Week 7 — Approximation: Monte Carlo & MCMC

Friday, June 12, 2026

Prof. Joseph Austerweil

Agenda

Admin + What *is* Monte Carlo? (poll)	0:00
Basic Monte Carlo	0:08
Importance sampling	0:23
Particle filtering	0:38
MCMC: design a chain for a target	0:48
Interactive: Gibbs & MH	1:03
Break	1:15
Programmable inference (GenJAX)	1:20
MCMC with People	1:36
Sampler for the Kemp model	1:42

What *is* Monte Carlo?

Admin — programming assignment + GenJAX

Today’s tools are the next programming assignment — a sampler you build yourself.

- We’ll end with the exact recipe for the **Kemp hierarchy sampler** (the Week 4 model). That recipe *is* the assignment.
- **Walkthrough + stencils:** hml.chibatech.dev/assignments.html — GenJAX / Python / R, one-click Colab.
- Reminder: **3 free late days pooled** across the four programming assignments.
- Questions? **DM me, or the class channel.**

Admin — final project proposal

Final project proposal due Sun Jun 28, 8:00 PM — about one page, graded pass/fail.

- Four sections: **Background · Question · Method · References** (at least 3).
- Three (non-exclusive) categories: **experiment** · **modeling/ML** · **math/theory** — any topic in human and machine learning, not just class material.
- **Talk to me about your idea before the deadline** — DM or email for 5 minutes.
- **Full guidelines:** hml.chibatech.dev/project.html

Looking ahead: presentations Fri Jul 17 (final session) · paper due Fri Jul 24

Poll — the hospital problem

A **large** hospital and a **small** hospital each record, over a year, the days where **more than 60% of the babies born are boys**.

Across the year, which hospital records **more** such days?

A. the larger hospital

B. the smaller hospital

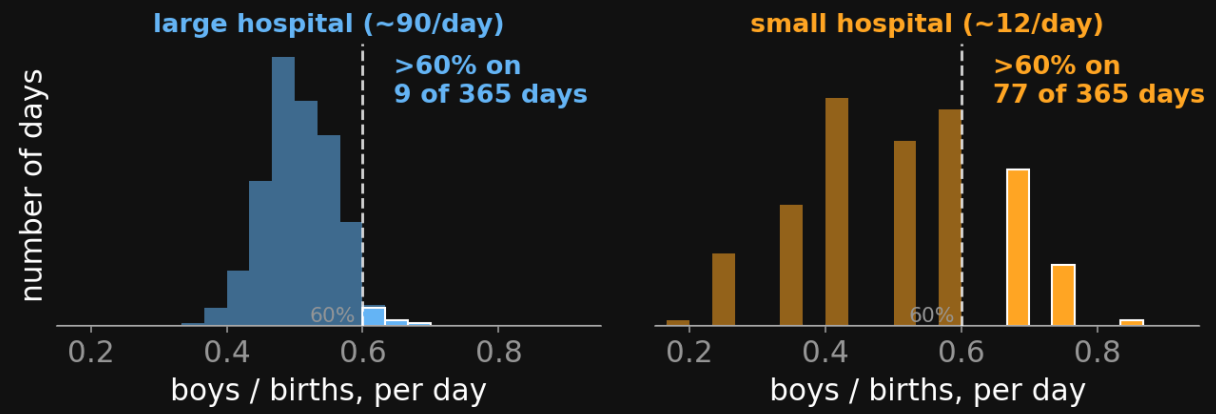
C. about the same

Poll — answer

B — the smaller hospital.

Small samples vary more — the **law of large numbers**. A big sample's boy-fraction sits tight near 50%; a small one swings wildly, so it crosses 60% on far more days.

Simulate one year: same 50% rate, but the small hospital's daily fraction swings wider — far more >60% days.



That was Monte Carlo

That was Monte Carlo.

You judged a probability by reasoning about **repeated random samples** — and more samples means a tighter answer. Today we make that the central tool.

Why approximate?

Exact Bayesian inference means summing/integrating over **all** hypotheses — usually intractable.

- **Top-down strategy:** borrow good samplers from CS/statistics; then ask whether they double as *psychological processes*.
- When they do, we get **rational process models** — algorithms that are both good inference *and* candidate accounts of how the mind computes. (Sanborn et al. 2010; Shi et al. 2010)

Basic Monte Carlo

The Monte Carlo estimator

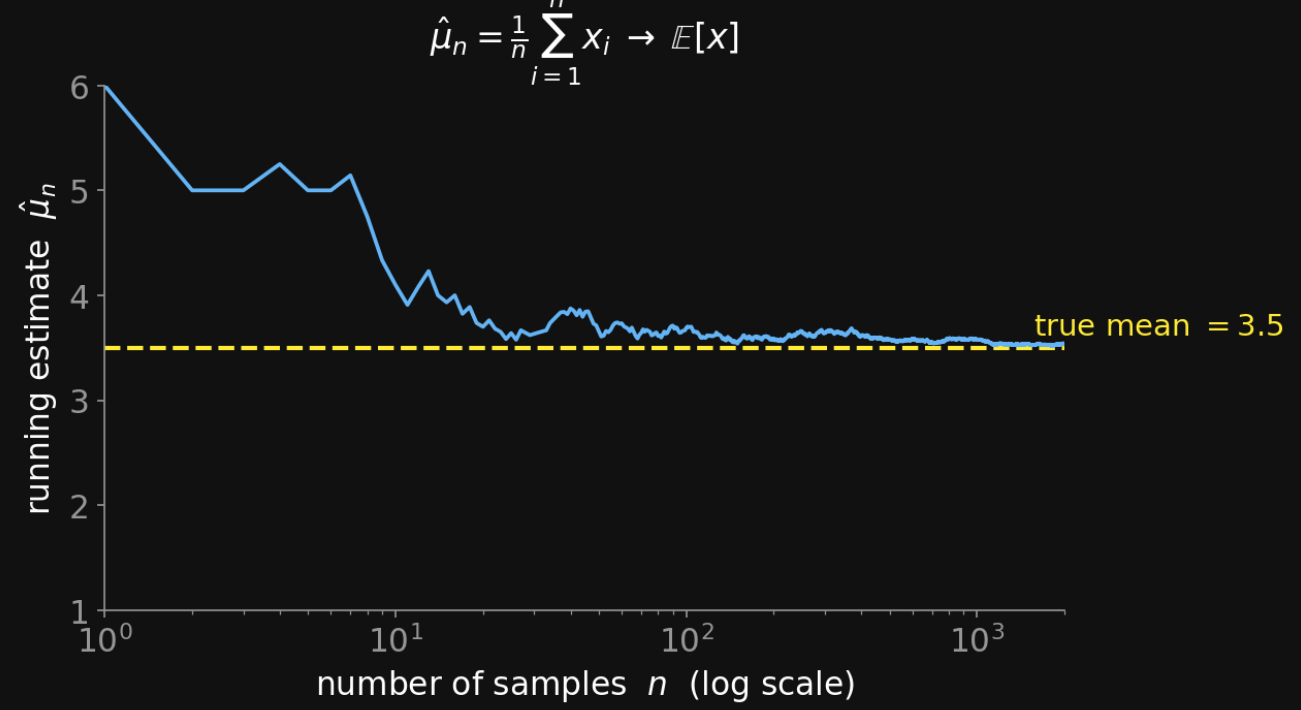
To estimate an expectation $\mathbb{E}_P[f(x)]$ (the average of f over draws from P), draw samples and **average**:

$$\hat{\mu}_n = \frac{1}{n}\sum_{i=1}^{n} f(x_i), \qquad x_i \sim P(x)$$

$\hat{\mu}_n$ = our n-sample **estimate** of that mean (the hat means “estimated”).

Example — a die roll. The average number of spots is $\mathbb{E}[x] = 3.5$.

Roll it n times and average. The estimate is noisy for small n and settles toward **3.5** as n grows.

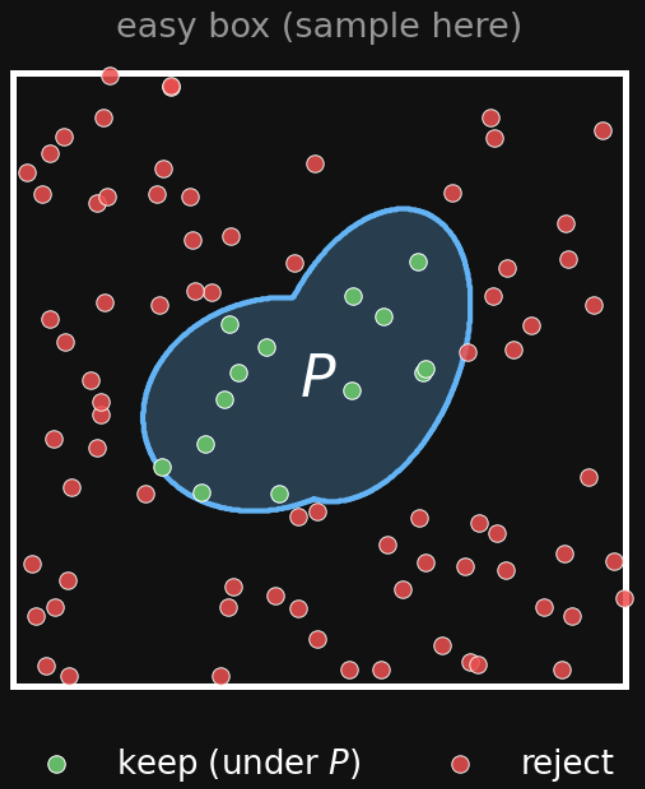


Rejection sampling

The average above needs samples from P — but often we can't draw from P directly. The simplest fix: **rejection sampling**.

- Sample from an **easy** region you *can* draw from (e.g. a box that contains P).
- **Keep** the points that fall under P (inside the target region); **throw away** the rest.
- The kept points are distributed exactly as P — and the *fraction kept* estimates P's area/probability.

Simple and exact, but it **wastes samples**: if P fills little of the box, most draws are rejected. The next example makes this concrete.



π by throwing darts — rejection in action

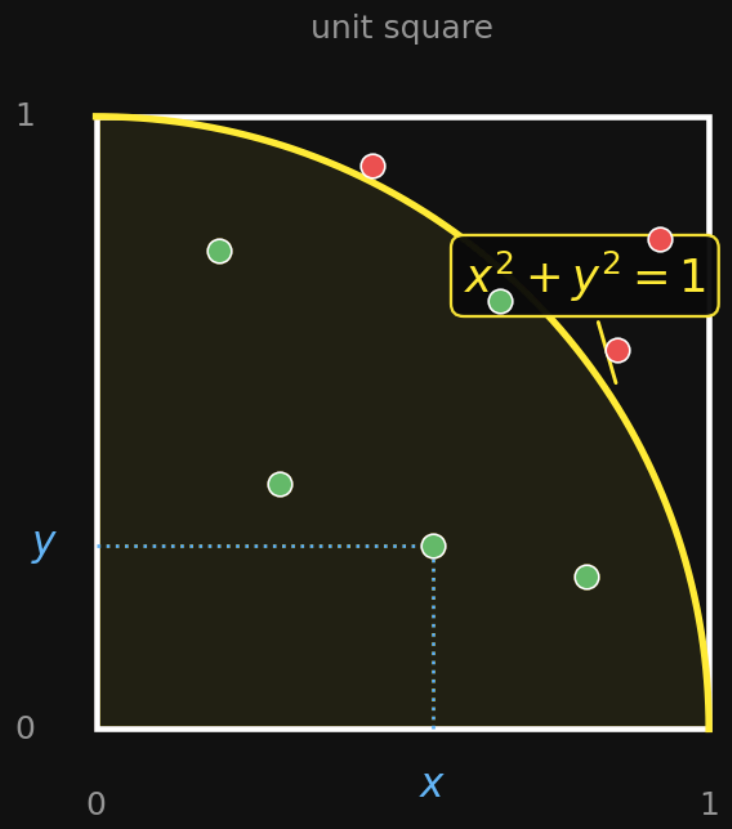
Rejection sampling in action. Throw darts at the **unit square** (the easy region): draw two uniforms $x, y \sim \mathrm{Uniform}[0, 1]$. **Keep** a dart when it's inside the quarter-circle, $x^2 + y^2 \leq 1$; reject the rest.

- Quarter-circle area $= \pi/4$; square area $= 1$.
- So the **fraction kept** $\approx \pi/4$:

$$\hat{\pi} = 4 \cdot \frac{\#\,\text{inside}}{\#\,\text{total}}$$

This is exactly $\hat{\mu}_n$ with $f(x, y) = 1[x^2 + y^2 \leq 1]$.

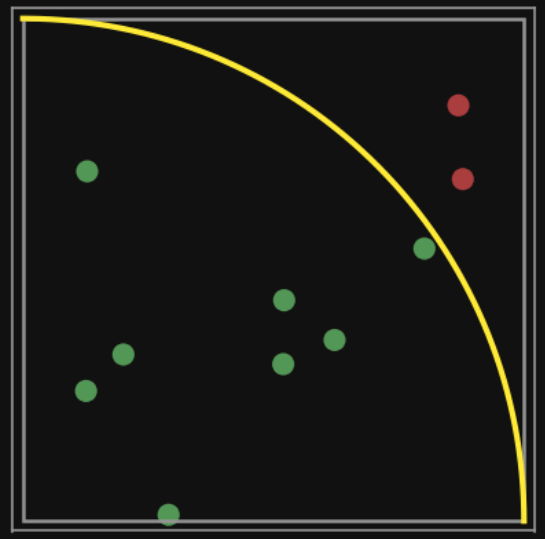
$1[\cdot]$ = the **indicator**: 1 when the condition holds, 0 otherwise.



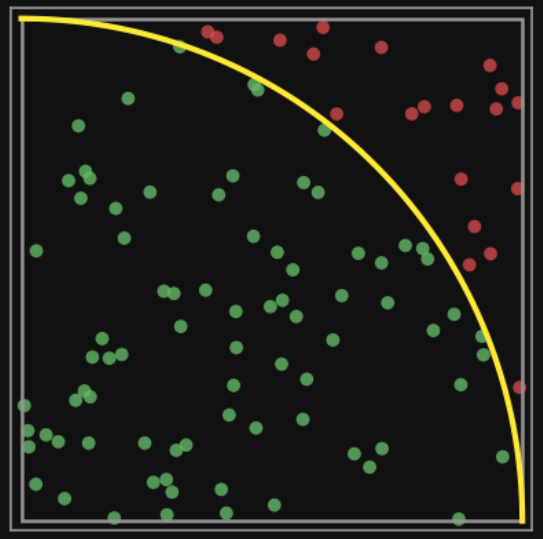
More darts → better estimate

$$x, y \sim \text{Uniform}[0, 1] \quad \hat{\pi} = 4 \cdot \frac{\#\,\text{inside}\ (x^2 + y^2 \le 1)}{\#\,\text{total}}$$

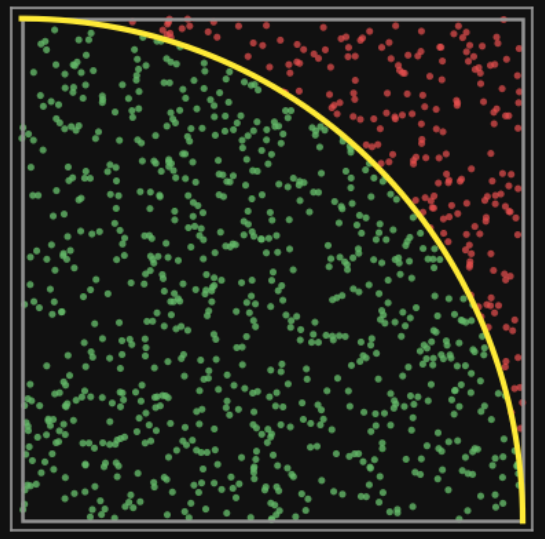
$n = 10$
$\hat{\pi} = 3.200$



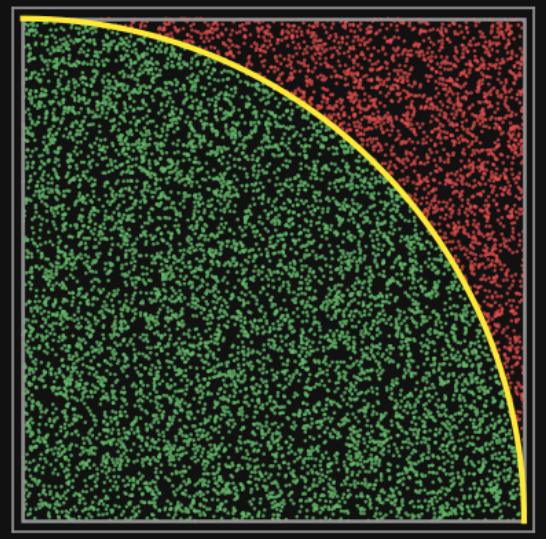
$n = 100$
$\hat{\pi} = 3.200$



$n = 1,000$
$\hat{\pi} = 3.172$



$n = 10,000$
$\hat{\pi} = 3.124$



Green = kept (inside), red = rejected (outside). The estimate tightens toward 3.14159… — the same law of large numbers, now in 2-D.

What Monte Carlo gives you

The estimator $\hat{\mu}_n$ is:

- **consistent** — converges to the truth as data grows: $\hat{\mu}_n \to \mathbb{E}[f]$ as $n \to \infty$.
- **unbiased** — correct *on average* even for small n: $\mathbb{E}[\hat{\mu}_n] = \mathbb{E}[f]$.
- **error shrinks like** $1/\sqrt{n}$ — *regardless of dimension* (unlike a grid, whose cost explodes as dimensions grow). The estimate's spread is approximately Gaussian — "asymptotically normal."

But: rejection sampling **throws most draws away**. Can we instead **use every sample**, each weighted by how much it tells us about the quantity we're estimating? → importance sampling.

In GenJAX: Monte Carlo is `simulate` + `vmap`

```python
@gen
def dart():
    x = uniform(0.0, 1.0) @ "x"    # throw a dart
    y = uniform(0.0, 1.0) @ "y"
    return (x*x + y*y <= 1.0)      # f = 𝕀[inside]

key  = jax.random.key(0)
keys = jax.random.split(key, 10_000)
sim  = lambda k: dart.simulate(k, ())
traces = jax.vmap(sim)(keys)       # 10k darts
f = traces.get_retval()            # 0/1 per dart
pi_hat = 4.0 * jnp.mean(f)         # π^ = 4·in/tota
```

- The **same darts** as before: two uniforms, keep when $x^2 + y^2 \leq 1$.
- `simulate` throws **one** dart (a *trace*); `vmap` throws n in parallel.
- `f` *is* the indicator $1[\cdot]$; its mean is the fraction inside, so $\hat{\pi} = 4 \cdot \bar{f}$.

A probabilistic program's `simulate` *is* the sampler in the estimator.

…and running it

```
# peek at the first 5 darts (the trace)
>>> traces.get_choices()["x"][:5]
Array([0.104, 0.091, 0.746, 0.849, 0.610])
>>> traces.get_choices()["y"][:5]
Array([0.228, 0.752, 0.743, 0.928, 0.989])
>>> f[:5]                # 𝟙[x²+y² ≤ 1]
Array([1, 1, 0, 0, 0])

>>> jnp.sum(f)           # darts inside
Array(7859)              # out of 10,000
>>> pi_hat
Array(3.1436, dtype=float32)
```

- Dart 1 = $(0.10, 0.23)$: $x^2 + y^2 = 0.06 \leq 1$ → **inside** ($f = 1$).
- Dart 3 = $(0.75, 0.74)$: $x^2 + y^2 = 1.11 > 1$ → **outside** ($f = 0$).
- $7859/10{,}000 = 0.786$ inside, so $\hat{\pi} = 4 \times 0.786 = 3.14$.

The estimate landed on π. Same trace, same indicator, same $4 \cdot \bar{f}$ as the darts slide — now it's a probabilistic program.

Importance sampling

Sample the wrong distribution — then reweight

We want $\mathbb{E}_P[f] = \int f(x)\, p(x)\, dx$, but we **can't sample from** P.

Idea: sample from an **easy** distribution q that we *can* draw from, and **correct** for using the wrong one.

p, q = the **densities** of the target P and the proposal q. Note $\mathbb{E}_P[f]$ already contains one factor of p — it's the $p(x)$ in the integral.

The trick: multiply by q/q

Start from the integral, multiply the integrand by $1 = \dfrac{q(x)}{q(x)}$, and regroup:

$$\mathbb{E}_P[f] = \int f(x)\, p(x)\, dx = \int f(x)\, p(x)\, \frac{q(x)}{q(x)}\, dx$$

$$= \int \underbrace{f(x)\, \frac{p(x)}{q(x)}}_{\text{new integrand}} \underbrace{q(x)\, dx}_{\text{the measure for } \mathbb{E}_q} = \mathbb{E}_q\left[f(x)\, \frac{p(x)}{q(x)}\right]$$

The leftover $q(x)\, dx$ is exactly what makes this an **expectation under** q — it's the density we integrate against. This is still an **exact identity**: no sampling yet. We *sample* from q only on the next slide, to *estimate* this expectation.

The estimator and the weight

Now it's an expectation under q — so estimate it the basic-MC way: draw $x_i \sim q$ and average.

First name the leftover factor — the **importance weight** $w(x)$:

$$w(x) = \frac{p(x)}{q(x)}$$

$$\mathbb{E}_P[f] = \mathbb{E}_q[f(x)\,w(x)] \approx \frac{1}{n}\sum_{i=1}^{n} f(x_i)\,w(x_i)$$

Up-weight where $p > q$, down-weight where $p < q$. **No samples are thrown away** — every draw counts, weighted by how useful it is.

In practice we usually divide by the **sum of the weights** instead of n — the **self-normalized** estimator:

$$\mathbb{E}_P[f] \approx \frac{\sum_i f(x_i)\,w(x_i)}{\sum_i w(x_i)}$$

Now **only ratios of weights** appear — which is what lets p be unnormalized (next slide).

Bonus: p can be unnormalized

With self-normalizing (previous slide), **only ratios of weights** appear, and each weight is itself a ratio $w = p/q$.

So p may be **unnormalized**: if $p = \tilde{p}/Z$ for some unknown constant Z, the Z **cancels** between numerator and denominator and never matters.

$$\frac{\sum_i f(x_i)\, w_i}{\sum_i w_i} = \frac{\sum_i f(x_i)\, \dfrac{\tilde{p}(x_i)}{Z\, q(x_i)}}{\sum_i \dfrac{\tilde{p}(x_i)}{Z\, q(x_i)}} = \frac{\sum_i f(x_i)\, \dfrac{\tilde{p}(x_i)}{q(x_i)}}{\sum_i \dfrac{\tilde{p}(x_i)}{q(x_i)}}$$

The same Z multiplies every term in **both** sums, so it factors out and cancels — gone.

This is huge for Bayesian work: a posterior $p(h \mid d) \propto p(d \mid h)\, p(h)$ is only known **up to** the constant $p(d)$ — and importance sampling doesn't need it.

Good q vs bad q: look at the weights

Weight variance = sampler quality. A few huge weights (right) ⇒ effectively a handful of samples (**ESS** ≈ 6 of 400) ⇒ a noisy estimate. Good q (left): ESS ≈ 230.

Effective sample size, as a number

Put a number on “how many of my T weighted samples actually count.” With **normalized** weights ($w_t \geq 0$, $\sum_t w_t = 1$):

$$N_{\text{eff}} = \frac{1}{\sum_{t=1}^{T} w_t^2}$$

- **All weights equal** ($w_t = 1/T$) ⇒ $N_{\text{eff}} = T$. Nothing wasted.
- **One weight dominates** ⇒ $N_{\text{eff}} \to 1$. The estimate rests on a single sample.
- It measures **how even the weights are** — i.e. how well q covers p — *not* how accurate your final estimate is. (Hold onto that distinction; the assignment’s Problem 3 turns on it.)

The simplest importance sampler

For a posterior $P(h \mid d)$, use the **prior as the proposal**: $q = P(h)$. Then the weights are just the likelihood:

$$w(h) = \frac{P(h \mid d)}{P(h)} = \frac{P(d \mid h)\,P(h)}{P(d)\,P(h)} = \frac{P(d \mid h)}{P(d)} \;\propto\; P(d \mid h)$$

Sample from the prior, weight by the likelihood. That's the whole recipe.

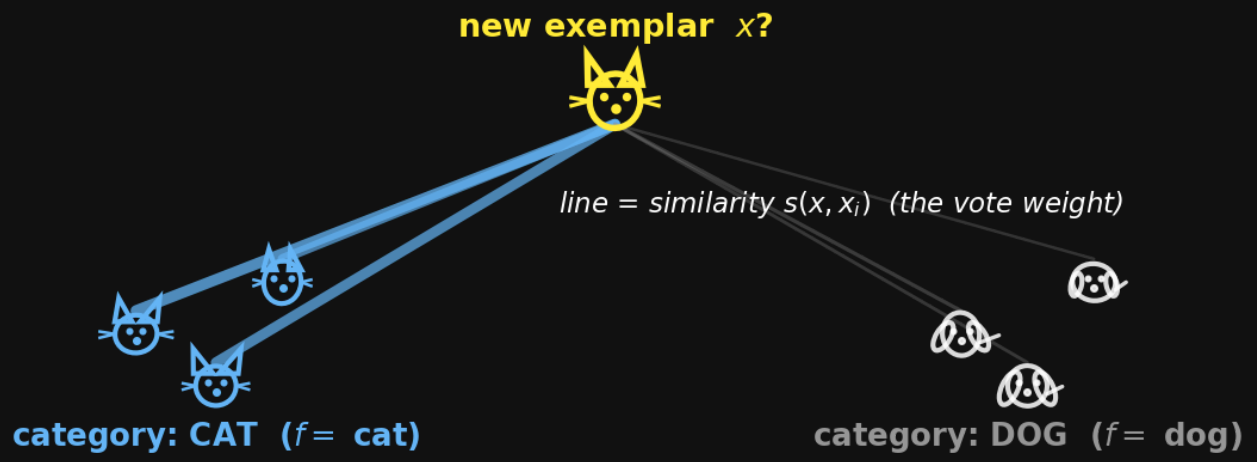
We don't know $P(d)$, so divide by the **sum of the weights** instead of $1/n$ — this is **self-normalized** importance sampling: $\mathbb{E}[f] \approx \sum_i f(h_i) w_i \,/\, \sum_i w_i$.

Exemplar models *are* importance samplers

A cognitive aside that recurs all term. **Exemplar models** (Nosofsky 1986) answer with a similarity-weighted vote over stored examples (x_i):

$$\text{response}(x) = \frac{\sum_i s(x, x_i)\, f(x_i)}{\sum_i s(x, x_i)}$$

$s(x, x_i)$ = **similarity** between the query x and stored example x_i (the weight); $f(x_i)$ = the **label/output** stored with example i.



This is **exactly** the self-normalized importance sampler from earlier ($\sum_i w\, f / \sum_i w$) — the stored exemplars are the samples, and similarity s plays the role of the weight w.

In GenJAX: importance sampling for free

```
# the model's PRIOR is the default proposal
trace, log_weight = model.importance(
    key, constraint, args)

# score a custom proposal yourself
log_p, retval = model.assess(choices, args)

# K particles (sequential Monte Carlo)
alg = smc.ImportanceK(
    Target(model, args, constraint),
    k_particles=100)
```

- `importance` returns a sample **and** its `log_weight` — “sample from prior, weight by likelihood” in one call.
- `assess` scores choices under the model — the primitive you’ll reuse for MCMC.

...importance sampling, run

```python
# coin with unknown bias; observe heads
@gen
def coin():
    theta = beta(2.0, 2.0) @ "theta"
    y = flip(theta) @ "y"
    return theta

obs = C["y"].set(True)          # condition: hea
def one(k):
    tr, logw = coin.importance(k, obs, ())
    return tr.get_retval(), logw
thetas, logws = jax.vmap(one)(
    jax.random.split(key, 5000))
# self-normalize: weight by the likelihood
w = jnp.exp(logws); w = w / w.sum()
post_mean = jnp.sum(w * thetas)
```

```
>>> logws[0]    # = log p(heads|θ)
Array(-0.216, dtype=float32)

>>> thetas[:3]  # prior draws of θ
Array([0.806, 0.485, 0.455],
      dtype=float32)

>>> post_mean   # E[θ | heads]
Array(0.605, dtype=float32)

# true posterior Beta(3,2)
#   mean = 0.600  ✓
```

Sampled θ from the **prior**, weighted each by the **likelihood** of heads — the estimate 0.605 matches the exact posterior mean 0.600.

Poll — when the prior is a bad proposal

You importance-sample a posterior using the **prior** as q. The prior and posterior **barely overlap**. What happens to your estimate?

A. a few samples get huge weights — the estimate is noisy

B. all weights are equal — the estimate is exact

C. every sample is rejected

A — weight variance blows up. This is *exactly* why the textbook's importance sampling for the Kemp hyperparameters was a **"blunt tool."** We fix it with MCMC.

Particle filtering

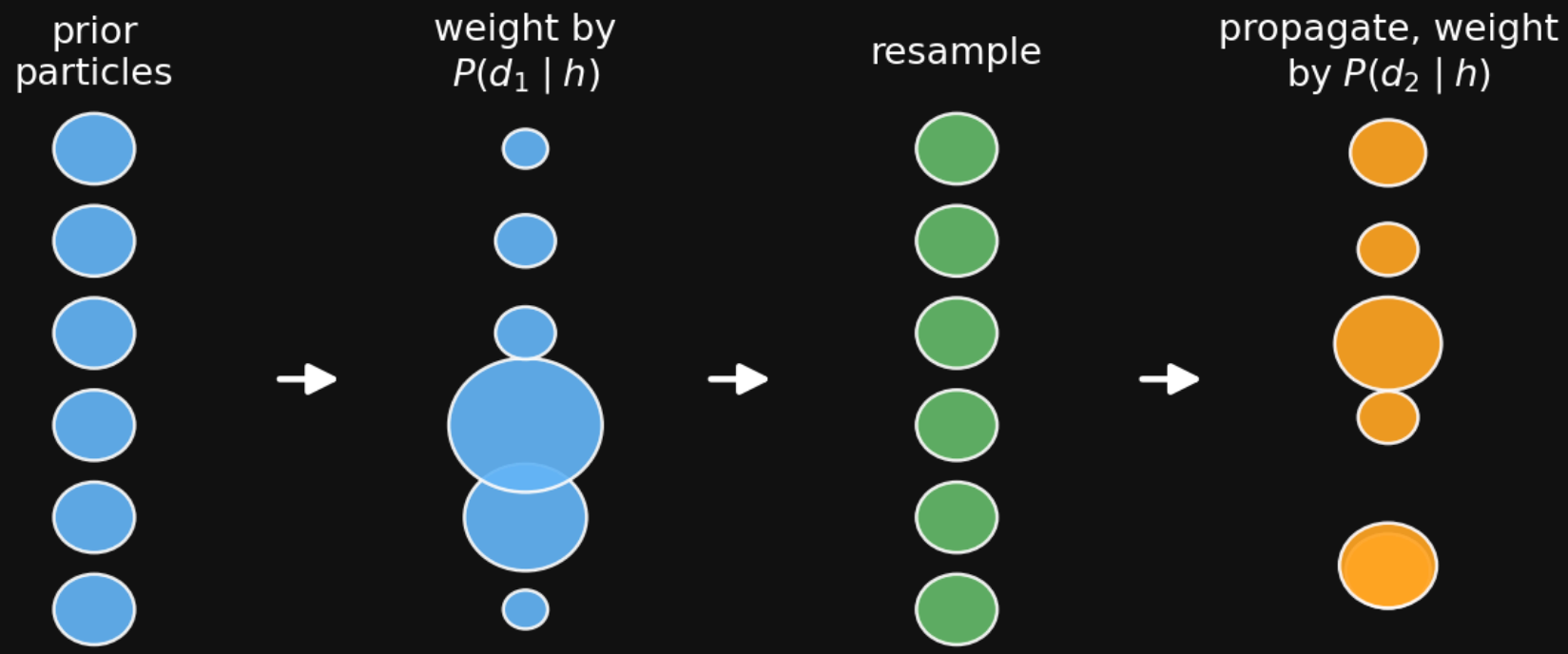
Data arriving over time

When data come one at a time, recomputing $P(h \mid d_1, \ldots, d_n)$ from scratch each step is wasteful. Exploit a simple fact:

$$P(h \mid d_1, \ldots, d_n) \;\propto\; P(d_n \mid h)\, \underbrace{P(h \mid d_1, \ldots, d_{n-1})}_{\text{yesterday's posterior}}$$

Yesterday's posterior is today's prior. Repeated importance sampling over a growing dataset = a **particle filter** (a.k.a. sequential Monte Carlo).

The particle filter



particle size = weight; resampling kills light particles, copies heavy ones

Init M particles from the prior. Per datum: **weight** by $P(d_n \mid h)$, normalize, and **resample** if the weights degenerate (kill light particles, copy heavy ones).

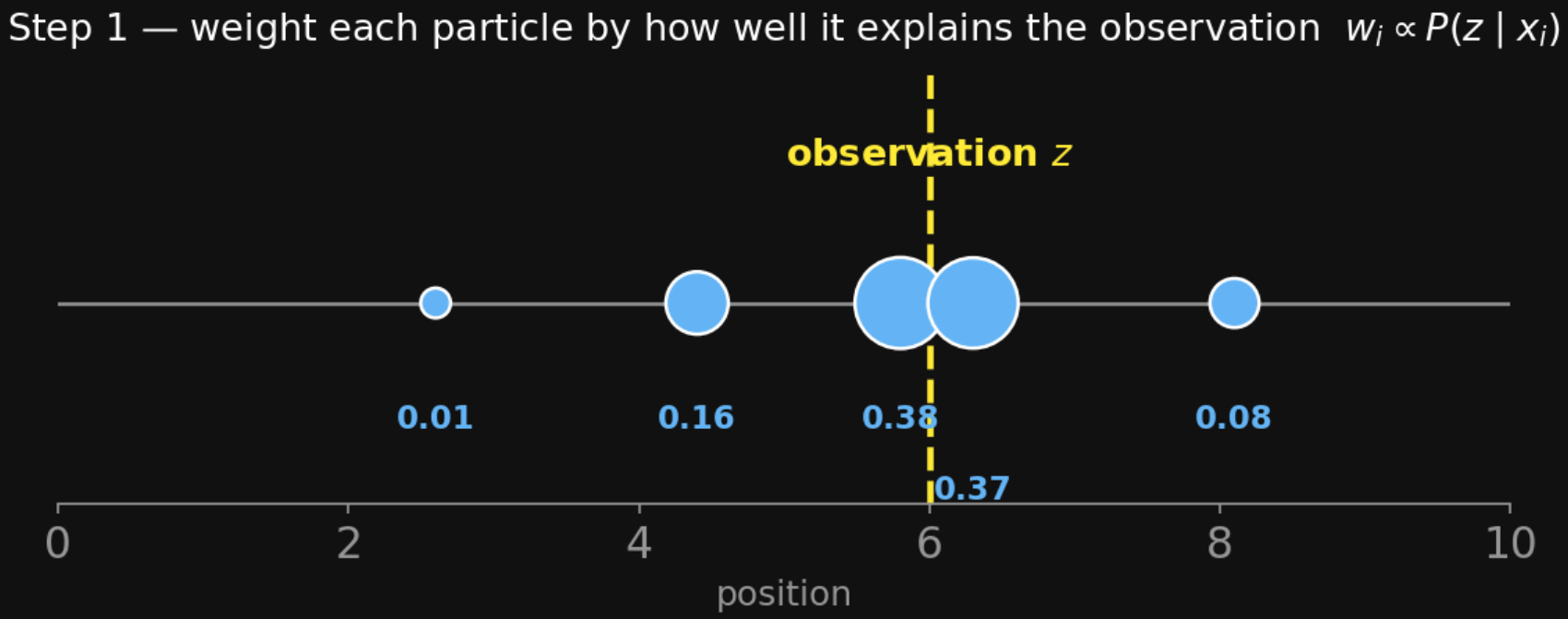
A worked example: tracking a moving object

A robot slides along a line; we want its position over time. We carry **5 particles** — guesses for where it is. Tracking over time needs **two modeling assumptions** — the same two any state-space model makes:

- **Observation model** $P(z \mid x)$ — how a state produces a sensor reading. Here: $z \sim \mathcal{N}(x, \sigma^2)$ (noisy sensor). Gives the **weight** $w_i \propto P(z \mid x_i)$.
- **Motion model** $P(x_t \mid x_{t-1})$ — how the state *evolves* between ticks. Here we **assume** the robot drifts $+1$ with a little noise. Used to **propagate**.

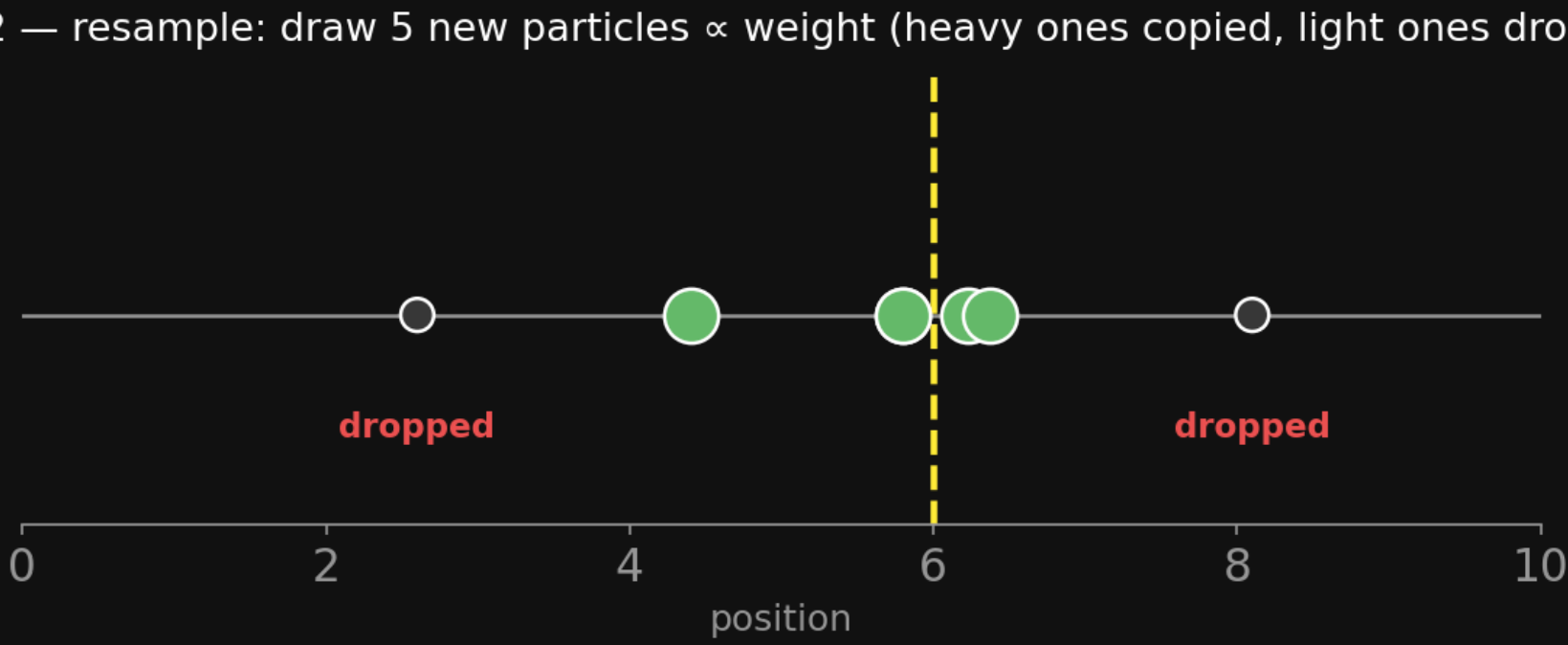
Both are *your* modeling choices, not facts. If the thing **doesn't move**, the motion model is just "stay put" ($x_t = x_{t-1}$) — particle filtering doesn't *require* dynamics, only that you state *some* transition.

Step 1 — weight by the observation



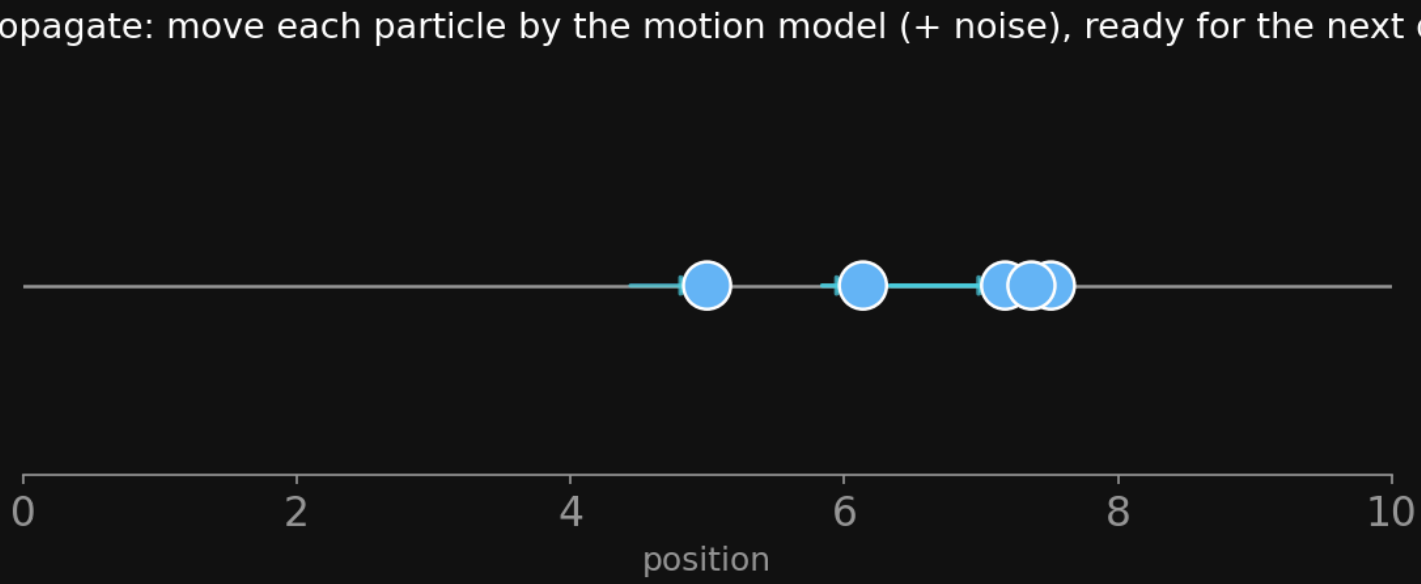
Each particle gets a weight $w_i \propto P(z \mid x_i)$ — bigger dots are heavier. The two particles flanking the observation carry almost all the weight; the far ones are nearly zero.

Step 2 — resample



Draw 5 **new** particles with replacement, each chosen with probability $\propto w_i$. Heavy particles get copied (sometimes twice); the two far-from-z particles are **dropped**. Same particle count, mass now concentrated where the data points.

Step 3 — propagate



Push every particle forward through the **assumed motion model** $P(x_t \mid x_{t-1})$ from the setup (here: drift $+1$, plus a little noise). The cloud is now this tick's **prior** — ready to be weighted by the next observation. Loop back to Step 1. *(Static target? This step is just $x_t = x_{t-1}$ — the particles stay put.)*

The particle filter in GenJAX

```python
@gen
def step(x_prev):
    x = normal(x_prev + 1.0, 0.5) @ "x"  # propagate
    z = normal(x, 1.2) @ "z"             # observe
    return x

def filter(key, observations, xs):
    for z in observations:
        k1, k2, key = jax.random.split(key, 3)
        # weight: importance over all particles at onc
        keys = jax.random.split(k1, len(xs))
        tr, log_w = jax.vmap(lambda k, xp:
            step.importance(k, C["z"].set(z), (xp,))
        )(keys, xs)
        # resample ∝ weight
        idx = jax.random.categorical(k2, log_w,
                                     shape=xs.shape)
        xs = tr.get_choices()["x"][idx]
    return xs
```

- The **same three steps**: `importance` = weight, `categorical` = resample, the next loop's `simulate` inside `step` = propagate.
- `importance` returns each particle **and** its `log_weight` — no hand-derived likelihood.
- `vmap` runs all particles in parallel on one device.

…the filter, run

```python
# object drifts +1 per tick from 1
true  = [1, 2, 3, 4, 5]
# noisy sensor readings
obs   = [1.3, 1.8, 3.4, 3.9, 5.2]

ests = []
xs = init_particles(key, M=1000)
for z, x in run(key, obs):
    ests.append(float(jnp.mean(xs)))
```

```
true  : [1.0, 2.0, 3.0, 4.0, 5.0]
obs   : [1.3, 1.8, 3.4, 3.9, 5.2]
PF est: [1.1, 1.96, 3.07, 4.01, 5.08]
```

- The filter's mean tracks the **true** position [1, 2, 3, 4, 5] — even though it only ever sees the **noisy** observations.
- It **smooths**: the estimate 1.96 at tick 2 is pulled back toward the track from the noisy 1.8, because the motion model says "you should be near 2."
- 1000 particles, real GenJAX 0.10.3 output.

Particle filters as process models

Why this, of all approximations? A particle filter is a probabilistic approximation that builds **incrementally** — yesterday's posterior is today's prior. But "incremental + probabilistic" alone isn't special (exact recursive Bayes is too). What makes it a *cognitive* model is the **shape** of its approximation: it holds a **small, fixed number of concrete hypotheses** (M particles), updated by **stochastic resampling**.

That specific form predicts the ways people *deviate* from full Bayes:

- **limited memory** — only M hypotheses carried forward (people track a handful, not a full distribution; M is a fittable "how-approximate" knob).
- **order effects** — early data decide which particles survive (Kruschke 2006); a parametric incremental filter wouldn't predict this.
- **variability** — resampling is random, so two people (two runs) can land on different beliefs from the same data.

Applied to feature learning (Austerweil & Griffiths 2013), categorization (Sanborn et al. 2006), associative learning, changepoint detection, sentence processing. **In GenJAX:** `smc.*` chained over the observations.

MCMC: design a chain to hit a target

Running Week 6 backwards

Last week you **found** a chain's stationary distribution π by running it.

π is stationary when $\pi P = \pi$ — one more step doesn't change it.

This week we **reverse it**: start with a target $P(x)$ we *want* (a posterior), and **design** a Markov chain whose stationary distribution **is** $P(x)$. That's **Markov chain Monte Carlo (MCMC)**.

Two schemes: **Metropolis–Hastings** and **Gibbs**.

Metropolis–Hastings

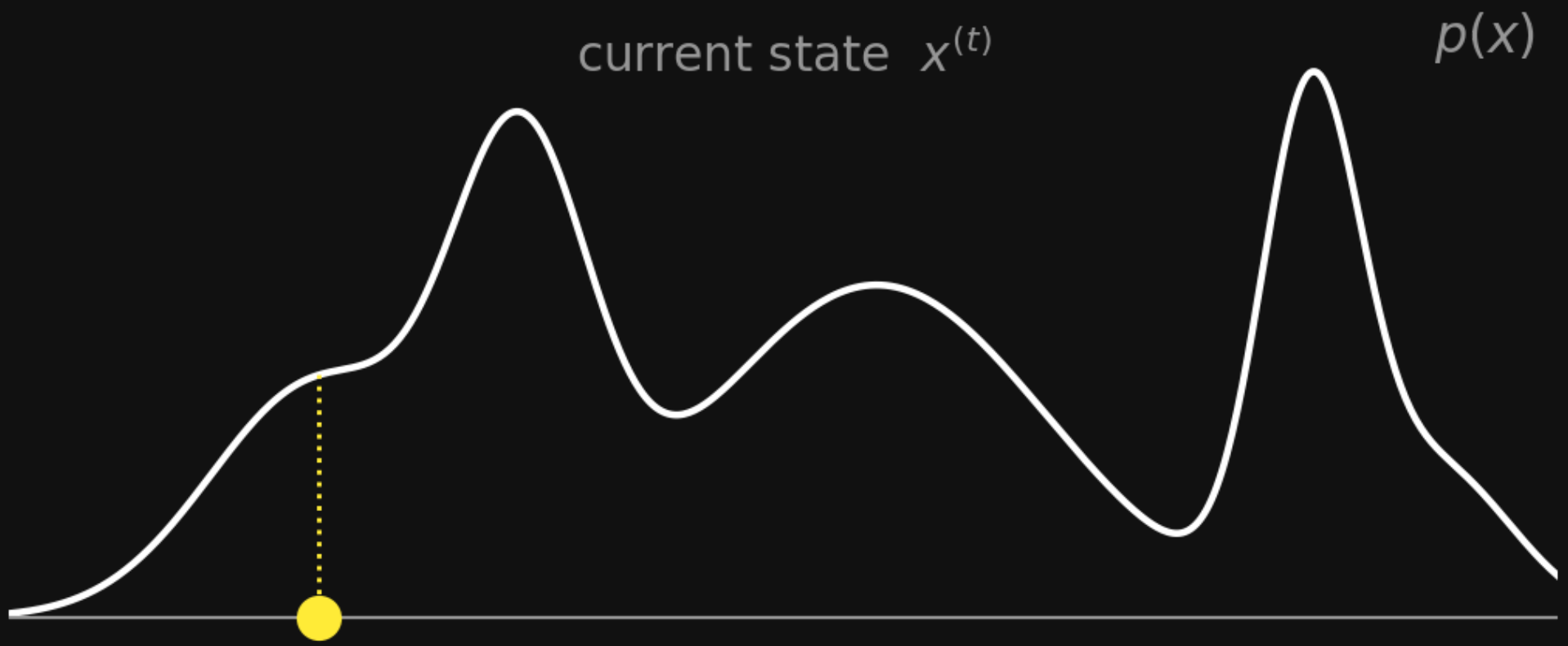
(Metropolis et al. 1953; Hastings 1970) $x^{(t)}$ = the state at iteration t.

- **Step 0.** Start anywhere: $x^{(0)}$.
- **Step 1. Propose** a move $x' \sim Q(x' \mid x^{(t)})$. Q = the **proposal** — usually a Gaussian step $Q = \mathcal{N}(x^{(t)}, \sigma^2)$, width σ. ($\mathcal{N}(\mu, \sigma^2)$ = Normal, mean μ, variance σ^2.)
- **Step 2. Accept** with probability $A = \min\left(1,\ \frac{P(x')}{P(x^{(t)})}\right)$ *(symmetric-proposal form)*. Otherwise stay.

Intuition: a move **uphill** (toward higher P) is **always** accepted; a move **downhill** is accepted *sometimes*, in proportion to how far down it goes. That's what lets the chain explore yet still favor high-probability regions.

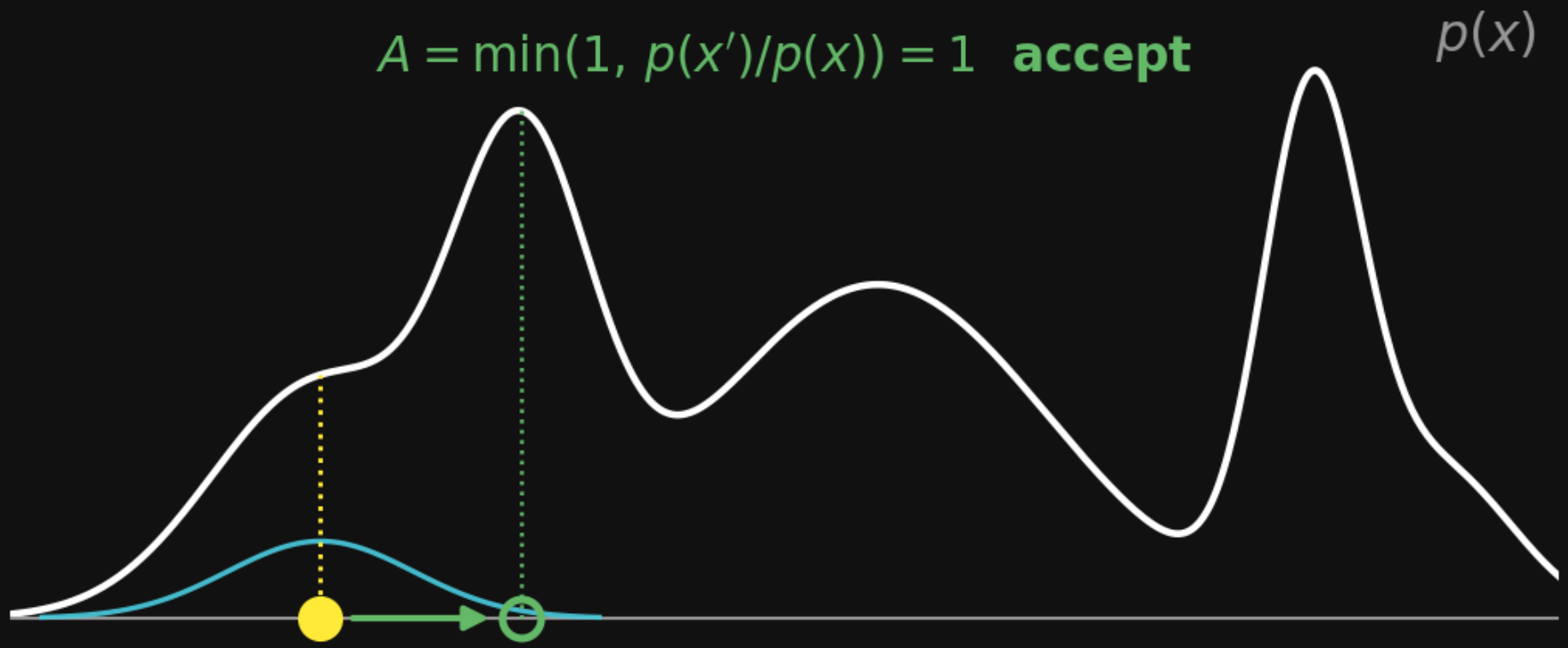
A Gaussian step is **symmetric** ($Q(x' \mid x) = Q(x \mid x')$), so the Q terms cancel and only $P(x')/P(x)$ is left — that's the **Metropolis** special case. (Full Hastings keeps a $Q(x \mid x')/Q(x' \mid x)$ factor for asymmetric proposals.) And only the **ratio** of P matters, so the **normalizing constant cancels** — you never need $P(d)$.

MH, step by step (1/3)



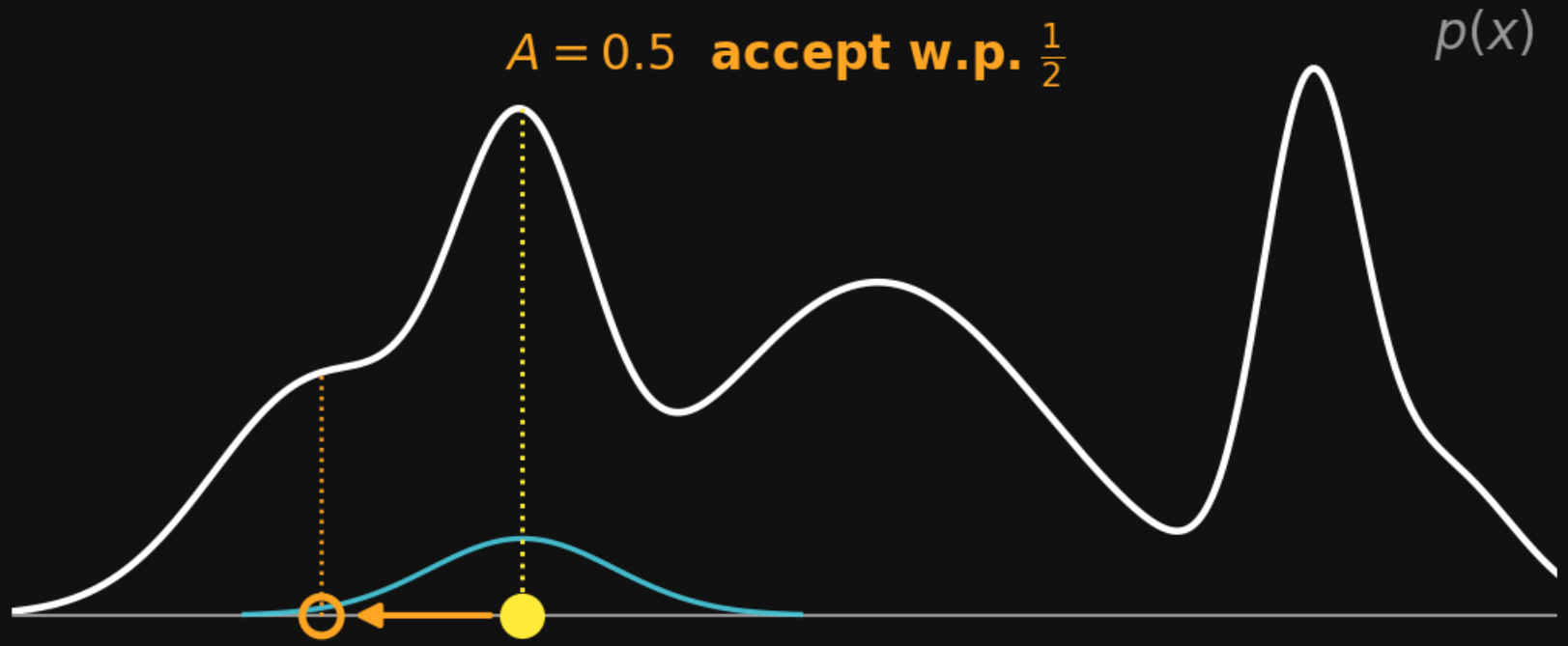
We want to sample this multimodal $p(x)$. The chain sits at the current state.

MH, step by step (2/3)



Propose a nearby move. It's **uphill** ($p(x') > p(x)$), so the ratio $p(x')/p(x) > 1$ and $A = \min(1, \cdot) = 1$ — **always accept**.

MH, step by step (3/3)



A **downhill** proposal (here $p(x')/p(x) = 0.5$): $A = 0.5$ — accept only **half the time**. Sometimes the chain moves to lower density; that's what lets it explore, not just hill-climb.

Gibbs sampling

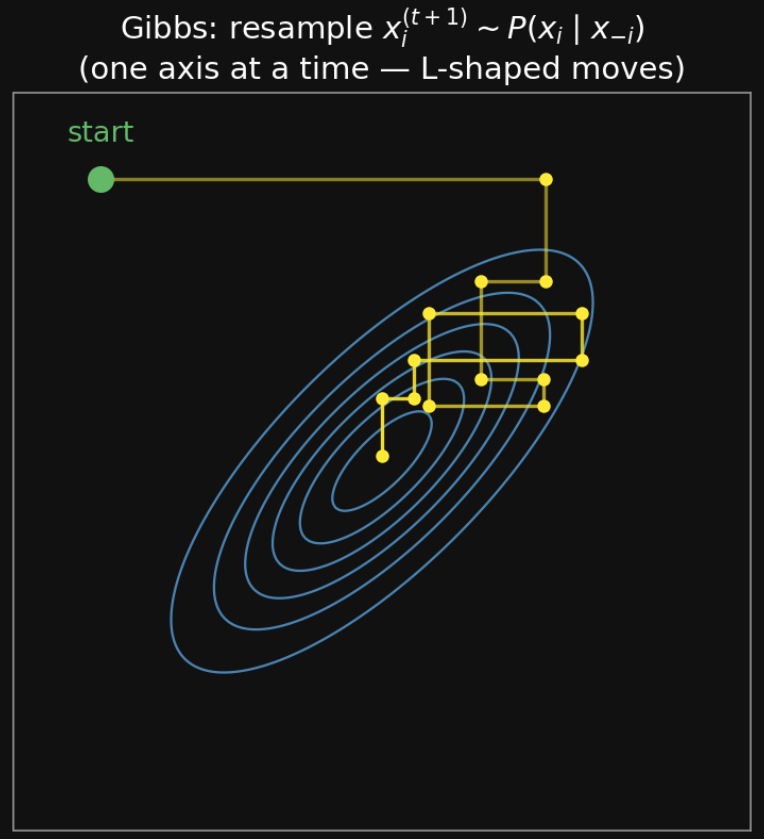
When you can sample each variable **given the rest**, resample one coordinate at a time:

$$x_i^{(t+1)} \sim P(x_i \mid x_{-i})$$

x_{-i} = all the other coordinates (everything except x_i).

- No accept/reject — **every move is accepted**.
- Moves are **axis-aligned** (one coordinate at a time).

Preview: in the Kemp sampler the θ_i step is **pure Gibbs** (conjugate); the hyperparameters need Metropolis.



MH vs Gibbs

Metropolis–Hastings

- works for **any** target
- propose + accept/reject
- needs only the **ratio** $P(x')/P(x)$
- proposal variance must be tuned

Gibbs

- needs the **conditionals** $P(x_i \mid x_{-i})$
- always accepts
- one coordinate at a time
- no tuning, but can mix slowly

Poll — does the starting point matter?

You run your MH chain from **two very different starting points**, for a long time. The histograms of visited states are…

A. the same — the chain forgets where it started

B. different — the start determines the answer

C. depends on the proposal

A — an **ergodic** chain forgets its start and converges to π = the target. That's the whole point of MCMC. **…if it has mixed** — which is exactly what the next demo is about.

Interactive: Gibbs & Metropolis–Hastings

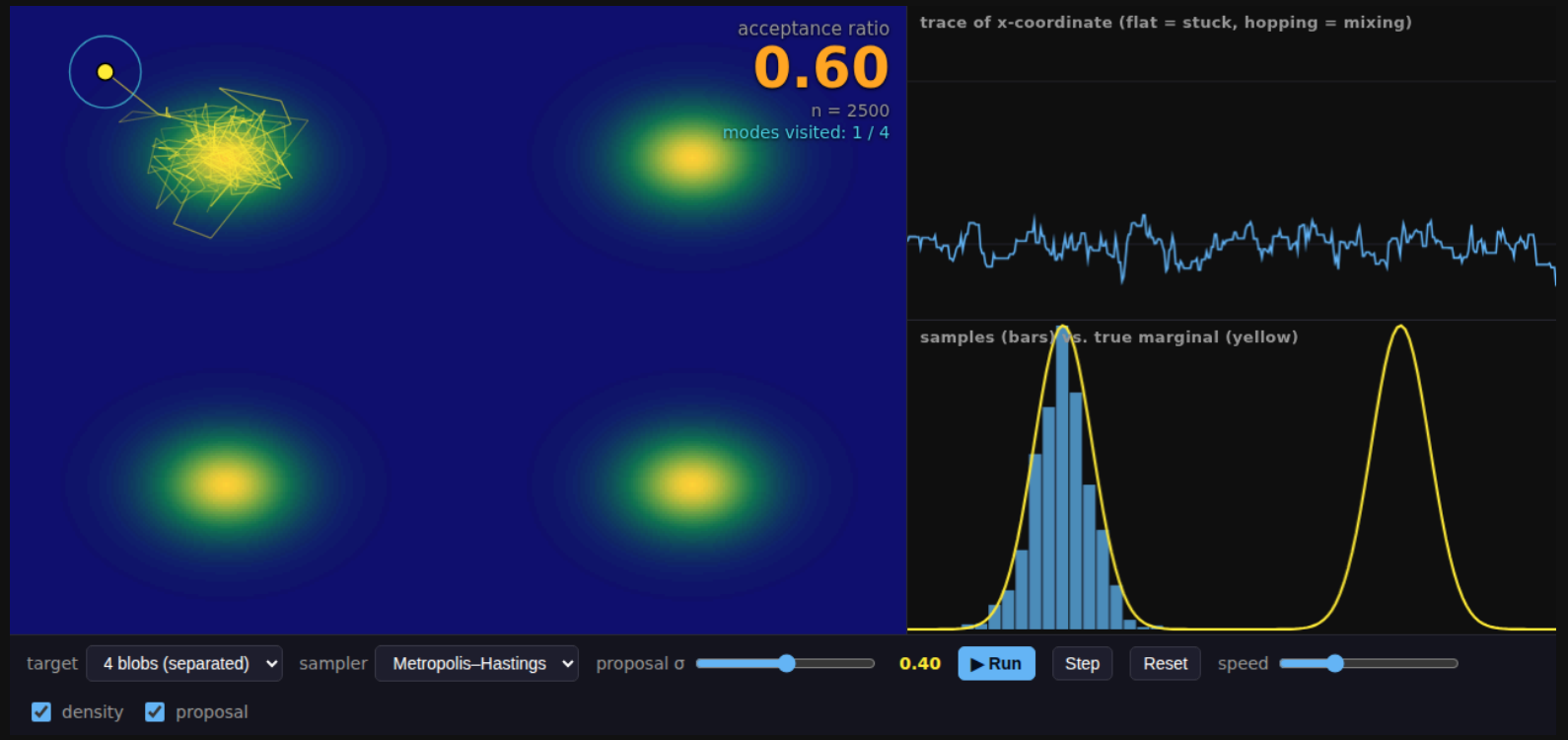
Live demo — what to watch

A multimodal 2-D Gaussian mixture. We control the **proposal variance** and watch the **acceptance ratio** and **mixing**.

- **σ too small** → acceptance ≈ 1, but a slow random walk — **stuck in one mode**.
- **σ too large** → almost everything rejected, acceptance ≈ 0 — chain frozen.
- **σ just right** → acceptance ≈ 0.2–0.5, the chain explores.
- **The catch** → even good local acceptance **can’t cross** between well-separated modes.

Gibbs & MH demo

If the live demo isn’t available



Trapped MH: acceptance is healthy (0.60) but **only 1 of 4 modes** was visited — the histogram fills just one peak. Good *local* acceptance ≠ good *global* mixing. Open `week07.html` to drive it live.

Poll — two runs disagree

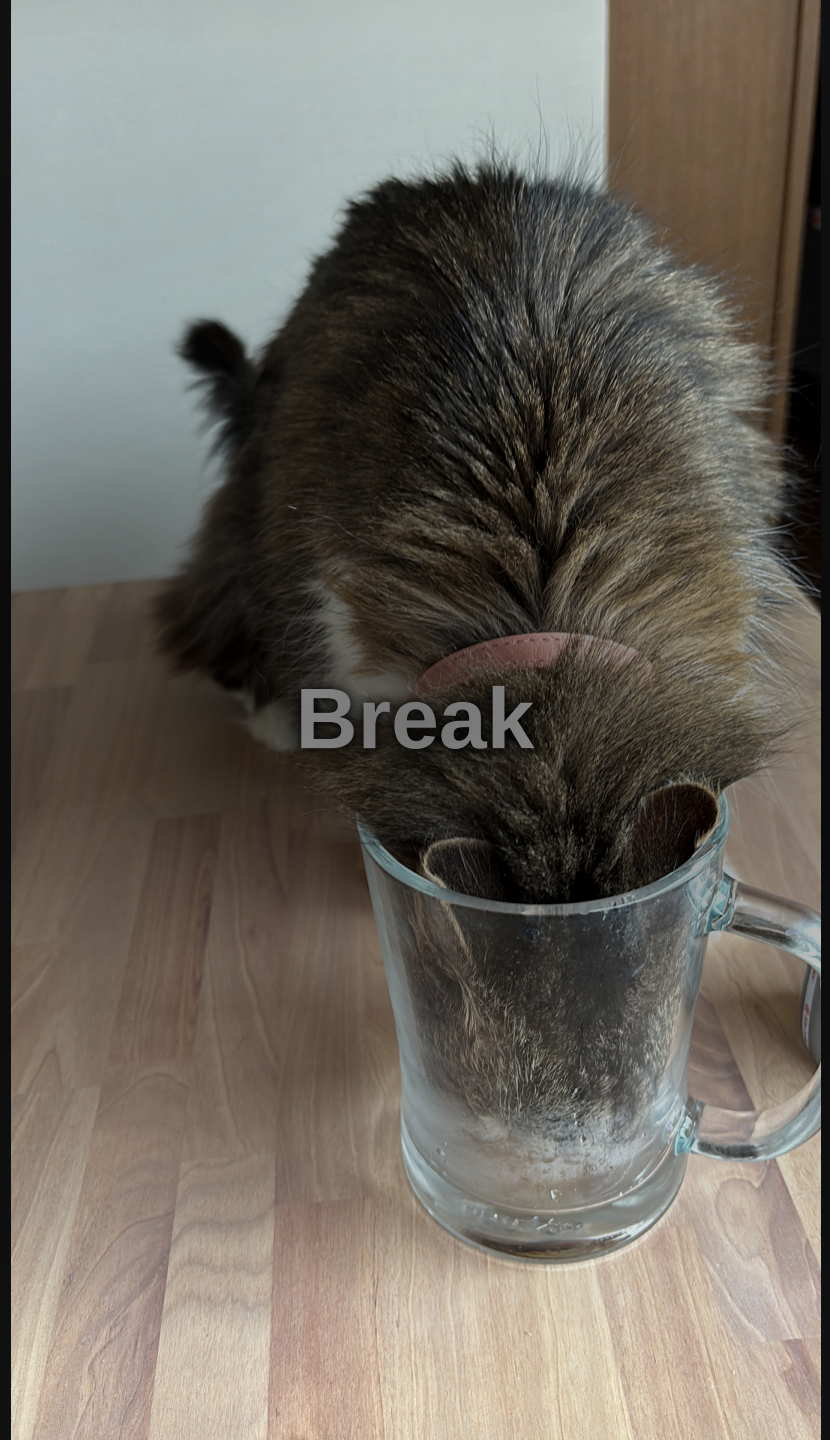
Two MH runs on the bimodal target give **very different histograms** after 1,000 steps. What went wrong?

A. neither chain has mixed — each got stuck in a different mode

B. MCMC is biased — this is expected

C. the target has no stationary distribution

A — poor mixing, not bias. Both chains are ergodic in theory (each *would* reach π eventually), but with a local proposal neither has crossed between modes in 1,000 steps — so each chain reflects only the mode it started in, and the two disagree. **That disagreement is the diagnostic:** run several chains from different starts; if they don't agree, none has mixed.



Programmable inference (GenJAX)

One picture: classical method ⇄ GenJAX

Classical

- basic Monte Carlo
- rejection sampling
- importance sampling
- K-particle IS / SMC
- particle filter
- MCMC / Metropolis–Hastings

GenJAX surface

- `simulate` + `vmap`
- filter `simulate` by the constraint
- `importance` → `(trace, log_weight)`
- `smc.ImportanceK(Target(…), k=N)`
- `smc.*` chained over observations
- **assemble it** from `update` / `assess`

The idea: probabilistic programming gives you importance sampling *for free*, and hands you the scoring primitive (`assess`) to **assemble** MCMC — exactly what we'll do for the Kemp model.

Deep-dive (1/4): the target and the skeleton

```python
@gen
def model():
    mu = normal(0.0, 1.0) @ "mu"   # prior
    y  = normal(mu, 0.5)  @ "y"    # likelihood
    return y

# score log p(mu, y=1.5) with assess
def log_p(mu):
    ch = C["mu"].set(mu).at["y"].set(1.5)
    logp, _ = model.assess(ch, ())
    return logp
```

- **Goal:** sample μ from its posterior after seeing $y = 1.5$.
- We pick **one tool**: `assess` — it returns $\log p(\mu, y)$ for any μ.
- Every MH step is the **same 3 lines**: *propose* → *score* → *accept*. The only thing that changes is the **proposal**.

Closed-form posterior here is $\mathcal{N}(1.2, 0.45)$ — so we can check the chain landed right.

Deep-dive (2/4): proposal A — a random walk

```python
def mh_step_A(key, mu, sigma=0.5):
    kp, ka = random.split(key)
    mu_new = mu + sigma * random.normal(kp)  # pro
    logA = log_p(mu_new) - log_p(mu)         # sco
    ok = jnp.log(random.uniform(ka)) < logA  # acc
    return jnp.where(ok, mu_new, mu)
```

```
start mu=0.0,  log_p=-5.65
propose mu'=0.266   logA=+1.42 → accept ✓
# uphill (closer to 1.2) ⇒ logA>0 ⇒ always accept
```

- **Random-walk proposal:** $\mu' = \mu + \sigma\,\varepsilon$, a small Gaussian step.
- It's **symmetric** ($q(\mu' \mid \mu) = q(\mu \mid \mu')$), so the q terms cancel and $\log A$ is **just** the score difference — the animation's rule.
- Run 4000 steps: mean 1.17, **accept rate 0.68**. Small steps, mostly accepted.

This is the workhorse proposal — the one in the interactive demo.

Deep-dive (3/4): proposal B — independent jumps

```python
def mh_step_B(key, mu):
    kp, ka = random.split(key)
    mu_new = random.normal(kp)          # propose ~
    logA = ( log_p(mu_new) - log_p(mu)
        + log_q(mu) - log_q(mu_new) )  # +q-correc
    ok = jnp.log(random.uniform(ka)) < logA
    return jnp.where(ok, mu_new, mu)
```

```
propose mu'=-2.378  logA=-25.6 → reject ✗
# jumped away from the posterior ⇒ logA≪0 ⇒ rejec
```

- **Independent proposal:** draw $\mu' \sim \mathcal{N}(0, 1)$ fresh — *ignore where you are*.
- The proposal is symmetric as a *bell curve*, but the **kernel isn't**: since it ignores μ, the forward and reverse probabilities differ, $q(\mu' \mid \mu) = \mathcal{N}(\mu'; 0, 1) \neq \mathcal{N}(\mu; 0, 1) = q(\mu \mid \mu')$. So you **must add** $\log q(\mu) - \log q(\mu')$ — drop it and the chain targets the *wrong* distribution.
- Run 4000 steps: mean 1.23, **accept rate 0.25**. Bold jumps, often rejected.

Deep-dive (4/4): same skeleton, your choice of proposal

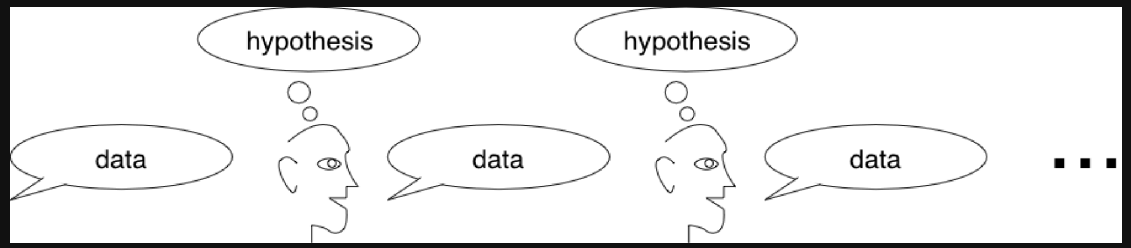
	A: random walk	B: independent
propose	$\mu + \sigma\varepsilon$	$\mu' \sim \mathcal{N}(0, 1)$
$q(\mu' \mid \mu) = q(\mu \mid \mu')$?	yes	no
q-correction	none	required
accept rate	0.68	0.25
post. mean	1.17	1.23

- **Both land on the truth** ($\mu \approx 1.2$) — MH is *correct* for any valid proposal.
- The proposal only changes **efficiency**: how often you accept, how fast you mix.
- The accept/reject rule never changed — only the proposal slotted into the skeleton did.

This is *programmable* inference: no black-box `mh()` in 0.10.3 — you assemble it, and you choose the proposal. (For Kemp, the proposal is a Gaussian step on $(\varphi, \log \kappa)$ — slide 60+.)

MCMC with People

What if a person is the accept step?



Show someone two stimuli — *“which looks more like a cat?”* — and treat their **choice** as the Metropolis accept/reject decision.

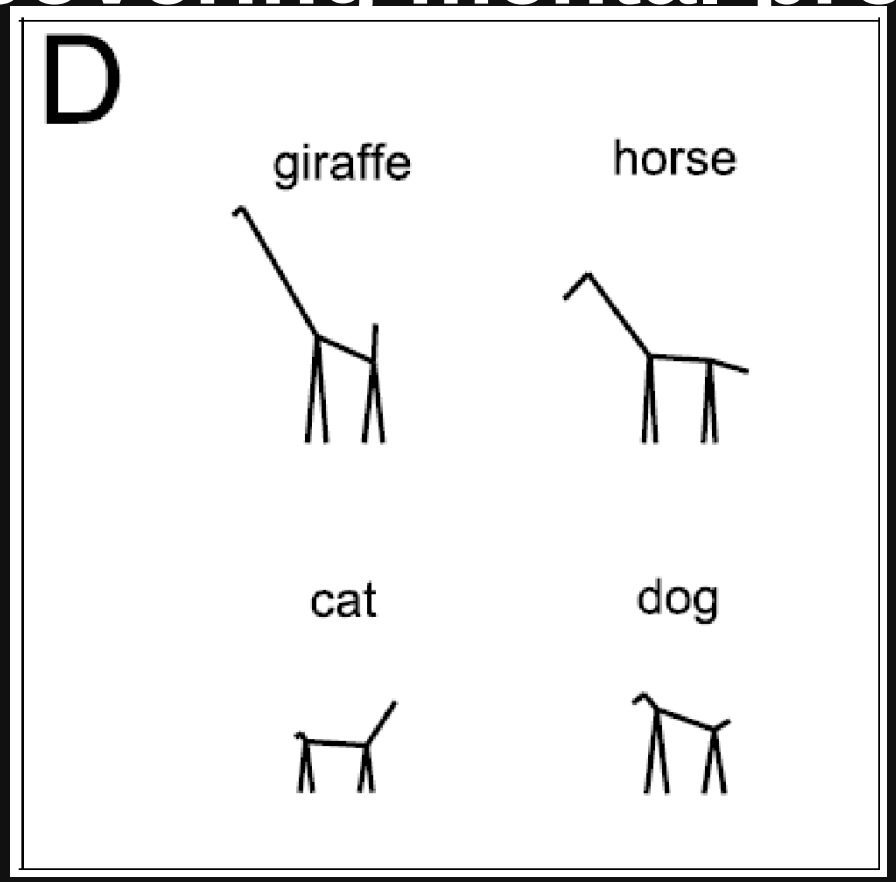
The chain alternates: a **hypothesis** in the head produces **data** (a response), which the next person reads as data and forms a new hypothesis.

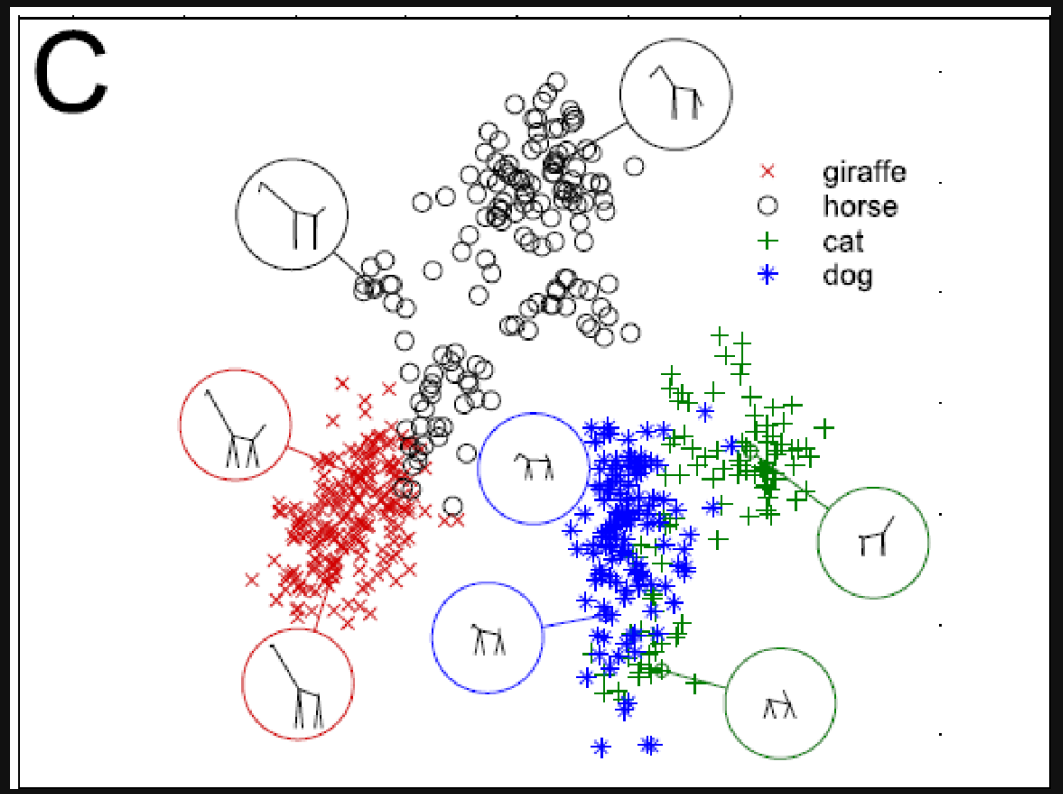
The chain converges to the prior in their heads

If learners **sample from their posterior** $P(h \mid d) \propto P(d \mid h)\, P(h)$, then the Markov chain on hypotheses **converges to the prior** $P(h)$ — the inductive biases in their heads.

You can read out someone's prior by running them as an MCMC chain.
(Bartlett 1932; Kirby 2001; Griffiths & Kalish 2007; Sanborn et al. 2010)

Recovering mental prototypes

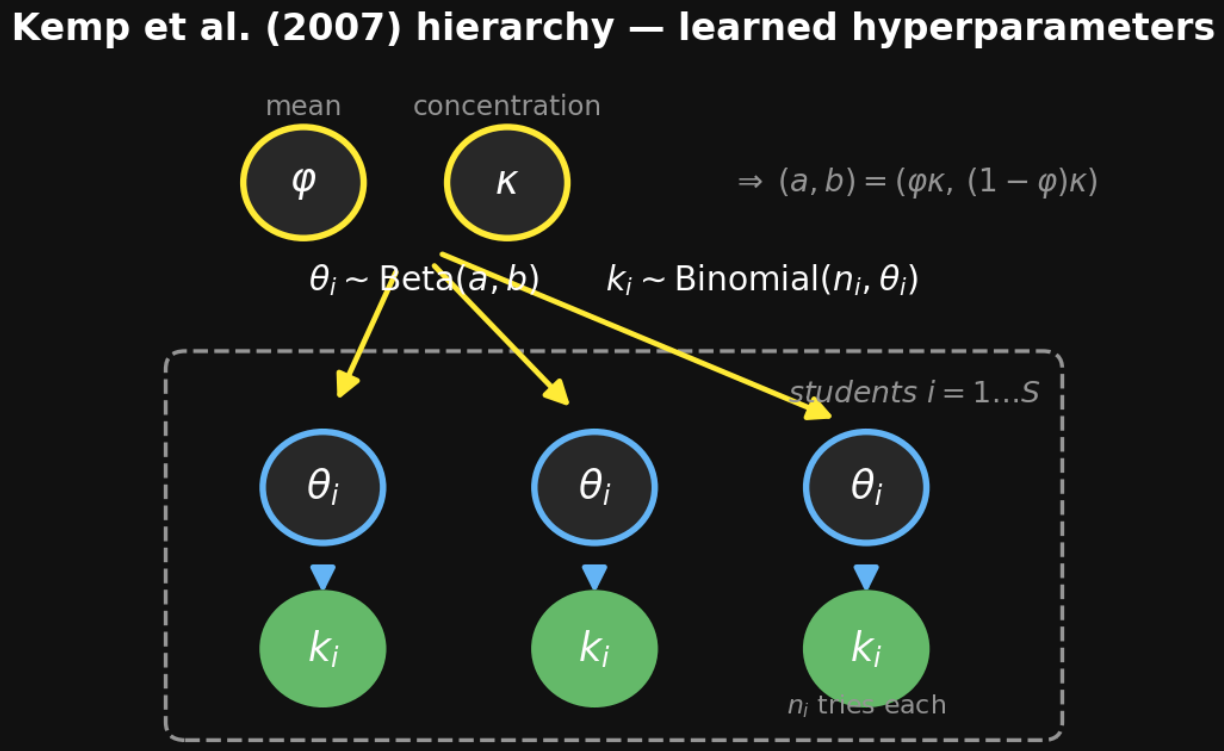




Sanborn & Griffiths (2007): MCMCP over a 9-D stick-figure space recovers each category's mental prototype — giraffe, horse, cat, dog separate cleanly.

A sampler for the Kemp hierarchy

The model — from Week 4



Each student i has a tonkatsu rate $\theta_i \sim \mathrm{Beta}(a, b)$; we observe counts $k_i \sim \mathrm{Binomial}(n_i, \theta_i)$.

The hyperparameters (a, b) are **unknown** and **learned** — the overhypotheses. (Kemp, Perfors & Tenenbaum 2007 — no DPMM here.)

We'll learn them through the **mean** $\varphi = \frac{a}{a+b}$ and **concentration** $\kappa = a + b$ (the figure's top nodes) — easier to put priors on.

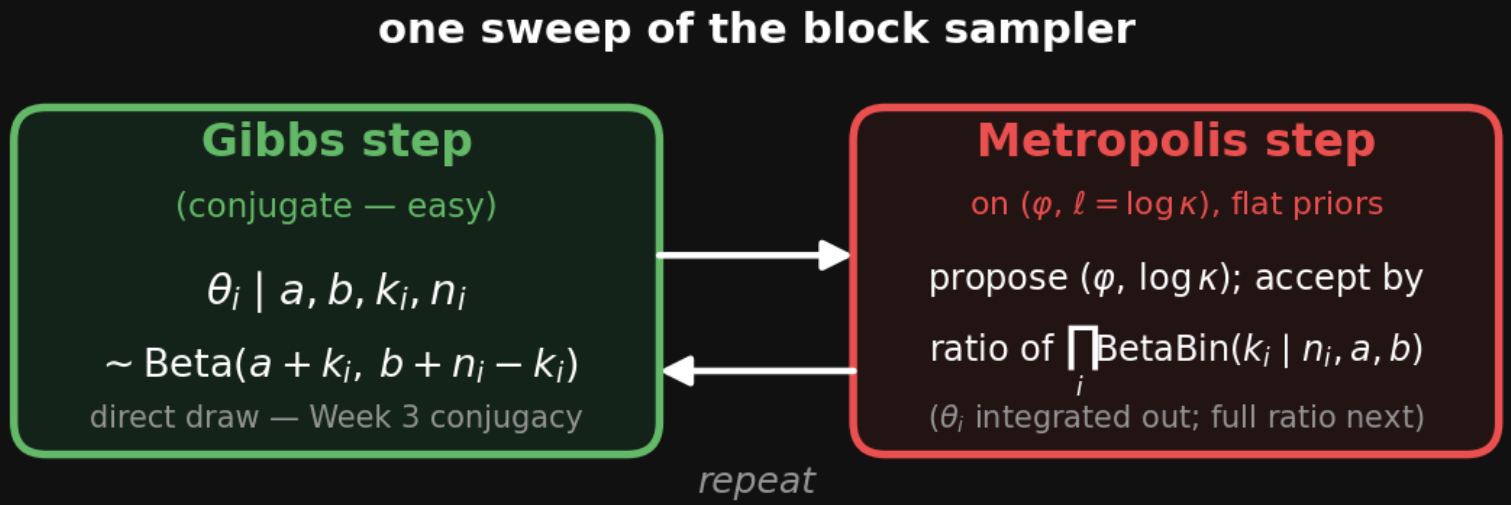
Why we need a sampler

The joint posterior $p(a, b, \{\theta_i\} \mid \text{data})$ has **no clean closed form**.

The textbook (T3 Ch 12) used **importance sampling** over the Beta-Binomial marginal — but with a broad hyperprior, most sampled (a, b) explain the data poorly, so the estimate wobbles. The **“blunt tool.”**

Now we build the sharp one: a chain that spends its samples where the posterior mass actually is.

The recipe — two steps per sweep



One half is free (conjugate Gibbs); the other half needs Metropolis. Alternate them.

(BetaBin = the Beta-Binomial marginal likelihood — defined in two slides.)

Step 1 — Gibbs the θ_i (free)

Hold (a, b) fixed. Each student's rate has a **conjugate** update — *exactly Week 3*:

$$\theta_i \mid a, b, k_i, n_i \;\sim\; \mathrm{Beta}(a + k_i,\; b + n_i - k_i)$$

A direct draw — no rejection, no tuning. You've had this since Week 3 conjugacy: prior $\mathrm{Beta}(a, b)$ + k_i successes in n_i tries → posterior $\mathrm{Beta}(a + k_i, b + n_i - k_i)$.

Integrate out θ_i: the Beta-Binomial marginal

To score the hyperparameters (a, b), we don't want each θ_i in the way — so **average it out** (the marginal likelihood of a student's count):

$$\mathrm{BetaBin}(k \mid n, a, b) = \int_0^1 \underbrace{\mathrm{Binomial}(k \mid n, \theta)}_{\text{likelihood}} \underbrace{\mathrm{Beta}(\theta \mid a, b)}_{\text{prior}} \, d\theta = \binom{n}{k} \frac{B(a + k,\ b + n - k)}{B(a, b)}$$

$\binom{n}{k}$ = the **binomial coefficient** ("n choose k"). $B(\alpha, \beta)$ = the **Beta function**, the normalizing constant of $\mathrm{Beta}(\alpha, \beta)$. Because Beta-Binomial is conjugate (Week 3), this integral is **closed-form** — no sampling needed. This is *why* θ_i vanishes from the accept ratio.

Step 2 — Metropolis the hyperparameters: propose

(a, b) are **not** conjugate to anything. Reparametrize to **mean** and **concentration**, and **work in** $(\varphi,\ \ell)$ where $\ell = \log \kappa$:

$$\varphi = \frac{a}{a+b} \in (0,1), \qquad \kappa = a + b > 0, \qquad \ell = \log \kappa$$

- **Priors:** flat (uniform) on **both** $\varphi \in (0,1)$ and ℓ (improper-flat — no bounds to track, so the prior ratio is exactly 1). Flat on $\ell = \log \kappa$ *is* the log-uniform prior on κ — spans orders of magnitude.
- **Propose** a symmetric Gaussian step on each (tune the two widths separately):

$$\varphi' = \varphi + \epsilon_\varphi, \quad \epsilon_\varphi \sim \mathcal{N}(0, \sigma_\varphi^2); \qquad \ell' = \ell + \epsilon_\ell, \quad \epsilon_\ell \sim \mathcal{N}(0, \sigma_\ell^2)$$

- **Map back** to evaluate the model: $\kappa' = e^{\ell'}, \ \ a' = \varphi' \kappa', \ \ b' = (1-\varphi')\kappa'$.
- If $\varphi' \notin (0,1)$ the flat prior is $0 \Rightarrow$ **reject** (set $A = 0$). $\sigma_\varphi, \sigma_\ell$ = the proposal widths you tune for mixing — the demo's knob.

Step 2 — Metropolis: the accept ratio

The general Metropolis–Hastings ratio has four factors. With **this** choice, three of them are 1:

$$A = \min\left(1,\ \underbrace{\frac{\prod_i \mathrm{BetaBin}(k_i \mid n_i, a', b')}{\prod_i \mathrm{BetaBin}(k_i \mid n_i, a, b)}}_{\text{marginal likelihood}} \times \underbrace{\frac{\pi(\varphi', \ell')}{\pi(\varphi, \ell)}}_{=1\ \text{(flat)}} \times \underbrace{\frac{q(\varphi, \ell \mid \varphi', \ell')}{q(\varphi', \ell' \mid \varphi, \ell)}}_{=1\ \text{(symmetric)}} \times \underbrace{J}_{=1\ \text{(no transform)}}\right)$$

$a' = \varphi'\kappa',\ b' = (1-\varphi')\kappa',\ \kappa' = e^{\ell'}$ — the proposed hyperparameters (from the previous slide).

- **prior ratio $= 1$:** both priors are flat in (φ, ℓ).
- **proposal ratio $= 1$:** the Gaussian step is symmetric.
- **Jacobian $J = 1$:** the prior lives on the **same** coordinates (φ, ℓ) we sample in — no change of variables. If you instead put a flat prior on (a, b) and sampled in (φ, ℓ), you *would* owe a Jacobian. Defining the prior in the sampling coordinates avoids it.

So A collapses to **just the marginal-likelihood ratio**. The $\binom{n_i}{k_i}$ in each BetaBin is constant in (a, b), so it cancels too — drop it. **Note:** this step scores (a, b) through the marginal, so the θ_i you just drew in the Gibbs step are **not** used here — they're integrated out.

In GenJAX: the sampler you assemble

The recipe

- θ_i step: direct Beta draw
- (φ, κ) step: MH on the marginal
- alternate, collect samples after **burn-in**

burn-in = the early samples, discarded so the chain can forget its start (Week 6's mixing) and reach π before you count.

In GenJAX

- Gibbs step → `beta(a+k, b+n-k).sample`
- MH step → score the **closed-form BetaBin** (slide 52) directly — *not* the model's joint `assess`, which would still contain θ_i.
- the MH skeleton is the deep-dive's `mh_step`. The assignment builds this same chain a slightly different way — next slide.

The chain spends its samples where the mass is — **sharp**. Importance sampling weighted a cloud of prior draws — **blunt**. Same model, better tool.

Two ways to score the same chain

This lecture — collapsed

$$A \propto \frac{\prod_i \mathrm{BetaBin}(k_i \mid n_i, a', b')}{\prod_i \mathrm{BetaBin}(k_i \mid n_i, a, b)}$$

- θ_i **integrated out** (closed form)
- flat prior on $\ell = \log \kappa$
- the elegant shortcut — fewest moving parts

The assignment — explicit

$$\log A = \sum_i \log \frac{\mathrm{Beta}(\theta_i|a',b')}{\mathrm{Beta}(\theta_i|a,b)} + \log \frac{p(\ell')}{p(\ell)}$$

- scores the θ_i you **just drew** in the Gibbs step
- weak **proper** $\log$-normal prior on κ (a real prior ratio)
- you see Gibbs **and** Metropolis working together

Same chain, same target π — they just differ in *what you condition on*. Collapsing θ_i gives lower-variance hyperparameter moves (less Monte-Carlo noise per step); keeping θ_i is more general — it's what you do the moment a step *isn't* conjugate and you can't integrate by hand. The assignment uses the explicit form so you build both moves yourself.

Does the choice matter for what's learned? The **answer** (the posterior over the overhypothesis) is identical either way. But lower variance per step means the collapsed chain pins down (φ, κ) in **fewer sweeps** — it *learns the overhypothesis faster per unit of compute.* That's the only sense in which the sampler choice touches learning here: speed, not conclusion.

Wrapping up

The toolkit ladder

- **Monte Carlo** — average samples to estimate an expectation.
- **Importance sampling** — sample the wrong distribution, reweight.
- **Particle filter** — importance sampling, updated over time.
- **MCMC (MH + Gibbs)** — design a chain whose stationary distribution is your target.
- **Applied:** read out a person's prior (MCMCP); learn the Kemp hierarchy's hyperparameters.

Each rung bought a sharper tool for a harder target.

Why this still matters in 2026. You'll rarely hand-write an accept ratio again — but the *sample → score → reweight-or-accept* loop didn't go away, it got a **learned proposal**. A **diffusion model** is a learned reverse-MCMC; **RLHF** samples from a policy and reweights by a reward; **best-of-N** is importance sampling. The mechanics you built today are the conceptual core of how today's models are trained, steered, and aligned.

Next week + the assignment

- **Assignment:** build the Kemp sampler — Gibbs the θ_i, Metropolis the (φ, κ). Stencil + Colab on the assignments page.
- **Week 8:** signal detection theory, MDPs, reinforcement learning — *from inferring beliefs to choosing actions.*