Contents

Notation

Throughout this text, following notation is used

$\mathbb{N}$	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 $(a_t, a_{t+1} \dots a_{t'-1}, a_{t'})$
$p(a_t)$	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 $\mathbb{R}^n \to \mathbb{R}$; $\int_{\operatorname{supp} p} p(x_1, x_2 \dots x_n) \, \mathrm{d}x_1 \mathrm{d}x_2 \dots \mathrm{d}x_n = 1$

[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

TODO motivatin for bayes filtering + a need for a convenient library (rapid prototyping vs. speed)

applications: robotics, navigation, + tracking of toxic plume after radiation accident.

Decision-making being a logical and natural “next step” - beyond the scope of this text.

[proposed citations:[15, 6, 8, 7, 10]]

na co se nezamerujeme: MCMC, Bayes networks

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

[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).

density function can be further simplified into (1.5).

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

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. [13]

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

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] 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.

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 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.

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.

1.4 Particle Filter

Particle filters represent 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 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

[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.

[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.

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 $i \in \mathbb{N} \quad i \leq N$, 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.

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.

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

[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.

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

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.

1.5 Marginalized Particle Filter

TODO: MPF. If in lack of time, write short mention and merge into PF section.

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

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 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 multiprocessor 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. 16) 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. 16) 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

[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).

downside, in author's opinion. We believe that these disadvantages are greatly outweighed 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. 9) 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.

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

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 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 26).

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 de-

[5] Integrated Development Environment

velopment and portable distribution options. Interpreted languages have been historically associated with considerable processing overhead, but *just-in-time compilation*[6] 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 `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. 24) 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

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

[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/

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 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].

[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.

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.

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. [14]

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, in-

[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/

credible 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 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 [17] 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 26.

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

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

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

[20] http://itpp.sourceforge.net/

[21] http://eigen.tuxfamily.org/

[22] Open Multi-Processing API and libraries: http://openmp.org/

[23] http://bayesclasses.sourceforge.net/

[24] http://www.mathworks.com/products/matlab/

[25] http://www.mathworks.com/

(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
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 copy-

[26] http://www.mathworks.com/products/matlab/whatsnew.html

[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).

ing 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 26.

We consider presented drawbacks significant and therefore decided not to use the MATLAB language for the desired Bayesian filtering library. BDM library [17] 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.

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 .NET Framework, Cython which is described in greater detail in the next section or PyPy mentioned briefly. 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:

[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/

[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/

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.[38] For example a statement `x <= y <= z` has its mathematical meaning, which is unusual for programming languages.

NumPy, SciPy, Matplotlib

NumPy[39] is the de facto standard Python library for numeric computing; NumPy provides N-dimensional array type that is massively supported in very high number of projects. Parts of NumPy are written in C and Cython for speed. SciPy[40] extends NumPy with more numerical routines. Matplotlib[41] is powerful plotting library that natively supports SVG output. Combining these three and Python gives a very vital MATLAB alternative.

interoperability

CPython makes it possible to write modules[42] in C;[43] Cython makes it easy and convenient. SciPy contains procedures to load and save data in MATLAB .mat format.

On the other hand, a few downsides exist:

overhead

CPython implementation incurs significant computational overhead especially in

NumPy.... parallelisation (approaches, improvements in Py 3.2) - GIL.. Py3k

PyMC - konkurenci knihovna? – spise ne

2.6 Cython

general info etc... extension types, building, ease of interfacing C (and F) code, .pxd files, NumPy support

[citations:[3, 12, 2]]

2.6.1 Gradual Optimisation

how can optimisaion be approached (gradually) and why this approach is superior

[see c_cy_py...]

[38] http://python.org/about/quotes/

[39] http://www.numpy.org/

[40] http://www.scipy.org/

[41] http://matplotlib.sourceforge.net/

[42] module in python sense is a code unit with its own namespace, normally each module corresponds to a .py file.

[43] http://docs.python.org/extending/index.html

2.6.2 Parallelisation

integrate_python_cython patched with OpenMP (13x speedup in 16-core system)

prange CEP – implemented!

[see c_cy_py...]

2.6.3 Pure Python mode

About it and why it should be used in a hypothetical bayesian python library

2.6.4 Limitations

2 types:

not-supported code (few cases, but bad, ongoing work)

not-optimised code (much more work needed, but not hard to fix in most cases)

- exception handling (functions returning void etc)
- limitations of pure python mode in regards to traditional .pyx files

2.6.5 Performance comparison with C and Python

benchmark_c_cy_py

Chapter 3

The PyBayes Library

Introduction, general directions, future considerations

+ open development on github, open-source

3.1 Interpreted and Compiled

3.2 Library Layout

[proposed citation: [16]]

3.2.1 Random Variable Meta-representation

Why it is needed (ref to ProdCPdf)

3.2.2 Probability Density Functions

Nice UML diagrams! (better more smaller UMLs than one big) One for general pdf layut, one for AbstractGaussPdf family, one for AbstractEmpPdf family

3.2.3 Bayesian Filters

UML

Nice graph of a run of a particle filter (Mirda has the plotting code)

similar of marginalized particle filter? (gausses would be plotted vertically)

[mention this:[13]]

3.3 Documentation, Testing and Profiling

TODO: move above Library Layout?

Documenting PyBayes using Sphinx, approach to documentation (mathematician-oriented), math in documentation

Testing - the separation of

- tests: test one class in isolation, quick, determinism (would be good, not achievable)

- stresses: test a great portion of code at once, run longer, non-determinism..

Note about coverage.py!!

Profiling python/cython - how, existing support in PyBayes

- how to correct profiling-induced overhead

3.4 Performance Comparison with BDM

[skip if in time press]

Conclusion

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. 6, 7, 8, 9, 10, 11

[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. 23

[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. 10

[5] K. Gadeyne. BFL: Bayesian Filtering Library. `http://www.orocos.org/bfl`, 2001. 20

[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. 4

[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. 4

[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. 4

[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. 18

[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. 4

[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. 8

[12] 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. 23

[13] V. Šmídl. Software analysis unifying particle filtering and marginalized particle filtering. In *Proceedings of the 13th International Conference on Information Fusion*. IET, 2010. 7, 25

[14] B. Stroustrup. *The C++ programming language, Third Edition*. Addison-Wesley, 2000. 19

[15] S. Thrun, W. Burgard, and D. Fox. *Probabilistic Robotics (Intelligent Robotics and Autonomous Agents)*. 2005. 4

[16] 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. 25

[17] 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. 20, 22