Advanced inference in probabilistic programs

Brooks Paige

Inference thus far

- Likelihood weighting / importance sampling
- MCMC (single-dimension, coded by hand)
- “Lightweight” Metropolis-Hastings (update one random choice at a time, by re-running the remainder of the program)

Inference: this talk

How can we make inference more computationally efficient?

- **Sequential Monte Carlo** uses importance sampling as a building block for an inference algorithm that can succeed in models with higher-dimensional latent spaces
- Algorithms which extend SMC: **Particle MCMC**, and **asynchronous SMC**
- What sort of proposal distributions should we be simulating from in these methods? Can we **learn importance sampling proposals** automatically?

Inference in Anglican

```
(doquery :algorithm model [args] options)
```

- How do you implement an inference algorithm in Anglican? (JW will show you this afternoon)

- Two important special forms are the interface between model code and inference code:

```
(sample ...)        (observe ...)
```

- **Q:** what kinds of inference algorithms can we develop and implement using this interface?

Incremental evidence

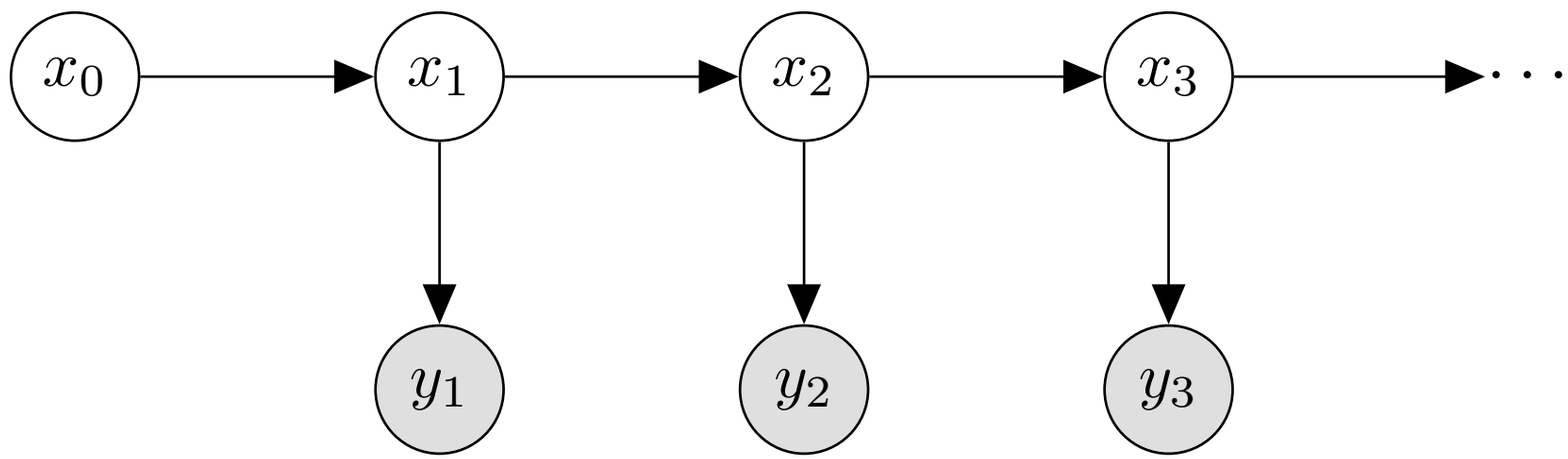
- If we can write our programs in such a way that we see **early, incremental evidence** then we can use more efficient inference algorithms.

```
(defquery monolithic-observe []
  ... ;; many sample statments
    (sample ...)
    (sample ...)
    (sample ...)
  ... ;; single observe /
      ;; conditioning statement
      ;; at the end
    (observe ...))
```

- Intuition: sample statements which come after observe statements can be informed by the data

```
(defquery incremental-observe []
  (loop ...
    ;; interleaved sample and
    ;; observe statements
    (sample ...)
    (observe ...)
    (recur ...)))
```

Hidden Markov model

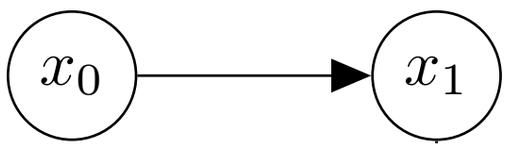


Hidden Markov model

x_0

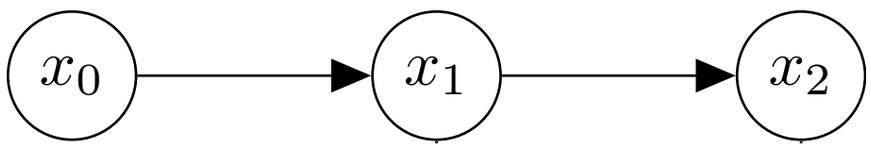
Place a massive `observe` statement at the end

Hidden Markov model



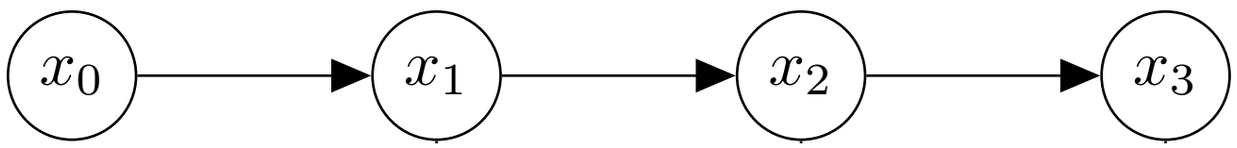
Place a massive `observe` statement at the end

Hidden Markov model



Place a massive `observe` statement at the end

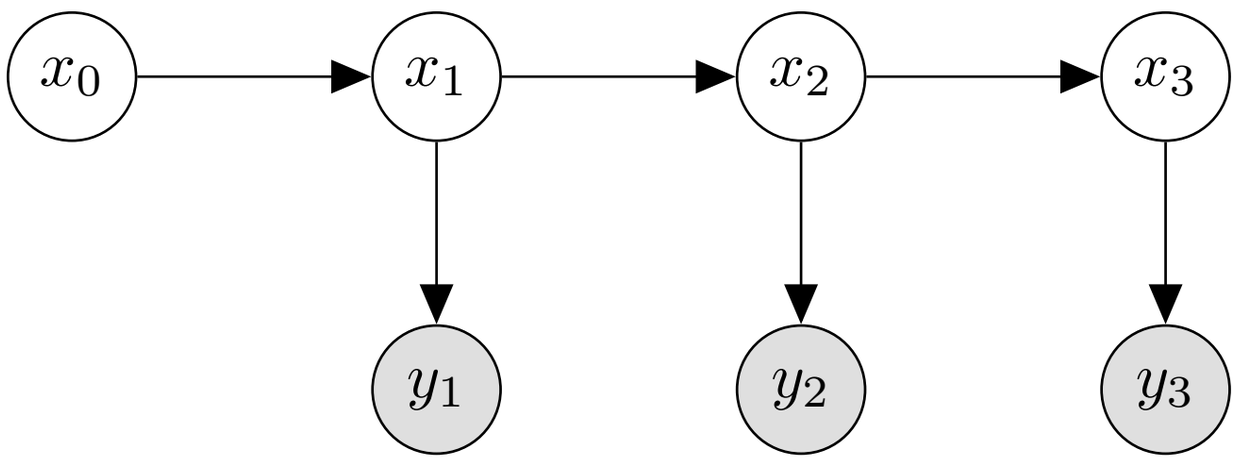
Hidden Markov model



Place a massive `observe` statement at the end

Hidden Markov model

No “feedback” until all random variables have been sampled



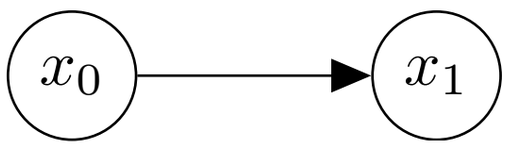
Place a massive `observe` statement at the end

Hidden Markov model

x_0

Place `observe` statements as early as possibly

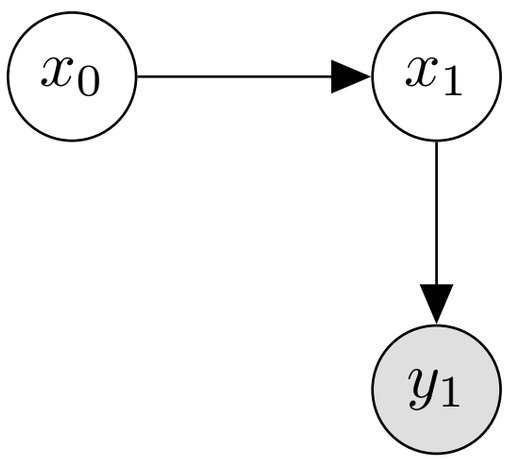
Hidden Markov model



Place `observe` statements as early as possibly

Hidden Markov model

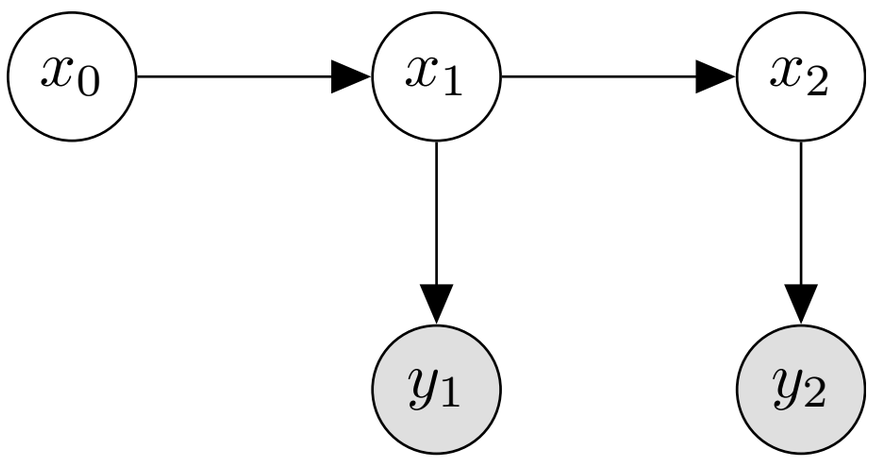
Does y_1 have high probability given x_0 and x_1?



Place `observe` statements as early as possibly

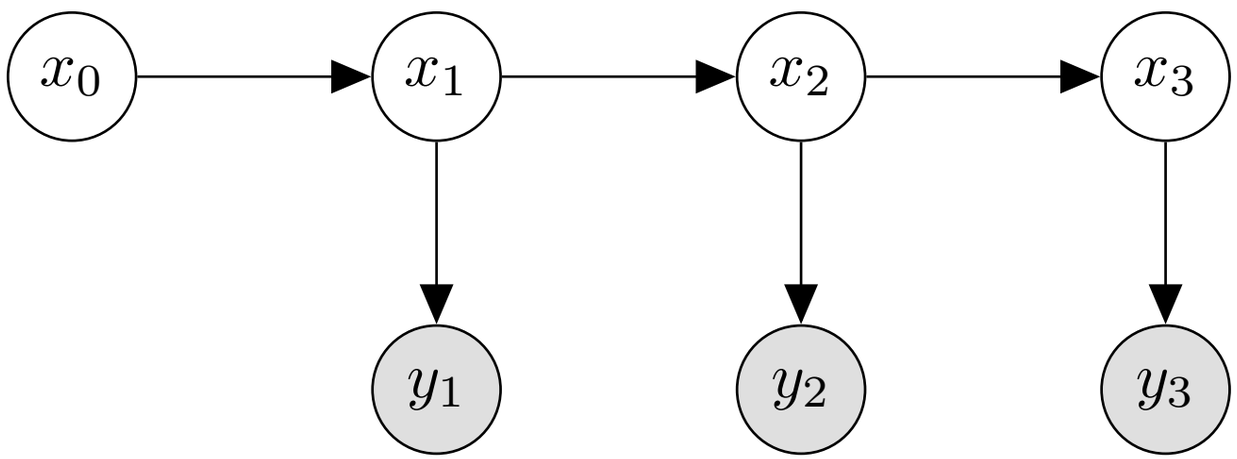
Hidden Markov model

Does y_2 have high probability given x_0, x_1, and x_2?



Place `observe` statements as early as possibly

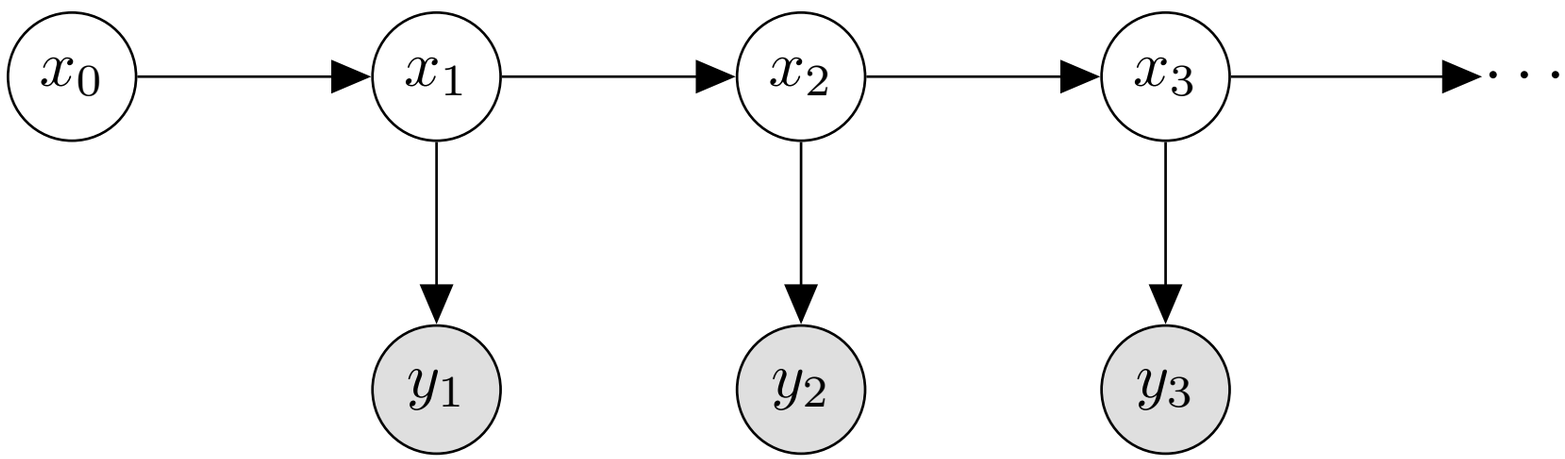
Hidden Markov model



Place `observe` statements as early as possibly

Hidden Markov model

Incremental evidence == computational efficiency?



Place `observe` statements as early as possibly

Incremental evidence

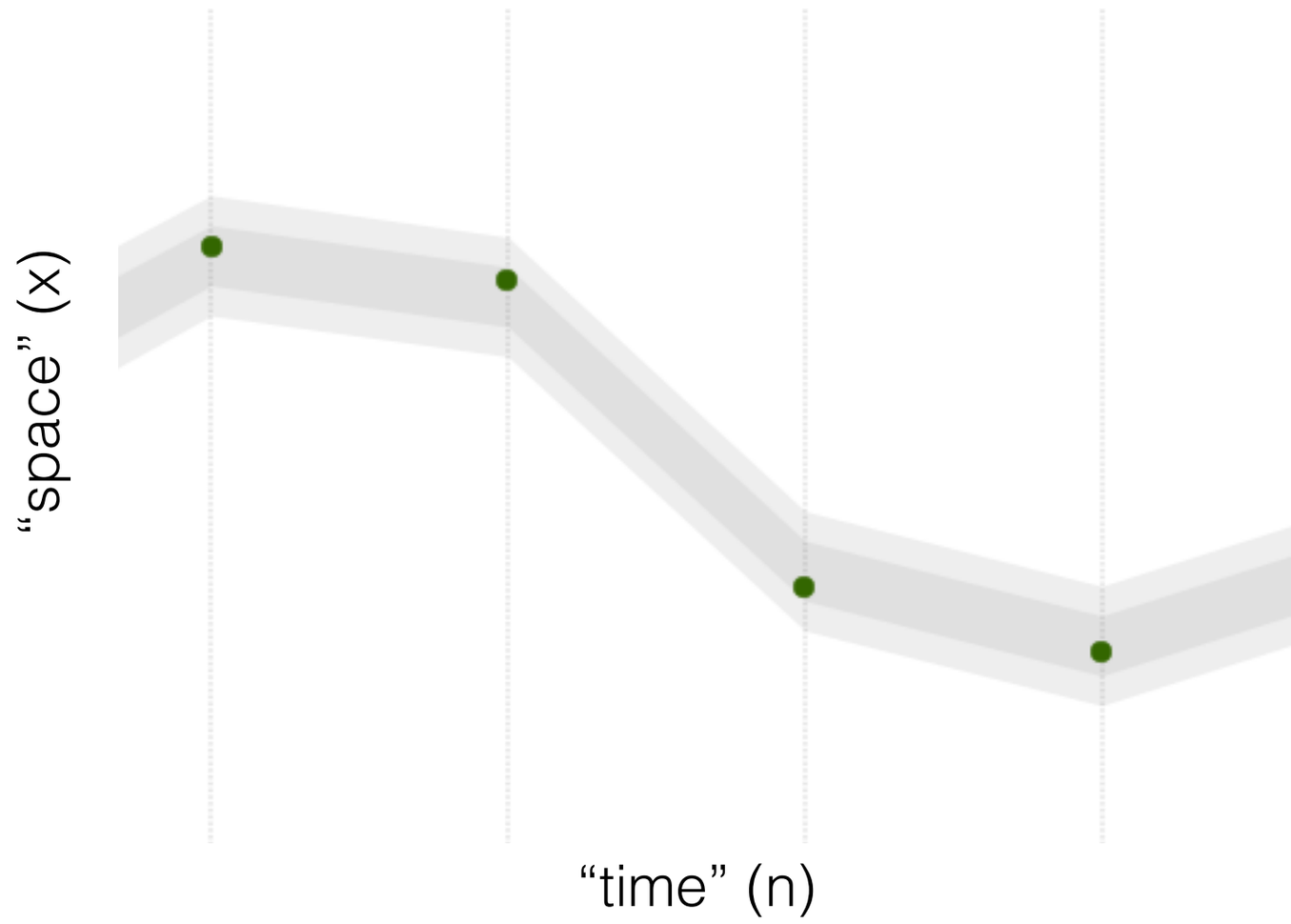
- Many models and settings are naturally written incrementally!
  - Canonical example: time series models (observe at discrete timesteps)
  - Planning problems (observe at discrete timesteps)
  - Models which factor into global and “local” (per-datapoint) observes, such as mixture models and many multilevel Bayesian models
  - Models such as image synthesis, where the entire “canvas” is always visible and can be evaluated according to a fitness function at any time

State-space models

- Running example: inference in state-space models

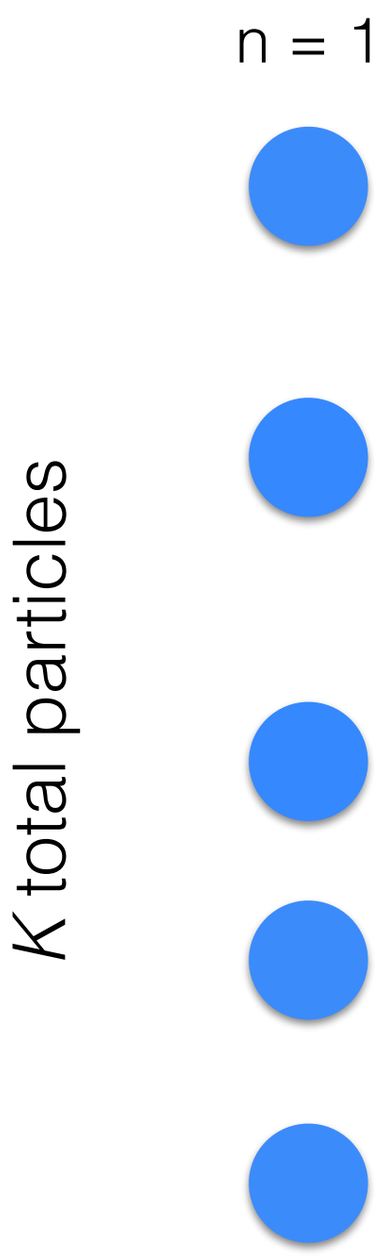
- Observed data y_n and latent state x_n

- Inference goals: estimate latent state; predict future data; estimate marginal likelihood



$$p(x_{0:N}, y_{0:N}) = \prod_{n=0}^{N} g(y_n|x_{0:n}) f(x_n|x_{0:n-1})$$

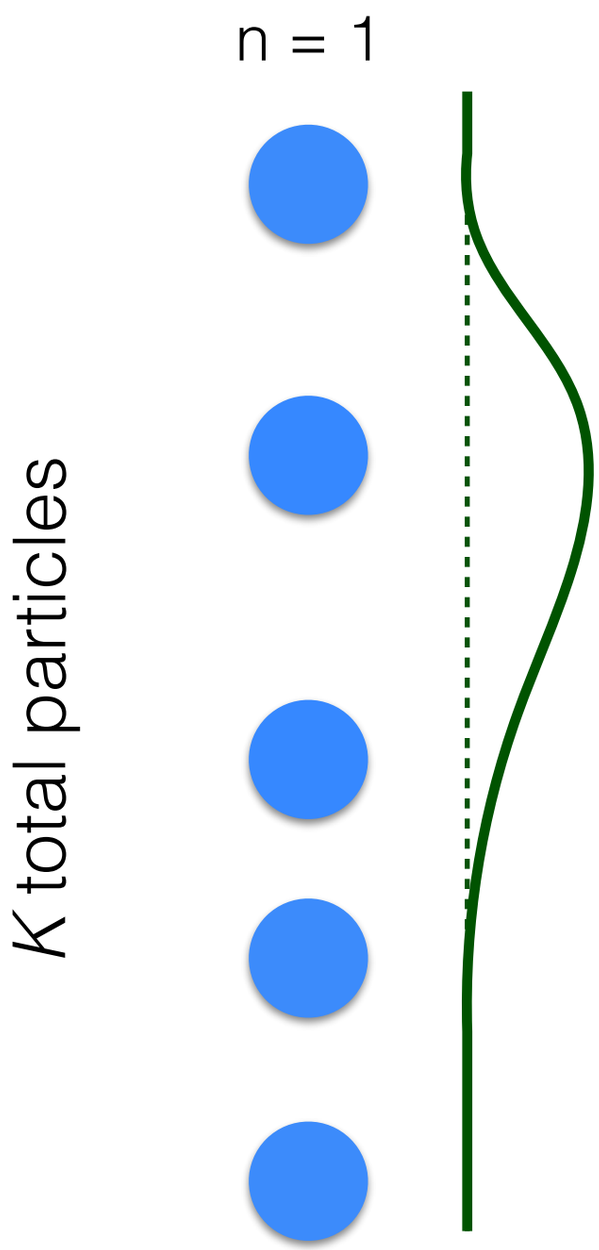
Sequential Monte Carlo



- Basic idea: approximate the posterior distribution using a weighted set of *K* particles $x_{0:n}^{(k)}$

- $p(x_{0:n}|y_{0:n}) \approx \sum_{k=1}^{K} w_n^{1:K} \delta_{x_{0:n}^{(k)}}(x_{0:n})$

Sequential Monte Carlo

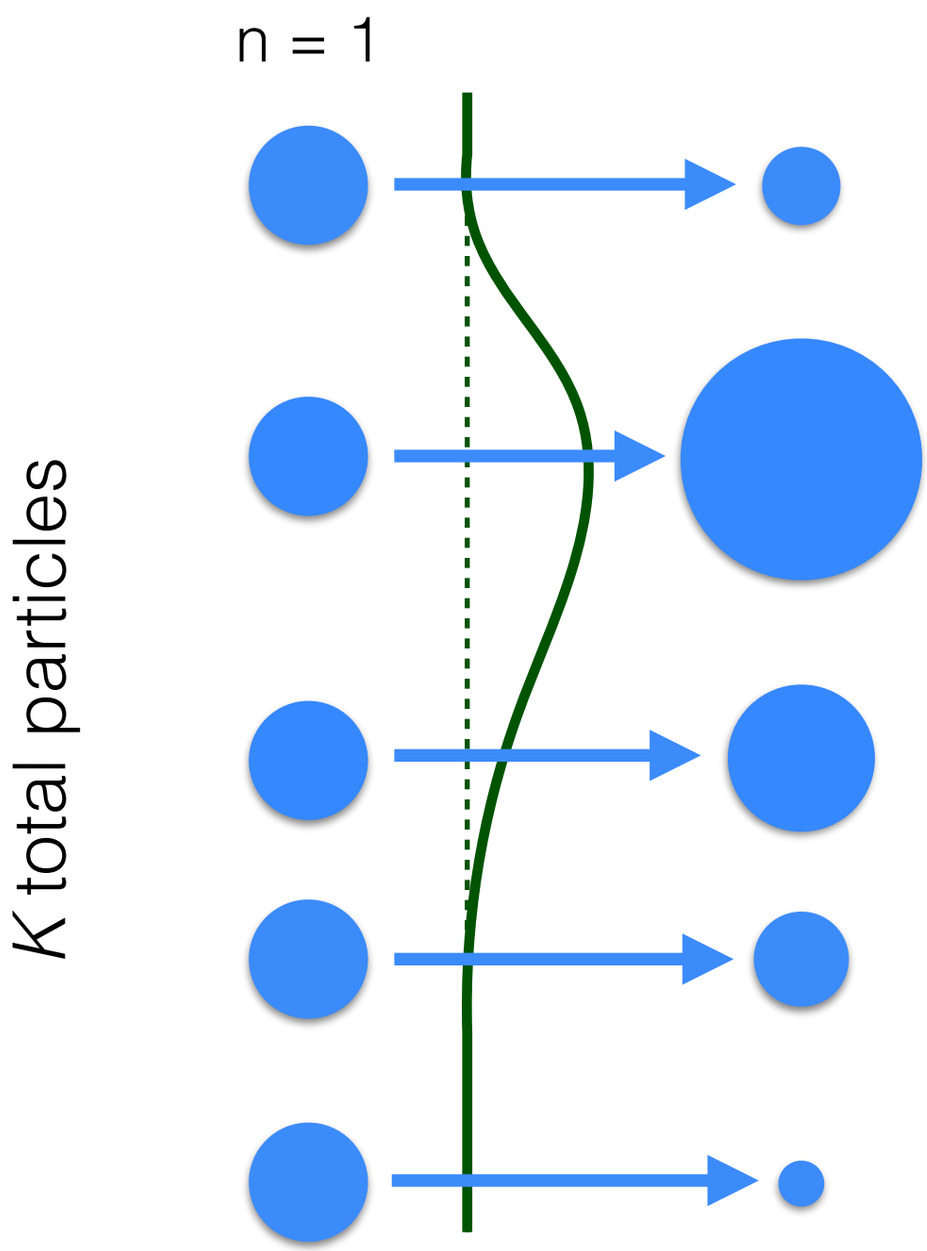


- Each particle is assigned an (unnormalized) weight based on its likelihood W_n^k

- $p(x_{0:n}|y_{0:n}) \approx \sum_{k=1}^{K} w_n^{1:K} \delta_{x_{0:n}^{(k)}}(x_{0:n})$

- $w_n^k \propto W_n^k$

Sequential Monte Carlo

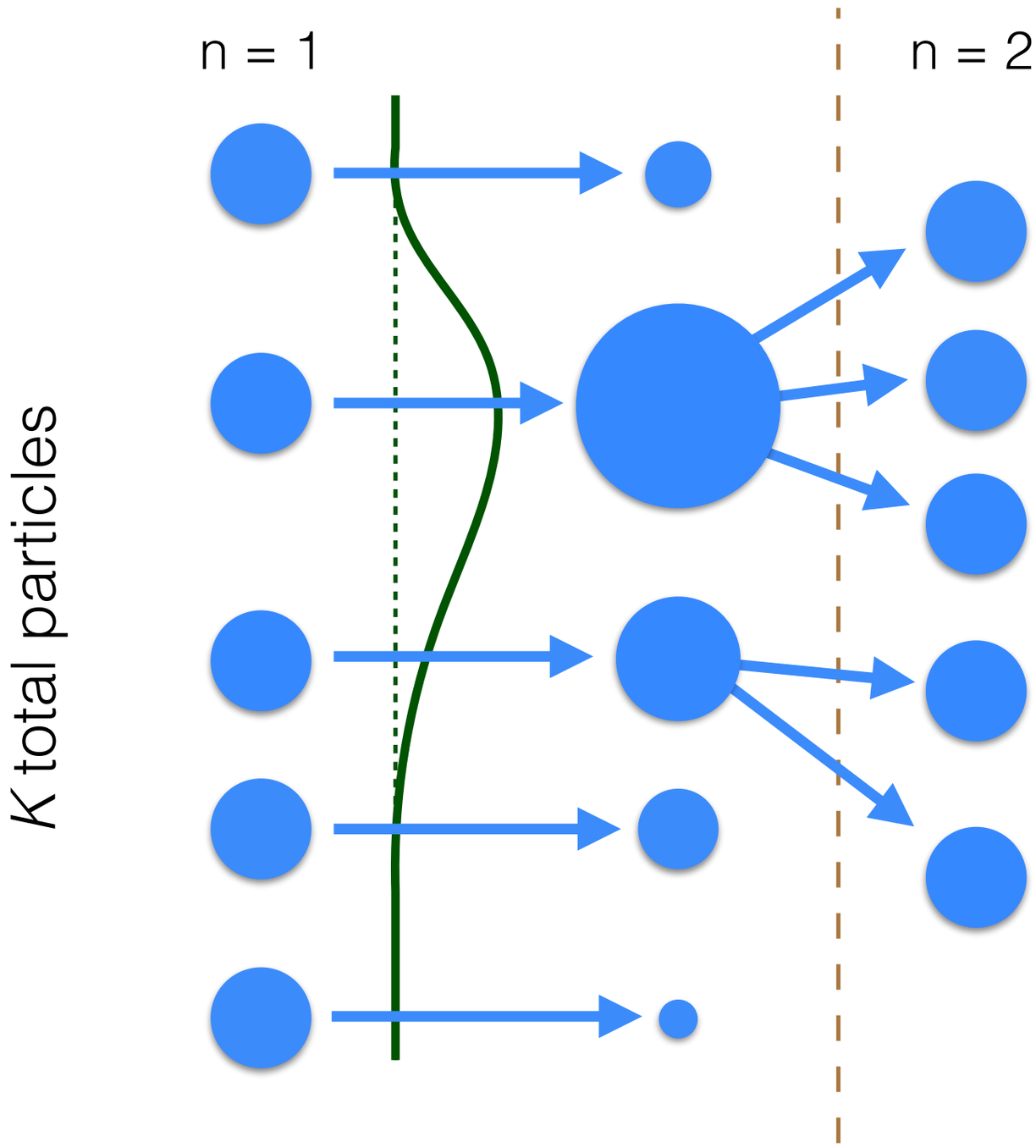


- Each particle is assigned an (unnormalized) weight based on its likelihood W_n^k

- $p(x_{0:n}|y_{0:n}) \approx \sum_{k=1}^{K} w_n^{1:K} \delta_{x_{0:n}^{(k)}}(x_{0:n})$

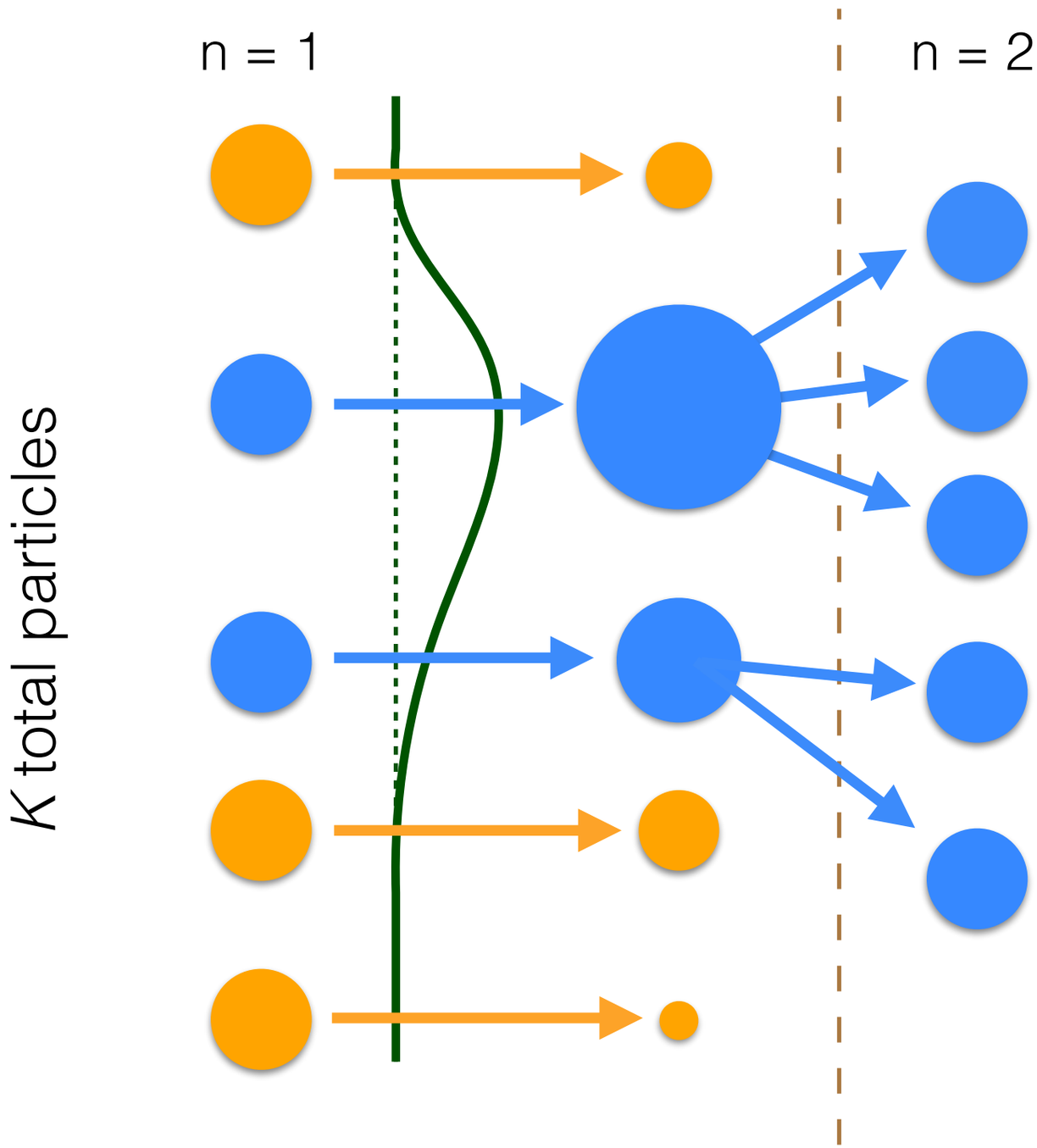
- $w_n^k \propto W_n^k$

Sequential Monte Carlo



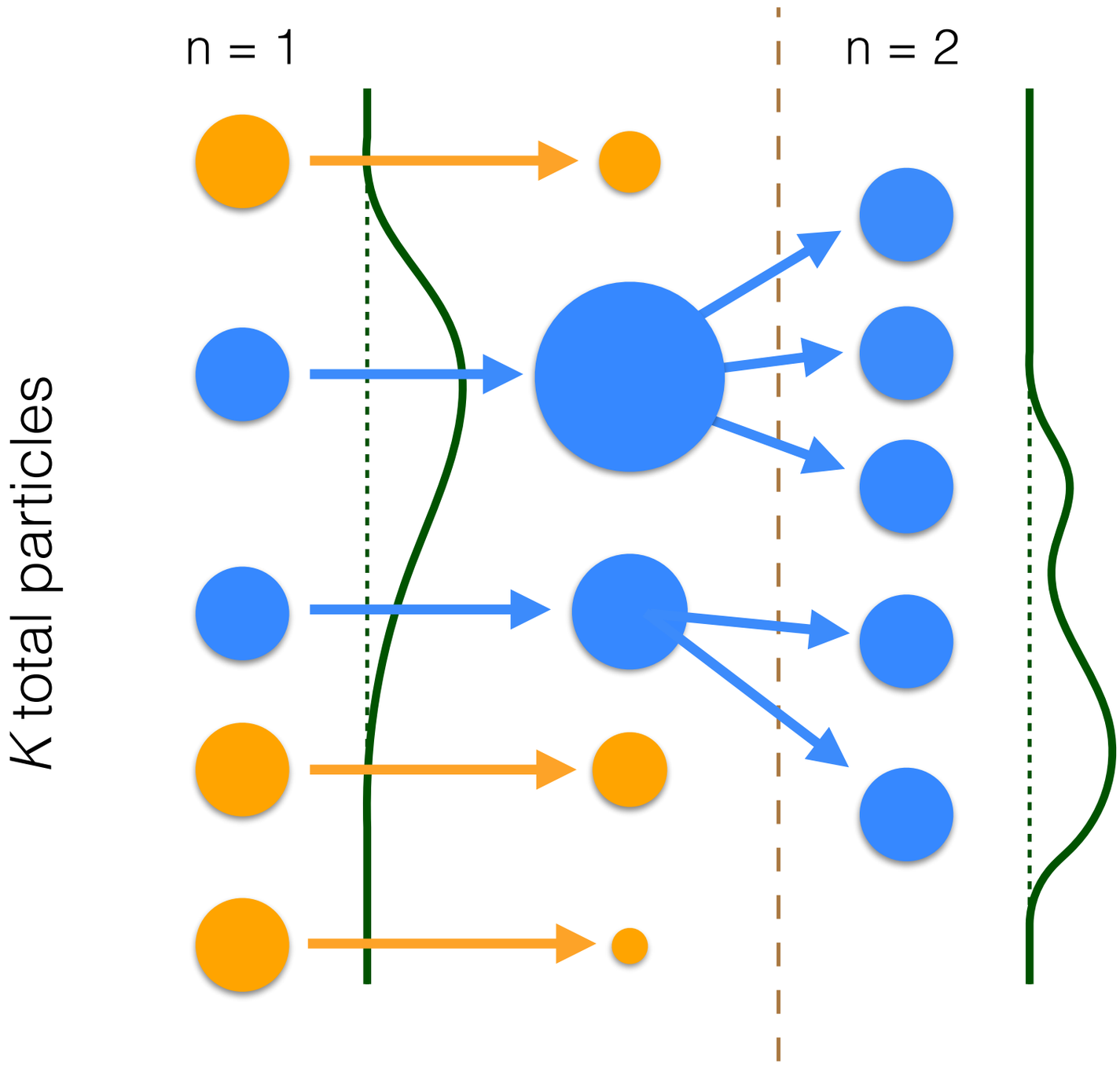
- Particles are **resampled** according to their weights, then **simulated** forward
- Each particle has zero or more children
- Number of children M_n^k is proportional to the weight W_n^k

Sequential Monte Carlo



- Particles with low weight are discarded, and particles with high weight are replicated
- Better-than-average particles are replicated more often
- $\mathbb{E}[M_k^n | W_n^{1:K}] = \frac{W_n^k}{\overline{W}_n}$

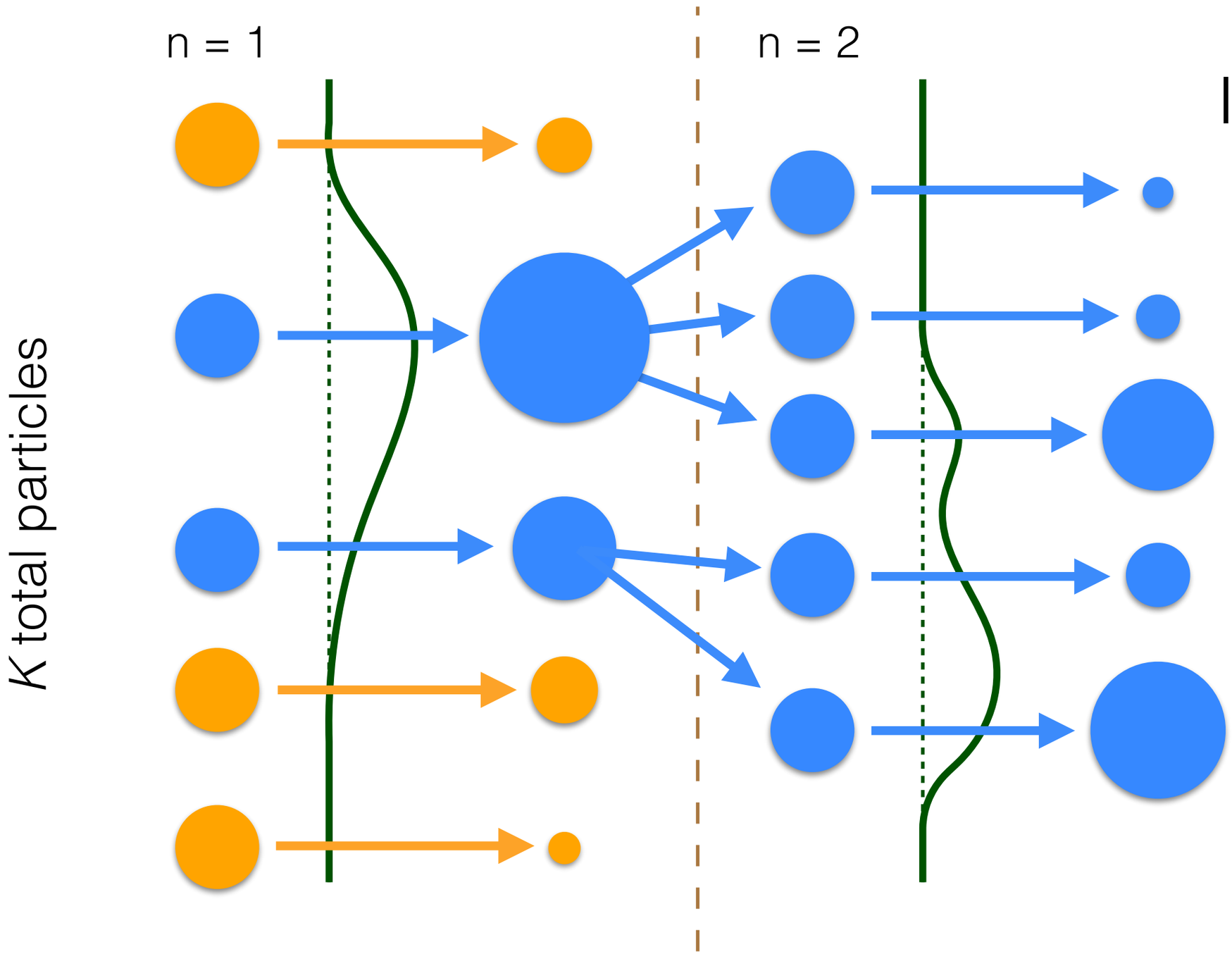
Sequential Monte Carlo



Iteratively,

- **simulate**
- **weight**
- **resample**

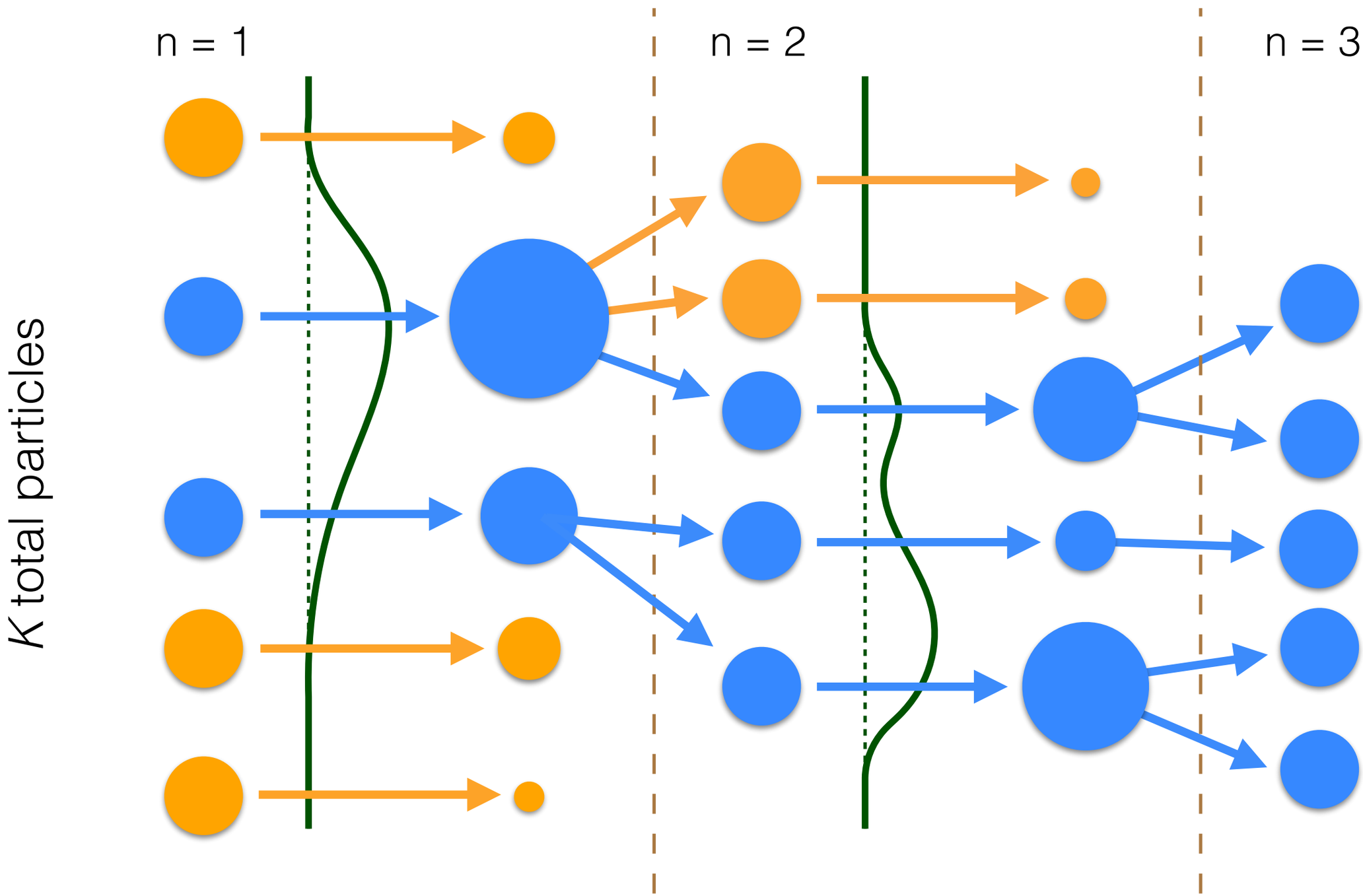
Sequential Monte Carlo



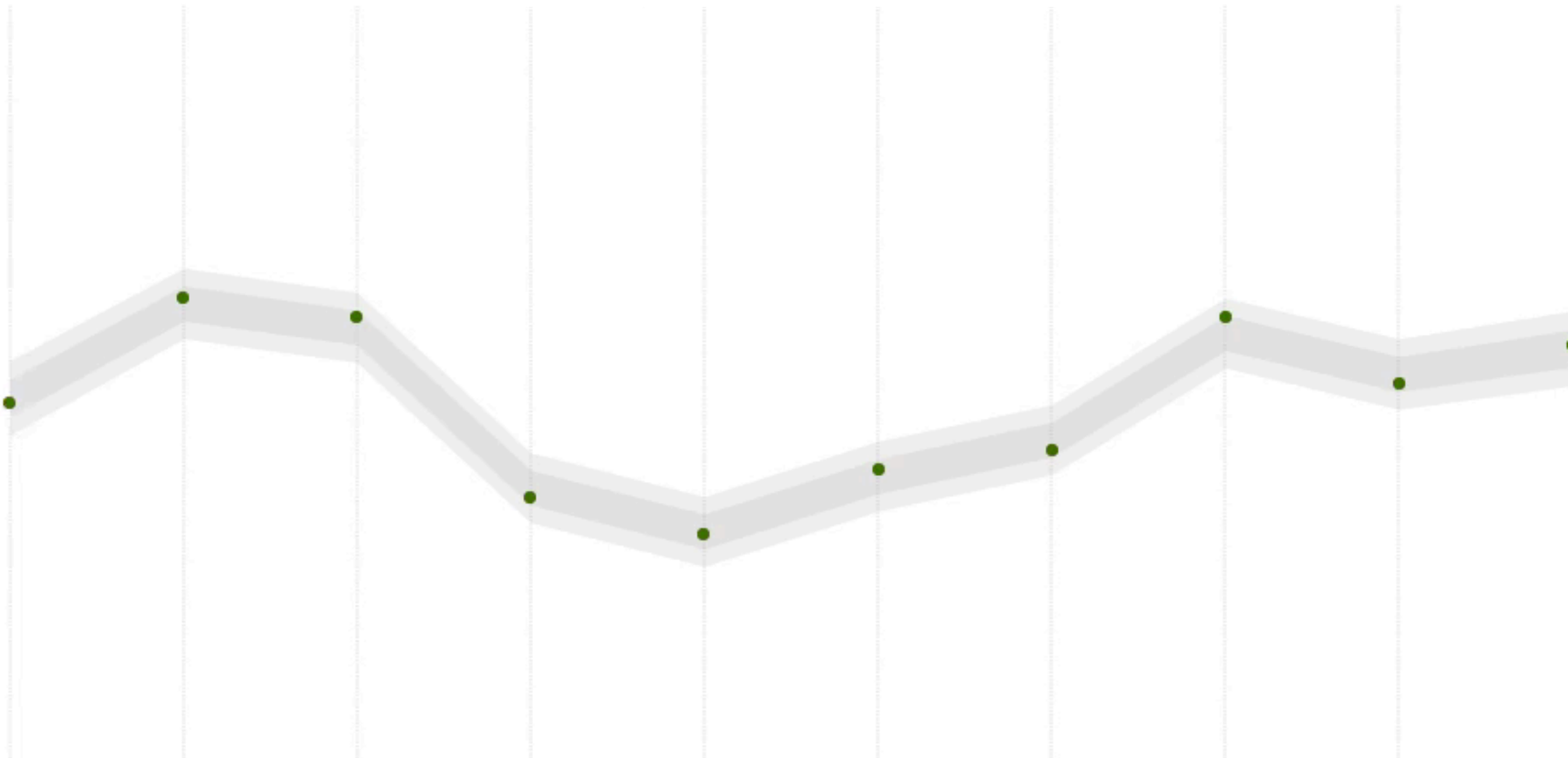
Iteratively,

- **simulate**
- **weight**
- **resample**

Sequential Monte Carlo



Sequential Monte Carlo



SMC in action: slowed down for clarity

Probabilistic programs as state spaces?

Trace

- Sequence of N **observe**'s

$$\{(g_i, \phi_i, y_i)\}_{i=1}^{N}$$

- Sequence of M **sample**'s

$$\{(f_j, \theta_j)\}_{j=1}^{M}$$

- Sequence of M sampled values

$$\{x_j\}_{j=1}^{M}$$

- Conditioned on these sampled values the entire computation is *deterministic*

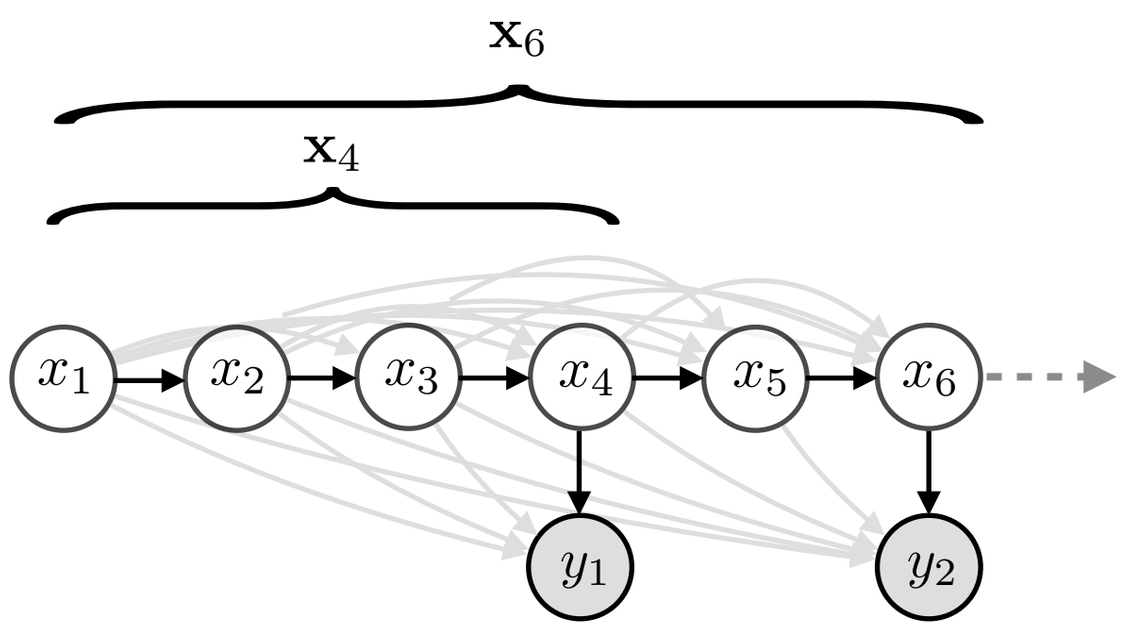
Trace Probability

- Defined as (up to a normalization constant)

$$\gamma(\mathbf{x}) \triangleq p(\mathbf{x}, \mathbf{y}) = \prod_{i=1}^{N} g_i(y_i|\phi_i) \prod_{j=1}^{M} f_j(x_j|\theta_j)$$

- Hides true dependency structure

$$\gamma(\mathbf{x}) = p(\mathbf{x}, \mathbf{y}) = \prod_{i=1}^{N} \tilde{g}_i(\mathbf{x}_{n_i}) \left(y_i \middle| \tilde{\phi}_i(\mathbf{x}_{n_i}) \right) \prod_{j=1}^{M} \tilde{f}_j(\mathbf{x}_{j-1}) \left(x_j \middle| \tilde{\theta}_j(\mathbf{x}_{j-1}) \right)$$



$\mathbf{x}_j = x_1 \times \cdots \times x_j$

etc

Likelihood Weighting

- Run K independent copies of program simulating from the prior

$$q(\mathbf{x}^k) = \prod_{j=1}^{M^k} f_j(x_j^k | \theta_j^k)$$

- Accumulate *unnormalized* weights (likelihoods)

$$w(\mathbf{x}^k) = \frac{\gamma(\mathbf{x}^k)}{q(\mathbf{x}^k)} = \prod_{i=1}^{N^k} g_i^k(y_i^k | \phi_i^k)$$

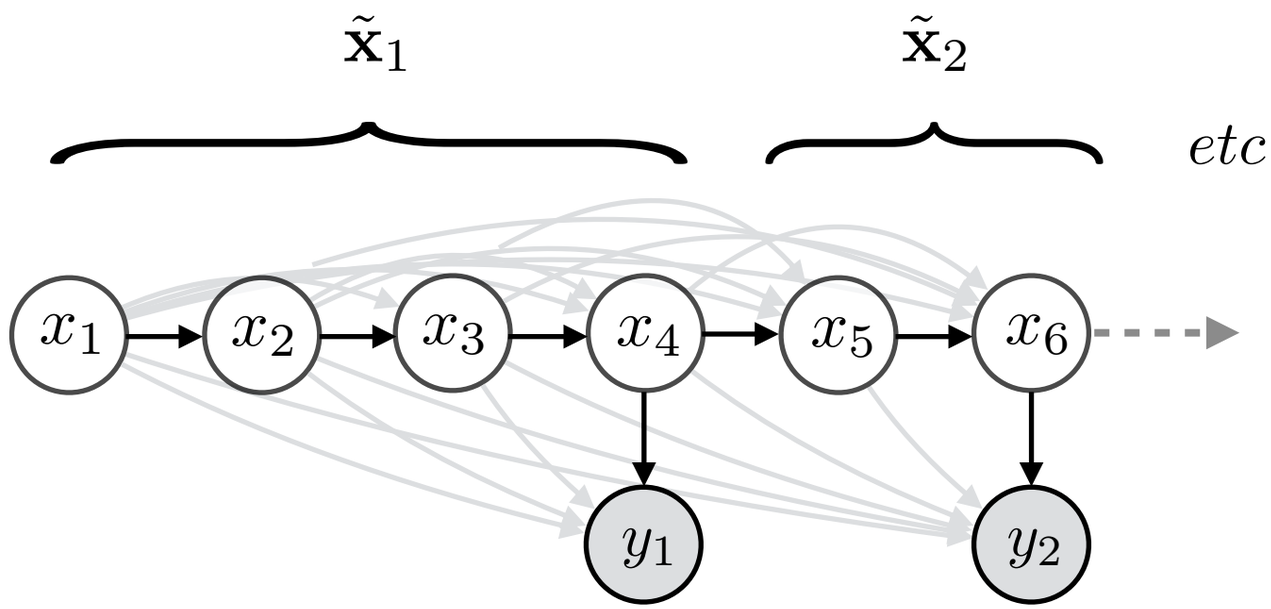
- Use in approximate (Monte Carlo) integration

$$W^k = \frac{w(\mathbf{x}^k)}{\sum_{\ell=1}^{K} w(\mathbf{x}^\ell)} \qquad \hat{\mathbb{E}}_\pi[R(\mathbf{x})] = \sum_{k=1}^{K} W^k R(\mathbf{x}^k)$$

Probabilistic programs as state spaces

- Notation

$$\tilde{\mathbf{x}}_{1:n} = \tilde{\mathbf{x}}_1 \times \cdots \times \tilde{\mathbf{x}}_n$$



- Incrementalized joint

$$\gamma_n(\tilde{\mathbf{x}}_{1:n}) = \prod_{n=1}^{N} g(y_n|\tilde{\mathbf{x}}_{1:n})p(\tilde{\mathbf{x}}_n|\tilde{\mathbf{x}}_{1:n-1})$$

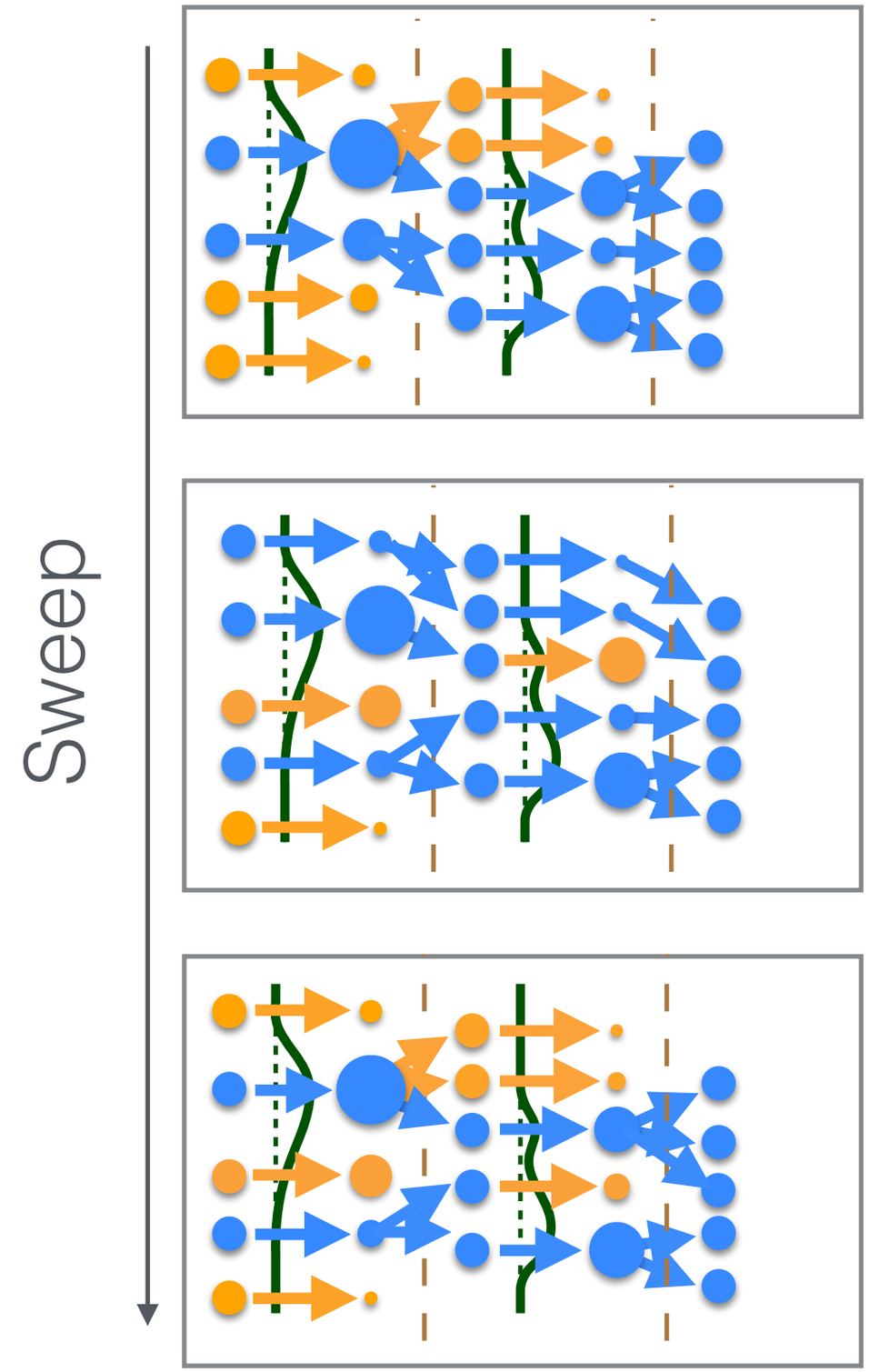
- Incrementalized target

$$\pi_n(\tilde{\mathbf{x}}_{1:n}) = \frac{1}{Z_n}\gamma_n(\tilde{\mathbf{x}}_{1:n})$$

Particle Markov chain Monte Carlo

Particle Markov Chain Monte Carlo

- Iterable SMC
  - PIMH : “particle independent Metropolis-Hastings”
  - PGIBBS : “iterated conditional SMC”
  - PGAS : “particle Gibbs ancestral sampling"



PIMH Math

- Each sweep of SMC can compute

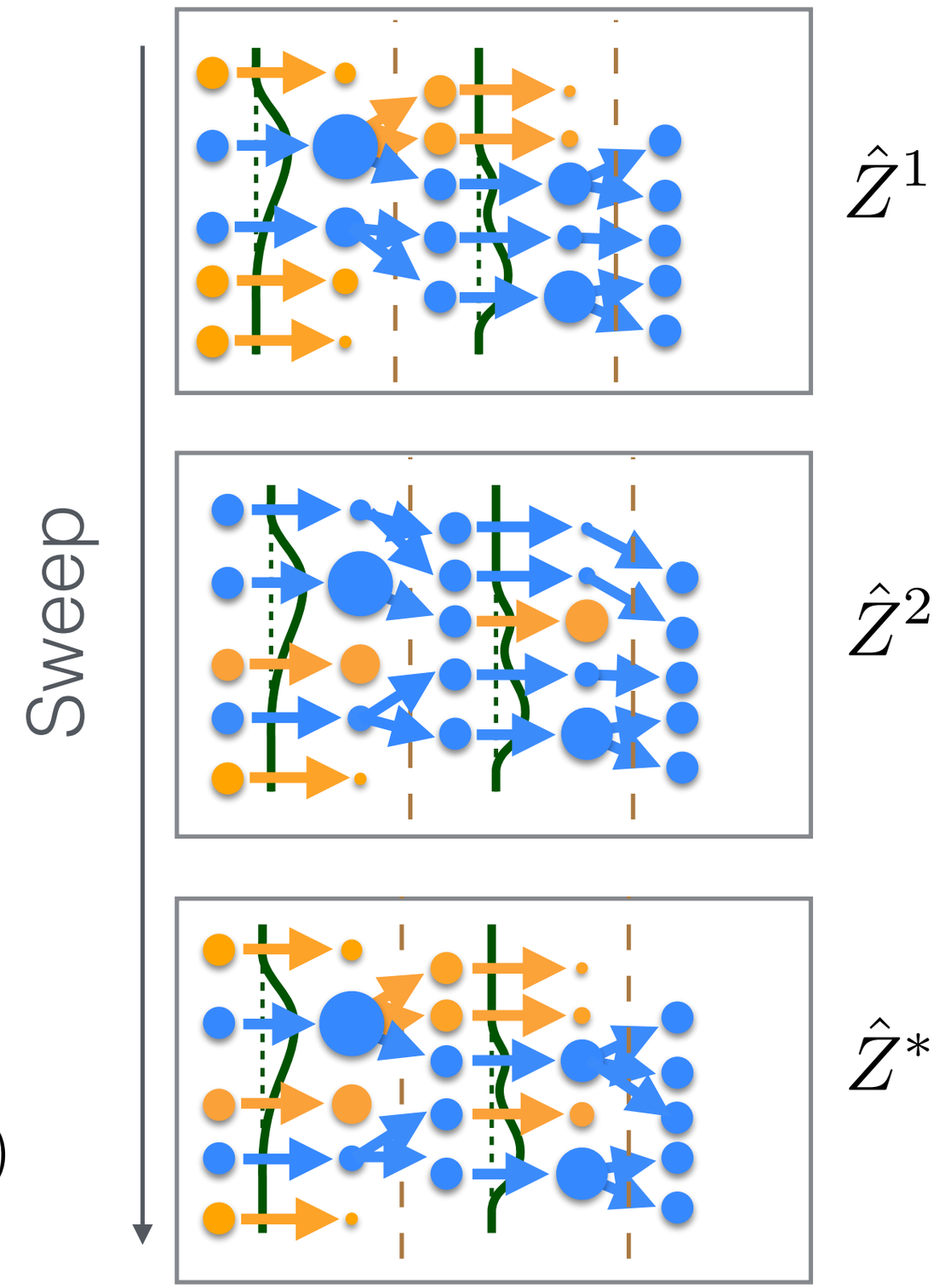
$$\hat{Z} = \prod_{n=1}^{N} \hat{Z}_n = \prod_{n=1}^{N} \frac{1}{K} \sum_{k=1}^{K} w(\tilde{\mathbf{x}}_{1:n}^k)$$

- PIMH is MH that accepts entire new particle sets w.p.

$$\alpha_{PIMH}^s = \min\left(1, \frac{\hat{Z}^\star}{\hat{Z}^{s-1}}\right)$$

- And all particles can be used

$$\hat{\mathbb{E}}_{PIMH}[R(\mathbf{x})] = \frac{1}{S} \sum_{s=1}^{S} \sum_{k=1}^{K} W^{s,k} R(\mathbf{x}^{s,k})$$



Asynchronous anytime sequential Monte Carlo

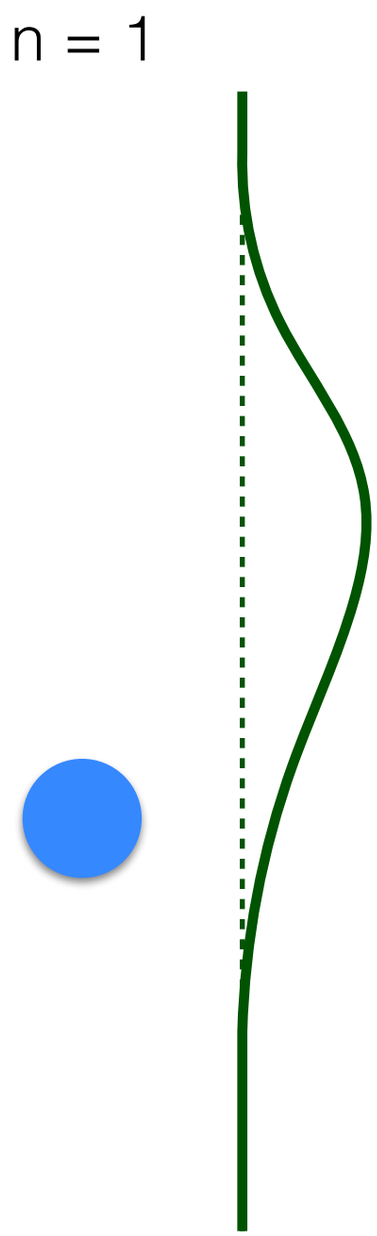
Parallelization in SMC

- Forward simulation trivially parallelizes
  - this is the sort of parallelization achieved through (e.g.) **`parfor`** in MATLAB, or **`pmap`** in functional programming languages
- The resampling step (normalizing weights, sampling child counts) is a global synchronous operation
  - cannot resample until all particles finish simulation

Particle Cascade

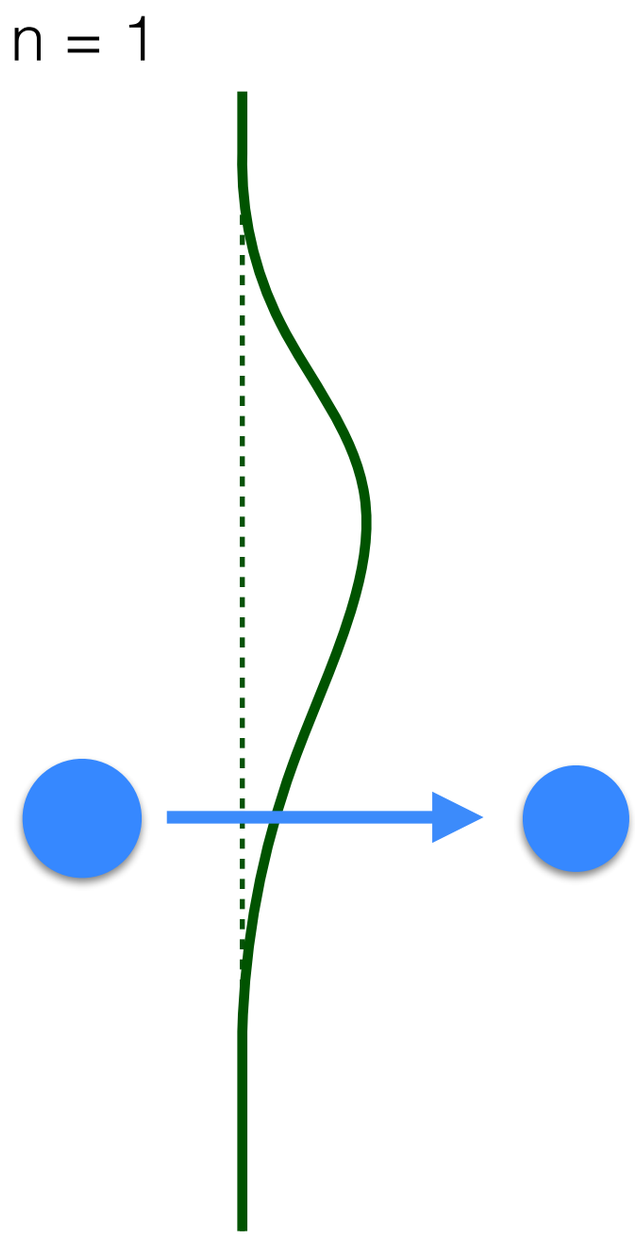
- Replace **resampling** step with **branching** step
- Launch particles **asynchronously**
- As each particle arrives at an observation, choose a number of offspring based only on the particles which have arrived so far
  - … don't need to wait for all particles to arrive
  - … only need to track average weights at each observation, which we compute online

Particle Cascade



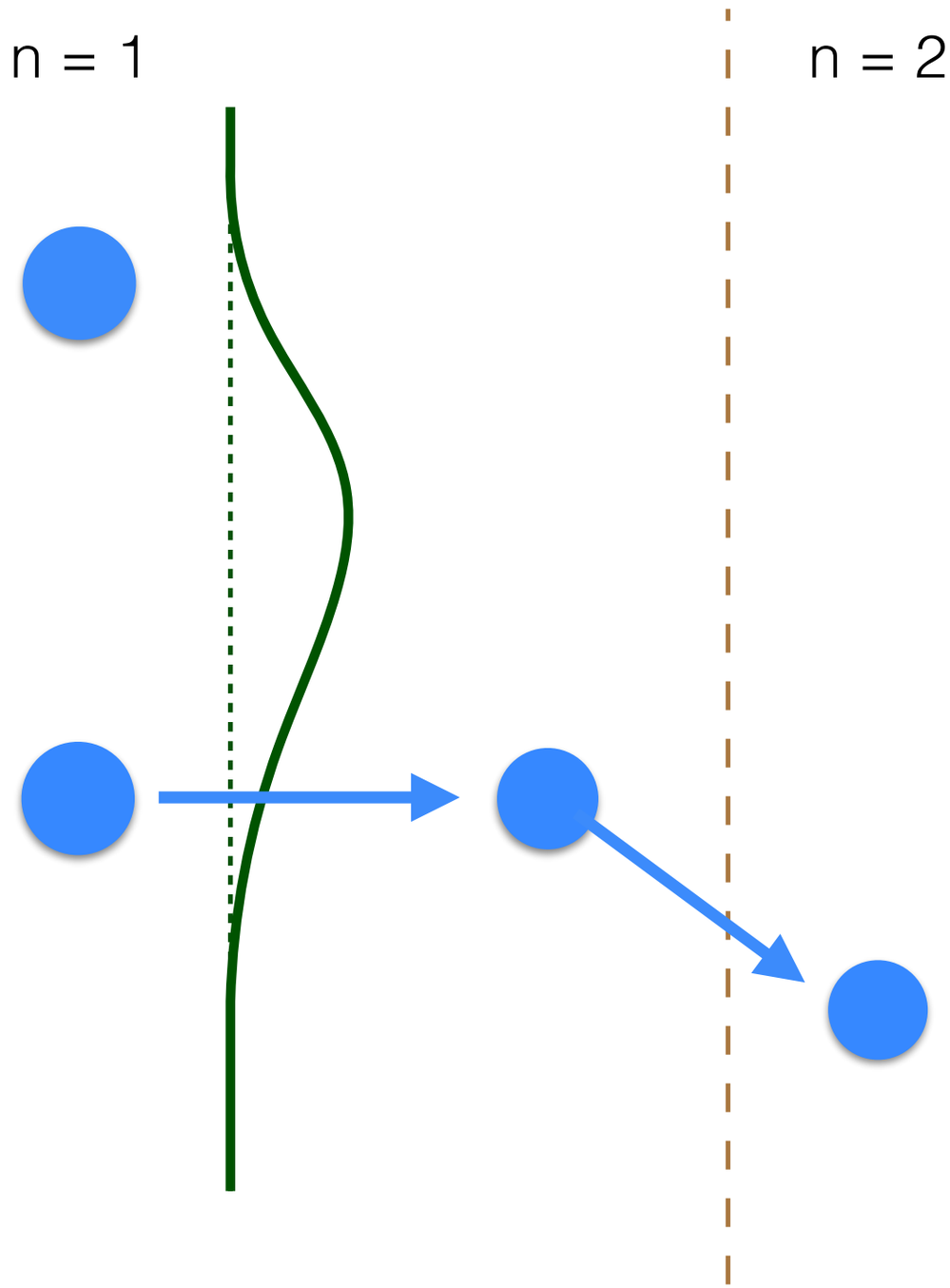
- Start by simulating particles, one at a time, from $f(x_n|x_{1:n-1})$
- Weight by likelihood $g(y_n|x_{1:n})$

Particle Cascade



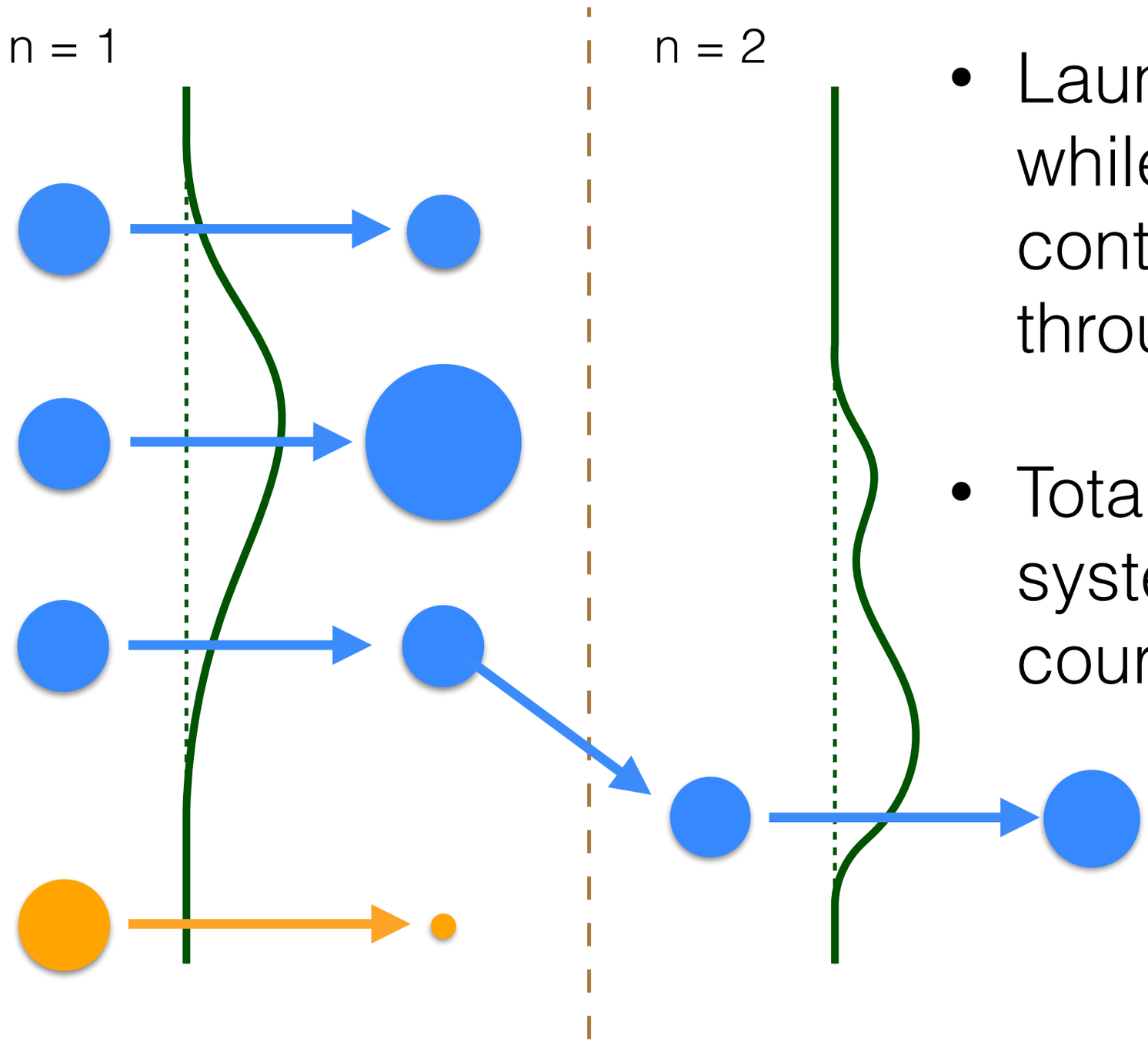
- Start by simulating particles, one at a time, from $f(x_n|x_{1:n-1})$
- Weight by likelihood $g(y_n|x_{1:n})$

Particle Cascade



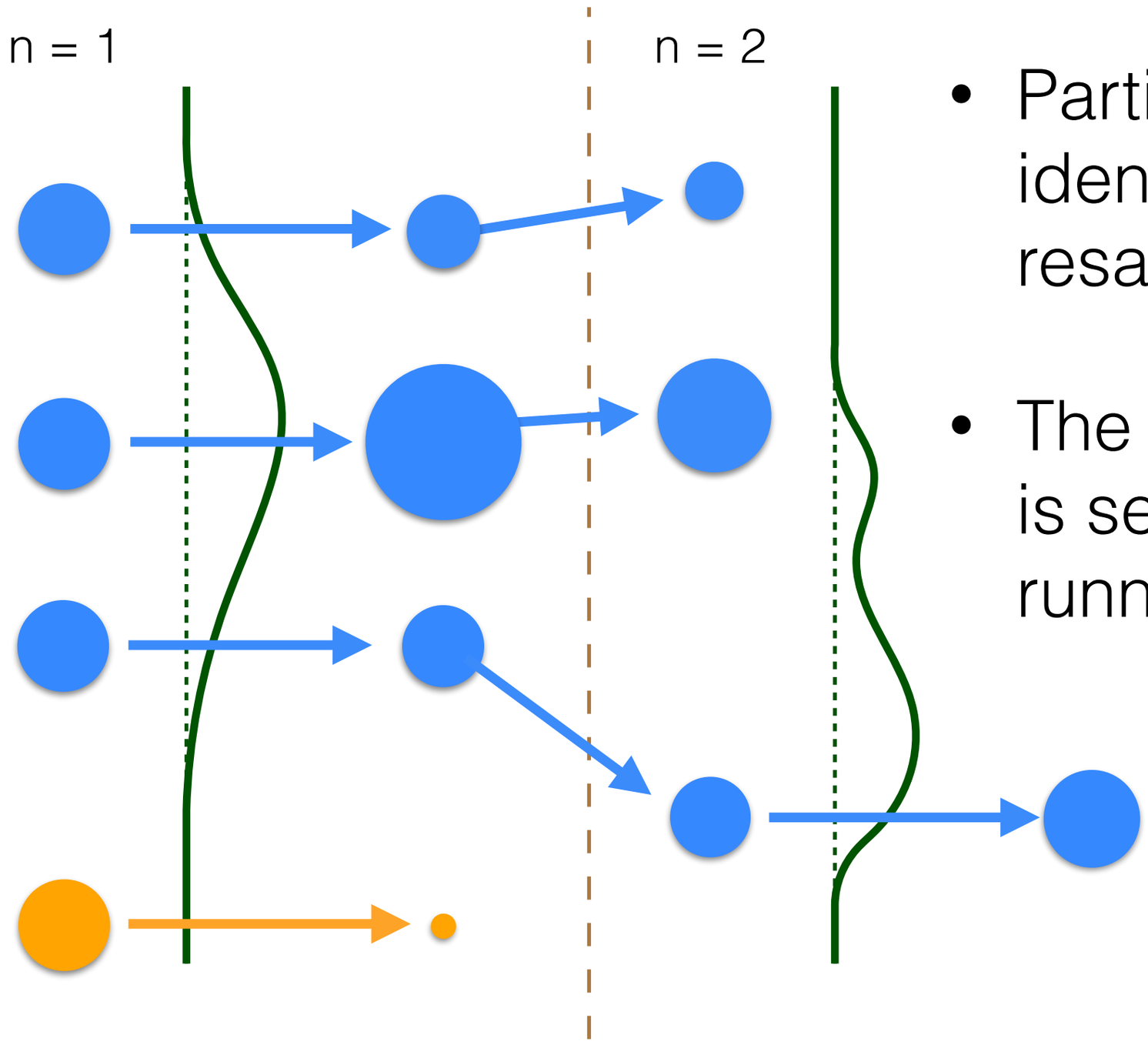
- Keep track of the running average weight $\overline{W}_n^k$ at each *n*, based only on first *k* particles to arrive
- Choose number of offspring **immediately**, no need to wait for other particles
- $\mathbb{E}[M_k^n | W_n^{1:k}] = \dfrac{W_n^k}{\overline{W}_n^k}$

Particle Cascade



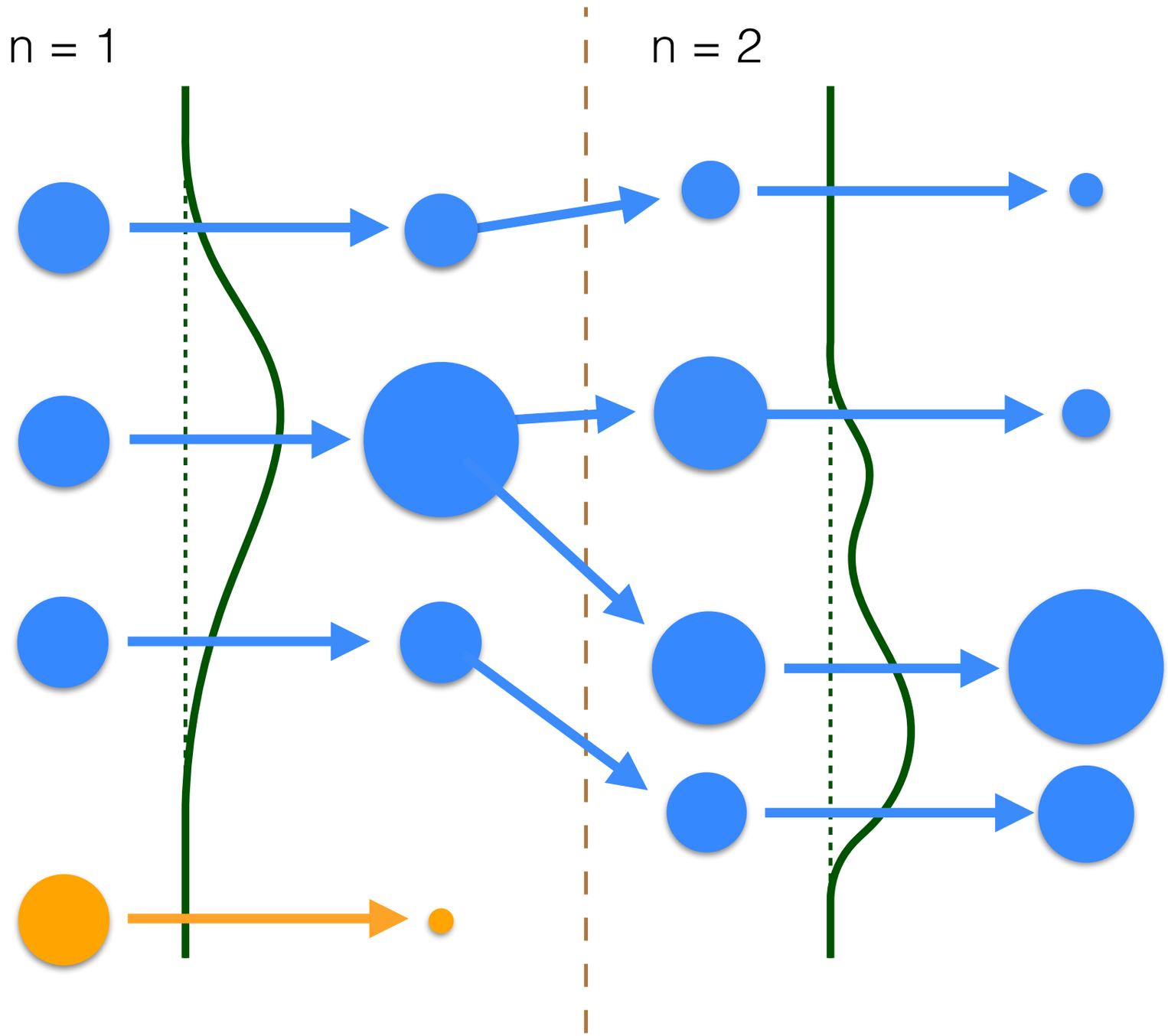
- Launch new particles while other particles continue moving forward through the system
- Total size of particle system may vary over course of execution

Particle Cascade



- Particles do not have identical weight after resampling
- The “outgoing” weight is set to the current running average $\overline{W}_n^k$

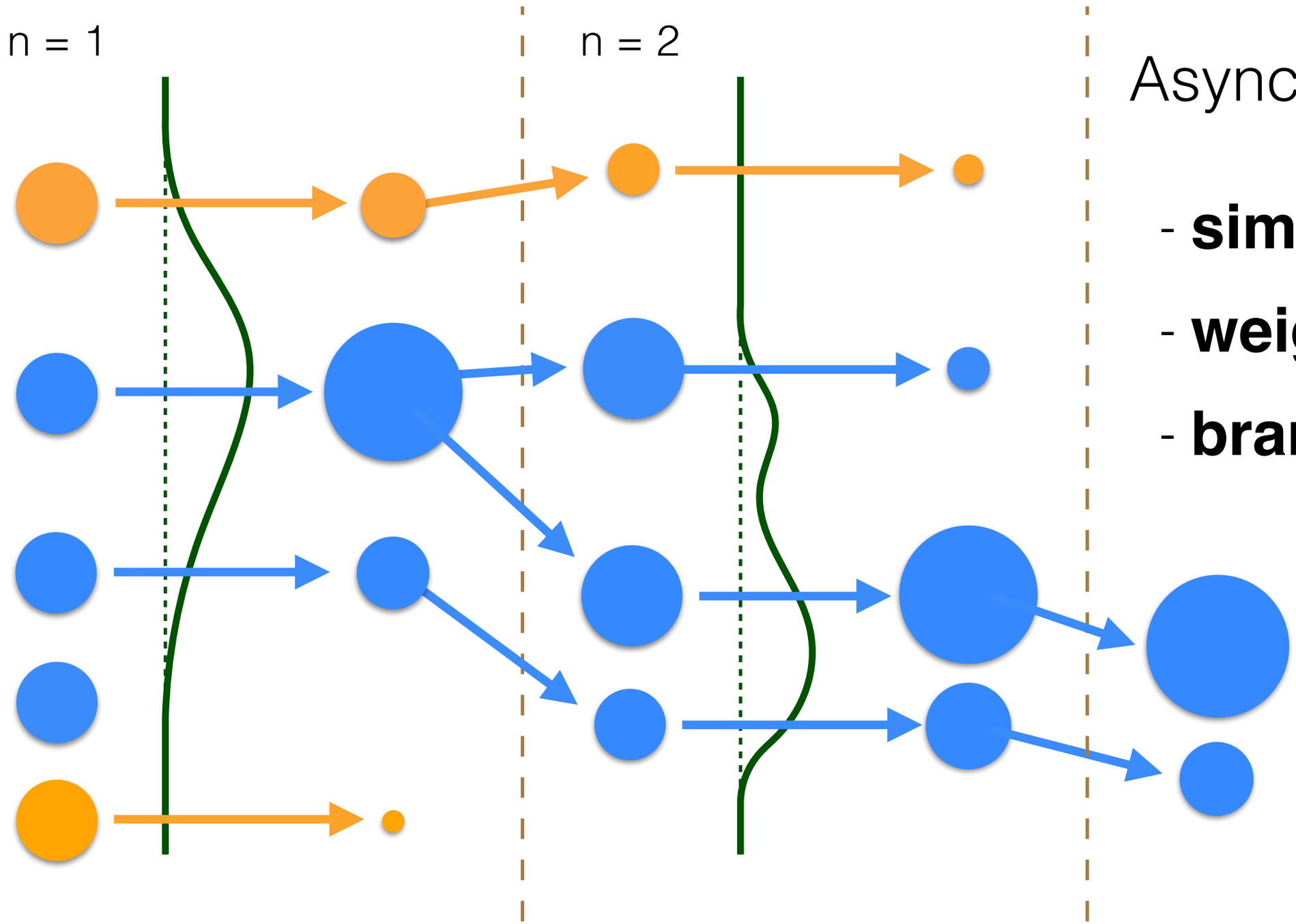
Particle Cascade



Asynchronously

- **simulate**
- **weight**
- **branch**

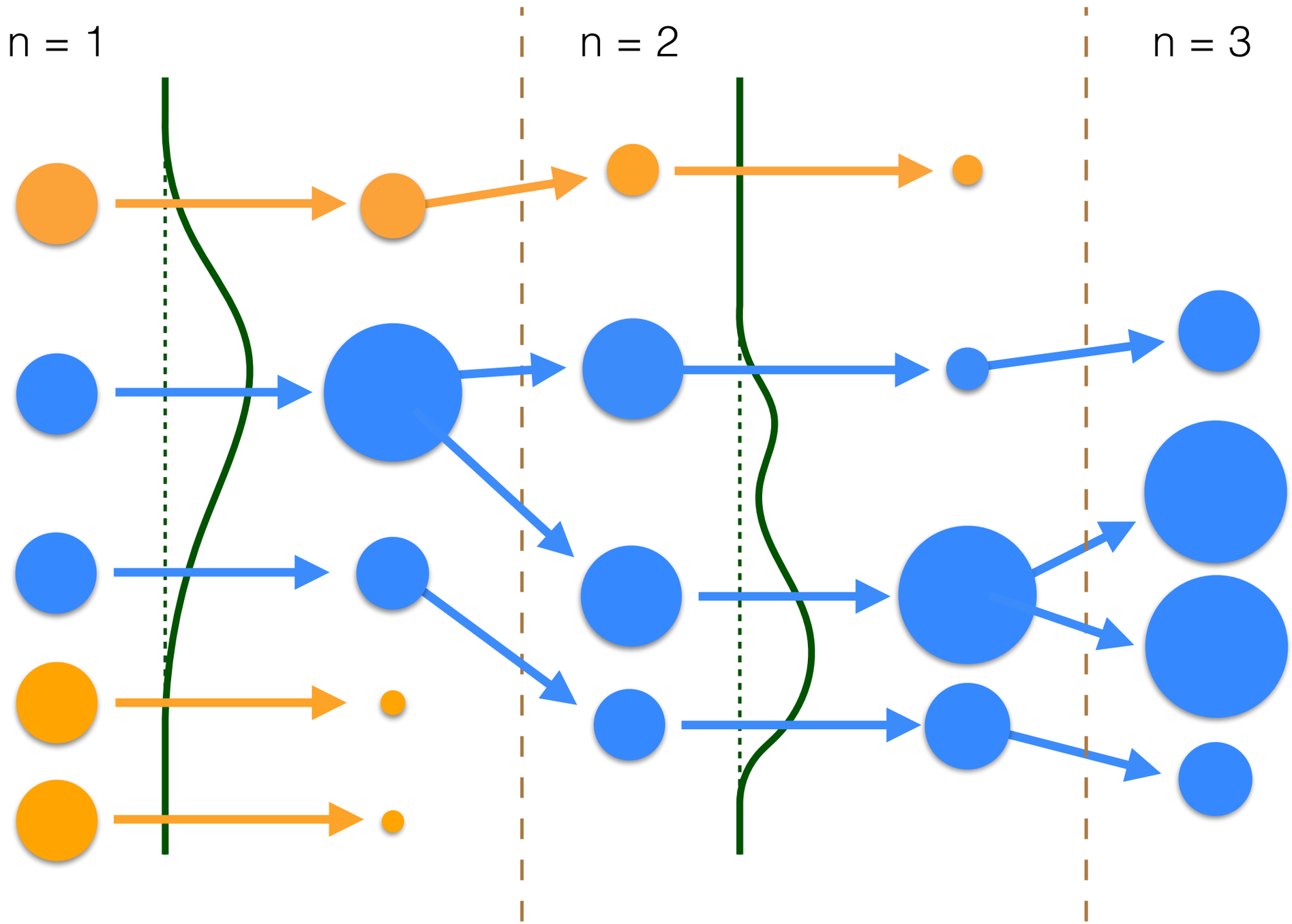
Particle Cascade



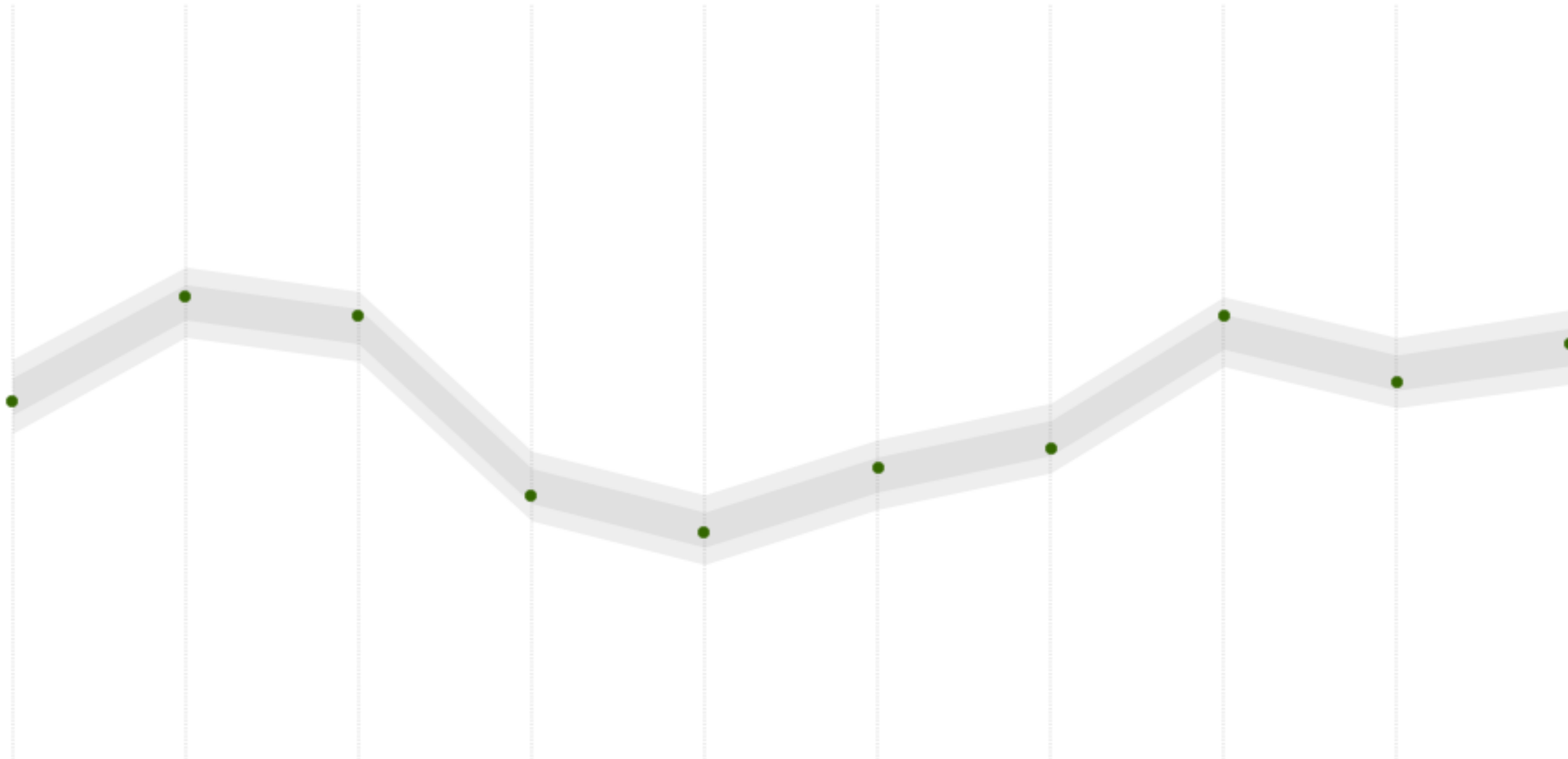
Asynchronously

- **simulate**
- **weight**
- **branch**

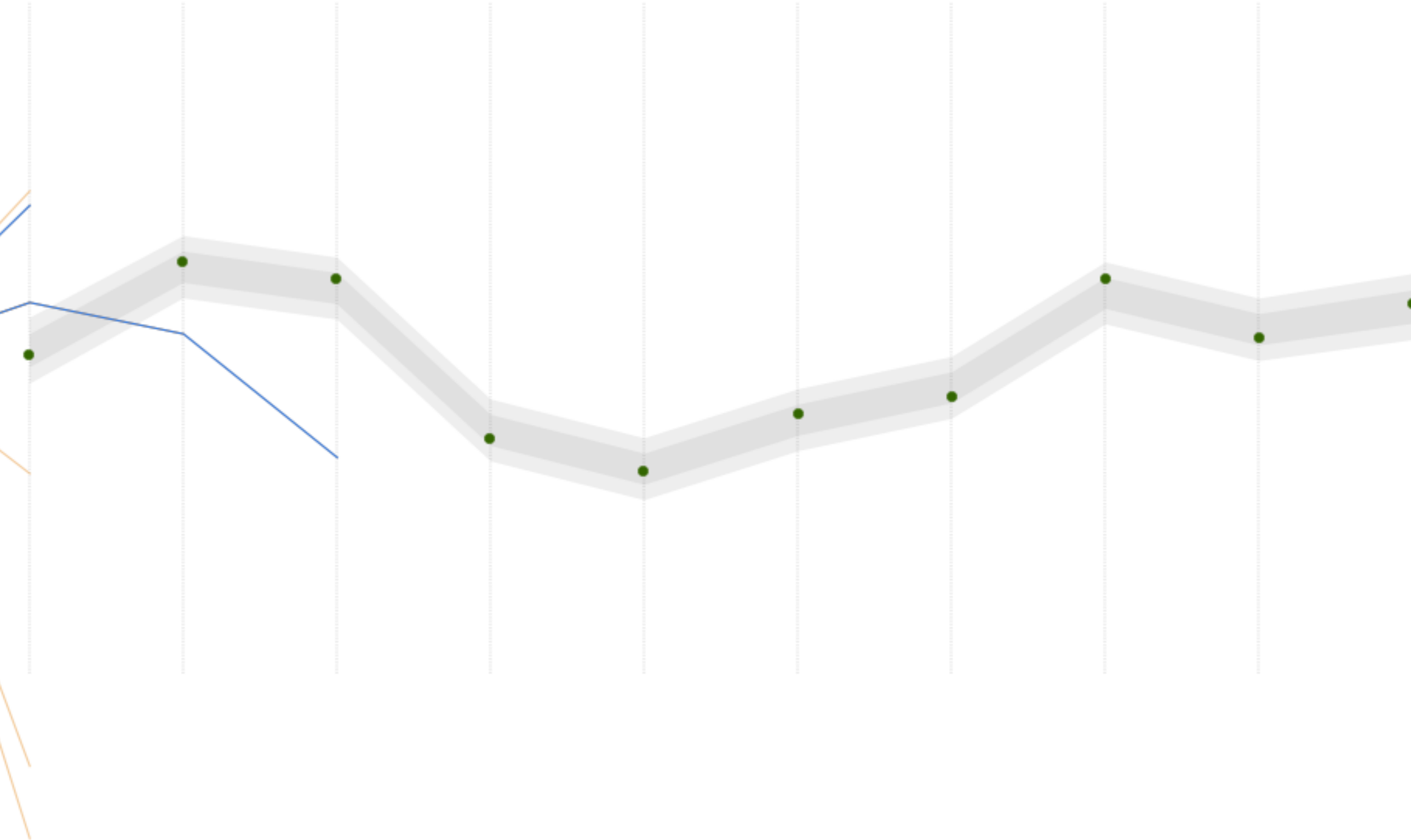
Particle Cascade



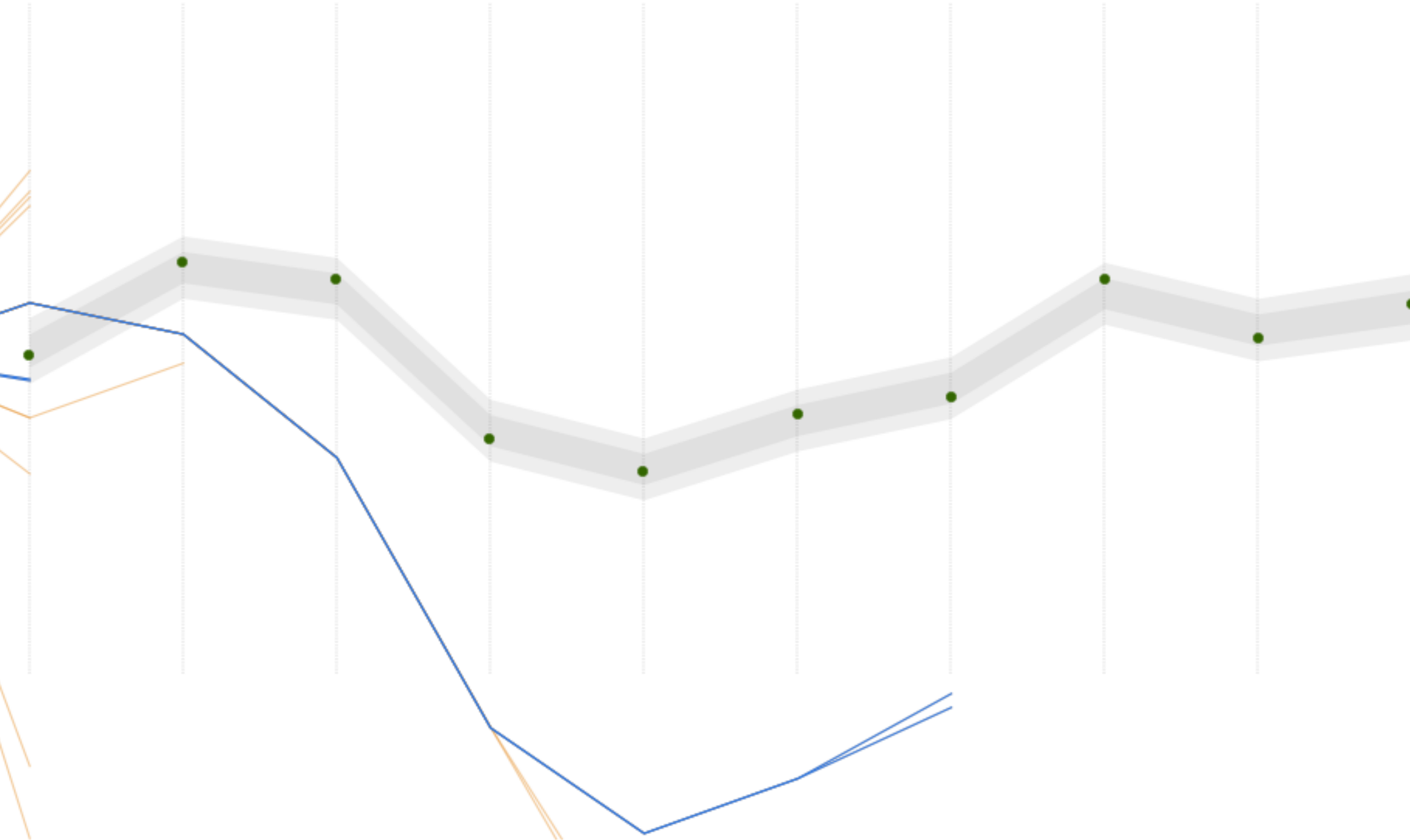
Particle Cascade



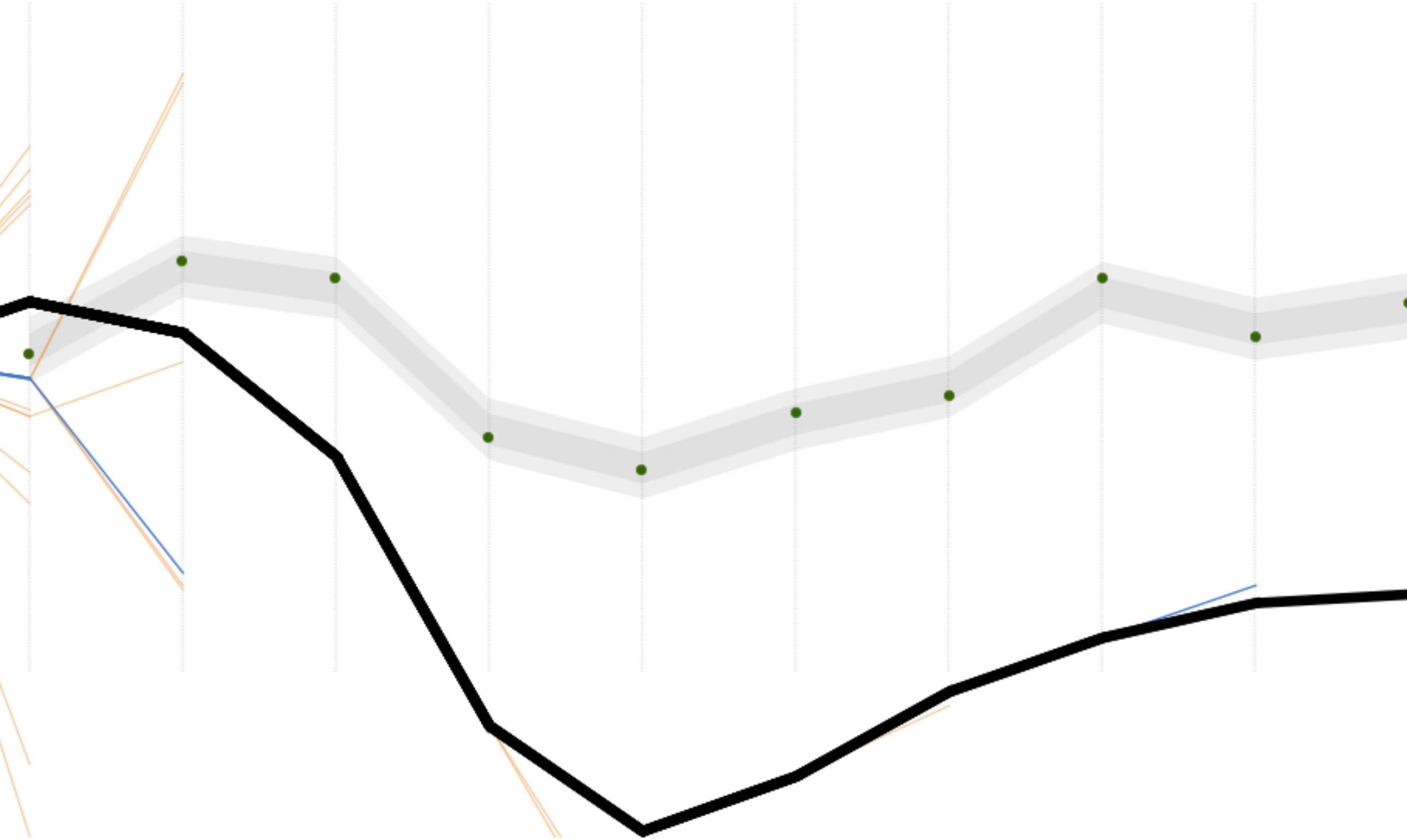
Particle Cascade



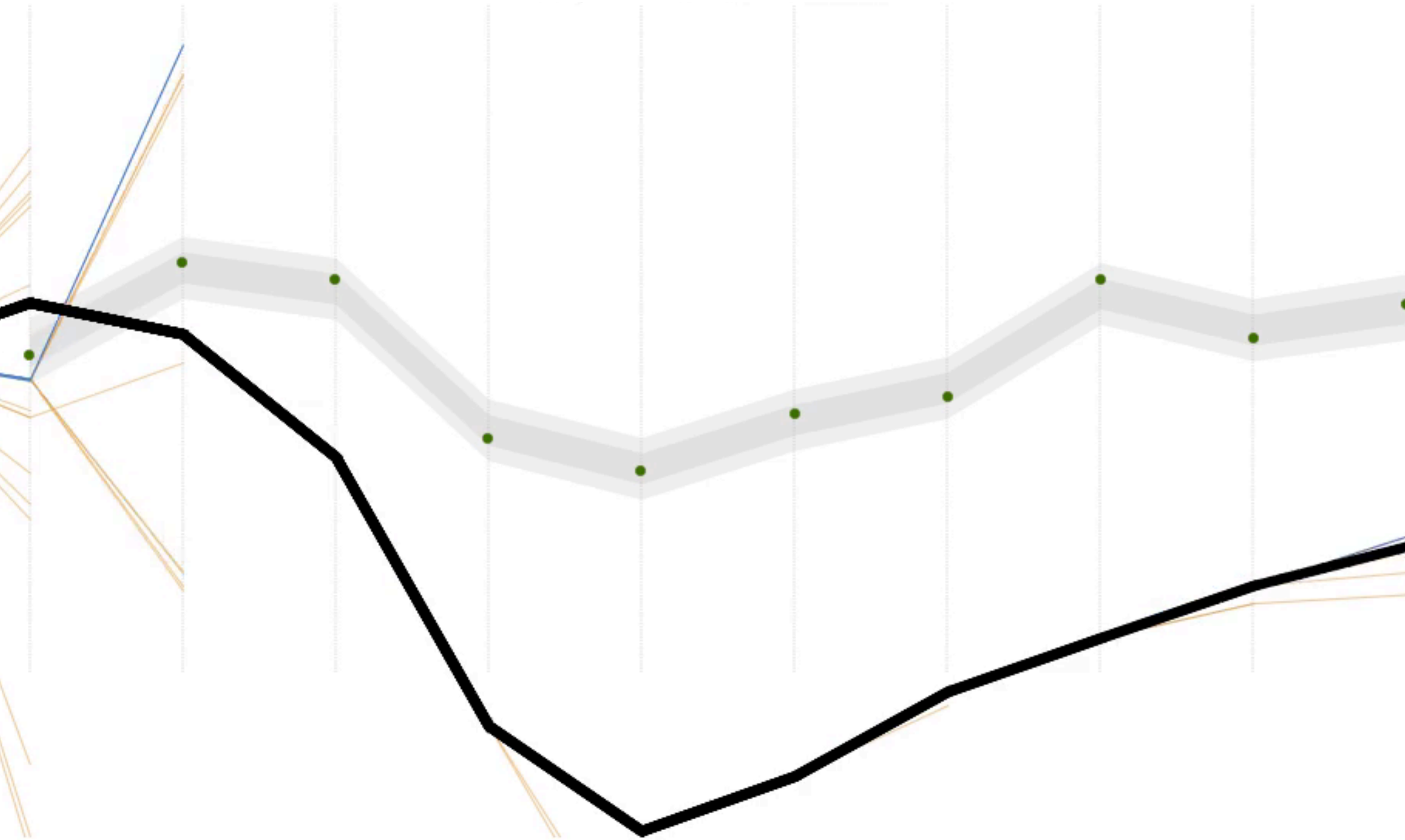
Particle Cascade



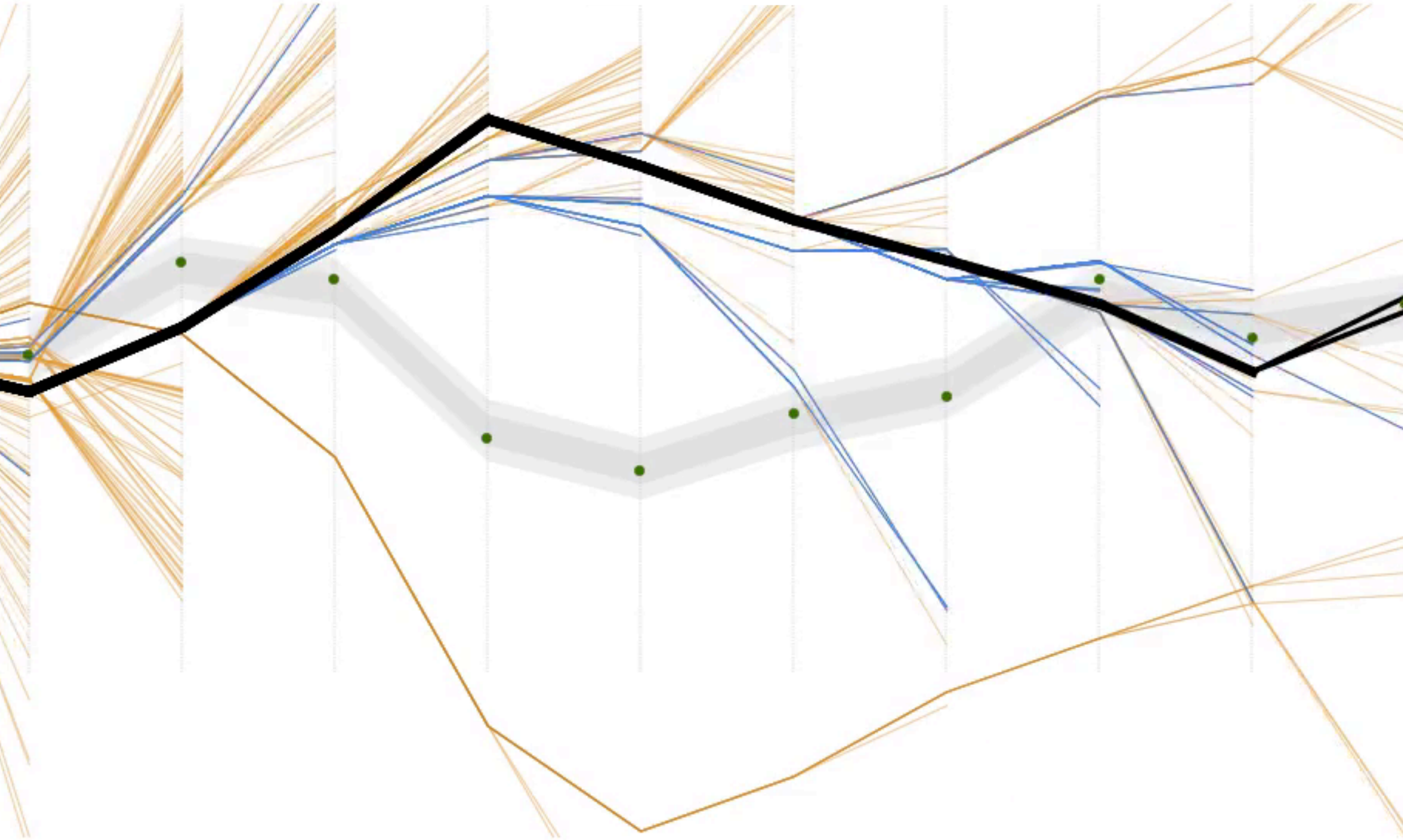
Particle Cascade



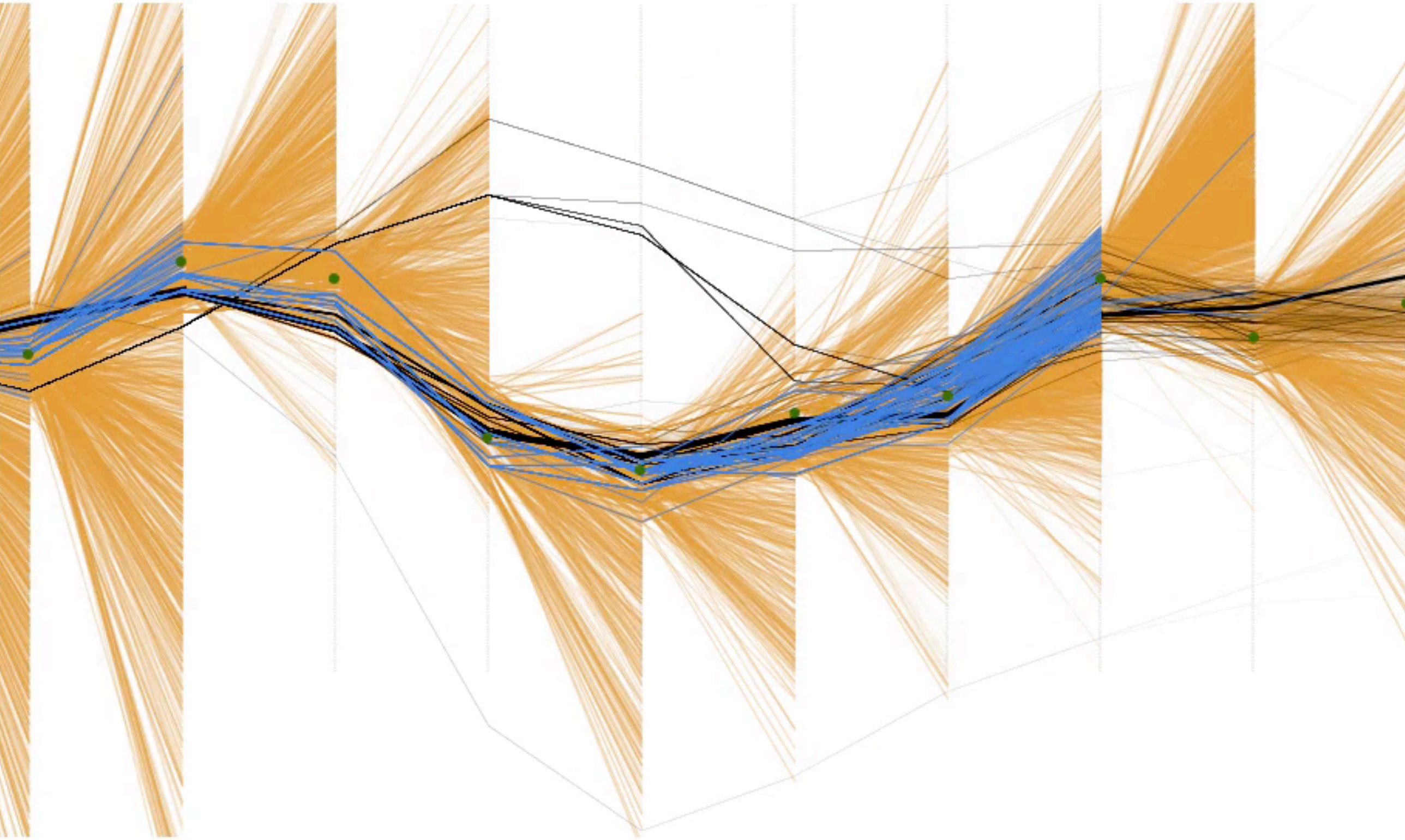
Particle Cascade



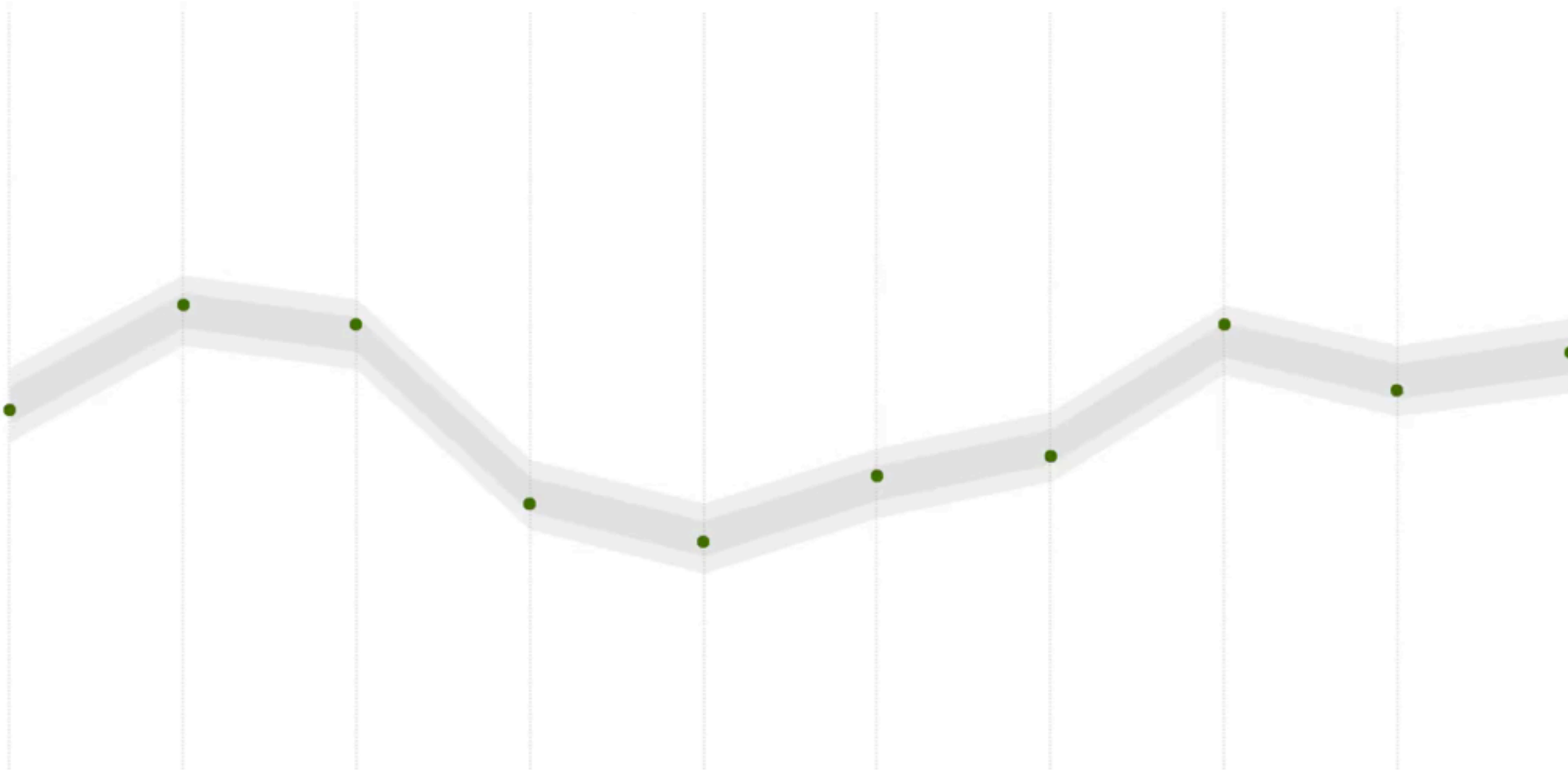
Particle Cascade



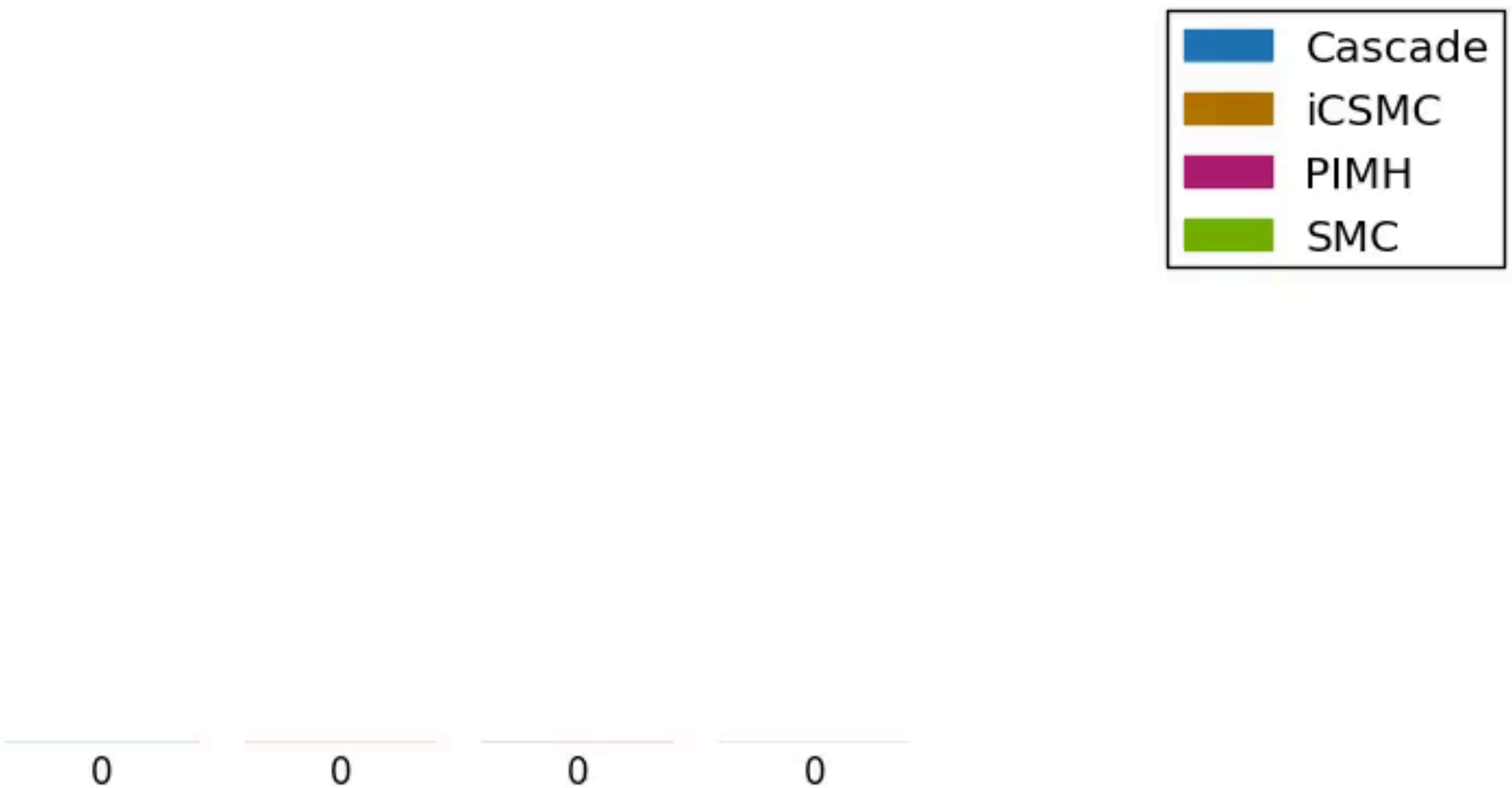
Particle Cascade



Particle Cascade

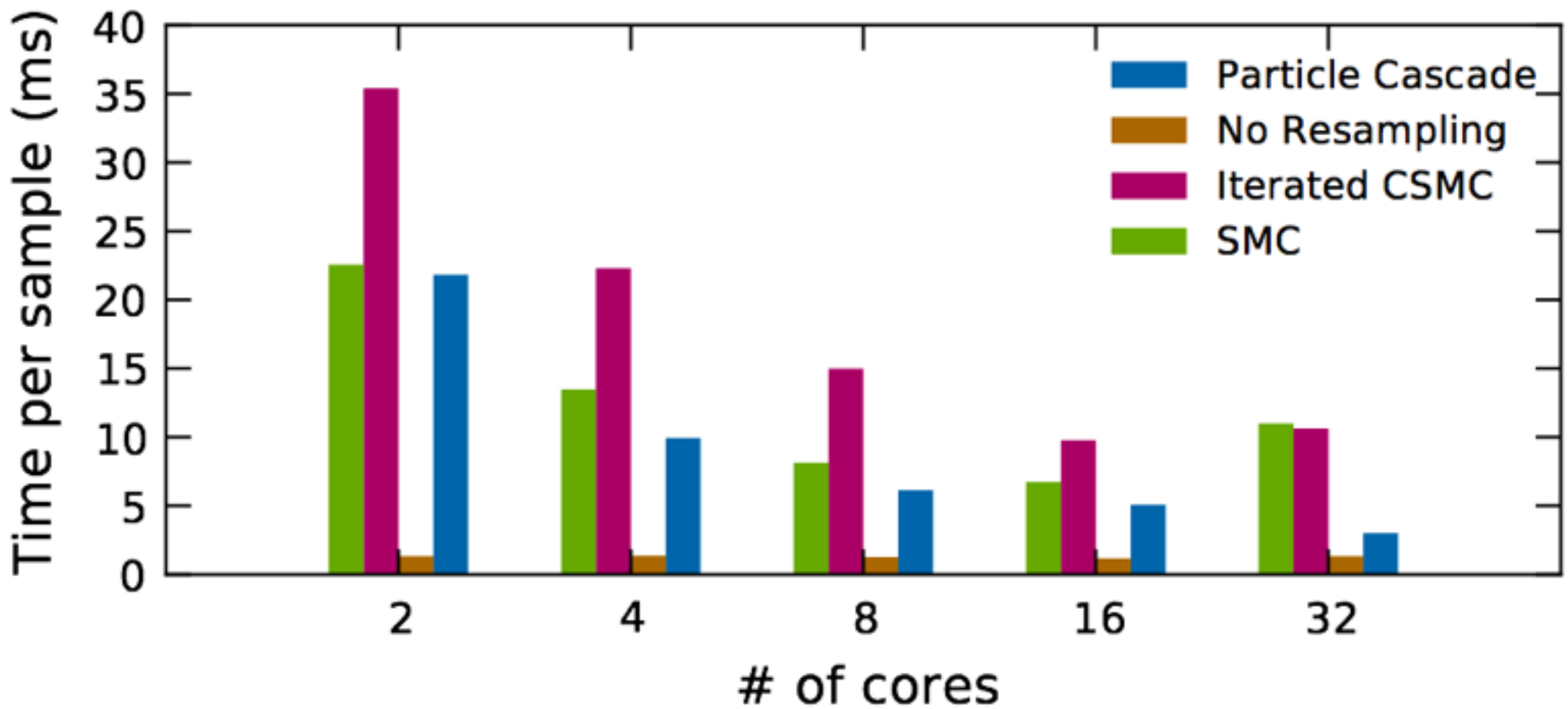


Scalability: Particle Count



- Comparison across particle-based inference approaches: raw speed of drawing samples
- Each particle runs as a separate CPU process

Scalability: Multiple Cores



- More cores == faster inference
- Scales to multiple cores more efficiently than other particle-based methods

Particle cascade summary

- Particle cascade is an asynchronous anytime drop-in replacement for SMC, with the added benefits of
  - … an **anytime** property similar to MCMC methods; keep running inference indefinitely, stop when satisfied with the current estimate
  - … **no barrier synchronizations**, yielding increased particle throughput and parallel scalability as compared to traditional SMC

Inference networks for sequential Monte Carlo

Executive Summary

We want to make model-based Bayesian inference efficient.

- In general: what artifacts can we learn offline to compile away the runtime costs of inference?
- Outside of specific (probably wrong) models, inference is fundamentally not a feed-forward computation!
- Sequential Monte Carlo for graphical models: approximate optimal importance sampling proposals

Inference in Graphical Models

Goal: posterior inference in generative models with latent variables **x** and observed variables **y**:

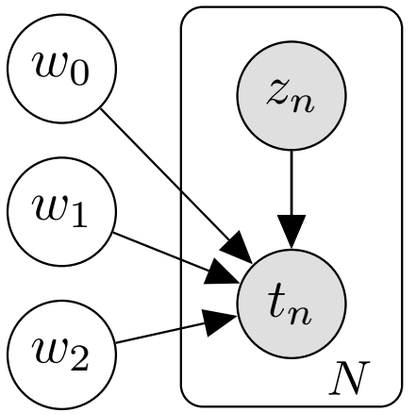
$$p(\mathbf{x}, \mathbf{y}) \triangleq \prod_{i=1}^{N} p\left(x_i | \mathrm{PA}(x_i)\right) \prod_{j=1}^{M} p\left(y_j | \mathrm{PA}(y_j)\right)$$

Importance sampling and SMC approximate the posterior $\pi(\mathbf{x}) \equiv p(\mathbf{x}|\mathbf{y})$ as weighted samples:

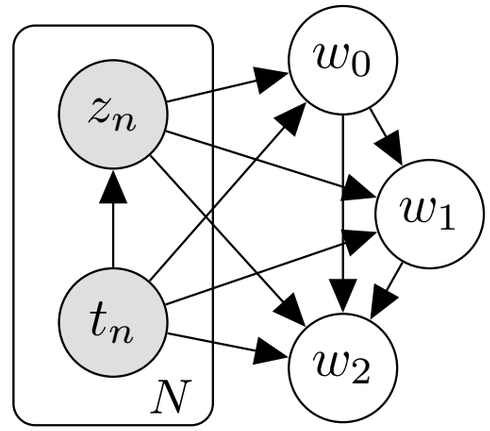
$$\hat{p}(\mathbf{x}|\mathbf{y}) = \sum_{k=1}^{K} W_k \delta_{\mathbf{x}_k}(\mathbf{x}) \qquad w(\mathbf{x}) = \frac{p(\mathbf{x}, \mathbf{y})}{q(\mathbf{x}|\lambda)} \qquad W_k = \frac{w(\mathbf{x}_k)}{\sum_{j=1}^{K} w(\mathbf{x}_j)}$$

Performance depends on quality of proposal $q(\mathbf{x}|\lambda)$!

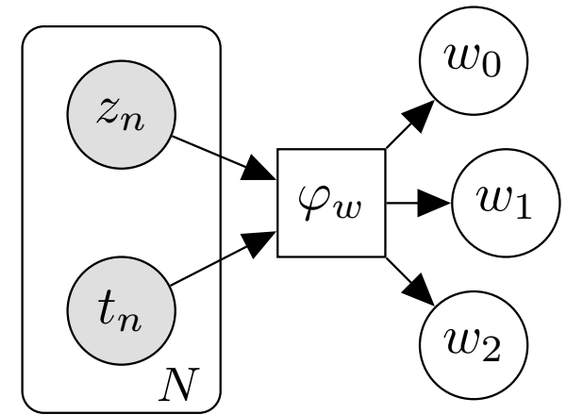
Inference Networks for Graphical Models



A probabilistic model **generates data**



An inverse model **generates latents**



Can we **learn how to sample** from the inverse model?

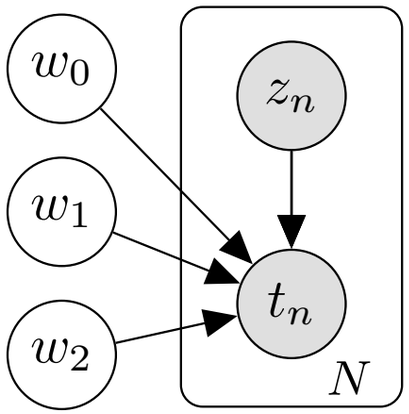
Learning an importance sampling proposal for a single dataset

Target density $\pi(\mathbf{x}) = p(\mathbf{x}|\mathbf{y})$, approximating family $q(\mathbf{x}|\lambda)$

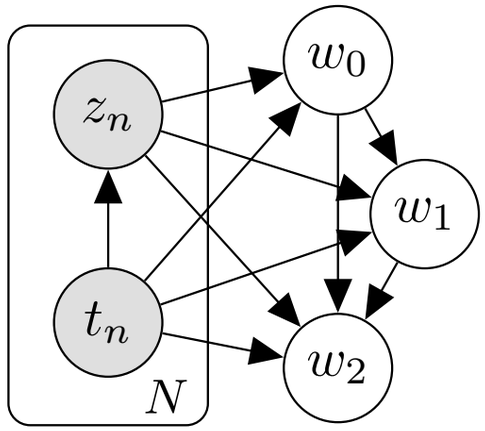
Single dataset $\mathbf{y}$: $\underset{\lambda}{\operatorname{argmin}}\, D_{KL}(\pi||q_\lambda)$

fit λ to learn an importance sampling proposal

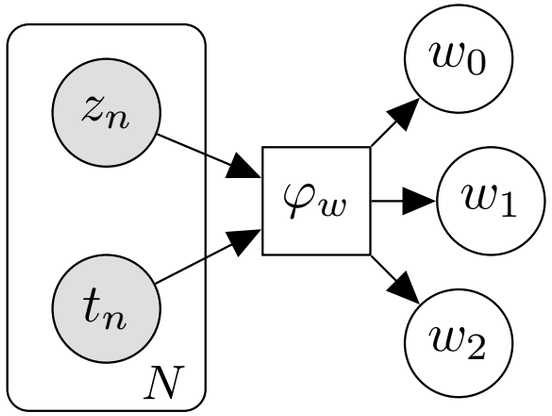
Inference Networks for Graphical Models



A probabilistic model **generates data**



An inverse model **generates latents**



Can we **learn how to sample** from the inverse model?

Idea: amortize inference by learning a map from data to target

Target density $\pi(\mathbf{x}) = p(\mathbf{x}|\mathbf{y})$, approximating family $q(\mathbf{x}|\lambda)$

Averaging over all possible datasets:

$$\lambda = \varphi(\eta, \mathbf{y})$$

$$\underset{\eta}{\operatorname{argmin}}\, \mathbb{E}_{p(\mathbf{y})}\left[D_{KL}(\pi || q_{\varphi(\eta,\mathbf{y})})\right]$$

learn a mapping from arbitrary datasets to λ

Compiling away runtime costs of inference

Learn to invert the generative model, before seeing data

Averaging over
all possible datasets:

$$\lambda = \varphi(\eta, \mathbf{y})$$

$$\underset{\eta}{\operatorname{argmin}}\, \mathbb{E}_{p(\mathbf{y})} \left[D_{KL}(\pi || q_{\varphi(\eta, \mathbf{y})}) \right]$$

expectation over any data
we might observe

Compiling away runtime costs of inference

Learn to invert the generative model, before seeing data

Averaging over
all possible datasets:

$$\lambda = \varphi(\eta, \mathbf{y})$$

$$\underset{\eta}{\operatorname{argmin}}\, \mathbb{E}_{p(\mathbf{y})} \left[D_{KL}(\pi || q_{\varphi(\eta,\mathbf{y})}) \right]$$

New objective function,
upper-level parameters:

$$\begin{aligned}
\mathcal{J}(\eta) &= \int D_{KL}(\pi || q_\lambda) p(\mathbf{y}) \mathrm{d}\mathbf{y} \\
&= \int p(\mathbf{y}) \int p(\mathbf{x}|\mathbf{y}) \log \left[\frac{p(\mathbf{x}|\mathbf{y})}{q(\mathbf{x}|\varphi(\eta,\mathbf{y}))} \right] \mathrm{d}\mathbf{x}\mathrm{d}\mathbf{y} \\
&= \mathbb{E}_{p(\mathbf{x},\mathbf{y})} \left[-\log q(\mathbf{x}|\varphi(\eta,\mathbf{y})) \right] + const.
\end{aligned}$$

expectation over (tractable)
joint distribution

Compiling away runtime costs of inference

Learn to invert the generative model, before seeing data

Averaging over
all possible datasets: $\lambda = \varphi(\eta, \mathbf{y})$

$$\underset{\eta}{\operatorname{argmin}}\, \mathbb{E}_{p(\mathbf{y})}\left[D_{KL}(\pi || q_{\varphi(\eta,\mathbf{y})})\right]$$

New objective function,
upper-level parameters:

$$\begin{aligned}\mathcal{J}(\eta) &= \int D_{KL}(\pi || q_\lambda) p(\mathbf{y}) \mathrm{d}\mathbf{y} \\ &= \int p(\mathbf{y}) \int p(\mathbf{x}|\mathbf{y}) \log\left[\frac{p(\mathbf{x}|\mathbf{y})}{q(\mathbf{x}|\varphi(\eta,\mathbf{y}))}\right] \mathrm{d}\mathbf{x}\mathrm{d}\mathbf{y} \\ &= \mathbb{E}_{p(\mathbf{x},\mathbf{y})}\left[-\log q(\mathbf{x}|\varphi(\eta,\mathbf{y}))\right] + const.\end{aligned}$$

approximate with samples
from the joint distribution

Tractable gradient!
Can train entirely offline: $\nabla_\eta \mathcal{J}(\eta) = \mathbb{E}_{p(\mathbf{x},\mathbf{y})}\left[-\nabla_\eta \log q(\mathbf{x}|\varphi(\eta,\mathbf{y}))\right]$

Choice of approximating family

Expected KL divergence: $\mathcal{J}(\eta) = \mathbb{E}_{p(\mathbf{y})}\left[D_{KL}(\pi || q_{\varphi(\eta,\mathbf{y})})\right]$

Gradient: $\nabla_\eta \mathcal{J}(\eta) = \mathbb{E}_{p(\mathbf{x},\mathbf{y})}\left[-\nabla_\eta \log q(\mathbf{x}|\varphi(\eta,\mathbf{y}))\right]$

approximate with samples from model

choose a known parametric family…

… and any differentiable function

Choice of approximating family

Expected KL divergence: $\mathcal{J}(\eta) = \mathbb{E}_{p(\mathbf{y})}\left[D_{KL}(\pi || q_{\varphi(\eta,\mathbf{y})})\right]$

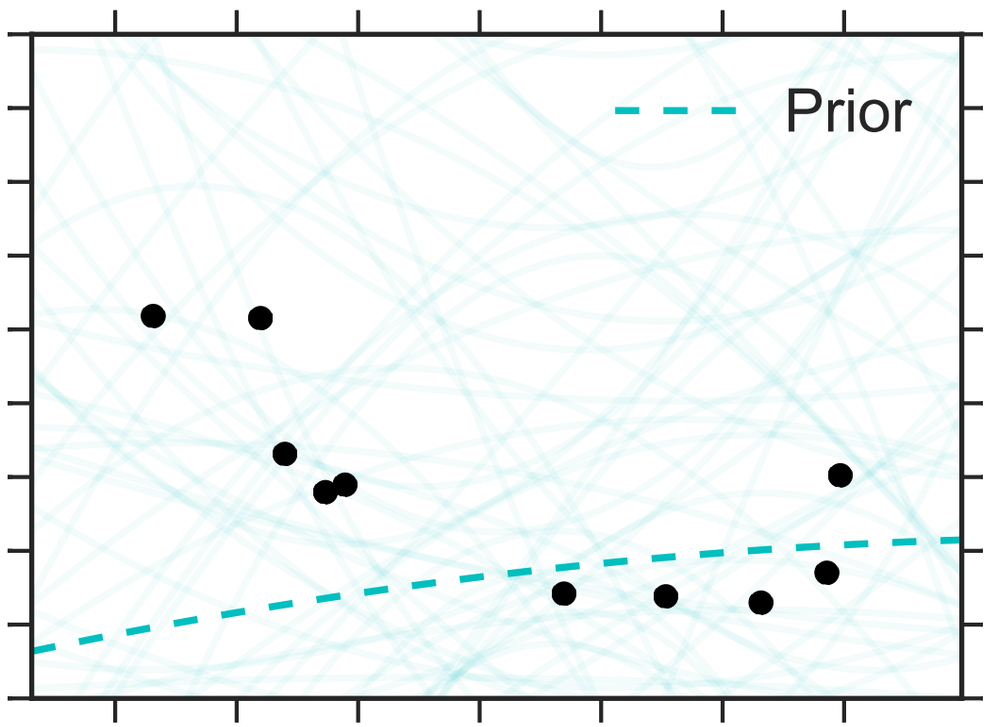
Gradient: $\nabla_\eta \mathcal{J}(\eta) = \mathbb{E}_{p(\mathbf{x},\mathbf{y})}\left[-\nabla_\eta \log q(\mathbf{x}|\varphi(\eta,\mathbf{y}))\right]$

Univariate **x**: mixture density network

Multivariate **x**: autoregressive neural density estimator

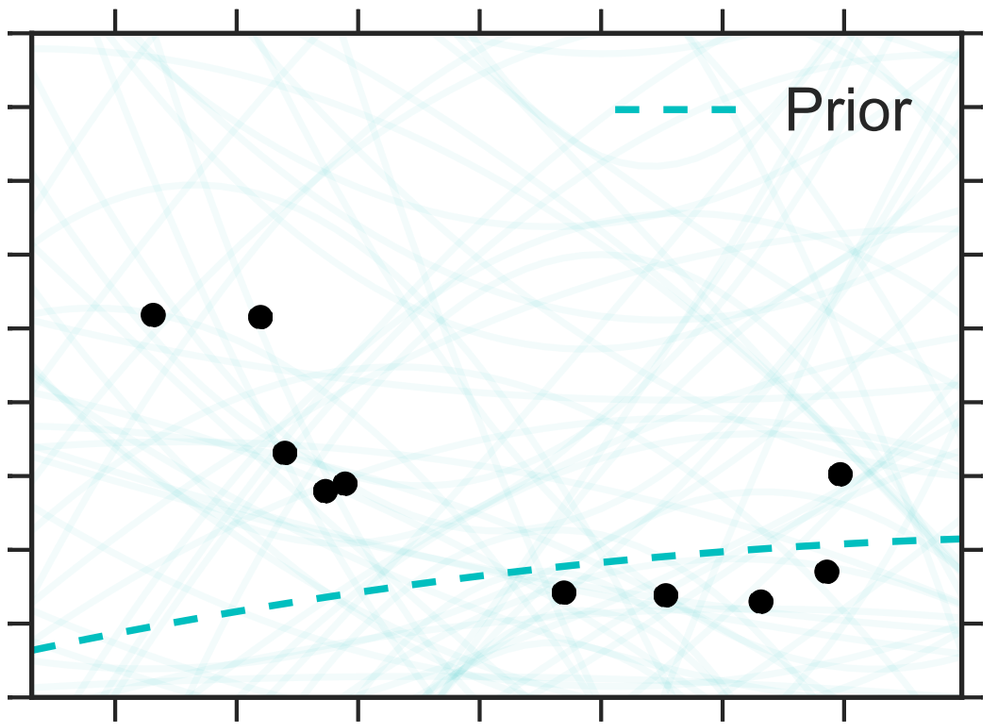
- Neural network outputs parameters of a parametric model for the next dimension, conditioned on previous dimensions
  - e.g. mixture of Gaussians, categorical, …
- MADE: efficient weight sharing for multivariate densities

Non-conjugate polynomial regression

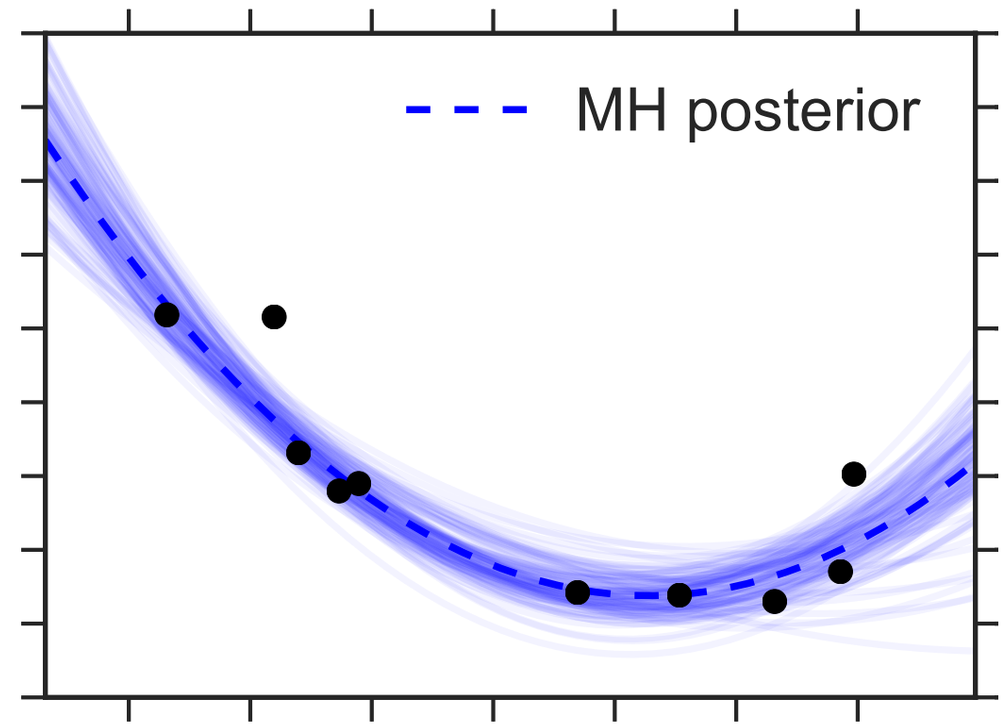


Samples from prior

Non-conjugate polynomial regression

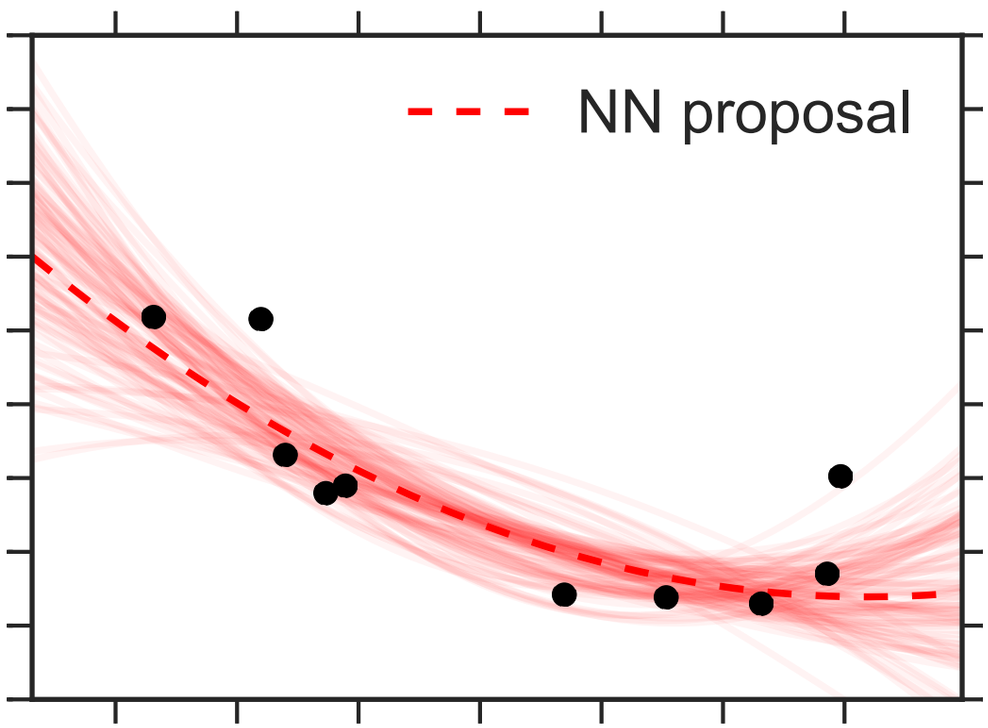


Samples from prior

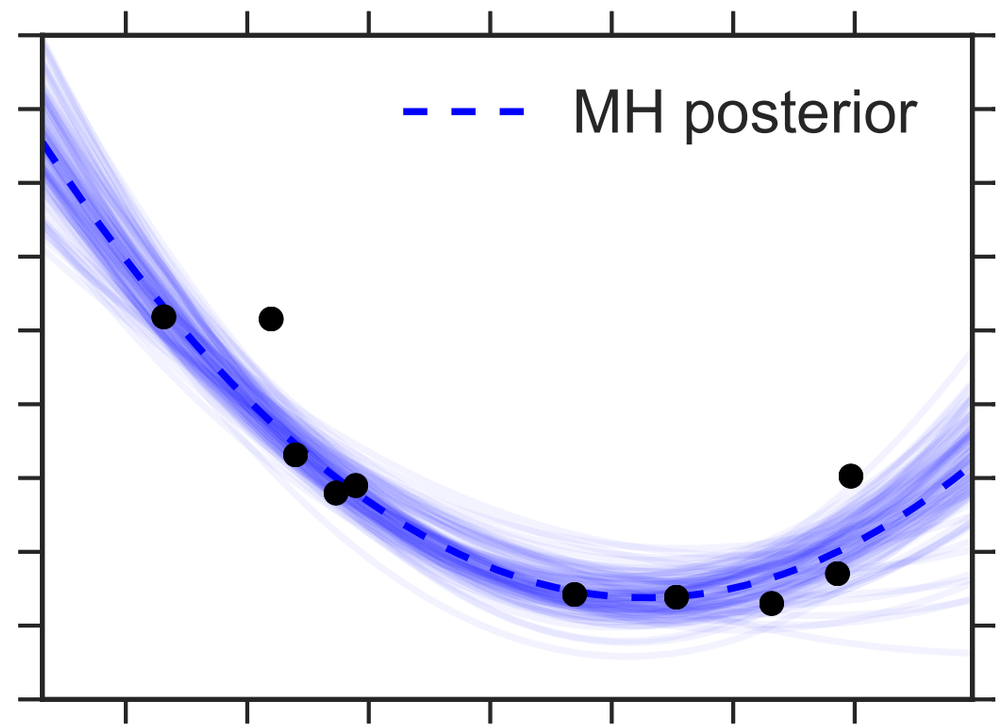


Metropolis-Hastings

Non-conjugate polynomial regression

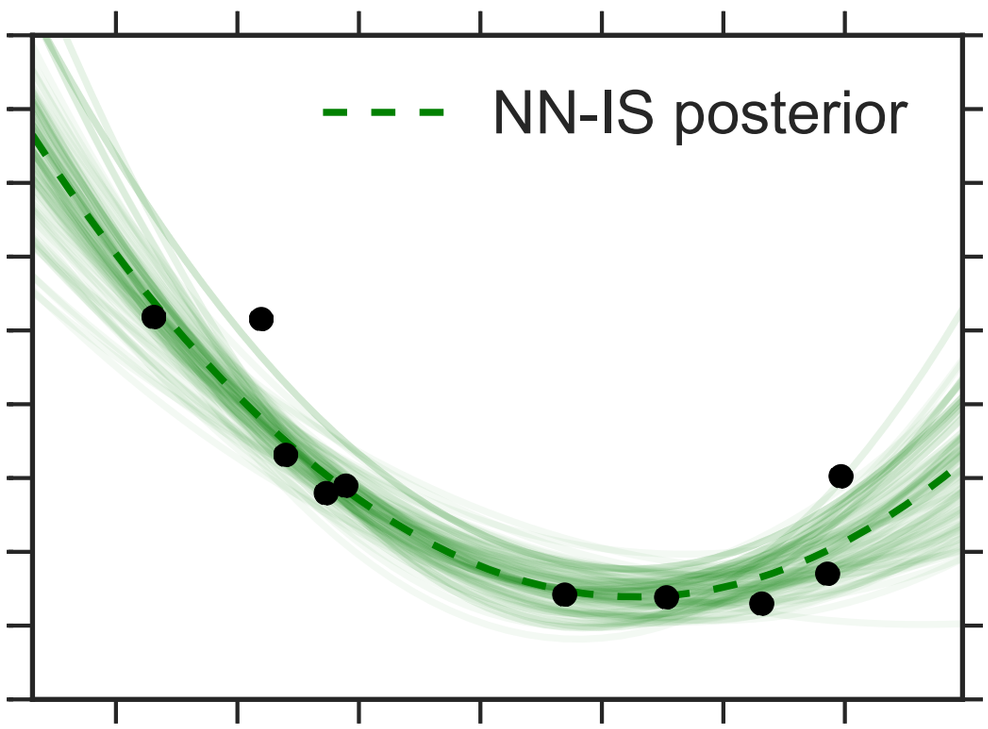


Samples from proposal

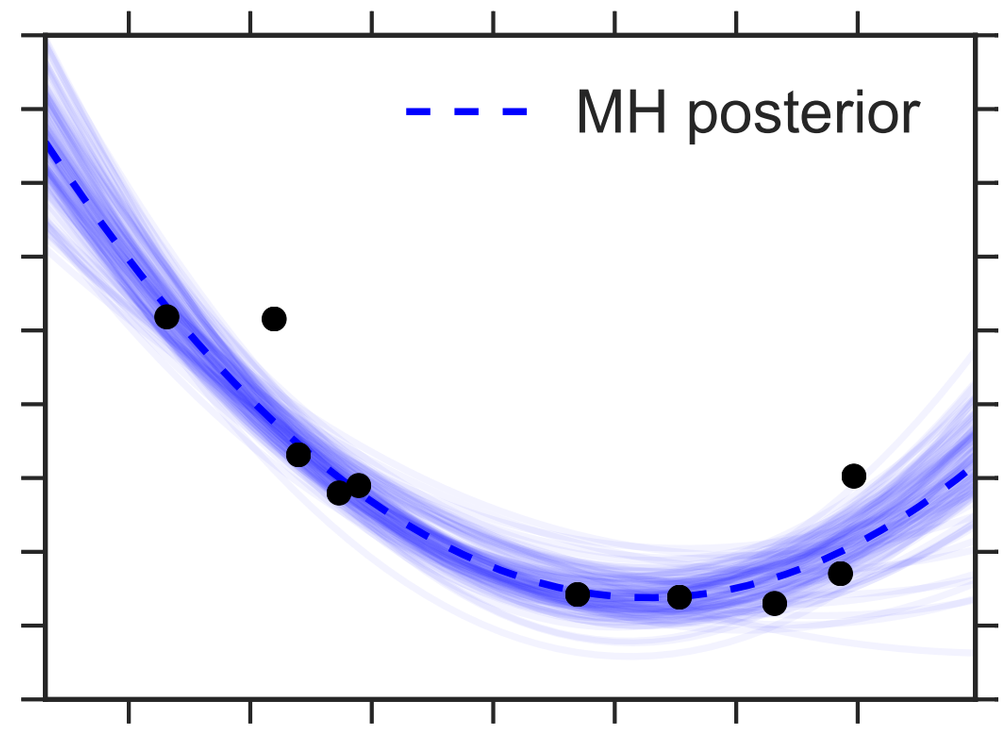


Metropolis-Hastings

Non-conjugate polynomial regression

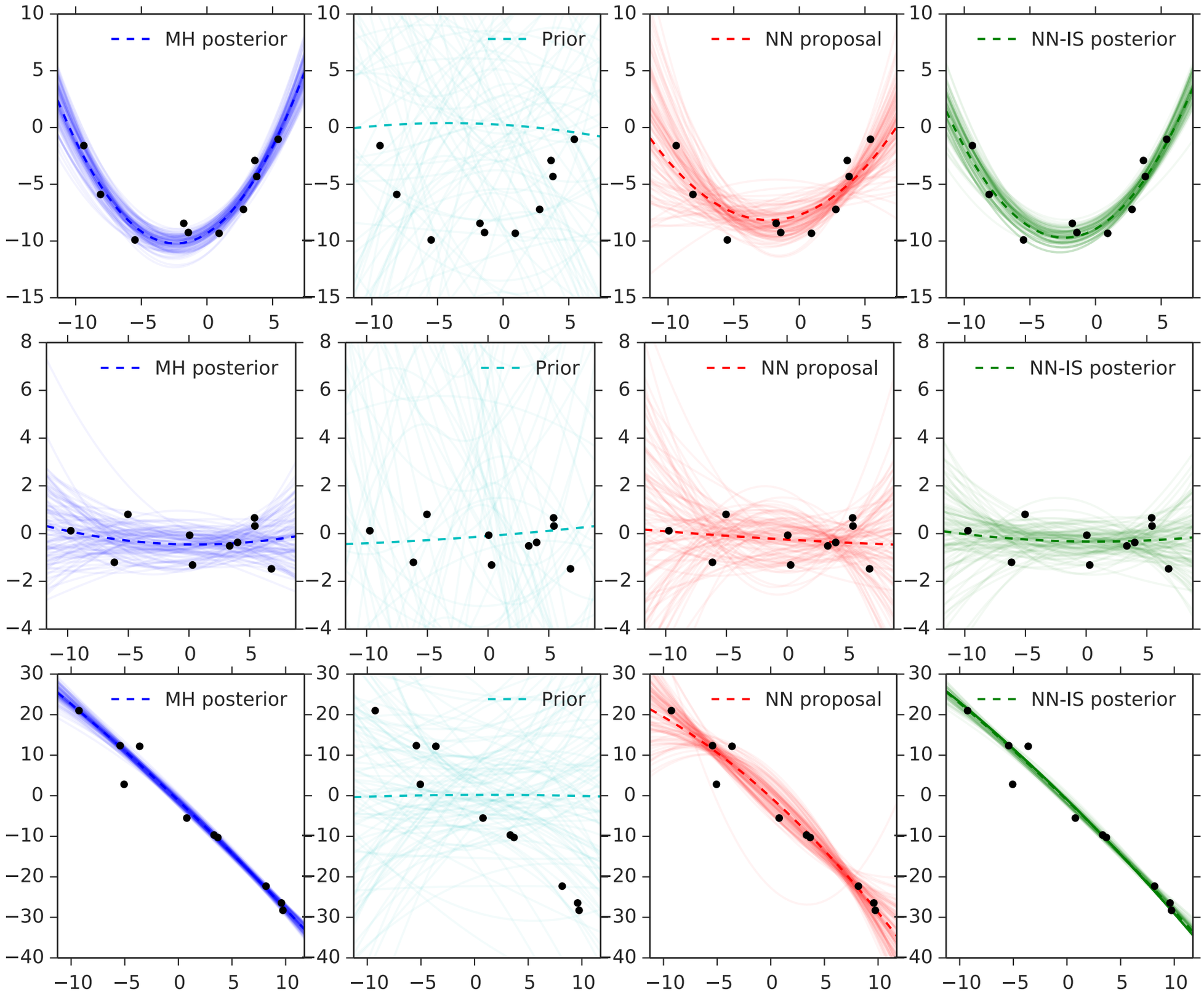


After importance weighting



Metropolis-Hastings

Non-conjugate polynomial regression



Bigger models:

Exploiting structure

Factorization of inverse models

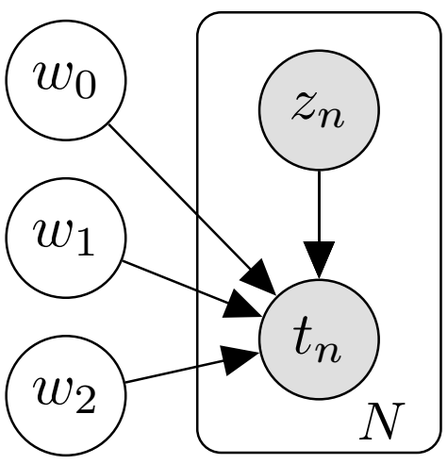
(1) There is an algorithm [Stuhlmüller et al., 2013] which takes a model and constructs an inverse model, in which the observed nodes come first.

(2) *Property:* this inverse model does not introduce any additional conditional independencies.

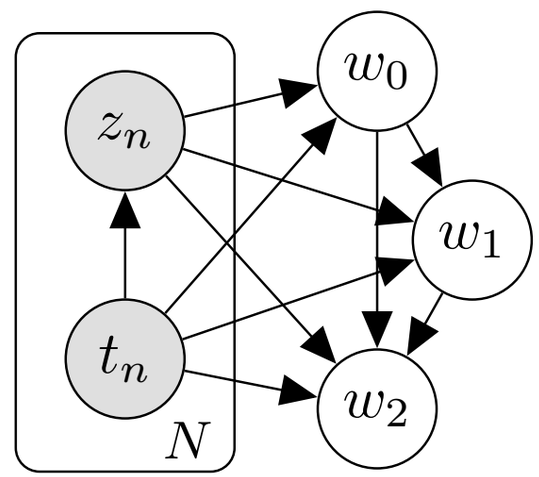
That is, if two random variables are independent given a third in the inverse model, this was also true in the original generative model.

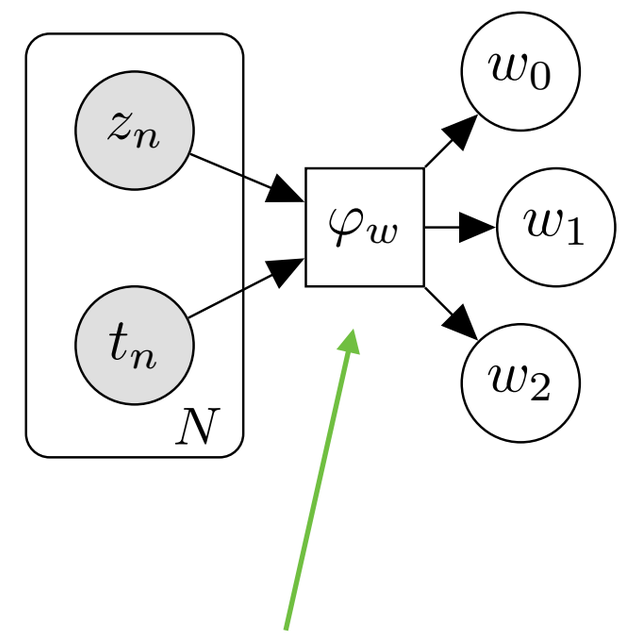
Factorization of inverse models

Generative model



Inverse model

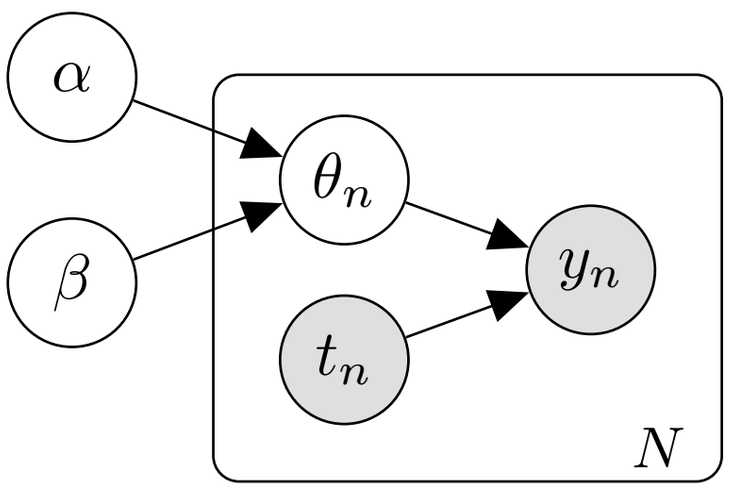




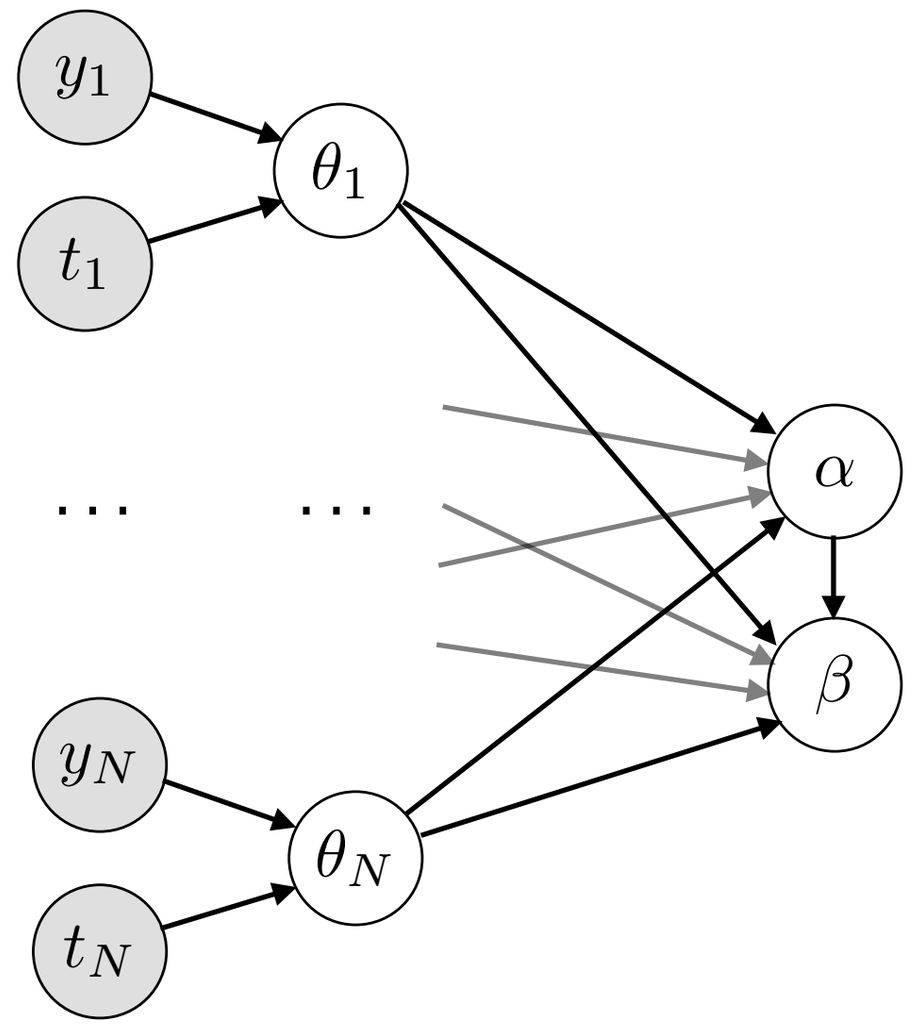
Single multivariate proposal

Inverting a multilevel model

Generative model

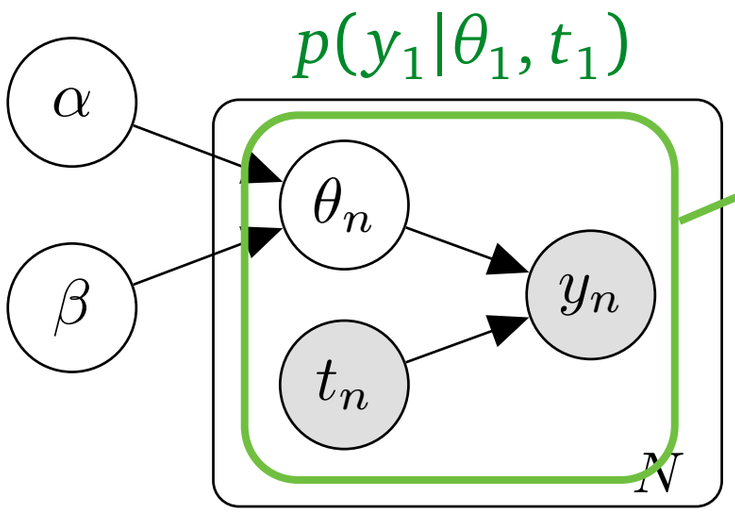


Inverse model

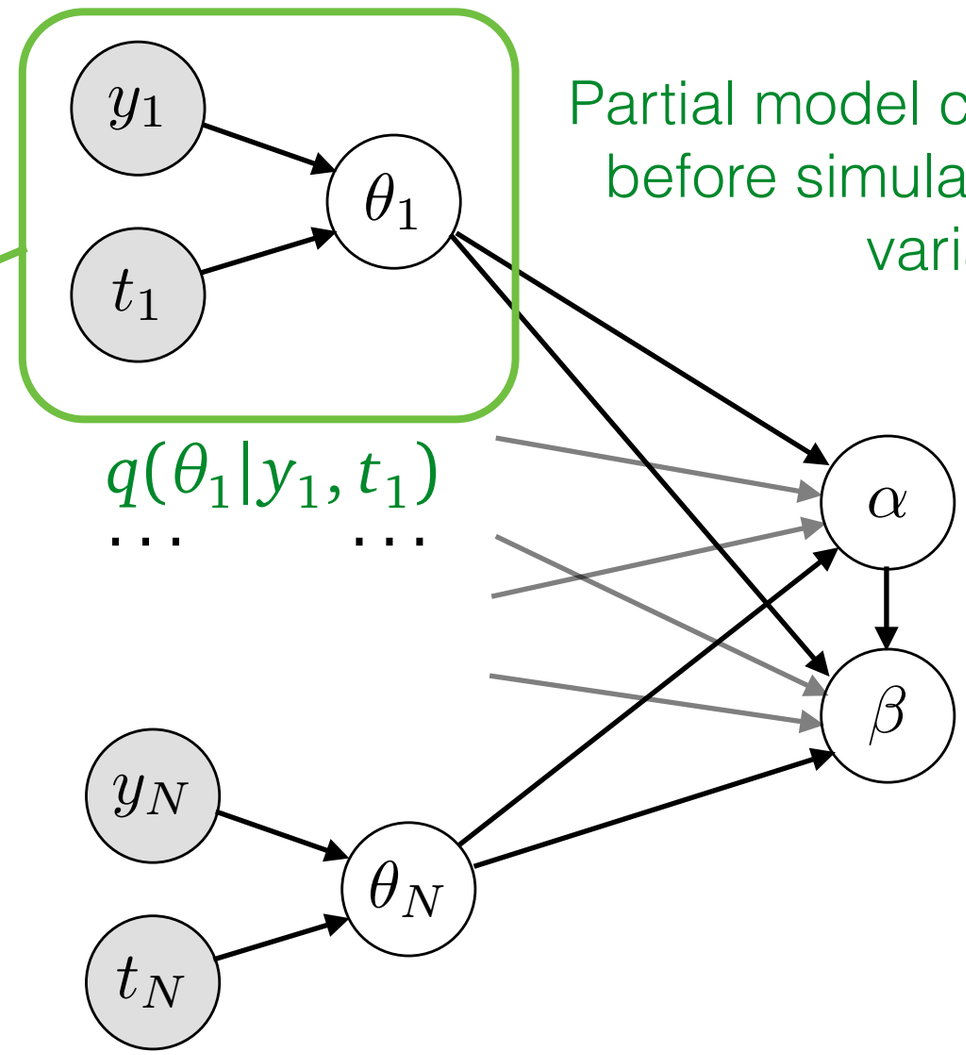


Inverting a multilevel model

Generative model



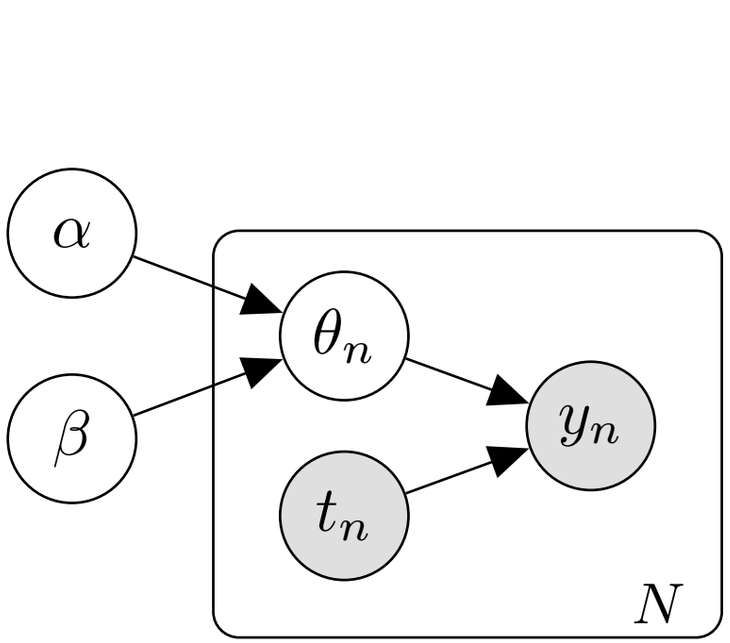
Inverse model



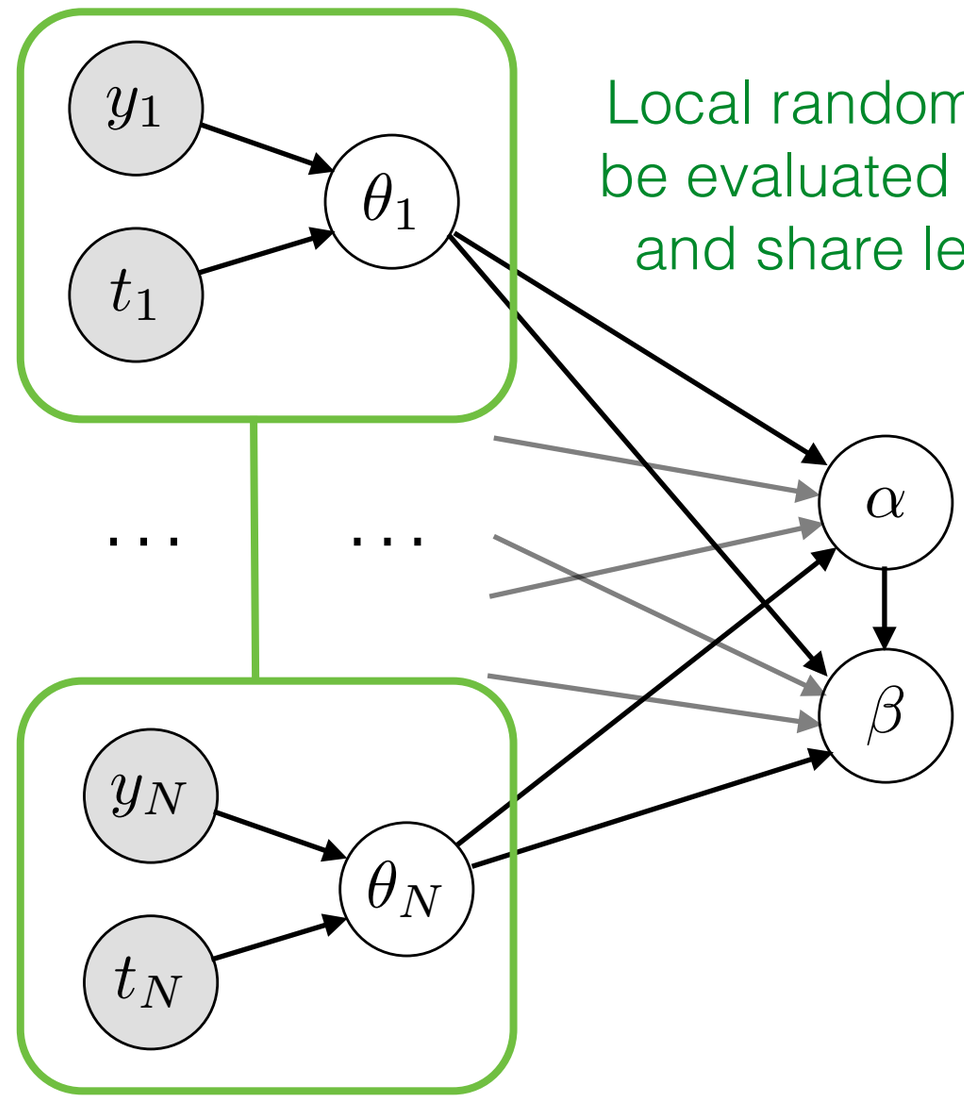
Partial model can be evaluated before simulating all random variables

Inverting a multilevel model

Generative model



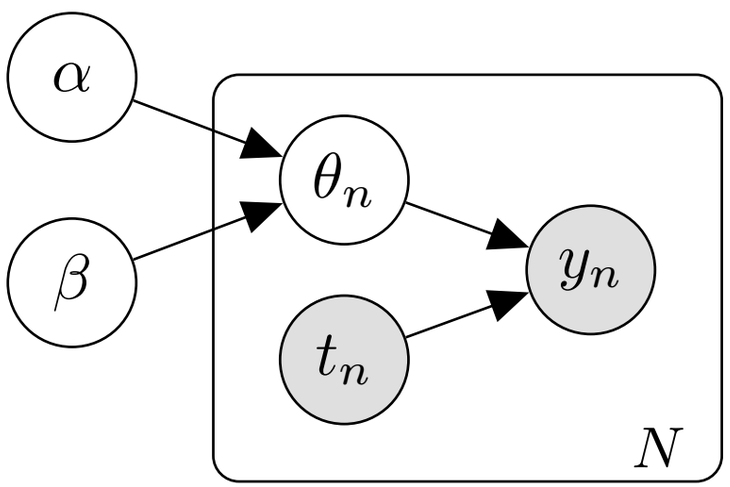
Inverse model



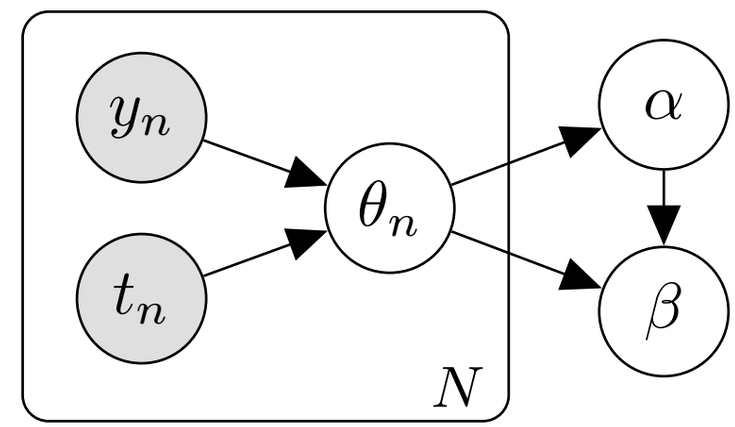
Local random variables can be evaluated independently, and share learned factors

Inverting a multilevel model

Generative model

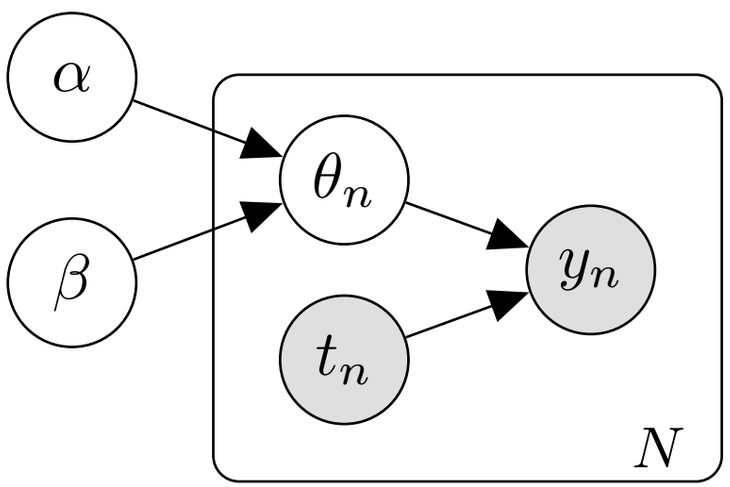


Inverse model

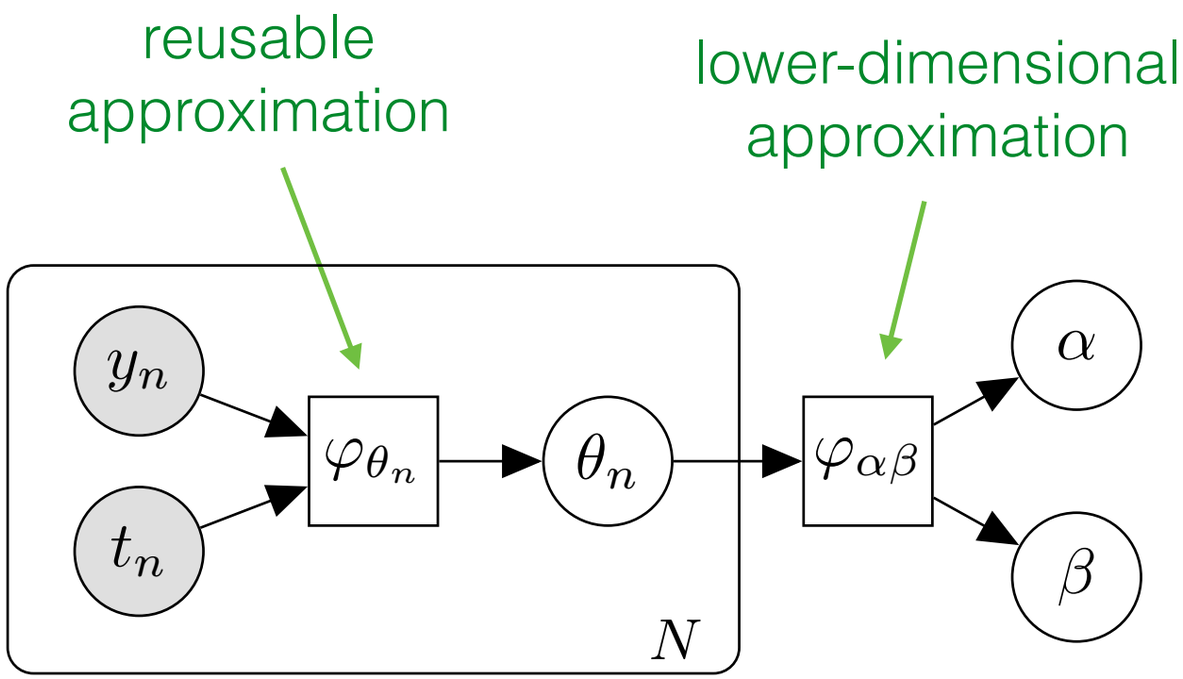


Inverting a multilevel model

Generative model

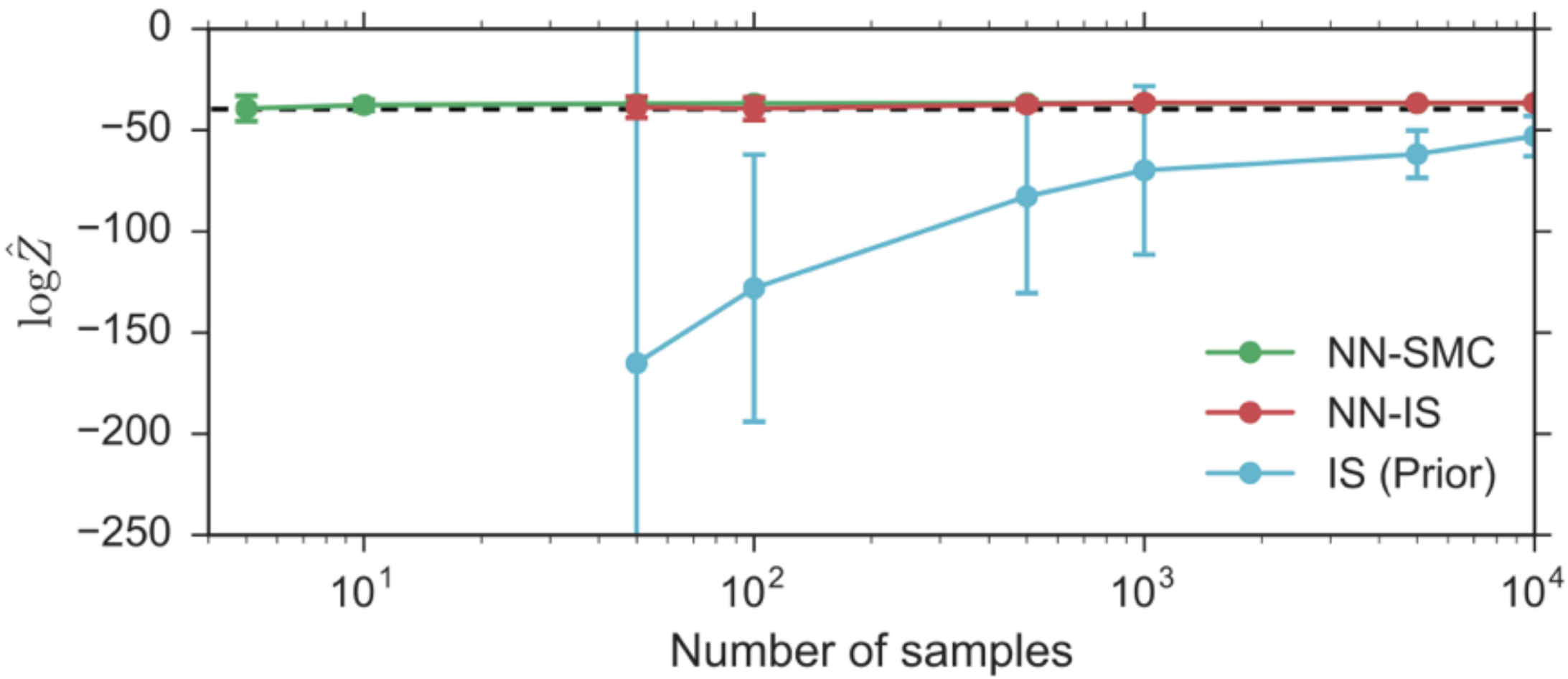


Inverse model



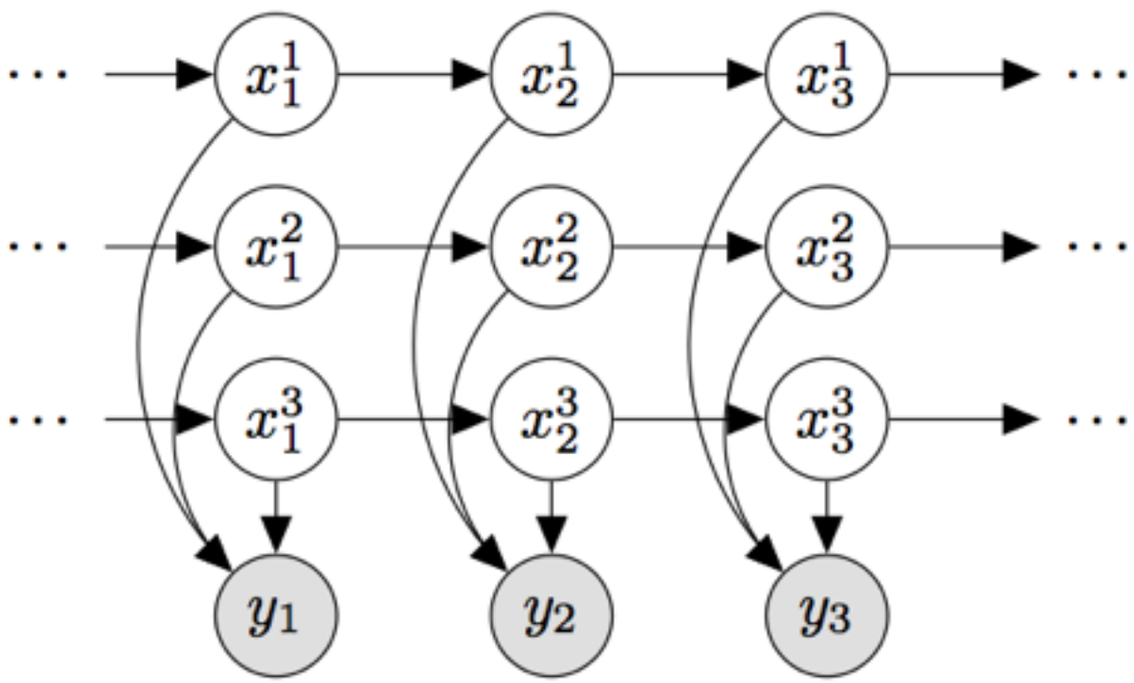
Heirarchical Poisson model

Orders of magnitude fewer samples required

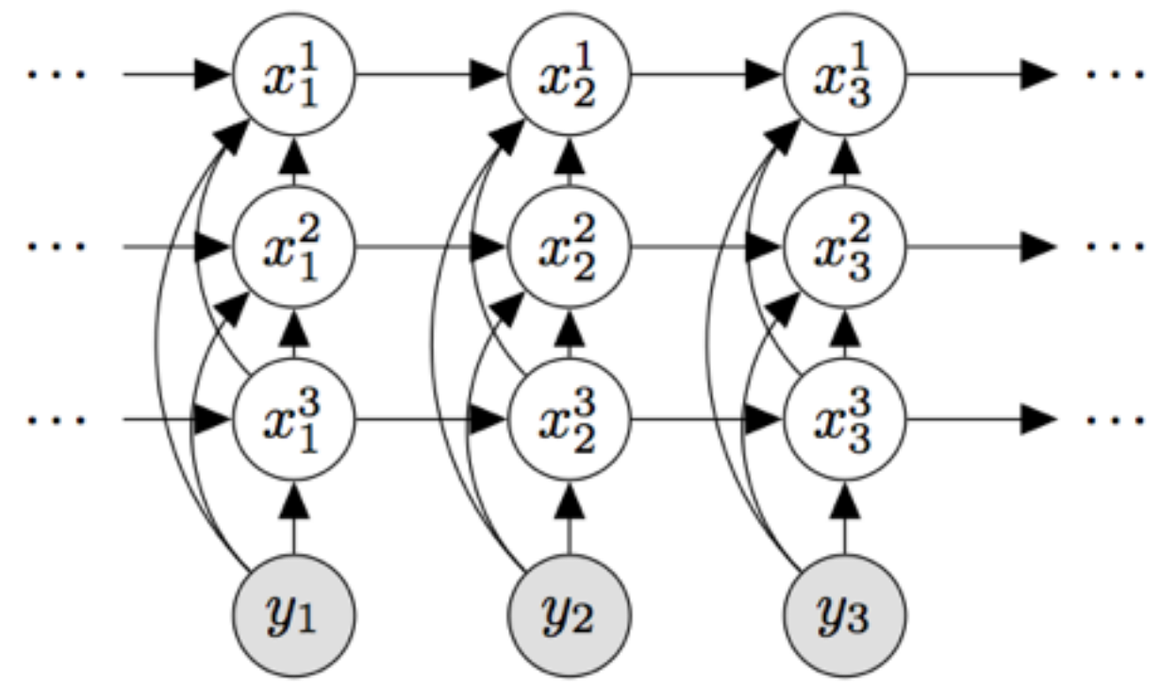


In sequential models

For models which are actually sequential, then this learns approximations to the optimal filtering proposal



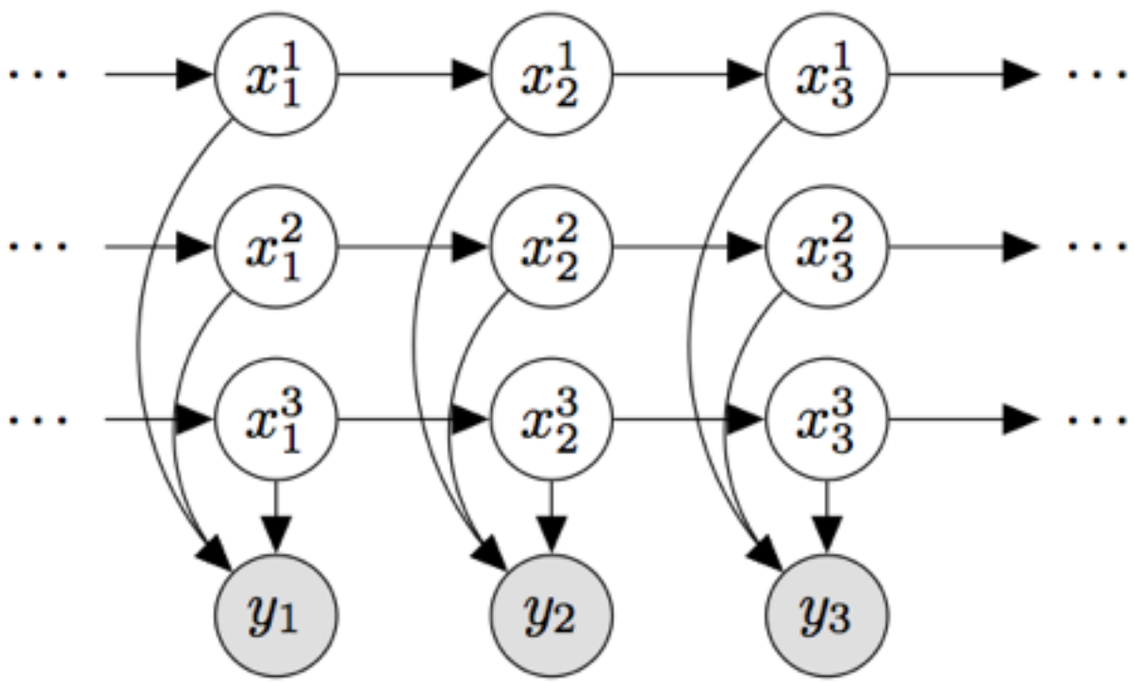
Factorial HMM
(partial figure)



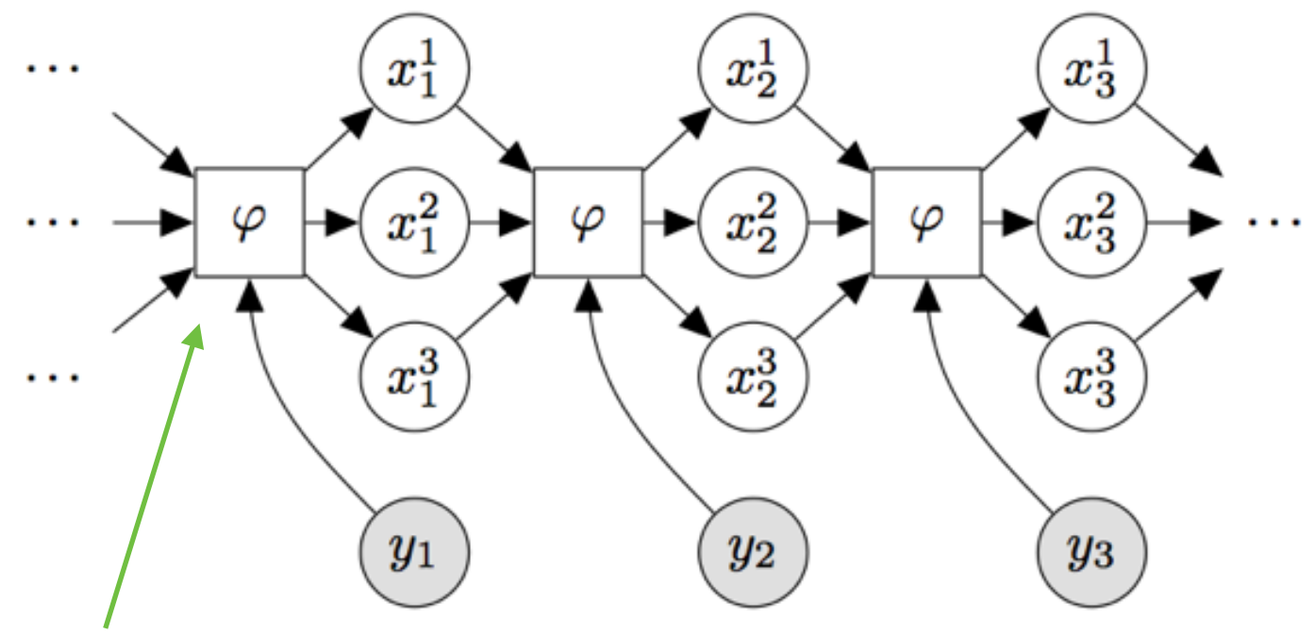
Inverting the
factorial HMM

In sequential models

For models which are actually sequential, then this learns approximations to the optimal filtering proposal



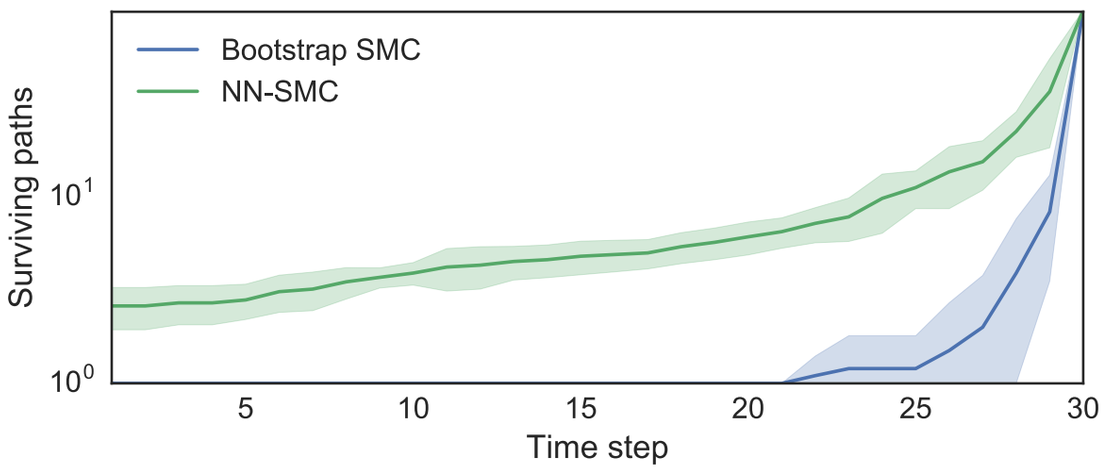
Factorial HMM
(partial figure)

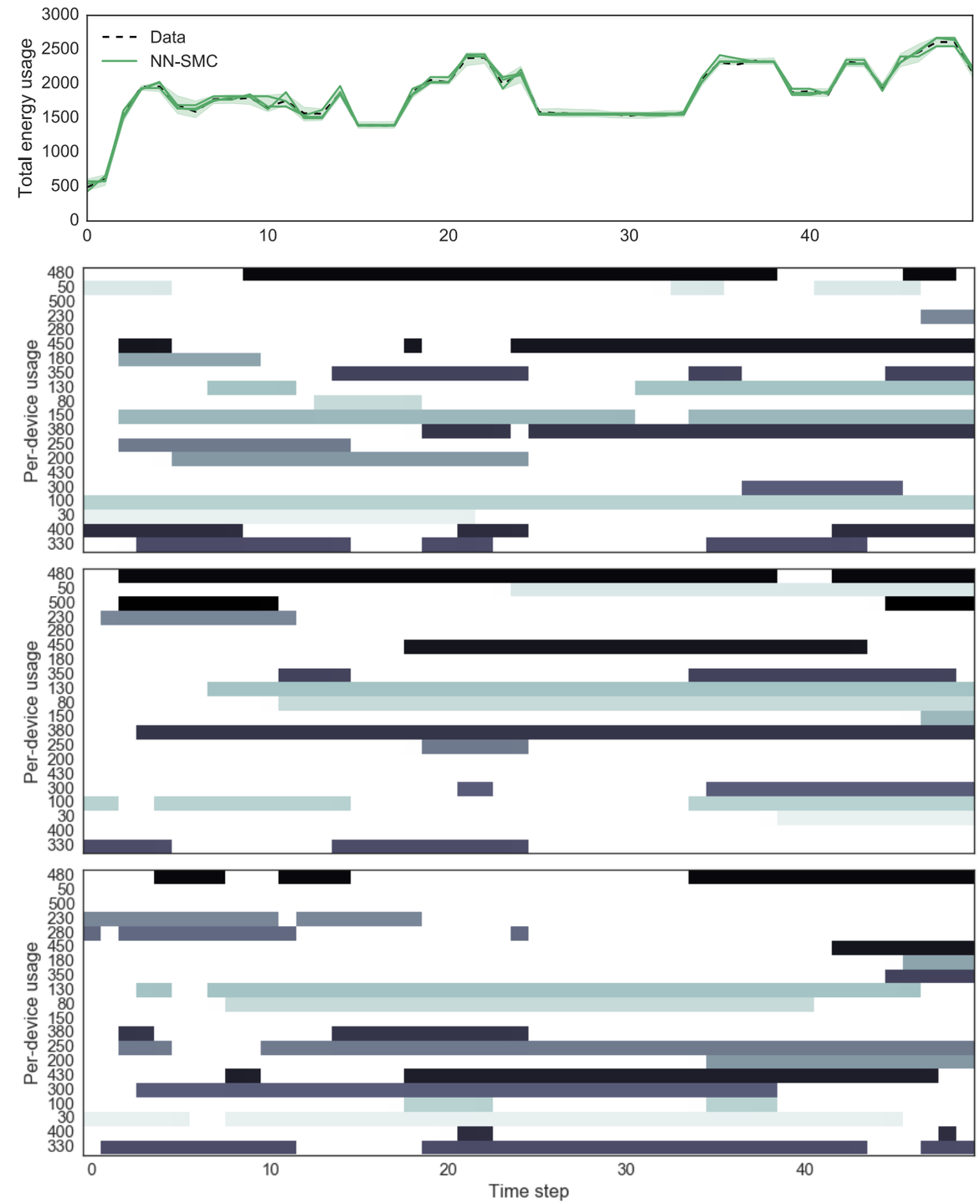


reusable
approximation

Inverting the
factorial HMM

Additive Factorial HMM





Example: energy usage disaggregation.

Combinatorial space:
2^20 or about 100k possible states at each timestep

Many diverse plausible interpretations

Discussion

We'd like to be able to completely automate this process!

- Ideally: here's a model, in some model specification language (BUGS, STAN, Anglican, …). Can we compile the model to an approximate inverse model?

- Open problems: topological sort is not unique! What makes a "good" inverse model?

  (1) structures the neural network so that training is easier (fewer overall parameters)
  (2) structures the sequence of target densities for SMC such that inference is easier