PyBayes API Documentation

Release 0.2-172-g7180254-dirty

Matěj Laitl

July 24, 2011

Contents

Chapter 1

PyBayes

1.1 About

PyBayes is an object-oriented Python library for recursive Bayesian estimation (Bayesian filtering) that is convenient to use. Already implemented are Kalman filter, particle filter and marginalized particle filter, all built atop of a light framework of probability density functions. PyBayes can optionally use Cython for lage speed gains (Cython build is several times faster).

Future plans include more specialised variants of Kalman/particle filters and speed optimisations.

PyBayes is being developed by Matěj Laitl, feel free to send me a mail to matej at laitl dot cz.

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When updating your repository (using `git pull`), git should inform you that some submodules have became outdated.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])

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

- NumPy numerical library for Python, version 1.5 or greater (NumPy is needed also in Python build, but older version suffice in that case)
- 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 **a** < **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).

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

`__init__(`*mean*, *cov*, *rv=None*`)`

Initialise Gaussian pdf.

Parameters

- **mean** (1D `numpy.ndarray`) – mean value; stored in **mu** attribute
- **cov** (2D `numpy.ndarray`) – covariance matrix; stored in **R** artibute

Covariance matrix **cov** must be *positive definite*. This is not checked during initialisation; it fail or give incorrect results in `eval_log()` (page 8) or `sample()` (page 9). 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).

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

$$\text{where} \quad x^{(i)} \text{ is value of the i}^{th} \text{ particle}$$

$$\omega_i \geq 0 \text{ is weight of the i}^{th} \text{ particle} \quad \sum \omega_i = 1$$

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 posterior density by `MarginalizedParticleFilter` (page 19). 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

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

$$\begin{aligned} \text{where} \quad \hat{a}^{(i)} &\text{ and } P^{(i)} \text{ is mean and covariance of i}^{th} \text{ gauss pdf} \\ b^{(i)} &\text{ is value of the (second part of the) i}^{th} \text{ particle} \\ \omega_i \geq 0 &\text{ is weight of the i}^{th} \text{ particle} \quad \sum \omega_i = 1 \end{aligned}$$

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

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).

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

`__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).

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

`__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).

$$f(x|c_1 c_2) \propto \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right) \qquad \text{where} \quad \mu = a c_1 + b \quad \text{and} \quad \sigma^2 = c c_2 + d$$

`__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 14) as this cpdf is least optimised. Extends `CPdf` (page 7).

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

`__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.

Please note that the way of specifying callback function f and g is not yet fixed and may be changed in future.

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).

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)

$$p(x_1 x_2 x_3 | c) = p(x_1 | x_2 x_3 c) p(x_2 | x_3 c) p(x_3 | c)$$
$$\text{or} \quad p(x_1 x_2 x_3) = p(x_1 | x_2 x_3) p(x_2 | x_3) p(x_3)$$

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

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

$$p(x_1 x_2 | y_1 y_2) = p_1(x_1 | y_1) p_2(x_2 | y_2 y_1)$$

```
>>> # 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
```

Please note: this will change in near future in following way: it will be always required to specify rvs and cond_rvs of factor pdfs (at least ones that are shared), but product rv and cond_rv will be inferred automatically when not specified.

`__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*, *cond=None*)

Perform approximate or exact bayes rule.

Parameters

- **yt** (1D `numpy.ndarray`) – observation at time t
- **cond** (1D `numpy.ndarray`) – condition at time t. Exact meaning is defined by each filter

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

`posterior()`

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

Returns posterior 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 likelihood*) 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_likelihood = filter.evidence_log(yt)
```

3.2 Kalman Filter

class pybayes.filters.KalmanFilter(*A*, *B=None*, *C=None*, *D=None*, *Q=None*, *R=None*, *state_pdf=None*)

Implementation of standard Kalman filter. **cond** in bayes() (page 18) is interpreted as control (intervention) input u_t to the system.

Kalman filter forms *optimal Bayesian solution* for the following system:

$$\begin{aligned} x_t &= A_t x_{t-1} + B_t u_t + v_{t-1} & A_t &\in \mathbb{R}^{n,n} \quad B_t \in \mathbb{R}^{n,k} \quad n \in \mathbb{N} \quad k \in \mathbb{N}_0 \text{ (may be zero)} \\ y_t &= C_t x_t + D_t u_t + w_t & C_t &\in \mathbb{R}^{j,n} \quad D_t \in \mathbb{R}^{j,k} \quad j \in \mathbb{N} \quad j \le n \end{aligned}$$

where $x_t \in \mathbb{R}^n$ is hidden state vector, $y_t \in \mathbb{R}^j$ is observation vector and $u_t \in \mathbb{R}^k$ is control vector. v_t is normally distributed zero-mean process noise with covariance matrix Q_t, w_t is normally distributed zero-mean observation noise with covariance matrix R_t. Additionally, intial pdf (**state_pdf**) has to be Gaussian.

__init__(*A*, *B=None*, *C=None*, *D=None*, *Q=None*, *R=None*, *state_pdf=None*)

Initialise Kalman filter.

Parameters

- **A** (2D numpy.ndarray) – process model matrix A_t from class description (page 17)
- **B** (2D numpy.ndarray) – process control model matrix B_t from class description (page 17); may be None or unspecified for control-less systems
- **C** (2D numpy.ndarray) – observation model matrix C_t from class description (page 17); must be full-ranked
- **D** (2D numpy.ndarray) – observation control model matrix D_t from class description (page 17); may be None or unspecified for control-less systems
- **Q** (2D numpy.ndarray) – process noise covariance matrix Q_t from class description (page 17); must be positive definite
- **R** (2D numpy.ndarray) – observation noise covariance matrix R_t from class description (page 17); must be positive definite

- **state_pdf** (GaussPdf (page 10)) – initial state pdf; this object is referenced and used throughout whole life of KalmanFilter, so it is not safe to reuse state pdf for other purposes

All matrices can be time-varying - you can modify or replace all above stated matrices providing that you don't change their shape and all constraints still hold. On the other hand, you **should not modify state_pdf** unless you really know what you are doing.

```
>>> # initialise control-less Kalman filter:
>>> kf = pb.KalmanFilter(A=np.array([[1.]]),
                         C=np.array([[1.]]),
                         Q=np.array([[0.7]]), R=np.array([[0.3]]),
                         state_pdf=pb.GaussPdf(...))
```

`bayes`(*yt*, *cond=array([], dtype=float64)*)
Perform exact bayes rule.

Parameters

- **yt** (1D `numpy.ndarray`) – observation at time t
- **cond** (1D `numpy.ndarray`) – control (intervention) vector at time t. May be unspecified if the filter is control-less.

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

3.3 Particle Filter Family

class `pybayes.filters.ParticleFilter`(*n*, *init_pdf*, *p_xt_xtp*, *p_yt_xt*)
Standard particle filter implementation with resampling.

Specifying proposal density is currently unsupported, but planned; speak up if you want it! Posterior pdf is represented using EmpPdf (page 11) and takes following 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*, *cond=None*)

Perform Bayes rule for new measurement y_t. The algorithm is as follows:

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

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*, *kalman_args*, *kalman_class=<class 'pybayes.filters.KalmanFilter'>*)

Simple marginalized particle filter implementation. Assume that tha state vector x can be divided into two parts $x_t = (a_t, b_t)$ and that the pdf representing the process model can be factorised as follows:

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

and that the a_t part (given b_t) can be estimated with (a subbclass of) the `KalmanFilter` (page 17). Such system may be suitable for the marginalized particle filter, whose posterior pdf takes the form

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

$$\text{where} \quad \begin{array}{c} p(a_t|y_{1:t}, b_{1:t}^{(i)}) \text{ is posterior pdf of i}^{th} \text{ Kalman filter} \\ b_t^{(i)} \text{ is value of the (b part of the) i}^{th} \text{ particle} \\ \omega_i \geq 0 \text{ is weight of the i}^{th} \text{ particle} \quad \sum \omega_i = 1 \end{array}$$

Note: currently :math:'b_t' is hard-coded to be process and observation noise covariance of the :math:'a_t' part. This will be changed soon and :math:'b_t' will be passed as condition to :meth:'KalmanFilter.bayes'.

`__init__`(*n*, *init_pdf*, *p_bt_btp*, *kalman_args*, *kalman_class=<class 'pybayes.filters.KalmanFilter'>*)

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*
- **kalman_args** (*dict*) – arguments for the Kalman filter, passed as dictionary; *state_pdf* key should not be speficied as it is supplied by the marginalized particle filter
- **kalman_class** (*class*) – class of the filter used for the a_t part of the system; defaults to `KalmanFilter` (page 17)

`bayes`(*yt*, *cond=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: $\omega_i = p(y_t|y_{1:t-1}, b_t^{(i)})\omega_i$ where $p(y_t|y_{1:t-1}, b_t^{(i)})$ is *evidence* (*marginal likelihood*) 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/wapperts/` subpackage is special, it is the only package whose modules have different implementation for cython and for python. It is accomplished by .py (Python) and .pyx, .pxd (Cython) files.

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