PFLib – An Object Oriented MATLAB Toolbox for Particle Filtering

Lingji Chen^a, Chihoon Lee^b, Amarjit Budhiraja^b and Raman K. Mehra^a

^{a}Scientific Systems Company Inc., Woburn, MA, USA;
^{b}University of North Carolina at Chapel Hill, Chapel Hill, NC, USA

ABSTRACT

Under a United States Army Small Business Technology Transfer (STTR) project, we have developed a MATLAB toolbox called `PFLib` to facilitate the exploration, learning and use of Particle Filters by a general user. This paper describes its object oriented design and programming interface. The software is available under a GNU GPL license.

Keywords: Particle Filter, Extended Kalman Filter, software, MATLAB toolbox, object oriented, Graphical User Interface (GUI)

1. INTRODUCTION

Recent years have witnessed tremendous interest in nonlinear filtering techniques in general and particle filters in particular. Many algorithms have been reported in literature, and some are available in the form of pseudo code or even real source code. However, there does not seem to be a unified software package that contains major algorithms and options and is readily available. Practitioners who are interested in exploring the use of particle filters for their problems often only have the choices of gleaning the Web or writing the code from scratch, neither of which is very attractive.

It is true that one of the advantages of particle filters is their ease of implementation. Still writing code from scratch takes a non-trivial amount of time and resources. A routine that calculates the definite integral of a function is not difficult to implement, but we as users no longer write such a routine ourselves; we count on its availability in the scientific computing software that we use. Similar things should happen with particle filters.

Scientific Systems Company Inc. and University of North Carolina at Chapel Hill, under a United States Army Small Business Technology Transfer (STTR) project "Advanced Computational Algorithms for Nonlinear Filtering for Real Time Environment," have developed a MATLAB toolbox for particle filters that we call `PFLib`. Its main features include the following:

- The *implementation* of a filtering algorithm and its *use* are separated, as a result of an object oriented design. People who want to know how things work can examine the source code and make modifications if they choose to, but people who do not can simply use these filtering *objects* as black boxes.

- Major algorithms and options have been implemented: Extended Kalman Filter, Bootstrap Particle Filter, Particle Filter with an EKF-type "proposal distribution," Auxiliary Particle Filter, and Instrumental Variable Particle Filter. Resampling schemes include "None," "Simple Resampling," "Residual Resampling," "Branch-and-Kill," and "System Resampling." Other available parameter choices include sampling frequency, number of particles, and specification of Jacobians. Despite the variety, the Application Programming Interface (API) is consistent and easy to use. To switch from one filter to another, one only needs to change a few lines of code that initialize the filtering objects. The code that performs the actual filtering need not be changed.

- To further facilitate a general user, a Graphical User Interface (GUI) is also included, so that choices of filters with relevant options can be made in an intuitive fashion. Moreover, the GUI automatically generates initialization code according to a user's selection, so that the code can be cut-and-pasted into other simulation scripts.

- We are releasing the software under a GNU GPL license (send an email to Lingji.Chen@ssci.com for a copy), which means, among other things, that the source code is available and can be modified and distributed under the same license.

2. MATLAB CLASSES

According to MATLAB Helpdesk,[1] "Classes and objects enable you to add new data types and new operations to MATLAB. The *class* of a variable describes the structure of the variable and indicates the kinds of operations and functions that can apply to the variable. An *object* is an instance of a particular class. The phrase *object-oriented programming* describes an approach to writing programs that emphasizes the use of classes and objects."

For a simple introduction, we will illustrate the concept using a class in `PFLib`, the class `GaussianDistr` that represents a Gaussian distribution. We first describe how such a class is used, and then describe how it is implemented.

2.1. How `GaussianDistr` is used

First, we create a `GaussianDistr` object by passing to the *constructor* a mean and a covariance:

```
>> theMean=[0;0]; theCov=[3 2; 2 4];
>> g=GaussianDistr(theMean, theCov)

g =
        GaussianDistr object: 1-by-1
```

Now the variable `g` represents such an object, with both "data" and "methods." Suppose we want to calculate the probability density of such a distribution at the point $[1; 2]$. We simply call the `density` method that has been implemented for this class:

```
>> density(g, [1;2])

ans =

    0.0341
```

Here MATLAB examines the class type of `g`, the first argument to the function `density`, then looks for the function file `density.m` under the class directory `@GaussianDistr`, and finally executes that method.

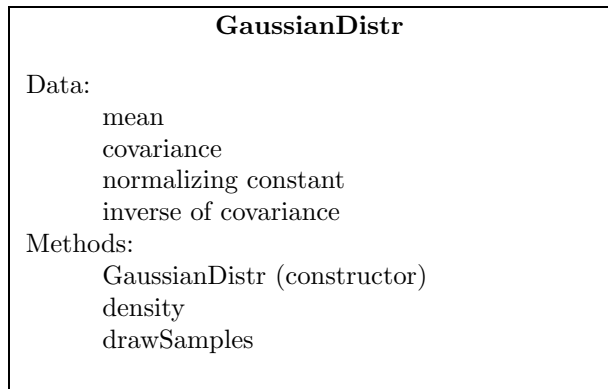If we want to draw 7 samples from such a distribution, we call the `drawSamples` method:

```
>> drawSamples(g, 7)
ans =

  -1.8740    0.5705   -2.3293    0.1448   -0.1243   -1.2140   -0.2409
   0.4130    1.9453   -1.1497    0.5987   -2.1232    0.5544    0.4513
```

Again the function file `drawSamples.m` is found under the class directory `@GaussianDistr` and is executed.

From a usage point of view, all we need to know is that these methods are available for this particular class. Next we will take a look at how the class is implemented in the first place.

GaussianDistr

Data:

       mean
       covariance
       normalizing constant
       inverse of covariance

Methods:

       GaussianDistr (constructor)
       density
       drawSamples

Figure 1. A class consists of data and methods.

```
function gDistr = GaussianDistr(theMean, theCov)

%%% error checking not shown

gDistr.mean = theMean(:);
gDistr.n = length(gDistr.mean);
gDistr.cov = theCov;
gDistr.const = 1 / sqrt((2 * pi) ^ gDistr.n * det(theCov));
gDistr.invCov = inv(theCov);

%%% other branches not shown

gDistr = class(gDistr, 'GaussianDistr');
```

Figure 2. The constructor of a class.

2.2. How `GaussianDistr` is implemented

First a subdirectory `@GaussianDistr` is created to store all files associated with this class. Conceptually the class is an encapsulated set of data and methods as shown in Figure 1.

The constructor takes two arguments passed to it and sets the mean and covariance in the data part. It also sets the normalizing constant used in the calculation of density, and the inverse of the covariance used in drawing samples. These quantities are stored in a structure and turned into a class by calling the MATLAB `class` function. Code snippets for the constructor and the density method are shown in Figures 2 and 3 for the benefit of readers interested in implementation.

The method `density` is vectorized, *i.e.*, we can pass a matrix whose columns represent the points at which we want the density values.

Note that the object `g` in the `density` method contains some data that were previously created by the class constructor, e.g., the normalizing constant `g.const` and the inverse of the covariance matrix of the distribution `g.invCov`.

```
function d = density(gDistr, x)

[nrow, ncol] = size(x);

%%% error checking not shown

d = zeros(1, ncol);
x = x - repmat(gDistr.mean, 1, ncol);

for i = 1:ncol
    d(i) = gDistr.const * exp(-x(:, i)' * gDistr.invCov * x(:, i) / 2);
end;
```

Figure 3. A method of a class.

3. PARTICLE FILTERS AS OBJECTS

We consider nonlinear discrete time systems with additive noises, in the following form[*]

$$
\begin{aligned}
x(k+1) &= f(x(k)) + v(k), \\
y(k) &= h(x(k)) + w(k),
\end{aligned}
\tag{1}
$$

where $x(k)$ is the state, $y(k)$ is the measurement, $v(k)$ and $w(k)$ are i.i.d. noises, and $k \in \mathbb{N}$.

From a simulation point of view, a filter is a dynamical system in the form

$$
\begin{aligned}
z(k) &= \eta(z(k-1), y(k)), \\
\hat{x}(k) &= \zeta(z(k), y(k)),
\end{aligned}
$$

where $z(k)$ is the internal state of the filter, $\hat{x}(k)$ is the estimate of the current state $x(k)$ based on the posterior distribution obtained by the filter, and the functions $\eta(\cdot)$ and $\zeta(\cdot)$ are parameterized by the functions $f(\cdot)$, $h(\cdot)$, and statistics of $x(0)$, $v(k)$ and $w(k)$. In other words, a filtering subsystem, for a given system (1), takes in an observation $y(k)$ and produces an estimate $\hat{x}(k)$, based upon its internal state $z(k)$, and the particular filtering algorithms used. In the case of Extended Kalman Filter, $z(k)$ includes the estimated mean and covariance. In the case of a Particle Filter, $z(k)$ includes all the particles and their weights, among others.

It therefore follows that we can use a MATLAB object to represent such a filter. An illustration of the PF_Simple class in PFLib is shown in Figure 4.

To use a filtering object, first we create one by choosing a desired class and properly initializing it:

```
>> aFilterObj = Particle_Filter_Class(initializing_parameters ...)
```
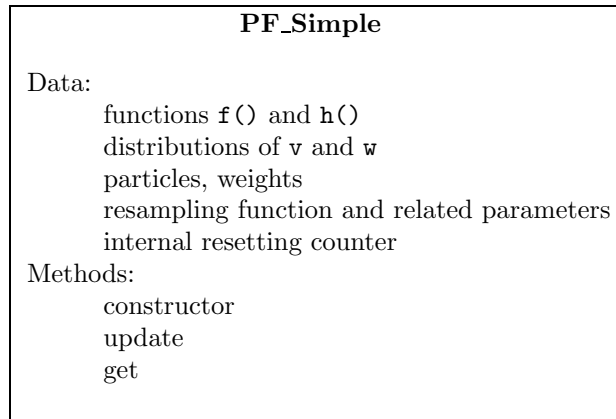
Then we can simply use it as follows:

```
>> for k = 1:Tmax
>> %%% other code omitted
>> aFilterObj = update(aFilterObj, y(k), ...);
>> xhat(k) = get(aFilterObj);
>> end
```

Here in each iteration, the filter object is updated with the new measurement. The get function can return particles and weights, as well as the state estimate.

---

[*]Particle filtering algorithms can be applied to systems of a more general form, but the implementation of a generic *toolbox* would be considerably harder, since systems will be specified by users at run time.

```
                    PF_Simple

Data:
        functions f() and h()
        distributions of v and w
        particles, weights
        resampling function and related parameters
        internal resetting counter
Methods:
        constructor
        update
        get
```

Figure 4. What's in a particle filter.

4. THE GRAPHICAL USER INTERFACE

We have implemented the following categories of filters:

- the Extended Kalman Filter,

- the Bootstrap Particle Filter,

- the Particle Filter with an EKF-type "proposal distribution,"

- the Auxiliary Particle Filter,

- the Instrumental Variable Particle Filter.

Resampling scheme choices that we have implemented include (i) None (for pedagogical purpose), (ii) Simple Resampling, (iii) Residual Resampling, (iv) Branch-and-Kill, and (v) System Resampling.

Other available choices include (i) sampling frequency (e.g., every 5 steps), (ii) number of particles, and (iii) Jacobians.

A partial list of references is.[2–6]

Given the above options, initializing a particle filter object can become tedious – one would have to read the documentation and understand how each option is specified. To ease the user's burden, we have created a Graphical User Interface that first lets the user pick and choose and then generates the initialization code for the user. To illustrate, two screen shots are shown in Figures 5 and 6, with the former showing how information about the system is collected, and the latter, the filter. (In the latter, since "filter type" has been chosen in the drop-down box on the upper-left corner, only choices relevant to this filter type is shown, hence the empty middle-part.)

After the user has made all the choices, initialization code will be shown in an editor window, as is displayed in Figure 7.

5. EXTENDING PFLIB

It is easy to extend PFLib to include more or new filtering algorithms. To give interested readers an idea of the encapsulated implementation, the update method for a simple bootstrap filter is listed in Figure 8. To implement a new algorithm, its author can use the code for the bootstrap filter in PFLib as a template, and change the update function to perform the calculations specified by the new algorithm. As far as the users are concerned, they have one more choice in the library, and can use it the same way as any other algorithm.

Figure 5. Collecting information on the system.

Figure 6. Collecting information on filter choices.

ACKNOWLEDGMENTS

Figure 7. GUI-generated initialization code

REFERENCES

1. *MATLAB Classes and Objects (Programming and Data Types)*. http://www.mathworks.com/access/helpdesk_r13/help/techdoc/matlab_prog/ch14_oop.html.

2. A. Doucet, J. de Freitas, and N. Gordon, *Sequential Monte Carlo Methods in Practice*, New York: SpringerVerlag, 2001.

3. M. Arulampalam, S. Maskell, N. Gordon, and T. Clapp, "A tutorial on particle filters for online nonlinear/non-gaussian bayesian tracking," *IEEE Transactions on Signal Processing* **50**(2), pp. 174–188, 2002.

4. J. S. Liu and R. Chen, "Sequential Monte Carlo methods for dynamic systems," *Journal of the American Statistical Association* **93**(443), pp. 1032–1044, 1998.

5. N. de Freitas, "Sequential monte carlo methods homepage." http://www.cs.ubc.ca/~nando/smc/index.html.

6. A. Budhiraja, L. Chen, and C. Lee, "A survey of numerical methods for nonlinear filtering problems," *Physica D* , 2006. doi:10.1016/j.physd.2006.08.015.

```
function fltr = update(fltr, y)

% function fltr = update(fltr, y)
% input:
% fltr:  the filter object
% y :   the current measurement
% output:
% fltr:  the updated filter object
% % Author:  Chihoon Lee, Lingji Chen
% Date :   Jan 20, 2006

N = length(fltr.w);

% check whether resampling is chosen, and whether it's time to resample
if ~isempty(fltr.r_fun) && mod(fltr.counter, fltr.T) == 0
    if(N < fltr.thresh * fltr.N ) % too few (as a result of branch-kill), rejuvenate
        outIndex = fcn_ResampSys(fltr.w, fltr.N);
    else % user-specified resampling scheme
        outIndex = fltr.r_fun(fltr.w);
    end;
    fltr.p = fltr.p(:, outIndex);
    N = length(outIndex);
    fltr.w = ones(1, N) / N;
end;

% internally keep track of when it's time to resample
fltr.counter = fltr.counter + 1;

w_smpl = drawSamples(fltr.w_d, N);
for i = 1:N
    % propagate particles
    fltr.p(:, i) = fltr.f(fltr.p(:, i)) + w_smpl(:, i);
    % noise-free y
    y_pi = fltr.h(fltr.p(:, i));
    % update weights
    fltr.w(i) = fltr.w(i) * density(fltr.v_d, y - y_pi);
end;

sum_w = sum(fltr.w);
if sum_w <= realmin
    error('weights are numerically zero; change parameters or method.');
end;

fltr.w = fltr.w / sum_w;
```

Figure 8. The update method of a simple particle filter.