Introduction to Particle Smoothing and pyParticleEst

Jerker Nordh

Dept. of Automatic Control
Lund University

2013-04-29

Why particle methods?

Benefits

- Can handle non-linear systems
- Can handle non-Gaussian noise

Drawbacks

- Approximate solutions
- Non-deterministic solutions
- Computationally demanding
  - both processing time and/or memory depending on the specific problem

Particle filtering algorithm

- For $i = 1..N$ initialize particles such that $x_0^{(i)}$ is sampled from the initial distribution $p(x_0)$. For each particle associate a weight, $\omega^{(i)}$, typically uniformly.
- Propagate system state forward in time, for each particle:
  - Sample from the input and/or state noise distributions
  - Propagate the state belief deterministically using the sampled noise
  - Update each particle weight as $\omega_t^{(i)} = p(y_t|x_t^{(i)})\omega_{t-1}^{(i)}$
- For each time instant the collection of particles with weights is a sampled approximation of the true probability density function for the filtering problem.
  - $p(x_t|y_t, ..., y_0) \approx \sum_i w_t^{(i)} \delta(x_t - x_t^{(i)})$

Particle filtering details

Issues

- Approximate solutions, guaranteed correct only for $N \to \inf$
- Non-deterministic solutions
- The approximation deteriorates when only a few particles remain likely, so called **particle depletion**

Implementation details

- Particle depletion can be mitigated by resampling, ie. discarding particles with low weights
  - After every time step
  - When the number of **effective particles** falls below a threshold
    - $N_eff = \frac{1}{\sum(w^{(i)})^2} < \frac{2}{3}$
- Typically it is numerically preferable to compute $log(\omega^{(i)})$ instead of $\omega^{(i)}$

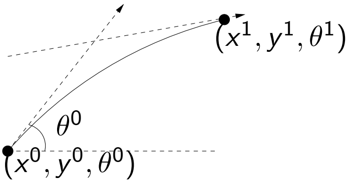
What is particle smoothing?

- Particle method using both past and future data, want to estimate $p(x_t|y_0,..,y_T), t \in (0,T)$
- Trivial solution, use the estimate $p(x_t|y_0,..,y_t)$ but use the weights $w_T^{(i)}$.
  - $p(x_t|y_T,..,y_t,..,y_0) \approx \sum_i w_T^{(i)} \delta(x_t - x_{t|t}^{(i)})$
  - Doesn't work if the particles have been resampled

Typical smoothing algorithm

- Generate filter estimates using a particle filter
  - $p(x_t|y_t,...,y_0) \approx \sum_i w_t^{(i)} \delta(x_t - x_t^{(i)})$
- Sample trajectories backwards
  - randomly choose previous state, $x_t^{(i)}$, according to $p(x_{t+1}|x_t^{(i)})$
- New difficulty: need to evaluate the next-state pdf
  - for filtering we only need to be able to sample from $p(x_{t+1}|x_t^{(i)})$

Not all models can benefit from smoothing



Typical differential drive model for wheel robots

- The bilinear transformation (the arc) is a second order approximation of the robot motion.
- The orientation at the endpoint is uniquely determined by the initial pose and the end point position
- By adding noise in the θ-state this uniqueness disappears, which is essential for particle methods
  - Helps avoid particle depletion, related to why standard particle methods aren't suitable for parameter estimation

Rao-Blackwellized methods

- The particle filter can be adapted to use a Kalman filter for the conditionally linear/guassian states
  - Called a Rao-Blackwellized Particle Filter
  - Saves both memory and computation time
- The particle smoother is not as easily extended in this way
  - F. Lindsten and T. Schön, "Rao-Blackwellised particle smoothers for mixed linear/nonlinear state-space models"
  - Lindsten, Schön assumes Gaussian noise, but the method can be adapted for other noise models

Rao-Blackwellized smoothing

- Run a RBPF forward in time yielding a filtered estimate
- Sample the Linear-Gaussian states before evaluating $p(x_{t+1}|x_t^{(i)})$
- Perform backward smoothing
  - $O(MN)$
  - Rejection sampling can improve this by not evaluating the density function for all particles
- Run a Rauch–Tung–Striebel Kalman smoother to obtain continuous estimates of the Linear-Gaussian states

What is pyParticleEst?

Python module/library implementing common tasks needed for particle methods

- Resampling
- Backward smoothing (optionally using rejection sampling)
- Object-Oriented structure
- Primitives for storing commonly used structures, eg. particle trajectories, collections of particles with weights

Code example - Setup

```
# Create a reference which we will try to estimate using a RBPS
correct = SimpleParticle(numpy.array([1.0, -0.5]),2.5)
# Create an array for our particles
particles = numpy.empty(num, type(correct))
# Initialize particles
for k in range(len(particles)):
    # Let the initial value of the non-linear state be U(2,3)
    particles[k] = SimpleParticle(numpy.array([[0.0],[0.0]]),
                                  numpy.random.uniform(2, 3))
# Create a particle approximation object from our particles
pa = PF.ParticleApproximation(particles=particles)
# Initialise a particle filter with our particle approximation
# of the initial state, set the resampling threshold to 0.67
pt = PF.ParticleTrajectory(pa,0.67)
```

Mathematical model

$$
\begin{aligned}
x_{k+1} &= \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} x_k + \begin{pmatrix} 0 & 0 \\ 1 & -1 \\ 0 & 0 \end{pmatrix} (u_k + v_k) + w_k \\
y_k &= \begin{pmatrix} x_k(3) & 0 & 0 \end{pmatrix} x_k + e_k \\
v_k &\sim N\left(0, \begin{pmatrix} 0.12 & 0 \\ 0 & 0.12 \end{pmatrix}\right) \\
w_k &\sim N\left(0, \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0.01 \end{pmatrix}\right) \\
e_k &\sim N(0, 1)
\end{aligned}
$$

Code example - class SimpleParticle

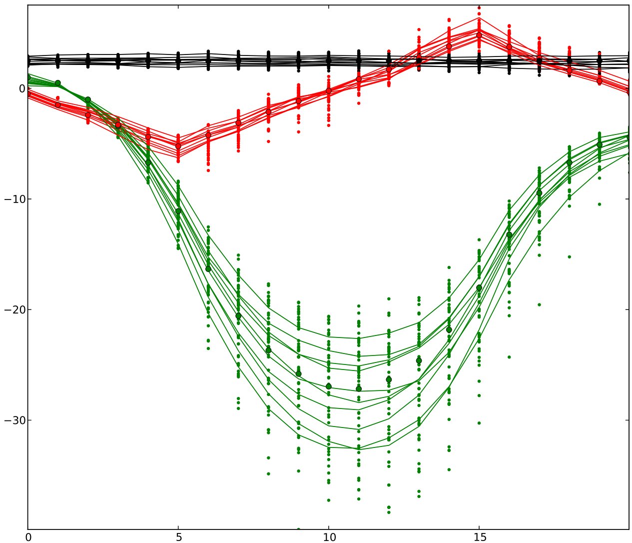
```
def __init__(self, x0, c):
    # Define all model variables (omitted for breivity)
    self.kf = kalman.KalmanSmoother(A,B,C,x0,P0=P,Q=None,R=R)
    self.c = c # Non-linear state
def sample_input_noise(self, u): #Return perturbed input
    s = math.sqrt(self.Q[0,0])
    tmp = numpy.random.normal(u[2],s)
    return numpy.vstack((u[:2], tmp))
def update(self, data): # Update states
    self.kf.time_update(u=self.linear_input(data),
                        Q=self.get_lin_Q()) # Cond. Linear
    self.c += data[2,0] # Non-linear
def measure(self, y):
    # measurement matrix C depends on the value of c
    C = numpy.array([[self.c, 0.0]])
    return numpy.log(self.kf.meas_update(y, C=C))
```

Code example - filtering+smoothing

```
# Run particle filter using the above generated data
for i in range(steps):
    u = uvec[:,i].reshape(-1,1)
    tmp = numpy.random.normal((0.0,0.0,0.0),
                              (0.1,0.1,0.0000000001))
    # Run PF using noise corrupted input signal
    pt.update(u+tmp.reshape((-1,1)))
    # Use noise corrupted measurements
    pt.measure(yvec[i]+numpy.random.normal(0.0,1.))

# Use the filtered estimates above to created smoothed estimates
nums = 10 # Number of backward trajectories to generate
straj = PS.do_smoothing(pt, nums)   # Do sampled smoothing
straj = PS.do_rb_smoothing(straj) # RBPS
```

Results



Implementation summary

- Filtering requires implementation of 3 methods
- Smoothing requires one additional methods
  - Two if using rejection sampling
- RBPF rather clean, RBPS currently requires some more code

Why use pyParticleEst?

- Implements the common parts of the algorithm, you save time and are less likely to introduce bugs
- Object-Oriented structure, should be easy to incorporate into your software
- Base-classes for common problem type(s)
  - Mixed Linear/Non-linear Gaussian
  - Differential Drive wheeled robotics
- You are working with Python, no other toolbox available
- Open Source license

Why not use pyParticleEst?

- Object-oriented structure might introduce unnecessary overhead for simple/performance critical problems
- You only work with linear Gaussian systems
- You don't want to work in Python
- Open Source license