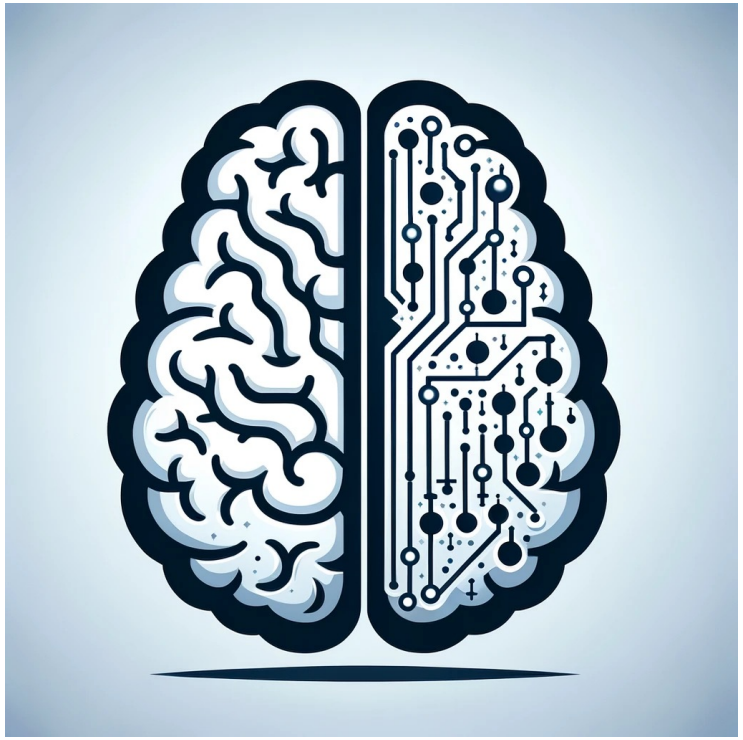
EN 601.473/601.673: Cognitive Artificial Intelligence (CogAI)



Lecture 10: Metropolis-Hastings, MCMC in Gen

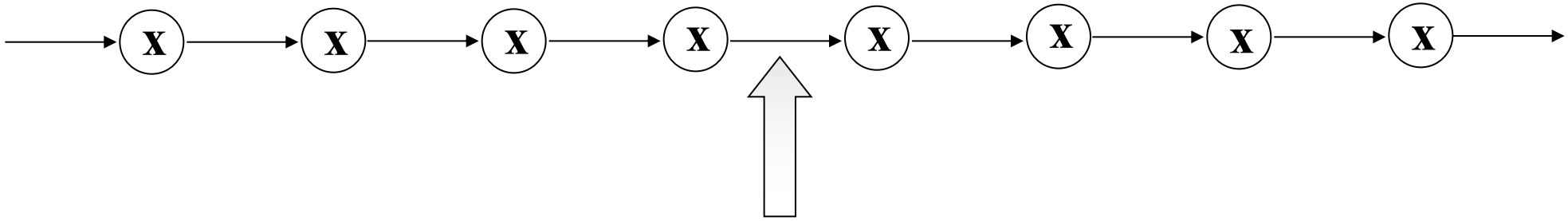
Tianmin Shu

Markov chain Monte Carlo

- Markov chains
- Gibbs sampling
- Metropolis-Hastings

Markov chain

- A random process that generate a sequence of states



Transition kernel

$$k(\boldsymbol{x} \to \boldsymbol{x}')$$

The chain has reached its stationary distribution if $\pi_t = \pi_{t+1}$

The defining equation of stationary distribution

$$\pi(\boldsymbol{x}') = \sum_{\boldsymbol{x}} \pi(\boldsymbol{x}) k(\boldsymbol{x} \to \boldsymbol{x}')$$

Detailed balance:

$$\pi(\boldsymbol{x}) k(\boldsymbol{x} \to \boldsymbol{x}') = \pi(\boldsymbol{x}') k(\boldsymbol{x}' \to \boldsymbol{x})$$

Markov chain Monte Carlo

- Markov chains
- Gibbs sampling
- Metropolis-Hastings

Gibbs sampling

- Directly sample from a joint probability $P(x_1, x_2, \cdots, x_n)$ is hard
- Much easier to sample from the conditional probability:

$$x_i \sim P(x_i | x_{-i}), \forall i = 1, \cdots, n$$

$$x_{-i}: excluding\ x_i$$

- Gibbs sampling:
- Start from a random value for each variable
- One **step** in Gibbs sampling: sampling one variable

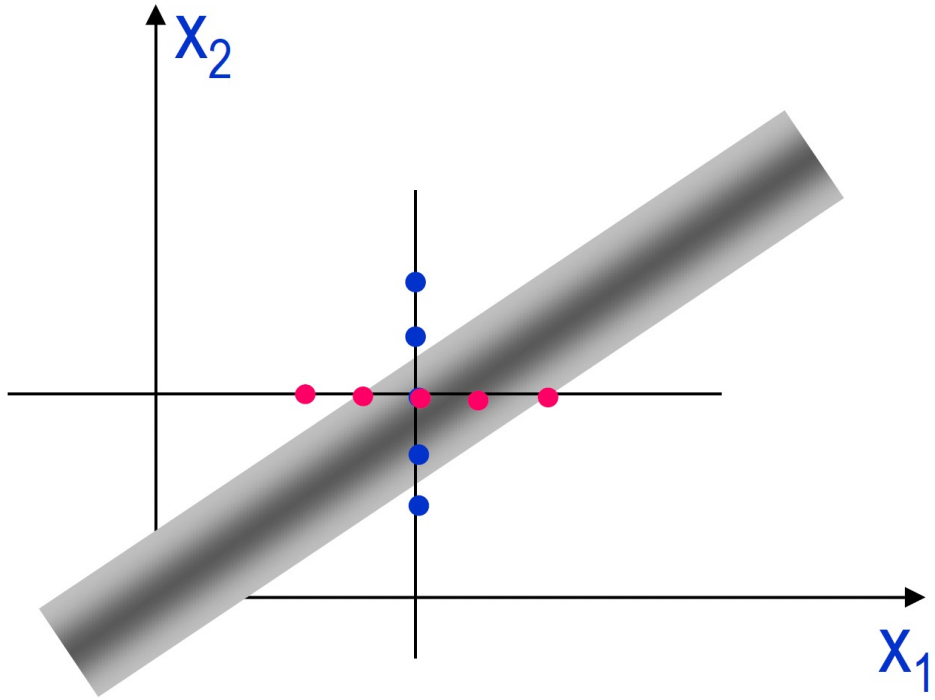
$$(x_1, \cdots, x_i, \cdots, x_n) \rightarrow (x_1, \cdots, x_i', \cdots, x_n)$$

- A **sweep** in Gibbs sampling: sampling every variable once

A major problem with Gibbs sampling

- For a joint probability whose probability mass is focused on a 1D line segment, sampling two 1D variables iteratively is inefficient, i.e., the chain is “jagging”

This is because the two variables are ***tightly coupled***. It is best if we move along the direction of the line.



Markov chain Monte Carlo

- Markov chains
- Gibbs sampling
- Metropolis-Hastings

Motivation for Metropolis-Hastings

- Suppose we can compute likelihood $p(d|h)$ and $p(h)$, but not $p(h|d)$:

$$p(h|d) = \frac{p(d|h)p(h)}{Z}$$

- We can compute relative posteriors:

$$\frac{p(h|d)}{p(h'|d)} = \frac{p(d|h)p(h)}{p(d|h')p(h')}$$

- Compared to Gibbs sampling, we want to consider global moves that change multiple state variables at once

Metropolis-Hastings algorithm

- Starting from a random value
- Each step has two stages
- 1. Sample a new state w.r.t. **proposal distribution**:

$$\boldsymbol{x}' \sim q(\boldsymbol{x}'|\boldsymbol{x})$$

- 2. Accept or reject the proposed new state $\boldsymbol{x}'$ w.r.t. the **acceptance probability**:

$$a(\boldsymbol{x}'|\boldsymbol{x}) = \min(1, \frac{\pi(\boldsymbol{x})q(\boldsymbol{x}|\boldsymbol{x}')}{\pi(\boldsymbol{x}')q(\boldsymbol{x}'|\boldsymbol{x})})$$

  - If rejected, the state remains at $\boldsymbol{x}$; otherwise, the state becomes $\boldsymbol{x}'$

$$k(\boldsymbol{x} \to \boldsymbol{x}') = q(\boldsymbol{x}'|\boldsymbol{x})a(\boldsymbol{x}'|\boldsymbol{x})$$

$$(\boldsymbol{x} \neq \boldsymbol{x}')$$

Metropolis-Hastings algorithm

- MH converges to the ***correct stationary distribution*** if the transition kernel is ergodic

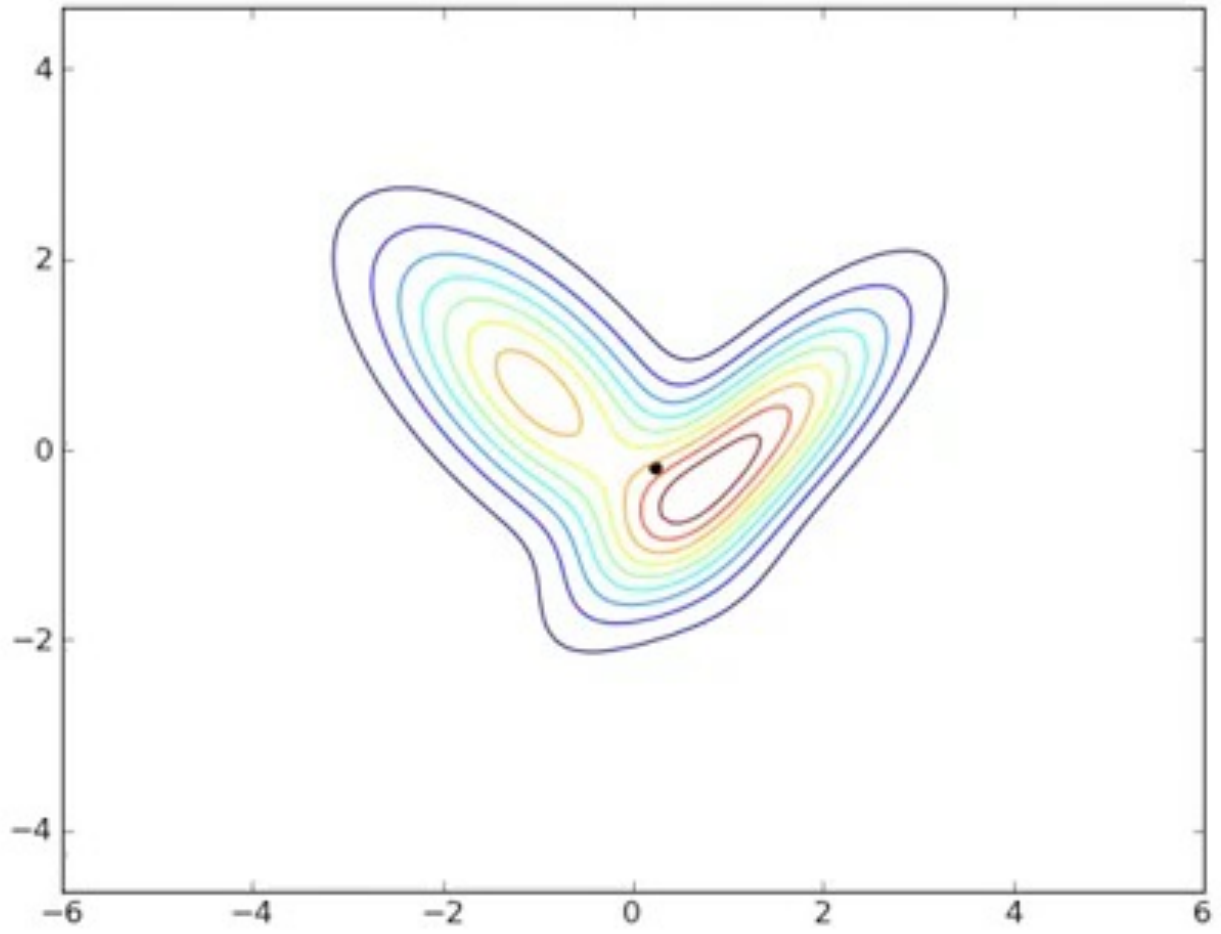
- Proof: detailed balance

$$\pi(\boldsymbol{x})q(\boldsymbol{x}'|\boldsymbol{x})a(\boldsymbol{x}'|\boldsymbol{x}) = \pi(\boldsymbol{x}')q(\boldsymbol{x}|\boldsymbol{x}')a(\boldsymbol{x}|\boldsymbol{x}')$$

- Excise for you: show why Gibbs sampling works

An animated example

- Becoming worse before becoming better → get out of the local minima
- Good proposal distribution allows effective moves

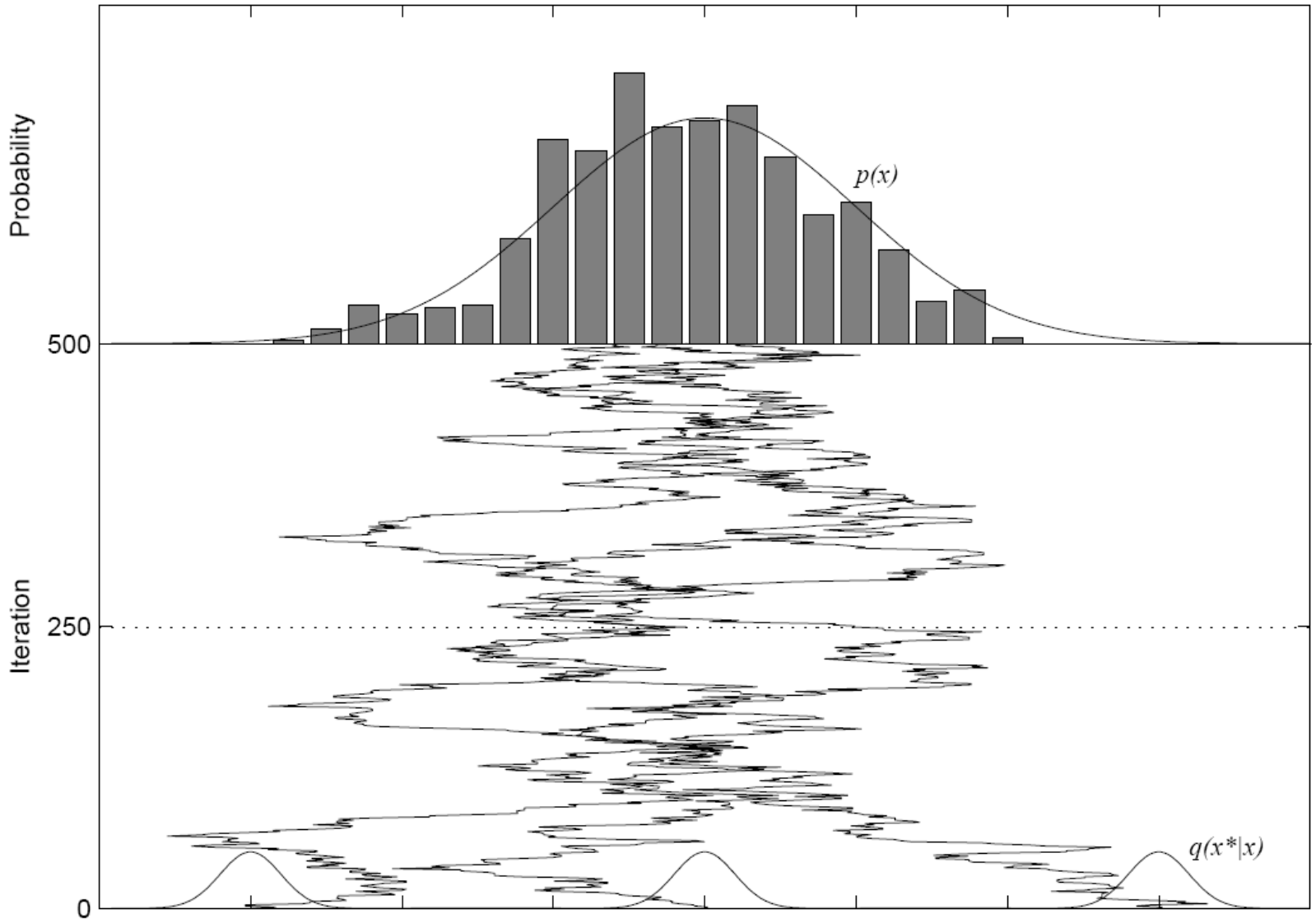


https://www.youtube.com/watch?v=4I6TaYo9j_Y

Burn in

Early dependence on initial state, but chains very similar after enough samples…

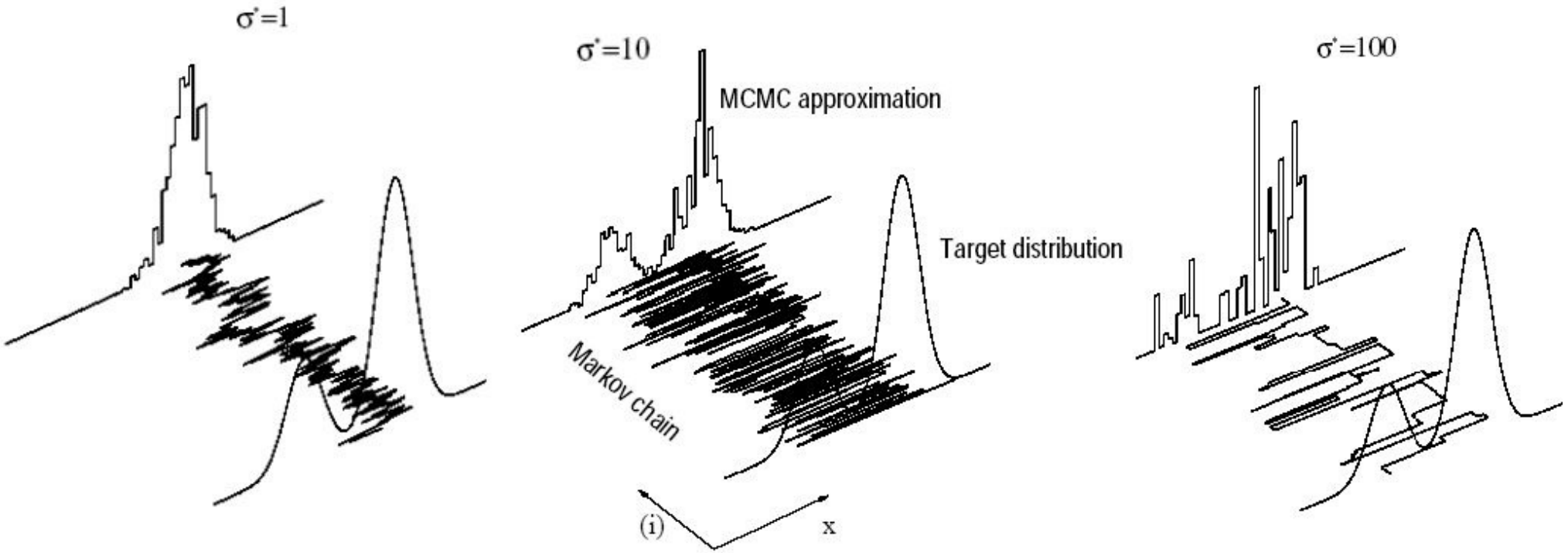
p(x): target distribution



q(x'|x): a gaussian distribution

Mixing

- **Mixing time:** time needed to reach the target distribution
- Effect of the std. dev. of Gaussian proposal distribution



Metropolis-Hastings for Bayesian inference

- Sample a new hypothesis

$$h' \sim q(h'|h)$$

  - Data-driven MCMC, data-driven proposal distribution:

$$h' \sim q(h'|h, d = D)$$

- Acceptance probability:

$$a(h'|h) = \min(1, \frac{p(h|D)q(h|h')}{p(h'|D)q(h'|h)})$$

Easy to compute:

$$\frac{p(h|D)}{p(h'|D)} = \frac{p(D|h)p(h)}{p(D|h')p(h')}$$

No normalization term Z

The power of MCMC

- Simple to compute
- Since we only ever need to evaluate the relative probabilities of two states, we can have very large state spaces (much of which we rarely reach)
- In fact, our state spaces can be *infinite*
  - common with nonparametric Bayesian models
- State spaces can be implicitly defined, with an infinite number of states we've never seen or imagined …
  - natural with probabilistic programs, and program induction (example for real world data modeling)
- But the guarantees it provides are asymptotic
  - making algorithms that converge in practical amounts of time is a significant challenge

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)
```

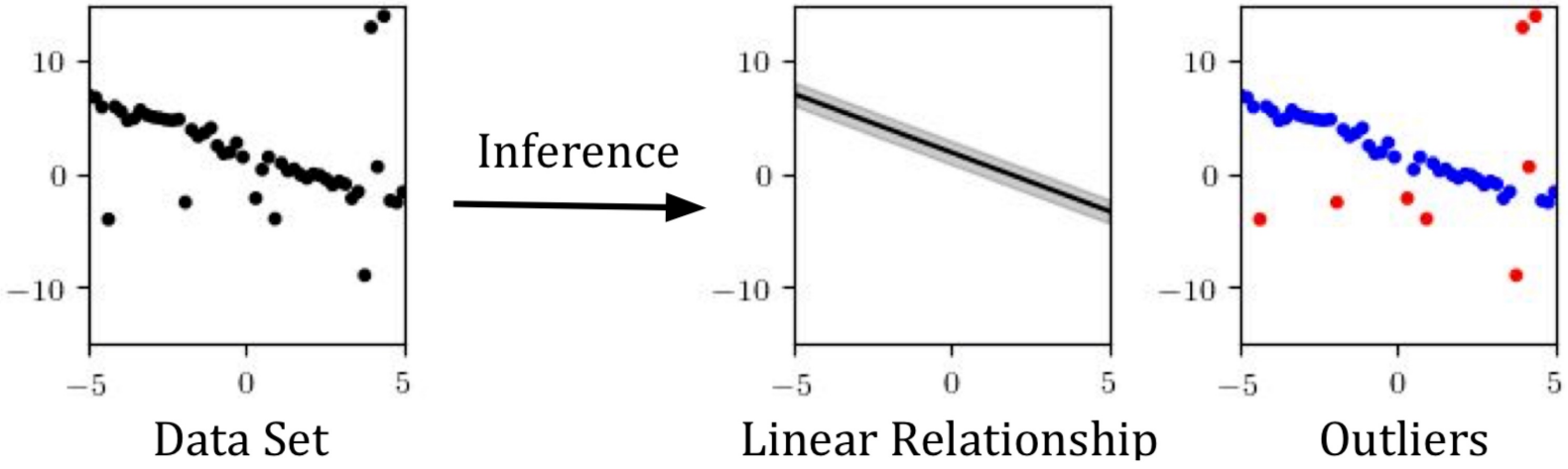
- Update *selected* variables in the trace (e.g., using Metropolis-Hasting)

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

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

MCMC in Gen

- Example: Bayesian curve-fitting with outliers
- Suppose we have a dataset of points in the x,y plane that is *mostly* explained by a linear relationship, but which also has several outliers. Our goal will be to automatically identify the outliers, and to find a linear relationship (a slope and intercept, as well as an inherent noise level) that explains rest of the points:



Writing the generative model

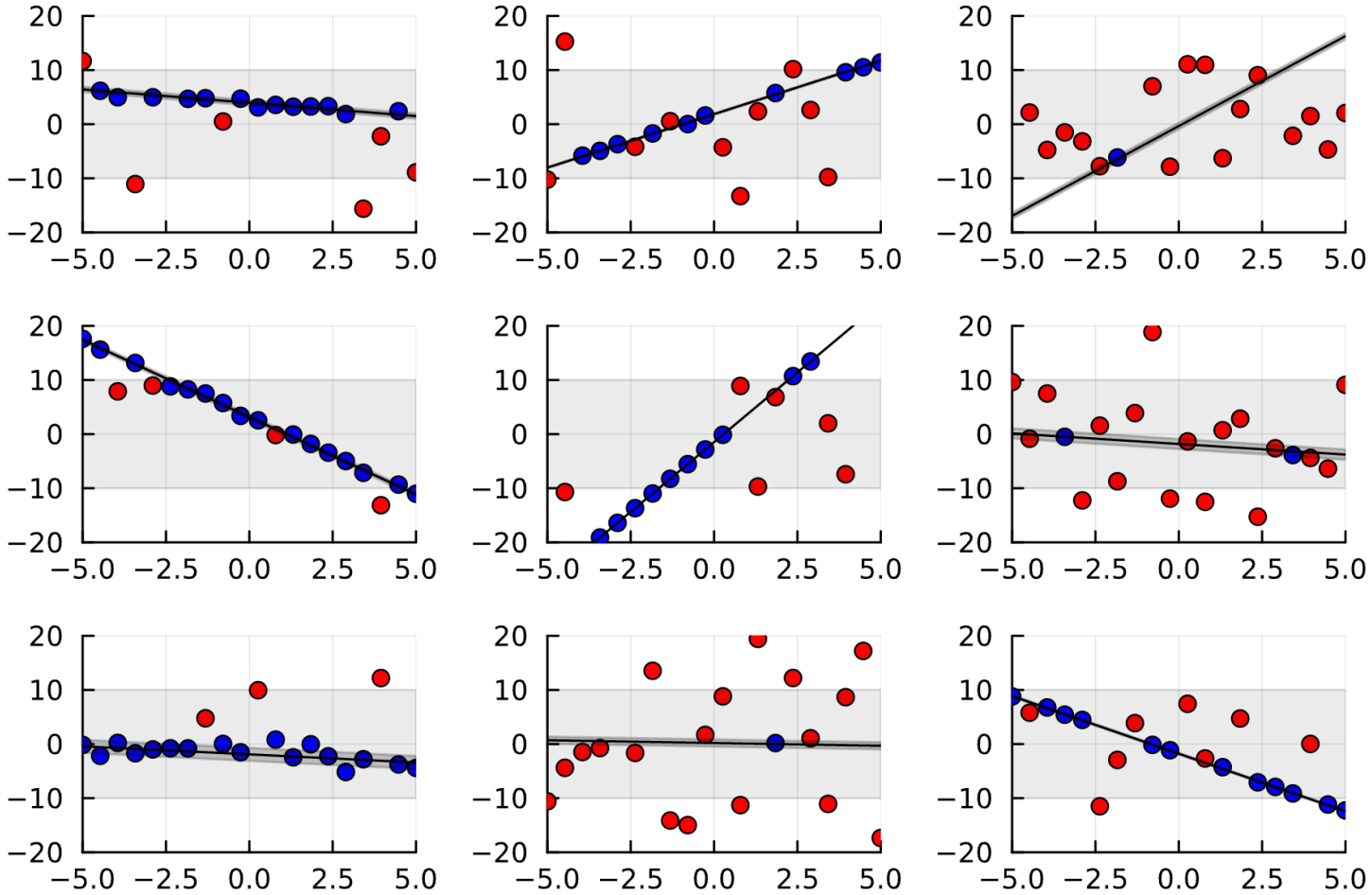
```
@gen function regression_with_outliers(xs::Vector{<:Real})
    # First, generate some parameters of the model. We make these
    # random choices, because later, we will want to infer them
    # from data. The distributions we use here express our assumptions
    # about the parameters: we think the slope and intercept won't be
    # too far from 0; that the noise is relatively small; and that
    # the proportion of the dataset that don't fit a linear relationship
    # (outliers) could be anything between 0 and 1.
    slope ~ normal(0, 2)
    intercept ~ normal(0, 2)
    noise ~ gamma(1, 1)
    prob_outlier ~ uniform(0, 1)

    # Next, we generate the observable y coordinates (outcome variable in linear regression).
    n = length(xs)
    ys = Float64[]

    for i = 1:n
        # Decide whether this point is an outlier, and set
        # mean and standard deviation accordingly
        if ({:data => i => :is_outlier} ~ bernoulli(prob_outlier))
            (mu, std) = (0., 10.)
        else
            (mu, std) = (xs[i] * slope + intercept, noise)
        end
        # Sample a y value for this point
        push!(ys, {:data => i => :y} ~ normal(mu, std))
    end
    ys
end;
```

What the model is doing: visualizing the prior

```
# Generate nine traces and visualize them
include("visualization/regression_viz.jl")
xs     = collect(range(-5, stop=5, length=20))
traces = [Gen.simulate(regression_with_outliers, (xs,)) for i in 1:9];
Plots.plot([visualize_trace(t) for t in traces]...)
```

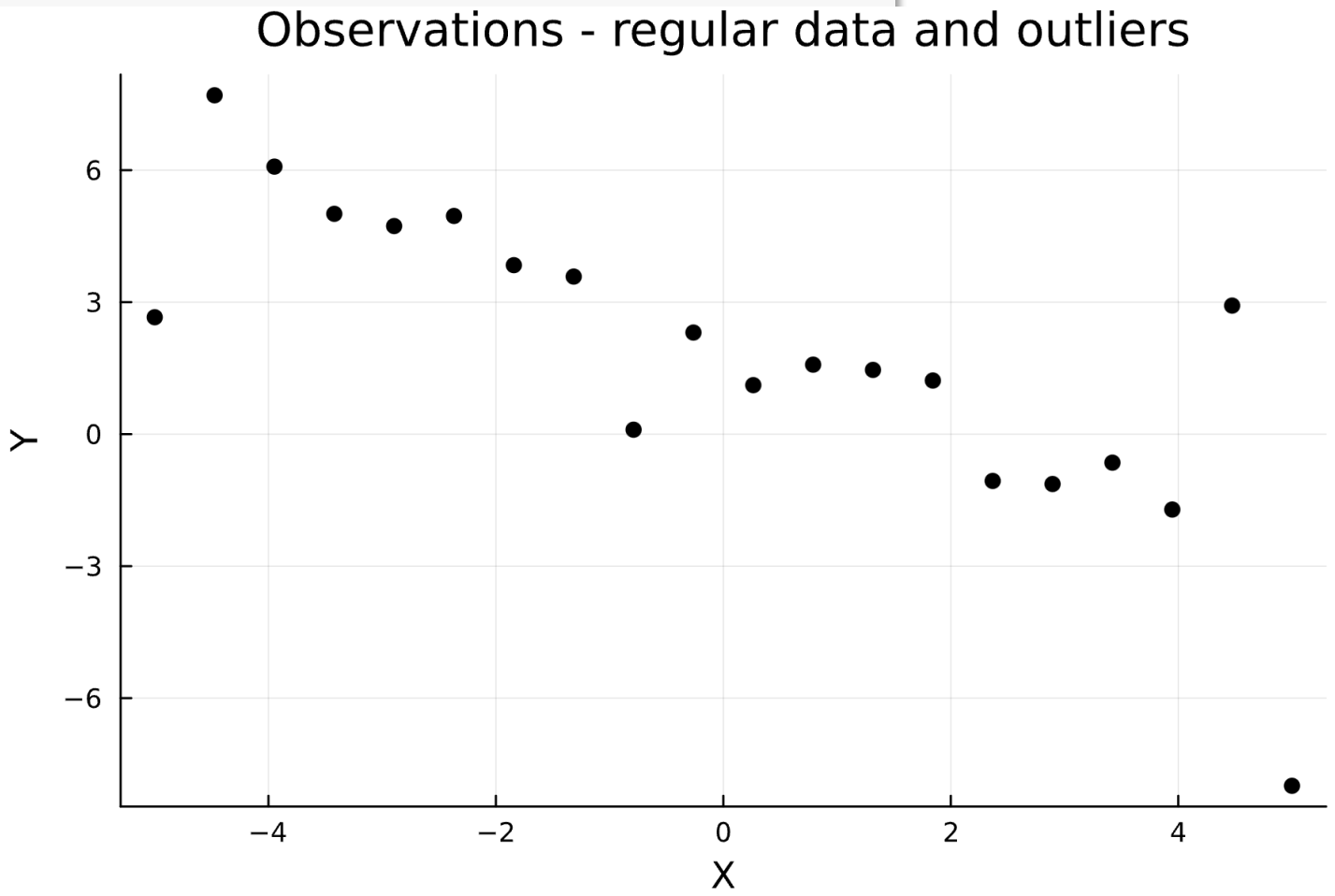
Based on a customized visualization function

- Sample a line
- Sample outliers
- For non-outliers, sample y w.r.t. the line + noise



Make a synthetic dataset

```
function make_synthetic_dataset(n)
    # setting the random seed ensures that the dataset is replicable
    Random.seed!(1)
    prob_outlier = 0.2
    true_inlier_noise = 0.5
    true_outlier_noise = 5.0
    true_slope = -1
    true_intercept = 2
    xs = collect(range(-5, stop=5,
    ys = Float64[]
    for (i, x) in enumerate(xs)
        if rand() < prob_outlier
            y = randn() * true_out
        else
            y = true_slope * x + t
        end
        push!(ys, y)
    end
    (xs, ys)
end

(xs, ys) = make_synthetic_dataset(
Plots.scatter(xs, ys, color="black
              label=nothing, title
```



Make constraints based on the observed data

- Express our observations as a **ChoiceMap** that constrains the values of certain random choices to their observed values
- In this case, constrain the values of the choices with address

:data => i => :y (i.e., the y values of the data points in the dataset)

```
function make_constraints(ys::Vector{Float64})
    constraints = Gen.choicemap()
    for i=1:length(ys)
        constraints[:data => i => :y] = ys[i]
    end
    constraints
end;
```

Apply this to our dataset to make a set of constraints for inference

```
observations = make_constraints(ys);
```

Importance sampling for inference

- Run importance sampling 9 times

```
traces    = [first(Gen.importance_resampling(regression_with_outliers, (xs,), observations, 2000)) for i in 1:9]
log_probs = [get_score(t) for t in traces]
println("Average log probability: $(logmeanexp(log_probs))")
Plots.plot([visualize_trace(t) for t in traces]...)
```

$$\sum_i \log P(y_i | x_i, \theta)$$

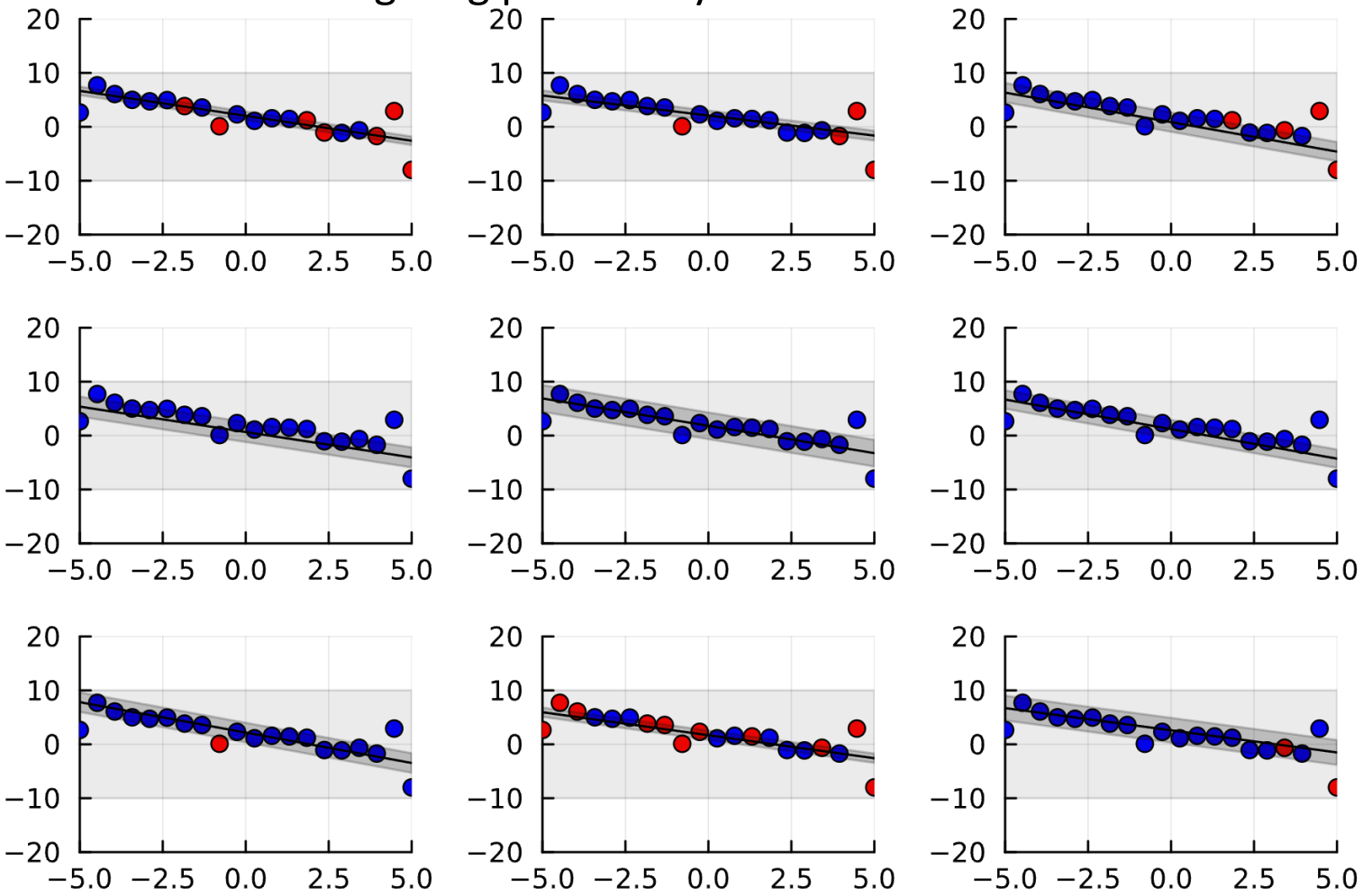
Compute the average log likelihood over multiple runs
(the fitness of the inferred curves)

```
function logmeanexp(scores)
    logsumexp(scores) - log(length(scores))
end;
```

Importance sampling for inference

- Run importance sampling 9 times

Average log probability: -51.8570956904098



Problems:

- There is little logic to the outlier classification
- The inferred noise around the line is too wide.

Limit of importance sampling:
There are just too many variables to get right, and so sampling everything **in one go** is highly unlikely to produce a perfect hit.

Iterative update via MCMC!

Block resimulation MH

- Update some variables conditioned on the remaining variables
- A more flexible version of Gibbs sampling

```
for iter=1:T
    # block 1
    tr = mh(tr, select(:variable_1, :variable_2))
    # block 2
    tr = mh(tr, select(:variable_3))
     ...
    # block n
    tr = mh(tr, select(:variable_K))
end
```

Block resimulation MH

Block 1: slope, intercept, and noise. These parameters determine the linear relationship; resimulating them is like picking a new line. Before too long, we're bound to sample something close to the right line.

Blocks 2 through N+1: Each is outlier, in its own block. One problem with importance sampling was that it tried to sample *every* outlier classification at once. Here, we can choose to resimulate each is outlier choice separately, and for each one, decide whether to use the resimulated value or not.

Block N+2: prob outlier. Finally, we can propose a new prob outlier value; in general, we can expect to accept the proposal when it is in line with the current hypothesized proportion of is outlier choices that are set to true.

```
# Perform a single block resimulation update of a trace.
function block_resimulation_update(tr)
    # Block 1: Update the line's parameters
    line_params = select(:noise, :slope, :intercept)
    (tr, _) = mh(tr, line_params)

    # Blocks 2-N+1: Update the outlier classifications
    (xs,) = get_args(tr)
    n = length(xs)
    for i=1:n
        (tr, _) = mh(tr, select(:data => i => :is_outlier))
    end

    # Block N+2: Update the prob_outlier parameter
    (tr, _) = mh(tr, select(:prob_outlier))

    # Return the updated trace
    tr
end;
```

Inference based on block resimulation MH

- Generate the initial trace constrained on the observations
- Update the trace via 500 iterations of MH

```
function block_resimulation_inference(xs, ys, observations)
    observations = make_constraints(ys)
    (tr, _) = generate(regression_with_outliers, (xs,), observations)
    for iter=1:500
        tr = block_resimulation_update(tr)
    end
    tr
end;
```

Test it

Block resimulation MH

```
scores = Vector{Float64}(undef, 10)
for i=1:10
    @time tr = block_resimulation_inference(xs, ys, observations)
    scores[i] = get_score(tr)
end
println("Log probability: ", logmeanexp(scores))
```

```
  1.343702 seconds (11.47 M allocations: 676.832 MiB, 4.73% gc 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
```

IS (with a similar amount of running time)

```
scores = Vector{Float64}(undef, 10)
for i=1:10
    @time (tr, _) = importance_resampling(regression_with_outliers, (xs,), observations, 17000)
    scores[i] = get_score(tr)
end
println("Log probability: ", logmeanexp(scores))
```

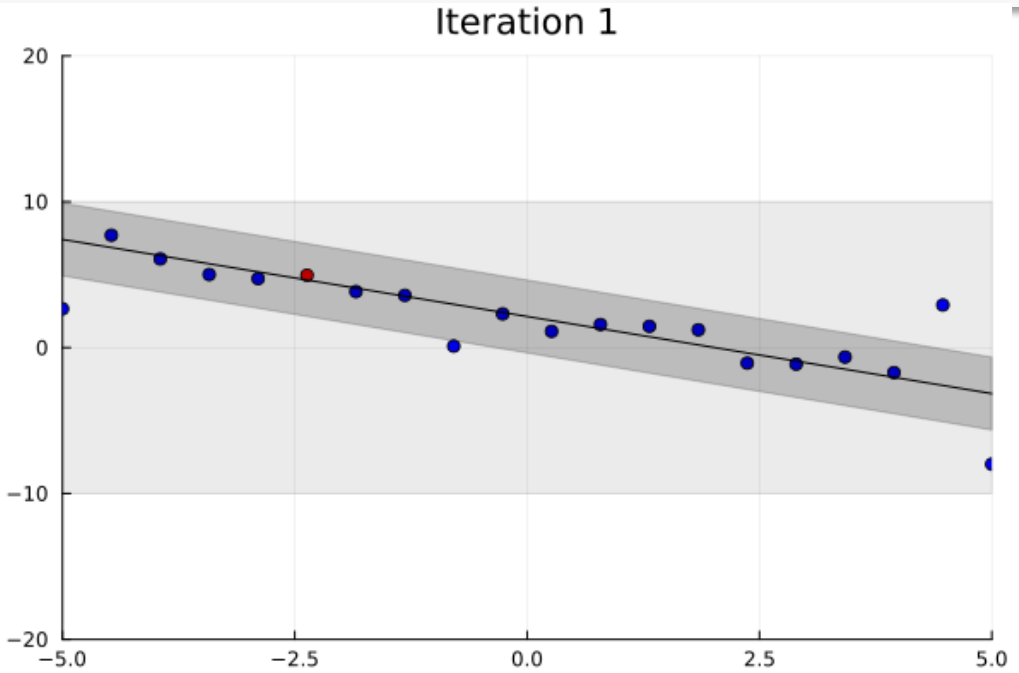
```
  0.766331 seconds (12.92 M allocations: 894.409 MiB, 7.85% gc time)
  0.849067 seconds (12.92 M allocations: 894.409 MiB, 6.97% gc time)
  0.777396 seconds (12.92 M allocations: 894.409 MiB, 8.40% gc time)
  0.803380 seconds (12.92 M allocations: 894.409 MiB, 8.25% gc time)
  0.953001 seconds (12.92 M allocations: 894.409 MiB, 7.85% gc time)
  0.999087 seconds (12.92 M allocations: 894.409 MiB, 8.60% gc time)
  1.002034 seconds (12.92 M allocations: 894.409 MiB, 8.10% gc time)
  0.900417 seconds (12.92 M allocations: 894.409 MiB, 7.86% gc time)
  0.897626 seconds (12.92 M allocations: 894.409 MiB, 7.42% gc time)
  1.011860 seconds (12.92 M allocations: 894.409 MiB, 7.58% gc time)
Log probability: -53.7625847077635
```

Visualizing MCMC updates

- A great tool for debugging and improving MCMC algorithms is visualization. We can use Plots.@animate to produce an animation:

```
t, = generate(regression_with_outliers, (xs,), observations)

viz = Plots.@animate for i in 1:500
    global t
    t = block_resimulation_update(t)
    visualize_trace(t; title="Iteration $i/500")
end;
gif(viz)
```

- Small changes over time
- After initial iterations, it has a harder time refining the continuous parameters



Random walk MH

- Given current proposal H, sample a small step, dH, and walk to a new proposal, H'=H+dH
- E.g., H' ~ uniform[H - max_step_size, H + max_step_size]
- Implement this via a custom proposal in Gen:

```
(tr, did_accept) = mh(tr, custom_proposal, custom_proposal_args)
```

- A special case, Gaussian drift MH, sample a new proposal value from a gaussian distribution with the mean being the previous proposal value:

```
@gen function line_proposal(current_trace)
    slope ~ normal(current_trace[:slope], 0.5)
    intercept ~ normal(current_trace[:intercept], 0.5)
end;
```

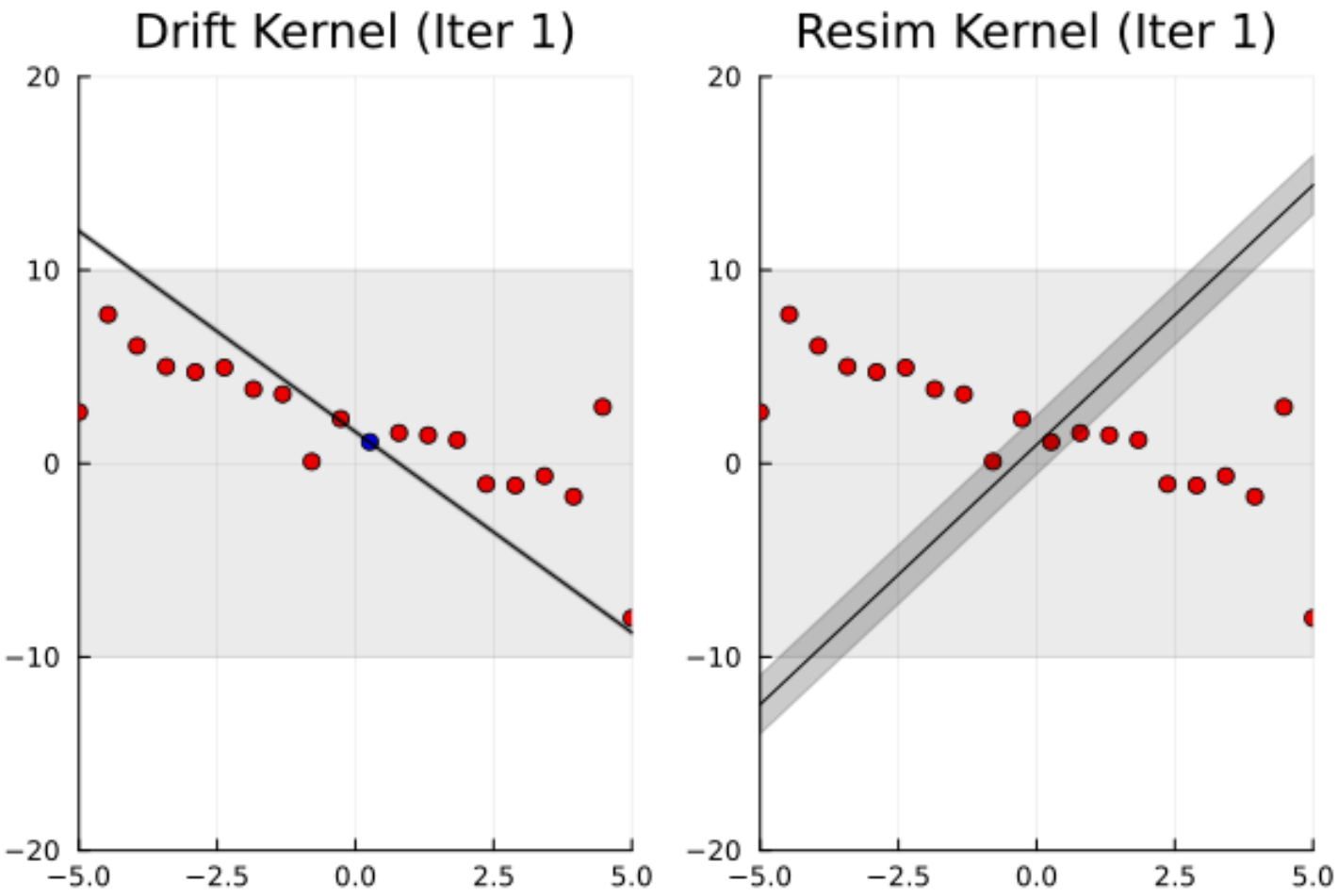
Gaussian drift MH update

```
function gaussian_drift_update(tr)
    # Gaussian drift on line params
    (tr, _) = mh(tr, line_proposal, ())

    # Block resimulation: Update the outlier classifications
    (xs,) = get_args(tr)
    n = length(xs)
    for i=1:n
        (tr, _) = mh(tr, select(:data => i => :is_outlier))
    end

    # Block resimulation: Update the prob_outlier parameter
    (tr, w) = mh(tr, select(:prob_outlier))
    (tr, w) = mh(tr, select(:noise))
    tr
end;
```

Note:

1. No need to select arguments
2. Previous trace is passed into the customized proposal by default

Gaussian drift MH vs block resimulation MH



Gaussian drift MH vs block resimulation MH

@time func()

Gaussian drift MH

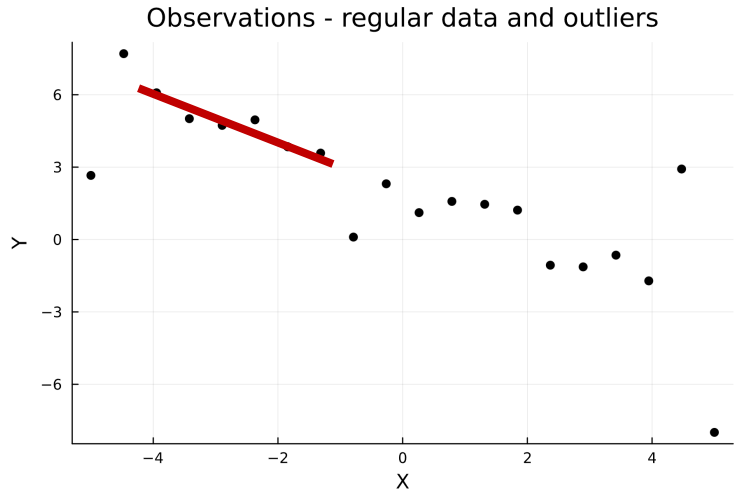
```
  1.051416 seconds (12.01 M allocations: 699.933 MiB, 5.93% gc time, 1.19% compilation time)
  1.161278 seconds (12.00 M allocations: 699.043 MiB, 5.36% gc time)
  1.032212 seconds (12.00 M allocations: 699.043 MiB, 5.30% gc time)
  0.732046 seconds (12.00 M allocations: 699.043 MiB, 6.20% gc time)
  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
```

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
```

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
```