UPPSALA
UNIVERSITET

Sequential Monte Carlo Methods

Lecture 17 – SMC for Probabilistic Programs

Lawrence Murray, Uppsala University

2017-08-29

Outline – Lecture 17

Aim:

- Introduce probabilistic programming as a modeling paradigm.
- Demonstrate SMC as an appropriate inference method.

Outline:

1. Probabilistic programs.
2. Some examples in Birch.
3. SMC for probabilistic programs.

Probabilistic programs

Programs as stochastic processes

- Consider a program that depends on random numbers.
- Execute that program on a processor.
- As it runs, its memory state evolves **dynamically** and **stochastically** in time.

Programs as stochastic processes

- Consider a program that depends on random numbers.
- Execute that program on a processor.
- As it runs, its memory state evolves **dynamically** and **stochastically** in time.

We can think of the running program as a **stochastic process**.

Programs as stochastic processes

- Let $k = 1, 2, \ldots$ denote a sequence of **checkpoints**.
- Let $(x_{1:k})_{k \geq 1}$ denote the (memory) state of the running program at checkpoint k, where $x_{1:k} \in \mathcal{X}_{1:k}$ and $\mathcal{X}_{1:k} = \mathcal{X}_k \times \mathcal{X}_{1:k-1}$.
- The state transitions according to $p_k(x_k \mid x_{1:k-1})$.

Programs as stochastic processes

- Let $k = 1, 2, \ldots$ denote a sequence of **checkpoints**.
- Let $(x_{1:k})_{k \geq 1}$ denote the (memory) state of the running program at checkpoint k, where $x_{1:k} \in \mathcal{X}_{1:k}$ and $\mathcal{X}_{1:k} = \mathcal{X}_k \times \mathcal{X}_{1:k-1}$.
- The state transitions according to $p_k(x_k \mid x_{1:k-1})$.

At each checkpoint we can manipulate the running program: pause execution, inspect memory state, consider distributions over that state, modify that state. This is what facilitates inference.

What is probabilistic programming?

Probabilistic programming is a programming paradigm that emphasises this perspective on programs.

Consider other programming paradigms that emphasise other perspectives: functional, imperative, object-oriented, aspect-oriented.

What is a probabilistic programming language?

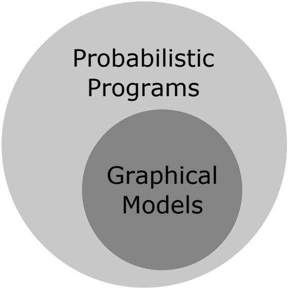
A **probabilistic programming language** (PPL) is a programming language that provides ergonomic support for the probabilistic programming paradigm.

A PPL may provide, for example:

- A library of probability distributions with the ability to evaluate and simulate them.
- Specialised language features for specifying probabilistic models.
- Specialised language features for writing probabilistic inference methods.

What is a probabilistic program?

A **probabilistic program** encodes a **probabilistic model** according to the semantics of a particular **probabilistic programming language**.



Probabilistic programs extend graphical models with support for **stochastic branching**.

Birch

The particular PPL that we will use is Birch, which is currently being developed at Uppsala University.

- It is the successor of LibBi (`www.libbi.org`).
- It is a probabilistic and object-oriented language.
- It compiles down to C++.

Example #1

```
x ~ Gaussian(0.0, 1.0);
y ~ Gaussian(x, 1.0);
z ~ Gaussian(y, 1.0);
```

Example #1

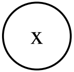
```
x ~ Gaussian(0.0, 1.0);
y ~ Gaussian(x, 1.0);
z ~ Gaussian(y, 1.0);
```

Adopting **operational semantics**, the interpretation of a program is defined by its execution. Here, the program encodes a **joint distribution**.

Example #1

```
x ~ Gaussian(0.0, 1.0);
y ~ Gaussian(x, 1.0);
z ~ Gaussian(y, 1.0);
```

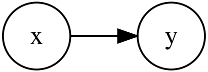
$p(x)$



Example #1

```
x ~ Gaussian(0.0, 1.0);
y ~ Gaussian(x, 1.0);
z ~ Gaussian(y, 1.0);
```

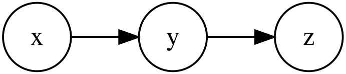
$p(y \mid x)$



Example #1

```
x ~ Gaussian(0.0, 1.0);
y ~ Gaussian(x, 1.0);
z ~ Gaussian(y, 1.0);
```

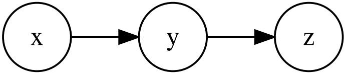
$p(z \mid y)$



Example #1

```
x ~ Gaussian(0.0, 1.0);
y ~ Gaussian(x, 1.0);
z ~ Gaussian(y, 1.0);
```

$p(x)$
$p(y \mid x)$
$p(z \mid y)$



Example #2

```
β ~ Bernoulli(0.5);
x ~ Gaussian(0.0, 1.0);
if (β) {
  y ~ Gaussian(x, 1.0);
} else {
  y ~ Gaussian(0.0, 1.0);
}
```

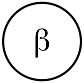
Example #2

```
β ~ Bernoulli(0.5);
x ~ Gaussian(0.0, 1.0);
if (β) {
  y ~ Gaussian(x, 1.0);
} else {
  y ~ Gaussian(0.0, 1.0);
}
```

Example #2

```
β ~ Bernoulli(0.5);
x ~ Gaussian(0.0, 1.0);
if (β) {
  y ~ Gaussian(x, 1.0);
} else {
  y ~ Gaussian(0.0, 1.0);
}
```

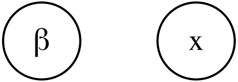
$p(\beta)$



Example #2

```
β ~ Bernoulli(0.5);
x ~ Gaussian(0.0, 1.0);
if (β) {
  y ~ Gaussian(x, 1.0);
} else {
  y ~ Gaussian(0.0, 1.0);
}
```

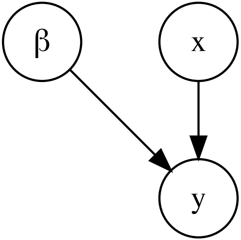
$p(x)$



Example #2

```
β ~ Bernoulli(0.5);
x ~ Gaussian(0.0, 1.0);
if (β) {
  y ~ Gaussian(x, 1.0);
} else {
  y ~ Gaussian(0.0, 1.0);
}
```

$p(y \mid x, \beta)$



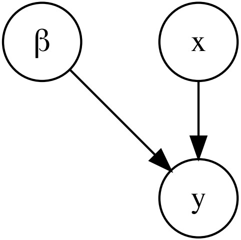
Example #2

```
β ~ Bernoulli(0.5);
x ~ Gaussian(0.0, 1.0);
if (β) {
  y ~ Gaussian(x, 1.0);
} else {
  y ~ Gaussian(0.0, 1.0);
}
```

$p(\beta)$

$p(x)$

$p(y \mid x, \beta)$



Example #3

```
x[1] ~ Gaussian(0.0, 1.0);
y[1] ~ Gaussian(x[1], 1.0);
for (t in 2..T) {
  x[t] ~ Gaussian(a*x[t - 1], 1.0);
  y[t] ~ Gaussian(x[t], 1.0);
}
```

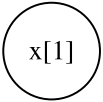
Example #3

```
x[1] ~ Gaussian(0.0, 1.0);
y[1] ~ Gaussian(x[1], 1.0);
for (t in 2..T) {
  x[t] ~ Gaussian(a*x[t - 1], 1.0);
  y[t] ~ Gaussian(x[t], 1.0);
}
```

Example #3

```
x[1] ~ Gaussian(0.0, 1.0);
y[1] ~ Gaussian(x[1], 1.0);
for (t in 2..T) {
  x[t] ~ Gaussian(a*x[t - 1], 1.0);
  y[t] ~ Gaussian(x[t], 1.0);
}
```

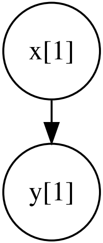
$p(x_1)$



Example #3

```
x[1] ~ Gaussian(0.0, 1.0);
y[1] ~ Gaussian(x[1], 1.0);
for (t in 2..T) {
  x[t] ~ Gaussian(a*x[t - 1], 1.0);
  y[t] ~ Gaussian(x[t], 1.0);
}
```

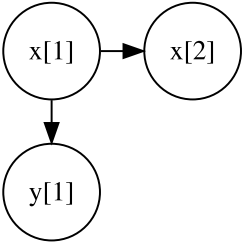
$p(y_1 \mid x_1)$



Example #3

```
x[1] ~ Gaussian(0.0, 1.0);
y[1] ~ Gaussian(x[1], 1.0);
for (t in 2..T) {
  x[t] ~ Gaussian(a*x[t - 1], 1.0);
  y[t] ~ Gaussian(x[t], 1.0);
}
```

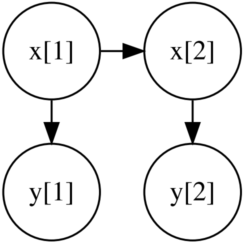
$p(x_t \mid x_{t-1})$



Example #3

```
x[1] ~ Gaussian(0.0, 1.0);
y[1] ~ Gaussian(x[1], 1.0);
for (t in 2..T) {
  x[t] ~ Gaussian(a*x[t - 1], 1.0);
  y[t] ~ Gaussian(x[t], 1.0);
}
```

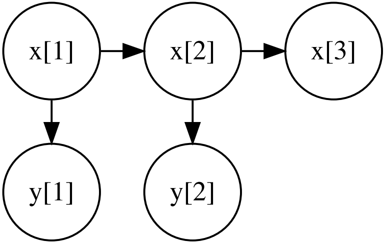
$p(y_t \mid x_t)$



Example #3

```
x[1] ~ Gaussian(0.0, 1.0);
y[1] ~ Gaussian(x[1], 1.0);
for (t in 2..T) {
  x[t] ~ Gaussian(a*x[t - 1], 1.0);
  y[t] ~ Gaussian(x[t], 1.0);
}
```

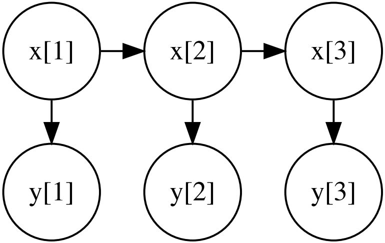
$p(x_t \mid x_{t-1})$



Example #3

```
x[1] ~ Gaussian(0.0, 1.0);
y[1] ~ Gaussian(x[1], 1.0);
for (t in 2..T) {
  x[t] ~ Gaussian(a*x[t - 1], 1.0);
  y[t] ~ Gaussian(x[t], 1.0);
}
```

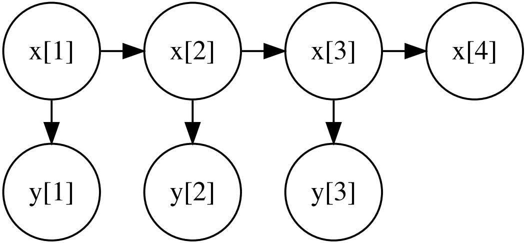
$p(y_t \mid x_t)$



Example #3

```
x[1] ~ Gaussian(0.0, 1.0);
y[1] ~ Gaussian(x[1], 1.0);
for (t in 2..T) {
  x[t] ~ Gaussian(a*x[t - 1], 1.0);
  y[t] ~ Gaussian(x[t], 1.0);
}
```

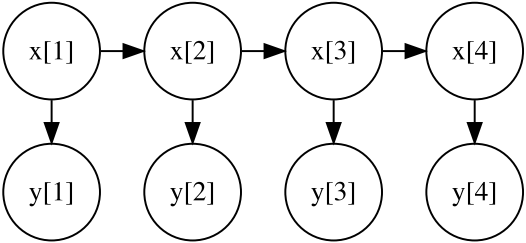
$p(x_t \mid x_{t-1})$



Example #3

```
x[1] ~ Gaussian(0.0, 1.0);
y[1] ~ Gaussian(x[1], 1.0);
for (t in 2..T) {
  x[t] ~ Gaussian(a*x[t - 1], 1.0);
  y[t] ~ Gaussian(x[t], 1.0);
}
```

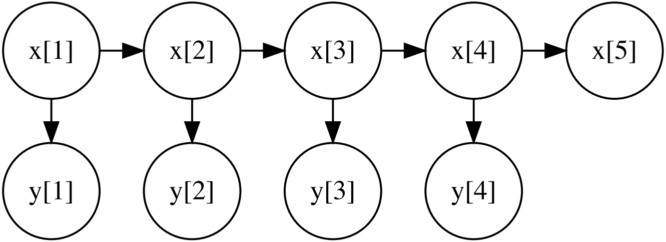
$p(y_t \mid x_t)$



Example #3

```
x[1] ~ Gaussian(0.0, 1.0);
y[1] ~ Gaussian(x[1], 1.0);
for (t in 2..T) {
  x[t] ~ Gaussian(a*x[t - 1], 1.0);
  y[t] ~ Gaussian(x[t], 1.0);
}
```

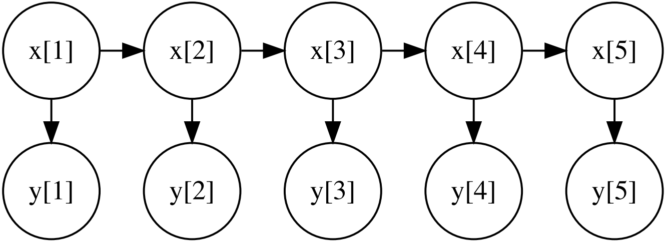
$p(x_t \mid x_{t-1})$



Example #3

```
x[1] ~ Gaussian(0.0, 1.0);
y[1] ~ Gaussian(x[1], 1.0);
for (t in 2..T) {
  x[t] ~ Gaussian(a*x[t - 1], 1.0);
  y[t] ~ Gaussian(x[t], 1.0);
}
```

$p(y_t \mid x_t)$



Example #3

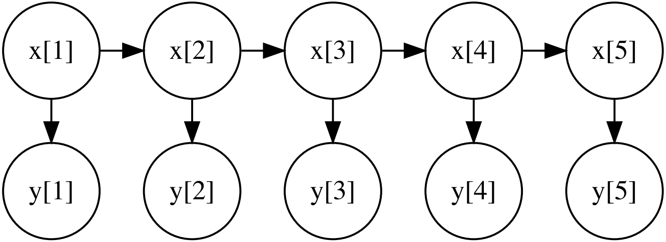
```
x[1] ~ Gaussian(0.0, 1.0);
y[1] ~ Gaussian(x[1], 1.0);
for (t in 2..T) {
  x[t] ~ Gaussian(a*x[t - 1], 1.0);
  y[t] ~ Gaussian(x[t], 1.0);
}
```

$p(x_1)$

$p(y_1 \mid x_1)$

$p(x_t \mid x_{t-1})$

$p(y_t \mid x_t)$



Checkpoints

The PPL will define checkpoints when interesting events happen in the running of the program. A typical setup uses two categories of checkpoint:

1. **Sample** when `x` is distributed according to some distribution `p` and should be sampled.
2. **Observe** when `x` is distributed according to some distribution `p` and should be observed to have some given value.

In Birch, these are triggered by special operators:

1. `x <~ p`
2. `x ~> p`

Example (Checkpoints)

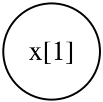
```
x[1] <~ Gaussian(0.0, 1.0);
y[1] ~> Gaussian(x[1], 1.0);
for (t in 2..T) {
  x[t] <~ Gaussian(a*x[t - 1], 1.0);
  y[t] ~> Gaussian(x[t], 1.0);
}
```

Now, the program explicitly states which variables must be sampled, and which have given values and should be observed. The program encodes a **posterior distribution**.

Example (Checkpoints)

```
x[1] <~ Gaussian(0.0, 1.0);
y[1] ~> Gaussian(x[1], 1.0);
for (t in 2..T) {
  x[t] <~ Gaussian(a*x[t - 1], 1.0);
  y[t] ~> Gaussian(x[t], 1.0);
}
```

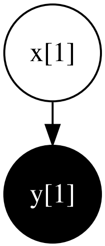
sample(x[1])



Example (Checkpoints)

```
x[1] <~ Gaussian(0.0, 1.0);
y[1] ~> Gaussian(x[1], 1.0);
for (t in 2..T) {
  x[t] <~ Gaussian(a*x[t - 1], 1.0);
  y[t] ~> Gaussian(x[t], 1.0);
}
```

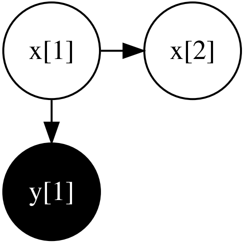
observe(x[1])



Example (Checkpoints)

```
x[1] <~ Gaussian(0.0, 1.0);
y[1] ~> Gaussian(x[1], 1.0);
for (t in 2..T) {
  x[t] <~ Gaussian(a*x[t - 1], 1.0);
  y[t] ~> Gaussian(x[t], 1.0);
}
```

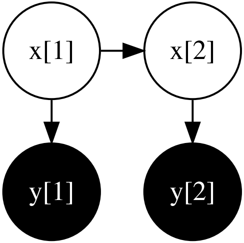
sample(x[t])



Example (Checkpoints)

```
x[1] <~ Gaussian(0.0, 1.0);
y[1] ~> Gaussian(x[1], 1.0);
for (t in 2..T) {
  x[t] <~ Gaussian(a*x[t - 1], 1.0);
  y[t] ~> Gaussian(x[t], 1.0);
}
```

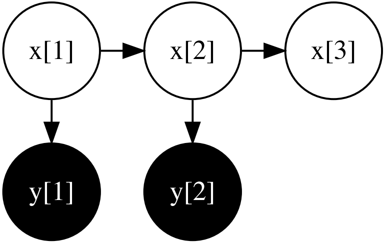
observe(x[t])



Example (Checkpoints)

```
x[1] <~ Gaussian(0.0, 1.0);
y[1] ~> Gaussian(x[1], 1.0);
for (t in 2..T) {
  x[t] <~ Gaussian(a*x[t - 1], 1.0);
  y[t] ~> Gaussian(x[t], 1.0);
}
```

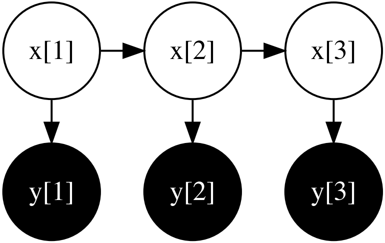
sample(x[t])



Example (Checkpoints)

```
x[1] <~ Gaussian(0.0, 1.0);
y[1] ~> Gaussian(x[1], 1.0);
for (t in 2..T) {
  x[t] <~ Gaussian(a*x[t - 1], 1.0);
  y[t] ~> Gaussian(x[t], 1.0);
}
```

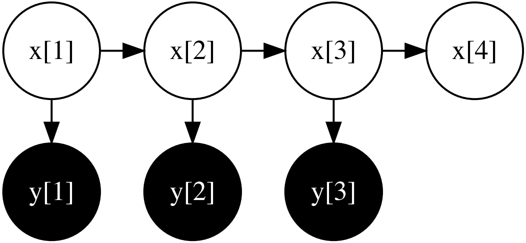
observe(x[t])



Example (Checkpoints)

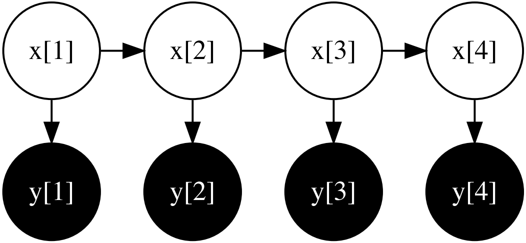
```
x[1] <~ Gaussian(0.0, 1.0);
y[1] ~> Gaussian(x[1], 1.0);
for (t in 2..T) {
  x[t] <~ Gaussian(a*x[t - 1], 1.0);
  y[t] ~> Gaussian(x[t], 1.0);
}
```

sample(x[t])



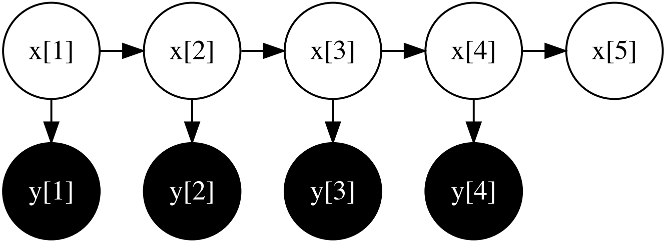
Example (Checkpoints)

```
x[1] <~ Gaussian(0.0, 1.0);
y[1] ~> Gaussian(x[1], 1.0);
for (t in 2..T) {
  x[t] <~ Gaussian(a*x[t - 1], 1.0);
  y[t] ~> Gaussian(x[t], 1.0);          observe(x[t])
}
```



Example (Checkpoints)

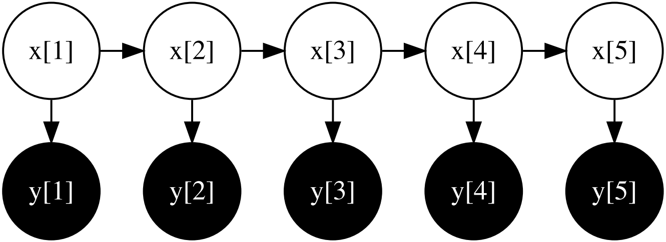
```
x[1] <~ Gaussian(0.0, 1.0);
y[1] ~> Gaussian(x[1], 1.0);
for (t in 2..T) {
  x[t] <~ Gaussian(a*x[t - 1], 1.0);          sample(x[t])
  y[t] ~> Gaussian(x[t], 1.0);
}
```



Example (Checkpoints)

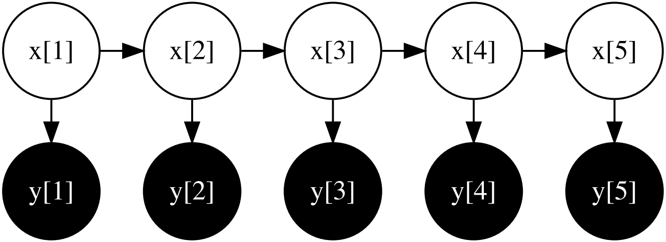
```
x[1] <~ Gaussian(0.0, 1.0);
y[1] ~> Gaussian(x[1], 1.0);
for (t in 2..T) {
  x[t] <~ Gaussian(a*x[t - 1], 1.0);
  y[t] ~> Gaussian(x[t], 1.0);
}
```

observe(x[t])



Example (Checkpoints)

```
x[1] <~ Gaussian(0.0, 1.0);                 sample(x[1])
y[1] ~> Gaussian(x[1], 1.0);                observe(x[1])
for (t in 2..T) {
  x[t] <~ Gaussian(a*x[t - 1], 1.0);        sample(x[t])
  y[t] ~> Gaussian(x[t], 1.0);              observe(x[t])
}
```



A probabilistic program encodes a **probabilistic model**.

A running probabilistic program is a **stochastic process**.

SMC for probabilistic programs

SMC for probabilistic programs

Recall that SMC can be used to approximate a sequence of probability distributions on a sequence of probability spaces of increasing dimension.

Let $\{\pi_k(x_{1:k})\}_{k\geq 1}$ be the sequence of target distributions

$$\pi_k(x_{1:k}) = \frac{\widetilde{\pi}_k(x_{1:k})}{Z_k}$$

Where

$$\pi_k(x_{1:k}) \approx \sum_{i=1}^{N} w_k^i \delta_{x_{1:k}^i}(x_{1:k})$$

and the weighted particle populations $\{x_{1:k}^i, w_k^i\}_{i=1}^N$ are generated sequentially for $k = 1, 2, \ldots$.

General Sequential Monte Carlo

We can use SMC for inference on running probabilistic programs.

- Each of the N particles is a running probabilistic program.
- We have:

$$\begin{aligned} \widetilde{\pi}_k(x_{1:k}) &= p_k(x_k \mid x_{1:k-1})\widetilde{\pi}_{k-1}(x_{1:k-1}) \\ q_k(x_k \mid x_{1:k-1}) &= p_k(x_k \mid x_{1:k-1}). \end{aligned}$$

- That is, the probabilistic program defines the target and the proposal, much like the bootstrap particle filter.

Assume that we have obtained $\{x^i_{1:k-1}, w^i_{k-1}\}_{i=1}^N$.

1. Resample: Sample a^i_k with $\mathbb{P}\big(a^i_k = j\big) = \nu^j_{k-1}$, $j = 1, \ldots, N$.
2. If this is a **sample** checkpoint, then propagate:

$$x^i_k \sim p_k(x_k \,|\, x^{a^i_k}_{1:k-1}) \text{ and } x^i_{1:k} = (x^{a^i_k}_{1:k-1}, x^i_k)$$

3. If this is an **observe** checkpoint, then weight:

$$w^i_k \propto \frac{w^{a^i_k}_{k-1} p_k(x^i_k \,|\, x^{a^i_k}_{1:k-1})}{\nu^{a^i_k}_{k-1}}.$$

The result is a new weighted set of particles $\{x^i_{1:k}, w^i_k\}_{i=1}^N$.

A few concepts to summarize lecture 17

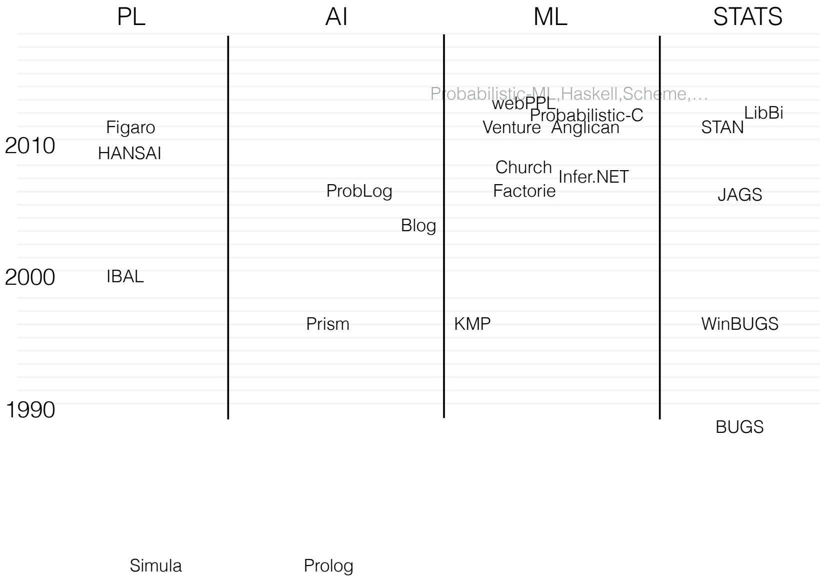
A probabilistic program encodes a probabilistic model according to the semantics of a particular probabilistic programming language.

The memory state of a running probabilistic program evolves dynamically and stochastically in time and so is a stochastic process.

General Sequential Monte Carlo can be applied to perform inference across a sequence of target distributions defined by the checkpoints of the running program.

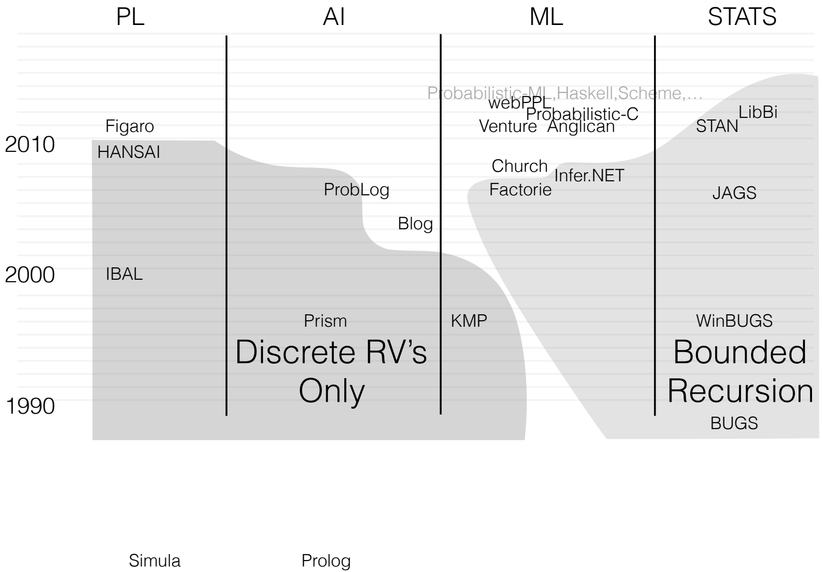
Further study

Probabilistic Programming Languages



(Figure courtesy of Frank Wood, Oxford.)

Probabilistic Programming Languages



(Figure courtesy of Frank Wood, Oxford.)