České vysoké učení technické v Praze
Fakulta jaderná a fyzikálně inženýrská

Katedra matematiky
Obor: Inženýrská Informatika
Zaměření: Softwarové inženýrství a matematická informatika

Prostředí pro implementaci algoritmů Bayesovské filtrace

Implementation environment for Bayesian filtering algorithms

BAKALÁŘSKÁ PRÁCE

Vypracoval: Matěj Laitl
Vedoucí práce: Ing. Václav Šmídl, Ph.D.
Rok: 2011

Před svázáním místo téhle stránky vložíte zadání práce s podpisem děkana (bude to jediný oboustranný list ve Vaší práci) !!!!

Prohlášení

Prohlašuji, že jsem svou bakalářskou práci vypracoval samostatně a použil jsem pouze podklady (literaturu, projekty, SW atd.) uvedené v přiloženém seznamu.

V Praze dne .....

.....
Matěj Laitl

Poděkování

Děkuji Ing. Václavu Šmídlovi, Ph.D. za vedení mé bakalářské práce a za podnětné návrhy, které ji obohatily.

Matěj Laitl

Název práce:
Prostředí pro implementaci algoritmů Bayesovské filtrace

Autor: Matěj Laitl

Obor: Inženýrská Informatika
Druh práce: Bakalářská práce

Vedoucí práce: Ing. Václav Šmídl, Ph.D.
Oddělení adaptivních systémů
Ústav teorie informace a automatizace
Akademie věd České republiky
Konzultant: —

Abstrakt: TODO Abstrakt práce (cca 7 vět, min. 80 slov)

Klíčová slova: TODO, klíčová slova, max. 5

Title:
Implementation environment for Bayesian filtering algorithms

Author: Matěj Laitl

Abstract: TODO English abstract

Key words: TODO, key, words

Contents

Introduction

TODO motivatin for bayes filtration + 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:[10, 4, 6, 5, 7]]

Notation

Throughout this text, following notation is used

$\mathbb{N}$	set of natural numbers
$\mathbb{R}$	set of real numbers
t	discrete time moment; $t \in \mathbb{N}$
a_t	value of quantity a at time t; $a_t \in \mathbb{R}^n, n \in \mathbb{N}$ unless noted otherwise, x_t denotes state vector at time t and y_t denotes observation vector at time t
$a_{i:j}$	sequence of quantities $(a_i, a_{i+1} \dots a_{j-1}, a_j)$

Chapter 1

Basics of Recursive Bayesian Estimation

In following sections the problem of recursive Bayesian estimation is stated and its anylitical 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 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 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

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

As a first step, we observe that probability density function $p(x_t|x_{t-1})$ can be derived from (1.1) (given the distribution of v_k) and that $p(y_t|x_t)$ can be derived from (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 $p(x_0|\{\})$, the initial probability density function, are known in order to be able to make the transition $t-1 \rightarrow t$.

In the first stage called *prediction*, *a priori* 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 splitted.

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

But $p(x_t|x_{t-1}, y_{1:t-1})$ equals $p(x_t|x_{t-1})$ (TODO: why? citation? linked to (1.1)), leaving us with the result (1.3).

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

[Main source for these 2 sections:[1]]

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.

1.3 Kalman Filter

1.4 Particle Filter

1.5 Marginalized Particle Filter

Chapter 2

Software analysis

2.1 Requirements

Ideal library for Bayesian filtering would posses following properties...:

2.2 Programming paradigms

Interpreted vs. Compiled

Object-oriented, procedural and Functional

pass-by reference vs. copy-on-write (Matlab)

2.3 Survey of Existing Libraries for Bayesian estimation/decision making

Brief survey of existing libraries ... not fullfilling all requirements..

The need to implement a new one :-)

2.4 C++

BDM

- advantages of C (speed, C prevalence (many optimised libraries, BLAS, LAPACK.., OpenMP)

- disadvantages of C in our "situation" (steep learning curve, coplexity because of

low-levelness high initial barriers (need to have compiler, libraries...), inconveniently long edit/build/test process)

2.5 Matlab

BDM (partially?)

- advantages (popularity, existing toolboxes, rapid development (high-level)

- disadv: strict copy-on-write, problematic object model (not in original design), difficulties interfacing existing C (F) code

2.6 Python

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

2.7 Cython

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

[citations:[3, 8, 2]]

2.7.1 Gradual Optimisation and Parallelisation

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

integrate_python_cython example ("100x" speedup for a special (very simple) case)

2.7.2 Parallelisation

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

prange CEP

2.7.3 Pure Python mode

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

2.7.4 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

2.8 Choice

python/cython was choosen ...

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: [11]]

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:[9]]

3.3 Documentation, Testing and Profiling

TODO: move above Library Layout?

Documenting 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..

Profiling python/cython - how, existing support in PyBayes

- how to correct profiling-induced overhead

3.4 Comparison with BDM

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. 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. 14

[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. 14

[4] 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. 8

[5] 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. 8

[6] 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. 8

[7] P. Pecha, Hofman R., and Šmídl V. 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. 8

[8] 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. 14

[9] V. Šmídl. Software analysis unifying particle filtering and marginalized particle filtering. In *Proceedings of the 13th International Conference on Information Fusion*. IET, 2010. 16

[10] S. Thrun, W. Burgard, and D. Fox. *Probabilistic Robotics (Intelligent Robotics and Autonomous Agents)*. 2005. 8

[11] 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. 16

Appendix

PyBayes API Documentation

Release 0.2

Matěj Laitl

May 02, 2011

Contents

Chapter 1

PyBayes

1.1 About

A long-term goal of PyBayes is to be the preferred python library for implementing Bayesian filtering (recursive estimation) and decision-making systems.

Already done are classes for both basic static and conditional probability densities ([c]pdfs) and a special cpdf representing a chain rule. Particle filter that uses cpdfs extensively is implemented, Kalman filter is also present.

Future plans include more specialised variants of Kalman/particle filters and speed optimisations. The project is also interesting technically as it is dual-mode: can be be used without cython at all or compiled to gain more speed - with nearly no code duplication.

Automatically generated **documentation** can be found at http://strohel.github.com/PyBayes-doc/

1.1.1 Licensing

PyBayes is currently distributed under GNU GPL v2+ license. The authors of PyBayes are however open to other licensing suggestions. (Do you want to use PyBayes in e.g. BSD-licensed project? Ask!)

1.2 Obtaining PyBayes

Development of PyBayes happens on http://github.com/strohel/PyBayes using git VCS and the most fresh development sources can be obtained using git. It should be noted that PyBayes uses git submodule to bundle Tokyo library, so the proper way to clone PyBayes repository would be:

```
# cd path/to/projects
# git clone git://github.com/strohel/PyBayes.git
Cloning into PyBayes...
```

```
(...)
# cd PyBayes
# git submodule update --init
Submodule 'tokyo' (git://github.com/strohel/Tokyo.git) registered for path 'tokyo'
Cloning into tokyo...
(...)
Submodule path 'tokyo': checked out '896d046b62cf50faf7faa7e58a8705fb2f22f19a'
```

When updating your repository (using `git pull`), git should inform you that some submodules have became outdated. In that case you should issue `git submodule update`.

1.3 Installing PyBayes

PyBayes uses standard Python distutils for building and installation. Follow these steps in order to install PyBayes:

- download PyBayes, let's assume PyBayes-0.1.tar.gz filename
- unpack it:

  `tar -xvf PyBayes-0.1.tar.gz`
- change directory into PyBayes source:

  `cd Pybayes-0.1`
- build and install (either run as root or install to a user-writeable directory [1]):

  `./setup.py install`

And you're done! However, if you want PyBayes to be *considerably faster*, please read the following section.

1.3.1 Advanced installation options

PyBayes can use Cython to build itself into binary Python module. Such binary modules are transparent to Python in a way that Python treats then as any other modules (you can `import` them as usual). Interpreter overhead is avoided and many other optimisation options arise this way.

In order to build optimised PyBayes, you'll additionally need:

- Cython Python to C compiler
- working C compiler (GCC on Unix-like systems, MinGW or Microsoft Visual C on Windows [2])
- NumPy numerical library for Python, version 1.5 or greater (NumPy is needed also in Python build, but older version suffice in that case)

[1] http://docs.python.org/install/#alternate-installation
[2] http://docs.cython.org/src/quickstart/install.html

- On some Debian-based Linux distributions (Ubuntu) you'll need python-dev package that contains `Python.h` file that is needed by PyBayes

Proceed with following steps:

1. Install all required dependencies. They should be already available in your package manager if you use a modern Linux Distribution.
2. Unpack and install PyBayes as described above, you should see following messages during build:

   ```
   Notice:  Cython found.

   Notice:  NumPy found.
   ```

   - in order to be 100% sure that optimised build is used, you can add `--use=cython=yes` option to the `./setup.py` call. You can force pure Python mode even when Cython is installed, pass `--use=cython=no`. By default, PyBayes auto-detects Cython and NumPy presence on system.
   - if you plan to profile code that uses optimised PyBayes, you may want to embed profiling information into PyBayes. This can be accomplished by passing `--profile=yes` to `./setup.py`. The default is to omit profiling information in order to avoid performance penalties.

1.3.2 Building Documentation

There is no need to build documentation yourself, an online version is at http://strohel.github.com/PyBayes-doc/

PyBayes uses Sphinx to prepare documentation, version 1.0 or greater is required. The documentation is built separately from the python build process. In order to build it, change directory to *doc/* under PyBayes source directory (`cd [path_to_pybayes]/doc`) and issue `make` command. This will present you with a list of available documentation formats. To generate html documentation, for example, run `make html` and then point your browser to *[path_to_pybayes]/doc/_build/html/index.html*.

> PyBayes docs contain many mathematical expressions; Sphinx can use LaTeX to embed them as images into resulting HTML pages. Be sure to have LaTeX-enabled Sphinx if you want to see such nice things.

1.4 Testing PyBayes

Once PyBayes is installed, you may want to run its tests in order to ensure proper functionality. The `examples` directory contains `run_tests.py` and `run_stresses.py` scripts that execute all PyBayes tests and stress tests respectively. Run these scripts with `-h` option to see usage.

> *Note: running tests from within source directory is discouraged and unsupported.*

For even greater convenience, `examples/install_test_stress.py` python script can clear, build, install, test, stress both Python and Cython build in one go. It is especially suitable for PyBayes hackers. Run `install_test_stress.py -h` to get usage information. Please be sure to add `--clean` or `-c` flag when you mix Python and Cython builds.

Chapter 2

Probability Density Functions

This module contains models of common probability density functions, abbreviated as pdfs.

All classes from this module are currently imported to top-level pybayes module, so instead of `from pybayes.pdfs import Pdf` you can type `from pybayes import Pdf`.

2.1 Random Variables and their Components

class `pybayes.pdfs.RV(`**components*`)`

Representation of a random variable made of one or more components. Each component is represented by `RVComp` (page 7) class.

Variables

- **dimension** (*int*) – cummulative dimension; do not change
- **name** (*str*) – pretty name, can be changed but needs to be a string
- **components** (*list*) – list of RVComps; do not change

Please take into account that all RVComp comparisons inside RV are instance-based and component names are purely informational. To demonstrate:

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

Right way to do this would be:

```
>>> a = RVComp(1, "arbitrary pretty name for a")
>>> rv = RV(a)
>>> ...
>>> rv.contains(a)
True
```

`__init__(*components)`
Initialise random variable meta-representation.

Parameters ***components** (RV (page 5), RVComp (page 7) or a sequence of RVComp (page 7) items) – components that should form the random variable. You may also pass another RVs which is a shotrcut for adding all their components.

Raises **TypeError** invalid object passed (neither a RV (page 5) or a RVComp (page 7))

Usual way of creating a RV could be:

```
>>> x = RV(RVComp(1, 'x_1'), RVComp(1, 'x_2'))
>>> x.name
'[x_1, x_2]'
>>> xy = RV(x, RVComp(2, 'y'))
>>> xy.name
'[x_1, x_2, y]'
```

`contains(component)`
Return True if this random variable contains the exact same instance of the **component**.

Parameters **component** (RVComp (page 7)) – component whose presence is tested

Return type bool

`contains_all(test_components)`
Return True if this RV contains all RVComps from sequence **test_components**.

Parameters **test_components** (sequence of RVComp (page 7) items) – list of components whose presence is checked

`contains_any(test_components)`
Return True if this RV contains any of **test_components**.

Parameters **test_components** (sequence of RVComp (page 7) items) – sequence of components whose presence is tested

`contained_in(test_components)`
Return True if sequence **test_components** contains all components from this RV (and perhaps more).

Parameters **test_components** (sequence of RVComp (page 7) items) – set of components whose presence is checked

`indexed_in(super_rv)`
Return index array such that this rv is indexed in **super_rv**, which must be a superset of this rv. Resulting array can be used with `numpy.take()` and `numpy.put()`.

Parameters **super_rv** (RV (page 5)) – returned indices apply to this rv

Return type 1D numpy.ndarray of ints with dimension = self.dimension

class pybayes.pdfs.RVComp(*dimension*, *name=None*)

Atomic component of a random variable.

Variables

- **dimension** (*int*) – dimension; do not change unless you know what you are doing
- **name** (*str*) – name; can be changed as long as it remains a string (warning: parent RVs are not updated)

__init__(*dimension*, *name=None*)

Initialise new component of a random variable RV (page 5).

Parameters

- **dimension** (*positive integer*) – number of vector components this component occupies
- **name** (*string or None*) – name of the component; default: None for anonymous component

Raises

- **TypeError** – non-integer dimension or non-string name
- **ValueError** – non-positive dimension

2.2 Probability Density Function prototype

class pybayes.pdfs.CPdf

Base class for all Conditional (in general) Probability Density Functions.

When you evaluate a CPdf the result generally also depends on a condition (vector) named *cond* in PyBayes. For a CPdf that is a Pdf (page 9) this is not the case, the result is unconditional.

Every CPdf takes (apart from others) 2 optional arguments to constructor: **rv** (RV (page 5)) and **cond_rv** (RV (page 5)). When specified, they denote that the CPdf is associated with a particular random variable (respectively its condition is associated with a particular random variable); when unspecified, *anonymous* random variable is assumed (exceptions exist, see ProdPdf (page 12)). It is an error to pass RV whose dimension is not same as CPdf's dimension (or cond dimension respectively).

Variables

- **rv** (*RV*) – associated random variable (always set in constructor, contains at least one RVComp)

- **cond_rv** (*RV*) – associated condition random variable (set in constructor to potentially empty RV)

While you can assign different rv and cond_rv to a CPdf, you should be cautious because sanity checks are only performed in constructor.

While entire idea of random variable associations may not be needed in simple cases, it allows you to express more complicated situations. Assume the state variable is composed of 2 components $x_t = [a_t, b_t]$ and following probability density function has to be modelled:

$$\begin{aligned} p(x_t|x_{t-1}) &:= p_1(a_t|a_{t-1}, b_t)p_2(b_t|b_{t-1}) \\ p_1(a_t|a_{t-1}, b_t) &:= \mathcal{N}(a_{t-1}, b_t) \\ p_2(b_t|b_{t-1}) &:= \mathcal{N}(b_{t-1}, 0.0001) \end{aligned}$$

This is done in PyBayes with associated RVs:

```
>>> a_t, b_t = RVComp(1, 'a_t'), RVComp(1, 'b_t')  # create RV components
>>> a_tp, b_tp = RVComp(1, 'a_{t-1}'), RVComp(1, 'b_{t-1}')  # t-1 case

>>> p1 = LinGaussCPdf(1., 0., 1., 0., RV(a_t), RV(a_tp, b_t))
>>> # params for p2:
>>> cov, A, b = np.array([[0.0001]]), np.array([[1.]]), np.array([0.])
>>> p2 = MLinGaussCPdf(cov, A, b, RV(b_t), RV(b_tp))

>>> p = ProdCPdf((p1, p2), RV(a_t, b_t), RV(a_tp, b_tp))

>>> p.sample(np.array([1., 2.]))
>>> p.eval_log()
```

`shape()`
Return shape of the random variable. `mean()` (page 8) and `variance()` (page 8) methods must return arrays of this shape.

Return type int

`cond_shape()`
Return shape of the condition.

Return type int

`mean(`*cond=None*`)`
Return (conditional) mean value of the pdf.

Return type `numpy.ndarray`

`variance(`*cond=None*`)`
Return (conditional) variance (diagonal elements of covariance).

Return type `numpy.ndarray`

`eval_log(`*x*, *cond=None*`)`
Return logarithm of (conditional) likelihood function in point x.

Parameters **x** (`numpy.ndarray`) – point which to evaluate the function in

Return type double

`sample(`*cond=None*`)`
Return one random (conditional) sample from this distribution

Return type `numpy.ndarray`

`samples(`*n*, *cond=None*`)`
Return n samples in an array. A convenience function that just calls `shape()` (page 8) multiple times.

Parameters **n** (*int*) – number of samples to return

Return type 2D `numpy.ndarray` of shape (*n*, m) where m is pdf dimension

class `pybayes.pdfs.Pdf`
Base class for all unconditional (static) multivariate Probability Density Functions. Subclass of `CPdf` (page 7).

As in CPdf, constructor of every Pdf takes optional **rv** (`RV` (page 5)) keyword argument (and no *cond_rv* argument as it would make no sense). For discussion about associated random variables see `CPdf` (page 7).

`cond_shape()`
Return zero as Pdfs have no condition.

2.3 Unconditional Probability Density Functions (pdfs)

class `pybayes.pdfs.UniPdf(`*a*, *b*, *rv=None*`)`
Simple uniform multivariate probability density function. Extends `Pdf` (page 9).

$$f(x) = \Theta(x-a)\Theta(b-x)\prod_{i=1}^{n}\frac{1}{b_i - a_i}$$

Variables

- **a** – left border
- **b** – right border

You may modify these attributes as long as you don't change their shape and assumption $\mathbf{a} < \mathbf{b}$ still holds.

`__init__(`*a*, *b*, *rv=None*`)`
Initialise uniform distribution.

Parameters

- **a** (`numpy.ndarray`) – left border
- **b** (`numpy.ndarray`) – right border

b must be greater (in each dimension) than **a**

class pybayes.pdfs.AbstractGaussPdf
Abstract base for all Gaussian-like pdfs - the ones that take vector mean and matrix covariance parameters. Extends Pdf (page 9).

Variables

- **mu** – mean value
- **R** – covariance matrix

You can modify object parameters only if you are absolutely sure that you pass allowable values - parameters are only checked once in constructor.

class pybayes.pdfs.GaussPdf(*mean*, *cov*, *rv=None*)
Unconditional Gaussian (normal) probability density function. Extends AbstractGaussPdf (page 10).

$$f(x) \propto \exp\left(-(x-\mu)' R^{-1} (x-\mu)\right)$$

__init__(*mean*, *cov*, *rv=None*)
Initialise Gaussian pdf.

Parameters

- **mean** (1D numpy.ndarray) – mean value
- **cov** (2D numpy.ndarray) – covariance matrix

To create standard normal distribution:

```
>>> # note that cov is a matrix because of the double [[ and ]]
>>> norm = GaussPdf(np.array([0.]), np.array([[1.]]))
```

class pybayes.pdfs.LogNormPdf(*mean*, *cov*, *rv=None*)
Unconditional log-normal probability density function. Extends AbstractGaussPdf (page 10).

More precisely, the density of random variable Y where $Y = exp(X);\ X \sim \mathcal{N}(\mu, R)$

__init__(*mean*, *cov*, *rv=None*)
Initialise log-normal pdf.

Parameters

- **mean** (1D numpy.ndarray) – mean value of the **logarithm** of the associated random variable
- **cov** (2D numpy.ndarray) – covariance matrix of the **logarithm** of the associated random variable

A current limitation is that LogNormPdf is only univariate. To create standard log-normal distribution:

```
>>> lognorm = LogNormPdf(np.array([0.]), np.array([[1.]]))  # note the shape of cov
```

class `pybayes.pdfs.AbstractEmpPdf`

An abstraction of empirical probability density functions that provides common methods such as weight normalisation. Extends `Pdf` (page 9).

Variables weights (*numpy.ndarray*) – 1D array of particle weights $\omega_i >= 0 \forall i; \quad \sum \omega_i = 1$

`normalise_weights()`

Multiply weights by appropriate constant so that $\sum \omega_i = 1$

Raises AttributeError when $\exists i : \omega_i < 0$ or $\forall i : \omega_i = 0$

`get_resample_indices()`

Calculate first step of resampling process (dropping low-weight particles and replacing them with more weighted ones.

Returns integer array of length n: $(a_1, a_2 \dots a_n)$ where a_i means that particle at ith place should be replaced with particle number a_i

Return type `numpy.ndarray` of ints

This method doesnt modify underlying pdf in any way - it merely calculates how particles should be replaced.

class `pybayes.pdfs.EmpPdf(`*init_particles*, *rv=None*`)`

Weighted empirical probability density function. Extends `AbstractEmpPdf` (page 11).

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

where $x^{(i)}$ is value of the ith particle

$\omega_i \geq 0$ is weight of the ith particle $\quad \sum \omega_i = 1$

Variables particles (*numpy.ndarray*) – 2D array of particles; shape: (n, m) where n is the number of particles, m dimension of this pdf

You may alter particles and weights, but you must ensure that their shapes match and that weight constraints still hold. You can use `normalise_weights()` (page 11) to do some work for you.

`__init__(`*init_particles*, *rv=None*`)`

Initialise empirical pdf.

Parameters init_particles (`numpy.ndarray`) – 2D array of initial particles; shape (n, m) determines that n m-dimensioned particles will be used. *Warning: EmpPdf does not copy the particles - it rather uses passed array through its lifetime, so it is not safe to reuse it for other purposes.*

`resample()`

Drop low-weight particles, replace them with copies of more weighted ones. Also reset weights to uniform.

class pybayes.pdfs.MarginalizedEmpPdf(*init_gausses*, *init_particles*, *rv=None*)

An extension to empirical pdf (EmpPdf (page 11)) used as aposteriori density by MarginalizedParticleFilter (page 18). Extends AbstractEmpPdf (page 11).

Assume that random variable x can be divided into 2 independent parts $x = [a, b]$, then probability density function can be written as

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

where $\hat{a}^{(i)}$ and $P^{(i)}$ is mean and covariance of ith gauss pdf

$b^{(i)}$ is value of the (second part of the) ith particle

$\omega_i \geq 0$ is weight of the ith particle $\sum \omega_i = 1$

Variables

- **gausses** (*numpy.ndarray*) – 1D array that holds GaussPdf (page 10) for each particle; shape: (n) where n is the number of particles
- **particles** (*numpy.ndarray*) – 2D array of particles; shape: (n, m) where n is the number of particles, m dimension of the "empirical" part of random variable
- **weights** (*numpy.ndarray*) – 1D array of particle weights

You may alter particles and weights, but you must ensure that their shapes match and that weight constraints still hold. You can use normalise_weights() (page 11) to do some work for you.

Note: this pdf could have been coded as ProdPdf of EmpPdf and a mixture of GaussPdfs. However it is implemented explicitly for simplicity and speed reasons.

__init__(*init_gausses*, *init_particles*, *rv=None*)

Initialise marginalized empirical pdf.

Parameters

- **init_gausses** (numpy.ndarray) – 1D array of GaussPdf (page 10) objects, all must have the dimension
- **init_particles** (numpy.ndarray) – 2D array of initial particles; shape (n, m) determines that n particles whose *empirical* part will have dimension m

Warning: MarginalizedEmpPdf does not copy the particles - it rather uses both passed arrays through its lifetime, so it is not safe to reuse them for other purposes.

class pybayes.pdfs.ProdPdf(*factors*, *rv=None*)

Unconditional product of multiple unconditional pdfs.

You can for example create a pdf that has uniform distribution with regards to x-axis and normal distribution along y-axis. The caller (you) must ensure that

individial random variables are independent, otherwise their product may have no mathematical sense. Extends Pdf (page 9).

$$f(x_1x_2x_3) = f_1(x_1)f_2(x_2)f_3(x_3)$$

__init__(*factors*, *rv=None*)

Initialise product of unconditional pdfs.

Parameters **factors** (sequence of Pdf (page 9)) – sequence of sub-distributions

As an exception from the general rule, ProdPdf does not create anonymous associated random variable if you do not supply it in constructor - it rather reuses components of underlying factor pdfs. (You can of course override this behaviour by bassing custom **rv**.)

Usual way of creating ProdPdf could be:

```
>>> prod = ProdPdf((UniPdf(...), GaussPdf(...)))  # note the double (( and ))
```

2.4 Conditional Probability Density Functions (cpdfs)

In this section, variable c in math exressions denotes condition.

class pybayes.pdfs.MLinGaussCPdf(*cov*, *A*, *b*, *rv=None*, *cond_rv=None*, *base_class=None*)

Conditional Gaussian pdf whose mean is a linear function of condition. Extends CPdf (page 7).

$$f(x|c) \propto \exp\left(-(x-\mu)' R^{-1} (x-\mu)\right) \qquad \text{where } \mu := Ac + b$$

__init__(*cov*, *A*, *b*, *rv=None*, *cond_rv=None*, *base_class=None*)

Initialise Mean-Linear Gaussian conditional pdf.

Parameters

- **cov** (2D numpy.ndarray) – covariance of underlying Gaussian pdf
- **A** (2D numpy.ndarray) – given condition c, $\mu = Ac + b$
- **b** (1D numpy.ndarray) – see above
- **base_class** (*class*) – class whose instance is created as a base pdf for this cpdf. Must be a subclass of AbstractGaussPdf (page 10) and the default is GaussPdf (page 10). One alternative is LogNormPdf (page 10) for example.

class pybayes.pdfs.LinGaussCPdf(*a*, *b*, *c*, *d*, *rv=None*, *cond_rv=None*, *base_class=None*)

Conditional one-dimensional Gaussian pdf whose mean and covariance are linear functions of condition. Extends CPdf (page 7).

$$f(x|c_1c_2) \propto \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right) \qquad \text{where} \quad \mu := ac_1 + b \quad \text{and} \quad \sigma^2 := cc_2 + d$$

__init__(*a*, *b*, *c*, *d*, *rv=None*, *cond_rv=None*, *base_class=None*)

Initialise Linear Gaussian conditional pdf.

Parameters

- **a, b** (*double*) – mean = a*cond_1 + b
- **c, d** (*double*) – covariance = c*cond_2 + d
- **base_class** (*class*) – class whose instance is created as a base pdf for this cpdf. Must be a subclass of AbstractGaussPdf (page 10) and the default is GaussPdf (page 10). One alternative is LogNormPdf (page 10) for example.

class pybayes.pdfs.GaussCPdf(*shape*, *cond_shape*, *f*, *g*, *rv=None*, *cond_rv=None*, *base_class=None*)

The most general normal conditional pdf. Use it only if you cannot use MLinGaussCPdf (page 13) or LinGaussCPdf (page 13) as this cpdf is least optimised. Extends CPdf (page 7).

$$f(x|c) \propto \exp\left(-(x-\mu)' R^{-1} (x-\mu)\right)$$
$$\text{where} \quad \mu := f(c) \text{ (interpreted n-dimensional vector)}$$
$$R := g(c) \text{ (interpreted as n*n matrix)}$$

__init__(*shape*, *cond_shape*, *f*, *g*, *rv=None*, *cond_rv=None*, *base_class=None*)

Initialise general gauss cpdf.

Parameters

- **shape** (*int*) – dimension of random vector
- **cond_shape** (*int*) – dimension of condition
- **f** (*callable*) – $\mu := f(c)$ where c = condition
- **g** (*callable*) – $R := g(c)$ where c = condition
- **base_class** (*class*) – class whose instance is created as a base pdf for this cpdf. Must be a subclass of AbstractGaussPdf (page 10) and the default is GaussPdf (page 10). One alternative is LogNormPdf (page 10) for example.

TODO: better specification of callback functions

class pybayes.pdfs.ProdCPdf(*factors*, *rv=None*, *cond_rv=None*)

Pdf that is formed as a chain rule of multiple conditional pdfs. Extends CPdf (page 7).

TODO: make aggreate [cond_]rv construction automatic and drop old constuctor.

In a simple textbook case denoted below it isn't needed to specify random variables at all. In this case when no random variable associations are passed, ProdCPdf ignores rv associations of its factors and everything is determined from their order. (x_i are arbitrary vectors)

$$f(x_1x_2x_3|c) = f_1(x_1|x_2x_3c)f_2(x_2|x_3c)f_3(x_3|c)$$
$$\text{or} \quad f(x_1x_2x_3) = f_1(x_1|x_2x_3)f_2(x_2|x_3)f_3(x_3)$$

```
>>> f = ProdCPdf((f1, f2, f3))
```

For less simple situations, specifiying random value associations is needed to estabilish data chain:

$$p(x_1x_2|y_1y_2) = p_1(x_1|x_2)p_2(x_2|y_2y_1)$$

```
>>> # prepare random variable components:
>>> x_1, x_2 = RVComp(1), RVComp(1, "name is optional")
>>> y_1, y_2 = RVComp(1), RVComp(1, "but recommended")

>>> p_1 = SomePdf(..., rv=RV(x_1), cond_rv=RV(x_2))
>>> p_2 = SomePdf(..., rv=RV(x_2), cond_rv=RV(y_2, y_1))
>>> p = ProdCPdf((p_2, p_1), rv=RV(x_1, x_2), cond_rv=RV(y_1, y_2))  # order of
>>> # pdfs is insignificant - order of rv components determines data flow
```

__init__(*factors*, *rv=None*, *cond_rv=None*)

Construct chain rule of multiple cpdfs.

Parameters **factors** (sequence of CPdf (page 7)) – sequence of densities that will form the product

Usual way of creating ProdCPdf could be:

```
>>> prod = ProdCPdf((MLinGaussCPdf(..), UniPdf(..)), RV(..), RV(..))
```

Chapter 3

Bayesian Filters

This module contains Bayesian filters.

All classes from this module are currently imported to top-level pybayes module, so instead of `from pybayes.filters import KalmanFilter` you can type `from pybayes import KalmanFilter`.

3.1 Filter prototype

class `pybayes.filters.Filter`

Abstract prototype of a bayesian filter.

`bayes(`*yt*, *ut=None*`)`

Perform approximate or exact bayes rule.

Parameters

- **yt** (`numpy.ndarray`) – observation at time t
- **ut** (`numpy.ndarray`) – intervence at time t (appliciable only to some filters)

Returns always returns True (see `posterior()` (page 16) to get aposteriori density)

`posterior()`

Return aposteriori probability density funcion (`Pdf` (page 9)).

Returns aposteriori density

Return type `Pdf` (page 9)

Filter implementations may decide to return a reference to their work pdf - it is not safe to modify it in any way, doing so may leave the filter in undefined state.

`evidence_log(`*yt*`)`

Return the logarithm of *evidence* function (also known as *marginal likehood*) evaluated in point yt.

Parameters **yt** (numpy.ndarray) – point which to evaluate the evidence in

Return type double

This is typically computed after bayes() (page 16) with the same observation:

```
>>> filter.bayes(yt)
>>> log_likehood = filter.evidence_log(yt)
```

3.2 Kalman Filter

class pybayes.filters.KalmanFilter(*A*, *B*, *C*, *D*, *Q*, *R*, *state_pdf*)
Kalman filter

__init__(*A*, *B*, *C*, *D*, *Q*, *R*, *state_pdf*)
TODO: documentation

3.3 Particle Filter Family

class pybayes.filters.ParticleFilter(*n*, *init_pdf*, *p_xt_xtp*, *p_yt_xt*)
A filter whose aposteriori density takes the form

$$p(x_t|y_{1:t}) = \sum_{i=1}^{n} \omega_i \delta(x_t - x_t^{(i)})$$

__init__(*n*, *init_pdf*, *p_xt_xtp*, *p_yt_xt*)
Initialise particle filter.

Parameters

- **n** (*int*) – number of particles
- **init_pdf** (Pdf (page 9)) – probability density which initial particles are sampled from
- **p_xt_xtp** (CPdf (page 7)) – $p(x_t|x_{t-1})$ cpdf of state in t given state in *t-1*
- **p_yt_xt** (CPdf (page 7)) – $p(y_t|x_t)$ cpdf of observation in t given state in t

bayes(*yt*, *ut=None*)
Perform Bayes rule for new measurement y_t. The algorithm is as follows:

1.generate new particles: $x_t^{(i)}$ = sample from $p(x_t^{(i)}|x_{t-1}^{(i)}) \quad \forall i$

2.recompute weights: $\omega_i = p(y_t|x_t^{(i)})\omega_i \quad \forall i$

3.normalise weights

4.resample particles

class `pybayes.filters.MarginalizedParticleFilter`(*n*, *init_pdf*, *p_bt_btp*)

Standard marginalized particle filter implementation. Assume that state variable x can be divided into two (TODO: independent?) parts: $x_t = [a_t, b_t]$, then aposteriori pdf can be denoted as:

TODO: better description.

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

$$p^{(i)}(a_t|b_{1:t}, y_{1:t}) = \mathcal{N}(\hat{a}_t^{(i)}, P_t^{(i)})$$

where $\hat{a}_t^{(i)}$ and $P_t^{(i)}$ is mean and covariance of ith gauss pdf

$b_t^{(i)}$ is value of the (b part of the) ith particle

$\omega_i \geq 0$ is weight of the ith particle $\sum \omega_i = 1$

`__init__`(*n*, *init_pdf*, *p_bt_btp*)

Initialise marginalized particle filter.

Parameters

- **n** (*int*) – number of particles
- **init_pdf** (`Pdf` (page 9)) – probability density which initial particles are sampled from. (both a_t and b_t parts)
- **p_bt_btp** (`CPdf` (page 7)) – $p(b_t|b_{t-1})$ cpdf of the (b part of the) state in t given state in *t-1*

`bayes`(*yt*, *ut=None*)

Perform Bayes rule for new measurement y_t. Uses following algorithm:

1.generate new b parts of particles: $b_t^{(i)} =$ sample from $p(b_t^{(i)}|b_{t-1}^{(i)}) \quad \forall i$

2.set $Q_i := b_t^{(i)} \quad R_i := b_t^{(i)}$ where Q_i, R_i is covariance of process (respectively observation) noise in ith Kalman filter.

3.perform Bayes rule for each Kalman filter using passed observation y_t

4.recompute weights: $\omega_i = p(y_t|y_{1:t}, b_t^{(i)})\omega_i$ where $p(y_t|y_{1:t}, b_t^{(i)})$ is *evidence* (*marginal likehood*) pdf of ith Kalman filter.

5.normalise weights

6.resample particles

Chapter 4

PyBayes Development

This document should serve as a reminder to me and other possible PyBayes hackers about PyBayes coding style and conventions.

4.1 General Layout and Principles

PyBayes is developed with special dual-mode technique - it is both perfectly valid pure Python library and optimised cython-built binary python module.

PyBayes modules are laid out with following rules:

- all modules go directly into `pybayes/<module>.py` (pure Python file) with cython augmentation file in `pybayes/module.pxd`
- in future, bigger independent units can form subpackages
- `pybayes/numpywrap.{pyx,py,pxd}` are special, it is the only module that has different implementation for cython and for python.

4.2 Tests and Stress Tests

All methods of all PyBayes classes should have a unit test. Suppose you have a module `pybayes/modname.py`, then unit tests for all classes in `modname.py` should go into `pybayes/tests/test_modname.py`. You can also write stress test (something that runs considerably longer than a test and perhaps provides a simple benchmark) that would go into `pybayes/tests/stress_modname.py`.

4.3 Imports and cimports

No internal module can `import pybayes`! That would result in an infinite recursion. External PyBayes clients can and should, however, only `import pybayes` (and in future also `import pybayes.subpackage`). From insibe PyBayes just import relevant pybayes

modules, e.g. `import pdfs`. Notable exception from this rule is cimport, where (presumable due to a cython bug) `from a.b cimport c` sometimes doesn't work and one has to type `from pybayes.a.b cimport c`.

Imports in *.py files should adhere to following rules:

- import first system modules (sys, io..), then external modules (matplotlib..) and then pybayes modules.
- **instead of** importing **numpy** directly use `import wrappers._numpy as np`. This ensures that fast C alternatives are used in compiled mode.
- **instead of** importing **numpy.linalg** directly use `import wrappers._linalg as linalg`.
- use `import module [as abbrev]` or, for commonly used symbols `from module import symbol`.
- `from module import *` shouldn't be used.

Following rules apply to *.pxd (cython augmentation) files:

- no imports, just cimports.
- use same import styles as in associated .py file. (`from module cimport` vs. `cimport module [as abbrev]`)
- for numpy use `cimport pybayes.wrappers._numpy as np`
- for numpy.linalg use `cimport pybayes.wrappers._linalg as linalg`

Above rules do not apply to `pybayes/tests`. *These modules are considered external and should behave as a client script.*

Index

Symbols

A

B

C

E

F

G

I

K

L

M

N

P

R

S

U

V