Probabilistic programming for Sequential Monte Carlo?

Nicolas Chopin, ENSAE IPP (Institut Polytechnique de Paris)

My areas of expertise

- Computational Statistics (and a little bit of Machine Learning): how to derive efficient algorithms to learn (estimate) models from data

My areas of expertise

- Computational Statistics (and a little bit of Machine Learning): how to derive efficient algorithms to learn (estimate) models from data
- Focus on Bayesian Statistics (uncertainty represented by a posterior distribution)

My areas of expertise

- Computational Statistics (and a little bit of Machine Learning): how to derive efficient algorithms to learn (estimate) models from data
- Focus on Bayesian Statistics (uncertainty represented by a posterior distribution)
- Particular interest in Monte Carlo methods (quasi-Monte Carlo, Markov chain Monte Carlo, Sequential Monte Carlo, . . .)

Sutor, ne ultra crepidam

(Casa Vasari, Florence)

Sutor, ne ultra crepidam

(Casa Vasari, Florence)

Which is just a pompous way to say I feel a bit out of my depth today.

This talk

1. A candid take on why PP (probabilistic programming) may or may not be useful to scientists like me;
2. An introduction to Sequential Monte Carlo, state-space models, and why they might be interesting to PP experts

A confession

I have never used any PP language...

A confession

I have never used any PP language...

- PPLs are for describing **probabilistic models**;

A confession

I have never used any PP language...

- PPLs are for describing **probabilistic models**;
- I need tools / languages to implement **stochastic algorithms**.

A confession

I have never used any PP language. . .

- PPLs are for describing **probabilistic models**;
- I need tools / languages to implement **stochastic algorithms**.

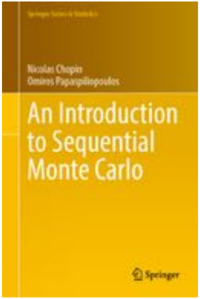
In other words, PPLs I am familiar with tends to force you to use a certain inferential algorithm, e.g.:

- Bugs / JAGS: Gibbs sampler
- STAN: no-U turn sampler

but my job is to derive alternative, possibly better algorithms. . .

Yet . . .

To complement this book:



I developed **particles**, a Python package that implements all the algorithms presented in the book. **Ironically**, I ended up implementing some (basic) form of PP.

Sequential Monte Carlo

One name, two types of applications:

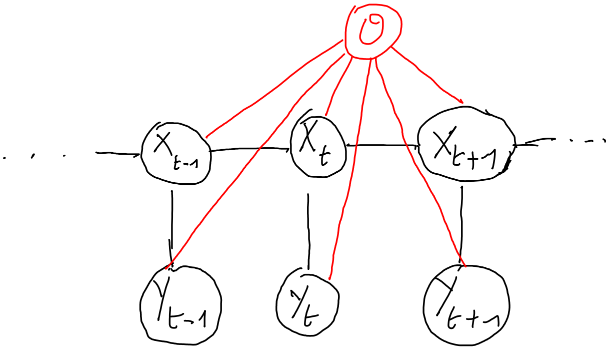
- **particle filters**: for sequential analysis of **state-space models**;

Sequential Monte Carlo

One name, two types of applications:

- **particle filters**: for sequential analysis of **state-space models**;
- **SMC samplers**: for simulating *one or several* probability distributions (and computing their normalising constants).

State-space models



- $X_0, \ldots, X_t, \ldots$ is an **unobserved** Markov chain;
- Data Y_t are a noisy observation of some function of X_t;
- model may be parametric, with parameter θ.

Example 1: Part-of-Speech tagging in NLP

Task: label each word in a sentence with its part of speech (noun, verb, adjective, . . .):

Example 1: Part-of-Speech tagging in NLP

Task: label each word in a sentence with its part of speech (noun, verb, adjective, . . .):

- Y_t: words (observed)
- X_t: parts of speech (to be inferred)

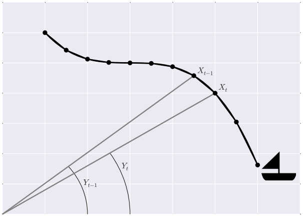
Example 1: Part-of-Speech tagging in NLP

Task: label each word in a sentence with its part of speech (noun, verb, adjective, ...):

- Y_t: words (observed)
- X_t: parts of speech (to be inferred)

Not entirely relevant for today, as you don't need a particle filter when the state-space is finite.

Example 2: (bearings-only) target tracking



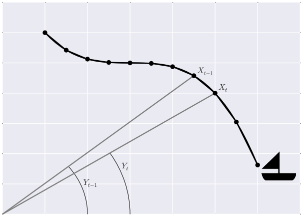
X_t is position of e.g. a ship

$$X_t = X_{t-1} + U_t, \quad U_t \sim N_2(0, \sigma^2 I_2),$$

and Y_t is a **radar** measurement

$$Y_t = \mathrm{atan}\left(\frac{X_t(2)}{X_t(1)}\right) + V_t, \quad V_t \sim N_1(0, \sigma_Y^2).$$

Example 2: (bearings-only) target tracking



X_t is position of e.g. a ship

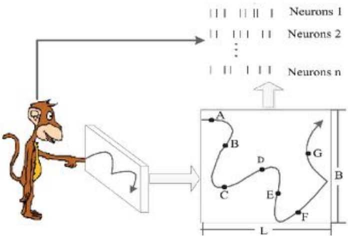
$$X_t = X_{t-1} + U_t, \quad U_t \sim N_2(0, \sigma^2 I_2),$$

and Y_t is a **radar** measurement

$$Y_t = \mathrm{atan}\left(\frac{X_t(2)}{X_t(1)}\right) + V_t, \quad V_t \sim N_1(0, \sigma_Y^2).$$

Task is to recover the current position (or complete trajectory) of the ship based on the radar measurements.

Example 3: neural decoding

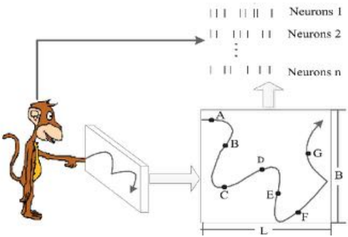


Y_t is a vector of d_y counts (spikes from neuron k),

$$Y_t(k)|X_t \sim \mathrm{Poisson}(\lambda_k(X_t)), \quad \log \lambda_k(X_t) = \alpha_k + \beta_k X_t,$$

and X_t is position+velocity of the subject's hand (in 3D).

Example 3: neural decoding



Y_t is a vector of d_y counts (spikes from neuron k),

$$Y_t(k)|X_t \sim \mathrm{Poisson}(\lambda_k(X_t)), \quad \log \lambda_k(X_t) = \alpha_k + \beta_k X_t,$$

and X_t is position+velocity of the subject's hand (in 3D).

Task is to recover the hand's movement based on neuronal measurements.

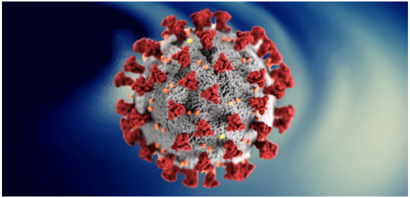
Example 4: population ecology

(Source: King Lab, UGA)

Each time you collect / tags specimens, you observe $Y_t = X_t + \mathrm{noise}$, where X_t is the log of the population size, which follows certain population dynamics such as

$$X_t = X_{t-1} + \theta_1 - \theta_2 \exp(\theta_3 X_{t-1}) + U_t, \quad U_t \sim N(0, \sigma^2)$$

Example 5: Epidemiology



- SIR (Susceptible, Infectious, Recovered) model for (X_t)
- We observe only one component, e.g. number of recovered (plus noise)

Other examples

- genetics: X_t is expression level at position t
- finance: stochastic volatility
- Elections: X_t is propensity to vote for a certain candidate, Y_t is opinion poll
- etc.

Sequential analysis of state-space models

We typically want to derive **sequentially** (at times $t = 0, 1, \ldots,$) the following quantities / distributions:

- Filtering: $X_t | Y_{0:t} = y_{0:t}$
- Smoothing: $X_{0:t} | Y_{0:t} = y_{0:t}$
- likelihood (for parameter estimation): $p_t^\theta(y_{0:t})$

Bootstrap filter

Most basic particle filter. Only requirements from the model:

- being able to simulate $X_t | X_{t-1}$
- being able to compute $f_t(y_t | x_t)$, the density of $Y_t | X_t = x_t$.

Bootstrap filter

Most basic particle filter. Only requirements from the model:

- being able to simulate $X_t|X_{t-1}$
- being able to compute $f_t(y_t|x_t)$, the density of $Y_t|X_t = x_t$.

Inputs:

- data: $y_0, y_1, \ldots$
- Number N of particles

Bootstrap filter algorithm

Each operation must be performed for all $n = 1, \ldots, N$.

At time 0:

(a) $X_0^n \sim p_0(x_0)$

(b) $w_0^n \leftarrow f_0(y_0|X_0^n)$, and $W_0^n \leftarrow w_0^n / \sum_{m=1}^N w_0^m$

At times $1, 2, \ldots, T$:

(a) $A_t^n \sim \mathcal{M}(W_{t-1}^{1:N})$ (i.e. $A_t^n \leftarrow m$ with prob. W_{t-1}^m)

(b) $X_t^n \sim p_t(x_t|X_{t-1}^{A_t^n})$

(c) $w_t^n \leftarrow f_t(y_t|X_t^n)$ and $W_t^n \leftarrow w_t^n / \sum_{m=1}^N w_t^m$

Output and validity

- essentially sequential importance sampling.

Output and validity

- essentially sequential importance sampling.
- At each time t, weighted sample $(X_t^{1:N}, W_t^{1:N})$ approximates the filtering distribution:

$$\sum_{n=1}^{N} W_t^n \varphi(X_t^n) \approx \mathbb{E}[\varphi(X_t) | Y_{0:t} = y_{0:t}]$$

Output and validity

- essentially sequential importance sampling.
- At each time t, weighted sample $(X_t^{1:N}, W_t^{1:N})$ approximates the filtering distribution:

$$\sum_{n=1}^{N} W_t^n \varphi(X_t^n) \approx \mathbb{E}[\varphi(X_t) | Y_{0:t} = y_{0:t}]$$

- Converges as $N \to +\infty$ at the usual $N^{-1/2}$ rate.

Output and validity

- essentially sequential importance sampling.
- At each time t, weighted sample $(X_t^{1:N}, W_t^{1:N})$ approximates the filtering distribution:

$$\sum_{n=1}^{N} W_t^n \varphi(X_t^n) \approx \mathbb{E}[\varphi(X_t) | Y_{0:t} = y_{0:t}]$$

- Converges as $N \to +\infty$ at the usual $N^{-1/2}$ rate.

Defining a state-space model in particles

To represent the following (stochastic volatility) model:

$$X_0 \sim N(0, \sigma^2)$$
$$X_t | X_{t-1} = x_{t-1} \sim N(\rho x_{t-1}, \sigma^2)$$
$$Y_t | X_t = x_t \sim N(0, e^{x_t})$$

```python
class StochVol(StateSpaceModel):
    def PX0(self):
        return Normal(scale=self.sigma)
    def PX(self, t, xp):
        return Normal(loc=self.rho * xp,
                      scale=self.sigma)
    def PY(self, t, xp, x):
        return Normal(scale=np.exp(0.5 * x))
```

Running a bootstrap filter

```
ssm = StochVol(rho=0.9, sigma=0.3)
x, y = ssm.simulate(100)
fk = Bootstrap(ssm=ssm, data=y)
pf = SMC(fk=fk, N=100, resampling='systematic')
pf.run()
```

At each time t, the algorithm generates a weighted sample whose empirical distribution approximates the distribution of $X_t | Y_{0:t}$.

Running PMMH (Bayesian parameter estimation)

```
prior_dict = {'sigma': Gamma(a=1., b=1.),
              'rho': Beta(9., 1.)}
my_prior = StructDist(prior_dict)
pmmh = PMMH(ssm_cls=StochVol, prior=my_prior,
            data=data, Nx=50, niter = 1000)
pmmh.run()
```

This runs a Markov chain that leaves invariant $\theta | Y_{0:T}$.

Under the hood

- In an OO language, it's easy to represent random distributions as **objects**, with methods for sampling, computing the log pdf, etc.

When things start to break apart...

Consider now the slightly more advanced (bearings-only tracking) state-space model:

$$X_t | X_{t-1} = x_{t-1} \sim N_2(x_{t-1}, \mathbf{I}_2)$$
$$Y_t | X_t = x_t \sim N_1 \left(\operatorname{atan} \left(\frac{X_t[1]}{X_t[2]} \right), \sigma_Y^2 \right)$$

```
class BearingsOnly(StateSpaceModel):
    # A few bits missing ...

    def PX(self, t, xp):
        return IndepProd(Normal(loc=xp[:, 0]),
                         Normal(loc=xp[:, 1]))

    def PY(self, t, xp, x):
        angle = np.arctan(x[:, 0] / x[:, 1])
        return dists.Normal(loc=angle,
                            scale=self.sigmaY)
```

Explanation

The probability distributions defined in **particles** actually operate on arrays of shape N, or (N, d); e.g. to simulate N particles X_t^n given an array of N ancestors (values of X_{t-1}).

Explanation

The probability distributions defined in **particles** actually operate on arrays of shape N, or (N, d); e.g. to simulate N particles X_t^n given an array of N ancestors (values of X_{t-1}).

See the notebook tutorials in the documentation for more details.

(pseudo-)Philosophical conclusion

"Those who don't understand PPL are condemned to reinvent them, poorly."

(pseudo-)Philosophical conclusion

"Those who don't understand PPL are condemned to reinvent them, poorly."

Would love to get feedback from PPL experts on how to make the specification of a state-space model more transparent to users (e.g. no reference to arrays).

Practical conclusion: should you use particles?

Pros:

- feature-rich: particle smoothing, SQMC, 7 resampling schemes, SMC samplers, variance estimators.
- Python: low barrier of entry (for users and contributors); big ecosystem (e.g. pytorch / JAX); numpy parallel

For raw performance, use instead Birch (+ Libbi), a true PPL for state-space modelling developed by Lawrence Murray.

How to make more ppl like me use PPLs?

Again, in computational stats & machine learning, what we need is tools to implement and experiment with *new* algorithms.

- modularity, open the box: makes it easy to recycle / adapt certain parts (even low-level ones)

How to make more ppl like me use PPLs?

Again, in computational stats & machine learning, what we need is tools to implement and experiment with *new* algorithms.

- modularity, open the box: makes it easy to recycle / adapt certain parts (even low-level ones)
- low barrier of entry

How to make more ppl like me use PPLs?

Again, in computational stats & machine learning, what we need is tools to implement and experiment with *new* algorithms.

- modularity, open the box: makes it easy to recycle / adapt certain parts (even low-level ones)
- low barrier of entry
- by wich I mean: Python

Alternatively

Targetting applied statisticians makes even more sense, of course.

- No more need to "open the box" so much.

Alternatively

Targetting applied statisticians makes even more sense, of course.

- No more need to "open the box" so much.
- focus on performance rather than implementing any existing method under the sun.

Alternatively

Targetting applied statisticians makes even more sense, of course.

- No more need to “open the box” so much.
- focus on performance rather than implementing any existing method under the sun.

Again, Birch seems to fit that bill very well for state-space models. See algo Andrew Gelman’ talk tonight and his perspective on STAN and related software.