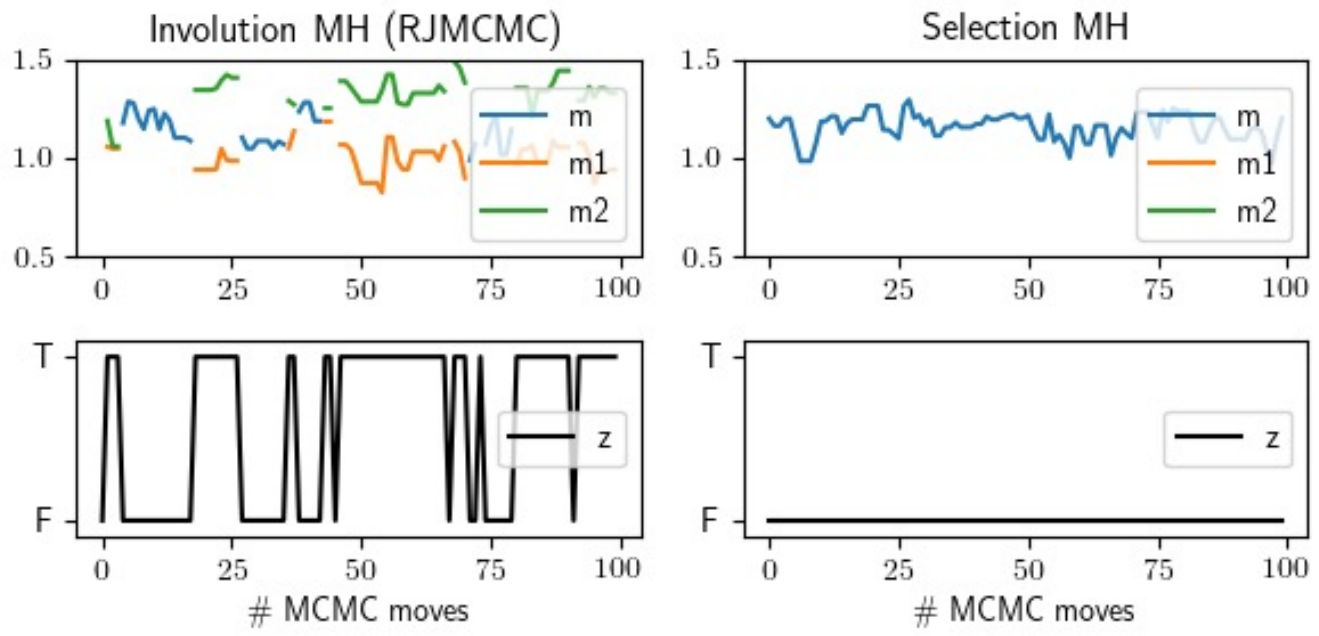
Transforms read the value of random choices from input trace(s) and write values of random choices to output trace(s)

```
@transform split_merge_involution (model_in, aux_in) to (model_out, aux_out) begin
    if @read(model_in[:z], :discrete)

        # currently two means, switch to one
        @write(model_out[:z], false, :discrete)
        m1 = @read(model_in[:m1], :continuous)
        m2 = @read(model_in[:m2], :continuous)
        (m, u) = merge_means(m1, m2)
        @write(model_out[:m], m, :continuous)
        @write(aux_out[:u], u, :continuous)
    else

        # currently one mean, switch to two
        @write(model_out[:z], true, :discrete)
        m = @read(model_in[:m], :continuous)
        u = @read(aux_in[:u], :continuous)
        (m1, m2) = split_mean(m, u)
        @write(model_out[:m1], m1, :continuous)
        @write(model_out[:m2], m2, :continuous)
    end
end
```

RJMCMC based on split/merge kernel

```
split_merge_kernel(trace) = mh(trace, split_merge_proposal, (), split_merge_involution)[1]
```

```
(y1, y2) = (1.0, 1.3)
trace, = generate(model, (), choicemap((:y1, y1), (:y2, y2), (:z, false), (:m, 1.)))
for iter=1:100
    trace = split_merge_kernel(trace)
    trace = fixed_structure_kernel(trace)
    println(trace[:z])
end
```



Sampling-based inference algorithms

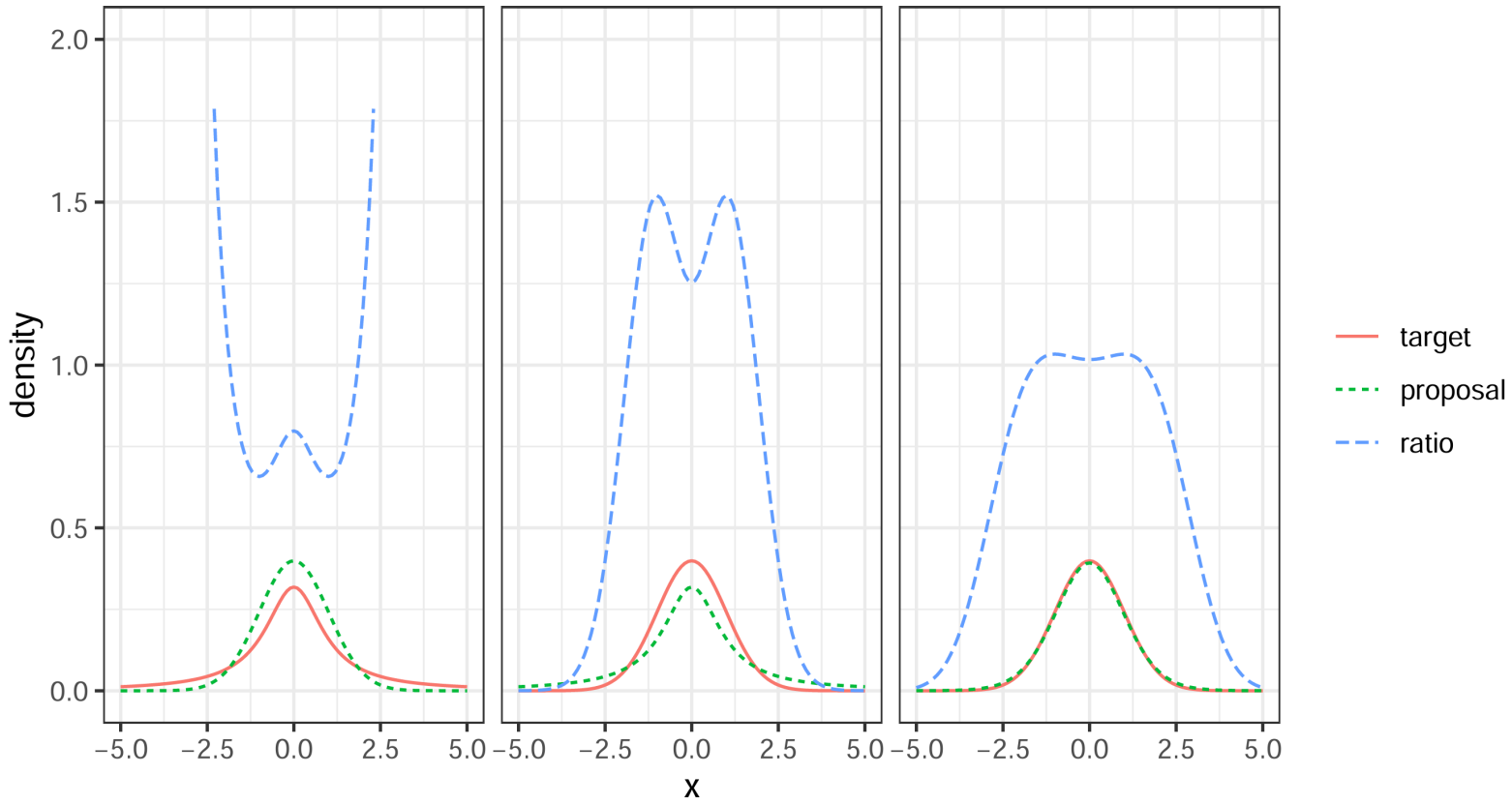
- Monte Carlo methos:

- Rejection sampling
- Importance sampling
- Markov chain Monte Carlo (MCMC)
- Sequential Monte Carlo (SMC)

How to use Gen to implement sampling-based inference algorithms?

Recap: Importance sampling

- **Goal:** approximate a *target* distribution $P(x)$, which is hard to sample from
- **Main idea:** Sample a set of particles from a simple *proposal* distribution $q(x)$ (e.g., a uniform distribution) and *weight* them to approximate the distribution
- For each particle *i*
  - Sample: $x_i \sim q(x)$
  - Weight: $w_i = \frac{p(x_i)}{q(x_i)}$
  - (x_i, w_i)



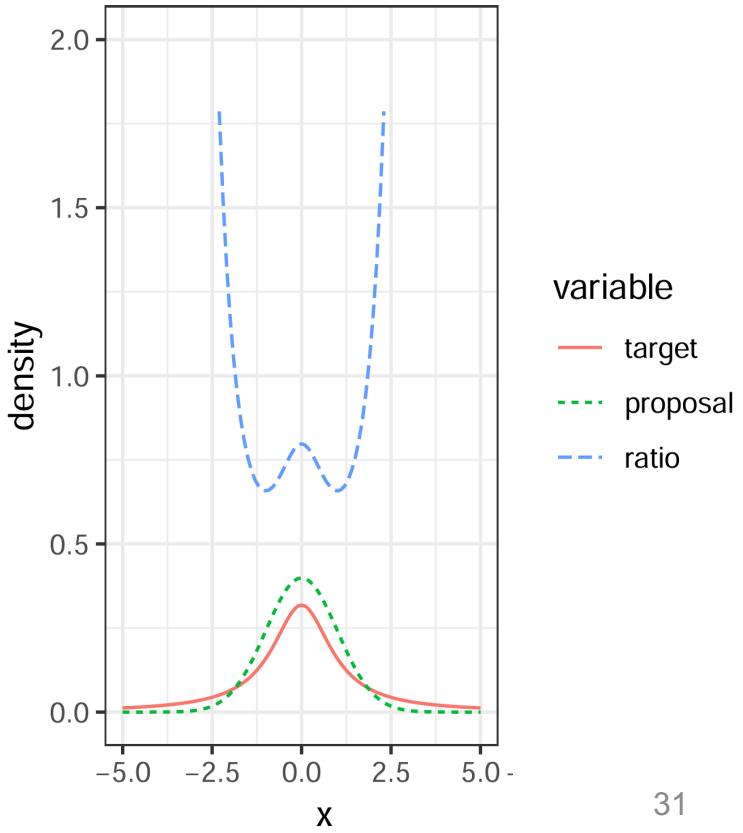
Recap: Importance resampling

- **Goal:** sample from the approximated distribution based on the particles and their weights
- Normalize the weights of particles (probability of each particle assuming the whole population is the sampled particles):

$$w_i = \frac{w_i}{\sum_j w_j}$$

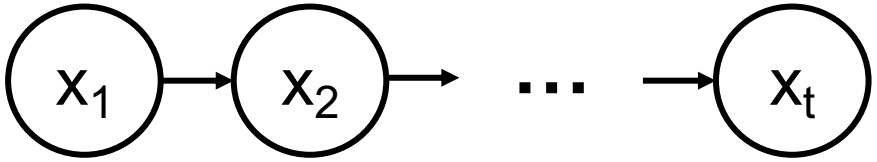
- For each new sample x,
  - Sample $j \in \{1,2,\cdots,K\}$, from with probability w
  - The new sample $x = x_j$
  - Set weight $w_i = 1/K$

Limit: proposal distribution needs to be reasonably close to the target distribution



Sequential data

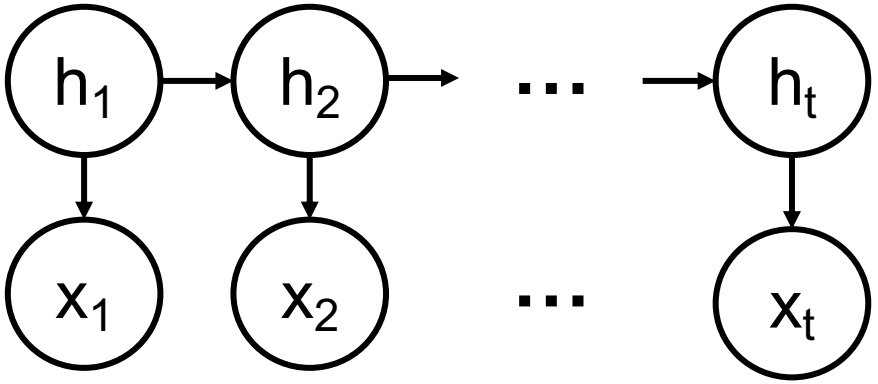
- Sample sequential data: $\{x_1, x_2, \cdots, x_t\} \sim p(x_1, x_2, \cdots, x_t)$
- Markov chain: $p(x_1, x_2, \cdots, x_t) = p(x_1) \prod_{\tau=2}^{t} p(x_\tau | x_{\tau-1})$



- Sample the initial variable: $x_1 \sim p(x_1)$
- Conditional sampling for other time steps: $x_t \sim p(x_t | x_{t-1}, \cdots, x_1)$
- **Basic idea for Sequential Monte Carlo (SMC):** For every step, we can approximate the true target distributions (the initial distribution and the conditional distribution at every step) using importance resampling

SMC for Bayesian inference

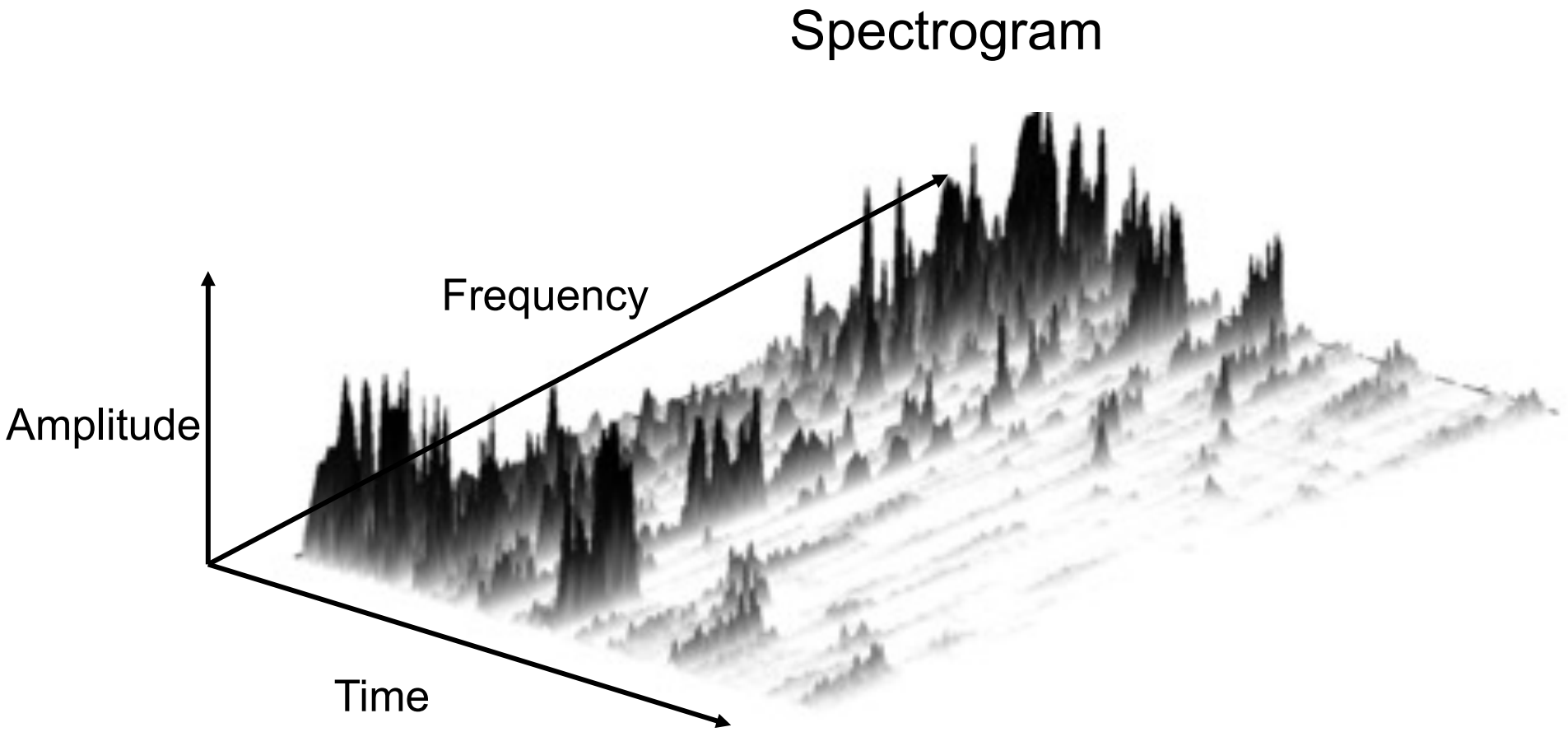
- Sample hypotheses for sequential data:
- $\{h_1, h_2, \cdots, h_t\} \sim p(h_1, h_2, \cdots, h_t | x_1, x_2, \cdots, x_t)$
- A Bayesian network:



- Sample the initial variable: $h_1 \sim p(h_1 | x_1)$
- Conditional sampling for other time steps: $h_t \sim p(h_t | h_{t-1}, x_1, \cdots, x_t)$

SMC in Gen

- Example 1: Music notes



SMC in Gen

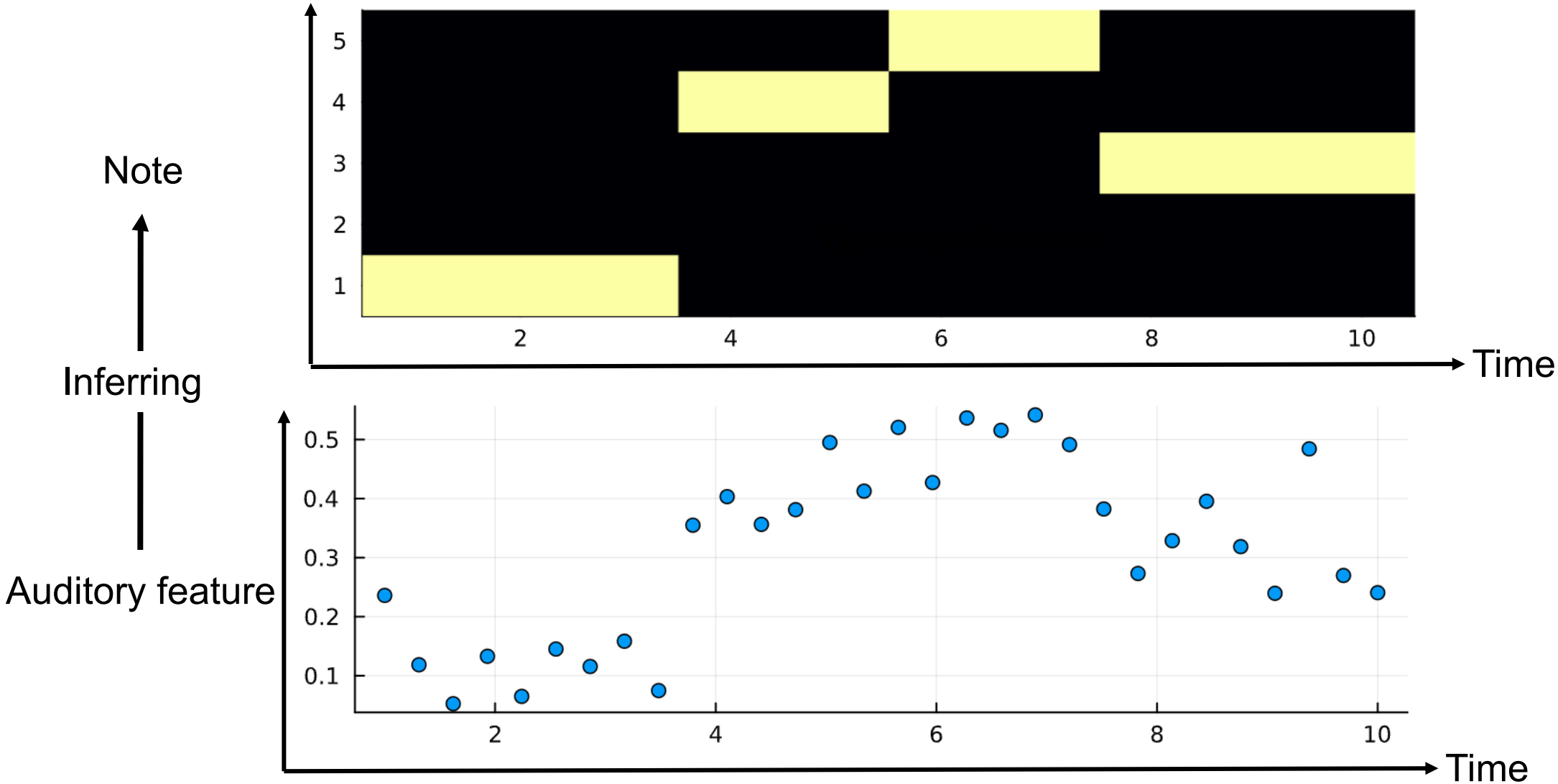
Why SMC instead of MCMC?

Temporal dependency

- Example 1: Music notes

$p(h_1, h_2, \cdots, h_t | x_1, x_2, \cdots, x_t)$

A simplified model



SMC

- Step 1: Initialization
  - Set $t = 1$
  - For $i \in \{1, \cdots, K\}$, sample $h_1^{(i)} \sim q(h_1)$ initial proposals
  - Weight $w_1^{(i)} = \frac{p\left(x_1 | h_1^{(i)}\right) p(h_1^{(i)})}{q(h_1^{(i)})}$
  - Normalize all weights → particles $\left\{\left(h_1^{(i)}, w_1^{(i)}\right)\right\}_{i=1}^{K}$
- Step 2: Resampling at step t
  - If the particles are not diverse enough, resample the particles, and set $w_t^{(i)} = \frac{1}{K}$ (importance resampling)
- Step 3: Sampling at step $t + 1$

hypotheses at all previous steps

  - Set $t = t + 1$; particles at previous step $\left\{\left(h_{1:t-1}^{(i)}, w_{t-1}^{(i)}\right)\right\}_{i=1}^{K}$
  - For $i \in \{1, \cdots, K\}$, sample $h_t^{(i)} \sim p\left(h_t | h_{t-1}^{(i)}\right)$ conditional distribution
  - Weight $w_t^{(i)} = w_{t-1}^{(i)} p\left(x_t \middle| h_t^{(i)}\right)$ reweighted by the current likelihood
  - Normalize all weights → particles $\left\{\left(h_{1:t}^{(i)}, w_t^{(i)}\right)\right\}_{i=1}^{K}$

When to resample?

- Effective sample size (ESS)

$$ESS = \left(\sum_{i=1}^{K} \left(w^{(j)} \right)^2 \right)^{-1}$$

- Large sample size: particles share similar weights
- Small sample size: only a small number of particles have large weights

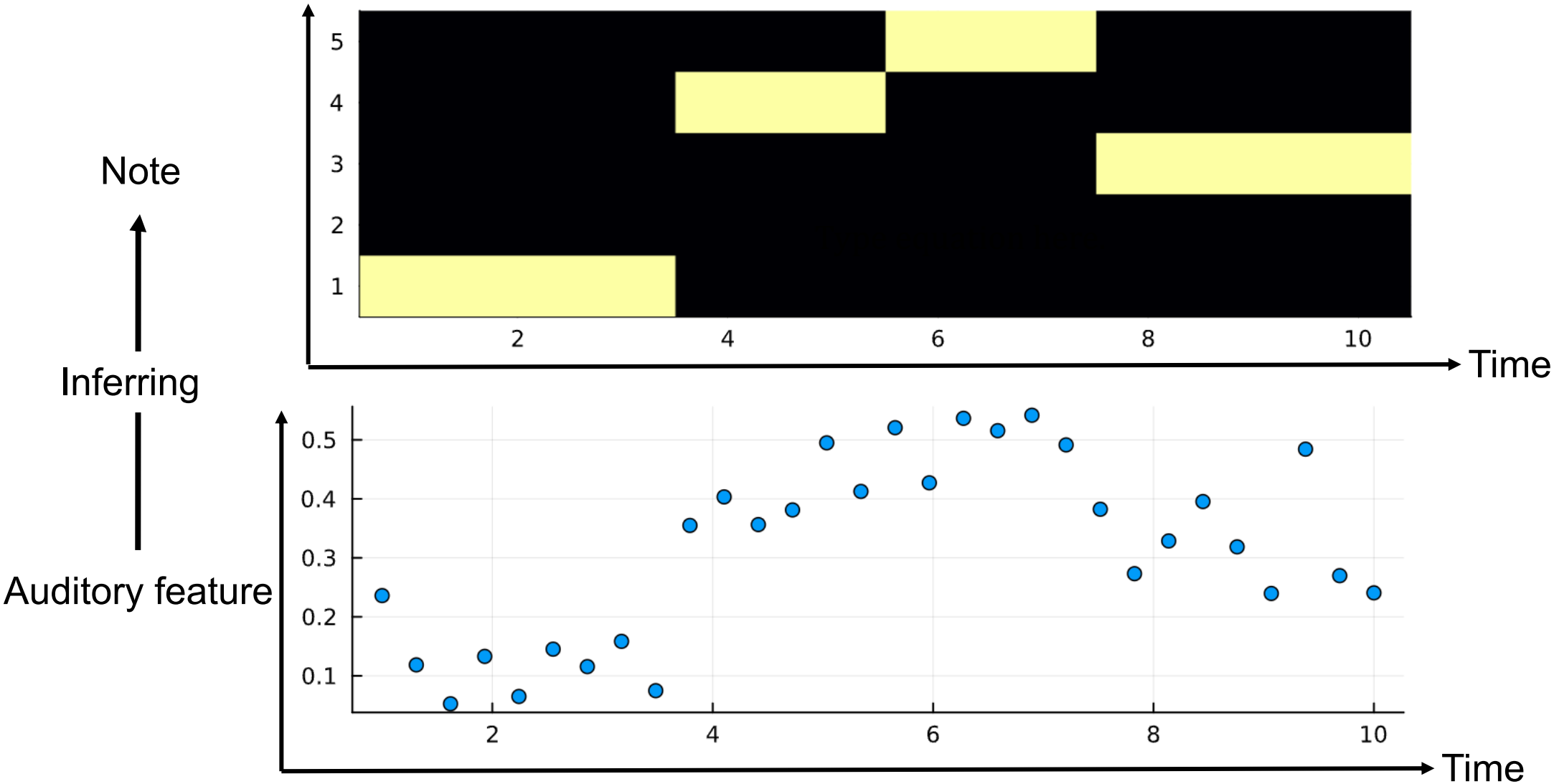
SMC

- Step 1: Initialization
  - Set $t = 1$
  - For $i \in \{1, \cdots, K\}$, sample $h_1^{(i)} \sim q(h_1)$ initial proposals
  - Weight $w_1^{(i)} = \frac{p\left(x_1 | h_1^{(i)}\right) p(h_1^{(i)})}{q(h_1^{(i)})}$
  - Normalize all weights → particles $\left\{\left(h_1^{(i)}, w_1^{(i)}\right)\right\}_{i=1}^{K}$
- Step 2: Resampling at step t
  - If ESS < threshold, resample the particles, and set $w_t^{(i)} = \frac{1}{K}$ (importance resampling)
- Step 3: Sampling at step $t + 1$
  - Set $t = t + 1$; particles at previous step $\left\{\left(h_{1:t-1}^{(i)}, w_{t-1}^{(i)}\right)\right\}_{i=1}^{K}$
  - For $i \in \{1, \cdots, K\}$, sample $h_t^{(i)} \sim p\left(h_t | h_{t-1}^{(i)}\right)$ conditional distribution
  - Weight $w_t^{(i)} = w_{t-1}^{(i)} p\left(x_t \middle| h_t^{(i)}\right)$ reweighted by the current likelihood
  - Normalize all weights → particles $\left\{\left(h_{1:t}^{(i)}, w_t^{(i)}\right)\right\}_{i=1}^{K}$

SMC in Gen (particle filtering)

- Example 1: Music notes

A simplified model



Generative model – transitions between notes $p(h_t|h_{t-1})$

- 5 musical notes, a composition is defined by a set of stochastic transitions between notes
- With a probability of 0.6, we stay on the current note; with a probability of 0.1, we transition to one of the new notes

```
function transition_prob(a, b)
    a == b ? 0.6 : 0.1
end

# setup transition probabilities
function transition_prob_matrix()
    note_transitions = Array{Float64}(undef, 5, 5)
    for i=1:5
        [note_transitions[i, j] = transition_prob(i,j) for j=1:5]
    end
    return note_transitions
end

# make this variable global (makes some of the coding more straightforward)
global note_transitions = transition_prob_matrix()
```

Generative model – notes & auditory features $p(x_{1:t}, h_{1:t})$

```
@gen function musical_notes(K::Int)
    # sample an initial "note"
    current_note = {:notes => 1 => :θ} ~ uniform_discrete(1, 5)

    # "play" it: projecting it to "auditory features" of length 3
    note_play_length = 3
    mu = current_note * 0.1
    {:data => 1 => :y} ~ broadcasted_normal(repeat([mu], note_play_length), repeat([0.05], note_play_length))

    # keep going for K notes
    for k=1:K
        current_note = {:notes => k+1 => :θ} ~ categorical(note_transitions[current_note, :])
        mu = current_note * 0.1
        {:data => k+1 => :y} ~ broadcasted_normal(repeat([mu], note_play_length), repeat([0.05], note_play_length))
    end
end
```

See jupyter notebook