Contents

Notation

Following notation is used throughout this text:

$\mathbb{N}$	set of natural numbers excluding zero
$\mathbb{N}_0$	set of natural numbers including zero
$\mathbb{R}$	set of real numbers
t	discrete time moment; $t \in \mathbb{N}_0$
a_t	value of quantity a at time t; $a_t \in \mathbb{R}^n, n \in \mathbb{N}$
$a_{t\|t'}$	quantity with two indices: t and t' there is no implicit link between a_t, $a_{t\|t'}$ and $a_{t'}$
$a_{t:t'}$	sequence of quantities $(a_t, a_{t+1}, \ldots, a_{t'-1}, a_{t'})$
$p(a_t)$	probability density function[1] of quantity a at time t (unless noted otherwise)
$p(a_t\|b_{t'})$	conditional probability density function of quantity a at time t given value of quantity b at time t'
$\delta(a)$	Dirac delta function; used exclusively in context of probability density functions to denote discrete distribution within framework of continuous distributions[2]
$\mathcal{N}(\mu, \Sigma)$	multivariate normal (Gaussian) probability density function with mean vector μ and covariance matrix Σ

[1] for the purpose of this text, probability density function p is multivariate non-negative function $\mathbb{R}^n \to \mathbb{R}$; $\int_{\operatorname{supp} p} p(x_1, x_2, \ldots, x_n)\, \mathrm{d}x_1 \mathrm{d}x_2 \cdots \mathrm{d}x_n = 1$

[2] so that $\int_{-\infty}^{\infty} f(x)\delta(x - \mu)\, \mathrm{d}x = f(\mu)$ and more complex expressions can be derived using integral linearity and Fubini's theorem.

Introduction

Bayesian filtering (or, recursive Bayesian estimation) is a very promising approach to estimation of dynamic systems; it can be applied to a wide range of real-world problems in areas such as robotics [17, 6] (tracking, navigation), environmental simulations — e.g. tracking of the radioactive plume upon nuclear accident [8, 7, 10], econometrics and many more.

While many Bayesian filtering algorithms are simple enough to be implemented in software on *ad-hoc* basis, it is proposed that a well designed library can bring many advantages such as ability to combine and interchange individual methods, better performance, programmer convenience and a tested code-base.

The text is organised as follows:

1. Theoretical background of Bayesian filtering is presented in the first chapter along with a description of well-known Bayesian filters: the Kalman filter, the particle filter and a simple form of the marginalized particle filter.
2. In chapter 2 a software analysis for a desired library for Bayesian filtering is performed: requirements are set up, general approaches to software development are discussed and programming languages C++, MATLAB and Python and their implementations are compared. Interesting results are achieved when Python is combined with Cython.
3. The PyBayes library that was developed as a part of this thesis is presented. PyBayes is written in Python with an option to use Cython and implements all algorithms presented in the first chapter. Performance of PyBayes is measured and confronted with concurrent implementations that use different implementation environments.

Bayesian filtering is a subtask of a broader topic of Bayesian decision-making [18]; while decision-making is not covered in this text, we expect the PyBayes library to form a good building block for implementing decision-making systems.

Chapter 1

Basics of Recursive Bayesian Estimation

In following sections the problem of recursive Bayesian estimation (Bayesian filtering) is stated and its analytical solution is derived. Later on, due to practical intractability of the solution in its general form, a few methods that either simplify the problem or approximate the solution are shown.

1.1 Problem Statement

Assume a dynamic system described by a hidden real-valued *state vector* x which evolves at discrete time steps according to a known function f_t (in this text called *process model*) as described by (1.1).

$$x_t = f_t(x_{t-1}, v_{t-1}) \tag{1.1}$$

Variable v_t in (1.1) denotes random *process noise*, which may come from various sources and is often inevitable. Sequence of v_t is assumed to be identically independently distributed random variable sequence.

The state of the system is hidden and can only be observed though a real-valued *observation vector* y that relates to the state x as in (1.2), but adds further *observation noise* w.

$$y_t = h_t(x_t, w_t) \tag{1.2}$$

In (1.2) h_t is known function called *observation model* in this text and w_t is identically independently distributed random variable sequence that denotes observation noise.

The goal of recursive[1] Bayesian estimation is to give an estimate of the state x_t

[1] by the word recursive we mean that it is not needed to keep track of the whole batch of previous observations in practical methods, only appropriate quantities from time moments $t-1$ and t are needed to estimate x_t. However, this does not apply to the derivation of the solution, where the notation of whole batch of observations $y_{1:t}$ is used.

given the observations $y_{1:t}$ provided the knowledge of the functions f_t and h_t. More formally, the goal is to find the probability density function $p(x_t|y_{1:t})$. Theoretical solution to this problem is known and is presented in next section.

1.2 Theoretical solution

At first, we observe that probability density function $p(x_t|x_{t-1})$ can be derived from the process model (1.1) (given the distribution of v_k) and that $p(y_t|x_t)$ can be derived from the observation model (1.2) respectively. (given the distribution of w_k)

Because recursive solution is requested, suppose that $p(x_{t-1}|y_{1:t-1})$ and $p(x_0)$ are known[2] in order to be able to make the transition $t-1 \rightarrow t$.

In the first stage that can be called *prediction*, *prior* probability density function $p(x_t|y_{1:t-1})$ is calculated without knowledge of y_t. We begin the derivation by performing the reverse of the marginalization over x_{k-1}.

$$p(x_t|y_{1:t-1}) = \int_{-\infty}^{\infty} p(x_t, x_{t-1}|y_{1:t-1}) \, \mathrm{d}x_{t-1}$$

Using chain rule for probability density functions, the element of integration can be split.

$$p(x_t|y_{1:t-1}) = \int_{-\infty}^{\infty} p(x_t|x_{t-1}, y_{1:t-1}) p(x_{t-1}|y_{1:t-1}) \, \mathrm{d}x_{t-1}$$

With an assumption that the modelled dynamic system (1.1) possesses *Markov Property*[3], $p(x_t|x_{t-1}, y_{1:t-1})$ equals $p(x_t|x_{t-1})$. [1] This leaves us with the result (1.3).

$$p(x_t|y_{1:t-1}) = \int_{-\infty}^{\infty} p(x_t|x_{t-1}) p(x_{t-1}|y_{1:t-1}) \, \mathrm{d}x_{t-1} \tag{1.3}$$

As we can see, prior probability density function only depends on previously known functions and therefore can be calculated.

We continue with the second stage that could be named *update*, where new observation y_t is taken into account and *posterior* probability density function $p(x_t|y_{1:t})$ is calculated. Bayes' theorem can be used to derive posterior probability density function (1.4).

$$p(x_t|y_{1:t}) = \frac{p(y_t|x_t, y_{1:t-1}) p(x_t|y_{1:t-1})}{p(y_t|y_{1:t-1})} \tag{1.4}$$

According to the observation model (1.2) and assuming Markov property, y_t only depends on x_t. That is $p(y_t|x_t, y_{1:t-1}) = p(y_t|x_t)$. Therefore posterior probability density function can be further simplified into (1.5).

$$p(x_t|y_{1:t}) = \frac{p(y_t|x_t) p(x_t|y_{1:t-1})}{p(y_t|y_{1:t-1})} \tag{1.5}$$

[2] $p(x_0)$ can be called initial probability density function of the state vector.

[3] an assumption of independence that states that system state in time t only depends on system state in $t-1$ (and is not directly affected by previous states).

While both probability density functions in the numerator of (1.5) are already known, $p(y_t|y_{1:t-1})$ found in the denominator can be calculated using the formula (1.6), where marginalization over x_t is preformed. Quantity (1.6) can also be interpreted as *marginal likelihood* (sometimes called *evidence*) of observation. [15]

$$p(y_t|y_{1:t-1}) = \int_{-\infty}^{\infty} p(y_t|x_t)p(x_t|y_{1:t-1}) \, \mathrm{d}x_t \tag{1.6}$$

Computing (1.6) isn't however strictly needed as it does not depend on x_t and serves as a normalising constant in (1.5). Depending on use-case the normalising constant may not be needed at all or may be computed alternatively using the fact that $p(x_t|y_{1:y})$ integrates to 1.

We have shown that so called *optimal Bayesian solution*[1] can be easily analytically inferred using only *chain rule for probability density functions*, *marginalization* and *Bayes' theorem*. (equations (1.3), (1.5) and (1.6) forming the main steps of the solution) On the other hand, using this method directly in practice proves difficult because at least one parametric multidimensional integration has to be performed (in (1.3)), which is (in its general form) hardly tractable for greater than small state vector dimensions.

This is a motivation for various simplifications and approximations among which we have chosen a Kalman filter described in the next section and a family of particle filters described later.

1.3 Kalman Filter

The Kalman filter[4] poses additional set of strong assumptions on modelled dynamic system, but greatly simplifies the optimal Bayesian solution (1.3), (1.5) into a sequence of algebraic operations with matrices. On the other hand, when these requirements can be fulfilled, there is no better estimator in the Bayesian point of view because the Kalman filter computes $p(x_t|y_{1:t})$ *exactly*.[5]

Assumptions additionally posed on system by the the Kalman filter are:

1. f_t in the process model (1.1) is a linear function of x_t and v_t.
2. $v_t \sim \mathcal{N}(0, Q_t)$ meaning that process noise v_t is normally distributed with zero mean[6] and with known covariance matrix Q_t.
3. h_t in the observation model (1.2) is a linear function of x_t and w_t.
4. $w_t \sim \mathcal{N}(0, R_t)$ meaning that observation noise w_t is normally distributed with zero mean and with known covariance matrix R_t.
5. initial state probability density function is Gaussian.

It can be proved that if the above assumptions hold, $p(x_t|y_{1:t})$ is Gaussian for all $t > 0$. [11] Furthermore, given assumptions 1. and 2. the process model (1.1) can be

[4] first presented by Rudolf Emil Kalman in 1960.
[5] not accounting for numeric errors that arise in practical implementations.
[6] zero mean assumption is not strictly needed, it is however common in many implementations.

reformulated as (1.7), where A_t is real-valued matrix that represents f_t. Using the same idea and assumptions 3. and 4. the observation model (1.2) can be expressed as (1.8), C_t being real-valued matrix representing h_t. Another common requirement used below in the algorithm description is that v_t and w_t are stochastically independent.

$$x_t = A_t x_{t-1} + \hat{v}_{t-1} \qquad A_t \in \mathbb{R}^{n,n} \quad n \in \mathbb{N} \tag{1.7}$$

$$y_t = C_t x_t + \hat{w}_t \qquad C_t \in \mathbb{R}^{j,n} \quad j \in \mathbb{N} \quad j \leq n \tag{1.8}$$

Note that we have marked the noises v_t and w_t as $\hat{v}_t$ and $\hat{w}_t$ when they are transformed through A_t, respectively C_t matrix. Let also $\hat{Q}_t$ denote the covariance matrix of $\hat{v}_t$ and $\hat{R}_t$ denote the covariance matrix of $\hat{w}_t$ in further text.

At this point we can describe the algorithm of the Kalman filter. As stated above, posterior probability density function is Gaussian and thus can be parametrised by mean vector μ and covariance matrix P. Let us denote posterior mean from previous iteration by $\mu_{t-1|t-1}$ and associated covariance by $P_{t-1|t-1}$ as in (1.9).

$$p(x_{t-1}|y_{1:t-1}) = \mathcal{N}(\mu_{t-1|t-1}, P_{t-1|t-1}) \tag{1.9}$$

Prior probability density function (1.10) can then be calculated as follows: [1]

$$\begin{aligned} p(x_t|y_{1:t-1}) &= \mathcal{N}(\mu_{t|t-1}, P_{t|t-1}) \\ \mu_{t|t-1} &= A_t \mu_{t-1|t-1} \\ P_{t|t-1} &= A_t P_{t-1|t-1} A_t^T + \hat{Q}_{t-1} \end{aligned} \tag{1.10}$$

Before introducing posterior probability density function it is useful to establish another Gaussian probability density function (1.11) that is not necessarily needed, but is useful because it represents marginal likelihood (1.6).

$$\begin{aligned} p(y_t|y_{1:t-1}) &= \mathcal{N}(\nu_{t|t-1}, S_{t|t-1}) \\ \nu_{t|t-1} &= C_t \mu_{t|t-1} \\ S_{t|t-1} &= C_t P_{t|t-1} C_t^T + \hat{R}_t \end{aligned} \tag{1.11}$$

The update phase of the Kalman filter can be performed by computing so-called *Kalman gain* matrix (1.12), posterior probability density function (1.13) is then derived from prior one using the Kalman gain K_t and observation y_t. [1]

$$K_t = P_{t|t-1} C_t^T S_{t|t-1}^{-1} \tag{1.12}$$

$$\begin{aligned} p(x_t|y_{1:t}) &= \mathcal{N}(\mu_{t|t}, P_{t|t}) \\ \mu_{t|t} &= \mu_{t|t-1} + K_t(y_t - \nu_{t|t-1}) \\ P_{t|t} &= P_{t|t-1} - K_t C_t P_{t|t-1} \end{aligned} \tag{1.13}$$

In all formulas above A^T denotes a transpose of matrix A and A^{-1} denotes inverse matrix to A. As can be seen, formulas (1.3) and (1.5) have been reduced to tractable algebraic operations, computing inverse matrix[7] being the most costly one.

[7] it can be shown that $S_{t|t-1}$ is positive definite given that C_t is full-ranked, therefore the inverse in (1.12) exists.

It should be further noted that the Kalman filter and described algorithm can be easily enhanced to additionally cope with an *intervention* (or control) vector applied to the system, making it suitable for the theory of decision-making. Numerous generalisations of the Kalman filter exist, for example an *extended Kalman filter* that relaxes the requirement of the linear system by locally approximating a non-linear system with Taylor series. These are out of scope of this text, but provide areas for subsequent consideration.

On the other hand, the assumption of Gaussian posterior probability density function cannot be easily overcome and for systems that show out non-Gaussian distributions of the state vector another approach have to be taken. [1] One such approach can be a Monte Carlo-based *particle filter* presented in the next section.

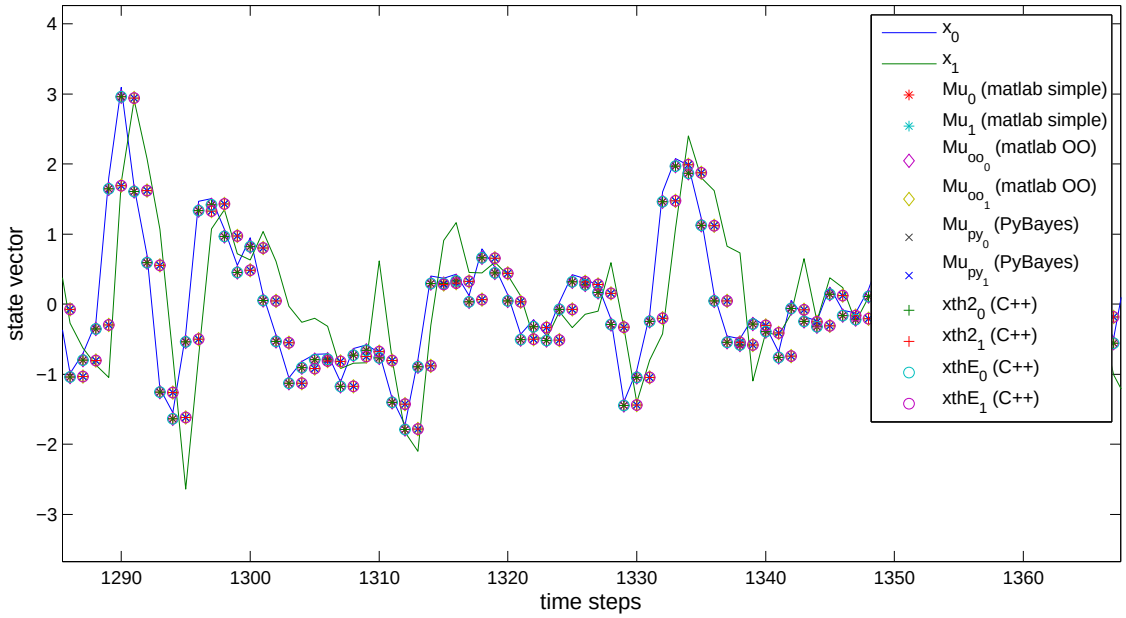

Figure 1.1: Example run of the Kalman filter. Lines are actual (hidden) state, dots estimation means of various implementations (all yielding the same values).

1.4 Particle Filter

Particle filters represent an approximate solution of the problem of the recursive Bayesian estimation, thus can be considered *suboptimal* methods. The underlying algorithm described below is most commonly named *sequential importance sampling (SIS)*. The biggest advantage of the particle filtering is that requirements posed on the modelled system are much weaker than those assumed by optimal methods such as the Kalman filter. Simple form of the particle filter presented in this section (that assumes that modelled system has Markov property) requires only the knowledge of probability density function $p(x_t|x_{t-1})$ representing the process model and the

knowledge of $p(y_t|x_t)$ representing the observation model.[8]

The sequential importance sampling approximates the posterior density by a weighted empirical probability density function (1.14).

$$p(x_t|y_{1:t}) \approx \sum_{i=1}^{N} \omega_t^{(i)} \delta(x_t - x_t^{(i)}) \tag{1.14}$$

$$\forall i \in \mathbb{N} \;\; i \leq N : \omega_i \geq 0 \qquad \sum_{i=1}^{N} \omega_i = 1$$

In (1.14) $x_t^{(i)}$ denotes value of i-th *particle*: possible state of the system at time t; $\omega_t^{(i)}$ signifies weight of i-th particle at time t: scalar value proportional to expected probability of the system being in state in small neighbourhood of $x_t^{(i)}$; N denotes total number of particles[9], a significant tunable parameter of the filter.

As the initial step of the described particle filter, N random particles are sampled from the initial probability density function $p(x_0)$. Let $i \in \mathbb{N} \;\; i \leq N$, transition $t-1 \;\rightarrow\; t$ can be performed as follows:

1. for each i compute $x_t^{(i)}$ by random sampling from conditional probability density function $p(x_t|x_{t-1})$ where $x_{t-1}^{(i)}$ substitutes x_{t-1} in condition. This step can be interpreted as a simulation of possible system state developments.
2. for each i compute weight $\omega_t^{(i)}$ using (1.15) by taking observation y_t into account. x_t is substituted by $x_t^{(i)}$ in condition in (1.15). Simulated system states are confronted with reality through observation.

$$\omega_t^{(i)} = p(y_t|x_t)\omega_{t-1}^{(i)} \tag{1.15}$$

3. normalise weights according to (1.16) so that approximation of posterior probability density function integrates to one.

$$\omega_t^{(i)} = \frac{\omega_t^{(i)}}{\sum_{j=1}^{N} \omega_t^{(j)}} \tag{1.16}$$

Relative computational ease of described algorithm comes with cost: first, the particle filter is in principle non-deterministic because of the random sampling in step 1, in other words, the particle filter is essentially a Monte Carlo method; second, appropriate number of particles N has to be chosen — too small N can lead to significant approximation error while inadequately large N can make the particle filter infeasibly time-consuming. It can be proved that the particle filter converges to true posterior density as N approaches infinity and certain other assumptions hold [4], therefore the number of particles should be chosen as a balance of accuracy and speed.

[8] both probability density functions are generally time-varying and their knowledge for all t is needed, but their representation (parametrised by conditioning variable) is frequently constant in time in practical applications.

[9] N is assumed to be arbitrary but fixed positive integer for our uses. Variants of the particle filter exist that use adaptive number of particles, these are not discussed here.

Only two operations with probability density functions were needed: sampling from $p(x_t|x_{t-1})$ and evaluating $p(y_t|x_t)$ in known point. Sometimes sampling from $p(x_t|x_{t-1})$ is not feasible[10] and/or better results are expected by taking an observation y_t into account during sampling (step 1). This can be achieved by introducing so-called *proposal density* (sometimes *importance density*) $q(x_t|x_{t-1}, y_t)$. Sampling in step 1 then uses $q(x_t|x_{t-1}, y_t)$ instead, where x_{t-1} in condition is substituted by $x_{t-1}^{(i)}$. Weight computation in step 2 have to be replaced with (1.17) that compensates different sampling distribution (every occurrence of x_t, x_{t-1} in the mentioned formula has to be substituted by $x_t^{(i)}$ and $x_{t-1}^{(i)}$ respectively). See [1] for a derivation of these formulas and for a discussion about choosing adequate proposal density.

$$\omega_t^{(i)} = \frac{p(y_t|x_t)p(x_t|x_{t-1})}{q(x_t|x_{t-1}, y_t)}\omega_{t-1}^{(i)} \tag{1.17}$$

Particle filters also suffer from a phenomenon known as *sample impoverishment* or *degeneracy problem*: after a few iterations all but one particles' weight falls close to zero.[11] One technique to diminish this is based on careful choice of proposal density (as explained in [1]), a second one is to add additional *resample* step to the above algorithm:

4. for each i resample $x_t^{(i)}$ from approximate posterior probability density function $\sum_{i=1}^{N} \omega_t^{(i)} \delta(x_t - x_t^{(i)})$ and reset all weights to $\frac{1}{N}$. Given that sampling is truly random and independent this means that each particle is in average copied n_i times, where n_i is roughly proportional to particle weight: $n_i \approx \omega_t^{(i)} N$. Statistics of posterior probability density function are therefore (roughly and on average) maintained while low-weight particles are eliminated.

Step 4 therefore facilitates avoidance of particles with negligible weight by replacing them with more weighted ones. Such enhanced algorithm is known as *sequential importance resampling (SIR)*.

Because particle resampling is computationally expensive operation, a technique can be used where resampling is skipped in some iterations, based on the following idea: a measurement of degeneracy can be obtained by computing an approximate of *effective sample size* N_{eff} at given time t using (1.18). [1]

$$N_{\text{eff}} \approx \left(\sum_{i=1}^{N} \left(\omega_t^{(i)} \right)^2 \right)^{-1} \tag{1.18}$$

Very small N_{eff} compared to N signifies a substantial loss of "active" particles, which is certainly undesirable as it hurts accuracy while leaving computational demands unchanged. Step 4 is then performed only when N_{eff} falls below certain threshold.

Recursive Bayesian estimation using SIR methods can be applied to a wide range of dynamic systems (even to those where more specialised methods fail) and can be tuned with number of particles N and proposal density q. On the other hand a method specially designed for a given system easily outperforms general particle filter in terms of speed and accuracy.

[10] but can be replaced by evaluation in known point.

[11] it has been shown that variance of particle weights continually raises as algorithm progresses. [1]

1.5 Marginalized Particle Filter

Main sources of this section are [12] and [13].

The marginalized particle filter (sometimes *Rao-Blackwellized particle filter*) is an extension to the particle filter that more accurately approximates the optimal Bayesian solution provided that the probability density function representing the process model $p(x_t|x_{t-1})$ can be obtained in a special form. Suppose that the state vector can be divided into two parts a_t and b_t (1.19) and that the process model probability density function can be expressed as a product of two probability density functions (1.20), where $p(a_t|a_{t-1}b_t)$ is analytically tractable (in general). We present a simple variant of the marginalized particle filter where, given b_t, process and observation model of the a_t part are linear with normally-distributed noise. The Kalman filter can be used to estimate a_t part of the state vector in this case.

$$x_t = (a_t, b_t) \tag{1.19}$$

$$p(x_t|x_{t-1}) = p(a_t, b_t|a_{t-1}, b_{t-1}) = p(a_t|a_{t-1}, b_t)p(b_t|b_{t-1}) \tag{1.20}$$

The posterior probability density function (1.21) can be represented as a product of a weighted empirical distribution and a normal distribution. Each (i-th) particle is thus associated with its Kalman filter (representing $a_t^{(i)}$ part) and $b_t^{(i)}$ quantity.

$$p(a_t, b_t|y_t) = \sum_{i=1}^{N} \omega_i p(a_t|y_{1:t}, b_{1:t}^{(i)})\ \delta(b_t - b_t^{(i)}) \tag{1.21}$$

In (1.21) N denotes the total number of particles, ω_i denotes weight of i-th particle, $p(a_t|y_{1:t}, b_{1:t}^{(i)})$ is posterior probability density function of i-th Kalman filter and $b_t^{(i)}$ is a value of the b_t part of i-th particle.

The algorithm of the described variant of the marginalized particle filter follows, note the similarities with the ordinary particle filter: at first, generate N b_t random samples from the initial distribution $p(b_0)$. Then the following procedure can be repeated for each measurement y_t:

1. for each i compute $b_t^{(i)}$ by random sampling from conditional probability density function $p(b_t|b_{t-1})$ where $b_{t-1}^{(i)}$ substitutes b_{t-1} in condition.
2. for each i compute the posterior probability density function $p(a_t|y_{1:t}, b_{1:t}^{(i)})$ of the i-th Kalman filter using y_t and $b_t^{(i)}$.
3. for each i update weight ω_i using the formula (1.22) where marginal likelihood of the i-th Kalman filter (1.11) is used.

$$\omega_i = p(y_t|y_{1:t-1}, b_{1:t}^{(i)})\ \omega_i \tag{1.22}$$

4. normalise weights (as described in the previous section).
5. resample particles (as described in the previous section).

It has been demonstrated in various publications that the marginalised particle filter outperforms the ordinary particle filter in both better accuracy and lower computational demands. Where applicable, it therefore forms an excellent alternative to traditional particle filtering.

Chapter 2

Software Analysis

In this chapter, general software development approaches and practices will be confronted with requirements posed on the desired software library for recursive Bayesian estimation. After stating these requirements, feasibility of various programming paradigms applied to our real-world problem is discussed. Continues a comparison of suitable features of 3 chosen programming languages: C++, MATLAB language and Python. Emphasis is put on the Python/Cython combination that was chosen for implementation.

In whole chapter, the term *user* refers to someone (a programmer) who uses the library in order to implement higher-level functionality (such as simulation of dynamic systems).

2.1 Requirements

Our intended audience is a broad scientific community interested in the field of the recursive Bayesian estimation and decision-making. Keeping this in mind and in order to formalise expectations for the desired library for Bayesian filtering, the following set of requirements was developed.

Functionality:

- Framework for working with potentially conditional probability density functions should be implemented including support for basic operations such as product and chain rule. The chain rule implementation should be flexible in a way that for example $p(a_t, b_t|a_{t-1}, b_{t-1}) = p(a_t|a_{t-1}, b_t)p(b_t|b_{t-1})$ product can be represented.
- Basic Bayesian filtering methods such as the Kalman and particle filter have to be present, plus at least one of more specialised algorithms — a marginalized particle filter or non-linear Kalman filter variants.

General:

- Up-to-date, complete and readable API[1] documentation is required. Such docu-

[1] Application Programming Interface, a set of rules that define how a particular library is used.

mentation should be well understandable by someone that already understands mathematical background of the particular algorithm.
- High level of interoperability is needed; data input/output should be straightforward as well as using existing solutions for accompanying tasks such as visualising the results.
- The library should be platform-neutral and have to run on major server and workstation platforms, at least on Microsoft Windows and GNU/Linux.
- The library should be Free/Open-source software as it is believed by the authors that such licensing/development model results in software of greatest quality in long term. Framework used by the library should make it easy to adapt and extend the library for various needs.

Usability:

- Initial barriers for installing and setting up the library should be lowest possible. For example a necessity to install third-party libraries from sources is considered infeasible.
- Implementation environment used for the library should allow for high programmer productivity; prototyping new solutions should be a quick and cheap (in terms of effort) operation. This requirement effectively biases towards higher-level programming languages.

Performance:

- Computational overhead[2] should be kept reasonably low.
- Applications built atop of the library should be able to scale well on multi-processor systems. This can be achieved for example by thread-safety of critical library objects or by explicit parallelisation provided by the library.

It is evident that some of the requirements are antagonistic, most prominent example being demand for *low computational overhead* while still offering *high programmer productivity* and rapid prototyping. The task of finding tradeoffs between contradictory tendencies or developing smart solutions that work around traditional limitations is left upon the implementations.

2.2 Programming paradigms

Many programming paradigms exist and each programming language usually suggests a particular paradigm, though many languages let programmers choose from or combine multiple paradigms. This section discusses how well could be three most prominent paradigms (procedural, object-oriented and functional) applied to the software library for Bayesian filtering. Later on additional features of implementation environments such as interpreted vs. compiled approach or argument passing convention are evaluated.

[2] excess computational costs not directly involved in solving particular problem; for example interpreter overhead.

2.2.1 Procedural paradigm

The procedural paradigm is the traditional approach that appeared along the first high-level programming languages. The procedural programming can be viewed as a structured variant of imperative programming, where programmer specifies steps (in form of orders) needed to reach desired program state. Structured approach that emphasizes dividing the code into logical and self-contained blocks (procedures, modules) is used to make the code more reusable, extensible and modular. Today's most notable procedural languages include C and Fortran.

Most procedural languages are associated with very low overhead (performance of programs compiled using optimising compiler tend to be very close to ideal programs written in assembly code); mentioned languages are also spread and well-known in scientific computing.

On the other hand, while possible, it is considered an elaborate task by the author to write a modular and extensible library in these languages. Another disadvantage is that usually only very basic building blocks are provided by the language — structures like lists and strings have to be supplied by the programmer or a third-party library. This only adds to the fact that the procedural paradigm-oriented languages are commonly not easy to learn and that programmer productivity associated with these languages may be much lower compared to more high-level languages.

2.2.2 Object-oriented paradigm

The object-oriented paradigm extends the procedural approach with the idea of *objects* — structures with procedures (called *methods*) and variables (called *attributes*) bound to them. Other feature frequently offered is *polymorphism* (an extension to language's type system that adds the notion of *subtypes* and a rule that subtype of a given type can be used everywhere where given type can be used) most often facilitated through a concept of *classes*, common models for sets of objects with same behaviour but different payload; objects are then said to be *instances* of classes. A subclass *inherits* methods and attributes from its superclass and can *override* them or add its own. *Encapsulation*, a language mechanism to restrict access to certain object attributes and methods, may be employed by the language to increase robustness by hiding implementation details. In order to be considered object-oriented, statically typed languages (p. 14) should provide *dynamic dispatch*[3], en essential complement to polymorphism, for certain or all object methods.

Notable examples of languages that support (although not exclusively) object-oriented paradigm are statically typed C++, Java and dynamically typed (p. 14) MATLAB language, Python, Smalltalk.

Object-oriented features typically have very small overhead compared to procedural code with equal functionality, so additional complexity introduced is the only downside, in author's opinion. We believe that these disadvantages are greatly out-

[3] a way of calling methods where the exact method to call is resolved at runtime based on actual (dynamic) object type (in contrast to static object type).

weighed by powerful features that object-oriented languages provide (when utilised properly).

It was also determined that the desired library for Bayesian filtering could benefit from many object-oriented techniques: probability density function and its conditional variant could be easily modelled as classes with abstract methods that would represent common operation such as evaluation in a given point or drawing random samples. Classes representing particular probability density functions would then subclass abstract base classes and implement appropriate methods while adding relevant attributes such as border points for uniform distribution. This would allow for example to create generic form of particle filter (p. 6) that would accept any conditional probability density function as a parameter. Bayesian filter itself can be abstracted into a class that would provide a method to compute posterior probability density function from prior one taking observation as a parameter.

2.2.3 Functional paradigm

Fundamental idea of the functional programming is that functions have no side effects — their result does not change or depend on program state, only on supplied parameters. A language where each function has mentioned attribute is called *purely functional* whereas the same adjective is applied to such functions in other languages. This is often accompanied by a principle that all data are immutable (apart from basic list-like container type) and that functions are so-called "first-class citizens" — they can be passed to a function and returned. Placing a restriction of no side-effect on functions allows compiler/interpreter to do various transformations: parallelisation of function calls whose parameters don't depend on each other's results, skipping function calls where the result is unused, caching return values for particular parameters.

Among languages specially designed for functional programming are: Haskell, Lisp dialects Scheme and Clojure, Erlang. Python supports many functional programming techniques[4].

While functional programming is popular subject of academic research, its use is much less widespread compared to procedural and object-oriented paradigms. Additionally, in the author's opinion, transition to functional programming requires significant change of programmer's mindset. Combined with the fact that syntax of the mentioned functionally-oriented languages differs significantly from many popular procedural or object-oriented languages, we believe that it would be unsuitable decision for a library that aims for wide adoption.

2.2.4 Other programming language considerations

Apart from recently discussed general approaches to programming, we should note a few other attributes of languages or their implementations that significantly affect

[4] e.g. functions as first-class citizens, closures, list comprehensions

software written using them. The first distinction is based on type system of a language — we may divide them into 2 major groups:

statically typed languages
: bind object types to *variables*; vast majority of type-checking is done at compile-time. This means that each variable can be assigned only values of given type (subject to polymorphism); most such languages require that variable (function parameter, object attribute) types are properly declared.

dynamically typed languages
: bind object types to *values*; vast majority of type-checking is done at runtime. Programmer can assign and reassign objects of arbitrary types to given variable. Variables (and object attributes) are usually declared by assignment.

We consider dynamically typed languages more convenient for programmers — we're convinced that the possibility of sensible variable reuse and lack of need to declare variable types lets the programmer focus more on the actual task, especially during prototyping stage. This convenience however comes with a cost: dynamic typing imposes inevitable computing overhead as method calls and attribute accesses must be resolved at runtime. Additionally, compiling a program written in statically typed language can reveal many simple programming errors such as calling mistyped methods, even in unreachable code-paths; this is not the case for dynamically-typed languages and we suggest compensating this with more thorough test-suite (code coverage tools can greatly help with creating proper test-suite, see section 3.3 on page 41).

Another related property is interpreted vs. compiled nature; we should emphasize that this property refers to language *implementation*, not directly to the language itself, e.g. C language is commonly regarded as compiled one, several C interpreters however exist. We use the term "language is compiled/interpreted" to denote that principal implementation of that language is compiled, respectively interpreted.

compiled implementations
: translate source code directly into machine code suitable for given target processor. Their advantage is zero interpreter overhead. Developers are required to install a compiler (and perhaps a build system) or an IDE[5] used by given project (library) to be able to modify it. Write-build-run-debug cycle is usually longer in comparison to interpreted implementations.

interpreted implementations
: either directly execute commands in source code or, more frequently, translate source code into platform-independent *intermediate representation* which is afterwards executed in a *virtual machine*. We may allow the translate and execute steps to be separated so that Java and similar languages can be included. Advantages include usually shorter write-run-debug cycle that speeds up development and portable distribution options. Interpreted languages have been historically associated with considerable processing overhead, but *just-in-time compilation*[6]

[5] Integrated Development Environment

[6] interpreter feature that translates portions of bytecode into machine code at runtime.

along with *adaptive optimisation*[7] present in modern interpreters can minimise or even reverse interpreter overhead: Paul Buchheit have shown[8] that second and onward iterations of fractal-generating Java program were actually 5% faster than equivalent C program. We have reproduced the test with following results: Java program was 10% slower (for second and subsequent iterations) than C program and 1600% slower when just-in-time compilation was disabled. Complete test environment along with instructions how to reproduce it be found in the `examples/benchmark_c_java` directory in the PyBayes source code repository.

There exists a historic link between statically typed and compiled languages, respectively dynamically typed and interpreted languages. Java which is itself statically typed and it's major implementation is interpreted and Erlang's (which is dynamically typed) compiled HiPE[9] implementation are some examples of languages that break the rule. We believe that this historic link is the source of a common misconception that interpreted languages are inherently slow. Our findings (see also Python/Cython/C benchmark on p. 25) indicate that the source of heavy overhead is likely to be the dynamic type system rather than overhead of modern just-in-time interpreters. In accordance with these findings, we may conclude that choice of language implementation type should rather be based on development and distribution convenience than on expected performance.

Each programming language may support one or more following function call conventions that determine how function parameters are passed:

call-by-value convention
: ensures that called function does not change variables passed as parameters from calling function by copying them at function call time. This provides clear semantics but incurs computational and memory overhead, especially when large data structures are used as parameters. As a form of optimisation, some language implementations may employ copy-on-write technique so that variables are copied only when they are mutated from within called function, thus saving space and time when some parameters are only read from.

call-by-reference convention
: hands fully-privileged references to parameters to called function. These references can be used to modify or assign to parameters within called function and these changes are visible to calling function. This approach minimises function call overhead but may appear confusing to a programmer when local variable is changed "behind her back" unexpectedly. On the other hand, call-by-reference allows for programming techniques impossible with call-by-value alone (e.g. a function that swaps two values).

call-by-object (call-by-sharing) convention
: can be viewed as a compromise between call-by-value and call-by-reference: parameters are passed as references that can be used to modify referred objects (unless marked immutable), but cannot be used to assign to referred objects (or

[7] a technique to use profiling data from recent past (collected perhaps when relevant portion of code was run in interpreted mode) to optimise just-in-time compiled code.

[8] `http://paulbuchheit.blogspot.com/2007/06/java-is-faster-than-c.html`

[9] The High-Performance Erlang Project: `http://www.it.uu.se/research/group/hipe/`

this assignment is invisible to calling function). When an object is marked as immutable, passing this object behaves like call-by-value call without copying overhead (in the calling function point of view). Java and Python use call-by-object as their sole function calling method[10] and both mark certain elementary types (most prominently numbers and strings) as immutable. C's pointer-to-const and C++'s reference-to-const parameters can be viewed as call-by-object methods where referred objects are marked as immutable in called function scope.

We suggest that a language that supports at least one of call-by-reference or call-by-object conventions is used for the desired recursive Bayesian estimation library; while call-by-value-only languages can be simpler to implement, we are convinced that they impose unnecessary restrictions on the library design and cause overhead in places where it could be avoided.

Last discussed aspect of programming languages relates to memory management:

garbage-collected languages
: provide memory management in the language itself. This fact considerably simplifies programming as programmer doesn't need to reclaim unused memory resources herself. Another advantage is that automatic memory management prevents most occurrences of several programming errors: memory leaks,[11] dangling pointers[12] and double-frees.[13] Two major approaches to garbage collection exist and both incur runtime computational or memory overhead. *Tracing garbage collector* repeatedly scans program heap[14] memory for objects with no references to them, then reclaims memory used by these objects. Program performance may be substantially impacted while tracings garbage collector performs its scan; furthermore the moment when garbage collector fires may be unpredictable. *Reference counting* memory management works by embedding an attribute, *reference count*, to each object that could be allocated on heap and then using this attribute to track number of references to given object. When reference count falls to zero, the object can be destroyed. Reference counting adds small memory overhead per each object allocated and potentially significant computational overhead as reference counts have to be kept up-to-date. However, techniques exist that minimise this overhead, for example those mentioned in [9].

non garbage-collected languages
: put the burden of memory management on shoulders of the programmer: she is responsible for correctly reclaiming resources when they are no longer in use. The advantages are clear: no overhead due to memory management, probably also smaller complexity of language implementation. However, as mentioned earlier, languages without automatic memory management make certain classes of programmer errors more likely to occur.

[10] python case: http://effbot.org/zone/call-by-object.htm

[11] an error condition when a region of memory is no longer used, but not reclaimed.

[12] a pointer to an object that has been already destroyed; such pointers are highly error-prone.

[13] an error condition where a single region of memory is reclaimed twice; memory corruption frequently occurs in this case.

[14] an area of memory used for dynamic memory allocation.

In our view, convenience of garbage-collected languages outweighs overhead they bring for a project like a library for recursive Bayesian estimation targeting wide adoption. We also believe that automatic memory management can simplify library design and its usage as there is no need to specify who is responsible for destroying involved objects on the library side and no need to think about it at the user side.

2.3 C++

C++ is regarded as one of the most popular programming languages today, along with Java and C;[15] it combines properties of both low-level and high-level languages, sometimes being described as intermediate-level language. C++ extensively supports both procedural and class-based object-oriented paradigm, forming a multi-paradigm language; generic programming is implemented by means of *templates*, which allow classes and functions to operate on arbitrary data types while still being type-safe. C++ is statically-typed, all major implementations are compiled, supports call-by-value (the default), call-by-reference and a variant of call-by-object function call conventions. C++ lacks implicit garbage collection for heap-allocated data — the programmer must reclaim memory used by those objects manually; use of *smart pointers*[16] may although help with this task. C++ is almost 100% compatible with the C language in a way that most C programs compile and run fine then compiled as C++ programs. C++ also makes it easy to use C libraries without a need to recompile them. [16]

When used as an implementation language for the desired library for recursive Bayesian estimation, we have identified potential advantages of the C++ language:

low overhead
: C++ was designed to incur minimal overhead possible. In all benchmarks we've seen (e.g. The Computer Language Benchmarks Game[17]), it is hard to outperform C++ by a significant margin (Fortran and assembly code would be candidates for that).

widespread
: C/C++ code forms large part of the software ecosystem. Thanks to that, incredible number of both proprietary and free IDEs, debuggers, profilers and other related coding tools is available. This fact makes development more convenient.

libraries
: Thanks to C++ popularity, several high-quality libraries for numerical calculations/computer algebra are available, many of them are free software or free to use. These are for example C interfaces to BLAS[18] and LAPACK[19] (both

[15] TIOBE Programming Community Index for July 2011: http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

[16] a template class that behaves like a pointer through use of operator overloading but adds additional memory management features such as reference counting

[17] http://shootout.alioth.debian.org/

[18] Basic Linear Algebra Subprograms: http://www.netlib.org/blas/

[19] Linear Algebra PACKage: http://www.netlib.org/lapack/

low-level and fixed function), higher-level IT++[20] built atop of BLAS/LAPACK or independent template-based library Eigen.[21] Additionally, OpenMP[22] can be used to parallelise existing algorithms without rewriting them.

However, using C++ would, in our opinion, bring following major drawbacks:

diversity
While there are many C/C++ libraries for specific tasks (such as data visualisation), it may prove difficult in our opinion to combine them freely as there are no *de facto* standard data types for e.g. vectors and matrices — many libraries use their own.

learning curve
C++ takes longer to learn and even when mastered, programmer productivity is subjectively lower compared to very high-level languages. We also fear that many members of out intended audience are simply unwilling to learn or use C++.

Moreover, discussion about statically-typed, compiled and non-garbage-collected languages from previous section also apply. Due to this, we have decided not to use C++ if an alternative with reasonable overhead is found.

Several object-oriented C++ libraries for recursive Bayesian estimation exist: Bayes++[23], BDM [19] and BFL [5]. BDM library is later used to compare performance of Cython, C++ and MATLAB implementations of the Kalman filter, see section 3.4 on page 43.

2.4 MATLAB language

MATLAB language is a very high-level language used exclusively by the MATLAB[24] environment, a proprietary platform developed by MathWorks.[25] MATLAB language extensively supports procedural programming paradigm and since version 7.6 (R2008a) class-based object oriented paradigm is also fully supported.[26] MATLAB language is dynamically-typed, interpreted language with automatic memory management.

MATLAB language possesses, in our belief, following favourable attributes when used to implement the desired library for Bayesian filtering:

popularity among academia
While MATLAB language is not as widespread as C++ on the global scale, it is very popular in scientific community, our intended audience.

performance

[20] http://itpp.sourceforge.net/
[21] http://eigen.tuxfamily.org/
[22] The OpenMP API specification for parallel programming: http://openmp.org/
[23] http://bayesclasses.sourceforge.net/
[24] http://www.mathworks.com/products/matlab/
[25] http://www.mathworks.com/
[26] http://www.mathworks.com/products/matlab/whatsnew.html

MATLAB language is very well optimised for numerical computing.

wide range of extensions
High number of well integrated extension modules (toolboxes) is bundled with MATLAB or available from third parties. This makes associated tasks such as data visualisation particularly straightforward.

rapid development
Being a very high-level language, we expect programmer productivity in the MATLAB language being fairly high. MATLAB environment is itself a good IDE and its interactive shell fosters rapid prototyping.

Following disadvantages of the MATLAB language were identified:

vendor lock-in
MATLAB is commercial software; free alternatives such as GNU Octave[27], Scilab[28] or FreeMat[29] exist, however all of them provide only limited compatibility with the MATLAB language. Developing for a non-standard proprietary platform always imposes risks of the vendor changing license or pricing policy etc.

problematic object model
We have identified in subsection 2.2.2 that object-oriented approach is important for a well-designed and usable library for Bayesian filtering. Nonetheless MATLAB's implementation of object-oriented programming is viewed as problematic by many, including us. For example, function call parameter passing convention is determined by the object class/data type — MATLAB distinguishes *value classes* that have call-by-value semantics and *handle classes* that have call-by-object semantics.[30] The resulting effect is that calling identical function with otherwise equivalent value and handle classes can yield very different behaviour.

hard-coded call-by-value semantics
2D array, a very central data-type of the MATLAB language, has call-by-value function call convention hard-coded; this results in potentially substantial function call overhead. Although current MATLAB versions try to minimise copying by employing copy-on-write technique[31] or performing some operations in-place,[32] our tests have shown that even combining these techniques doesn't eliminate unnecessary copying overhead which we believe is the main source of grave performance regression of object-oriented code with regards to imperative code; see section 3.4 on page 43.

We consider presented drawbacks significant and therefore decided not to use the MATLAB language for the desired Bayesian filtering library. BDM library [19] contains both object oriented and imperative implementation of the Kalman filter in the MATLAB language; these are compared with our implementation in section 3.4.

[27] http://www.gnu.org/software/octave/
[28] http://www.scilab.org/
[29] http://freemat.sourceforge.net/
[30] call-by-object semantics tested in version 7.11 (R2010b).
[31] http://blogs.mathworks.com/loren/2006/05/10/memory-management-for-functions-and-variables/
[32] http://blogs.mathworks.com/loren/2007/03/22/in-place-operations-on-data/

2.5 Python

Python[33] is a very high level programming language designed for outstanding code readability and high programmer productivity actively developed by the Python Software Foundation.[34] Python extensively supports procedural and class-based object-oriented programming paradigms and some features of the functional programming. Python is dynamically-typed language with automatic memory management that exclusively employs call-by-object function call parameter passing convention; elementary numeric types, strings and tuples are immutable[35] so that this approach doesn't become inconvenient.

Principal Python implementation, CPython, is written in C, is cross-platform and of interpreted type: it translates Python code into bytecode which is subsequently executed in a virtual machine. Many alternative implementations are available, to name a few: Jython[36] that translates Python code into Java bytecode (itself written in Java), IronPython[37] itself implemented on top of the .NET Framework, just-in-time compiling PyPy[38] written in Python itself or Cython which is described in greater detail in the next section. All the mentioned implementations qualify as free/open-source software.

Python language is bundled with a comprehensive standard library so that writing new projects is quick from the beginning. Two major Python versions exists: Python 2, considered legacy and receiving only bugfix updates, and Python 3, actively developed and endorsed version that brings a few incompatible changes to the language syntax and to the standard library. Porting Python 2 code to version 3 is however usually straightforward and can be automated to a great extent with tools bundled with Python 3.

In our belief, Python shows following favourable attributes when used for the desired Bayesian filtering library:

development convenience, readability, rapid prototyping
: Python developers claim that Python in an easy to learn, powerful programming language and our experience confirms their claims. Python code is easy to prototype, understand and modify in our opinion; prototyping is with bundled interactive Python shell. While all these statements are subjective, they are shared among many.[39] For example a statement `x <= y <= z` has its mathematical meaning, which is unusual for programming languages.

NumPy, SciPy, Matplotlib
: NumPy project[40] is the de facto standard Python library for numeric computing; NumPy provides N-dimensional array type that is massively supported in very

[33] http://www.python.org/
[34] http://www.python.org/psf/
[35] http://docs.python.org/reference/datamodel.html
[36] http://www.jython.org/
[37] http://ironpython.net/
[38] http://pypy.org/
[39] http://python.org/about/quotes/
[40] http://www.numpy.org/

high number of projects. Parts of NumPy are written in C and Cython for speed. SciPy[41] extends NumPy with more numerical routines. Matplotlib[42] is powerful plotting library that natively supports SVG output. Combining these three and Python gives a very vital MATLAB alternative.

interoperability with C
: CPython makes it possible to write modules[43] in C[44] (that are then called *extension modules*). Cython makes it easy and convenient to write extension modules. Sadly, alternative implementations PyPy, Jython and IronPython don't currently fully support extension modules and are therefore ruled-out for our purposes because they in turn don't support NumPy.[45]

interoperability with MATLAB
: SciPy contains procedures to load and save data in MATLAB .mat format; Matplotlib includes programming interface that resembles MATLAB's plotting procedures.

On the other hand, a few downsides exist:

overhead
: CPython implementation shows significant computational overhead, especially for numerical computations; CPython doesn't currently utilise any form of just-in-time compiling. NumPy is often used to trade off computational overhead for memory overhead: The Computer Language Benchmarks Game[46] contains an example where a program heavily using NumPy is 20× faster but consumes 12× more memory than a program that performed the same task and used solely the Python standard library. We have reproduced the benchmark with similar results; mentioned programs can me found in the `examples/benchmark_py_numpy` directory in the PyBayes source code repository. We still consider this workaround suboptimal.

peculiar parallelisation
: While Python natively supports threads and they are useful for tasks such as background I/O operations, Python threads don't scale on multiprocessor systems for CPU-bound processing; such code often runs at single-processor speed (when run in CPython). The reason behind that is that CPython employs a global-interpreter-lock (GIL) to to assure that only one thread executes Python bytecode at a time.[47] This restriction can be worked around by using multiple python interpreters that communicate with each other; Python module *multiprocessing* makes it almost as convenient as using threads.

Python compares favourably to other implementation environments presented before in our view; sole major obstacle being excessive overhead of the CPython interpreter.

[41] http://www.scipy.org/
[42] http://matplotlib.sourceforge.net/
[43] module in python sense is a code unit with its own namespace, normally each module corresponds to a .py file.
[44] http://docs.python.org/extending/index.html
[45] PyPy and IronPython are nonetheless interesting for future consideration as both have NumPy support actively worked on.
[46] http://shootout.alioth.debian.org/
[47] http://docs.python.org/glossary.html

It is discussed how this issue can be solved using Cython in the next section.

We haven't found any Python library for recursive Bayesian estimation that would fulfil the requirements presented at the beginning of this chapter.

2.6 Cython

Cython [2] is both an extension to the Python language and an implementation of it (a compiler). Cython works by translating Cython modules (.pyx or .py files) into the C language which is then compiled to form binary Python extension modules (dynamically loaded libraries — .dll files on Windows and .so files on UNIX platforms). Cython aims to be a strict superset of Python in a way that a valid Python module is also a valid Cython module that behaves equally. Current development snapshot of Cython virtually achieves this goal as it successfully passes 97.6% of the Python 2.7.1 regression test suite. Additionally, interpreted Python modules and Cython-compiled modules can be mixed and interchanged freely as Cython-compiled code can transparently interact with interpreted Python code and vice-versa. However, Cython is not a replacement of CPython — Cython-compiled modules need to be executed by CPython because they make heavy use of CPython internals written in C (C code emitted by the Cython compiler is largely composed of calls to functions from CPython C API); Cython-compiled modules merely circumvent the virtual machine (bytecode interpreter) part of CPython, thus virtually eliminating interpreter overhead. This is in fact probably the most serious limitation of Cython — it is tightly bound to one specific (although the most prevalent) Python implementation, CPython.

Another key feature Cython supports is static typing of variables, function parameters and object attributes. In addition, Cython allows statically typed variables to be native C types such as `int`, `char *` or `struct sockaddr` in addition to Python types; automatic conversion on Python/C boundary is provided by Cython for C types with Python equivalents, C `double` is wrapped as Python `float` and C `long` is wrapped as Python `int` for example. The main purpose of static typing are impressive speed gains; an extreme case exploited by our test case showed 65× speed increase of typed Cython code compared to untyped Cython code. In important thing to mention is that static typing is completely *voluntary* for the programmer; recent Cython versions also support experimental basic type inference[48] (that is guaranteed not to change semantics). Simply put, static typing prevents significant overhead caused by highly dynamic nature of Python.

2.6.1 Cython Features

We continue by a brief technical description of some Cython extensions to Python and other features so that Cython's benefits can be evaluated as a whole later. Unless noted otherwise, Cython version 0.14.1 is described here; please note that Cython

[48] a way to guess and prove type of certain variable using code analysis

is a rapidly evolving project and some of the mentioned features or limitations may well change in future versions.

cdef variables
Cython supports so-called *cdef variables* in both global (module) scope and function (method) scope that can be typed; in addition to Python types, they can also have native C types. Cdef variables aren't visible to Python code, but some overhead is eliminated as they are, for example, not reference-counted.

cdef and cpdef functions
In addition to traditional Python functions (`def function(x)` in code) that use rather expensive calling convention (e.g. all their positional arguments are packed into a tuple and all their keyword arguments are packed into a dictionary on each call) Cython supports so-called *cdef functions* and *cpdef functions*. Cdef functions (defined as `cdef function(x)`, hence the name) use native C calling convention and are only callable from Cython code, with greatly reduced overhead; their return value can be typed (parameters can be typed even for traditional Python functions). Cpdef functions (defined as `cpdef function(x)`) are same as cdef functions but in addition a Python wrapper around the C function with the same name is generated; such function is callable from Python (with overhead) and fast to call from Cython, cpdef functions therefore combine the benefits of both. Neither cdef or cpdef functions can be Python class methods, but see the next entry.

extension types (cdef classes)
In addition to traditional Python classes marked as (2.1) in code that store their attributes and methods in a class dictionary, which leads to inefficient attribute lookups and method calls, Cython supports so-called *cdef classes* (or extension types) that are defined using (2.2) and use C structs to store their attributes and method table, a significantly faster approach than Python dictionary.

```
class ClassName(SuperClass) :          (2.1)
cdef class ClassName(SuperClass) :     (2.2)
```

Cdef classes can also have cdef and cpdef methods as their members; they can also contain cdef variables as attributes with optional additional modifiers `public` (read-write from Python) or `readonly` (read-only from Python). Cdef classes are, in contrast to similarly named cdef functions, visible to Python code where they appear as built-in types. Compared to traditional Python classes, cdef classes are however subject to 2 major limitations: all their attributes have to be statically declared (as opposed to dynamically created at runtime) plus multiple inheritance cannot be used. One can overcome both these limitations by subclassing them in Python, which is indeed possible.

interfacing C/C++ code
As described in [3], Cython can be used to conveniently call C or C++[49] library functions. Suppose we want to call the `sin()` function form the C standard library that is defined in `math.h`. Following Cython module accomplishes that:

[49] C code emitted by the Cython compiler is compilable also when treated as C++, thus allows interfacing with C++ code.

sin_wrapper.pyx

```
cdef extern from math.h:
    double sin(double x)

sin(0.123)  # call C function directly from Cython

cpdef double sin_wrapper(double x):
    return sin(x)
```

In the example above, the `sin_wrapper()` function can be called from Python code. Cython is therefore an excellent tool to create Python wrappers around C/C++ libraries or use them directly.

NumPy support

As was noted above, many core parts of NumPy are written in C (or Cython), including the crucial data-type, N-dimensional array. Cython provides explicit support for NumPy data-types and core array-manipulation methods and functions allowing the programmer to take advantage of the speed gains of both. [14] On the other hand, NumPy support in Cython (or Cython support in NumPy) is far from complete, for example matrix multiplication, linear system solving and matrix decomposition functions are usually speeded up using C (and BLAS, LAPACK) internally, but to our knowledge it is presently impossible to call these functions from Cython without Python overhead (the overhead is however significant only for small matrix sizes).

pure Python mode

It is clear that in order to make use of all important features of Cython, one has to use syntax that is no longer valid Python code; this can be disadvantageous in many cases. To combat this, Cython offers so-called *pure Python mode* where Cython-specific syntax is wrapped in Python constructs, e.g. (2.3) can be alternatively formulated as (2.4) — such constructs are implemented by Cython shadow code to be no-ops when interpreted by Python and treated accordingly when compiled by Cython.

```
cdef int i                                  (2.3)
x = cython.declare(cython.int)              (2.4)
```

Another variant of pure Python mode is to use so-called *augmenting files*, .pxd files that accompany equally-named .py files. Such augmenting file can contain declarations of variables, functions and classes that appear in associated .py files and add Cython-specific features to them. For example, suppose that it is desired to have a function that is inexpensive to call from Cython but still providing full Python compatibility, one can write following Python module:

module.py

```
def f(x):
    return x*x
```

And associated augmenting file:

module.pxd

```
cdef double f(double x)
```

It is forbidden to provide implementation in .pxd files. This alternative approach is advantageous to the first one because it doesn't require Cython to be installed on a target machine (on the other hand, some very special Cython features currently cannot be expressed in augmenting files). .pxd files also serve for sharing declarations between Cython modules in a way similar to C header (.h) files.[50]

parallelisation
One recent feature of current development snapshots of Cython is a native support for C-level parallelisation through OpenMP[51] — Cython adds new `prange()` function that is similar to the Python `range()` function except that loop statements are executed in parallel. Currently only loops whose statements can be called without holding the GIL[52] can be efficiently parallelised. Our tests have shown that such parallel loops scale well with the number of processors, see subsection 2.6.2 on page 25.

Python 3 compatibility
Cython provides full compatibility with Python 3 in particularly robust way: Cython modules can be written in both Python 2 or Python 3 and C files that Cython produces can be compiled against both CPython 2 or CPython 3, independently from the source file version.[53] This effectively makes Cython a 2-way compatibility bridge between Python 2 and Python 3.

There are many areas where Cython can get better, however most of them are just missed optimisation possibilities. We mention 2 major cases where Cython currently doesn't accept valid Python constructs: the *yield* statement[54] used to create generator functions is currently unsupported and *generator expressions*[55] are supported only in special cases. Other minor inconveniences exist but we don't consider them worth discussing here. Another important attribute of Cython is, in our belief, its active developer community, for example we've discovered a bug related to Cython's pure Python mode capability that was fixed upon providing a test-case.[56]

2.6.2 Performance comparison with C and Python

In order to evaluate performance of Cython, we've conducted a benchmark where equivalent Python, Cython-compiled Python, Cython and C programs are compared. Inspired by the Cython tutorial, the test program performs simple numerical integration of the function x^2 from 0 to 3. Python version of the test program is shown here, complete test environment along with instructions how to reproduce can be found in the `examples/benchmark_c_cy_py` directory in the PyBayes source code repository.

[50] http://docs.cython.org/src/userguide/sharing_declarations.html
[51] The OpenMP API specification for parallel programming: http://openmp.org/
[52] global-interpreter lock http://docs.python.org/glossary.html
[53] http://wiki.cython.org/FAQ
[54] http://docs.python.org/reference/simple_stmts.html
[55] http://docs.python.org/reference/expressions.html
[56] http://trac.cython.org/cython_trac/ticket/583

integrate_python.py

```
def f(x):
    return x*x

def integrate(a, b, N):
    s = 0
    dx = (b-a)/N
    for i in xrange(N):
        s += f(a + (i + 1./2.)*dx)*dx
    return s
```

The test was performed on a 64-bit dual-core Intel Core i5-2520M CPU clocked at 2.50 Ghz with Intel Turbo Boost and Hyper-threading enabled, giving the total of 4 logical processor cores (each physical core being able to execute 2 CPU threads); operating system is Gentoo Linux compiled for the x86_64 platform. Versions of relevant software packages are listed below:

Python 2.7.1
GNU C Compiler 4.4.5; `-O2` optimisation flag used when compiling C files
Cython 0.14.1+ (git revision 0.14.1-1002-g53e4c10)
PyPy 1.5.0-alpha0 with JIT compiler enabled
PyBayes 0.3 (contains test program sources)

Program abbreviations used in benchmark results:

`c_omp`	C version, for loop parallelised using OpenMP
`cy_typed_omp`	Cython version with all variables typed, parallelised using `prange()`
`pypy`	PyPy-executed Python version, single-threaded
`c`	C version, single-threaded
`cython_typed`	Cython version with all variables typed, single-threaded
`cython`	Cython-compiled Python version, single-threaded
`python`	Python version, single-threaded

typical benchmark run

```
Numerical integration from 0.0 to 3.0 of x^2 with 200000000 steps:

       c_omp: result = 9.0; real time = 0.446s; cpu time = 1.72s
cy_typed_omp: result = 9.0; real time = 0.447s; cpu time = 1.73s
        pypy: result = 9.0; real time = 0.851s; cpu time = 0.84s
           c: result = 9.0; real time = 1.597s; cpu time = 2.00s
cython_typed: result = 9.0; real time = 1.590s; cpu time = 1.58s
      cython: result = 9.0; real time = 33.26s; cpu time = 33.2s
      python: result = 9.0; real time = 95.05s; cpu time = 94.8s

Relative speedups:
      cython/python:       2.8571203030
cython_typed/cython:       20.9136660253
```

```
            c/cython_typed: 0.99693008347

cy_typed_omp/cython_typed: 3.55556994329
       c_omp/c:            3.57674225915

       c_omp/cy_typed_omp: 1.00186053812
```

The `cpu time` quantity is not reliable in our belief and is mentioned only for illustration; all measurements are based on the `real time` quantity that measures wall-clock time needed to perform the algorithm. The number of steps was chosen artificially high to get timings that are easier to measure. The benchmark had little variance in results across runs, the relative sample standard deviation (2.5) was under 3% for all measured quantities (run times) with $N = 10$.

$$s_{\text{rel}} = \frac{1}{\overline{x}} \sqrt{\frac{1}{N-1} \sum_{i=1}^{N} (x_i - \overline{x})^2} \tag{2.5}$$

The test produced a couple of interesting results: the same Python code was 2.9× faster when compiled by Cython; we are convinced that this is due to the Python interpreter overhead. Adding static type declarations to the Cython code resulted in additional 21-fold speed-up giving the total 60× increase in performance of a statically-typed Cython procedure compared to the dynamically typed Python one, forming a very impressive result. Somewhat surprising results were obtained from parallelisation tests — parallelised Cython version was 3.5× faster than equivalent singe-threaded code on a system with only 2 physical processor cores, we speculate that this is due to the Hyper-threading technology employed by the processor that reduced wasted CPU cycles (where the CPU waits for memory fetches). Nonetheless this shows that Cython (and C) parallelisation techniques using OpenMP scale very well. C and Cython versions of the algorithm performed virtually equally in both singe-threaded and multi-threaded cases giving an indication that Cython performs a very good job at optimising Python code in such simple cases. Another surprise was PyPy (operating on unmodified Python code) performance — it was considerably faster that both C and optimised Cython code making it a very interesting option once it supports NumPy.

We should however note that these extreme Cython performance gains are specific for such simple and highly numeric algorithms. We expect smaller benefits for more high-level code; Kalman filter tests (section 3.4 on page 43) support our claims.

2.6.3 Discussion

Cython, in our view, fixes the last possible barrier (the CPython overhead) before Python can be used for the desired library for Bayesian filtering. We consider especially important that optimisation can be approached *gradually* — one can write code purely in Python and add static type definitions only for performance

critical parts (that show up as bottlenecks during profiling) and only once the high performance is needed; Cython developers soundly discourage adding static types everywhere. Second key feature of Cython, as identified by us and for out purposes, is the pure Python mode. Employing Cython brings the disadvantages of compiled languages, mainly a prolonged write-build-run-debug cycle. When the pure Python mode[57] is used, all these shortcomings are effectively voided (as the code works also under plain Python) at the cost of very minor loss of programming convenience. We expect that many potential library users don't have high performance requirements, these could ignore Cython entirely and use the library as any other plain python module.

Python/Cython combination was therefore chosen as the implementation environment for the desired library for recursive Bayesian estimation; the library was named *PyBayes* and is presented in the next chapter.

[57] specifically, the variant where only augmenting files are used

Chapter 3

The PyBayes Library

In this chapter the PyBayes library that is being developed with the aim to fulfil the requirements posed in the previous chapter (p. 10) is presented. After a brief introduction the library design, which builds on the performed software analysis, is shown and discussed. Various development practices used are later examined and the chapter is concluded by a performance comparison of various implementations of the Kalman filter (from PyBayes and BDM) benchmarked under 4 different implementation environments.

3.1 Introduction to PyBayes

PyBayes[1] is a Python/Cython library for recursive Bayesian estimation actively developed by the author of this text, a result of the software analysis carried-out. The development happens publicly and openly using the git[2] version control system on the GitHub[3] source-code hosting service at the address http://github.com/strohel/PyBayes that also serves as the home page of the project; PyBayes is also accessible from the Python Package Index (PyPI).[4] PyBayes is a free/open-source software licensed under the GNU GPL[5], version 2 or later. Version 0.3 of PyBayes is described in this text; we expect PyBayes to evolve in future and thus some claims present this chapter may become outdated. All currently planned future changes are however mentioned at appropriate places.

The goal of PyBayes is to provide a Python library that satisfies the posed requirements, is very convenient to develop with even when prototyping novel algorithms, but fast enough to be deployed in production. Library design should be object-oriented and very clean to be well comprehensible. In order to achieve both

[1] the name PyBayes had been previously used for an unrelated project dealing with Bayesian networks by Denis Deratani Maua, who later proclaimed the project dead and allowed us to use the name.

[2] http://git-scm.com/

[3] http://github.com

[4] http://pypi.python.org/pypi/PyBayes

[5] GNU General Public License: http://www.gnu.org/licenses/gpl.html

of these usually contradicting demands, PyBayes uses a special technique where the same source code can be interpreted by Python as usual (giving all advantages of Python) or compiled using Cython which makes use of additional *augmenting files* that are present in sources to provide static type declarations to performance-critical code-paths; PyBayes thus employs Cython's *pure Python mode*. The Cython build is currently 50% to 200% faster than Python depending on the algorithm and level of optimisation applied to it, see section 3.4 (p. 43) for example measurements. PyBayes' `setup.py`, the use of which is the standard way to install Python packages, automatically detects whether Cython is installed on the system and uses it when possible. NumPy's `ndarray` (N-dimensional array) of Python `floats`[6] is used as principal numeric type for vectors and matrices for its low overhead, convenience and interoperability.

PyBayes sources are maintained to be compatible with Python versions 2.5, 2.5 and 2.7; Python 3 compatibility can achieved using the 2to3[7] automatic code conversion tool, the sources are kept to be convertible without interaction (CPython's `-3` command-line can be used for this task). To promote code readability, coding style prescribed by the PEP 8[8] is followed when feasible.

The sections below present the library design and explain some decisions taken during development; they complement the **PyBayes API Documentation**, which is a reference guide intended for PyBayes users. API Documentation is available online at http://strohel.github.com/PyBayes-doc/.

All class diagrams in this text utilise standard UML[9] notation and are not an exhaustive reference of all classes, members and methods — they rather illustrate the API Documentation; inherited attributes and methods are not shown in diagrams. Unless noted otherwise, all references to files and folders in this chapter refer to the respective files/folders in the PyBayes source code repository.[10] Python software nomenclature is used, most notably the following terms:

module a file with .py extension (but denoted without it) that contains Python code and has its own namespace.

package a folder that contains above modules and possibly other packages; package namespace is identical with its `__init__` module that has to be present.

3.2 Library Layout

The source code of PyBayes is arranged as follows:

[6] Python `float` (`numbers.Real`) corresponds to C `double`

[7] http://docs.python.org/library/2to3.html

[8] Python Enhancement Proposal 8: http://www.python.org/dev/peps/pep-0008/

[9] Unified Modelling Language: http://www.uml.org/

[10] http://github.com/strohel/PyBayes

`doc/`	control files for generating documentation (see also section 3.3).
`examples/`	auxiliary scripts and benchmark sources.
`pybayes/`	PyBayes Python package; the actual implementation is located in this package.
`scratch/`	miscellaneous and temporary files.
`thesis/`	source code of this text.
`tokyo/`	source code of the Tokyo project, bundled with PyBayes (see subsection 3.2.7).
`COPYING`	the text of GNU GPL v2, the PyBayes license.
`HACKING.rst`	a guide for PyBayes developers; can be viewed as plain-text.
`README.rst`	general information and installation instructions.
`setup.py`	setup script, a tool to build and install PyBayes.

The `pybayes` package, the most important part that forms the actual PyBayes library, contains 3 supportive packages that are considered private to PyBayes (that may change without notice), and 2 following modules that form the public API of the library (overview shown in Figure 3.1):

pdfs module contains a framework of probability density functions and related classes.

filters module contains Bayesian filters.

All classes mentioned in this chapter are *Cython extension classes* in the Cython build of PyBayes (for smaller overhead) and ordinary Python classes in the Python "build".

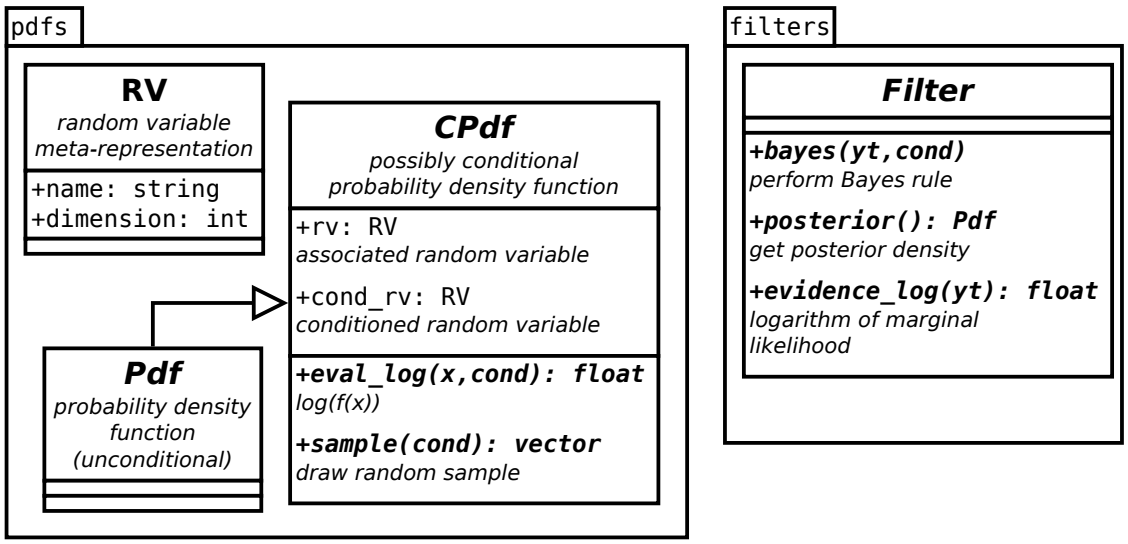

Figure 3.1: High-level overview of the PyBayes library; simplified

3.2.1 Probability Density Functions

Probability density functions have central role in the theory of Bayesian filtering and thus should receive great attention during library design. Probability density functions should be both flexible and lightweight as copying them is needed for example

in the marginalized particle filter. In PyBayes probability density functions are multivariate (even distributions that cannot be generalised to multiple dimensions take singe-valued vectors as parameters for consistency and easier static typing).

2 basic categories of probability density functions can be distinguished from the implementation point of view: unconditional ones whose statistics are fixed once the distribution is constructed and conditional probability density functions whose statistics depend on a free parameter; depending on one's standpoint, unconditional probability density functions may be viewed as a subclass of unconditional ones (that would have additional method `set_condition(cond)` or similar) or the other way around where unconditional probability density function is viewed as a specialisation of the conditional ones with a restriction that condition is empty. The latter approach is used by PyBayes for being less error-prone in our belief. Alternatively, conditional and unconditional densities could be unrelated (impractical in our case) or unified in one class without specifying conditionality (in fact, PyBayes is not far from this).

All probability density functions are represented using an abstract class *CPdf* that provides interface for querying random variable and conditioning variable dimensions (methods `shape()` and `cond_shape()`), for computing expected value and variance (methods `mean()` and `varience()`) that take condition as a parameter, for computing natural logarithm of the probability density function value in a given point (`eval_log()`) and for drawing random samples (`sample()`), both also accepting condition in parameters. A few support methods for use by subclasses not are provided to reduce code duplication, these aren't discussed here. The CPdf class also holds references to random variable and conditioning variable descriptions (attributes `rv` and `cond_rv`) which are talked bout in the next chapter. By convention `rv` and `cond_rv` to a valid *RV* object that can, however, signify "empty random variable".

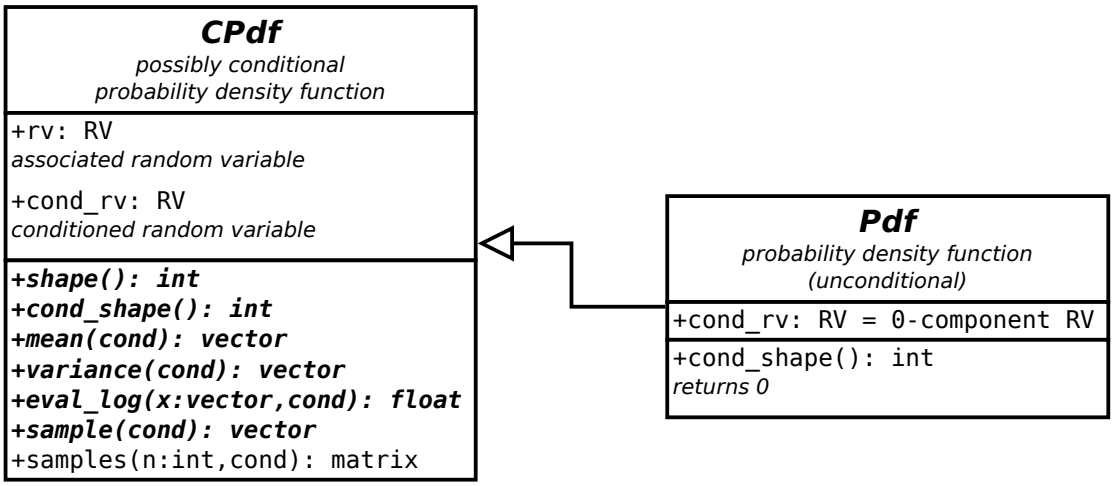

Figure 3.2: Class diagram of the probability density function prototypes

The *Pdf* is a very thin subclass of CPdf providing an implementation of the `cond_shape()` that returns zero to signify empty condition; by convention, Pdf subclasses also should set `cond_rv` to "empty random variable". The function of the

Pdf class is therefore rather semantic than technical distinction between conditional and unconditional probability density functions.

Outline of probability density function prototypes is shown in the Figure 3.2.

3.2.2 Random Variable Meta-representation

In mathematics, the relation between random variables and probability density functions is follows: a probability density function is associated to a random variable, e.g. "random variable X is normally distributed". This is impractical in software (should drawing a million bare samples produce a million copies of a random variable?) but let us show that the relation between random variables and probability density functions couldn't be entirely dropped without a substitution, let's look at that following example.

The chain rule for probability density functions is heavily used in Bayesian filtering and should be adequately supported by software. While simple cases can be represented without problems, when for example $p(\mathbf{a}, \mathbf{b}|\mathbf{c}, \mathbf{d})$ from (3.1) is desired to be represented, additional information have to be supplied by the user (programmer) so that the implementation "knows" how and what parts of involved vectors are passed to underlying densities p_1 and p_2. The implementation wouldn't be able to guess evaluation order, order of components in p_1 condition etc. without such additional information.

$$p(\mathbf{a}, \mathbf{b}|\mathbf{c}, \mathbf{d}) = p_1(\mathbf{a}|\mathbf{c}, \mathbf{b})p_2(\mathbf{b}|\mathbf{d}) \tag{3.1}$$

$$\mathbf{z} = (\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}) = (a_1, a_2, b_1, b_2, c_1, c_2, d_1, d_2) \tag{3.2}$$

In software it is practical to combine both random and conditioning variable of the left hand side of (3.1) into one vector; let $\mathbf{z}$ (3.2) denote such vector ($\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}$ are thus all two-dimensional). Now suppose that the implementation has to pass the vector representing condition $(\mathbf{c}, \mathbf{b})$ to the distribution p_1 assuming that p_2 was already evaluated. The problem that the software must solve therefore reduces to the task of finding indexes of the $(\mathbf{c}, \mathbf{b})$ components within vector $\mathbf{z}$; the solution is indeed an index vector $(5, 6, 3, 4)$, assuming one-based indexing.

The first option of passing such information from the user to the implementation would be to force her to specify the relations between distributions manually using index vectors (or a similar measure) directly; we believe that it is however error-prone and inconvenient. The other method is to make a symbolic association between probability density functions and random variable descriptions, but the other way around — probability density functions would "contain" the description of random variables they make use of, in our example above p_1 would contain an information that it is associated with random variable $(\mathbf{a})$ and conditioning variable $(\mathbf{c}, \mathbf{b})$.

The second approach is used in PyBayes even though it brings some computational overhead (that we think is worth the simplicity it brings). As mentioned in the previous section, the CPdf class has `rv` and `cond_rv` attributes that hold instances of the RV class. Simply put, the concept of random variable meta-representation

can be viewed as a kind of *semantic*, or *symbolic indexing* that should make the life of the PyBayes user easier.

The *RV* class is essentially a list of "random variable components" represented using the RVComp objects. RV provides a few methods to test relationships between 2 random variables (whether a RV is subset of another RV etc.) and one notable method, `indexed_in()` (that happen to be shown in the Figure 3.7 on page 43). Suppose that RV x has components $(x_1, x_2, \ldots, x_n)$ and RV y is a subset of x and contains components $(x_{i_1}, x_{i_2}, \ldots, x_{i_m})$; `y.indexed_in(x)` then returns an index array $(i_1, i_2, \ldots, i_m)$ which is suitable for NumPy array indexing methods.[11]

The *RVComp* class is a simple container for the `dimension` (which must be greater than zero) and `name` (which is optional) attributes; RV caches aggregate name and dimension. An important principle in PyBayes is that RVComp comparisons are *instance-based*: 2 RVComp objects are considered equal if and only if they refer to the same instance.[12] This is fast (compared to for example name-based equality), saves memory, prevents collisions and is convenient in Python thanks to its call-by-object semantics. The effects are best demonstrated in the following recording of an interactive Python session:

RV and RVComp demonstration

```
>>> rv = RV(RVComp(1, "a"))
>>> rv.contains(RVComp(1, "a"))
False

>>> a = RVComp(1, "pretty name")
>>> b = RVComp(1, "pretty name")  # same name, different instance
>>> rv = RV(a)
>>> rv.contains(a)
True
>>> rv.contains(b)
False
```

A RVComp without a name (with `name` attribute set to None), that can be called *anonymous component*, is created in CPdf when the user doesn't pass RV to the constructor, but is otherwise insignificant.

The concept of random variables might be used also for some Bayesian filters in future should there be a need for it. On the other hand, documented conventions (such as ordering of vector components) are used rather than RVs where feasible. Overview of RV and RVComp classes can be seen in the Figure 3.3.

[11] the notation used here is simplified; actual implementation allows for multivariate components x_i — i_j in returned index array are therefore ranges of integers.

[12] the name attribute thus serves only for aesthetic purposes.

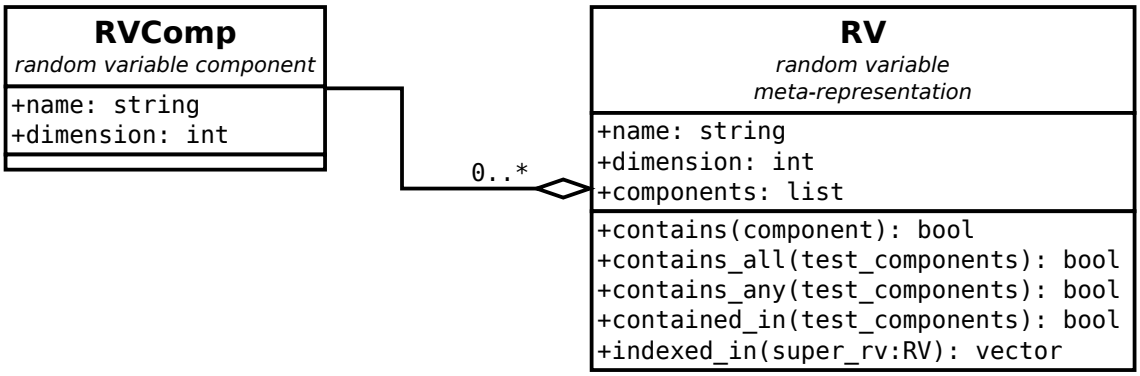

Figure 3.3: Class diagram of the random variable framework

3.2.3 Gaussian Probability Density Functions

We continue by a brief mention of implemented probability density functions; our policy is to add new distributions on as-needed basis rather than trying to have exhaustive set from the beginning. Every user of PyBayes can create its own distributions by subclassing Pdf of CPdf and implementing meaningful methods (there is no requirement for implementing unused methods).

PyBayes ships standard multivariate normal (Gaussian) probability density function through the *GaussPdf* class; related log-normal probability density function *LogNormPdf* is also provided and shares common abstract superclass, *AbstractGaussPdf* with GaussPdf. The AbstractGaussPdf class only holds mean attribute `mu` and covariance matrix attribute `R` and is useful mainly for the family of conditional Gaussian probability density functions described below.

The most general conditional Gaussian distribution is the *GaussCPdf* class that takes two functions `f` and `g` as parameters[13] plus the optional `base_class` parameter in constructor. The `base_class` parameter defaults to GaussPdf, but can be set tu LogNormPdf (to any AbstractGaussPdf subclass in general); the base class parameter determines resulting density — both conditional normal and log-normal distributions can be obtained without any code duplication, thanks to abstraction provided by AbstractGaussPdf. GaussPdf transforms supplied condition c using (3.3), substitutes to AbstractGaussPdf and calls respective `base_class` method.

$$\begin{aligned} \mu &= f(c) \\ R &= g(c) \end{aligned} \tag{3.3}$$

First specialisation of GaussCPdf is the *LinGaussCPdf* class that assumes that `f` and `g` functions are linear, the transformation is thus according to (3.4) where condition is divided into parts (c_1, c_2). The `A`, `C` (matrices), `b` and `d` (vector) parameters are passed to the constructor. LinGaussCPdf exists mainly for performance reasons and slightly higher convenience when passing arrays compared to functions;

[13] currently any Python callable objects are accepted; NumPy `ufunc` class will be evaluated for suitability in future.

LinGaussCPdf also benefits from generalisation offered AbstractGaussPdf.

$$\begin{aligned} \mu &= Ac_1 + b \\ R &= Cc_2 + d \end{aligned} \tag{3.4}$$

The last GaussCPdf specialisation is the MLinGaussCPdf class which works almost identically as LinGaussCPdf with the exception that the `R` (covariance) parameter is fixed. Transformation used by MLinGaussCPdf is thus defined by (3.5) where c is the conditioning variable. MLinGaussCPdf also supports setting `base_class` as usual.

$$\mu = Ac + b \tag{3.5}$$

See the Figure 3.4 for a survey of Gaussian and related probability density functions.

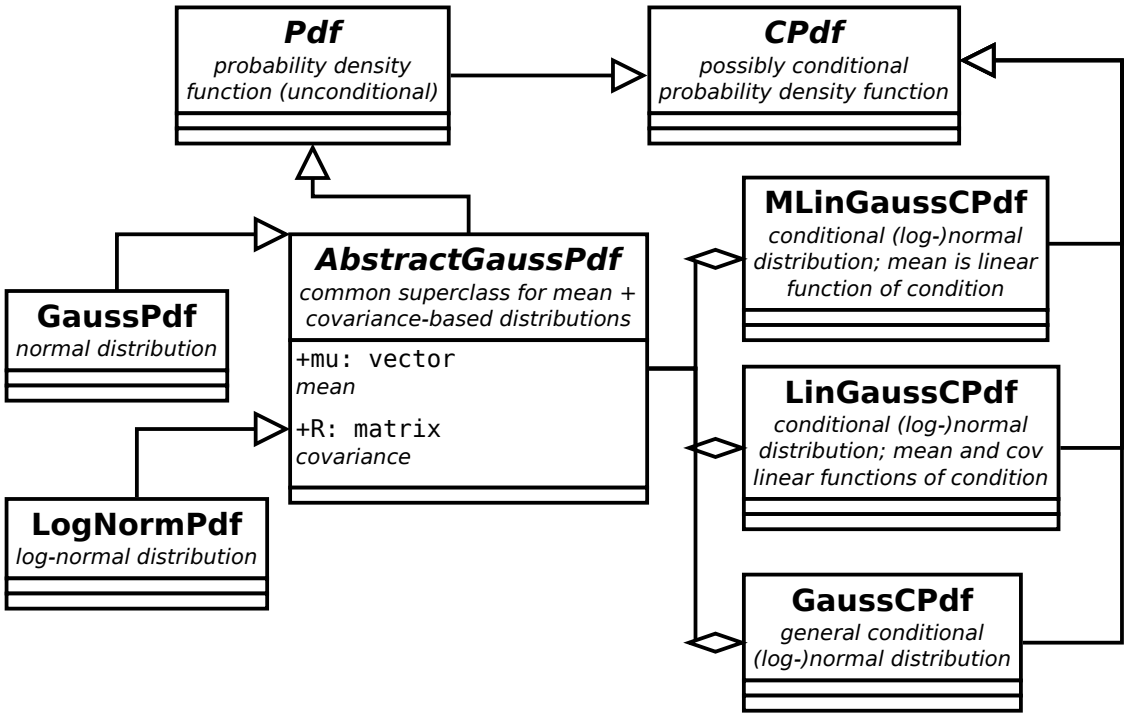

Figure 3.4: Class diagram of Gaussian and related distributions

3.2.4 Empirical Probability Density Functions

Another very useful set of distributions is the empirical family suitable for particle filters. Weighted empirical distribution named EmpPdf in PyBayes is the posterior probability density function of the particle filter while a special product of weighted empirical distribution and a mixture of Gaussian distributions is the posterior probability density function of the marginalized particle filter and is thus named MarginalizedEmpPdf. Both inherit from AbstractEmpPdf in order to reuse code. Neither of the empirical densities implement `eval_log()` or `sample()` — while the latter

would be possible, we are yet to find a valid use-case for it (resampling particles being implemented differently).

The *AbstractEmpPdf* class holds the `weights` parameter, a vector of particle weights denoted as $\omega = (\omega_1, \omega_2, \ldots, \omega_n)$ in formulas. The usual constraints (3.6) must hold. A simple method `normalise_weights()` normalises weights according to (3.7).

$$\omega_i >= 0 \quad \forall i \qquad \sum_{i=1}^{n} \omega_i = 1 \tag{3.6}$$

$$\omega_i' = \frac{\omega_i}{\sum_{i=1}^{n} \omega_i} \tag{3.7}$$

AbstractEmpPdf provides one more method called `get_resample_indices()` that (given that there are n particles) draws n random samples from itself and returns their indices. The algorithm is however optimised in a way that only one random sampling is performed; the results are thus more predictable (or, "less random"), but this is desired when used for resampling in particle filters — its primary (and currently only) use.

The *EmpPdf* class is the standard weighted empirical distribution (3.8) that extends AbstractEmpPdf with the `particles` attribute (a matrix) where each row $x^{(i)}$ represents one particle. It also provides the `resample()` method that resamples particles using `get_resample_indices()` and resets weights to be uniformly distributed. EmpPdf has an extra role in PyBayes, it is used to test `sample()` of other probability density functions using the moment method (sufficient number of samples is generated and sample mean and variance is compared with theoretical results).

$$p(x) = \sum_{i=1}^{n} \omega_i \delta(x - x^{(i)}) \tag{3.8}$$

Related to the empirical density is the *MarginalizedEmpPdf* that exists solely to form the posterior probability density function of the marginalized particle filter. It extends AbstractEmpPdf with a vector of GaussPdf objects `gausses`, i-th GaussPdf is denoted as $\mathcal{N}\left(\hat{a}^{(i)}, P^{(i)}\right)$ and a matrix `particles` where i-th row is denoted as $b^{(i)}$ in (3.9).

$$p(a, b) = \sum_{i=1}^{n} \omega_i \left[\mathcal{N}\left(\hat{a}^{(i)}, P^{(i)}\right)\right]_a \delta(b - b^{(i)}) \tag{3.9}$$

MarginalizedEmpPdf doesn't provide a method for resampling as this task have to be done in the particle filter implementation anyway at it has to deal also with the Kalman filters.

The class diagram of empirical probability density functions and related is displayed in the Figure 3.5.

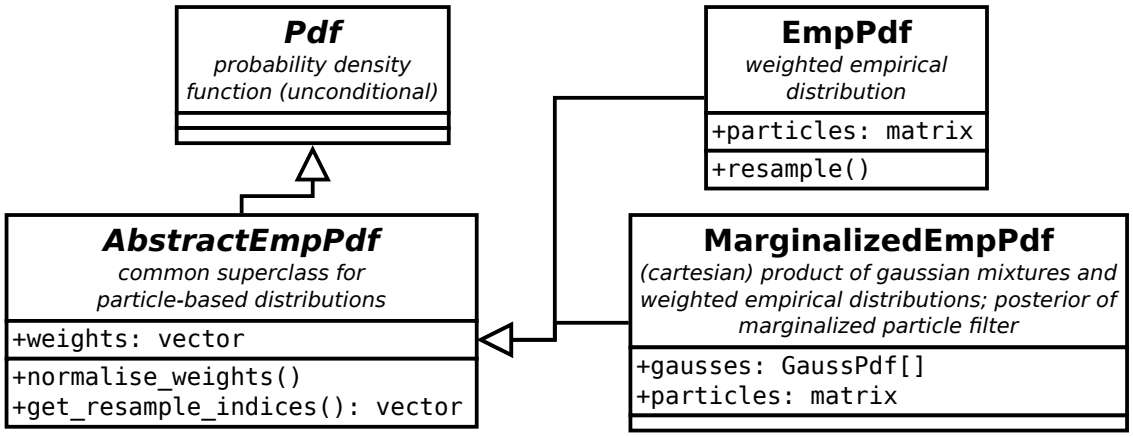

Figure 3.5: Class diagram of empirical distributions

3.2.5 Other Probability Density Functions

We conclude the discussion about the implemented probability density functions with the ones that don't fit elsewhere.

The *ProdPdf* class represents unconditional product of n independent random variables $x_1, x_2, \ldots, x_n$ as depicted in (3.10). As an exception from the general rule, ProdPdf constructs its random variable association (the `rv` attribute) using factor random variables for convenience; it however currently doesn't make any use of the random variable meta-representation as it would be of limited usability — only permutation of random variable components within x_i $(\forall i)$ would be additionally possible (the order of factors is already specified by the user. ProdPdf implements all abstract methods of Pdf by delegating work to factor probability density functions.

$$p(x_1, x_2, \ldots, x_n) = p_1(x_1)p_2(x_2)\cdots p_n(x_n) \tag{3.10}$$

A more sophisticated variant of the ProdPdf is the *ProdCPdf* class that is potentially conditional and allows for conditionally dependent random variables. In general it can represent a chain rule for probability density functions shown in (3.11) with an additional constraint that the right hand side makes sense (that means that there exists a sequence $(p_{j_1}, p_{j_2}, \ldots, p_{j_m})$ for which (3.12) holds). The relation "$C \subset \{x_i, x_j, x_k, \ldots\}$" denotes "random variable C is composed of the (subset of) $x_i, x_j, x_k, \ldots$ random variable components" in the following formulas.

$$\begin{gathered} p(x_1, x_2, \ldots, x_m | x_{m+1}, x_{m+2}, \ldots, x_n) = p_1(x_1|C_1)p_2(x_2|C_1)\cdots p_m(x_m|C_m) \\ \text{where} \quad m \leq n; \quad \forall i \in \{1, 2, \ldots, m\} : C_i \subset \{x_1, x_2, \ldots, x_n\} \end{gathered} \tag{3.11}$$

$$\forall k \in \{1, 2, \ldots, m\} : C_{j_k} \subset \{x_1, x_2, \ldots, x_{k-1}\} \cup \{x_{m+1}, x_{m+2}, \ldots, x_n\} \tag{3.12}$$

As in ProdPdf, all abstract methods of CPdf are implemented. ProdCPdf makes extensive use of the random variable meta-representation described earlier; it uses random variable descriptions of factor densities $p_1, p_2, \ldots, p_m$ to construct the data-flow (the $(p_{j_1}, p_{j_2}, \ldots, p_{j_m})$ sequence); the order of passed factor distributions is therefore insignificant — ProdCPdf always computes correct evaluation order if it

exists. For this reason the random variable components of factor probability density functions need to be specified (at least those that are "shared" between multiple factor distributions). Currently, there is also a limitation that compound random variable representations have to be additionally passed to ProdCPdf; in future, ProdCPdf will be able to infer compound random variables from factor distributions. Following code example constructs a simple probability density function from (3.13):

$$p(a, b) = p_1(a|b)p_2(b) \tag{3.13}$$

ProdCPdf example

```
# prepare random variables:
a, b = RVComp(m, "name of a"), RVComp(n, "b name")
p_1 = SomeCPdf(..., rv=RV(a), cond_rv=RV(b))
p_2 = OtherPdf(..., rv=RV(b))
p = ProdCPdf((p_1, p_2), rv=RV(a, b), cond_rv=RV()) # empty cond_rv

# version 0.4 of PyBayes will allow:
p = ProdCPdf((p_1, p_2))
```

PyBayes also provides a multivariate uniform distribution which is implemented by the *UniPdf* class.

3.2.6 Bayesian Filters

Bayesian filters are the *raison d'être* of PyBayes. It turned out however that with a solid framework of probability density functions, their implementation is rather straightforward. All filters in PyBayes extend an abstract class *Filter* that acts as a prototype of all filters. Filter defines following methods that can/should be implemented by subclasses:

- `bayes(yt : vector, cond : vector = None)`
  compute posterior probability density function given the observation `yt`; semantics of the optional `cond` parameter are defined by filter implementations. The method name comes from the fact that computing the posterior probability density function involves applying (exact or approximate) Bayes rule; see (1.4) on page 3.
- `posterior() : Pdf`
  return a reference to the posterior probability density function $p(x_t|y_{1:t})$ (1.5) (p. 3).
- `evidence_log(yt : vector) : float`
  return logarithm of the marginal likelihood $p(y_t|y_{1:t-1})$ (1.6) (p. 4) evaluated in point `yt`. Subclasses may choose not to implement this method if it is not feasible.

Fists subclass of Filter is the *KalmanFilter* class that implements slightly extended version of the Kalman filter presented in the first chapter — KalmanFilter can optionally accept control vector in its `bayes` method (passed through the `cond` parameter) making it suitable also for the theory of Bayesian decision-making.

Speaking about the particle filter family, it has been suggested in [15] that the particle filter and the marginalized particle filter can be merged into one general class using a recursive structure of classes representing the Bayes rule (e.g. Filter in PyBayes). This approach has not been used in PyBayes for performance and simplicity reasons. On the other hand, particle filters in PyBayes offload much work to probability density functions in PyBayes where code is reused thanks to AbstractEmpPdf.

The *ParticleFilter* class implements a simple version of the particle filter as presented in the first chapter. ParticleFilter takes the process model $p(x_t|x_{t-1})$ and the observation model $p(y_t|x_t)$ distributions in constructor (along with the initial density and number of particles) and employs `resample()` and `normalise_weights()` of the EmpPdf class that it uses for the posterior distribution. ParticleFilter currently doesn't support specifying the proposal density, although it is planned in future.

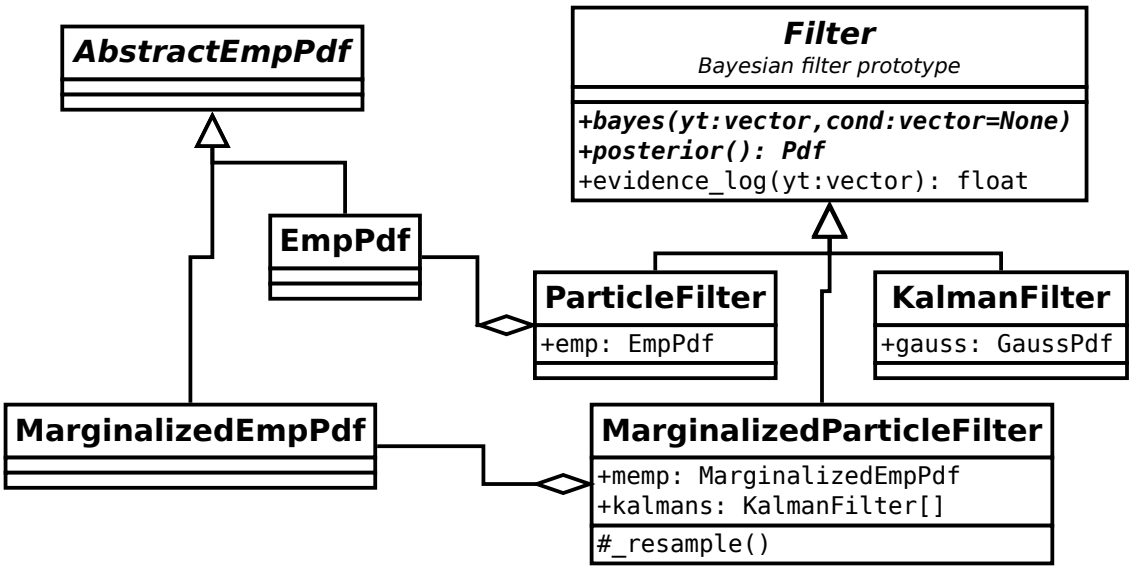

Figure 3.6: Class diagram of Bayesian filters

MarginalizedParticleFilter also implements the respective algorithm shown in the first chapter and offloads some work to its posterior probability density function, MarginalizedEmpPdf, but has to provide its own array of KalmanFilter objects. The MarginalizedParticleFilter class ensures that the i-th Kalman filter shares its posterior Gaussian distribution with MarginalizedEmpPdf's i-th particle. This is the reason why resampling cannot be done in MarginalizedEmpPdf and is instead performed in the `_resample()` method of MarginalizedParticleFilter that makes use of the `get_resample_indices()` method of AbstractEmpPdf.

In constructor MarginalizedParticleFilter accepts the initial distribution of the state vector $p(x_0) = p(a_0, b_0)$, process model of the empirical part of the state vector $p(b_t|b_{t-1})$, the class of the desired Kalman filter implementation (that has to be a subclass of KalmanFilter) and its parameters in form of a Python dictionary. That way, thanks to Python capabilities, MarginalizedEmpPdf will support other variants of the Kalman filter in future without a need to be changed. It also means that the

observation model is specified using the Kalman filter implementation and parameters for it, which makes MarginalizedParticleFilter more flexible. The process model is given by the combination of $p(b_t|b_{t-1})$ and supplied Kalman filter implementation (and its parameters). Current implementation hard-codes b_t to be the observation and process noise of the modelled system; this silly limitation will be lifted in future where MarginalizedParticleFilter will pass the b_t vector as the `cond` argument to the `bayes()` method of the specified Kalman filter implementation.

A diagram of filtering classes in shown in the Figure 3.6.

3.2.7 Wrappers

Favourable performance was one of the criteria for the desired library for Bayesian filtering. When the performance of the KalmanFilter class was benchmarked with a small system (both state and observation vectors were 2-dimensional); it was discovered that a great portion of total run time was spent in the boilerplate code of NumPy functions `dot()` (for matrix product) and `inv()` (for matrix inversion). Even though NumPy uses BLAS and LAPACK internally, the time spent in the intermediate Python code was unacceptable (it was probably made more visible due to the small size of the system); see the subsection 2.6.1 on page 22 for information about Cython ↔ NumPy co-operation.

Fortunately, a project that approaches popular BLAS and LAPACK functions more directly was found: Shane Legg's Tokyo[14] wraps BLAS (and a couple of LAPACK) procedures in Cython using NumPy's `ndarray` data-type. Quick tests have shown great speed-ups — the mentioned functions ceased to be performance bottlenecks. It was therefore decided that the Tokyo project should be used in the Cython mode of PyBayes and it was forked[15] on github to provide a couple of bug-fixes we made to the public. For convenience, Tokyo is also bundled with PyBayes releases and is built along it to reduce dependencies on obscure libraries.

A special approach in PyBayes has been taken in order to support both Python and Cython mode: wrapper modules for `numpy` and `numpy.linalg` were created in the `pybayes.wrappers` package; Python versions of the wrappers (.py files) just import appropriate NumPy modules. Cython versions do nearly the same, but provide own implementations (that call Tokyo) of the offending NumPy functions (and delegate the rest to NumPy). Other code then imports `wrappers._numpy` instead of `numpy`, likewise for `numpy.linalg`.

3.3 Documentation and Testing

Second pillar of each well-written software library is, in our belief, its *documentation*; the first one being the library design and the actual implementation being no better

[14] http://www.vetta.org/2009/09/tokyo-a-cython-blas-wrapper-for-fast-matrix-math/

[15] https://github.com/strohel/Tokyo

than the third pillar (in our view, the implementation can be easily modified in a well-designed library). PyBayes reflects that and puts great emphasis on the API Documentation, that is accessible on-line at http://strohel.github.com/PyBayes-doc/.

Documentation is generated almost solely from the documenting comments — Python "Docstrings" as defined in PEP 257,[16] using the Sphinx Python documentation generator.[17] Sphinx supports additional syntax in Docstrings and is able to generate documentation in a wide range of formats including HTML, Qt Help,[18] Devhelp,[19] LaTeX, PDF, man-pages and many others. One very valuable feature of Sphinx is the ability to parse LaTeX-encoded mathematics embedded directly into Docstrings; these are then rendered to images in HTML output for example.

Every publicly available class and method in PyBayes is extensively documented and enhanced with mathematical formulas where appropriate. We believe that this approach makes PyBayes more easily usable by mathematicians and eliminates any possible misunderstanding of the textual description of classes and methods. An example of how Docstrings look like in code can be seen in the Figure 3.7. We must say we are very satisfied with Sphinx can only recommend using it.

As noted in the discussion about dynamically-typed languages, almost all programmer errors in such languages are only discovered at runtime, therefore a need for a comprehensive test-suite was stressed out. PyBayes follows this advice and provides two packages that accomplish testing:

- `pybayes.tests`
  package contains unit-tests of virtually all PyBayes classes and methods. Unit-testing evaluates classes and methods *in isolation* (to highest possible extent) and therefore forms an excellent tool for finding precise location of possible bugs. Unit-testing should last no longer than a few seconds so that it can be run on per-commit basis. One problem faced in PyBayes are non-deterministic methods such as `CPdf.sample()` or `bayes()` methods of particle filters. `sample()` methods are currently tested by generating high enough number of particles and then comparing their mean and variance with theoretical values. Another option would be mocking the random number generator to force deterministic results, this could however produce false-positives when an implementation of a given `sample()` method changed.
  PyBayes currently contains 99 unit-tests that run in approximately 0.2 seconds.
- `pybayes.stresses`
  package contains so-called stress-tests — longer-running procedures that test greater portion of code and cooperation if individual modules. Results of stresses are not intended be checked automatically, they rather require human evaluation. PyBayes currently has three stresses, one for each Bayesian filter implementation, that are run with various parameters to ensure that the filters produce valid-looking results, to measure their performance, and (for particle filters) to test

[16] Python Enhancement Proposal 257: http://www.python.org/dev/peps/pep-0257/
[17] http://sphinx.pocoo.org/
[18] http://doc.qt.nokia.com/qthelp-framework.html
[19] http://live.gnome.org/devhelp

their convergence as the number of particles increases.

To ensure that all code-paths are properly tested, it is advisable to employ a *code coverage* tool that shows which code statements were visited during tests. We've used Ned Batchelder's excellent *coverage.py*[20] to discover that 86% statements in the `pdfs` module and 83% statements in the `filters` module are covered by tests and stress-tests combined. An example output of coverage.py is shown in the Figure 3.7, where everything except one code-path throwing an exception is covered by a test.

```
196     def indexed_in(self, super_rv):
197         """Return index array such that this rv is indexed in **super_rv**, which
198         must be a superset of this rv. Resulting array can be used with :func:`numpy.take`
199         and :func:`numpy.put`.
200 
201         :param super_rv: returned indices apply to this rv
202         :type super_rv: :class:`RV`
203         :rtype: 1D :class:`numpy.ndarray` of ints with dimension = self.dimension
204         """
205         ret = np.empty(self.dimension, dtype=int)
206         ret_ind = 0  # current index in returned index array
207         # process each component from target rv
208         for comp in self.components:
209             # find associated component in source_rv components:
210             src_ind = 0  # index in source vector
211             for source_comp in super_rv.components:
212                 if source_comp is comp:
213                     ret[ret_ind:] = np.arange(src_ind, src_ind + comp.dimension)
214                     ret_ind += comp.dimension
215                     break;
216                 src_ind += source_comp.dimension
217             else:
218                 raise AttributeError("Cannont find component "+str(comp)+" in source_rv.co
219         return ret
```

Figure 3.7: coverage.py shows that one code-path in indexed_in() is uncovered

3.4 Performance Comparison with BDM

The chapter about the PyBayes library is concluded by a benchmark of four various Kalman filter implementations from PyBayes and BDM [19]. All tested implementations use exactly the same algorithm and equivalent data-types, neither is explicitly parallelised (however, see notes about MATLAB), operate on the same data and gave exactly the same results as shown in the Figure 1.1 on page 6. Following versions of involved software were used:

Python 2.7.1
GNU C Compiler 4.4.5; `-O2` optimisation flag used when compiling C files
Cython 0.14.1+ (git revision 0.14.1-1002-g53e4c10)
MATLAB 7.11.0.584 (R2010b) 64-bit (glnxa64)

PyBayes 0.3
BDM SVN revision r1378

[20] http://nedbatchelder.com/code/coverage/

The test was performed on a 64-bit dual-core Intel Core i5-2520M CPU clocked at 2.50 Ghz with Intel Turbo Boost and Hyper-threading enabled; operating system is Gentoo Linux compiled for the x86_64 platform.

The test consists of running 3000 iterations of the Kalman filter with various state-space dimensions: 2, 30 and 60; observation vector is has the same dimensionality as the state vector. Wall-clock time needed to complete all iterations is measured. Each implementation was tested 10 times, mean values are shown; to measure variance across runs, relative sample standard deviation s_{rel} computed using (2.5) (page 27) was measured. Additionally, relative speed-up with regards to reference version **PyBayes Cy** is displayed for illustration. Following versions of Kalman filter/implementation environments were under test:

PyBayes Py
: KalmanFiler class from PyBayes `pybayes/filters.py`; Python build

PyBayes Cy
: KalmanFiler class from PyBayes `pybayes/filters.py`; Cython build

MATLAB imper.
: Imperative MATLAB implementation where the whole algorithm is written in a single for loop; comes from BDM, file `library/tests/stressuite/kalman_stress.m`. While not explicitly parallelised, later experiments shown that MATLAB implicitly parallelised the code behind curtain at least in higher-dimensional cases.

MATLAB o-o
: Object-oriented MATLAB implementation from BDM where the filter and Gaussian probability density function is represented using MATLAB classes; file `applications/bdmtoolbox/mex/mexKalman.m`.

BDM
: Object-oriented C++ class KalmanFull from BDM implemented in `/library/bdm/estim/kalman.cpp`.

Benchmark results are shown in tables 3.1 to 3.3. The greatest variance in results is achieved in a small (2-dimensional) system, where the C++ version is the fastest, outperforming Cython build of PyBayes by 260%, and object-oriented MATLAB version is embarrassingly 15× slower than the PyBayes Cy.

Raising the dimensionality of state-space to 30 produced more even results with object-oriented MATLAB still lagging behind, PyBayes Cy and C++ version from BDM being nearly equal and imperative MATLAB version taking lead by being approximately 30% faster.

A huge 60-dimensional system sees PyBayes Py, PyBayes Cy and BDM versions being even closer, imperative MATLAB version extending its advantage and object-oriented MATLAB version still largely unusable due to its poor performance.

The chart shown in the Figure 3.8 allows for some speculations about possible reasons of performance differences. First, it seems that the Python version adds moderate per-statement overhead that doesn't raise with number of dimensions (we blame NumPy for Python version being on par with the C++ and Cython versions) while object-oriented MATLAB adds **enormous** overhead that worsens slightly with

	PyBayes Py	PyBayes Cy	MATLAB imper.	MATLAB o-o	BDM
time [s]	0.254	0.091	0.069	1.378	0.026
s_{rel}	4.4%	4.2%	8.5%	4.1%	9.3%
speedup	0.4×	1.0×	1.3×	0.1×	3.6×

Table 3.1: Performance of Kalman filters: 2-dimensional state-space

	PyBayes Py	PyBayes Cy	MATLAB imper.	MATLAB o-o	BDM
time [s]	0.689	0.535	0.424	1.780	0.518
s_{rel}	3.0%	5.4%	5.9%	2.7%	5.7%
speedup	0.8×	1.0×	1.3×	0.3×	1.0×

Table 3.2: Performance of Kalman filters: 30-dimensional state-space

	PyBayes Py	PyBayes Cy	MATLAB imper.	MATLAB o-o	BDM
time [s]	2.120	1.816	1.274	3.849	1.948
s_{rel}	2.4%	2.6%	6.3%	9.9%	2.1%
speedup	0.9×	1.0×	1.4×	0.5×	0.9×

Table 3.3: Performance of Kalman filters: 60-dimensional state-space

number of dimensions.

The clear advantage of the imperative MATLAB version can be accounted to its capability to parallelise code behind the scenes, in our belief. Our informal late tests have shown that it performs very close to the PyBayes Cy version on uniprocessor systems.

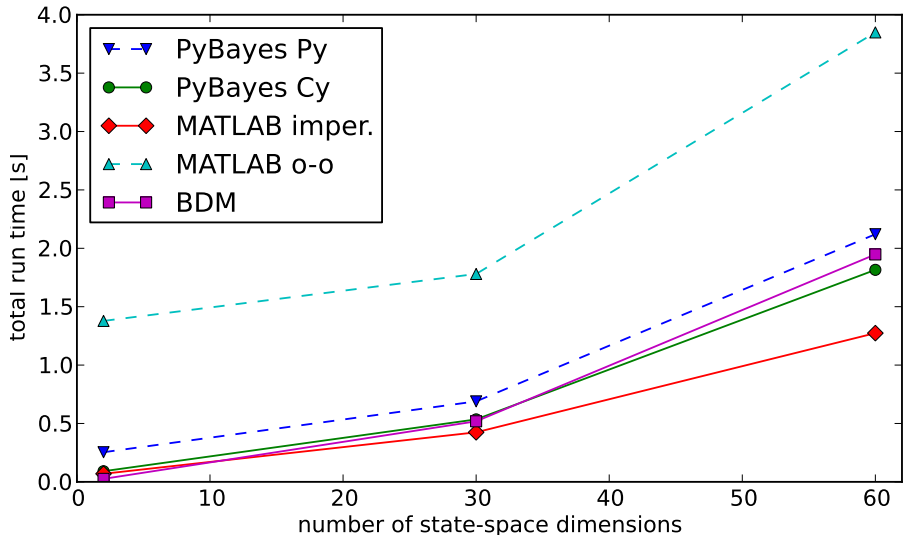

Figure 3.8: Run time against dimensionality of various Kalman filter implementations

Conclusion

The theory of Bayesian filtering is introduced in the first chapter and the *optimal Bayesian solution* of the problem of recursive estimation is derived. Continues a survey of well-known Bayesian filtering methods — the Kalman filtering, particle filtering and the marginalized particle filtering is described and properties of individual algorithms are discussed.

The second chapter contains a software analysis performed with the aim to identify the best approach to software development and programming language for a desired library for Bayesian filtering. Object-oriented approach is chosen along with the Python programming language, which is found optimal except its potentially significant computational overhead. Cython is evaluated for the task to improve Python performance with great success: a simple Python algorithm was 60× faster when compiled using Cython.

The last chapters presents the PyBayes library that was developed as a part of this thesis. PyBayes builds on the software analysis performed in the previous chapter and is therefore object-oriented and uses Python/Cython combination as its implementation environment and implements all presented Bayesian filtering methods. To compare performance of Python/Cython combination in a real-world example, the Kalman filter from PyBayes is benchmarked against MATLAB and C++ implementations from BDM [19] with favourable results.

We believe that the **key contributions** of this thesis are:

- The performed software analysis, that can be reused for a wide variety of software projects. In particular, we have shown that the choice of a high-level and convenient language such as Python is *not necessarily* the enemy of speed. The analysis includes benchmarks with quite surprising results that show that Cython and PyPy are great speed boosters of Python.
- The PyBayes library itself. While it is not yet feature-complete, it provides a solid base for future development and is unique due to its dual-mode approach: it can be both treated as ordinary Python package with all the convenience it brings or compiled using Cython for performance gains.

Future work includes extending PyBayes with more filtering algorithms (non-liner Kalman filter variants etc.) in the long term and fixing little inconveniences that currently exist in PyBayes in the sort term; version 0.4 that would incorporate all future changes mentioned in the third chapter is planned to be released within a few months. We are also looking forward to incorporate emerging projects into our software analysis, for example the PyPy project looks very promising.

List of Figures

Bibliography

[1] M. Sanjeev Arulampalam, Simon Maskell, and Neil Gordon. A tutorial on particle filters for online nonlinear/non-gaussian bayesian tracking. *IEEE Transactions on Signal Processing*, 50:174–188, 2002. 3, 4, 5, 6, 8

[2] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D.S. Seljebotn, and K. Smith. Cython: The best of both worlds. *Computing in Science Engineering*, 13(2):31–39, march-april 2011. 22

[3] Stefan Behnel, Robert W. Bradshaw, and Dag Sverre Seljebotn. Cython tutorial. In Gaël Varoquaux, Stéfan van der Walt, and Jarrod Millman, editors, *Proceedings of the 8th Python in Science Conference*, pages 4–14, Pasadena, CA USA, 2009. 23

[4] D. Crisan and A. Doucet. A survey of convergence results on particle filtering methods for practitioners. *Signal Processing, IEEE Transactions on*, 50(3):736–746, 2002. 7

[5] K. Gadeyne. BFL: Bayesian Filtering Library. http://www.orocos.org/bfl, 2001. 18

[6] F. Gustafsson, F. Gunnarsson, N. Bergman, U. Forssell, J. Jansson, R. Karlsson, and P.J. Nordlund. Particle filters for positioning, navigation, and tracking. *Signal Processing, IEEE Transactions on*, 50(2):425–437, 2002. 1

[7] R. Hofman, V. Šmídl, and P. Pecha. Data assimilation in early phase of radiation accident using particle filter. In *The Fifth WMO International Symposium on Data Assimilation*, Melbourne, Australia, 2009. 1

[8] R. Hofman and Šmídl V. Assimilation of spatio-temporal distribution of radionuclides in early phase of radiation accident. *Bezpečnost jaderné energie*, 18:226–228, 2010. 1

[9] Y. Levanoni and E. Petrank. An on-the-fly reference counting garbage collector for Java. *ACM Transactions on Programming Languages and Systems*, 28(1), January 2006. 16

[10] P. Pecha, Hofman R., and V. Šmídl. Bayesian tracking of the toxic plume spreading in the early stage of radiation accident. In *Proceedings of the 2009 European Simulation and Modelling Conference*, Leicester, GB, 2009. 1

[11] V. Peterka. Bayesian approach to system identification. In P. Eykhoff, editor, *Trends and Progress in System identification*, pages 239–304. Pergamon Press, Oxford, 1981. 4

[12] T. B. Schön, F. Gustafsson, and P.-J. Nordlund. Marginalized particle filters for mixed linear/nonlinear state-space models. *Signal Processing, IEEE Transactions on*, 53(7):2279–2289, july 2005. 9

[13] T. B. Schön, R. Karlsson, and F. Gustafsson. The marginalized particle filter — analysis, applications and generalizations, 2006. 9

[14] Dag Sverre Seljebotn. Fast numerical computations with cython. In Gaël Varoquaux, Stéfan van der Walt, and Jarrod Millman, editors, *Proceedings of the 8th Python in Science Conference*, pages 15–22, Pasadena, CA USA, 2009. 24

[15] V. Šmídl. Software analysis unifying particle filtering and marginalized particle filtering. In *Proceedings of the 13th International Conference on Information Fusion*. IET, 2010. 4, 40

[16] B. Stroustrup. *The C++ programming language, Third Edition*. Addison-Wesley, 2000. 17

[17] S. Thrun, W. Burgard, and D. Fox. *Probabilistic Robotics (Intelligent Robotics and Autonomous Agents)*. 2005. 1

[18] V. Šmídl. *Software analysis of Bayesian distributed dynamic decision making*. PhD thesis, University of West Bohemia, Faculty of Applied Sciences, Pilsen, Czech Republic, Plzeň, 2005. 1

[19] V. Šmídl and M. Pištěk. Presentation of Bayesian Decision Making Toolbox (BDM). In M. Janžura and J. Ivánek, editors, *Abstracts of Contributions to 5th International Workshop on Data — Algorithms — Decision Making*, page 37, Praha, 2009. 18, 19, 43, 46