

# Probabilistic Programming

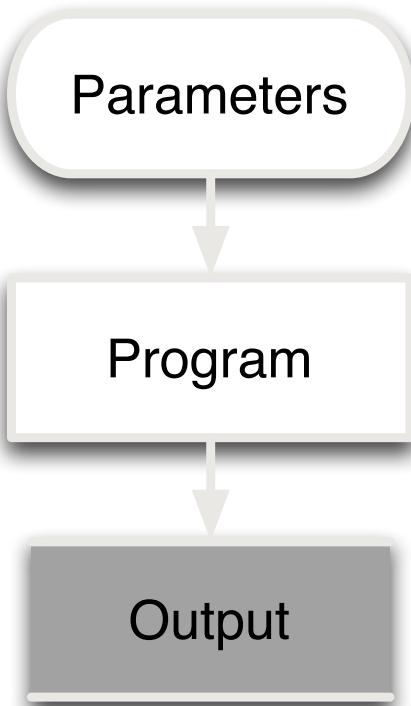
Frank Wood

[fwood@robots.ox.ac.uk](mailto:fwood@robots.ox.ac.uk)

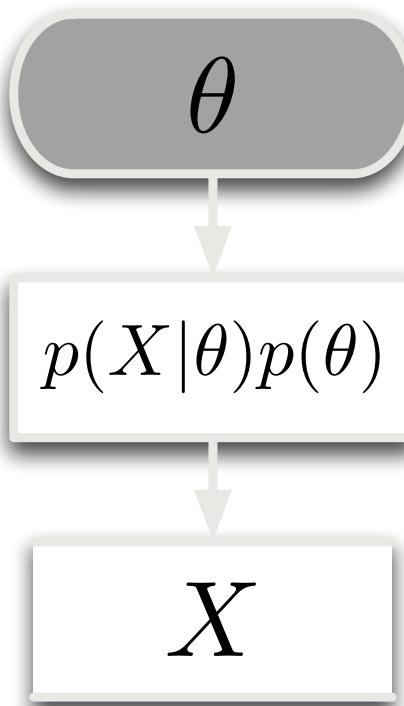


# What Is It?

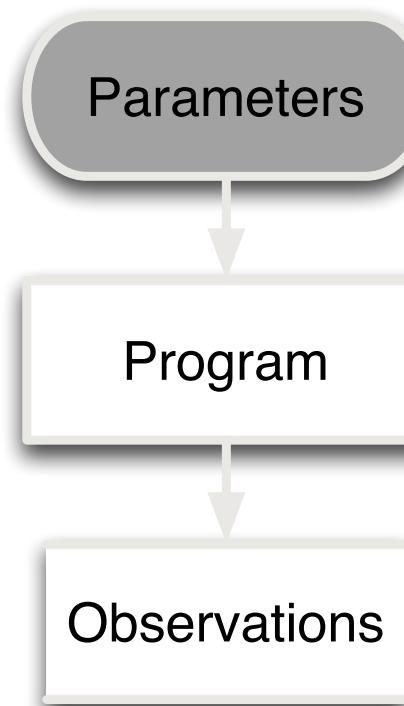
Computer Science



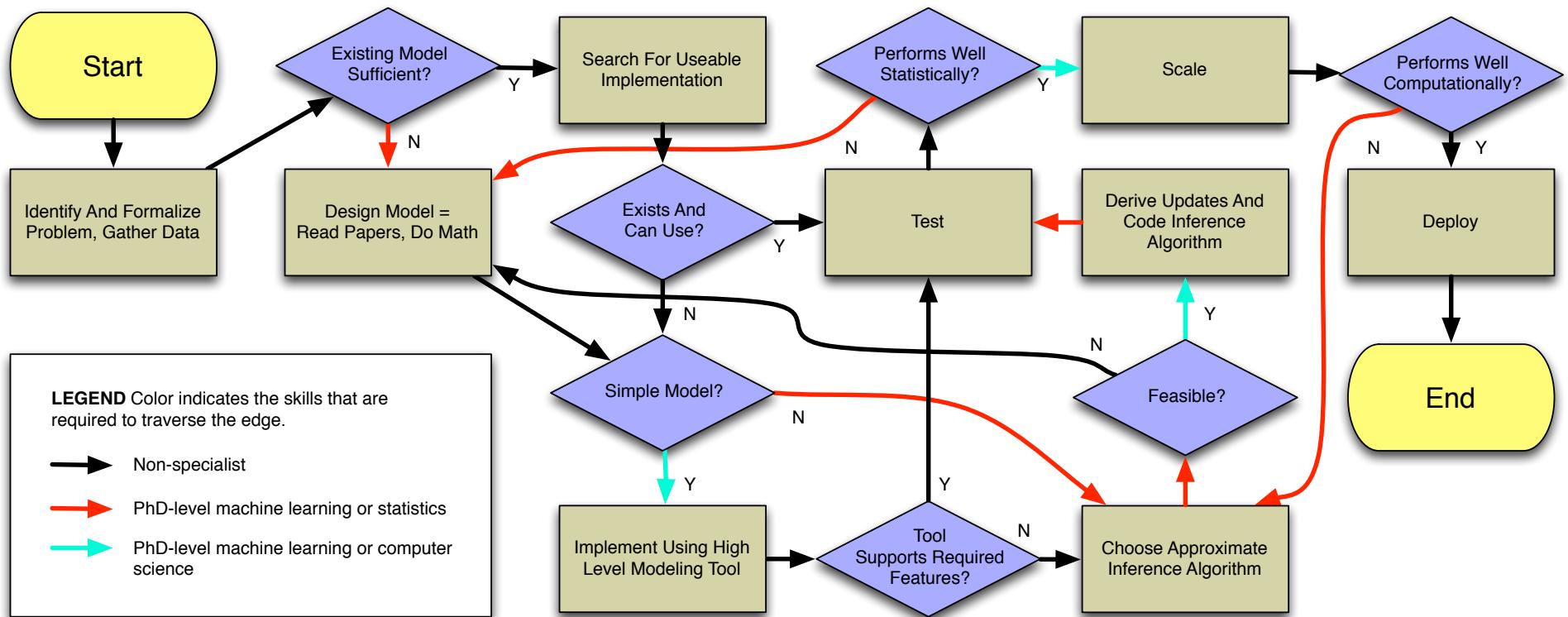
Statistics



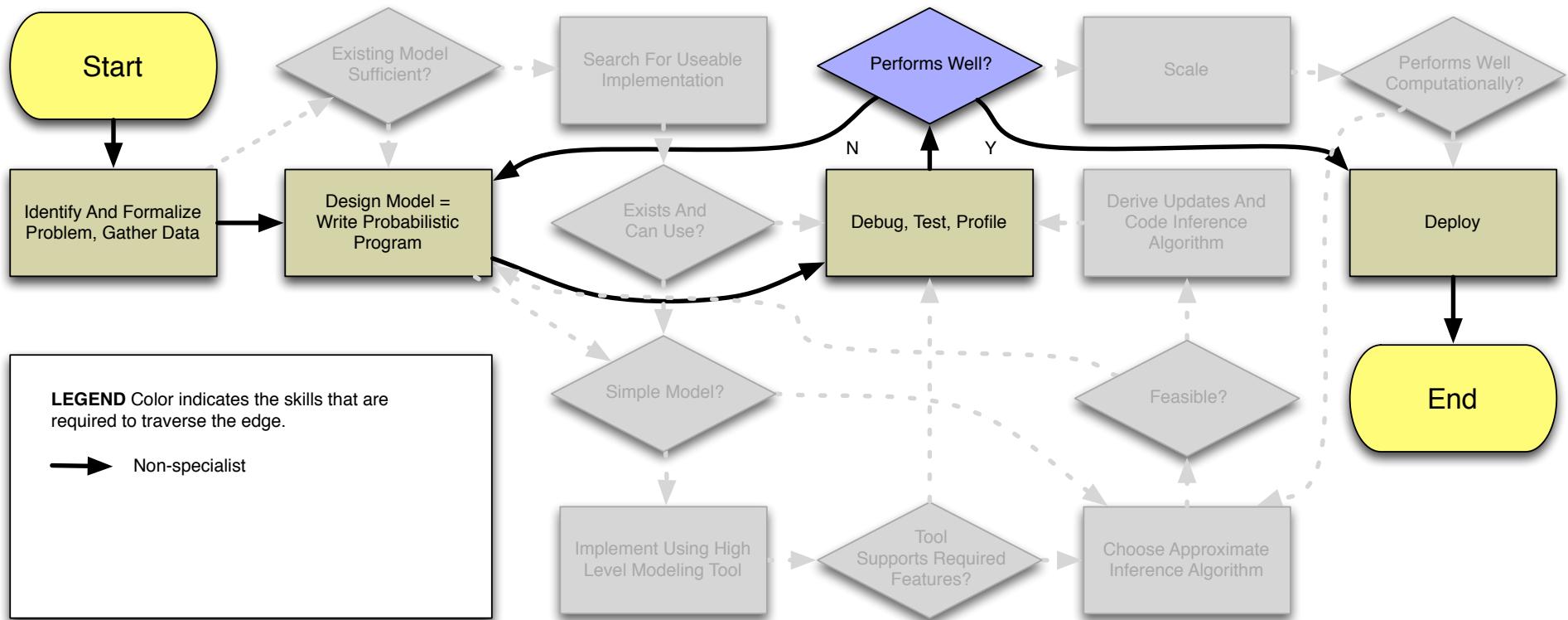
Probabilistic Programming



# The Way Machine Learning Is

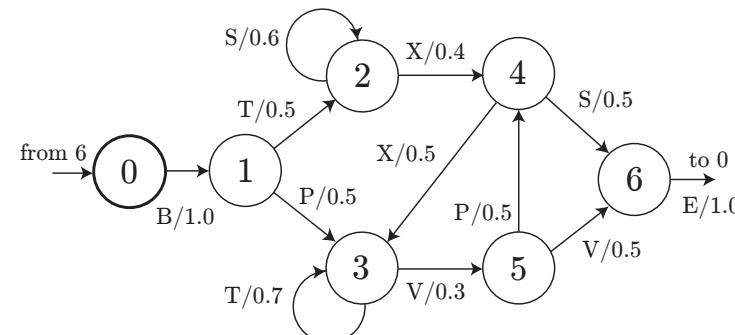
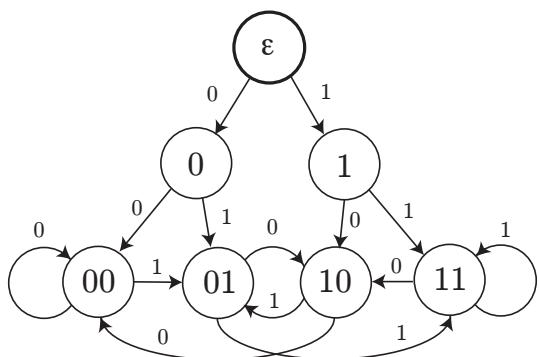


# The Way Machine Learning Will Be



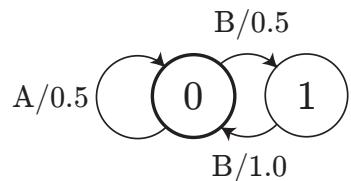
# Larger Goal: Expressive and Compact World Models For Planning and Inference

## Probabilistic Deterministic Finite Automata

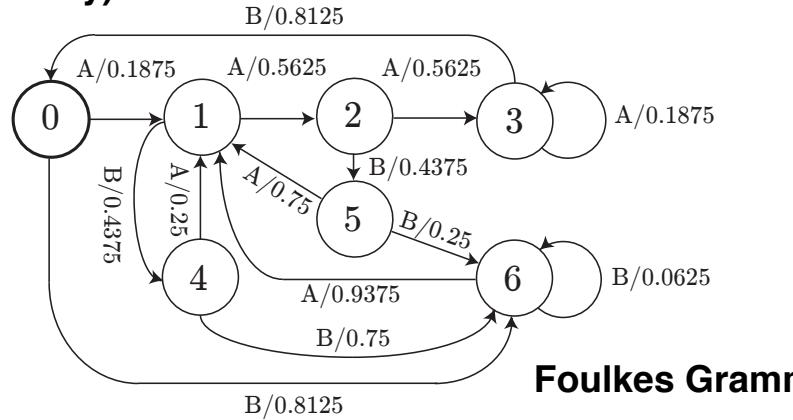


## Reber Grammar

## Trigram as DFA (without probability)



## Even Process



## Foulkes Grammar



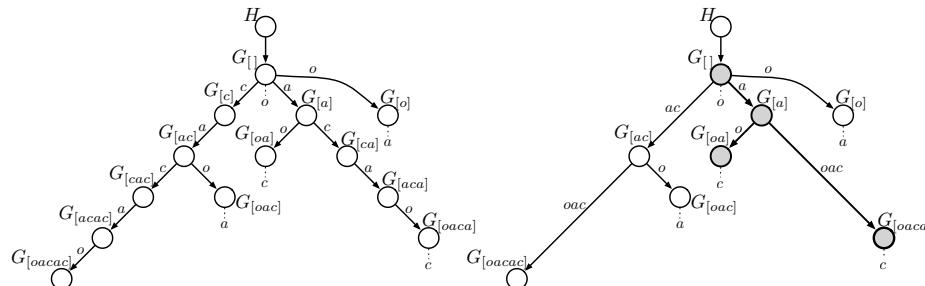
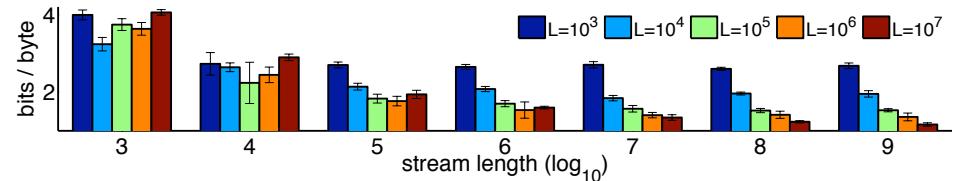
# (Streaming) Sequence Memoizer

## Model

- World state = (infinite) history of emissions
- Per-state emissions *learned*
  - Requires careful smoothing
- Deterministic transitions *fixed*
- Wikipedia next-byte predictive performance in range of Shannon's human-estimate of the entropy of written English

## Problem

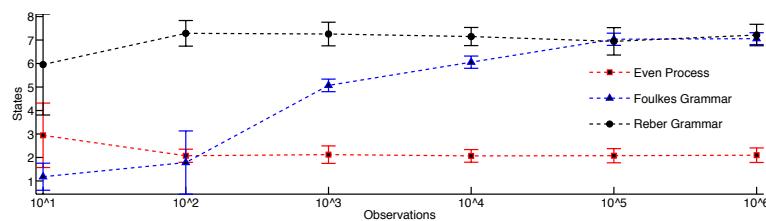
- ~2500 lines of Java code
- New student re-implementation
  - 6-12 months
- High-arity state space brings out statistical inefficiencies



# Probabilistic Deterministic Infinite Automata

## Model

- World state  $\approx$  sufficient statistic of emissions
- Per-state emissions *learned*
- Per-state deterministic transition functions *learned*
- Unsupervised PDFA structure learning biases towards compact (few states) world models that are fast approximate predictors



## Problem

- ~4000 lines of Java code
- New student re-implementation
  - 6-12 months

A Prior over PDFA with a bounded number of states			
$\sigma_0$	$\delta$	$\sigma_1$	$\sigma_2$
$q_0$	$q_3 \quad q_1 \quad q_{12}$		
$q_1$	• $q_3$ •		
$q_2$	$q_6 \quad q_4$	•	
$q_3$	$q_3$ •	$q_5$	
$q_4$	$q_3 \quad q_{42} \quad q_5$		
$q_5$	• $q_1 \quad q_2$		
:	⋮	⋮	⋮
		$\pi$	
$q_0$	(iid probability vector)	$\sigma_0$	$\sigma_1$
$q_1$	(iid probability vector)		$\sigma_2$
:	⋮	⋮	⋮
$q_0 \ q_1 \ q_3 \ q_3 \ q_5 \ q_2 \ q_4 \ q_5 \ q_2 \dots$			
1 1 0 0 2 2 1 2 1 0 ...			

$\mu \sim \text{Dir}(\alpha_0 / |Q|)$   
 $\phi_j \sim \text{Dir}(\alpha \mu) \quad j = 0, \dots, |\Sigma| - 1$   
 $\delta(q_i, \sigma_j) = \delta_{ij} \sim \phi_j \quad i = 0, \dots, |Q| - 1$   
 $\pi_{q_i} \sim \text{Dir}(\beta / |\Sigma|) \quad i = 0, \dots, |Q| - 1$   
 $\xi_0 = q_0, \xi_t = \delta(\xi_{t-1}, x_{t-1})$   
 $x_t \sim \pi_{\xi_t}$

Notation:

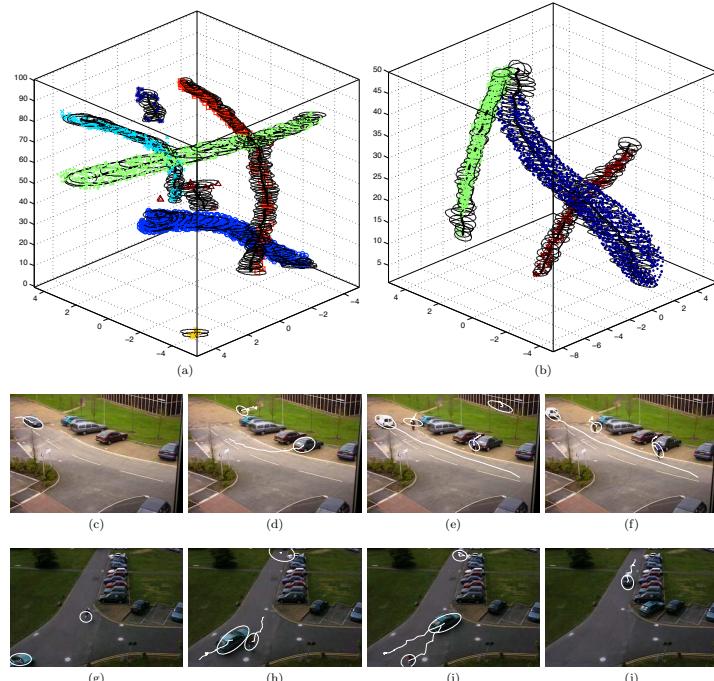
- $Q$  - finite set of states
- $\Sigma$  - finite alphabet
- $\delta : Q \times \Sigma \rightarrow Q$  - transition function
- $\pi : Q \times \Sigma \rightarrow [0, 1]$  - emission distribution
- $q_0 \in Q$  - initial state
- $x_t \in \Sigma$  - data at time t
- $\xi_t \in Q$  - state at time t
- $\alpha, \alpha_0, \beta$  - hyperparameters



# Mixture of Objects Markov Model

## Model

- World state  $\approx$  infinite mixture of objects
- Per-state, per-object emissions *learned*
- Per-state, per-object *complex* transition functions *learned*



## Problem

- ~5000 lines of Matlab code
- Implementation
  - ~1 year
- Generative model
  - ~1 page latex math
- Inference algorithm
  - ~3 pages latex math



# Motivation, Proposal, Honesty

- Existing tools for modeling are cumbersome
  - Akin to writing code in assembly language
  - Model specification and forward sampling “easy”
  - Inference hard
- Probabilistic programming systems
  - Efficient model development and testing
  - Decoupling of modeling and inference
  - More compact notation
  - Bigger more expressive models
  - Automated inference
- Problem
  - Existing probabilistic programming systems not quite there yet



# Probabilistic Programming

- Inverse Computing
- Automated Inference For Generative Modeling
- Stochastic Black-box Simulator Inversion
- “Probabilistic programs are usual functional or imperative programs with two added constructs: (1) the ability to draw values at random from distributions, and (2) the ability to condition values of variables in a program via observations. Models from diverse application areas such as computer vision, coding theory, cryptographic protocols, biology and reliability analysis can be written as probabilistic programs.”



Gordon et al, “Probabilistic Programming”, ICSE 2014

# Teaching and Research Language

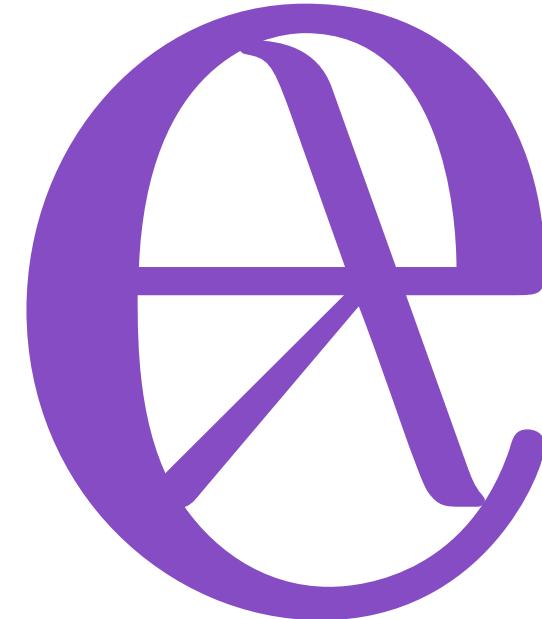
## Anglican

A “Church” of England “Venture”

<http://www.robots.ox.ac.uk/~fwood/anglican/>

*Please report bugs to*

<https://bitbucket.org/fwood/anglican/issues>



# Modeling Language Syntax

## Outer Directives

[assume symbol expr]

[observe expr value]

[predict expr]

## Inner Expressions - Scheme/Lisp/Clojure

- Functional except stochastic procedures
- Pure (no side-effects) except mem
- Higher order



# Ugh, Why Lisp?

- Redefinition prohibited (“pure functional”)

```
[assume (a (normal 5 10))]  
[assume (b (normal a 2 ))]  
[assume (a (normal b 7 ))]  
=> Error
```

- Imperative languages (i.e. Probabilistic-C) allow (!?!!)

```
int a = normal(5,10);  
int b = normal(a,2 );  
int a = normal(b,7 );
```



# Anglican Stochastic Procedures

**(flip p)** sample a single binomial trial. Returns true with probability p and false with probability 1-p.

**(gamma a b)** samples from a Gamma distribution with shape a and rate b. Returns a double on the domain (0, Inf).

**(invgamma a b)** samples from an inverse Gamma distribution with shape a and rate b. Returns a double on the domain (0, Inf).

**(normal m s)** samples from a univariate normal distribution with mean m and stdev s. Returns a double on the domain (-Inf, Inf)



# Higher Order

(lambda (& symbols) body) => compound procedure  
(lambda symbol body) => compound procedure

Constructs a compound procedure.

Example

```
((lambda (n m)
          (* (+ n 1) m))
 1 2)
```

=> 4



# Memoization

(mem proc)

Constructs a memoized procedure instance from an expression proc that must evaluate to a procedure. If a memoized procedure call is made with a previously used set of arguments, a cached value is returned instead of re-doing computation. This is typically used both to get dynamic programming for free and to incrementally construct complex datastructures.

Example

```
[assume H (mem (lambda (k) (list (normal 3 4) (gamma 1 1))))]  
[assume theta_1 (H 1)]  
[assume theta_2 (H 2)]  
[assume theta_3 (H 1)]  
[predict (= theta_1 theta_3 )] => always true
```



# Complex Control Flow

```
(if bool-expr cons-expr alt-expr)
```

Example

```
(if (= 1 (poisson 2))
    "the predicate is true"
    (normal 18 (/ 45 3.98)))
```

```
=> "the predicate is true" w.p. 0.2707
```



# Birthday Coincidence

Approximately, what's the probability that in a room filled with 23 people at least one pair of people have the same birthday?



# Solution

```
[assume birthday (mem (lambda (i) (uniform-discrete 1 366)))]  
[assume N 23]  
[assume pair-equal  
  (lambda (i j)  
    (if (> i N)  
        false  
        (if (> j N)  
            (pair-equal (+ i 1) (+ i 2))  
            (if (= (birthday i) (birthday j))  
                true  
                (pair-equal i (+ j 1))))))]  
[predict (pair-equal 1 2)]
```



# Invoking Anglican

```
anglican -s *yoursourcefile*
```

- or -

```
cat *yoursourcefile* | anglican
```

Some command line switches

Switches	Default	Desc
-----	-----	-----
-h, --no-help, --help	false	Show help
-s, --source-file	*in*	Anglican source file to interpret
-p, --predict-file	*out*	File into which to print predicts
-n, --num-samples	Infinity	Number of samples



# Anglican Semantics Lite

- Applying a (random) procedure generates a sample.
- Running an Anglican program yields a stream of predict expression samples generated from a sequence of program execution paths sampled via a Markov chain Monte Carlo exploration of the space of execution paths.



# Inference

$$\mu \sim N(1, 5)$$

$$y_i | \mu \sim N(\mu, 2)$$

$$y_1 = 9$$

$$y_2 = 8$$

$$\mu | y_{1:2} \sim N(7.25, 0.8333)$$



# Solution

```
[assume sigma (sqrt 2)]  
[assume mu (normal 1 (sqrt 5))]  
[observe (normal mu sigma) 9]  
[observe (normal mu sigma) 8]  
[predict mu]
```



# Addition

What numbers added together equal seven?



# Solution

```
[assume a (- (poisson 100) 100)]  
[assume b (- (poisson 100) 100)]  
[observe (normal (+ a b) .00001) 7]  
[predict (list a b)]
```



# Anglican Semantics

- Running an Anglican program yields a stream of predict expression samples generated from a dependent sequence of program execution paths sampled via a Markov chain Monte Carlo exploration of the posterior of execution paths conditioned on observed data.
- Test function averages converge in the usual sense.



# Leaving The Beaten Path

$$\mu \sim \text{Poisson}(1)$$

$$y_i | \mu \sim N(\mu, 2)$$

$$y_1 = 9$$

$$y_2 = 8$$

$$\mu | y_{1:2} \sim ?$$



# Solution

```
[assume sigma (sqrt 2)]  
[assume mu (poisson 1)]  
[observe (normal mu sigma) 9]  
[observe (normal mu sigma) 8]  
[predict mu]
```



# Multivariate Logistic Regression

$$\sigma^2 \sim \text{Gamma}(1, 1)$$

$$\beta_j \sim \text{Normal}(0, \sigma^2)$$

$$p(z_i = 1) = \frac{1}{1 + e^{-\beta^T x}}$$



# Solution

```
[assume dot-product (lambda (u v)
  (if (= (count u) 0)
      0
      (+ (* (first u) (first v))
          (dot-product (rest u) (rest v))))]
[assume sigma (sqrt (gamma 1 1))]
[assume beta (list (normal 0 sigma) (normal 0 sigma) (normal 0 sigma) (normal 0 sigma)
(normal 0 sigma))]
[assume z (lambda (x)
  (/ 1 (+ 1 (exp (* -1 (dot-product beta x))))))]
[observe-csv
  "http://www.robots.ox.ac.uk/~fwood/anglican/examples/logistic_regression/iris.csv"
  (flip (z (list 1 $1 $2 $3 $4))) (= $5 "Iris-setosa")]
[predict beta]
; should be Iris-setosa, i.e. 1 (from training data)
[predict (z (list 1 5.1 3.5 1.4 0.2))]
; should be Iris-virginica, i.e. 0 (from training data)
[predict (z (list 1 7.7 2.6 6.9 2.3))]
```



# Hidden Markov Model

```
[assume initial-state-dist (list (/ 1 3) (/ 1 3) (/ 1 3))]  
[assume get-state-transition-dist  
  (lambda (s) (cond ((= s 0) (list .1 .5 .4))  
                      ((= s 1) (list .2 .2 .6))  
                      ((= s 2) (list .15 .15 .7))))]  
[assume transition (lambda (prev-state)  
  (discrete (get-state-transition-dist prev-state)))]  
[assume get-state (mem (lambda (index)  
  (if (<= index 0) (discrete initial-state-dist)  
      (transition (get-state (- index 1))))))]  
[assume get-state-observation-mean  
  (lambda (s) (cond ((= s 0) -1)  
                      ((= s 1) 1)  
                      ((= s 2) 0)))]  
[observe (normal (get-state-obs-mean (get-state 1 )) 1) .9]  
[observe (normal (get-state-obs-mean (get-state 2 )) 1) .8]  
...  
[observe (normal (get-state-obs-mean (get-state 16)) 1) -1]  
[predict (get-state 0)]  
[predict (get-state 1)]  
...  
[predict (get-state 16)]
```



# Bayesian Nonparametrics

```
[assume H (lambda ()
  (begin
    (define v (/ 1.0 (gamma 1 10)))
    (list (normal 0 (sqrt (* 10 v))) (sqrt v))))]
[assume gaussian-mixture-model-parameters (DPmem 1.72 H)]
[observe-csv "http:// ... "
  (apply normal (gaussian-mixture-model-parameters)) $2]
[predict (apply normal (gaussian-mixture-model-parameters))]
```

Example:

```
curl -s http://www.robots.ox.ac.uk/~fwood/anglican/examples/
dp_mixture_model/dp-church.anglican | anglican | grep 'normal' |
awk -F',' '{print $2}' | feedgnuplot --stream --histogram 0 --with
boxes --xlabel 'x' --ylabel Frequency --binwidth 1
```



# Expressivity

- Easily implement modern machine learning methods
  - 10's of lines of code
- Higher-order functionality



# Symbolic Function Induction

What's the next value? And the function?

Input	Output
1	5
2	3
3	1
4	?



# Solution

```
[assume get-int-constant
  (lambda () (uniform-discrete 0 10))]

[assume safe-div
  (lambda (x y) (if (= y 0) 0 (/ x y))]

[assume pcfg
  (lambda ()
    (define expression-type (discrete (list 0.40 0.30 0.30)))
    (cond
      ((= expression-type 0) (get-int-constant))
      ((= expression-type 1) 'x)
      (else
        (list
          (nth (list (quote +) (quote -) (quote *) (quote safe-div))
            (discrete (list 0.25 0.25 0.25 0.25)))
          (pcfg) (pcfg))))))

[assume induced-procedure-code (list 'lambda (list 'x) (pcfg))]
[assume induced-procedure (eval induced-procedure-code)]

[assume noise 0.00001]
[observe (normal (induced-procedure 1) noise) 5]
[observe (normal (induced-procedure 2) noise) 3]
[observe (normal (induced-procedure 3) noise) 1]

[predict induced-procedure-code]
[predict (induced-procedure 4)]
```



# Goals and Aims

## (i) Accelerate iteration over models

- Inference is automatic
- Writing generative code is easier than deriving model inverses
- Lower technical barrier of entry to development of new models

## (ii) Accelerate iteration over inference procedures

- Computer language is an abstraction barrier
  - Inference procedures can be tested against a library of models
  - Inference procedures become “compiler optimizations”

## (iii) Enable development of more expressive models

- Probabilistic programs can express a superset of graphical models
- Modern machine learning models are tens of lines of code



# How Does it Work?



# Probabilistic Programming Concepts

- First half
  - Procedures “sample”
  - Programs are generative models
- This half
  - “Sampling” execution traces = inference
  - Different traces arise from stochastic procedure outputs
    - Elementary
      - flip, normal, discrete, poisson, gamma, ...
    - Compound
      - (lambda (a b) (if (flip a)  
(+ (poisson b) 7)  
(normal a b)))
  - Various sampling algorithms apply
    - Rejection sampling
    - Metropolis Hastings
    - Sequential Monte Carlo
    - Particle Markov Chain Monte Carlo



# Outline

- Trace Probability
- Probabilistic Program Interpretation
- Monte Carlo-based probabilistic inference
  - Rejection Sampling
  - MCMC
  - SMC
  - PMCMC



# Probabilistic Inference

Inference, prediction, and inspection can all be expressed as expectations

$$\mathbb{E}[f] \equiv \int f(\mathbf{x}) p(\mathbf{x}) d\mathbf{x}$$

Where  $\mathbf{x}$  is *all* latent variables,  $f$  is a test function, and  $p$  is the distribution against which we're integrating.



# Monte Carlo Integration

## Recipe

- ① Sample  $\mathbf{x}^{(\ell)} \sim p(\mathbf{x})$  for  $\ell = 1 \dots L$
- ② Estimate  $\mathbb{E}[f] \approx \hat{f} = \frac{1}{L} \sum_{\ell=1}^L f(\mathbf{x}^{(\ell)})$



# How To Sample Execution Traces

- What is an execution trace?
- What is its probability?



# Execution Trace Probability

Posterior Distribution of Trace Given Observations

$$p(\mathbf{x}_{1:N} | \mathbf{y}_{1:N}) \propto \tilde{p}(\mathbf{y}_{1:N}, \mathbf{x}_{1:N}) \equiv \prod_{n=1}^N g(y_n | \theta_{t_n}, \mathbf{x}_{1:n}) f(\mathbf{x}_n | \mathbf{x}_{1:n-1})$$

Joint Distribution of Trace And Observations

Observed value

Parameter of observation distribution

Type of observation distribution

Interpreter memory state

Parameter of stochastic procedure

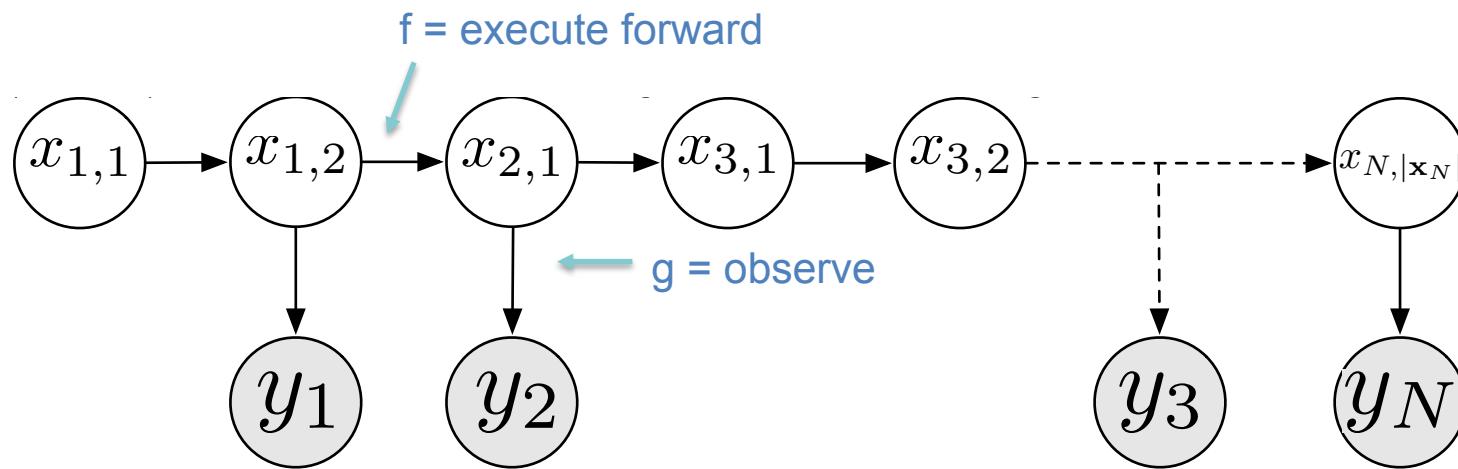
$$f(\mathbf{x}_n | \mathbf{x}_{1:n-1}) = \prod_{k=1}^{|\mathbf{x}_n|} f(x_{n,k} | \theta_{t_{n,k}}, x_{n,1:(k-1)}, \mathbf{x}_{1:(n-1)})$$

Type of stochastic procedure



# Suggests Relationship To State Space Modeling

- Program generates all random variables



- State is interpreter memory state
- Transition is stochastic procedure application
- Only observes need be indexed



# Program Interpretation

## eval

```
(define (eval exp env)
  (cond
    ((self-evaluating? exp) exp)
    ((variable? exp) (lookup-variable-value exp env))
    ((quoted? exp) (text-of-quotation exp))
    ((assignment? exp) (eval-assignment exp env))
    ((definition? exp) (eval-definition exp env))
    ((if? exp) (eval-if exp env))
    ((lambda? exp)
      (make-procedure (lambda-parameters exp)
                     (lambda-body exp) env))
    ((begin? exp)
      (eval-sequence (begin-actions exp) env))
    ((cond? exp) (eval (cond->if exp) env))
    ((application? exp)
      (apply (eval (operator exp) env)
            (list-of-values (operands exp) env))))
    (else
      (error
        "Unknown expression type -- EVAL" exp))))
```

## apply

```
(define (apply procedure arguments)
  (cond
    ((primitive-procedure? procedure)
     (apply-primitive-procedure procedure arguments))
    ((compound-procedure? procedure)
     (eval-sequence
       (procedure-body procedure)
       (extend-environment
         (procedure-parameters procedure)
         arguments
         (procedure-environment procedure)))
     )
    )
    )
  (else
    (error
      "Unknown procedure type -- APPLY" procedure))))
```



# What is $x_1$ ?

$$\begin{aligned}x_{1,1} &\sim \text{Beta}(7, 4) \\x_{1,2}|x_{1,1} = 0.4 &\sim \text{Poisson}(8) \\x_{1,3}|x_{1,2} = 6, x_{1,1} = 0.4 &\sim \text{Binomial}(0.4) \\x_{1,4}|x_{1,3} = \text{true}, x_{1,2} = 6, x_{1,1} = 0.4 &\sim \text{Poisson}(7) \\y_1 = 18|x_{1,4} = 7, x_{1,3} = \text{true}, x_{1,2} = 6, x_{1,1} = 0.4 &\sim \text{Normal}(14, 1) \\x_{2,1}|\dots &\sim \text{Gamma}(0.4, 7) \\y_2 = 6|x_{2,1} = 6.92, \dots &\sim \text{Normal}(0, 6.92)\end{aligned}$$

```
[assume poi-1 (beta 7 4)]
[assume poi-2 (+ 1 (poisson 8))]
[assume generative-model-part-1 (lambda (a b)
                                         (if (flip a)
                                             (+ (poisson b) 7)
                                             (normal a b)))]
[assume generative-model-part-2 (lambda () (gamma poi-1 poi-2))]
[observe (normal (generative-model-part-1 poi-1 poi-2) 1) 18]
[observe (normal 0 (generative-model-part-2)) 6]
[predict (list poi-1 poi-2)]
```

# What is $\mathbf{x}_{1:2}$ ?

$$\begin{aligned}
 x_{1,1} &\sim \text{Beta}(7, 4) \\
 x_{1,2}|x_{1,1} = 0.4 &\sim \text{Poisson}(8) \\
 x_{1,3}|x_{1,2} = 6, x_{1,1} = 0.4 &\sim \text{Binomial}(0.4) \\
 x_{1,4}|x_{1,3} = \text{true}, x_{1,2} = 6, x_{1,1} = 0.4 &\sim \text{Poisson}(7) \\
 y_1 = 18|x_{1,4} = 7, x_{1,3} = \text{true}, x_{1,2} = 6, x_{1,1} = 0.4 &\sim \text{Normal}(14, 1) \\
 x_{2,1}|\dots &\sim \text{Gamma}(0.4, 7) \\
 y_2 = 6|x_{2,1} = 6.92, \dots &\sim \text{Normal}(0, 6.92)
 \end{aligned}$$

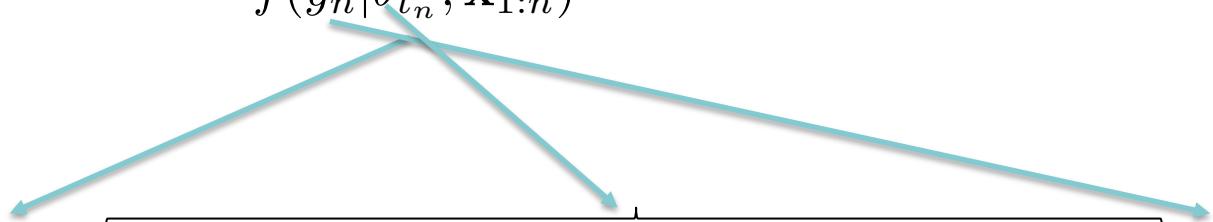
```

[assume poi-1 (beta 7 4)]
[assume poi-2 (+ 1 (poisson 8))]
[assume generative-model-part-1 (lambda (a b)
                                         (if (flip a)
                                             (+ (poisson b) 7)
                                             (normal a b)))]
[assume generative-model-part-2 (lambda () (gamma poi-1 poi-2))]
[observe (normal (generative-model-part-1 poi-1 poi-2) 1) 18]
[observe (normal 0 (generative-model-part-2)) 6]
[predict (list poi-1 poi-2)]

```

# Observe Statements

$f(y_n | \theta_{t_n}, \mathbf{x}_{1:n})$



```
[observe (normal (generative-model-part-1 poi-1 poi-2) 1) 18]
```



# *Original* Church : Rejection Sampling

- Run program forward and condition on all observations exactly matching the observed output

```
1 (define (rejection-query thunk condition)
2   (let ((val (thunk)))
3     (if (condition val)
4         val
5         (rejection-query thunk condition))))
```



# Review: Rejection Sampling

Assume

- Want sample from  $p(\mathbf{x})$
- $p(\mathbf{x})$  is easy to evaluate, but only up to an unknown normalising constant, i.e.

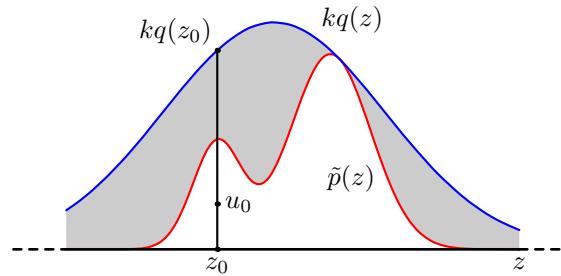
$$p(\mathbf{x}) = \frac{1}{Z_p} \tilde{p}(\mathbf{x})$$

- A *proposal distribution*  $q(\mathbf{x})$  s.t.  $kq(\mathbf{x}) \geq \tilde{p}(\mathbf{x})$  for all  $\mathbf{x}$  can be designed

Note  $\mathbf{x}$  is, in general, a vector of random variables.



# Rejection Sampling



Sampling  $\mathbf{x}^{(\tau)} \sim q$  and  $u^{(\tau)} \sim \text{Uniform}(0, kq(\mathbf{x}^{(\tau)}))$  yields a pair of values uniformly distributed in the gray region.

If  $u_0 \leq \tilde{p}(\mathbf{x})$  then  $\mathbf{x}^{(\tau)}$  is accepted, otherwise it is rejected and the process repeats until a sample is accepted.

Accepted pairs are uniformly distributed in the white area; dropping  $u^{(\tau)}$  yields a sample distributed according to  $\tilde{p}(\mathbf{x})$ , and equivalently,  $p(\mathbf{x})$ .



# Rejection Sampling

Assume we have a model  $p(\mathbf{x})$ , some variables of which are known, some of which are not. Also let  $\mathbf{x}_{\text{obs}}$  be the “observed” variables and  $\mathbf{x}_{\text{lat}}$  be latent variables such that  $\mathbf{x}_{\text{obs}} \cup \mathbf{x}_{\text{lat}} = \mathbf{x}$ .

We would like samples from  $p(\mathbf{x}_{\text{lat}} | \mathbf{x}_{\text{obs}}) = \frac{p(\mathbf{x})}{p(\mathbf{x}_{\text{obs}})}$

Equivalently we can write the conditional distribution of interest as an unnormalised distribution  $\tilde{p}(\mathbf{x}_{\text{lat}} | \mathbf{x}_{\text{obs}}) = p(\mathbf{x}) \mathbb{I}[\mathbf{x}_{\text{obs}} = \mathbf{v}]$  using an indicator function that imposes the constraint that the observed variables are constrained to take values  $\mathbf{v}$ .

Rejection sampling with  $q(\mathbf{x}) = p(\mathbf{x})$  (i.e. proposing via ancestral sampling of the joint) can be used to generate samples distributed according to  $\tilde{p}(\mathbf{x}_{\text{lat}} | \mathbf{x}_{\text{obs}})$ . Note that  $q(\mathbf{x}) \geq \tilde{p}(\mathbf{x}_{\text{lat}} | \mathbf{x}_{\text{obs}}) \forall \mathbf{x}$  by construction.



# Rejection Sampling

## Conditioning via Rejection and Ancestral Sampling

- ① Sample  $\mathbf{x}^{(\tau)} \sim q(\mathbf{x})$  (i.e. generate via ancestral sampling)
- ② Sample  $u^{(\tau)} \sim U(0, q(\mathbf{x}))$
- ③ Accept  $\mathbf{x}^{(\tau)}$  only if  $u^{(\tau)} \leq p(\mathbf{x})\mathbb{I}[\mathbf{x}_{\text{obs}} = \mathbf{v}]$
- ④ Repeat

A sample will only ever be accepted when  $\mathbf{x}_{\text{obs}} = \mathbf{v}$  and then it will always be because  $q(\mathbf{x}) = p(\mathbf{x})$

Unless the prior and posterior are extremely well matched this will be an extremely inefficient sampler.



# *Original* Church : Rejection Sampling

- Run program forward and condition on all observations exactly matching the observed output

```
1 (define (rejection-query thunk condition)
2   (let ((val (thunk)))
3     (if (condition val)
4         val
5         (rejection-query thunk condition))))
```



# New Church : Single-Site Independent MH

Sample posterior distribution of execution traces using joint with observed values plugged in

$$p(\mathbf{x}|\mathbf{y}) \propto \tilde{p}(\mathbf{y} = \text{observes}, \mathbf{x})$$

Metropolis-Hastings acceptance rule

$$\min \left( 1, \frac{p(\mathbf{y}|\mathbf{x}') p(\mathbf{x}') q(\mathbf{x}|\mathbf{x}')} {p(\mathbf{y}|\mathbf{x}) p(\mathbf{x}) q(\mathbf{x}'|\mathbf{x})} \right)$$

Need

Proposal

Have

Likelihoods (via observe statement restrictions)

Prior (sequence of ERP returns; scored in interpreter)



Wingate, Stuhlmüller et al Lightweight Implementations of Probabilistic Programming Languages Via Transformational Compilation, 2011  
Wood, van de Meent, and Mansinghka “A New Approach to Probabilistic Programming Inference” AISTATS 2014  
Mansinghka, Selsam, and Perov “Venture: an interactive, Turing-complete probabilistic programming platform” arXiv 2014

# Review : Metropolis Hastings

## Algorithm

Initialize  $\tau \leftarrow 1, \mathbf{x}^{(\tau)} \leftarrow ?$

Repeat Forever Yielding  $\{\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \dots\}$

- ① Propose  $\mathbf{x}^* \sim q(\mathbf{x}^* | \mathbf{x}^{(\tau)})$
- ② Accept  $\mathbf{x}^*$  w.p.  $A(\mathbf{x}^*, \mathbf{x}^{(\tau)}) = \min \left( 1, \frac{p(\mathbf{x}^*)q(\mathbf{x}^{(\tau)}|\mathbf{x}^*)}{p(\mathbf{x}^{(\tau)})q(\mathbf{x}^*|\mathbf{x}^{(\tau)})} \right)$
- ③ If  $\mathbf{x}^*$  accepted set  $\mathbf{x}^{(\tau+1)} \leftarrow \mathbf{x}^*$  else  $\mathbf{x}^{(\tau+1)} \leftarrow \mathbf{x}^{(\tau)}$
- ④ Increment  $\tau$

Common choices of proposal include  $q(\mathbf{x}^* | \mathbf{x}^{(\tau)}) = \mathcal{N}(\mathbf{x}^{(\tau)} | \sigma^2 \mathbf{I})$  (random-walk Metropolis) and/or  $q(\mathbf{x}^* | \mathbf{x}^{(\tau)}) = q(\mathbf{x}^*)$  (independent MH). Rules of thumb suggest aiming for acceptance rates of between 25% and 50% by tuning the proposal distribution.



# Random Database (RDB) MH Proposal

$$q(\mathbf{x}'|\mathbf{x}) = \frac{\kappa(x'_{m,j}|x_{m,j})}{|\mathbf{x}|} \frac{p(\mathbf{x}' \setminus \mathbf{x} | \mathbf{x}' \cap \mathbf{x})}{p(x'_{m,j} | \mathbf{x}' \cap \mathbf{x})}$$

Single stochastic procedure (SP) output

Probability of new part of proposed execution trace

Number of SP's in original trace

Probability of new SP return value (sample) given trace prefix



# RDB Implementation

Single site update = sample from the prior = run program forward

$$\kappa(x'_{m,j} | x_{m,j}) = p(x'_{m,j} | \mathbf{x}' \cap \mathbf{x})$$

MH acceptance ratio simplifies

$$\frac{p(\mathbf{y}|\mathbf{x}') p(\mathbf{x}') |\mathbf{x}| p(\mathbf{x} \setminus \mathbf{x}' | \mathbf{x} \cap \mathbf{x}')} {p(\mathbf{y}|\mathbf{x}) p(\mathbf{x}) |\mathbf{x}'| p(\mathbf{x}' \setminus \mathbf{x} | \mathbf{x}' \cap \mathbf{x})}$$

Number of SP applications in original trace      Probability of regenerating current trace continuation given proposal trace beginning

Number of SP applications in new trace      Probability of generating proposal trace continuation given current trace beginning



# RDB Implementation Sketch

$$\begin{aligned}x_{1,1} &\sim \text{Beta}(7, 4) \\x_{1,2}|x_{1,1} = 0.4 &\sim \text{Poisson}(8) \\x_{1,3}|x_{1,2} = 6, x_{1,1} = 0.4 &\sim \text{Binomial}(0.4) \\x_{1,4}|x_{1,3} = \text{false}, x_{1,2} = 6, x_{1,1} = 0.4 &\sim \text{Poisson}(7) \\y_1 = 18|x_{1,4} = 7, x_{1,3} = \text{true}, x_{1,2} = 6, x_{1,1} = 0.4 &\sim \text{Normal}(14, 1) \\x_{2,1} | \dots &\sim \text{Gamma}(0.4, 7) \\y_2 = 6|x_{2,1} = 6.92, \dots &\sim \text{Normal}(0, 6.92)\end{aligned}$$

```
[assume poi-1 (beta 7 4)]
[assume poi-2 (+ 1 (poisson 8))]
[assume generative-model-part-1 (lambda (a b)
                                         (if (flip a)
                                             (+ (poisson b) 7)
                                             (normal a b)))]
[assume generative-model-part-2 (lambda () (gamma poi-1 poi-2))]
[observe (normal (generative-model-part-1 poi-1 poi-2) 1) 18]
[observe (normal 0 (generative-model-part-2)) 6]
[predict (list poi-1 poi-2)]
```

# SMC for Prob. Prog. Inference

State-space-model-like decomposition

$$p(\mathbf{x}_{1:n} | y_{1:n}) = g(y_n | \mathbf{x}_{1:n}) f(\mathbf{x}_n | \mathbf{x}_{1:n-1}) p(\mathbf{x}_{1:n-1} | y_{1:n-1})$$

Suggests Sequential Importance Resampling (SIR)

$$\frac{p(\mathbf{x}_{1:n} | y_{1:n})}{q(\mathbf{x}_{1:n} | y_{1:n})} = \frac{g(y_n | \mathbf{x}_{1:n}) f(\mathbf{x}_n | \mathbf{x}_{1:n-1}) p(\mathbf{x}_{1:n-1} | y_{1:n-1})}{f(\mathbf{x}_n | \mathbf{x}_{1:n-1}) p(\mathbf{x}_{1:n-1} | y_{1:n-1})} = g(y_n | \mathbf{x}_{1:n})$$

Proposal

Run program forward  
until next observe directive

Weight of particle  
Is observation likelihood



Fischer, Kiselyov, and Shan “Purely functional lazy non-deterministic programming” ACM Sigplan 2009  
Wood, van de Meent, and Mansinghka “A New Approach to Probabilistic Programming Inference” AISTATS 2014  
Paige and Wood “A Compilation Target for Probabilistic Programming Languages” ICML 2014

# Review : Sequential Importance Resampling

SIR Targets

$$p(\mathbf{x}_{1:N} | \mathbf{y}_{1:N}) \propto \tilde{p}(\mathbf{y}_{1:N}, \mathbf{x}_{1:N}) \equiv \prod_{n=1}^N g(y_n | \mathbf{x}_{1:n}) f(\mathbf{x}_n | \mathbf{x}_{1:n-1})$$

With a weighted set of particles

$$p(\mathbf{x}_{1:N} | y_{1:N}) \approx \sum_{\ell=1}^L w_N^\ell \delta_{\mathbf{x}_{1:N}^\ell}(\mathbf{x}_{1:N})$$

Noting the identity

$$p(\mathbf{x}_{1:n} | y_{1:n}) = g(y_n | \mathbf{x}_{1:n}) f(\mathbf{x}_n | \mathbf{x}_{1:n-1}) p(\mathbf{x}_{1:n-1} | y_{1:n-1})$$

We can use importance sampling to generate samples from

$$p(\mathbf{x}_{1:n} | y_{1:n})$$

Given our sample-based approximation to

$$p(\mathbf{x}_{1:n-1} | y_{1:n-1})$$



# Review : Importance Sampling

$$\begin{aligned} E_{p(\mathbf{x}|\mathbf{y})}[h(\mathbf{x})] &= \int h(\mathbf{x})p(\mathbf{x}|\mathbf{y})d\mathbf{x} \\ &= \int h(\mathbf{x}) \frac{p(\mathbf{x}|\mathbf{y})}{q(\mathbf{x}|\mathbf{y})} q(\mathbf{x}|\mathbf{y})d\mathbf{x} \\ &\approx \frac{1}{L} \sum_{\ell=1}^L h(\mathbf{x}^\ell) \frac{p(\mathbf{x}^\ell|\mathbf{y})}{q(\mathbf{x}^\ell|\mathbf{y})} & x^\ell \sim q(\mathbf{x}|\mathbf{y}) \\ &= \frac{1}{L} \sum_{\ell=1}^L h(\mathbf{x}^\ell) w_n^\ell & x^\ell \sim q(\mathbf{x}|\mathbf{y}), w_n^\ell = \frac{p(\mathbf{x}^\ell|\mathbf{y})}{q(\mathbf{x}^\ell|\mathbf{y})} \end{aligned}$$



# Review: Sequential Importance Resampling

$$p(\mathbf{x}_{1:n-1} | \mathbf{y}_{1:n-1}) \approx \sum_{\ell=1}^L w_{n-1}^\ell \delta_{\mathbf{x}_{1:n-1}^\ell}(\mathbf{x}_{1:n-1})$$

$$p(\mathbf{x}_{1:n} | y_{1:n}) = g(y_n | \mathbf{x}_{1:n}) f(\mathbf{x}_n | \mathbf{x}_{1:n-1}) p(\mathbf{x}_{1:n-1} | y_{1:n-1})$$

$$q(\mathbf{x}_{1:n} | y_{1:n}) = f(\mathbf{x}_n | \mathbf{x}_{1:n-1}) p(\mathbf{x}_{1:n-1} | y_{1:n-1})$$

$$p(\mathbf{x}_{1:n} | y_{1:n}) \approx \sum_{\ell=1}^L g(y_n | \mathbf{x}_{1:n}^\ell) \delta_{\mathbf{x}_{1:n}^\ell}(\mathbf{x}_{1:n}), \quad \mathbf{x}_{1:n}^\ell = \mathbf{x}_n^\ell \mathbf{x}_{1:n-1}^{a_{n-1}^\ell} \sim f$$



# SMC Methods Discussed Require Only

## Initialization

$p(\mathbf{x}_1)$  can be sampled

## Forward Simulation

$f(\mathbf{x}_n | \mathbf{x}_{1:n-1})$  can be sampled (blackbox)

## Observation Likelihood Weight Computation

$g(y_n | \mathbf{x}_{1:n})$  can be point-wise evaluated up to constant multiple



# Sequential Monte Carlo for Prob. Prog.

---

**Algorithm 1** Parallel SMC program execution

---

**Assume:**  $N$  observations,  $L$  particles

```
    launch  $L$  copies of the program           (parallel)
    for  $n = 1 \dots N$  do
        wait until all  $L$  reach observe  $y_n$       (barrier)
        update unnormalized weights  $\tilde{w}_n^{1:L}$    (serial)
        if  $ESS < \tau$  then
            sample number of offspring  $O_n^{1:L}$      (serial)
            set weight  $\tilde{w}_n^{1:L} = 1$              (serial)
            for  $\ell = 1 \dots L$  do
                fork or exit                         (parallel)
            end for
        else
            set all number of offspring  $O_n^\ell = 1$  (serial)
        end if
        continue program execution             (parallel)
    end for
    wait until  $L$  program traces terminate   (barrier)
    predict from  $L$  samples from  $\hat{p}(\mathbf{x}_{1:N}^{1:L} | y_{1:N})$  (serial)
```

---



# Probabilistic-C

```
#include "probabilistic.h"
#define K 3
#define N 11

/* Markov transition matrix */
static double T[K][K] = { { 0.1, 0.5, 0.4 },
                        { 0.2, 0.2, 0.6 },
                        { 0.15, 0.15, 0.7 } };

/* Observed data */
static double data[N] = { NAN, .9, .8, .7, 0, -.025,
                         -5, -2, -.1, 0, 0.13 };

/* Prior distribution on initial state */
static double initial_state[K] = { 1.0/3, 1.0/3, 1.0/3 };

/* Per-state mean of Gaussian emission distribution */
static double state_mean[K] = { -1, 1, 0 };

/* Generative program for a HMM */
int main(int argc, char **argv) {

    int states[N];
    for (int n=0; n<N; n++) {
        states[n] = (n==0) ? discrete_rng(initial_state, K)
                           : discrete_rng(T[states[n-1]], K);
        if (n > 0) {
            observe(normal_lnp(data[n], state_mean[states[n]], 1));
        }
        predict("state[%d],%d\n", n, states[n]);
    }

    return 0;
}
```



# Inverse Stochastic Simulation

- Deterministic simulator exists as code
- Parameter uncertainties exist
  - Varying parameters to simulator = stochastic simulator
- What to do with observations?
  - Update estimates of parameters
  - Posterior predictions



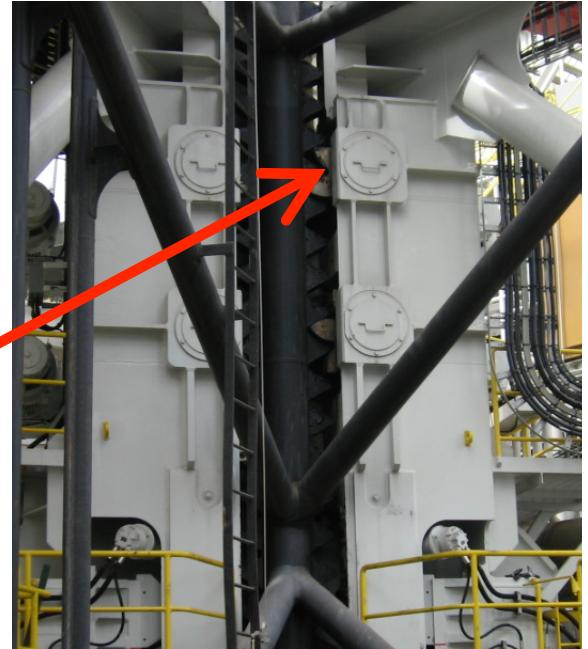


## Example : Jack-Up Units

60m



Maersk



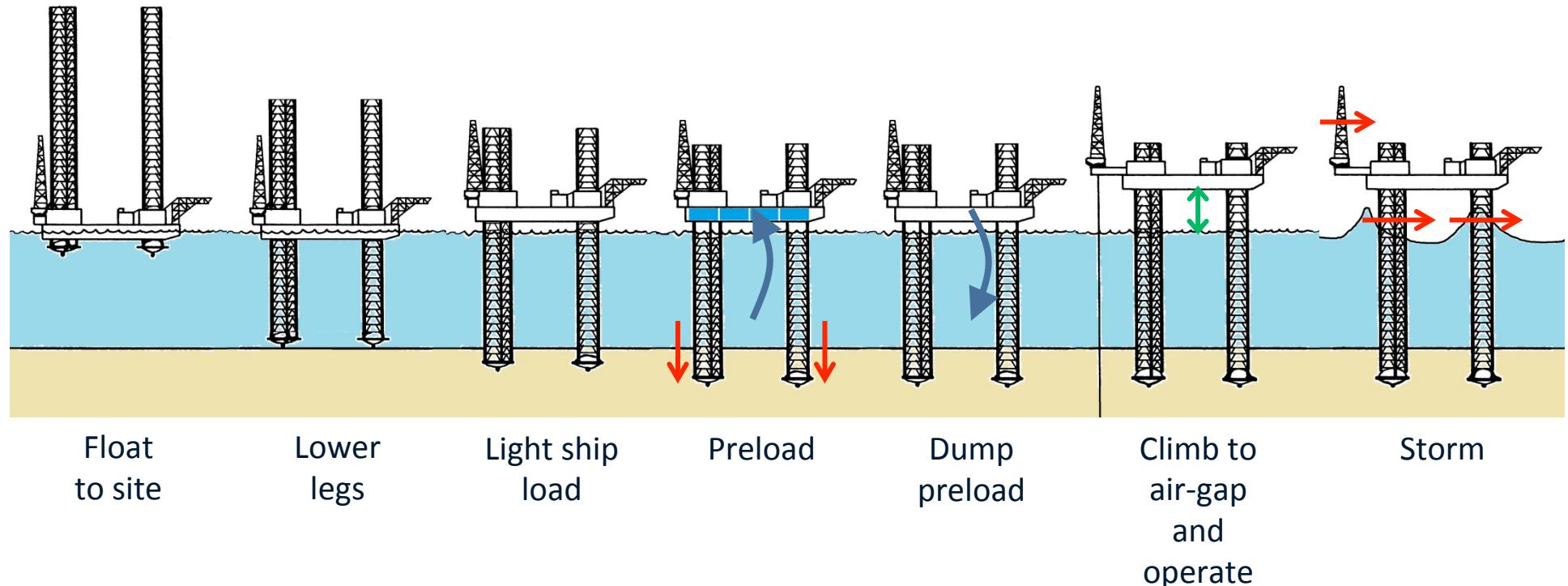
Keppel FELS



Keppel FELS



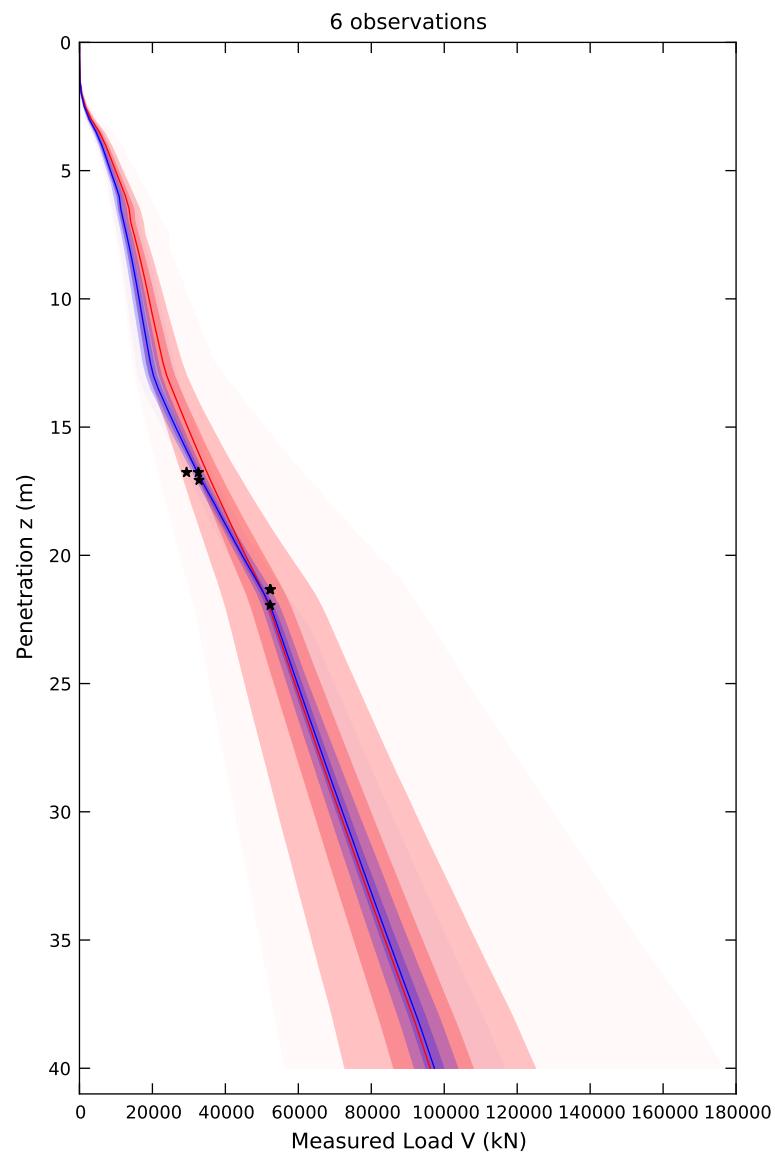
# Jack-up operations



sketches after Poulos (1988)  
Slide from Housby

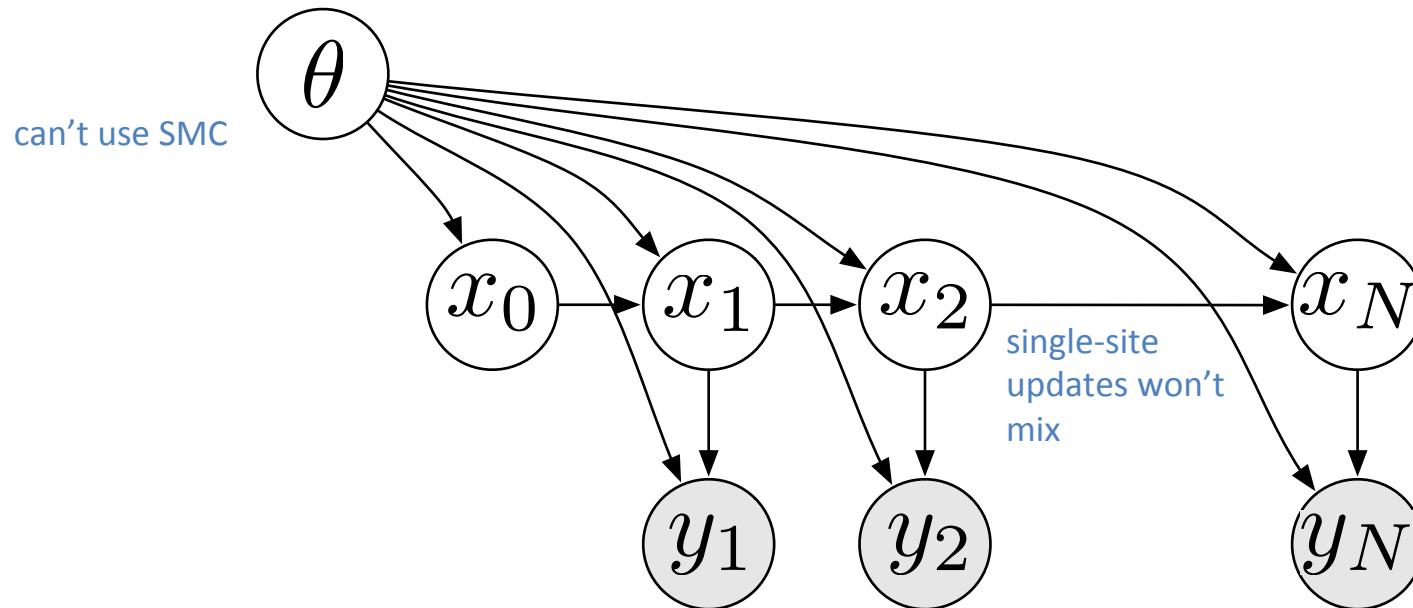
# Spudcan Simulator + Probabilistic-C -> Inference

- Deterministic simulator
  - ~750 lines of C code
  - 10-100's of parameters
  - Black-box
  - Not differentiable
- Stochastic simulator
  - +150 lines of C code
  - Priors on parameters
- Automatic inference
  - +15 lines of Probabilistic-C
- ~1000 samples / second



# Review : Inference In State Space Models

Consider inference in a state space model that depends on fixed parameters

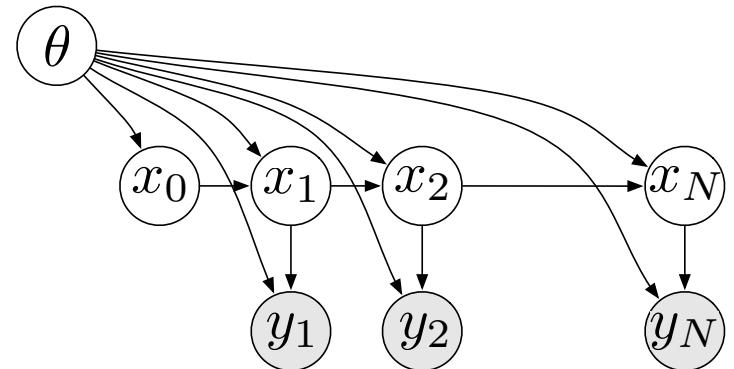


# “Ideal” Inference

Ideal MH

$$\min \left( 1, \frac{p(y_{1:N}|\theta') p(\theta') q(\theta|\theta')}{p(y_{1:N}|\theta) p(\theta) q(\theta'|\theta)} \right)$$

intractable.



SMC provides *unbiased* estimate

$$\hat{Z} = p(y_{1:N}|\theta) \approx \prod_{n=1}^N \left[ \frac{1}{N} \sum_{\ell=1}^L w_n^\ell \right]$$

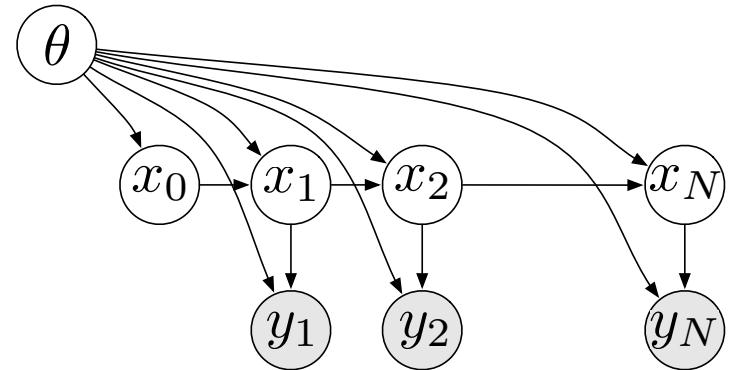


# Particle Marginal Metropolis Hastings

MH with unbiased likelihood estimates

$$\min \left( 1, \frac{\hat{Z}' p(\theta') q(\theta | \theta')}{\hat{Z} p(\theta) q(\theta' | \theta)} \right)$$

computed via SMC proposal



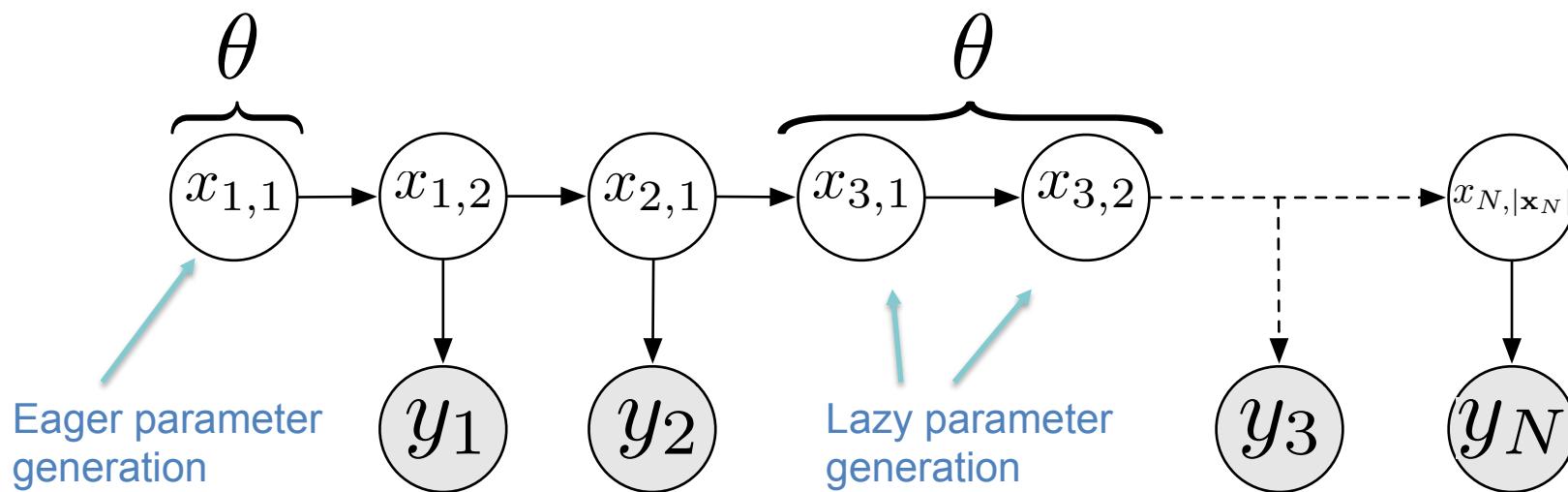
$$\hat{Z} \equiv p(y_{1:N} | \theta) \approx \prod_{n=1}^N \left[ \frac{1}{N} \sum_{\ell=1}^L w_n^\ell \right]$$

targets correct distribution!



# Conditional SMC for Prob. Prog. Inference

- No fixed parameter
- Program generates all random variables



- State is interpreter memory state
- Transition is stochastic procedure application
- Only observes need be indexed



# PIMH For Probabilistic Programming

- Run SMC Once
- Compute marginal likelihood estimate

$$\hat{Z} \equiv p(y_{1:N}) \approx \prod_{n=1}^N \left[ \frac{1}{N} \sum_{\ell=1}^L w_n^\ell \right]$$

No theta!

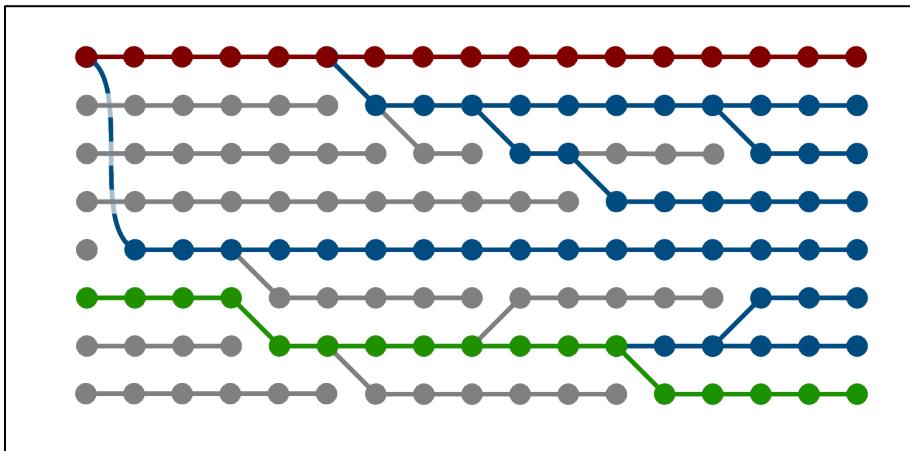
- Do forever
  - Re-run SMC
  - Compute new marginal likelihood estimate  
 $\hat{Z}'$
  - Accept particle set with probability  
 $\min(1, \hat{Z}' / \hat{Z})$
  - Emit predictions from all particles in next set (new and/or old)



# Particle Gibbs for Prob. Prog.

- MH w/ accept prob. = 1
- SMC “inner-loop” proposal
- “Retained particle”
- *Non-local*
  - Single “sweep” can propose changes to many variable values at once

[Holenstein 2009; Andrieu, Doucet, Holenstein 2010; etc]



**Algorithm 1** Particle Gibbs Prob. Prog. Inference

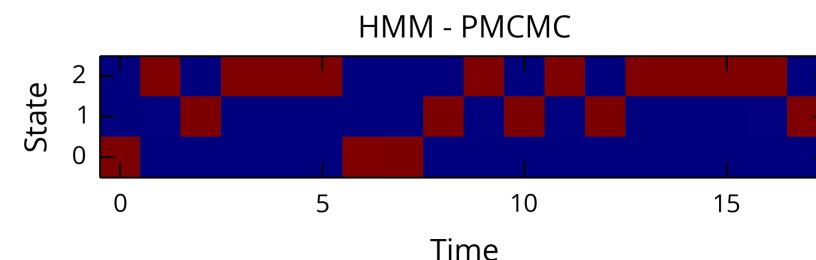
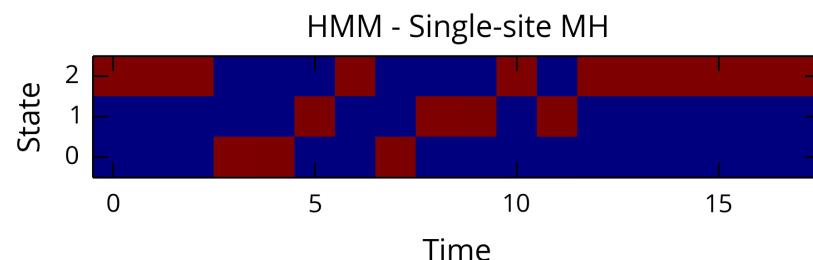
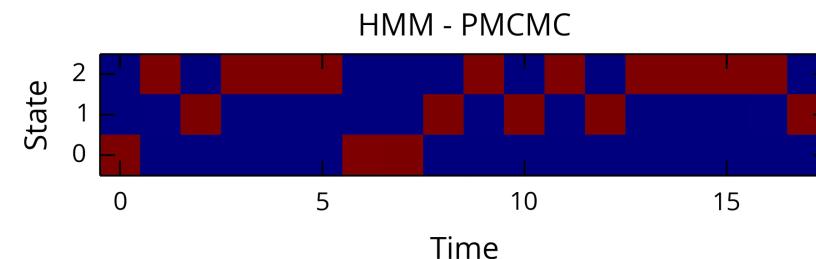
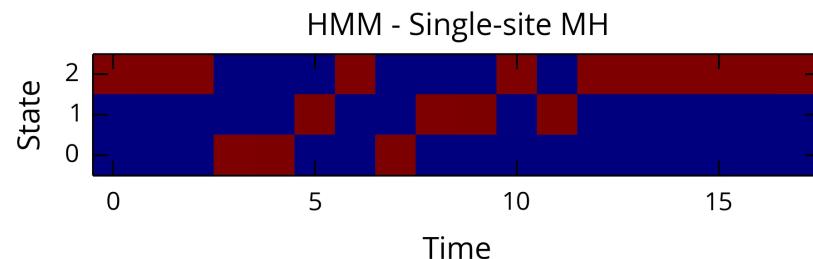
```

 $L \leftarrow$  number of particles
 $S \leftarrow$  number of sweeps
 $\{\tilde{w}_N^{(\ell)}, \mathbf{x}_{1:N}^{(\ell)}\} \leftarrow$  Run SMC
for  $s < S$  do
     $\{\cdot, \mathbf{x}_{1:N}^*\} \leftarrow \mathbf{r}(1, \{1/L, \mathbf{x}_{1:N}^{(\ell)}\})$ 
     $\{\cdot, \mathbf{x}_0^{(\ell)} = \emptyset\} \leftarrow$  initialize  $L - 1$  interpreters
    for  $d \in$  ordered lines of program do
        for  $\ell < L - 1$  do
             $\bar{\mathbf{x}}_{1:(n-1)}^{(\ell)} \leftarrow \text{fork}(\mathbf{x}_{1:(n-1)}^{(\ell)})$ 
        end for
        if directive( $d$ ) == assume then
            for  $\ell < L - 1$  do
                 $\bar{\mathbf{x}}_{1:n}^{(\ell)} \leftarrow \text{interpret}(d, \bar{\mathbf{x}}_{1:(n-1)}^{(\ell)})$ 
            end for
             $\{\mathbf{x}_{1:n}^{(\ell)}\} \leftarrow \{\bar{\mathbf{x}}_{1:n}^{(\ell)}\} \cup \mathbf{x}_{1:n}^*$ 
        else if directive( $d$ ) == predict then
            for  $\ell < L - 1$  do
                 $\text{interpret}(d, \bar{\mathbf{x}}_{1:(n-1)}^{(\ell)})$ 
            end for
             $\text{interpret}(d, \mathbf{x}_{1:(n-1)}^*)$ 
        else if directive( $d$ ) == observe then
            for  $\ell < L - 1$  do
                 $\{\bar{w}_n^{(\ell)}, \bar{\mathbf{x}}_{1:n}^{(\ell)}\} \leftarrow \text{interpret}(d, \bar{\mathbf{x}}_{1:(n-1)}^{(\ell)})$ 
            end for
             $\mathcal{T} \leftarrow \mathbf{r}(L - 1, \{\bar{w}_N^{(\ell)}, \bar{\mathbf{x}}_{1:n}^{(\ell)}\} \cup \{\tilde{w}_{1:n}^*, \mathbf{x}_{1:n}^*\})$ 
             $\{\tilde{w}_n^{(\ell)}, \mathbf{x}_{1:n}^{(\ell)}\} \leftarrow \mathcal{T} \cup \{\tilde{w}_n^*, \mathbf{x}_{1:n}^*\}$ 
        end if
    end for
end for
end for

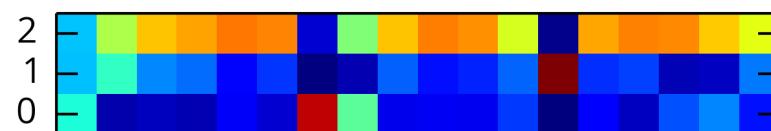
```



# PMCMC Prob. Prog. Example



forward-backward



# Goals and Aims

## (i) Accelerate iteration over models

- Inference is automatic
- Writing generative code is easier than deriving model inverses
- Lower technical barrier of entry to development of new models

## (ii) Accelerate iteration over inference procedures

- Computer language is an abstraction barrier
  - Inference procedures can be tested against a library of models
  - Inference procedures become “compiler optimizations”

## (iii) Enable development of more expressive models

- Probabilistic programs can express a superset of graphical models
- Modern machine learning models are tens of lines of code

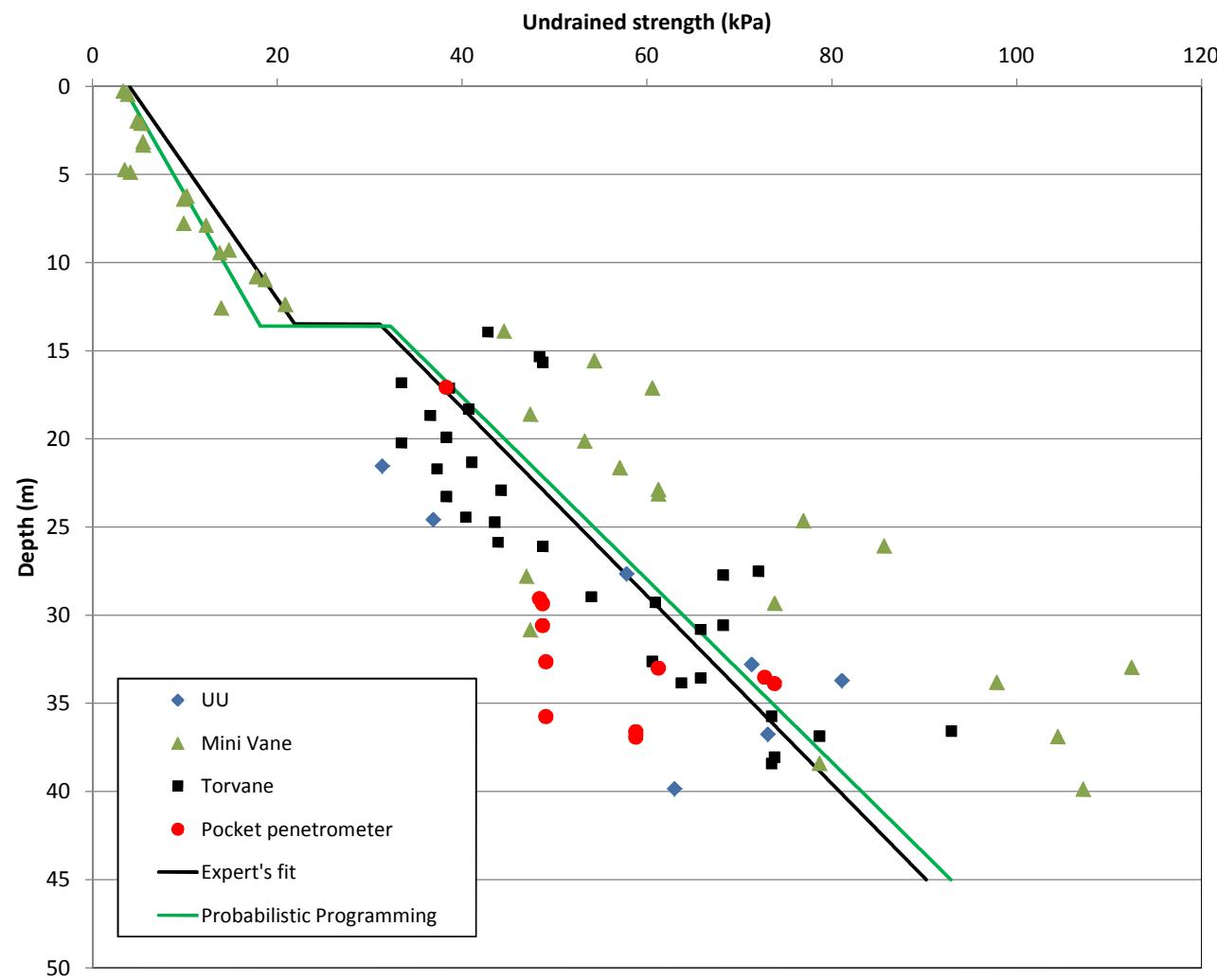


# Wrap-Up

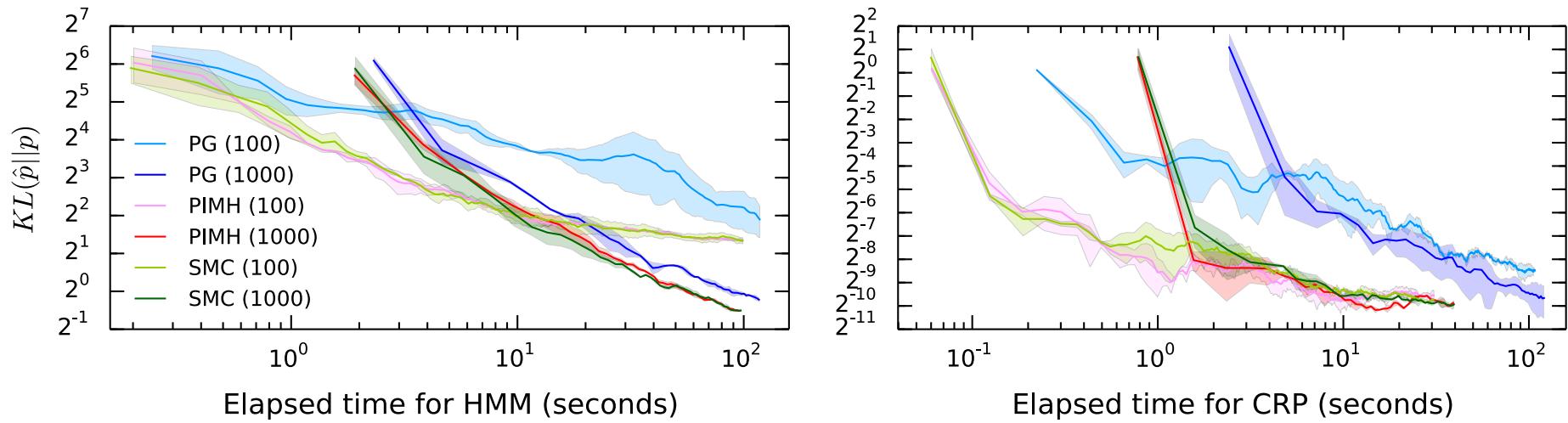
- Research
  - New paths to efficient, scalable probabilistic programming inference
  - True hope for general purpose automatic inference
  - New models (soon)
- Resources
  - <http://www.robots.ox.ac.uk/~fwood/anglican/>
  - <http://www.robots.ox.ac.uk/~brooks/probabilistic-c/>
  - <http://probabilistic-programming.org/wiki/Home>
  - <http://forestdb.org/>



# Parameter Posterior vs. Expert



# Compiled PMCMC Algorithm Performance



# What if dirac Observes?

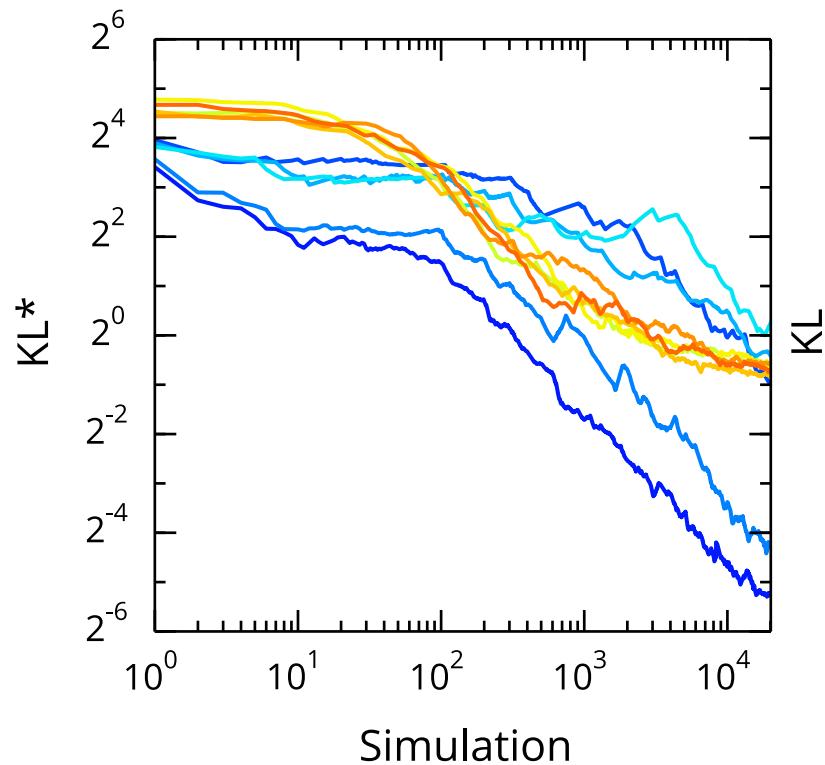
dirac observes are constraints

$$\begin{aligned} p(\mathbf{x}_{1:N} | \mathbf{y}_{1:N}) \propto \tilde{p}(\mathbf{y}_{1:N}, \mathbf{x}_{1:N}) &= \prod_{n=1}^N \mathbb{I}[y_n = a_n(\mathbf{x}_{1:n})] f(\mathbf{x}_n | \mathbf{x}_{1:n-1}) \\ &= p(\mathbf{x}_{1:N}) \mathbb{I}[\mathbf{y}_{1:N} = \mathbf{a}_{1:N}(\mathbf{x}_{1:N})] \end{aligned}$$

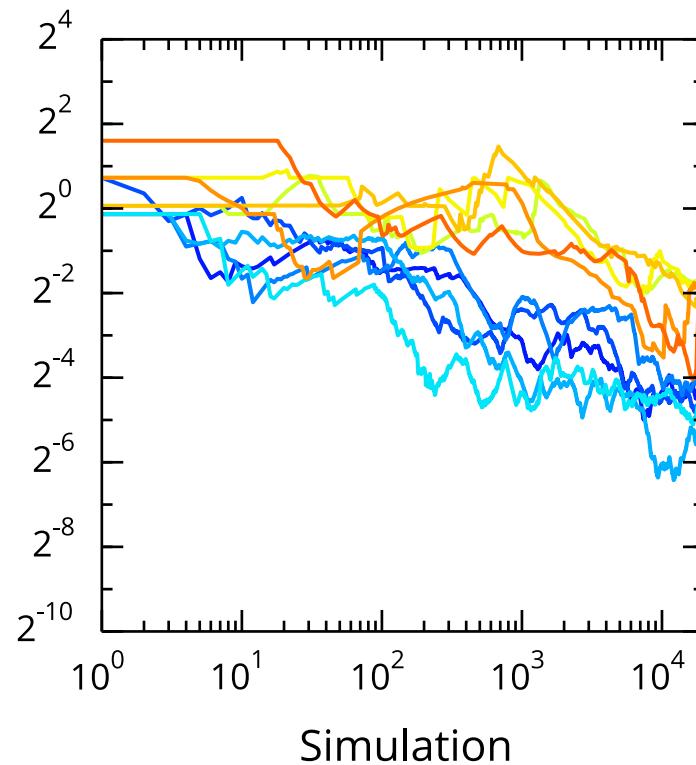
=> SMC / PMCMC reduce to rejection / repeated rejection sampling  
if all observes are constraints



# Opportunity : Optimizing Inference by Program Line Reordering



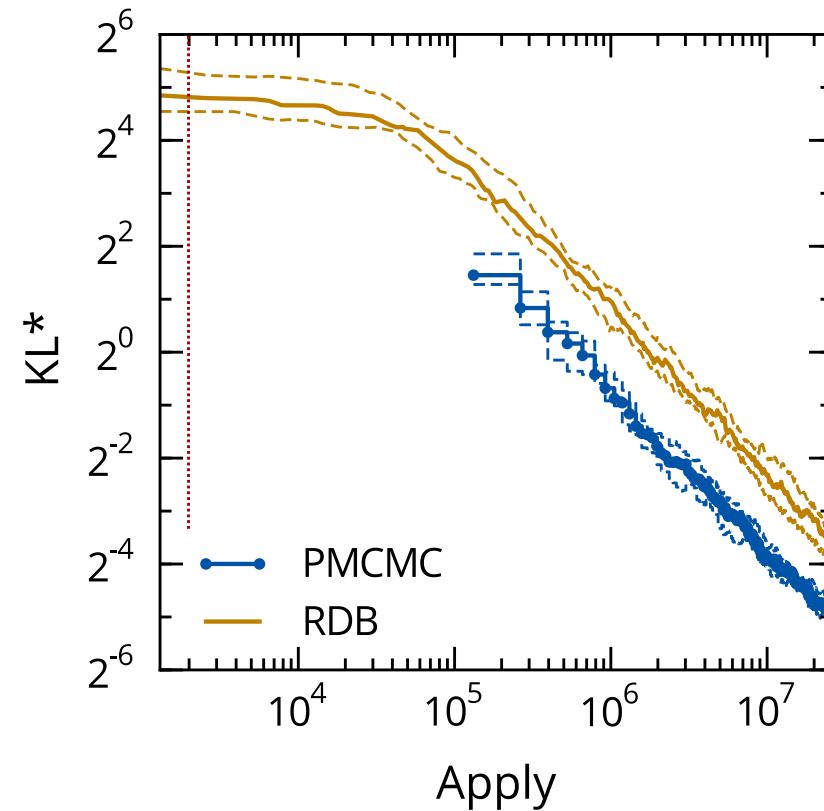
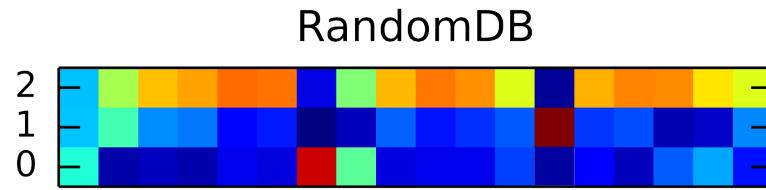
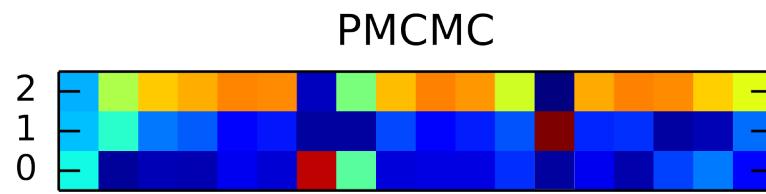
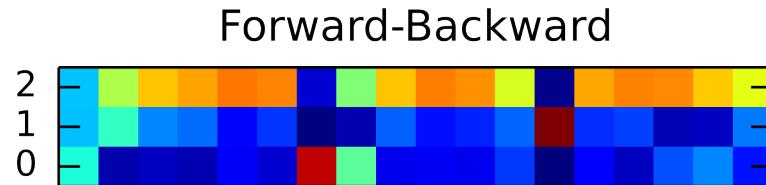
(a) HMM



(b) DP Mixture



# Anglican : Particle MCMC Inference



Wood, van de Meent, Mansinghka "A new approach to probabilistic programming inference." AISTATS, 2014

Wingate et al "Lightweight implementations of probabilistic programming languages via transformational compilation" AISTATS, 2011

