Probabilistic Programming and Sequential Monte Carlo

Lawrence Murray
Probabilistic Programming: The Full Stack

Methodology

1 Models
2 Methods

Language

3 Memory
4 Compute
5 I/O

Hardware

Problem and solution formulation
Abstract toolbox
Concrete toolbox
Efficient execution
Dedicated hardware

Integration with other systems

Cloud
Language
Probabilistic models can be expressed in many different ways:

**Mathematical**

\[ p(\sigma^2)p(\beta | \sigma^2)p(y | \beta, \sigma^2) \]

**Graphical**

![Graphical model](attachment:image.png)

**Programmatic**

\[ \sigma^2 \sim \text{InverseGamma}(3.0, 0.4); \]
\[ \beta \sim \text{Gaussian}(0.0, \sigma^2); \]
\[ y \sim \text{Gaussian}(x*\beta, \sigma^2); \]
Probabilistic programming languages

What might we want out of a probabilistic programming language?

- Ergonomic representation of models
- Automatic inference

What are the building blocks?

- Automatic differentiation
- Automatic marginalization
- Automatic conditioning
Given a scalar function \( f : \mathbb{R}^D \rightarrow \mathbb{R} \), we may be interested in evaluating its gradient \( \nabla f(x) \) at a given point \( x \in \mathbb{R}^D \).

The aim of **automatic differentiation** is to evaluate \( \nabla f(x) \) given only a program that implements \( f \).

Typically, a gradient is computed for the purpose of a gradient-based Markov kernel—such as a Langevin or Hamiltonian kernel—and \( f \) is a log-posterior density function:

\[
    f(x) := \log p(x \mid y).
\]
Automatic marginalization

Given a joint distribution $p(x, y) = p(x)p(y \mid x)$, **marginalization** is the computation:

$$p(y) = \int_x p(x)p(y \mid x)dx.$$

That is, we **marginalize out** $x$ to obtain the **marginal distribution** over $y$.

When $x$ is a discrete-valued random variable, we may instead write the integral as a sum:

$$p(y) = \sum_{x \in X} p(x)p(y \mid x).$$

The aim of **automatic marginalization** is to determine $p(y)$ given only $p(x)$ and $p(y \mid x)$.

While we can't support everything, automatic marginalization for standard conjugate forms, linear transformations, and sums and differences of discrete random variables is nice.
Automatic conditioning

Given a joint distribution $p(x, y) = p(x)p(y \mid x)$, *conditioning* is the computation:

$$p(x \mid y) = \frac{p(y \mid x)p(dx)}{p(y)}.$$

That is, we *condition* on the value of $y$ to obtain the *conditional distribution* of $x$ given $y$.

We may also refer to this as *Bayesian updating*, insofar as we interpret $p(x)$ as a prior distribution that we update to a posterior distribution $p(x \mid y)$.

The aim of *automatic conditioning* is to determine $p(x \mid y)$ given only $p(x)$ and $p(y \mid x)$.

Again, we can't support everything, but automatic conditioning for the same forms as automatic marginalization is nice.
For demonstration, we'll use a probabilistic programming language called Birch (https://birch.sh).
We can declare a variable and assign it an initial value:

```plaintext
let x ← 0.0;
```

We can simulate from a distribution with the `simulate` operator (<~):

```plaintext
x ← Gaussian(0.0, 4.0);
```

We can observe a variate with the `observe` operator (~>):

```plaintext
x ~> Gaussian(0.0, 4.0);
```

We can declare a random variable and associate a distribution for it with the `assume` operator (~):

```plaintext
y:Random<Real>;
  y ~ Gaussian(0.0, 4.0);
```
Birch supports both *immediate* and *delayed* expressions. The latter facilitate automatic differentiation.

Assume that we have variables $a$, $x$, and $c$, declared in code as:

```plaintext
let x ← 5.0;
let a ← 2.0;
let c ← -1.0;
```

Then, in:

```plaintext
let y ← exp(a*x + c);
```

$y$ will contain the result of evaluation $\exp(ax + c) = \exp(2.0 \times 5.0 - 1.0) \approx 8103.08$, i.e. immediate evaluation.
Automatic differentiation

On the other hand, if one or more of \(a, x,\) or \(c\) is declared to have type \(\text{Random}<\text{Real}>:\)

\[
x: \text{Random}<\text{Real}>;
\]
\[
\text{let } a \leftarrow 2.0;
\]
\[
\text{let } c \leftarrow -1.0;
\]

then this will construct a delayed expression, rather than evaluating the expression:

\[
\text{let } y \leftarrow \exp(a \times x + c);
\]

\(y\) is now a delayed expression. It can be evaluated later by calling \(y.\text{value()}\), but \(x\) will need a value first!

With delayed expressions, we can compute the gradient with respect to all the Random objects that appear in the expression.
Automatic marginalization

Consider:

```plaintext
let x ~ Beta(2,0, 2.0);
let y ~ Binomial(100, x);
```

The code defines $p(x, y) = p(x)p(y | x)$, and from this, automatic marginalization determines that $p(y)$ is a beta-binomial distribution.

This means that $y$ can be simulated from its marginal distribution, or we can compute its marginal likelihood, without first simulating $x$. 
Similarly, we could use:

```plaintext
let x ~ Gamma(2.0, 1.0);
let y ~ Poisson(x);
```

because the gamma-Poisson form is recognized, or:

```plaintext
let x ~ Gaussian(0.0, 4.0);
let y ~ Gaussian(x, 4.0);
let z ~ Gaussian(y, 4.0);
```

because chains of Gaussians are recognized—here both $x$ and $y$ are marginalized out.
Automatic conditioning

Consider the following, where $x$ is initially marginalized out:

```plaintext
let x ~ Gamma(2.0, 1.0);
let y ~ Poisson(x);
```

Conditioning is later triggered if $y$ obtains a value, such as by calling $y$.value(). The distribution associated with $x$ is then updated to the conditional distribution of $x$ given $y$, in this case another gamma distribution.

Automatic marginalization and conditioning are implemented using a heuristic called *delayed sampling*. It's most evident in the way that conditioning is only performed on-demand by requesting values of random variables.

Methods

Methodology

Models

Language

Memory

Compute

I/O

Hardware
SMC: It's modular
Auxiliary Particle Filter (c.f. Lecture 7)

1. **Initialization**
   1. Sample $x_0^i \sim p(x_0)$.
   2. Set initial weights: $w_0^i = 1/N$.

2. for $t = 1, \ldots, T$
   1. **Resample:** sample ancestor indices $a_t^i \sim \mathcal{C}(\{v_t^j\}_{j=1}^N)$.
   2. **Propagate:** sample $x_t^i \sim q(x_t \mid x_{t-1}^{a_t^i}, y_t)$.
   3. **Weight:** compute

   $$\tilde{w}_t^i = \frac{w_{t-1}^{a_t^i}}{v_{t-1}^{a_t^i}} \frac{p(y_t \mid x_t^i)p(x_t^i \mid x_{t-1}^{a_t^i})}{q(x_t^i \mid x_{t-1}^{a_t^i}, y_t)}$$

   and normalize $w_t^i = \tilde{w}_t^i / \sum_{j=1}^N \tilde{w}_t^j$. 
In the fully-adapted case:

- \( \nu_{t-1}^i = w_{t-1}^i p(y_t \mid x_{t-1}^i) \).
- \( q(x_t \mid x_{t-1}^i, y_t) = p(x_t \mid x_{t-1}^i, y_t) \).
In the fully-adapted case:

- \( \nu_{t-1}^i = w_{t-1}^i p(y_t \mid x_{t-1}^i) \).
- \( q(x_t \mid x_{t-1}^i, y_t) = p(x_t \mid x_{t-1}^i, y_t) \).

The first we get from automatic marginalization, the second we get from automatic conditioning.
1. Initialization
   1. Sample $x_0^i \sim p(x_0)$.
   2. Set initial weights: $w_0^i = 1/N$.

2. for $t = 1, \ldots, T$
   1. Weight: compute $\tilde{w}_t^i = p(y_t \mid x_{t-1}^i)$ and normalize $w_t^i = \tilde{w}_t^i / \sum_{j=1}^{N} \tilde{w}_t^j$.
   2. Resample: sample ancestor indices $a_t^i \sim C(\{w_t^j\}_{j=1}^{N})$.
   3. Propagate: sample $x_t^i \sim p(x_t \mid x_{t-1}^{a_t^i}, y_t)$.

We still implement the model by defining $p(x_t \mid x_{t-1})$ and $p(y_t \mid x_t)$, as is natural, but with automatic marginalization and automatic conditioning, the above is effectively what gets executed.
Resample-Move Particle Filter

1. **Initialization**
   1. Sample $x^i_0 \sim p(x_0)$.
   2. Set initial weights: $w^i_0 = 1/N$.

2. for $t = 1, \ldots, T$
   1. **Weight**: compute $\tilde{w}^i_t = p(y_t \mid x^i_{t-1})$ and normalize $w^i_t = \tilde{w}^i_t / \sum_{j=1}^{N} \tilde{w}^j_t$.
   2. **Resample**: sample ancestor indices $a^i_t \sim C(\{w^j_t\}_{j=1}^{N})$.
   3. **Move**: sample $\bar{x}^i_{1:t-1} \sim \kappa(\bar{x}_{1:t-1} \mid x^{a^i}_1)$.
   4. **Propagate**: sample $x^i_t \sim p(x_t \mid \bar{x}^{a^i}_{t-1}, y_t)$.

Here, $\kappa$ is a Markov kernel invariant to $p(x_{1:t-1} \mid y_{1:t-1})$; it can even be a gradient-based kernel that makes use of automatic differentiation, e.g. a Langevin or Hamiltonian kernel.

We can embed MCMC in SMC!
Extensions

- Rao-Blackwellized particle filters fall out of this too.
- With more complex models, we may only have partial adaptation, but automatic marginalization and conditioning will help yield that partial adaptation.
- Similarly, the kernel may only operate on a subset of the variables of the model, especially for a gradient-based kernel with a mix of continuous and discrete variables.
Models
Programmatic models

Consider a simple linear regression model with the programmatic representation:

\[
\sigma^2 \sim \text{InverseGamma}(3.0, 0.4); \\
\beta \sim \text{Gaussian}(\text{vector}(0.0, P), \sigma^2 \cdot \text{identity}(P)); \\
y \sim \text{Gaussian}(X^*\beta, \sigma^2);
\]

We can represent the model mathematically as:

\[
p(\sigma^2, \beta, y) = p(\sigma^2)p(\beta \mid \sigma^2)p(y \mid \beta, \sigma^2),
\]

and graphically as:

Each statement in the programmatic representation defines a new factor in the mathematical representation, and a new node in the graphical representation.
Many useful models can be represented in these three ways. Consider also a linear-Gaussian state-space model (hidden Markov model), represented programmatically as:

\[
x[1] \sim \text{Gaussian}(0.0, 4.0);
\]
\[
y[1] \sim \text{Gaussian}(b \times x[1], 1.0);
\]
\[
\text{for } t \text{ in } 2..4 
\begin{align*}
x[t] & \sim \text{Gaussian}(a \times x[t - 1], 4.0); \\
y[t] & \sim \text{Gaussian}(b \times x[t], 1.0);
\end{align*}
\]

mathematically as:

\[
p(x_{1:T}, y_{1:T}) = p(x_1)p(y_1 \mid x_1) \prod_{t=2}^{T} p(x_t \mid x_{t-1})p(y_t \mid x_t),
\]

and graphically as:

![Diagram of a linear-Gaussian state-space model](image-url)
A key feature of programmatic models is that random variables can influence control flow of the program. This is referred to as *stochastic branching*.

\[
t \sim \text{Gaussian}(25.0, 4.0);
c \sim \text{Bernoulli}(0.9);
\text{if } c \{ \\
    r \sim \text{Gamma}(2.0, \ 5.0 + t/25.0);
\} \text{ else } \{ \\
    r \leftarrow 0.0;
\}
\]

Here, \( c \) mediates whether or not \( r \) depends on \( t \).
An if statement may have significant deviation between branches. Consider a model selection task:

c <- Bernoulli(0.5);
if c {
  // run model A
} else {
  // run model B
}

The two models could involve very different sets of random variables and dependencies between them, i.e. different graphical models.
Loops can also exhibit stochastic branching. Consider enumerating the components of a Gaussian mixture model with a random number of components:

\[
K \sim \text{Geometric}(0.25);
\]

\[
\text{for } k \text{ in } 1..K \{
\quad \sigma^2_k \sim \text{InverseGamma}(2.0, 5.0);
\quad \mu_k \sim \text{Gaussian}(0.0, 0.1 \times \sigma^2_k);
\}
\]

The random variable \(K\) affects the control flow of the program. Each time the program is run, the for loop iterates a random number of times, and so the graphical model has a random number of nodes.
Consider a population model of an animal species, simulated with the Gillespie algorithm, where a random number of birth or death events occur in any given time interval:

\[
T \leftarrow 10.0; \quad \text{// end time} \\
x \leftarrow 100; \quad \text{// starting population} \\
t \leftarrow \text{Exponential}(1.0); \quad \text{// time of first event} \\
\text{while } t < T \{ \\
    b \leftarrow \text{Bernoulli}(0.5); \\
    \text{if } b \{ \\
        x \leftarrow x + 1; \quad \text{// birth event} \\
    \} \text{ else } \{ \\
        x \leftarrow x - 1; \quad \text{// death event} \\
    \} \\
    \Delta \leftarrow \text{Exponential}(1.0); \quad \text{// time to next event} \\
    t \leftarrow t + \Delta; \quad \text{// time of next event} \\
}\]

Here, the while loop executes a random number of times, until the \( t < T \) condition fails.
All of these programs exhibit stochastic branching. On each run they may generate:

- a different set of random variables and dependencies between them,
- i.e. a different set of nodes and edges between them,
- i.e. a different graphical model.
All of these programs exhibit stochastic branching. On each run they may generate:

- a different set of random variables and dependencies between them,
- i.e. a different set of nodes and edges between them,
- i.e. a different graphical model.

A programmatic model defines a distribution over graphical models.
Memory
Prince of Persia (1989)

But the real breakthrough this week was invisible: I moved a bunch of stuff around so the main game code can use the auxiliary language card. Basically, I’ve just freed up an extra 12K.

That gives me some breathing room I’ll sorely need if I’m going to put in all this swordfighting.

It was a good weekend.

— Jordan Mechner, The Making of Prince of Persia, p.75

Memory use for state-space models
Memory use for state-space models

○ ○ ○ ○ ○ ○ ○
Memory use for state-space models
Memory use for state-space models
Memory use for state-space models
Memory use for state-space models
Memory use for state-space models
Memory use for state-space models
Memory use for state-space models
Memory use for state-space models
Memory use for state-space models
Memory use for state-space models
Memory use for state-space models
Memory use for state-space models
Memory use for state-space models
Memory use for state-space models
Memory use for state-space models
Memory use for state-space models
Memory use for state-space models

$| \mathcal{C} | \sim O(NT)$
Memory use for state-space models

\[ \Theta(N) \]

\[ \sim O(NT) \]

\[ \sim O(N \log N + T) \]
Memory use for state-space models

\[ | \bullet | \sim O(N \log N + T) \]


Memory use for state-space models

- We now have a tree of objects—a recursive data structure.
- We can share objects between particles.
- **Copy on write:** to write to an object, if it is currently shared, it must first be copied.
But state-space models have a well-defined structure that naturally forms these trees. In general, programmatic models can use more complex data structures. How to handle this?
We can convert any arbitrary graph of objects, even containing cycles, into a tree of *biconnected components* connected by *bridges*, using Tarjan's bridge-finding algorithm.


[6] L.M. Murray (2020). *Lazy object copy as a platform for population-based probabilistic programming*. Note: This is the previous algorithm used by Birch, that based on bridge-finding is newer.
• We now have a tree of biconnected components, rather than a tree of objects.
• Instead of sharing objects, we share biconnected components.
• **Copy on write:** to write to a biconnected component, if it is currently shared, it must first be copied.
Compute
Generally, SMC is highly parallelizable.

- Propagating, weighting, and moving are independent between particles.
- Only the resampling requires interaction between particles.
Parallelism

Generally, SMC is highly parallelizable.

- Propagating, weighting, and moving are independent between particles.
- Only the resampling requires interaction between particles.

In practice it can be a little more complicated, e.g.:

- Reference-counted garbage collection may require atomic operations.
- Variable task length between particles may require load balancing.
Anytime Monte Carlo

- SMC$^2$ with ~4 billion particles (4096 $\theta$-particles each with 1048576 $x$-particles).
- Lorenz '96 model, 1 parameter, 8 state variables.
- Run on 128 GPUs on AWS, taking ~20 minutes.

This was done with an older probabilistic programming language: LibBi (https://libbi.org).

GPU Strategies: One Big Kernel

One CPU thread launches one big kernel on the GPU to propagate particles in parallel.

Fast: GPUs are good at this.

Computations may be too complex for single threads (registers, cache).

Will perform poorly if particles diverge, mileage will vary for programmatic models.

Higher GPU memory use: all particles on GPU at a time.
Each CPU thread maintains its own stream on the GPU, and independently launches smaller kernels to propagate particles one by one.

Not as fast: scheduling overhead for concurrent kernel launches.

Requires intra-particle parallelism to fully occupy the GPU.

No problem if particles diverge, so supports programmatic models well.

Lower GPU memory use: fewer particles on GPU at a time.
## GPU Strategies: Take Home Message

<table>
<thead>
<tr>
<th></th>
<th>One Big Kernel</th>
<th>Streaming Concurrent Kernels</th>
</tr>
</thead>
<tbody>
<tr>
<td>Supports divergence</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Requires parallelism within particles</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>GPU memory use</td>
<td>High $O(N)$</td>
<td>Moderate $O(P)$</td>
</tr>
<tr>
<td>Target use case</td>
<td>Small models?</td>
<td>Large models?</td>
</tr>
</tbody>
</table>

(where $N =$ number of particles, $P =$ number of CPU threads.)
Summary

Methodology

1. Models
2. Methods

Language

3. Memory
4. Compute
5. I/O

Hardware

Problem and solution formulation
Abstract toolbox
Concrete toolbox
Efficient execution
Dedicated hardware

Cloud
Integration with other systems

Hardware

1. Language
2. Methods
3. Models

Problem and solution formulation
Abstract toolbox
Concrete toolbox
Efficient execution
Dedicated hardware

Cloud
Integration with other systems
1. **Language:** simulate, observe, assume, automatic differentiation, automatic marginalization, automatic conditioning, delayed expressions.

2. **Methods:** the building blocks of the language mean that SMC methods such as the auxiliary, Rao-Blackwellized and resample-move particle filters emerge.

3. **Models:** stochastic branching, where random variables affect the control flow of a program, programmatic models, which define a distribution over graphical models.

4. **Memory:** $O(N \log N + T)$ memory use is possible, for programmatic models need some sort of object sharing, Birch does so using garbage collection and Tarjan's bridge-finding algorithm.

5. **Compute:** GPUs are great, can we make them work for the general class of programmatic models?

To learn more about Birch, see [https://birch.sh](https://birch.sh)