BayesPy Documentation

Release 0.2.3

Jaakko Luttinen

February 20, 2015

CONTENTS

CHAPTER

ONE

INTRODUCTION

BayesPy provides tools for Bayesian inference with Python. The user constructs a model as a Bayesian network, observes data and runs posterior inference. The goal is to provide a tool which is efficient, flexible and extendable enough for expert use but also accessible for more casual users.

Currently, only variational Bayesian inference for conjugate-exponential family (variational message passing) has been implemented. Future work includes variational approximations for other types of distributions and possibly other approximate inference methods such as expectation propagation, Laplace approximations, Markov chain Monte Carlo (MCMC) and other methods. Contributions are welcome.

1.1 Project information

- Documentation:
  - http://bayespy.org
  - PDF file
  - RST format in `doc` directory
- Repository: https://github.com/bayespy/bayespy.git
- Bug reports: https://github.com/bayespy/bayespy/issues
- Mailing list: bayespy@googlegroups.com
- IRC: #bayespy @ freenode
- Author: Jaakko Luttinen jaakko.luttinen@iki.fi
- Latest release:
- Build status:
- Unit test coverage:

1.2 Similar projects

VIBES (http://vibes.sourceforge.net/) allows variational inference to be performed automatically on a Bayesian network. It is implemented in Java and released under revised BSD license.

Bayes Blocks (http://research.ics.aalto.fi/bayes/software/) is a C++/Python implementation of the variational building block framework. The framework allows easy learning of a wide variety of models using variational Bayesian learning. It is available as free software under the GNU General Public License.

Infer.NET (http://research.microsoft.com/infernet/) is a .NET framework for machine learning. It provides message-passing algorithms and statistical routines for performing Bayesian inference. It is partly closed source and licensed for non-commercial use only.

PyMC (https://github.com/pymc-devs/pymc) provides MCMC methods in Python. It is released under the Academic Free License.

OpenBUGS (http://www.openbugs.info) is a software package for performing Bayesian inference using Gibbs sampling. It is released under the GNU General Public License.

Dimple (http://dimple.probprog.org/) provides Gibbs sampling, belief propagation and a few other inference algorithms for Matlab and Java. It is released under the Apache License.

Stan (http://mc-stan.org/) provides inference using MCMC with an interface for R and Python. It is released under the New BSD License.

PBNT - Python Bayesian Network Toolbox (http://pbnt.berlios.de/) is Bayesian network library in Python supporting static networks with discrete variables. There was no information about the license.

1.3 Contributors

The list of contributors:

- Jaakko Luttinen
- Hannu Hartikainen

Each file or the git log can be used for more detailed information.

1.4 Version history

1.4.1 Version 0.2.3 (2014-12-03)

- Fix matplotlib compatibility broken by recent changes in matplotlib
- Add random sampling for Binomial and Bernoulli nodes
- Fix minor bugs, for instance, in plot module

1.4.2 Version 0.2.2 (2014-11-01)

- Fix normalization of categorical Markov chain probabilities (fixes HMM demo)
- Fix initialization from parameter values

1.4.3 Version 0.2.1 (2014-09-30)

- Add workaround for matplotlib 1.4.0 bug related to interactive mode which affected monitoring
- Fix bugs in Hinton diagrams for Gaussian variables

1.4.4 Version 0.2 (2014-08-06)

- Added all remaining common distributions: Bernoulli, binomial, multinomial, Poisson, beta, exponential.
- Added Gaussian arrays (not just scalars or vectors).
- Added Gaussian Markov chains with time-varying or swithing dynamics.
- Added discrete Markov chains (enabling hidden Markov models).
- Added joint Gaussian-Wishart and Gaussian-gamma nodes.
- Added deterministic gating node.
- Added deterministic general sum-product node.
- Added parameter expansion for Gaussian arrays and time-varying/switching Gaussian Markov chains.
- Added new plotting functions: pdf, Hinton diagram.
- Added monitoring of posterior distributions during iteration.
- Finished documentation and added API.

1.4.5 Version 0.1 (2013-07-25)

- Added variational message passing inference engine.
- Added the following common distributions: Gaussian vector, gamma, Wishart, Dirichlet, categorical.
- Added Gaussian Markov chain.
- Added parameter expansion for Gaussian vectors and Gaussian Markov chain.
- Added stochastic mixture node.
- Added deterministic dot product node.
- Created preliminary version of the documentation.

CHAPTER

TWO

USER GUIDE

2.1 Installation

BayesPy is a Python 3 package and it can be installed from PyPI or the latest development version from GitHub. The instructions below explain how to set up the system by installing required packages, how to install BayesPy and how to compile this documentation yourself. However, if these instructions contain errors or some relevant details are missing, please file a bug report at https://github.com/bayespy/bayespy/issues.

2.1.1 Installing BayesPy

BayesPy can be installed easily by using Pip if the system has been properly set up. If you have problems with the following methods, see the following section for some help on installing the requirements.

For users

First, you may want to set up a virtual environment. Using virtual environment is optional but recommended. To create and activate a new virtual environment, run (in the folder in which you want to create the environment):

```
virtualenv -p python3 --system-site-packages ENV
source ENV/bin/activate
```

The latest release of BayesPy can be installed from PyPI simply as

```
pip install bayespy
```

If you want to install the latest development version of BayesPy, use GitHub instead:

```
pip install https://github.com/bayespy/bayespy/archive/master.zip
```

For developers

If you want to install the development version of BayesPy in such a way that you can easily edit the package, follow these instructions. Get the git repository:

```
git clone https://github.com/bayespy/bayespy.git
cd bayespy
```

Create and activate a new virtual environment (optional but recommended):

```
virtualenv -p python3 --system-site-packages ENV
source ENV/bin/activate
```

Install BayesPy in editable mode:

```
pip install -e .
```

Checking installation

If you have problems installing BayesPy, read the next section for more details. It is recommended to run the unit tests in order to check that BayesPy is working properly. Thus, install Nose and run the unit tests:

```
pip install nose
nosetests bayespy
```

2.1.2 Installing requirements

BayesPy requires Python 3.2 (or later) and the following packages:

- NumPy (>=1.8.0),
- SciPy (>=0.13.0)
- matplotlib (>=1.2)
- h5py

Ideally, Pip should install the necessary requirements and a manual installation of these dependencies is not required. However, there are several reasons why the installation of these dependencies needs to be done manually in some cases. Thus, this section tries to give some details on how to set up your system. A proper installation of the dependencies for Python 3 can be a bit tricky and you may refer to http://www.scipy.org/install.html for more detailed instructions about the SciPy stack. Detailed instructions on installing recent SciPy stack for various platforms is out of the scope of these instructions, but we provide some general guidance here. There are basically three ways to install the dependencies:

1. Install a Python distribution which includes the packages. For Windows, Mac and Linux, there are several Python distributions which include all the necessary packages: http://www.scipy.org/install.html#scientific-python-distributions. For instance, you may try Anaconda or Enthought.
2. Install the packages using the system package manager. On Linux, the packages might be called something like `python-scipy` or `scipy`. However, it is possible that these system packages are not recent enough for BayesPy.
3. Install the packages using Pip: `pip install "numpy>=1.8.0" "scipy>=0.13.0" "matplotlib>=1.2" h5py`. However, this may require that the system has the libraries needed for compiling (e.g., C compiler, Python development files, BLAS/LAPACK). For instance, on Ubuntu (>= 12.10), you may install the required system libraries for each package as:

   ```
   sudo apt-get build-dep python3-numpy
   sudo apt-get build-dep python3-scipy
   sudo apt-get build-dep python3-matplotlib
   sudo apt-get build-dep python-h5py
   ```

   Then installation using Pip should work. Also, make sure you have recent enough version of Distribute (required by Matplotlib): `pip install "distribute>=0.6.28"`.

2.1.3 Compiling documentation

This documentation can be found at http://bayespy.org/ in HTML and PDF formats. The documentation source files are also readable as such in reStructuredText format in `doc/source/` directory. It is possible to compile the documentation into HTML or PDF yourself. In order to compile the documentation, Sphinx is required and a few extensions for it. Those can be installed as:

```
pip install "sphinx>=1.2.3" sphinxcontrib-tikz sphinxcontrib-bayesnet sphinxcontrib-bibtex "numpydoc>
```

In order to visualize graphical models in HTML, you need to have `ImageMagick` or `Netpbm` installed. The documentation can be compiled to HTML and PDF by running the following commands in the `doc` directory:

```
make html
make latexpdf
```

You can also run doctest to test code snippets in the documentation:

```
make doctest
```

or in the docstrings:

```
nosetests --with-doctest bayespy
```

2.2 Quick start guide

This short guide shows the key steps in using BayesPy for variational Bayesian inference by applying BayesPy to a simple problem. The key steps in using BayesPy are the following:

- Construct the model
- Observe some of the variables by providing the data in a proper format
- Run variational Bayesian inference
- Examine the resulting posterior approximation

To demonstrate BayesPy, we'll consider a very simple problem: we have a set of observations from a Gaussian distribution with unknown mean and variance, and we want to learn these parameters. In this case, we do not use any real-world data but generate some artificial data. The dataset consists of ten samples from a Gaussian distribution with mean 5 and standard deviation 10. This dataset can be generated with NumPy as follows:

```
>>> import numpy as np
>>> data = np.random.normal(5, 10, size=(10,))
```

2.2.1 Constructing the model

Now, given this data we would like to estimate the mean and the standard deviation as if we didn't know their values. The model can be defined as follows:

$$\begin{aligned} p(\mathbf{y}|\mu,\tau) &= \prod_{n=0}^{9} \mathcal{N}(y_n|\mu,\tau) \\ p(\mu) &= \mathcal{N}(\mu|0,10^{-6}) \\ p(\tau) &= \mathcal{G}(\tau|10^{-6},10^{-6}) \end{aligned}$$

where $\mathcal{N}$ is the Gaussian distribution parameterized by its mean and precision (i.e., inverse variance), and $\mathcal{G}$ is the gamma distribution parameterized by its shape and rate parameters. Note that we have given quite uninformative priors for the variables μ and τ. This simple model can also be shown as a directed factor graph: This model can be constructed in BayesPy as follows:

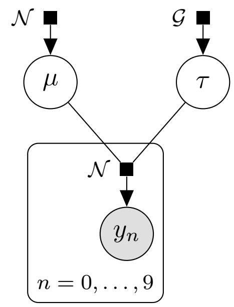


Figure 2.1: Directed factor graph of the example model.

```
>>> from bayespy.nodes import GaussianARD, Gamma
>>> mu = GaussianARD(0, 1e-6)
>>> tau = Gamma(1e-6, 1e-6)
>>> y = GaussianARD(mu, tau, plates=(10,))
```

This is quite self-explanatory given the model definitions above. We have used two types of nodes `GaussianARD` and `Gamma` to represent Gaussian and gamma distributions, respectively. There are much more distributions in `bayespy.nodes` so you can construct quite complex conjugate exponential family models. The node `y` uses keyword argument `plates` to define the plates $n = 0, \ldots, 9$.

2.2.2 Performing inference

Now that we have created the model, we can provide our data by setting `y` as observed:

```
>>> y.observe(data)
```

Next we want to estimate the posterior distribution. In principle, we could use different inference engines (e.g., MCMC or EP) but currently only variational Bayesian (VB) engine is implemented. The engine is initialized by giving all the nodes of the model:

```
>>> from bayespy.inference import VB
>>> Q = VB(mu, tau, y)
```

The inference algorithm can be run as long as wanted (max. 20 iterations in this case):

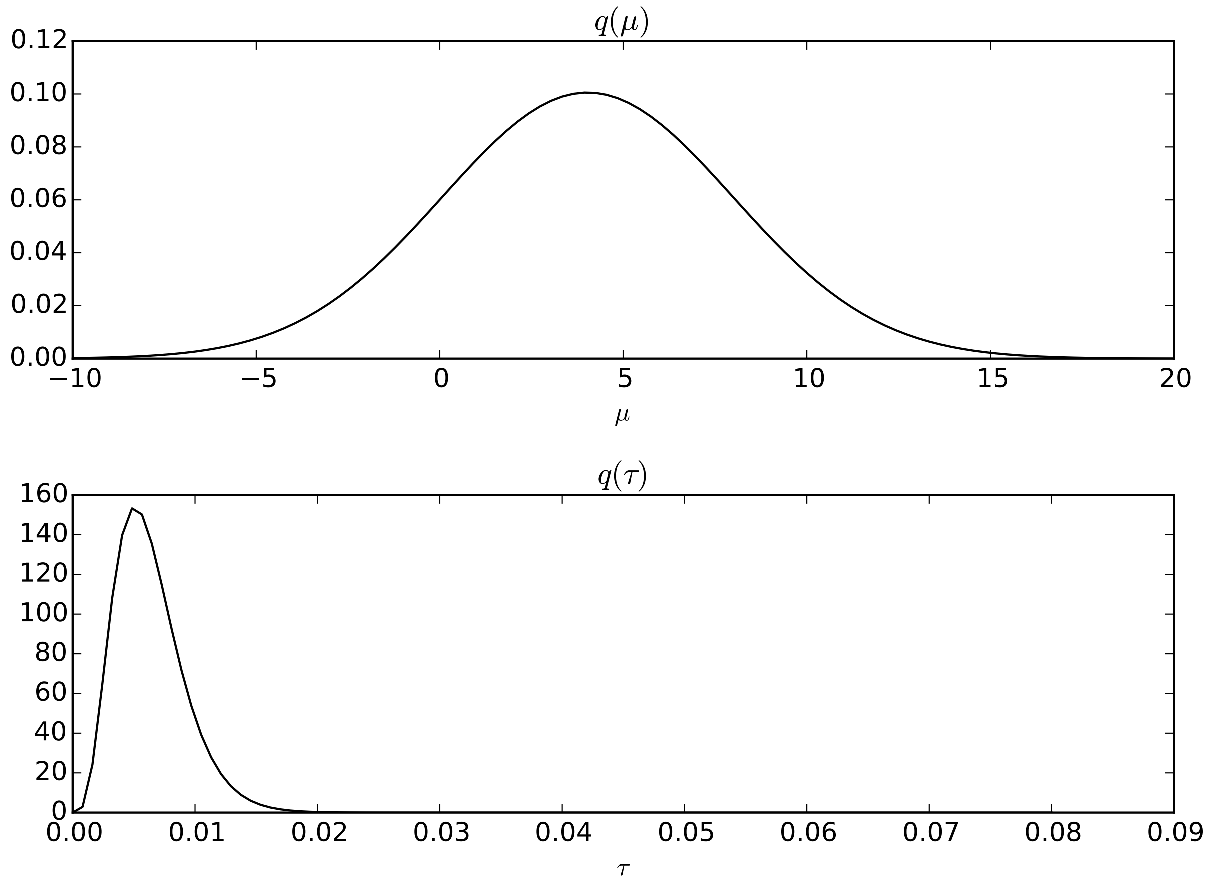
```
>>> Q.update(repeat=20)
Iteration 1: loglike=-6.020956e+01 (... seconds)
Iteration 2: loglike=-5.820527e+01 (... seconds)
Iteration 3: loglike=-5.820290e+01 (... seconds)
Iteration 4: loglike=-5.820288e+01 (... seconds)
Converged at iteration 4.
```

Now the algorithm converged after four iterations, before the requested 20 iterations. VB approximates the true posterior $p(\mu, \tau|\mathbf{y})$ with a distribution which factorizes with respect to the nodes: $q(\mu)q(\tau)$.

2.2.3 Examining posterior approximation

The resulting approximate posterior distributions $q(\mu)$ and $q(\tau)$ can be examined, for instance, by plotting the marginal probability density functions:

```
>>> import bayespy.plot as bpplt
>>> bpplt.pyplot.subplot(2, 1, 1)
<matplotlib.axes...AxesSubplot object at 0x...>
>>> bpplt.pdf(mu, np.linspace(-10, 20, num=100), color='k', name=r'\mu')
[<matplotlib.lines.Line2D object at 0x...>]
>>> bpplt.pyplot.subplot(2, 1, 2)
<matplotlib.axes...AxesSubplot object at 0x...>
>>> bpplt.pdf(tau, np.linspace(1e-6, 0.08, num=100), color='k', name=r'\tau')
[<matplotlib.lines.Line2D object at 0x...>]
>>> bpplt.pyplot.tight_layout()
>>> bpplt.pyplot.show()
```



This example was a very simple introduction to using BayesPy. The model can be much more complex and each phase contains more options to give the user more control over the inference. The following sections give more details about the phases.

2.3 Constructing the model

In BayesPy, the model is constructed by creating nodes which form a directed network. There are two types of nodes: stochastic and deterministic. A stochastic node corresponds to a random variable (or a set of random variables) from a specific probability distribution. A deterministic node corresponds to a deterministic function of its parents. For a list of built-in nodes, see the *User API*.

2.3.1 Creating nodes

Creating a node is basically like writing the conditional prior distribution of the variable in Python. The node is constructed by giving the parent nodes, that is, the conditioning variables as arguments. The number of parents and their meaning depend on the node. For instance, a `Gaussian` node is created by giving the mean vector and the precision matrix. These parents can be constant numerical arrays if they are known:

```
>>> from bayespy.nodes import Gaussian
>>> X = Gaussian([2, 5], [[1.0, 0.3], [0.3, 1.0]])
```

or other nodes if they are unknown and given prior distributions:

```
>>> from bayespy.nodes import Gaussian, Wishart
>>> mu = Gaussian([0, 0], [[1e-6, 0],[0, 1e-6]])
>>> Lambda = Wishart(2, [[1, 0], [0, 1]])
>>> X = Gaussian(mu, Lambda)
```

Nodes can also be named by providing `name` keyword argument:

```
>>> X = Gaussian(mu, Lambda, name='x')
```

The name may be useful when referring to the node using an inference engine.

For the parent nodes, there are two main restrictions: non-constant parent nodes must be conjugate and the parent nodes must be mutually independent in the posterior approximation.

Conjugacy of the parents

In Bayesian framework in general, one can give quite arbitrary probability distributions for variables. However, one often uses distributions that are easy to handle in practice. Quite often this means that the parents are given conjugate priors. This is also one of the limitations in BayesPy: only conjugate family prior distributions are accepted currently. Thus, although in principle one could give, for instance, gamma prior for the mean parameter `mu`, only Gaussian-family distributions are accepted because of the conjugacy. If the parent is not of a proper type, an error is raised. This conjugacy is checked automatically by BayesPy and `NoConverterError` is raised if a parent cannot be interpreted as being from a conjugate distribution.

Independence of the parents

Another a bit rarely encountered limitation is that the parents must be mutually independent (in the posterior factorization). Thus, a node cannot have the same stochastic node as several parents without intermediate stochastic nodes. For instance, the following leads to an error:

```
>>> from bayespy.nodes import Dot
>>> Y = Dot(X, X)
Traceback (most recent call last):
    ...
ValueError: Parent nodes are not independent
```

The error is raised because `X` is given as two parents for `Y`, and obviously `X` is not independent of `X` in the posterior approximation. Even if `X` is not given several times directly but there are some intermediate deterministic nodes, an error is raised because the deterministic nodes depend on their parents and thus the parents of `Y` would not be independent. However, it is valid that a node is a parent of another node via several paths if all the paths or all except one path has intermediate stochastic nodes. This is valid because the intermediate stochastic nodes have independent posterior approximations. Thus, for instance, the following construction does not raise errors:

```
>>> from bayespy.nodes import Dot
>>> Z = Gaussian(X, [[1,0], [0,1]])
>>> Y = Dot(X, Z)
```

This works because there is now an intermediate stochastic node `Z` on the other path from `X` node to `Y` node.

2.3.2 Effects of the nodes on inference

When constructing the network with nodes, the stochastic nodes actually define three important aspects:

1. the prior probability distribution for the variables,
2. the factorization of the posterior approximation,
3. the functional form of the posterior approximation for the variables.

Prior probability distribution

First, the most intuitive feature of the nodes is that they define the prior distribution. In the previous example, `mu` was a stochastic `GaussianARD` node corresponding to μ from the normal distribution, `tau` was a stochastic `Gamma` node corresponding to τ from the gamma distribution, and `y` was a stochastic `GaussianARD` node corresponding to y from the normal distribution with mean μ and precision τ. If we denote the set of all stochastic nodes by Ω, and by π_X the set of parents of a node X, the model is defined as

$$p(\Omega) = \prod_{X \in \Omega} p(X|\pi_X),$$

where nodes correspond to the terms $p(X|\pi_X)$.

Posterior factorization

Second, the nodes define the structure of the posterior approximation. The variational Bayesian approximation factorizes with respect to nodes, that is, each node corresponds to an independent probability distribution in the posterior approximation. In the previous example, `mu` and `tau` were separate nodes, thus the posterior approximation factorizes with respect to them: $q(\mu)q(\tau)$. Thus, the posterior approximation can be written as:

$$p(\tilde{\Omega}|\hat{\Omega}) \approx \prod_{X \in \tilde{\Omega}} q(X),$$

where $\tilde{\Omega}$ is the set of latent stochastic nodes and $\hat{\Omega}$ is the set of observed stochastic nodes. Sometimes one may want to avoid the factorization between some variables. For this purpose, there are some nodes which model several variables jointly without factorization. For instance, `GaussianGammaISO` is a joint node for μ and τ variables from the normal-gamma distribution and the posterior approximation does not factorize between μ and τ, that is, the posterior approximation is $q(\mu, \tau)$.

Functional form of the posterior

Last, the nodes define the functional form of the posterior approximation. Usually, the posterior approximation has the same or similar functional form as the prior. For instance, `Gamma` uses gamma distribution to also approximate the posterior distribution. Similarly, `GaussianARD` uses Gaussian distribution for the posterior. However, the posterior approximation of `GaussianARD` uses a full covariance matrix although the prior assumes a diagonal covariance matrix. Thus, there can be slight differences in the exact functional form of the posterior approximation but the rule of thumb is that the functional form of the posterior approximation is the same as or more general than the functional form of the prior.

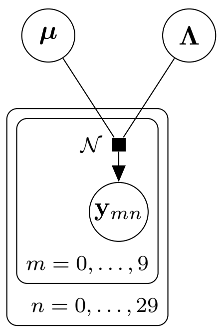
2.3.3 Using plate notation

Defining plates

Stochastic nodes take the optional parameter `plates`, which can be used to define plates of the variable. A plate defines the number of repetitions of a set of variables. For instance, a set of random variables $\mathbf{y}_{mn}$ could be defined as

$$\mathbf{y}_{mn} \sim \mathcal{N}(\boldsymbol{\mu}, \mathbf{\Lambda}), \qquad m = 0, \ldots, 9, \quad n = 0, \ldots, 29.$$

This can also be visualized as a graphical model:



The variable has two plates: one for the index m and one for the index n. In BayesPy, this random variable can be constructed as:

```
>>> y = Gaussian(mu, Lambda, plates=(10,30))
```

Note: The plates are always given as a tuple of positive integers.

Plates also define indexing for the nodes, thus you can use simple NumPy-style slice indexing to obtain a subset of the plates:

```
>>> y_0 = y[0]
>>> y_0.plates
(30,)
>>> y_even = y[:,::2]
>>> y_even.plates
(10, 15)
>>> y_complex = y[:5, 10:20:5]
>>> y_complex.plates
(5, 2)
```

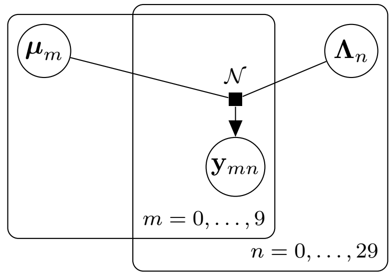
Note that this indexing is for the plates only, not for the random variable dimensions.

Sharing and broadcasting plates

Instead of having a common mean and precision matrix for all $\mathbf{y}_{mn}$, it is also possible to share plates with parents. For instance, the mean could be different for each index m and the precision for each index n:

$$\mathbf{y}_{mn} \sim \mathcal{N}(\boldsymbol{\mu}_m, \mathbf{\Lambda}_n), \qquad m = 0, \ldots, 9, \quad n = 0, \ldots, 29.$$

which has the following graphical representation:



This can be constructed in BayesPy, for instance, as:

```
>>> from bayespy.nodes import Gaussian, Wishart
>>> mu = Gaussian([0, 0], [[1e-6, 0],[0, 1e-6]], plates=(10,1))
>>> Lambda = Wishart(2, [[1, 0], [0, 1]], plates=(1,30))
>>> X = Gaussian(mu, Lambda)
```

There are a few things to notice here. First, the plates are defined similarly as shapes in NumPy, that is, they use similar broadcasting rules. For instance, the plates `(10,1)` and `(1,30)` broadcast to `(10,30)`. In fact, one could use plates `(10,1)` and `(30,)` to get the broadcasted plates `(10,30)` because broadcasting compares the plates from right to left starting from the last axis. Second, `X` is not given `plates` keyword argument because the default plates are the plates broadcasted from the parents and that was what we wanted so it was not necessary to provide the keyword argument. If we wanted, for instance, plates `(20,10,30)` for `X`, then we would have needed to provide `plates=(20,10,30)`.

The validity of the plates between a child and its parents is checked as follows. The plates are compared plate-wise starting from the last axis and working the way forward. A plate of the child is compatible with a plate of the parent if either of the following conditions is met:

1. The two plates have equal size
2. The parent has size 1 (or no plate)

Table below shows an example of compatible plates for a child node and its two parent nodes:

node	plates					
parent1		3	1	1	1	8	10
parent2			1	1	5	1	10
child	5	3	1	7	5	8	10

Plates in deterministic nodes

Note that plates can be defined explicitly only for stochastic nodes. For deterministic nodes, the plates are defined implicitly by the plate broadcasting rules from the parents. Deterministic nodes do not need more plates than this because there is no randomness. The deterministic node would just have the same value over the extra plates, but it is not necessary to do this explicitly because the child nodes of the deterministic node can utilize broadcasting anyway. Thus, there is no point in having extra plates in deterministic nodes, and for this reason, deterministic nodes do not use `plates` keyword argument.

Plates in constants

It is useful to understand how the plates and the shape of a random variable are connected. The shape of an array which contains all the plates of a random variable is the concatenation of the plates and the shape of the variable. For instance, consider a 2-dimensional Gaussian variable with plates `(3,)`. If you want the value of the constant mean vector and constant precision matrix to vary between plates, they are given as `(3,2)`-shape and `(3,2,2)`-shape arrays, respectively:

```
>>> import numpy as np
>>> mu = [ [0,0], [1,1], [2,2] ]
>>> Lambda = [ [[1.0, 0.0],
...             [0.0, 1.0]],
...            [[1.0, 0.9],
...             [0.9, 1.0]],
...            [[1.0, -0.3],
...             [-0.3, 1.0]] ]
>>> X = Gaussian(mu, Lambda)
>>> np.shape(mu)
(3, 2)
>>> np.shape(Lambda)
(3, 2, 2)
>>> X.plates
(3,)
```

Thus, the leading axes of an array are the plate axes and the trailing axes are the random variable axes. In the example above, the mean vector has plates `(3,)` and shape `(2,)`, and the precision matrix has plates `(3,)` and shape `(2,2)`.

Factorization of plates

It is important to undestand the independency structure the plates induce for the model. First, the repetitions defined by a plate are independent a priori given the parents. Second, the repetitions are independent in the posterior approximation, that is, the posterior approximation factorizes with respect to plates. Thus, the plates also have an effect on the independence structure of the posterior approximation, not only prior. If dependencies between a set of variables need to be handled, that set must be handled as a some kind of multi-dimensional variable.

Irregular plates

The handling of plates is not always as simple as described above. There are cases in which the plates of the parents do not map directly to the plates of the child node. The user API should mention such irregularities.

For instance, the parents of a mixture distribution have a plate which contains the different parameters for each cluster, but the variable from the mixture distribution does not have that plate:

```
>>> from bayespy.nodes import Gaussian, Wishart, Categorical, Mixture
>>> mu = Gaussian([[0], [0], [0]], [ [[1]], [[1]], [[1]] ])
>>> Lambda = Wishart(1, [ [[1]], [[1]], [[1]]])
>>> Z = Categorical([1/3, 1/3, 1/3], plates=(100,))
>>> X = Mixture(Z, Gaussian, mu, Lambda)
>>> mu.plates
(3,)
>>> Lambda.plates
(3,)
>>> Z.plates
(100,)
>>> X.plates
(100,)
```

The plates `(3,)` and `(100,)` should not broadcast according to the rules mentioned above. However, when validating the plates, `Mixture` removes the plate which corresponds to the clusters in `mu` and `Lambda`. Thus, `X` has plates which are the result of broadcasting plates `()` and `(100,)` which equals `(100,)`.

Also, sometimes the plates of the parents may be mapped to the variable axes. For instance, an automatic relevance determination (ARD) prior for a Gaussian variable is constructed by giving the diagonal elements of the precision matrix (or tensor). The Gaussian variable itself can be a scalar, a vector, a matrix or a tensor. A set of five 4×3 -dimensional Gaussian matrices with ARD prior is constructed as:

```
>>> from bayespy.nodes import GaussianARD, Gamma
>>> tau = Gamma(1, 1, plates=(5,4,3))
>>> X = GaussianARD(0, tau, shape=(4,3))
>>> tau.plates
(5, 4, 3)
>>> X.plates
(5,)
```

Note how the last two plate axes of `tau` are mapped to the variable axes of `X` with shape `(4,3)` and the plates of `X` are obtained by taking the remaining leading plate axes of `tau`.

2.3.4 Example model: Principal component analysis

Now, we'll construct a bit more complex model which will be used in the following sections. The model is a probabilistic version of principal component analysis (PCA):

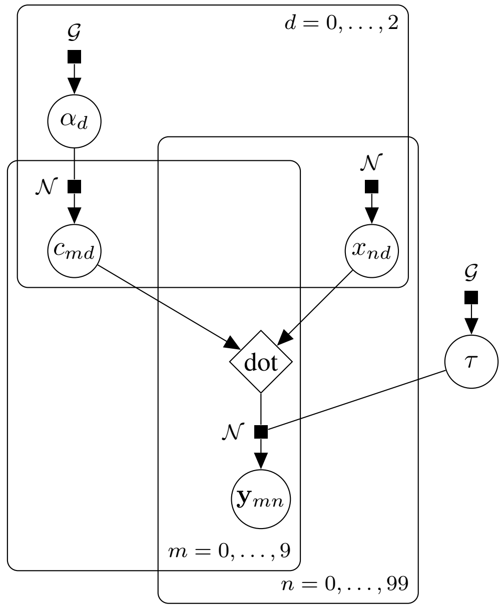
$$\mathbf{Y} = \mathbf{C}\mathbf{X}^T + \text{noise}$$

where $\mathbf{Y}$ is $M \times N$ data matrix, $\mathbf{C}$ is $M \times D$ loading matrix, $\mathbf{X}$ is $N \times D$ state matrix, and noise is isotropic Gaussian. The dimensionality D is usually assumed to be much smaller than M and N.

A probabilistic formulation can be written as:

$$\begin{aligned}
p(\mathbf{Y}) &= \prod_{m=0}^{M-1} \prod_{n=0}^{N-1} \mathcal{N}(y_{mn}|\mathbf{c}_m^T \mathbf{x}_n, \tau) \\
p(\mathbf{X}) &= \prod_{n=0}^{N-1} \prod_{d=0}^{D-1} \mathcal{N}(x_{nd}|0, 1) \\
p(\mathbf{C}) &= \prod_{m=0}^{M-1} \prod_{d=0}^{D-1} \mathcal{N}(c_{md}|0, \alpha_d) \\
p(\boldsymbol{\alpha}) &= \prod_{d=0}^{D-1} \mathcal{G}(\alpha_d|10^{-3}, 10^{-3}) \\
p(\tau) &= \mathcal{G}(\tau|10^{-3}, 10^{-3})
\end{aligned}$$

where we have given automatic relevance determination (ARD) prior for $\mathbf{C}$. This can be visualized as a graphical model:



Now, let us construct this model in BayesPy. First, we'll define the dimensionality of the latent space in our model:

```
>>> D = 3
```

Then the prior for the latent states $\mathbf{X}$:

```
>>> X = GaussianARD(0, 1,
...                 shape=(D,),
...                 plates=(1,100),
...                 name='X')
```

Note that the shape of `X` is `(D,)`, although the latent dimensions are marked with a plate in the graphical model and they are conditionally independent in the prior. However, we want to (and need to) model the posterior dependency of the latent dimensions, thus we cannot factorize them, which would happen if we used `plates=(1,100,D)` and `shape=()`. The first plate axis with size 1 is given just for clarity.

The prior for the ARD parameters $\boldsymbol{\alpha}$ of the loading matrix:

```
>>> alpha = Gamma(1e-3, 1e-3,
...               plates=(D,),
...               name='alpha')
```

The prior for the loading matrix $\mathbf{C}$:

```
>>> C = GaussianARD(0, alpha,
...                 shape=(D,),
...                 plates=(10,1),
...                 name='C')
```

Again, note that the shape is the same as for `X` for the same reason. Also, the plates of `alpha`, `(D,)`, are mapped to the full shape of the node `C`, `(10,1,D)`, using standard broadcasting rules.

The dot product is just a deterministic node:

```
>>> F = Dot(C, X)
```

However, note that `Dot` requires that the input Gaussian nodes have the same shape and that this shape has exactly one axis, that is, the variables are vectors. This the reason why we used shape `(D,)` for `X` and `C` but from a bit different perspective. The node computes the inner product of D-dimensional vectors resulting in plates `(10,100)` broadcasted from the plates `(1,100)` and `(10,1)`:

```
>>> F.plates
(10, 100)
```

The prior for the observation noise τ:

```
>>> tau = Gamma(1e-3, 1e-3, name='tau')
```

Finally, the observations are conditionally independent Gaussian scalars:

```
>>> Y = GaussianARD(F, tau, name='Y')
```

Now we have defined our model and the next step is to observe some data and to perform inference.

2.4 Performing inference

Approximation of the posterior distribution can be divided into several steps:

- Observe some nodes
- Choose the inference engine
- Initialize the posterior approximation
- Run the inference algorithm

In order to illustrate these steps, we'll be using the PCA model constructed in the previous section.

2.4.1 Observing nodes

First, let us generate some toy data:

```
>>> c = np.random.randn(10, 2)
>>> x = np.random.randn(2, 100)
>>> data = np.dot(c, x) + 0.1*np.random.randn(10, 100)
```

The data is provided by simply calling `observe` method of a stochastic node:

```
>>> Y.observe(data)
```

It is important that the shape of the `data` array matches the plates and shape of the node `Y`. For instance, if `Y` was `Wishart` node for 3×3 matrices with plates `(5,1,10)`, the full shape of `Y` would be `(5,1,10,3,3)`. The `data` array should have this shape exactly, that is, no broadcasting rules are applied.

Missing values

It is possible to mark missing values by providing a mask which is a boolean array:

```
>>> Y.observe(data, mask=[[True], [False], [False], [True], [True],
...                       [False], [True], [True], [True], [False]])
```

`True` means that the value is observed and `False` means that the value is missing. The shape of the above mask is `(10,1)`, which broadcasts to the plates of Y, `(10,100)`. Thus, the above mask means that the second, third, sixth and tenth rows of the 10×100 data matrix are missing.

The mask is applied to the *plates*, not to the data array directly. This means that it is not possible to observe a random variable partially, each repetition defined by the plates is either fully observed or fully missing. Thus, the mask is applied to the plates. It is often possible to circumvent this seemingly tight restriction by adding an observable child node which factorizes more.

The shape of the mask is broadcasted to plates using standard NumPy broadcasting rules. So, if the variable has plates `(5,1,10)`, the mask could have a shape `()`, `(1,)`, `(1,1)`, `(1,1,1)`, `(10,)`, `(1,10)`, `(1,1,10)`, `(5,1,1)` or `(5,1,10)`. In order to speed up the inference, missing values are automatically integrated out if they are not needed as latent variables to child nodes. This leads to faster convergence and more accurate approximations.

2.4.2 Choosing the inference method

Inference methods can be found in `bayespy.inference` package. Currently, only variational Bayesian approximation is implemented (`bayespy.inference.VB`). The inference engine is constructed by giving the stochastic nodes of the model.

```
>>> from bayespy.inference import VB
>>> Q = VB(Y, C, X, alpha, tau)
```

There is no need to give any deterministic nodes. Currently, the inference engine does not automatically search for stochastic parents and children, thus it is important that all stochastic nodes of the model are given. This should be made more robust in future versions.

A node of the model can be obtained by using the name of the node as a key:

```
>>> Q['X']
<bayespy.inference.vmp.nodes.gaussian.GaussianARD object at 0x...>
```

Note that the returned object is the same as the node object itself:

```
>>> Q['X'] is X
True
```

Thus, one may use the object `X` when it is available. However, if the model and the inference engine are constructed in another function or module, the node object may not be available directly and this feature becomes useful.

2.4.3 Initializing the posterior approximation

The inference engines give some initialization to the stochastic nodes by default. However, the inference algorithms can be sensitive to the initialization, thus it is sometimes necessary to have better control over the initialization. For VB, the following initialization methods are available:

- `initialize_from_prior`: Use the current states of the parent nodes to update the node. This is the default initialization.
- `initialize_from_parameters`: Use the given parameter values for the distribution.
- `initialize_from_value`: Use the given value for the variable.
- `initialize_from_random`: Draw a random value for the variable. The random sample is drawn from the current state of the node's distribution.

Note that `initialize_from_value` and `initialize_from_random` initialize the distribution with a value of the variable instead of parameters of the distribution. Thus, the distribution is actually a delta distribution with a peak on the value after the initialization. This state of the distribution does not have proper natural parameter values nor normalization, thus the VB lower bound terms are `np.nan` for this initial state.

These initialization methods can be used to perform even a bit more complex initializations. For instance, a Gaussian distribution could be initialized with a random mean and variance 0.1. In our PCA model, this can be obtained by

```
>>> X.initialize_from_parameters(np.random.randn(1, 100, D), 10)
```

Note that the shape of the random mean is the sum of the plates `(1, 100)` and the variable shape `(D,)`. In addition, instead of variance, `GaussianARD` uses precision as the second parameter, thus we initialized the variance to $\frac{1}{10}$. This random initialization is important in our PCA model because the default initialization gives `C` and `X` zero mean. If the mean of the other variable was zero when the other is updated, the other variable gets zero mean too. This would lead to an update algorithm where both means remain zeros and effectively no latent space is found. Thus, it is important to give non-zero random initialization for `X` if `C` is updated before `X` the first time. It is typical that at least some nodes need be initialized with some randomness.

By default, nodes are initialized with the method `initialize_from_prior`. The method is not very time consuming but if for any reason you want to avoid that default initialization computation, you can provide `initialize=False` when creating the stochastic node. However, the node does not have a proper state in that case, which leads to errors in VB learning unless the distribution is initialized using the above methods.

2.4.4 Running the inference algorithm

The approximation methods are based on iterative algorithms, which can be run using `update` method. By default, it takes one iteration step updating all nodes once:

```
>>> Q.update()
Iteration 1: loglike=-9.305259e+02 (... seconds)
```

The `loglike` tells the VB lower bound. The order in which the nodes are updated is the same as the order in which the nodes were given when creating `Q`. If you want to change the order or update only some of the nodes, you can give as arguments the nodes you want to update and they are updated in the given order:

```
>>> Q.update(C, X)
Iteration 2: loglike=-8.818976e+02 (... seconds)
```

It is also possible to give the same node several times:

```
>>> Q.update(C, X, C, tau)
Iteration 3: loglike=-8.071222e+02 (... seconds)
```

Note that each call to `update` is counted as one iteration step although not variables are necessarily updated. Instead of doing one iteration step, `repeat` keyword argument can be used to perform several iteration steps:

```
>>> Q.update(repeat=10)
Iteration 4: loglike=-7.167588e+02 (... seconds)
Iteration 5: loglike=-6.827873e+02 (... seconds)
Iteration 6: loglike=-6.259477e+02 (... seconds)
Iteration 7: loglike=-4.725400e+02 (... seconds)
Iteration 8: loglike=-3.270816e+02 (... seconds)
Iteration 9: loglike=-2.208865e+02 (... seconds)
Iteration 10: loglike=-1.658761e+02 (... seconds)
Iteration 11: loglike=-1.469468e+02 (... seconds)
Iteration 12: loglike=-1.420311e+02 (... seconds)
Iteration 13: loglike=-1.405139e+02 (... seconds)
```

The VB algorithm stops automatically if it converges, that is, the relative change in the lower bound is below some threshold:

```
>>> Q.update(repeat=1000)
Iteration 14: loglike=-1.396481e+02 (... seconds)
...
Iteration 488: loglike=-1.224106e+02 (... seconds)
Converged at iteration 488.
```

Now the algorithm stopped before taking 1000 iteration steps because it converged. The relative tolerance can be adjusted by providing `tol` keyword argument to the `update` method:

```
>>> Q.update(repeat=10000, tol=1e-6)
Iteration 489: loglike=-1.224094e+02 (... seconds)
...
Iteration 847: loglike=-1.222506e+02 (... seconds)
Converged at iteration 847.
```

Making the tolerance smaller, may improve the result but it may also significantly increase the iteration steps until convergence.

Instead of using `update` method of the inference engine `VB`, it is possible to use the `update` methods of the nodes directly as

```
>>> C.update()
```

or

```
>>> Q['C'].update()
```

However, this is not recommended, because the `update` method of the inference engine `VB` is a wrapper which, in addition to calling the nodes' `update` methods, checks for convergence and does a few other useful minor things. But if for any reason these direct update methods are needed, they can be used.

Parameter expansion

Sometimes the VB algorithm converges very slowly. This may happen when the variables are strongly coupled in the true posterior but factorized in the approximate posterior. This coupling leads to zigzagging of the variational parameters which progresses slowly. One solution to this problem is to use parameter expansion. The idea is to add an auxiliary variable which parameterizes the posterior approximation of several variables. Then optimizing this auxiliary variable actually optimizes several posterior approximations jointly leading to faster convergence.

The parameter expansion is model specific. Currently in BayesPy, only state-space models have built-in parameter expansions available. These state-space models contain a variable which is a dot product of two variables (plus some noise):

$$y = \mathbf{c}^T\mathbf{x} + \text{noise}$$

The parameter expansion can be motivated by noticing that we can add an auxiliary variable which rotates the variables $\mathbf{c}$ and $\mathbf{x}$ so that the dot product is unaffected:

$$y = \mathbf{c}^T\mathbf{x} + \text{noise} = \mathbf{c}^T\mathbf{R}\mathbf{R}^{-1}\mathbf{x} + \text{noise} = (\mathbf{R}^T\mathbf{c})^T(\mathbf{R}^{-1}\mathbf{x}) + \text{noise}$$

Now, applying this rotation to the posterior approximations $q(\mathbf{c})$ and $q(\mathbf{x})$, and optimizing the VB lower bound with respect to the rotation leads to parameterized joint optimization of $\mathbf{c}$ and $\mathbf{x}$.

The available parameter expansion methods are in module `transformations`:

```
>>> from bayespy.inference.vmp import transformations
```

First, you create the rotation transformations for the two variables:

```
>>> rotX = transformations.RotateGaussianARD(X)
>>> rotC = transformations.RotateGaussianARD(C, alpha)
```

Here, the rotation for `C` provides the ARD parameters `alpha` so they are updated simultaneously. In addition to `RotateGaussianARD`, there are a few other built-in rotations defined, for instance, `RotateGaussian` and `RotateGaussianMarkovChain`. It is extremely important that the model satisfies the assumptions made by the rotation class and the user is mostly responsible for this. The optimizer for the rotations is constructed by giving the two rotations and the dimensionality of the rotated space:

```
>>> R = transformations.RotationOptimizer(rotC, rotX, D)
```

Now, calling `rotate` method will find optimal rotation and update the relevant nodes (`X`, `C` and `alpha`) accordingly:

```
>>> R.rotate()
```

Let us see how our iteration would have gone if we had used this parameter expansion. First, let us re-initialize our nodes and VB algorithm:

```
>>> alpha.initialize_from_prior()
>>> C.initialize_from_prior()
>>> X.initialize_from_parameters(np.random.randn(1, 100, D), 10)
>>> tau.initialize_from_prior()
>>> Q = VB(Y, C, X, alpha, tau)
```

Then, the rotation is set to run after each iteration step:

```
>>> Q.callback = R.rotate
```

Now the iteration converges to the relative tolerance 10^{-6} much faster:

```
>>> Q.update(repeat=1000, tol=1e-6)
Iteration 1: loglike=-9.363500e+02 (... seconds)
...
Iteration 18: loglike=-1.221354e+02 (... seconds)
Converged at iteration 18.
```

The convergence took 18 iterations with rotations and 488 or 847 iterations without the parameter expansion. In addition, the lower bound is improved slightly. One can compare the number of iteration steps in this case because the cost per iteration step with or without parameter expansion is approximately the same. Sometimes the parameter expansion can have the drawback that it converges to a bad local optimum. Usually, this can be solved by updating the nodes near the observations a few times before starting to update the hyperparameters and to use parameter expansion. In any case, the parameter expansion is practically necessary when using state-space models in order to converge to a proper solution in a reasonable time.

2.5 Examining the results

After the results have been obtained, it is important to be able to examine the results easily. The results can be examined either numerically by inspecting numerical arrays or visually by plotting distributions of the nodes. In addition, the posterior distributions can be visualized during the learning algorithm and the results can saved into a file.

2.5.1 Plotting the results

The module `plot` offers some plotting basic functionality:

```
>>> import bayespy.plot as bpplt
```

The module contains `matplotlib.pyplot` module if the user needs that. For instance, interactive plotting can be enabled as:

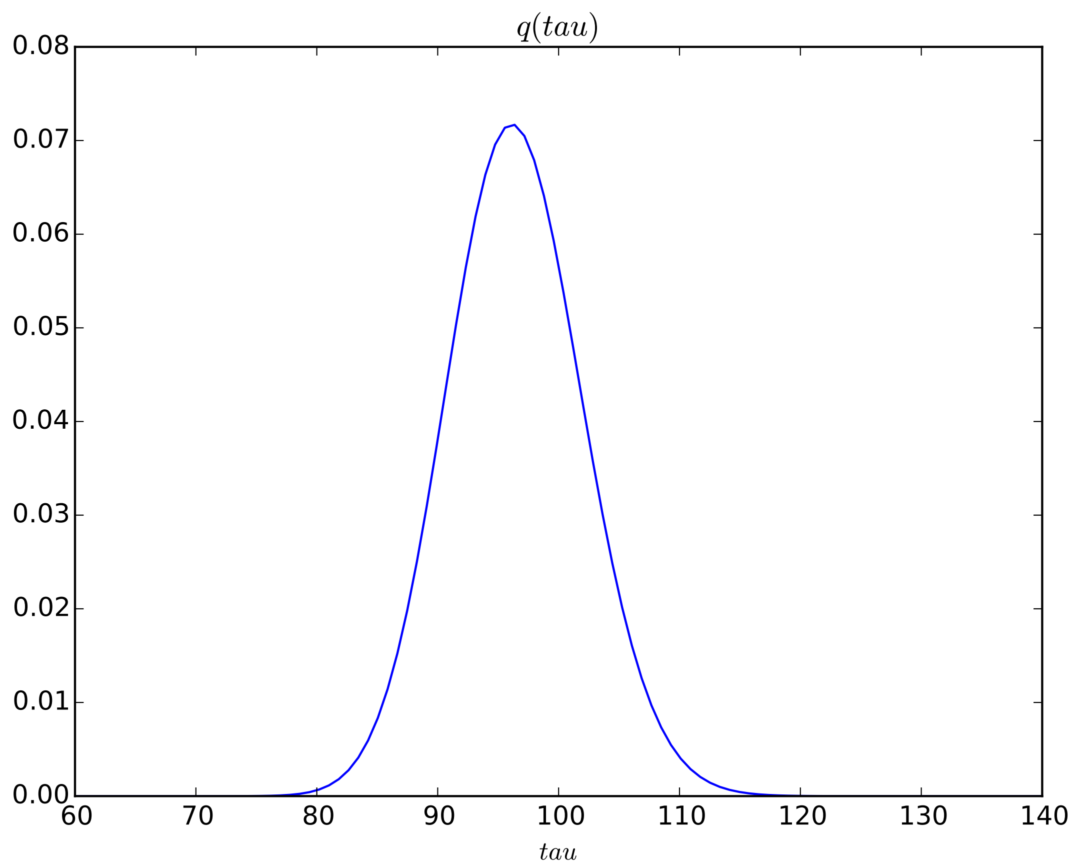
```
>>> bpplt.pyplot.ion()
```

The `plot` module contains some functions but it is not a very comprehensive collection, thus the user may need to write some problem- or model-specific plotting functions. The current collection is:

- `pdf()`: show probability density function of a scalar
- `contour()`: show probability density function of two-element vector
- `hinton()`: show the Hinton diagram
- `plot()`: show value as a function

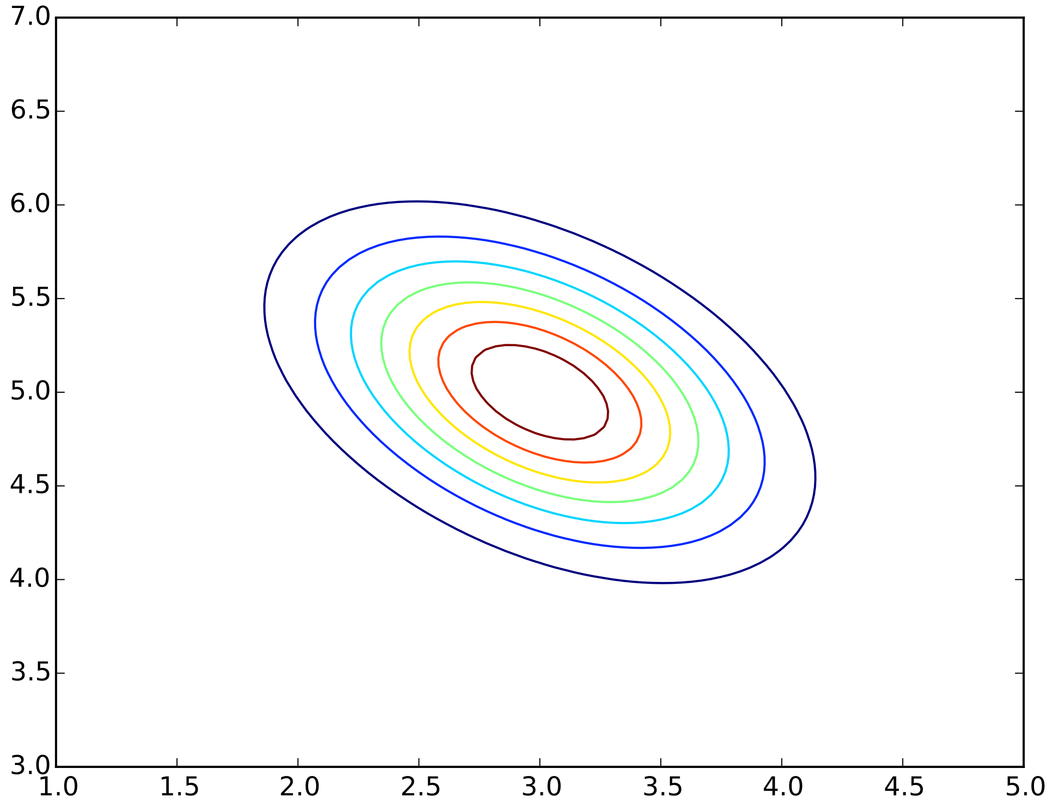
The probability density function of a scalar random variable can be plotted using the function `pdf()`:

```
>>> bpplt.pyplot.figure()
<matplotlib.figure.Figure object at 0x...>
>>> bpplt.pdf(Q['tau'], np.linspace(60, 140, num=100))
[<matplotlib.lines.Line2D object at 0x...>]
```



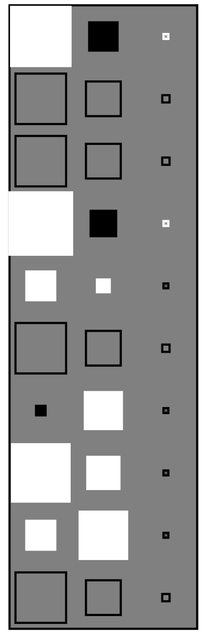
The variable `tau` models the inverse variance of the noise, for which the true value is $0.1^{-2} = 100$. Thus, the posterior captures the true value quite accurately. Similarly, the function `contour()` can be used to plot the probability density function of a 2-dimensional variable, for instance:

```
>>> V = Gaussian([3, 5], [[4, 2], [2, 5]])
>>> bpplt.pyplot.figure()
<matplotlib.figure.Figure object at 0x...>
>>> bpplt.contour(V, np.linspace(1, 5, num=100), np.linspace(3, 7, num=100))
<matplotlib.contour.QuadContourSet object at 0x...>
```



Both `pdf()` and `contour()` require that the user provides the grid on which the probability density function is computed. They also support several keyword arguments for modifying the output, similarly as `plot` and `contour` in `matplotlib.pyplot`. These functions can be used only for stochastic nodes. A few other plot types are also available as built-in functions. A Hinton diagram can be plotted as:

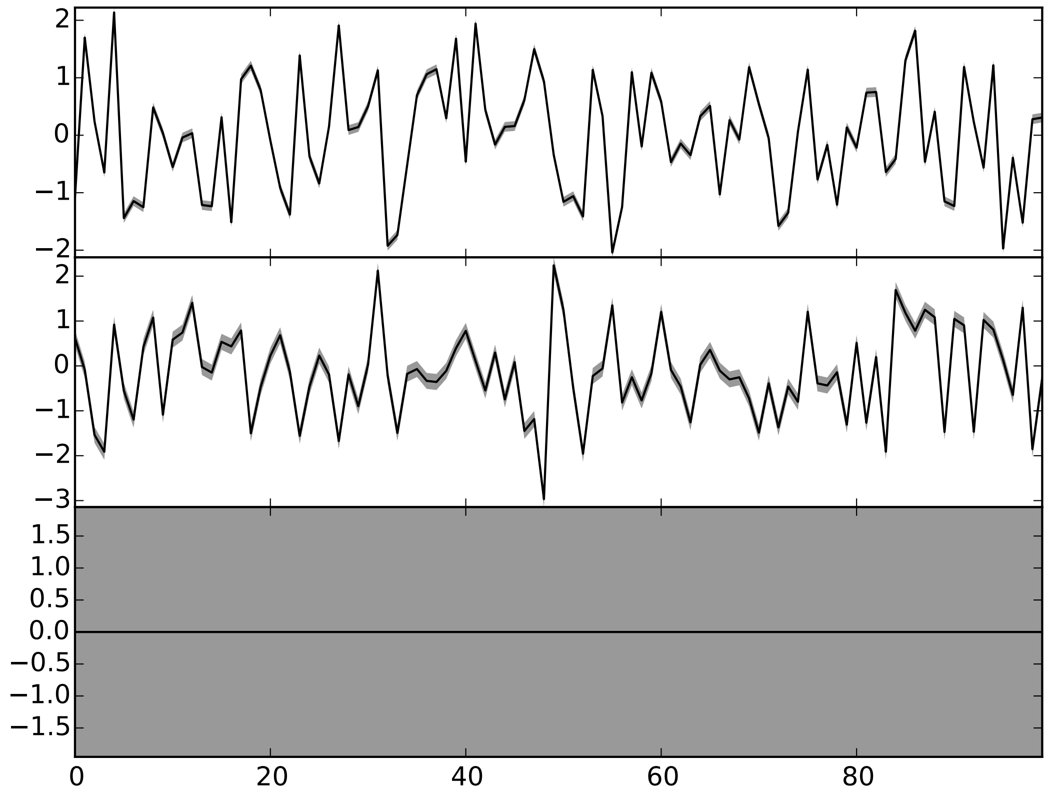
```
>>> bpplt.pyplot.figure()
<matplotlib.figure.Figure object at 0x...>
>>> bpplt.hinton(C)
```



The diagram shows the elements of the matrix C. The size of the filled rectangle corresponds to the absolute value of the element mean, and white and black correspond to positive and negative values, respectively. The non-filled rectangle shows standard deviation. From this diagram it is clear that the third column of C has been pruned out and the rows that were missing in the data have zero mean and column-specific variance. The function `hinton()` is a simple wrapper for node-specific Hinton diagram plotters, such as `gaussian_hinton()` and `dirichlet_hinton()`. Thus, the keyword arguments depend on the node which is plotted.

Another plotting function is `plot()`, which just plots the values of the node over one axis as a function:

```
>>> bpplt.pyplot.figure()
<matplotlib.figure.Figure object at 0x...>
>>> bpplt.plot(X, axis=-2)
```



Now, the `axis` is the second last axis which corresponds to $n = 0, \ldots, N-1$. As $D = 3$, there are three subplots. For Gaussian variables, the function shows the mean and two standard deviations. The plot shows that the third component has been pruned out, thus the method has been able to recover the true dimensionality of the latent space. It also has similar keyword arguments to `plot` function in `matplotlib.pyplot`. Again, `plot()` is a simple wrapper over node-specific plotting functions, thus it supports only some node classes.

2.5.2 Monitoring during the inference algorithm

It is possible to plot the distribution of the nodes during the learning algorithm. This is useful when the user is interested to see how the distributions evolve during learning and what is happening to the distributions. In order to utilize monitoring, the user must set plotters for the nodes that he or she wishes to monitor. This can be done either when creating the node or later at any time.

The plotters are set by creating a plotter object and providing this object to the node. The plotter is a wrapper of one of the plotting functions mentioned above: `PDFPlotter`, `ContourPlotter`, `HintonPlotter` or `FunctionPlotter`. Thus, our example model could use the following plotters:

```
>>> tau.set_plotter(bpplt.PDFPlotter(np.linspace(60, 140, num=100)))
>>> C.set_plotter(bpplt.HintonPlotter())
>>> X.set_plotter(bpplt.FunctionPlotter(axis=-2))
```

These could have been given at node creation as a keyword argument `plotter`:

```
>>> V = Gaussian([3, 5], [[4, 2], [2, 5]],
...              plotter=bpplt.ContourPlotter(np.linspace(1, 5, num=100),
...                                           np.linspace(3, 7, num=100)))
```

When the plotter is set, one can use the `plot` method of the node to perform plotting:

```
>>> V.plot()
<matplotlib.contour.QuadContourSet object at 0x...>
```

Nodes can also be plotted using the `plot` method of the inference engine:

```
>>> Q.plot('C')
```

This method remembers the figure in which a node has been plotted and uses that every time it plots the same node. In order to monitor the nodes during learning, it is possible to use the keyword argument `plot`:

```
>>> Q.update(repeat=5, plot=True, tol=np.nan)
Iteration 19: loglike=-1.221354e+02 (... seconds)
Iteration 20: loglike=-1.221354e+02 (... seconds)
Iteration 21: loglike=-1.221354e+02 (... seconds)
Iteration 22: loglike=-1.221354e+02 (... seconds)
Iteration 23: loglike=-1.221354e+02 (... seconds)
```

Each node which has a plotter set will be plotted after it is updated. Note that this may slow down the inference significantly if the plotting operation is time consuming.

2.5.3 Posterior parameters and moments

If the built-in plotting functions are not sufficient, it is possible to use `matplotlib.pyplot` for custom plotting. Each node has `get_moments` method which returns the moments and they can be used for plotting. Stochastic exponential family nodes have natural parameter vectors which can also be used. In addition to plotting, it is also possible to just print the moments or parameters in the console.

2.5.4 Saving and loading results

The results of the inference engine can be easily saved and loaded using `VB.save()` and `VB.load()` methods:

```
>>> Q.save(filename='tmp.hdf5')
>>> Q.load(filename='tmp.hdf5')
```

The results are stored in a HDF5 file. The user may set an autosave file in which the results are automatically saved regularly. Autosave filename can be set at creation time by `autosave_filename` keyword argument or later using `VB.set_autosave()` method. If autosave file has been set, the `VB.save()` and `VB.load()` methods use that file by default. In order for the saving to work, all stochastic nodes must have been given (unique) names.

However, note that these methods do *not* save nor load the node definitions. It means that the user must create the nodes and the inference engine and then use `VB.load()` to set the state of the nodes and the inference engine. If there are any differences in the model that was saved and the one which is tried to update using loading, then loading does not work. Thus, the user should keep the model construction unmodified in a Python file in order to be able to load the results later. Or if the user wishes to share the results, he or she must share the model construction Python file with the HDF5 results file.

2.6 Advanced topics

This section contains brief information on how to implement some advanced methods in BayesPy. These methods include Riemannian conjugate gradient methods, pattern search, simulated annealing, collapsed variational inference, stochastic variational inference and black box variational inference. In order to use these methods properly, the user should understand them to some extent. They are also considered experimental, thus you may encounter bugs or unimplemented features. In any case, these methods may provide huge performance improvements easily compared to the standard VB-EM algorithm.

2.6.1 Gradient-based optimization

Variational Bayesian learning basically means that the parameters of the approximate posterior distributions are optimized to maximize the lower bound of the marginal log likelihood [3]. This optimization can be done by using gradient-based optimization methods. In order to improve the gradient-based methods, it is recommended to take into account the information geometry by using the Riemannian (a.k.a. natural) gradient. In fact, the standard VB-EM algorithm is equivalent to a gradient ascent method which uses the Riemannian gradient and step length 1. Thus, it is natural to try to improve this method by using non-linear conjugate gradient methods instead of gradient ascent. These optimization methods are especially useful when the VB-EM update equations are not available but one has to use fixed form approximation. But it is possible that the Riemannian conjugate gradient method improve performance even when the VB-EM update equations are available.

The optimization algorithm in `VB.optimize()` has a simple interface. Instead of using the default Riemannian geometry, one can use the Euclidean geometry by giving `riemannian=False`. It is also possible to choose the optimization method from gradient ascent (`method='gradient'`) or conjugate gradient methods (only `method='fletcher-reeves'` implemented at the moment). For instance, we could optimize nodes `C` and `X` jointly using Euclidean gradient ascent as:

```
>>> Q = VB(Y, C, X, alpha, tau)
>>> Q.optimize(C, X, riemannian=False, method='gradient', maxiter=5)
Iteration ...
```

Note that this is very inefficient way of updating those nodes (bad geometry and not using conjugate gradients). Thus, one should understand the idea of these optimization methods, otherwise one may do something extremely inefficient. Most likely this method can be found useful in combination with the advanced tricks in the following sections.

Note: The Euclidean gradient has not been implemented for all nodes yet. The Euclidean gradient is required by the Euclidean geometry based optimization but also by the conjugate gradient methods in the Riemannian geometry. Thus, the Riemannian conjugate gradient may not yet work for all models.

It is possible to construct custom optimization algorithms with the tools provided by `VB`. For instance, `VB.get_parameters()` and `VB.set_parameters()` can be used to handle the parameters of nodes. `VB.get_gradients()` is used for computing the gradients of nodes. The parameter and gradient objects are not numerical arrays but more complex nested lists not meant to be accessed by the user. Thus, for simple arithmetics with the parameter and gradient objects, use functions `VB.add()` and `VB.dot()`. Finally, `VB.compute_lowerbound()` and `VB.has_converged()` can be used to monitor the lower bound.

2.6.2 Collapsed inference

The optimization method can be used efficiently in such a way that some of the variables are collapsed, that is, marginalized out [1]. The collapsed variables must be conditionally independent given the observations and all other variables. Probably, one also wants that the size of the marginalized variables is large and the size of the optimized variables is small. For instance, in our PCA example, we could optimize as follows:

```
>>> Q.optimize(C, tau, maxiter=10, collapsed=[X, alpha])
Iteration ...
```

The collapsed variables are given as a list. This optimization does basically the following: It first computes the gradients for `C` and `tau` and takes an update step using the desired optimization method. Then, it updates the collapsed variables by using the standard VB-EM update equations. These two steps are taken in turns. Effectively, this corresponds to collapsing the variables `X` and `alpha` in a particular way. The point of this method is that the number of parameters in the optimization reduces significantly and the collapsed variables are updated optimally. For more details, see [1].

It is possible to use this method in such a way, that the collapsed variables are not conditionally independent given the observations and all other variables. However, in that case, the method does not anymore correspond to collapsing the variables but just using VB-EM updates after gradient-based updates. The method does not check for conditional independence, so the user is free to do this.

Note: Although the Riemannian conjugate gradient method has not yet been implemented for all nodes, it may be possible to collapse those nodes and optimize the other nodes for which the Euclidean gradient is already implemented.

2.6.3 Pattern search

The pattern search method estimates the direction in which the approximate posterior distributions are updating and performs a line search in that direction [4]. The search direction is based on the difference in the VB parameters on successive updates (or several updates). The idea is that the VB-EM algorithm may be slow because it just zigzags and this can be fixed by moving to the direction in which the VB-EM is slowly moving.

BayesPy offers a simple built-in pattern search method `VB.pattern_search()`. The method updates the nodes twice, measures the difference in the parameters and performs a line search with a small number of function evaluations:

```
>>> Q.pattern_search(C, X)
Iteration ...
```

Similarly to the collapsed optimization, it is possible to collapse some of the variables in the pattern search. The same rules of conditional independence apply as above. The collapsed variables are given as list:

```
>>> Q.pattern_search(C, tau, collapsed=[X, alpha])
Iteration ...
```

Also, a maximum number of iterations can be set by using `maxiter` keyword argument. It is not always obvious whether a pattern search will improve the rate of convergence or not but if it seems that the convergence is slow because of zigzagging, it may be worth a try. Note that the computational cost of the pattern search is quite high, thus it is not recommended to perform it after every VB-EM update but every now and then, for instance, after every 10 iterations. In addition, it is possible to write a more customized VB learning algorithm which uses pattern searches by using the different methods of `VB` discussed above.

2.6.4 Deterministic annealing

The standard VB-EM algorithm converges to a local optimum which can often be inferior to the global optimum and many other local optima. Deterministic annealing aims at finding a better local optimum, hopefully even the global optimum [5]. It does this by increasing the weight on the entropy of the posterior approximation in the VB lower bound. Effectively, the annealed lower bound becomes closer to a uniform function instead of the original multimodal lower bound. The weight on the entropy is recovered slowly and the optimization is much more robust to initialization.

In BayesPy, the annealing can be set by using `VB.set_annealing()`. The given annealing should be in range $(0, 1]$ but this is not validated in case the user wants to do something experimental. If annealing is set to 1, the original

VB lower bound is recovered. Annealing with 0 would lead to an improper uniform distribution, thus it will lead to errors. The entropy term is weighted by the inverse of this annealing term. An alternative view is that the model probability density functions are raised to the power of the annealing term.

Typically, the annealing is used in such a way that the annealing is small at the beginning and increased after every convergence of the VB algorithm until value 1 is reached. After the annealing value is increased, the algorithm continues from where it had just converged. The annealing can be used for instance as:

```
>>> beta = 0.1
>>> while beta < 1.0:
...     beta = min(beta*1.5, 1.0)
...     Q.set_annealing(beta)
...     Q.update(repeat=100, tol=1e-4)
Iteration ...
```

Here, the `tol` keyword argument is used to adjust the threshold for convergence. In this case, it is a bit larger than by default so the algorithm does not need to converge perfectly but a rougher convergence is sufficient for the next iteration with a new annealing value.

2.6.5 Stochastic variational inference

In stochastic variational inference [2], the idea is to use mini-batches of large datasets to compute noisy gradients and learn the VB distributions by using stochastic gradient ascent. In order for it to be useful, the model must be such that it can be divided into "intermediate" and "global" variables. The number of intermediate variables increases with the data but the number of global variables remains fixed. The global variables are learnt in the stochastic optimization.

By denoting the data as $Y = [Y_1, \ldots, Y_N]$, the intermediate variables as $Z = [Z_1, \ldots, Z_N]$ and the global variables as θ, the model needs to have the following structure:

$$p(Y, Z, \theta) = p(\theta) \prod_{n=1}^{N} p(Y_n | Z_n, \theta) p(Z_n | \theta)$$

The algorithm consists of three steps which are iterated: 1) a random mini-batch of the data is selected, 2) the corresponding intermediate variables are updated by using normal VB update equations, and 3) the global variables are updated with (stochastic) gradient ascent as if there was as many replications of the mini-batch as needed to recover the original dataset size.

The learning rate for the gradient ascent must satisfy:

$$\sum_{i=1}^{\infty} \alpha_i = \infty \qquad \text{and} \qquad \sum_{i=1}^{\infty} \alpha^2 < \infty,$$

where i is the iteration number. An example of a valid learning parameter is $\alpha_i = (\delta + i)^{-\gamma}$, where $\delta \geq 0$ is a delay and $\gamma \in (0.5, 1]$ is a forgetting rate.

Stochastic variational inference is relatively easy to use in BayesPy. The idea is that the user creates a model for the size of a mini-batch and specifies a multiplier for those plate axes that are replicated. For the PCA example, the mini-batch model can be costructed as follows. We decide to use `X` as an intermediate variable and the other variables are global. The global variables `alpha`, `C` and `tau` are constructed identically as before. The intermediate variable `X` is constructed as:

```
>>> X = GaussianARD(0, 1,
...                 shape=(D,),
...                 plates=(1,5),
...                 plates_multiplier=(1,20),
...                 name='X')
```

Note that the plates are `(1,5)` whereas they are `(1,100)` in the full model. Thus, we need to provide a plates multiplier `(1,20)` to define how the plates are replicated to get the full dataset. These multipliers do not need to be integers, in this case the latter plate axis is multiplied by $100/5 = 20$. The remaining variables are defined as before:

```
>>> F = Dot(C, X)
>>> Y = GaussianARD(F, tau, name='Y')
```

Note that the plates of `Y` and `F` also correspond to the size of the mini-batch and they also deduce the plate multipliers from their parents, thus we do not need to specify the multiplier here explicitly (although it is ok to do so).

Let us construct the inference engine for the new mini-batch model:

```
>>> Q = VB(Y, C, X, alpha, tau)
```

Use random initialization for `C` to break the symmetry in `C` and `X`:

```
>>> C.initialize_from_random()
```

Then, stochastic variational inference algorithm could look as follows:

```
>>> Q.ignore_bound_checks = True
>>> for n in range(200):
...     subset = np.random.choice(100, 5)
...     Y.observe(data[:,subset])
...     Q.update(X)
...     learning_rate = (n + 2.0) ** (-0.7)
...     Q.gradient_step(C, alpha, tau, scale=learning_rate)
Iteration ...
```

First, we ignore the bound checks because they are noisy. Then, the loop consists of three parts: 1) Draw a random mini-batch of the data (5 samples from 100). 2) Update the intermediate variable `X`. 3) Update global variables with gradient ascent using a proper learning rate.

2.6.6 Black-box variational inference

TODO

CHAPTER

THREE

EXAMPLES

3.1 Linear regression

3.1.1 Data

The true parameters of the linear regression:

```
>>> k = 2 # slope
>>> c = 5 # bias
>>> s = 2 # noise standard deviation
```

Generate data:

```
>>> import numpy as np
>>> x = np.arange(10)
>>> y = k*x + c + s*np.random.randn(10)
```

3.1.2 Model

The regressors, that is, the input data:

```
>>> X = np.vstack([x, np.ones(len(x))]).T
```

Note that we added a column of ones to the regressor matrix for the bias term. We model the slope and the bias term in the same node so we do not factorize between them:

```
>>> from bayespy.nodes import GaussianARD
>>> B = GaussianARD(0, 1e-6, shape=(2,))
```

The first element is the slope which multiplies `x` and the second element is the bias term which multiplies the constant ones. Now we compute the dot product of `X` and `B`:

```
>>> from bayespy.nodes import SumMultiply
>>> F = SumMultiply('i,i', B, X)
```

The noise parameter:

```
>>> from bayespy.nodes import Gamma
>>> tau = Gamma(1e-3, 1e-3)
```

The noisy observations:

```
>>> Y = GaussianARD(F, tau)
```

3.1.3 Inference

Observe the data:

```
>>> Y.observe(y)
```

Construct the variational Bayesian (VB) inference engine by giving all stochastic nodes:

```
>>> from bayespy.inference import VB
>>> Q = VB(Y, B, tau)
```

Iterate until convergence:

```
>>> Q.update(repeat=1000)
Iteration 1: loglike=-4.595948e+01 (... seconds)
...
Iteration 5: loglike=-4.495017e+01 (... seconds)
Converged at iteration 5.
```

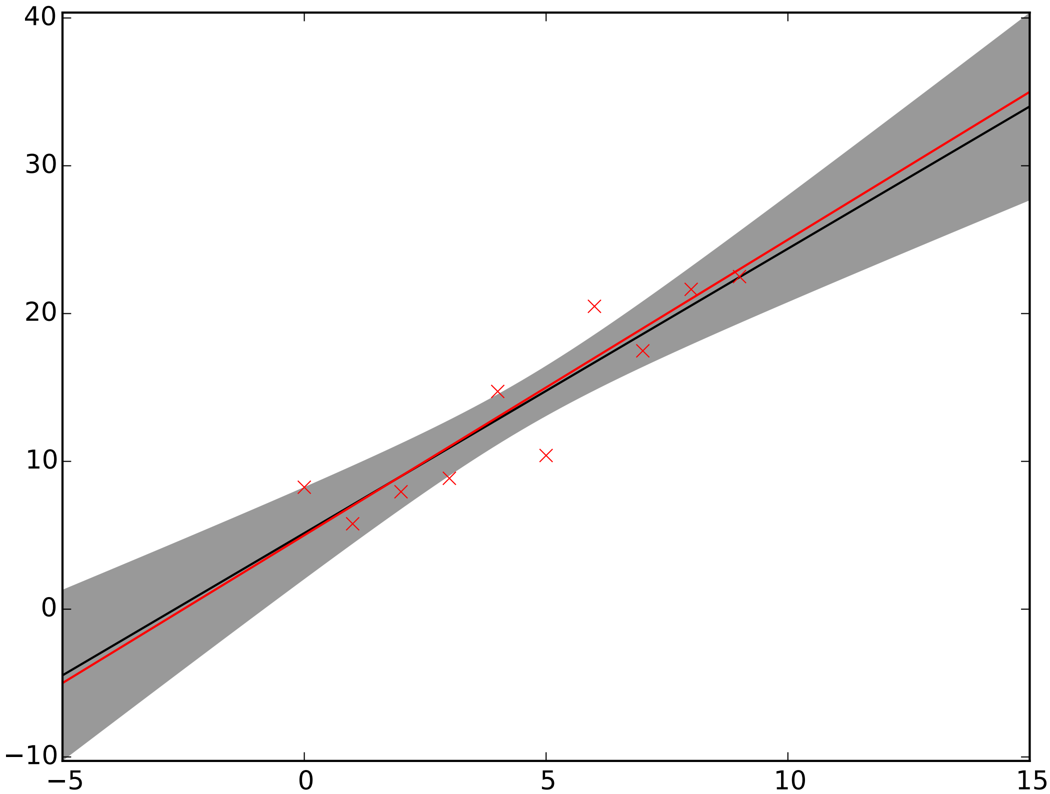
3.1.4 Results

Create a simple predictive model for new inputs:

```
>>> xh = np.linspace(-5, 15, 100)
>>> Xh = np.vstack([xh, np.ones(len(xh))]).T
>>> Fh = SumMultiply('i,i', B, Xh)
```

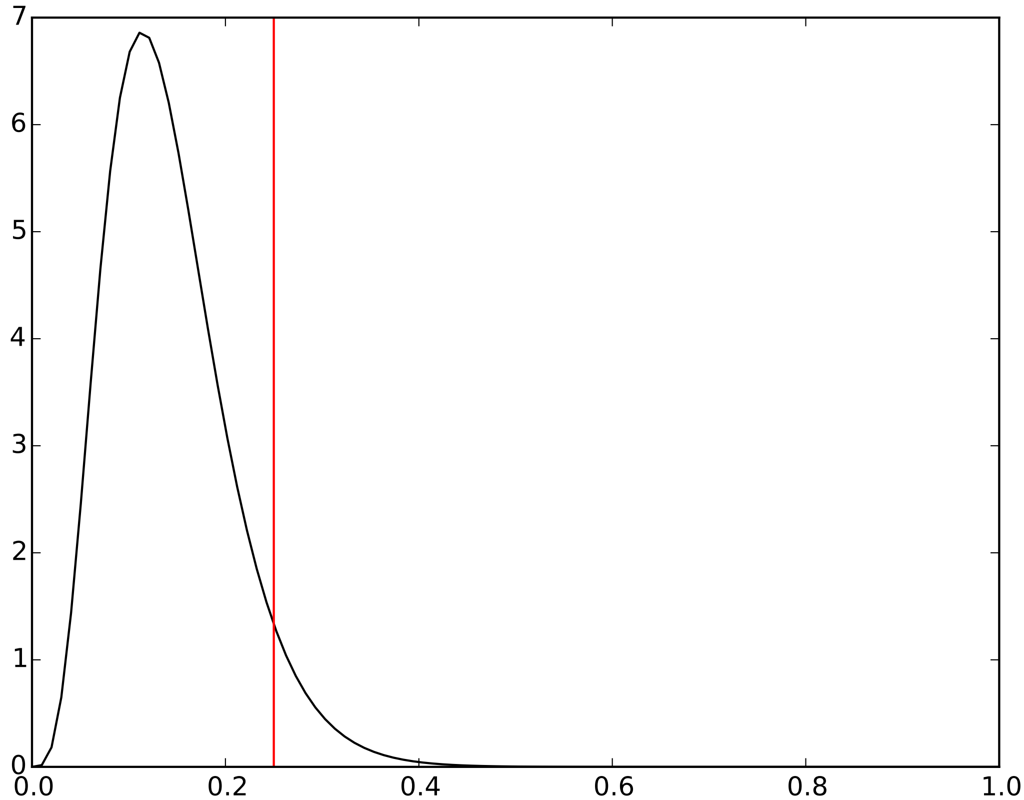
Note that we use the learned node `B` but create a new regressor array for predictions. Plot the predictive distribution of noiseless function values:

```
>>> import bayespy.plot as bpplt
>>> bpplt.pyplot.figure()
<matplotlib.figure.Figure object at 0x...>
>>> bpplt.plot(Fh, x=xh, scale=2)
>>> bpplt.plot(y, x=x, color='r', marker='x', linestyle='None')
>>> bpplt.plot(k*xh+c, x=xh, color='r');
```



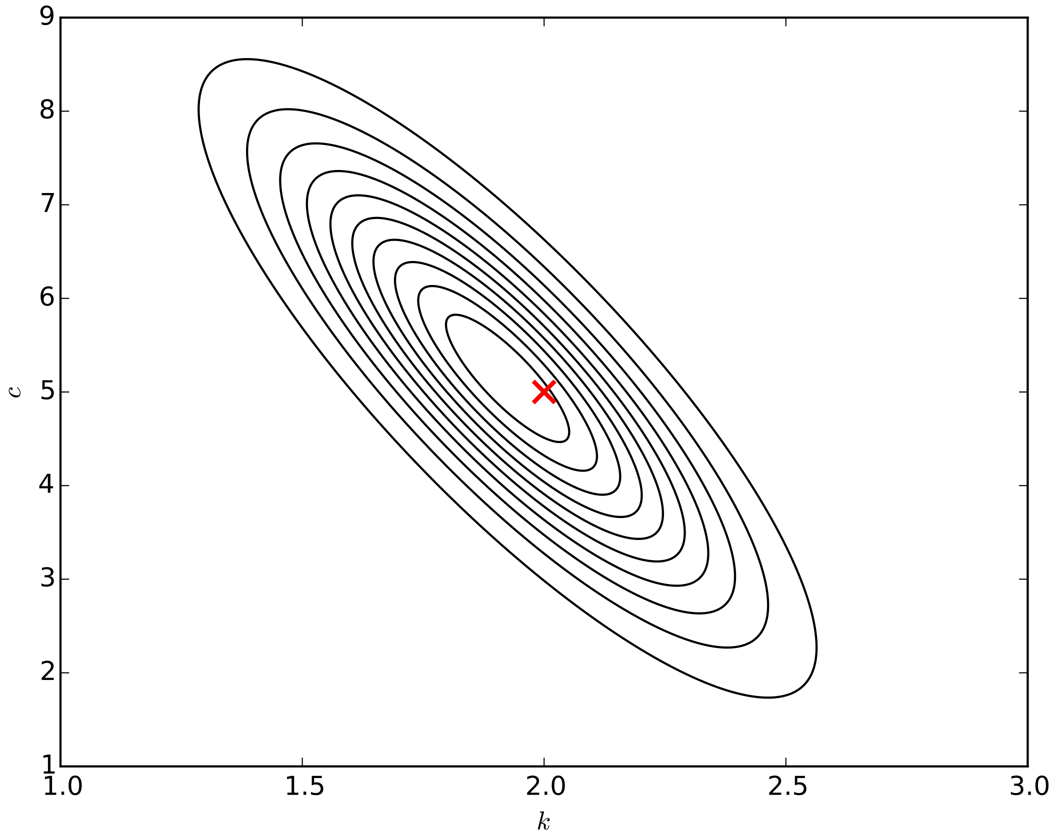
Note that the above plot shows two standard deviation of the posterior of the noiseless function, thus the data points may lie well outside this range. The red line shows the true linear function. Next, plot the distribution of the noise parameter and the true value, $2^{-2} = 0.25$:

```
>>> bpplt.pyplot.figure()
<matplotlib.figure.Figure object at 0x...>
>>> bpplt.pdf(tau, np.linspace(1e-6,1,100), color='k')
[<matplotlib.lines.Line2D object at 0x...>]
>>> bpplt.pyplot.axvline(s**(-2), color='r')
<matplotlib.lines.Line2D object at 0x...>
```



The noise level is captured quite well, although the posterior has more mass on larger noise levels (smaller precision parameter values). Finally, plot the distribution of the regression parameters and mark the true value:

```
>>> bpplt.pyplot.figure()
<matplotlib.figure.Figure object at 0x...>
>>> bpplt.contour(B, np.linspace(1,3,1000), np.linspace(1,9,1000),
...               n=10, colors='k')
<matplotlib.contour.QuadContourSet object at 0x...>
>>> bpplt.plot(c, x=k, color='r', marker='x', linestyle='None',
...            markersize=10, markeredgewidth=2)
>>> bpplt.pyplot.xlabel(r'$k$')
<matplotlib.text.Text object at 0x...>
>>> bpplt.pyplot.ylabel(r'$c$');
<matplotlib.text.Text object at 0x...>
```



In this case, the true parameters are captured well by the posterior distribution.

3.1.5 Improving accuracy

The model can be improved by not factorizing between `B` and `tau` but learning their joint posterior distribution. This requires a slight modification to the model by using `GaussianGammaISO` node:

```
>>> from bayespy.nodes import GaussianGammaISO
>>> B_tau = GaussianGammaISO(np.zeros(2), 1e-6*np.identity(2), 1e-3, 1e-3)
```

This node contains both the regression parameter vector and the noise parameter. We compute the dot product similarly as before:

```
>>> F_tau = SumMultiply('i,i', B_tau, X)
```

However, `Y` is constructed as follows:

```
>>> Y = GaussianARD(F_tau, 1)
```

Because the noise parameter is already in `F_tau` we can give a constant one as the second argument. The total noise parameter for `Y` is the product of the noise parameter in `F_tau` and one. Now, inference is run similarly as before:

```
>>> Y.observe(y)
>>> Q = VB(Y, B_tau)
>>> Q.update(repeat=1000)
```

```
Iteration 1: loglike=-4.678478e+01 (... seconds)
Iteration 2: loglike=-4.678478e+01 (... seconds)
Converged at iteration 2.
```

Note that the method converges immediately. This happens because there is only one unobserved stochastic node so there is no need for iteration and the result is actually the exact true posterior distribution, not an approximation. Currently, the main drawback of using this approach is that BayesPy does not yet contain any plotting utilities for nodes that contain both Gaussian and gamma variables jointly.

3.1.6 Further extensions

The approach discussed in this example can easily be extended to non-linear regression and multivariate regression. For non-linear regression, the inputs are first transformed by some known non-linear functions and then linear regression is applied to this transformed data. For multivariate regression, `X` and `B` are concatenated appropriately: If there are more regressors, add more columns to both `X` and `B`. If there are more output dimensions, add plates to `B`.

3.2 Gaussian mixture model

This example demonstrates the use of Gaussian mixture model for flexible density estimation, clustering or classification.

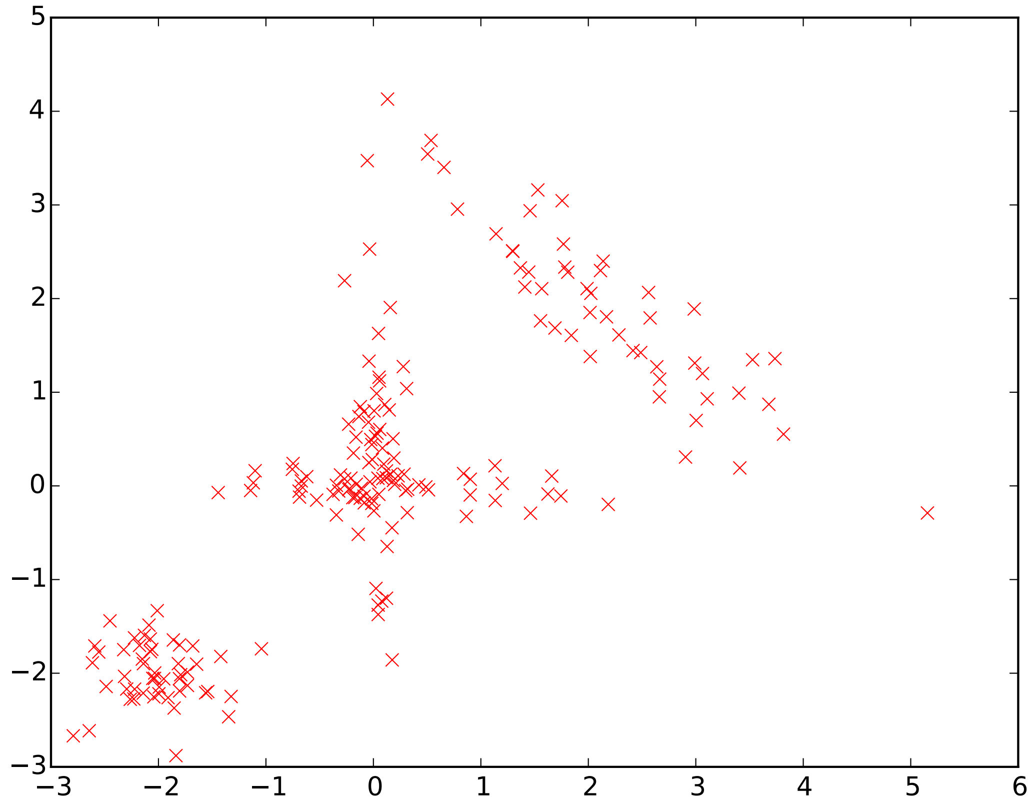
3.2.1 Data

First, let us generate some artificial data for the analysis. The data are two-dimensional vectors from one of the four different Gaussian distributions:

```
>>> import numpy as np
>>> y0 = np.random.multivariate_normal([0, 0], [[2, 0], [0, 0.1]], size=50)
>>> y1 = np.random.multivariate_normal([0, 0], [[0.1, 0], [0, 2]], size=50)
>>> y2 = np.random.multivariate_normal([2, 2], [[2, -1.5], [-1.5, 2]], size=50)
>>> y3 = np.random.multivariate_normal([-2, -2], [[0.5, 0], [0, 0.5]], size=50)
>>> y = np.vstack([y0, y1, y2, y3])
```

Thus, there are 200 data vectors in total. The data looks as follows:

```
>>> import bayespy.plot as bpplt
>>> bpplt.pyplot.plot(y[:,0], y[:,1], 'rx')
[<matplotlib.lines.Line2D object at 0x...>]
```



3.2.2 Model

For clarity, let us denote the number of the data vectors with N

```
>>> N = 200
```

and the dimensionality of the data vectors with D:

```
>>> D = 2
```

We will use a "large enough" number of Gaussian clusters in our model:

```
>>> K = 10
```

Cluster assignments Z and the prior for the cluster assignment probabilities alpha:

```
>>> from bayespy.nodes import Dirichlet, Categorical
>>> alpha = Dirichlet(1e-5*np.ones(K),
...                   name='alpha')
>>> Z = Categorical(alpha,
...                 plates=(N,),
...                 name='z')
```

The mean vectors and the precision matrices of the clusters:

```
>>> from bayespy.nodes import Gaussian, Wishart
>>> mu = Gaussian(np.zeros(D), 1e-5*np.identity(D),
...               plates=(K,),
...               name='mu')
>>> Lambda = Wishart(D, 1e-5*np.identity(D),
...                  plates=(K,),
...                  name='Lambda')
```

If either the mean or precision should be shared between clusters, then that node should not have plates, that is, `plates=()`. The data vectors are from a Gaussian mixture with cluster assignments `Z` and Gaussian component parameters `mu` and `Lambda`:

```
>>> from bayespy.nodes import Mixture
>>> Y = Mixture(Z, Gaussian, mu, Lambda,
...             name='Y')

>>> Z.initialize_from_random()

>>> from bayespy.inference import VB
>>> Q = VB(Y, mu, Lambda, Z, alpha)
```

3.2.3 Inference

Before running the inference algorithm, we provide the data:

```
>>> Y.observe(y)
```

Then, run VB iteration until convergence:

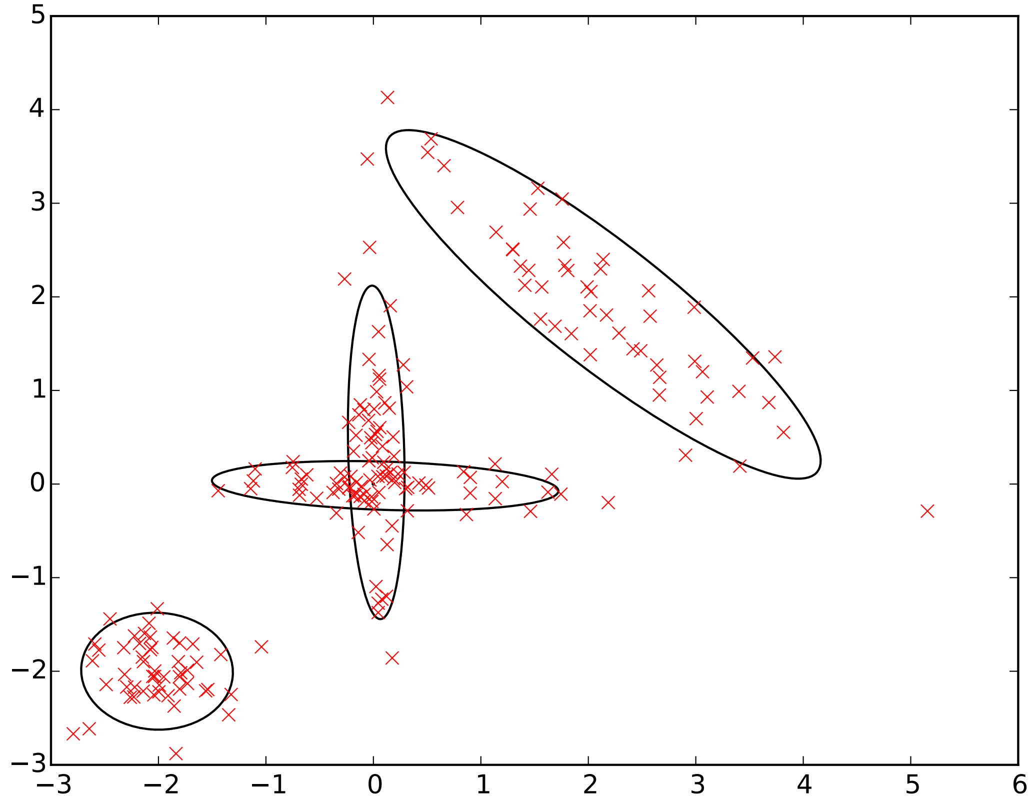
```
>>> Q.update(repeat=1000)
Iteration 1: loglike=-1.401968e+03 (... seconds)
...
Iteration 48: loglike=-1.017893e+03 (... seconds)
Converged at iteration 48.
```

The algorithm converges very quickly. Note that the default update order of the nodes was such that `mu` and `Lambda` were updated before `Z`, which is what we wanted because `Z` was initialized randomly.

3.2.4 Results

For two-dimensional Gaussian mixtures, the mixture components can be plotted using `gaussian_mixture()`:

```
>>> bpplt.gaussian_mixture(Y, scale=2)
```



The function is called with `scale=2` which means that each ellipse shows two standard deviations. From the ten cluster components, the model uses effectively the correct number of clusters (4). These clusters capture the true density accurately.

In addition to clustering and density estimation, this model could also be used for classification by setting the known class assignments as observed.

3.2.5 Advanced next steps

Joint node for mean and precision

The next step for improving the results could be to use `GaussianWishart` node for modelling the mean vectors `mu` and precision matrices `Lambda` jointly without factorization. This should improve the accuracy of the posterior approximation and the speed of the VB estimation. However, the implementation is a bit more complex.

Fast collapsed inference

3.3 Bernoulli mixture model

This example considers data generated from a Bernoulli mixture model. One simple example process could be a questionnaire for election candidates. We observe a set of binary vectors, where each vector represents a candidate in the election and each element in these vectors correspond to a candidate's answer to a yes-or-no question. The goal is to find groups of similar candidates and analyze the answer patterns of these groups.

3.3.1 Data

First, we generate artificial data to analyze. Let us assume that the questionnaire contains ten yes-or-no questions. We assume that there are three groups with similar opinions. These groups could represent parties. These groups have the following answering patterns, which are represented by vectors with probabilities of a candidate answering yes to the questions:

```
>>> p0 = [0.1, 0.9, 0.1, 0.9, 0.1, 0.9, 0.1, 0.9, 0.1, 0.9]
>>> p1 = [0.1, 0.1, 0.1, 0.1, 0.1, 0.9, 0.9, 0.9, 0.9, 0.9]
>>> p2 = [0.9, 0.9, 0.9, 0.9, 0.9, 0.1, 0.1, 0.1, 0.1, 0.1]
```

Thus, the candidates in the first group are likely to answer no to questions 1, 3, 5, 7 and 9, and yes to questions 2, 4, 6, 8, 10. The candidates in the second group are likely to answer yes to the last five questions, whereas the candidates in the third group are likely to answer yes to the first five questions. For convenience, we form a NumPy array of these vectors:

```
>>> import numpy as np
>>> p = np.array([p0, p1, p2])
```

Next, we generate a hundred candidates. First, we randomly select the group for each candidate:

```
>>> from bayespy.utils import random
>>> z = random.categorical([1/3, 1/3, 1/3], size=100)
```

Using the group patterns, we generate yes-or-no answers for the candidates:

```
>>> x = random.bernoulli(p[z])
```

This is our simulated data to be analyzed.

3.3.2 Model

Now, we construct a model for learning the structure in the data. We have a dataset of hundred 10-dimensional binary vectors:

```
>>> N = 100
>>> D = 10
```

We will create a Bernoulli mixture model. We assume that the true number of groups is unknown to us, so we use a large enough number of clusters:

```
>>> K = 10
```

We use the categorical distribution for the group assignments and give the group assignment probabilities an uninformative Dirichlet prior:

```
>>> from bayespy.nodes import Categorical, Dirichlet
>>> R = Dirichlet(K*[1e-5],
...               name='R')
>>> Z = Categorical(R,
...                 plates=(N,1),
...                 name='Z')
```

Each group has a probability of a yes answer for each question. These probabilities are given beta priors:

```
>>> from bayespy.nodes import Beta
>>> P = Beta([0.5, 0.5],
...          plates=(D,K),
...          name='P')
```

The answers of the candidates are modelled with the Bernoulli distribution:

```
>>> from bayespy.nodes import Mixture, Bernoulli
>>> X = Mixture(Z, Bernoulli, P)
```

Here, `Z` defines the group assignments and `P` the answering probability patterns for each group. Note how the plates of the nodes are matched: `Z` has plates `(N,1)` and `P` has plates `(D,K)`, but in the mixture node the last plate axis of `P` is discarded and thus the node broadcasts plates `(N,1)` and `(D,)` resulting in plates `(N,D)` for `X`.

3.3.3 Inference

In order to infer the variables in our model, we construct a variational Bayesian inference engine:

```
>>> from bayespy.inference import VB
>>> Q = VB(Z, R, X, P)
```

This also gives the default update order of the nodes. In order to find different groups, they must be initialized differently, thus we use random initialization for the group probability patterns:

```
>>> P.initialize_from_random()
```

We provide our simulated data:

```
>>> X.observe(x)
```

Now, we can run inference:

```
>>> Q.update(repeat=1000)
Iteration 1: loglike=-6.872145e+02 (... seconds)
...
Iteration 17: loglike=-5.236921e+02 (... seconds)
Converged at iteration 17.
```

The algorithm converges in 17 iterations.

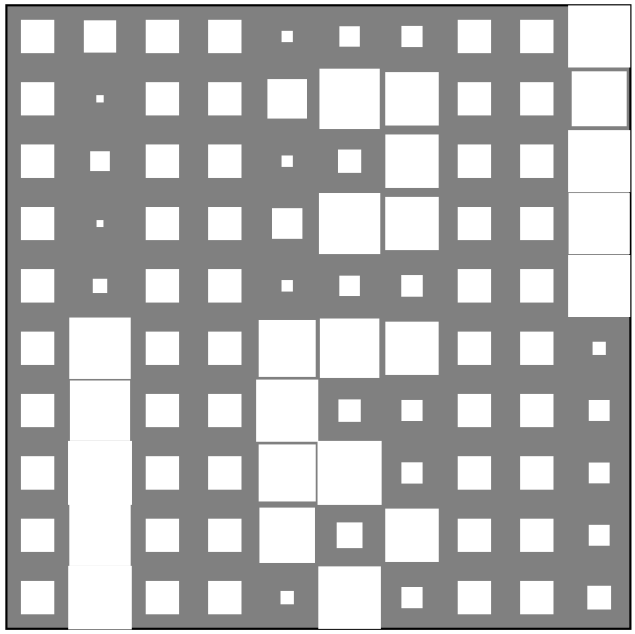
3.3.4 Results

Now we can examine the approximate posterior distribution. First, let us plot the group assignment probabilities:

```
>>> import bayespy.plot as bpplt
>>> bpplt.hinton(R)
```

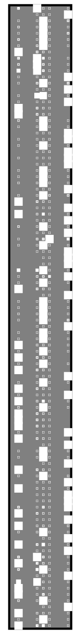
This plot shows that there are three dominant groups, which is equal to the true number of groups used to generate the data. However, there are still two smaller groups as the data does not give enough evidence to prune them out. The yes-or-no answer probability patterns for the groups can be plotted as:

```
>>> bpplt.hinton(P)
```



The three dominant groups have found the true patterns accurately. The patterns of the two minor groups some kind of mixtures of the three groups and they exist because the generated data happened to contain a few samples giving evidence for these groups. Finally, we can plot the group assignment probabilities for the candidates:

```
>>> bpplt.hinton(Z)
```



This plot shows the clustering of the candidates. It is possible to use `HintonPlotter` to enable monitoring during the VB iteration by providing `plotter=HintonPlotter()` for `Z`, `P` and `R` when creating the nodes.

3.4 Hidden Markov model

In this example, we will demonstrate the use of hidden Markov model in the case of known and unknown parameters. We will also use two different emission distributions to demonstrate the flexibility of the model construction.

3.4.1 Known parameters

This example follows the one presented in Wikipedia.

Model

Each day, the state of the weather is either 'rainy' or 'sunny'. The weather follows a first-order discrete Markov process. It has the following initial state probabilities

```
>>> a0 = [0.6, 0.4] # p(rainy)=0.6, p(sunny)=0.4
```

and state transition probabilities:

```
>>> A = [[0.7, 0.3], # p(rainy->rainy)=0.7, p(rainy->sunny)=0.3
...      [0.4, 0.6]] # p(sunny->rainy)=0.4, p(sunny->sunny)=0.6
```

We will be observing one hundred samples:

```
>>> N = 100
```

The discrete first-order Markov chain is constructed as:

```
>>> from bayespy.nodes import CategoricalMarkovChain
>>> Z = CategoricalMarkovChain(a0, A, states=N)
```

However, instead of observing this process directly, we observe whether Bob is 'walking', 'shopping' or 'cleaning'. The probability of each activity depends on the current weather as follows:

```
>>> P = [[0.1, 0.4, 0.5],
...      [0.6, 0.3, 0.1]]
```

where the first row contains activity probabilities on a rainy weather and the second row contains activity probabilities on a sunny weather. Using these emission probabilities, the observed process is constructed as:

```
>>> from bayespy.nodes import Categorical, Mixture
>>> Y = Mixture(Z, Categorical, P)
```

Data

In order to test our method, we'll generate artificial data from the model itself. First, draw realization of the weather process:

```
>>> weather = Z.random()
```

Then, using this weather, draw realizations of the activities:

```
>>> activity = Mixture(weather, Categorical, P).random()
```

Inference

Now, using this data, we set our variable Y to be observed:

```
>>> Y.observe(activity)
```

In order to run inference, we construct variational Bayesian inference engine:

```
>>> from bayespy.inference import VB
>>> Q = VB(Y, Z)
```

Note that we need to give all random variables to `VB`. In this case, the only random variables were `Y` and `Z`. Next we run the inference, that is, compute our posterior distribution:

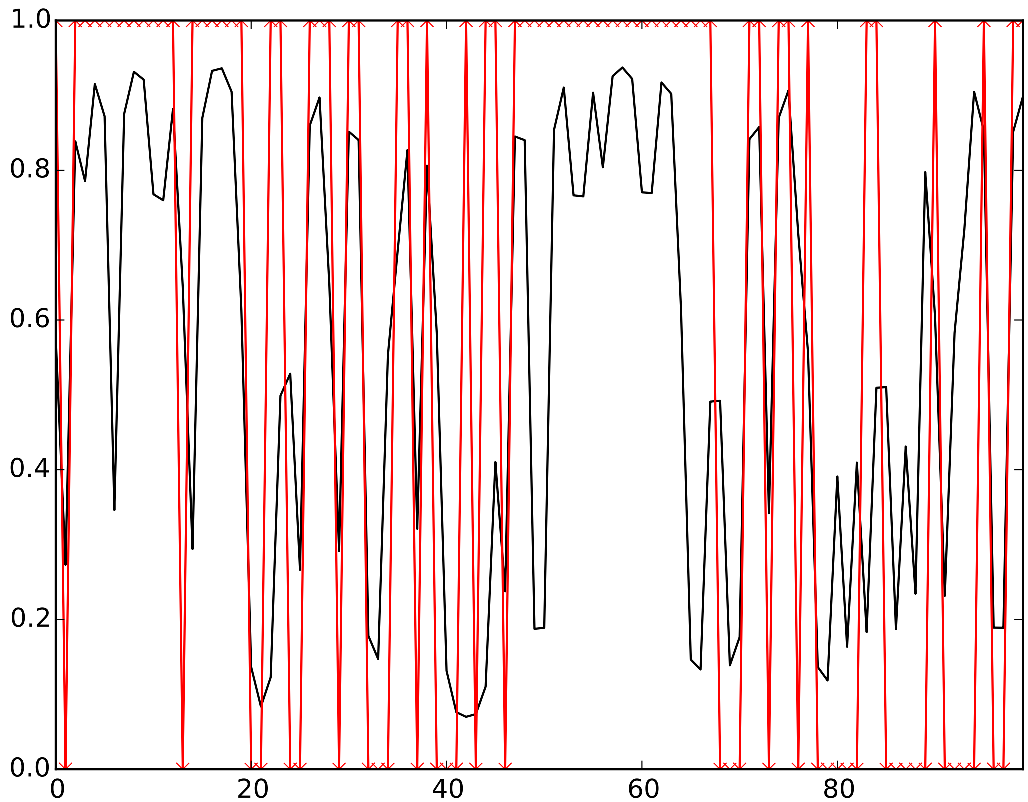
```
>>> Q.update()
Iteration 1: loglike=-1.095883e+02 (... seconds)
```

In this case, because there is only one unobserved random variable, we recover the exact posterior distribution and there is no need to iterate more than one step.

Results

One way to plot a 2-class categorical timeseries is to use the basic `plot()` function:

```
>>> import bayespy.plot as bpplt
>>> bpplt.plot(Z)
>>> bpplt.plot(1-weather, color='r', marker='x')
```



The black line shows the posterior probability of rain and the red line and crosses show the true state. Clearly, the method is not able to infer the weather very accurately in this case because the activies do not give that much information about the weather.

3.4.2 Unknown parameters

In this example, we consider unknown parameters for the Markov process and different emission distribution.

Data

We generate data from three 2-dimensional Gaussian distributions with different mean vectors and common standard deviation:

```
>>> import numpy as np
>>> mu = np.array([ [0,0], [3,4], [6,0] ])
>>> std = 2.0
```

Thus, the number of clusters is three:

```
>>> K = 3
```

And the number of samples is 200:

```
>>> N = 200
```

Each initial state is equally probable:

```
>>> p0 = np.ones(K) / K
```

State transition matrix is such that with probability 0.9 the process stays in the same state. The probability to move one of the other two states is 0.05 for both of those states.

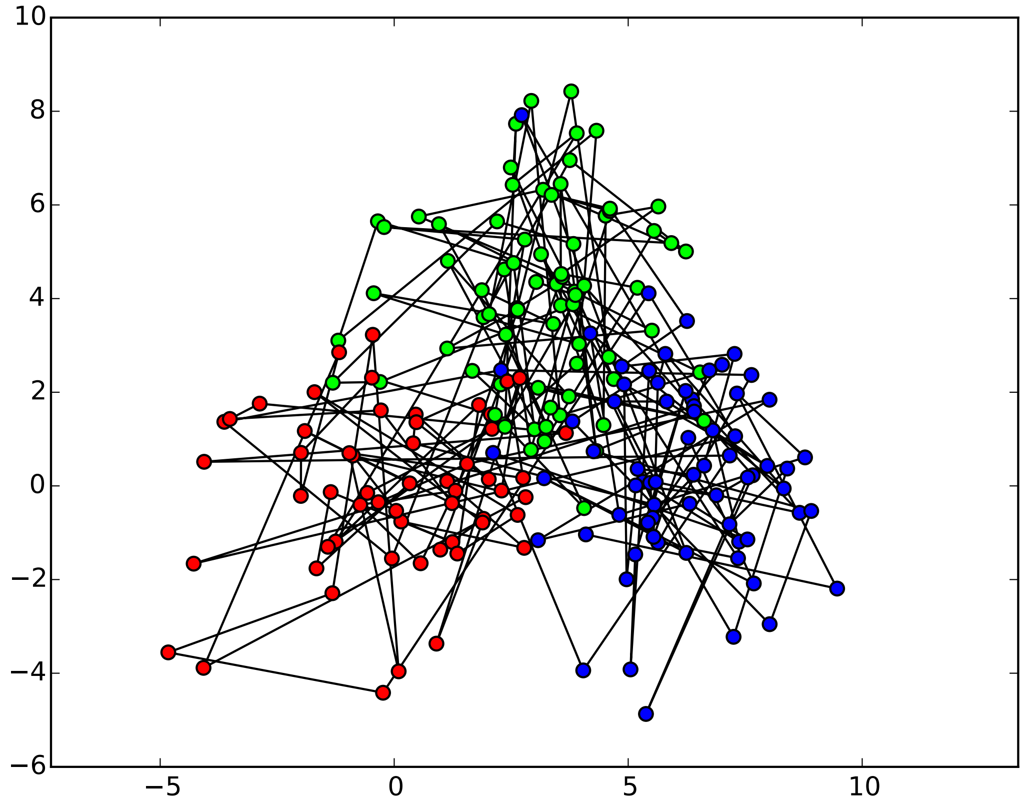
```
>>> q = 0.9
>>> r = (1-q) / (K-1)
>>> P = q*np.identity(K) + r*(np.ones((3,3))-np.identity(3))
```

Simulate the data:

```
>>> y = np.zeros((N,2))
>>> z = np.zeros(N)
>>> state = np.random.choice(K, p=p0)
>>> for n in range(N):
...     z[n] = state
...     y[n,:] = std*np.random.randn(2) + mu[state]
...     state = np.random.choice(K, p=P[state])
```

Then, let us visualize the data:

```
>>> bpplt.pyplot.figure()
<matplotlib.figure.Figure object at 0x...>
>>> bpplt.pyplot.axis('equal')
(...)
>>> colors = [ [[1,0,0], [0,1,0], [0,0,1]][int(state)] for state in z ]
>>> bpplt.pyplot.plot(y[:,0], y[:,1], 'k-', zorder=-10)
[<matplotlib.lines.Line2D object at 0x...>]
>>> bpplt.pyplot.scatter(y[:,0], y[:,1], c=colors, s=40)
<matplotlib.collections.PathCollection object at 0x...>
```



Consecutive states are connected by a solid black line and the dot color shows the true class.

Model

Now, assume that we do not know the parameters of the process (initial state probability and state transition probabilities). We give these parameters quite non-informative priors, but it is possible to provide more informative priors if such information is available:

```
>>> from bayespy.nodes import Dirichlet
>>> a0 = Dirichlet(1e-3*np.ones(K))
>>> A = Dirichlet(1e-3*np.ones((K,K)))
```

The discrete Markov chain is constructed as:

```
>>> Z = CategoricalMarkovChain(a0, A, states=N)
```

Now, instead of using categorical emission distribution as before, we'll use Gaussian distribution. For simplicity, we use the true parameters of the Gaussian distributions instead of giving priors and estimating them. The known standard deviation can be converted to a precision matrix as:

```
>>> Lambda = std**(-2) * np.identity(2)
```

Thus, the observed process is a Gaussian mixture with cluster assignments from the hidden Markov process `Z`:

```
>>> from bayespy.nodes import Gaussian
>>> Y = Mixture(Z, Gaussian, mu, Lambda)
```

Note that `Lambda` does not have cluster plate axis because it is shared between the clusters.

Inference

Let us use the simulated data:

```
>>> Y.observe(y)
```

Because `VB` takes all the random variables, we need to provide `A` and `a0` also:

```
>>> Q = VB(Y, Z, A, a0)
```

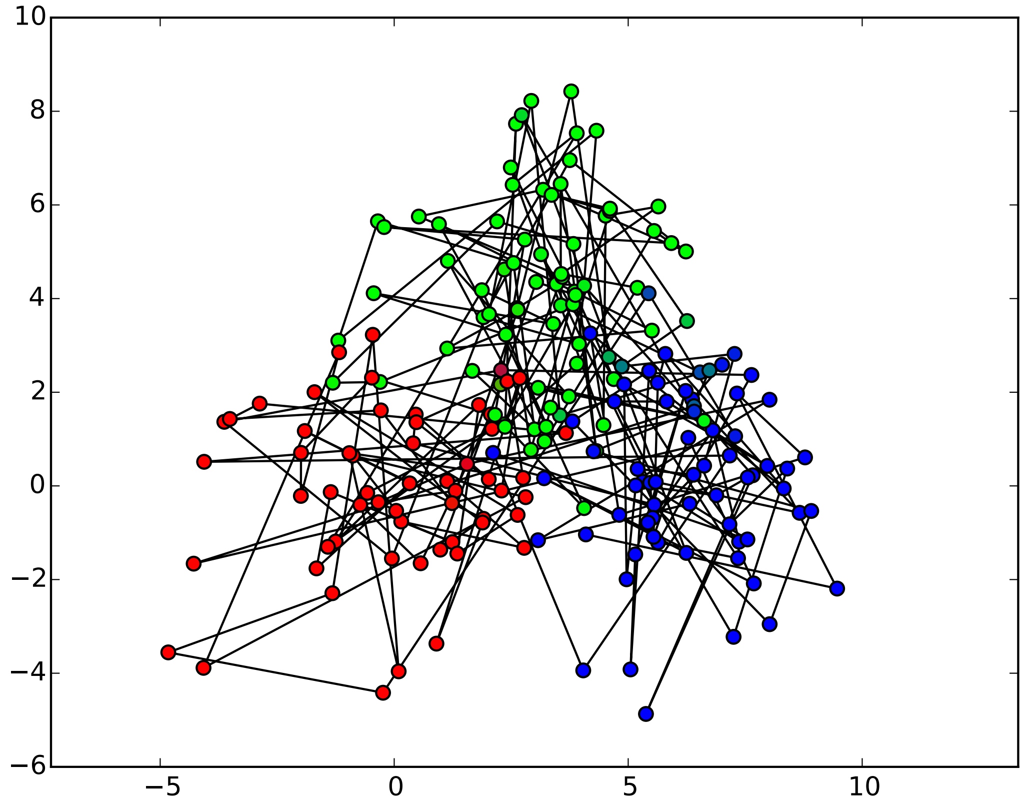
Then, run VB iteration until convergence:

```
>>> Q.update(repeat=1000)
Iteration 1: loglike=-9.963054e+02 (... seconds)
...
Iteration 8: loglike=-9.235053e+02 (... seconds)
Converged at iteration 8.
```

Results

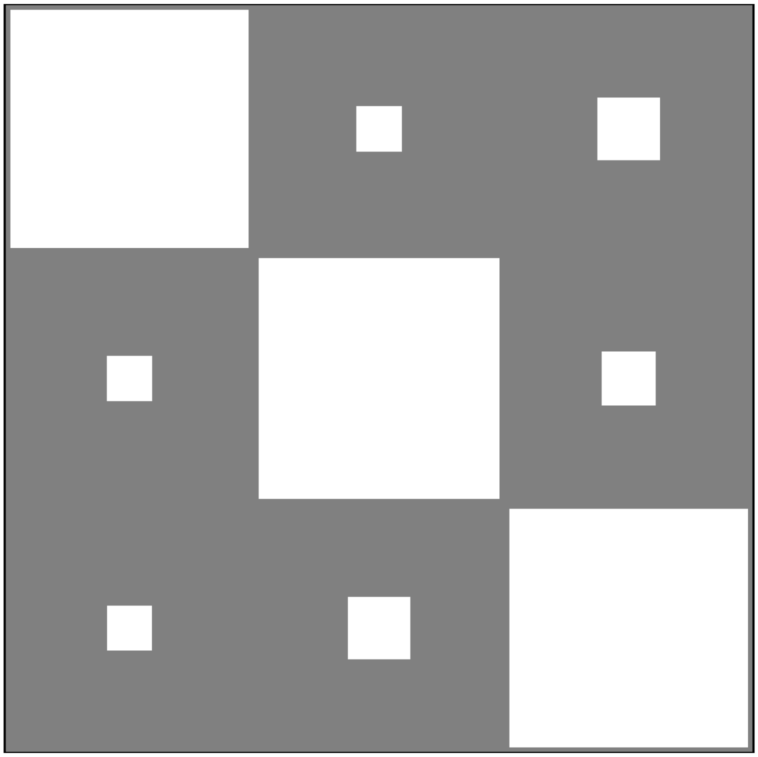
Plot the classification of the data similarly as the data:

```
>>> bpplt.pyplot.figure()
<matplotlib.figure.Figure object at 0x...>
>>> bpplt.pyplot.axis('equal')
(...)
>>> colors = Y.parents[0].get_moments()[0]
>>> bpplt.pyplot.plot(y[:,0], y[:,1], 'k-', zorder=-10)
[<matplotlib.lines.Line2D object at 0x...>]
>>> bpplt.pyplot.scatter(y[:,0], y[:,1], c=colors, s=40)
<matplotlib.collections.PathCollection object at 0x...>
```



The data has been classified quite correctly. Even samples that are more in the region of another cluster are classified correctly if the previous and next sample provide enough evidence for the correct class. We can also plot the state transition matrix:

```
>>> bpplt.hinton(A)
```



Clearly, the learned state transition matrix is close to the true matrix. The models described above could also be used for classification by providing the known class assignments as observed data to `Z` and the unknown class assignments as missing data.

3.5 Principal component analysis

This example uses a simple principal component analysis to find a two-dimensional latent subspace in a higher dimensional dataset.

3.5.1 Data

Let us create a Gaussian dataset with latent space dimensionality two and some observation noise:

```
>>> M = 20
>>> N = 100

>>> import numpy as np
>>> x = np.random.randn(N, 2)
>>> w = np.random.randn(M, 2)
>>> f = np.einsum('ik,jk->ij', w, x)
>>> y = f + 0.1*np.random.randn(M, N)
```

3.5.2 Model

We will use 10-dimensional latent space in our model and let it learn the true dimensionality:

```
>>> D = 10
```

Import relevant nodes:

```
>>> from bayespy.nodes import GaussianARD, Gamma, SumMultiply
```

The latent states:

```
>>> X = GaussianARD(0, 1, plates=(1,N), shape=(D,))
```

The loading matrix with automatic relevance determination (ARD) prior:

```
>>> alpha = Gamma(1e-5, 1e-5, plates=(D,))
>>> C = GaussianARD(0, alpha, plates=(M,1), shape=(D,))
```

Compute the dot product of the latent states and the loading matrix:

```
>>> F = SumMultiply('d,d->', X, C)
```

The observation noise:

```
>>> tau = Gamma(1e-5, 1e-5)
```

The observed variable:

```
>>> Y = GaussianARD(F, tau)
```

3.5.3 Inference

Observe the data:

```
>>> Y.observe(y)
```

We do not have missing data now, but they could be easily handled with `mask` keyword argument. Construct variational Bayesian (VB) inference engine:

```
>>> from bayespy.inference import VB
>>> Q = VB(Y, X, C, alpha, tau)
```

Initialize the latent subspace randomly, otherwise both `X` and `C` would converge to zero:

```
>>> C.initialize_from_random()
```

Now we could use `VB.update()` to run the inference. However, let us first create a parameter expansion to speed up the inference. The expansion is based on rotating the latent subspace optimally. This is optional but will usually improve the speed of the inference significantly, especially in high-dimensional problems:

```
>>> from bayespy.inference.vmp.transformations import RotateGaussianARD
>>> rot_X = RotateGaussianARD(X)
>>> rot_C = RotateGaussianARD(C, alpha)
```

By giving `alpha` for `rot_C`, the rotation will also optimize `alpha` jointly with `C`. Now that we have defined the rotations for our variables, we need to construct an optimizer:

```
>>> from bayespy.inference.vmp.transformations import RotationOptimizer
>>> R = RotationOptimizer(rot_X, rot_C, D)
```

In order to use the rotations automatically, we need to set it as a callback function:

```
>>> Q.set_callback(R.rotate)
```

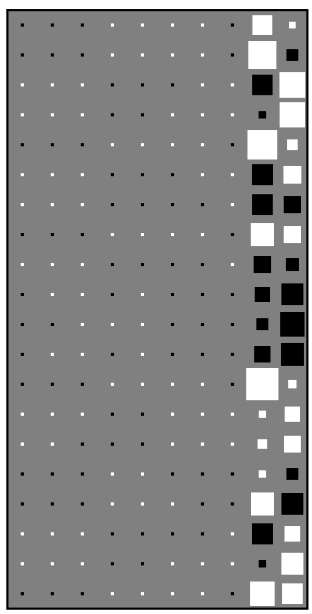
For more information about the rotation parameter expansion, see [7] and [6]. Now we can run the actual inference until convergence:

```
>>> Q.update(repeat=1000)
Iteration 1: loglike=-2.339710e+03 (... seconds)
...
Iteration 23: loglike=6.500706e+02 (... seconds)
Converged at iteration 23.
```

3.5.4 Results

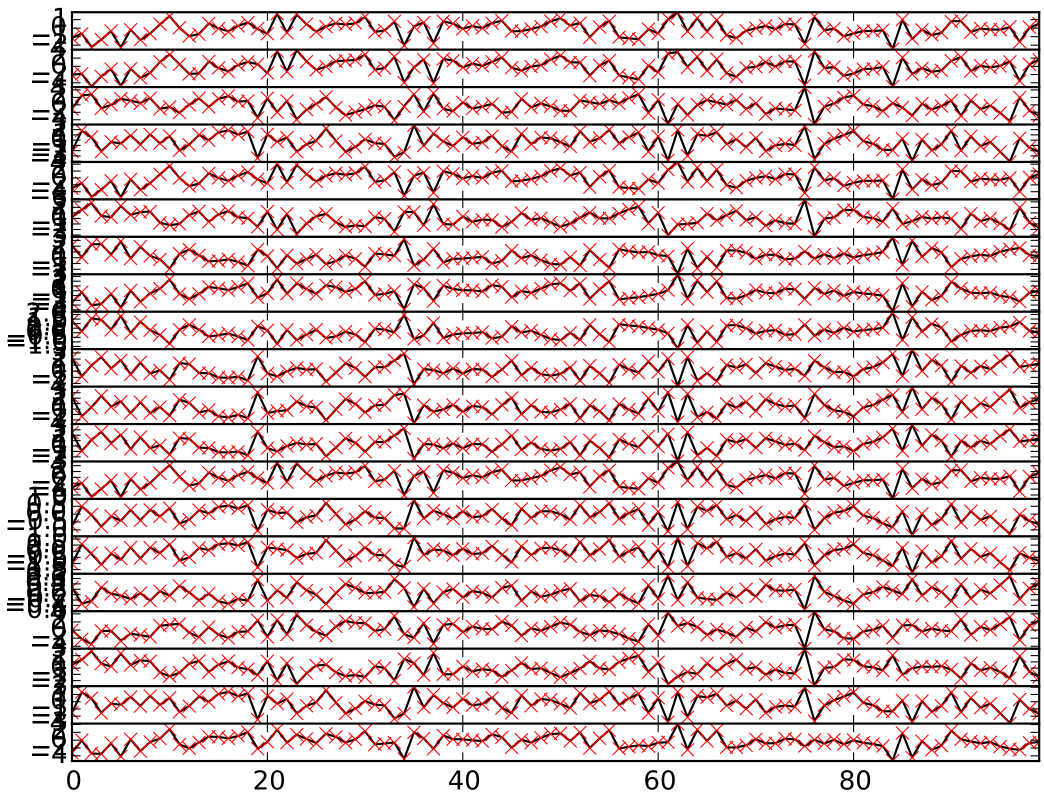
The results can be visualized, for instance, by plotting the Hinton diagram of the loading matrix:

```
>>> import bayespy.plot as bpplt
>>> bpplt.pyplot.figure()
<matplotlib.figure.Figure object at 0x...>
>>> bpplt.hinton(C)
```



The method has been able to prune out unnecessary latent dimensions and keep two components, which is the true number of components.

```
>>> bpplt.pyplot.figure()
<matplotlib.figure.Figure object at 0x...>
>>> bpplt.plot(F)
>>> bpplt.plot(f, color='r', marker='x', linestyle='None')
```



The reconstruction of the noiseless function values are practically perfect in this simple example. Larger noise variance, more latent space dimensions and missing values would make this problem more difficult. The model construction could also be improved by having, for instance, `C` and `tau` in the same node without factorizing between them in the posterior approximation. This can be achieved by using `GaussianGammaISO` node.

3.6 Linear state-space model

3.6.1 Model

In linear state-space models a sequence of M-dimensional observations $\mathbf{Y} = (\mathbf{y}_1, \ldots, \mathbf{y}_N)$ is assumed to be generated from latent D-dimensional states $\mathbf{X} = (\mathbf{x}_1, \ldots, \mathbf{x}_N)$ which follow a first-order Markov process:

$$\mathbf{x}_n = \mathbf{A}\mathbf{x}_{n-1} + \text{noise}\,,$$
$$\mathbf{y}_n = \mathbf{C}\mathbf{x}_n + \text{noise}\,,$$

where the noise is Gaussian, $\mathbf{A}$ is the $D \times D$ state dynamics matrix and $\mathbf{C}$ is the $M \times D$ loading matrix. Usually, the latent space dimensionality D is assumed to be much smaller than the observation space dimensionality M in order to model the dependencies of high-dimensional observations efficiently.

In order to construct the model in BayesPy, first import relevant nodes:

```
>>> from bayespy.nodes import GaussianARD, GaussianMarkovChain, Gamma, Dot
```

The data vectors will be 30-dimensional:

```
>>> M = 30
```

There will be 400 data vectors:

```
>>> N = 400
```

Let us use 10-dimensional latent space:

```
>>> D = 10
```

The state dynamics matrix $\mathbf{A}$ has ARD prior:

```
>>> alpha = Gamma(1e-5,
...               1e-5,
...               plates=(D,),
...               name='alpha')
>>> A = GaussianARD(0,
...                 alpha,
...                 shape=(D,),
...                 plates=(D,),
...                 name='A')
```

Note that $\mathbf{A}$ is a $D \times D$-dimensional matrix. However, in BayesPy it is modelled as a collection (`plates=(D,)`) of D-dimensional vectors (`shape=(D,)`) because this is how the variables factorize in the posterior approximation of the state dynamics matrix in `GaussianMarkovChain`. The latent states are constructed as

```
>>> X = GaussianMarkovChain(np.zeros(D),
...                         1e-3*np.identity(D),
...                         A,
...                         np.ones(D),
...                         n=N,
...                         name='X')
```

where the first two arguments are the mean and precision matrix of the initial state, the third argument is the state dynamics matrix and the fourth argument is the diagonal elements of the precision matrix of the innovation noise. The node also needs the length of the chain given as the keyword argument `n=N`. Thus, the shape of this node is `(N,D)`.

The linear mapping from the latent space to the observation space is modelled with the loading matrix which has ARD prior:

```
>>> gamma = Gamma(1e-5,
...               1e-5,
...               plates=(D,),
...               name='gamma')
>>> C = GaussianARD(0,
...                 gamma,
...                 shape=(D,),
...                 plates=(M,1),
...                 name='C')
```

Note that the plates for `C` are `(M,1)`, thus the full shape of the node is `(M,1,D)`. The unit plate axis is added so that `C` broadcasts with `X` when computing the dot product:

```
>>> F = Dot(C,
...         X,
...         name='F')
```

This dot product is computed over the D-dimensional latent space, thus the result is a $M \times N$-dimensional matrix which is now represented with plates `(M,N)` in BayesPy:

```
>>> F.plates
(30, 400)
```

We also need to use random initialization either for `C` or `X` in order to find non-zero latent space because by default both `C` and `X` are initialized to zero because of their prior distributions. We use random initialization for `C` and then we must update `X` the first time before updating `C`:

```
>>> C.initialize_from_random()
```

The precision of the observation noise is given gamma prior:

```
>>> tau = Gamma(1e-5,
...             1e-5,
...             name='tau')
```

The observations are noisy versions of the dot products:

```
>>> Y = GaussianARD(F,
...                 tau,
...                 name='Y')
```

The variational Bayesian inference engine is then construced as:

```
>>> from bayespy.inference import VB
>>> Q = VB(X, C, gamma, A, alpha, tau, Y)
```

Note that `X` is given before `C`, thus `X` is updated before `C` by default.

3.6.2 Data

Now, let us generate some toy data for our model. Our true latent space is four dimensional with two noisy oscillator components, one random walk component and one white noise component.

```
>>> w = 0.3
>>> a = np.array([[np.cos(w), -np.sin(w), 0, 0],
...               [np.sin(w), np.cos(w),  0, 0],
...               [0,         0,          1, 0],
...               [0,         0,          0, 0]])
```

The true linear mapping is just random:

```
>>> c = np.random.randn(M,4)
```

Then, generate the latent states and the observations using the model equations:

```
>>> x = np.empty((N,4))
>>> f = np.empty((M,N))
>>> y = np.empty((M,N))
>>> x[0] = 10*np.random.randn(4)
>>> f[:,0] = np.dot(c,x[0])
>>> y[:,0] = f[:,0] + 3*np.random.randn(M)
>>> for n in range(N-1):
...     x[n+1] = np.dot(a,x[n]) + [1,1,10,10]*np.random.randn(4)
...     f[:,n+1] = np.dot(c,x[n+1])
...     y[:,n+1] = f[:,n+1] + 3*np.random.randn(M)
```

We want to simulate missing values, thus we create a mask which randomly removes 80% of the data:

```
>>> from bayespy.utils import random
>>> mask = random.mask(M, N, p=0.2)
>>> Y.observe(y, mask=mask)
```

3.6.3 Inference

As we did not define plotters for our nodes when creating the model, it is done now for some of the nodes:

```
>>> import bayespy.plot as bpplt
>>> X.set_plotter(bpplt.FunctionPlotter(center=True, axis=-2))
>>> A.set_plotter(bpplt.HintonPlotter())
>>> C.set_plotter(bpplt.HintonPlotter())
>>> tau.set_plotter(bpplt.PDFPlotter(np.linspace(0.02, 0.5, num=1000)))
```

This enables plotting of the approximate posterior distributions during VB learning. The inference engine can be run using `VB.update()` method:

```
>>> Q.update(repeat=10)
Iteration 1: loglike=-1.439704e+05 (... seconds)
...
Iteration 10: loglike=-1.051441e+04 (... seconds)
```

The iteration progresses a bit slowly, thus we'll consider parameter expansion to speed it up.

Parameter expansion

Section *Parameter expansion* discusses parameter expansion for state-space models to speed up inference. It is based on a rotating the latent space such that the posterior in the observation space is not affected:

$$\mathbf{y}_n = \mathbf{C}\mathbf{x}_n = (\mathbf{C}\mathbf{R}^{-1})(\mathbf{R}\mathbf{x}_n)\,.$$

Thus, the transformation is $\mathbf{C} \rightarrow \mathbf{C}\mathbf{R}^{-1}$ and $\mathbf{X} \rightarrow \mathbf{R}\mathbf{X}$. In order to keep the dynamics of the latent states unaffected by the transformation, the state dynamics matrix $\mathbf{A}$ must be transformed accordingly:

$$\mathbf{R}\mathbf{x}_n = \mathbf{R}\mathbf{A}\mathbf{R}^{-1}\mathbf{R}\mathbf{x}_{n-1}\,,$$

resulting in a transformation $\mathbf{A} \rightarrow \mathbf{R}\mathbf{A}\mathbf{R}^{-1}$. For more details, refer to [6] and [7]. In BayesPy, the transformations are available in `bayespy.inference.vmp.transformations`:

```
>>> from bayespy.inference.vmp import transformations
```

The rotation of the loading matrix along with the ARD parameters is defined as:

```
>>> rotC = transformations.RotateGaussianARD(C, gamma)
```

For rotating `X`, we first need to define the rotation of the state dynamics matrix:

```
>>> rotA = transformations.RotateGaussianARD(A, alpha)
```

Now we can define the rotation of the latent states:

```
>>> rotX = transformations.RotateGaussianMarkovChain(X, rotA)
```

The optimal rotation for all these variables is found using rotation optimizer:

```
>>> R = transformations.RotationOptimizer(rotX, rotC, D)
```

Set the parameter expansion to be applied after each iteration:

```
>>> Q.callback = R.rotate
```

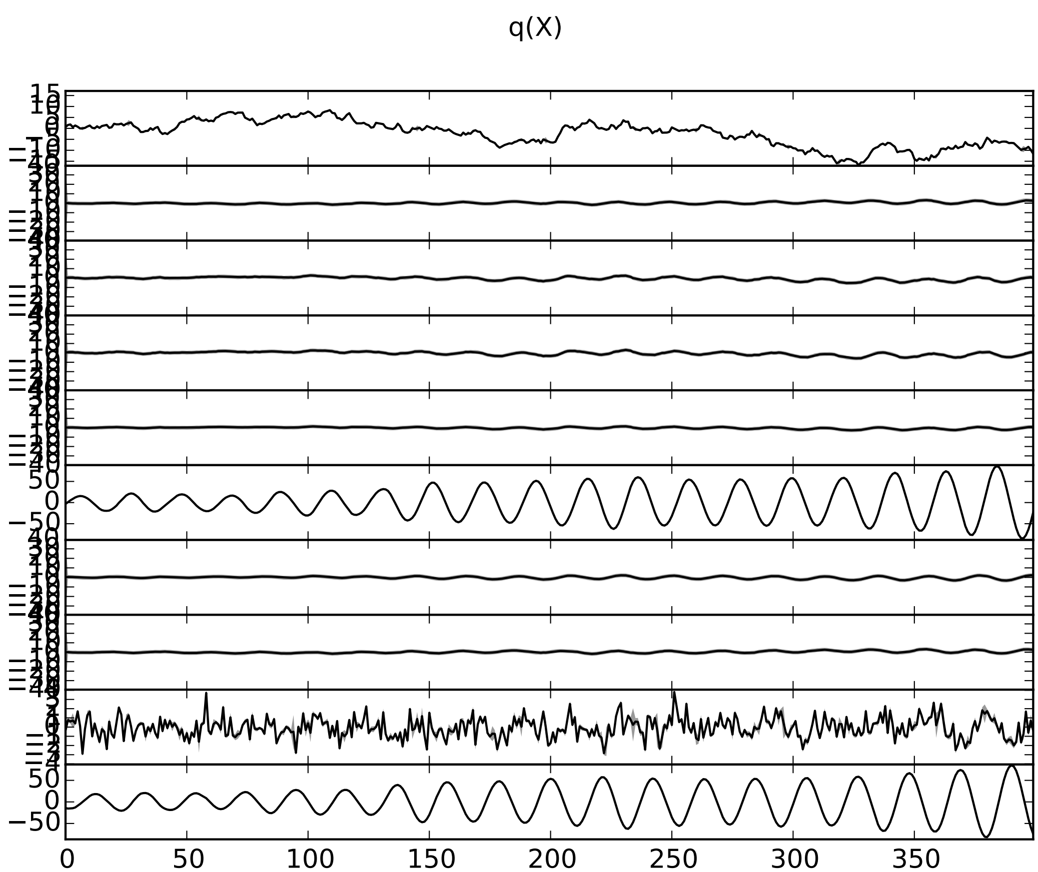
Now, run iterations until convergence:

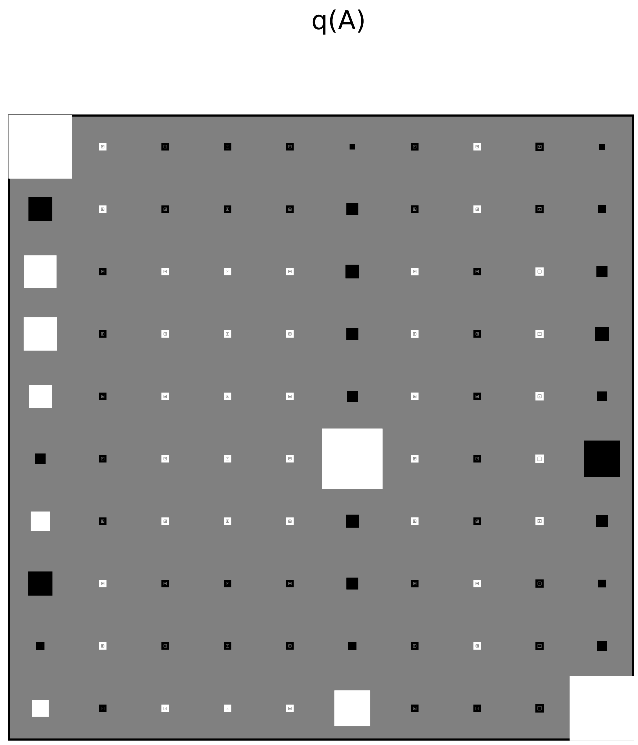
```
>>> Q.update(repeat=1000)
Iteration 11: loglike=-1.010806e+04 (... seconds)
...
Iteration 59: loglike=-8.906325e+03 (... seconds)
Converged at iteration 59.
```

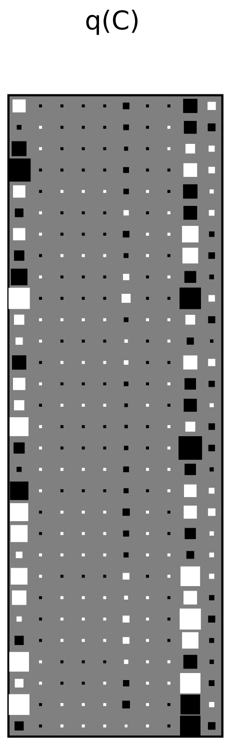
3.6.4 Results

Because we have set the plotters, we can plot those nodes as:

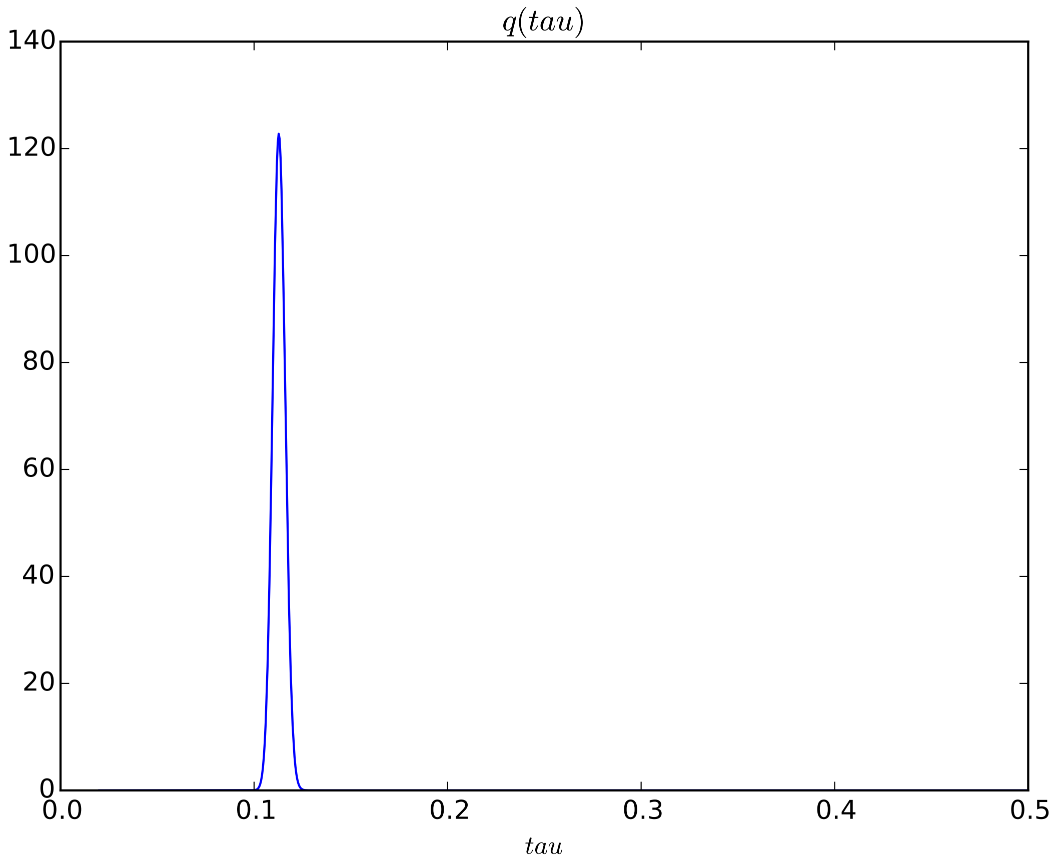
```
>>> Q.plot(X, A, C, tau)
```




q(A)

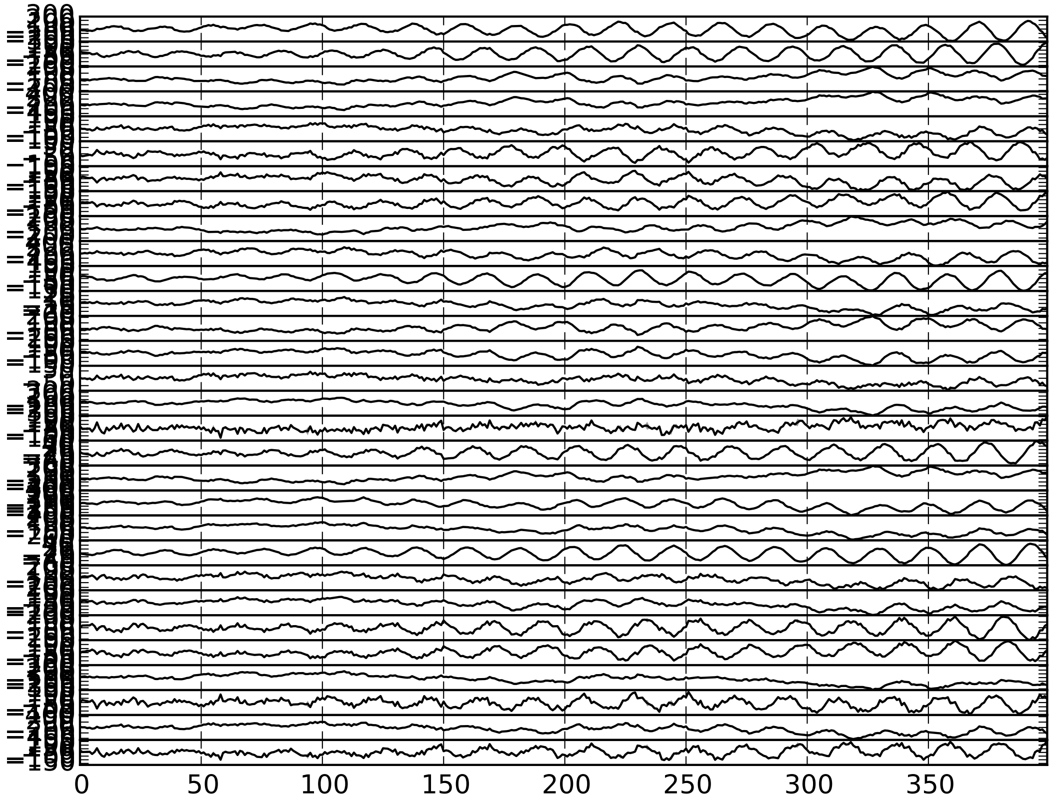

q(C)

q(tau)



There are clearly four effective components in `X`: random walk (component number 1), random oscillation (7 and 10), and white noise (9). These dynamics are also visible in the state dynamics matrix Hinton diagram. Note that the white noise component does not have any dynamics. Also `C` shows only four effective components. The posterior of `tau` captures the true value $3^{-2} \approx 0.111$ accurately. We can also plot predictions in the observation space:

```
>>> bpplt.plot(F, center=True)
```



We can also measure the performance numerically by computing root-mean-square error (RMSE) of the missing values:

```
>>> from bayespy.utils import misc
>>> misc.rmse(y[~mask], F.get_moments()[0][~mask])
5.18...
```

This is relatively close to the standard deviation of the noise (3), so the predictions are quite good considering that only 20% of the data was used.

CHAPTER FOUR

DEVELOPER GUIDE

This chapter provides basic information for developers about contributing, the theoretical background and the core structure. It is assumed that the reader has read and is familiar with *User guide*.

4.1 Workflow

The main forum for BayesPy development is GitHub. Bugs and other issues can be reported at https://github.com/bayespy/bayespy/issues. Contributions to the code and documentation are welcome and should be given as pull requests at https://github.com/bayespy/bayespy/pulls. In order to create pull requests, it is recommended to fork the git repository, make local changes and submit these changes as a pull request. The style guide for writing docstrings follows the style guide of NumPy, available at https://github.com/numpy/numpy/blob/master/doc/HOWTO_DOCUMENT.rst.txt. Detailed instructions on development workflow can be read from NumPy guide, available at http://docs.scipy.org/doc/numpy/dev/gitwash/development_workflow.html. BayesPy uses the following acronyms to start the commit message:

- API: an (incompatible) API change
- BLD: change related to building numpy
- BUG: bug fix
- DEMO: modification in demo code
- DEP: deprecate something, or remove a deprecated object
- DEV: development tool or utility
- DOC: documentation
- ENH: enhancement
- MAINT: maintenance commit (refactoring, typos, etc.)
- REV: revert an earlier commit
- STY: style fix (whitespace, PEP8)
- TST: addition or modification of tests
- REL: related to releasing numpy

4.2 Variational message passing

This section briefly describes the variational message passing (VMP) framework, which is currently the only implemented inference engine in BayesPy. The variational Bayesian (VB) inference engine in BayesPy assumes that the posterior approximation factorizes with respect to nodes and plates. VMP is based on updating one node at a time (the plates in one node can be updated simultaneously) and iteratively updating all nodes in turns until convergence.

4.2.1 Standard update equation

The general update equation for the factorized approximation of node $\boldsymbol{\theta}$ is the following:

$$\log q(\boldsymbol{\theta}) = \langle \log p\left(\boldsymbol{\theta}|\mathrm{pa}(\boldsymbol{\theta})\right)\rangle + \sum_{\mathbf{x}\in\mathrm{ch}(\boldsymbol{\theta})} \langle \log p(\mathbf{x}|\mathrm{pa}(\mathbf{x}))\rangle + \mathrm{const}, \tag{4.1}$$

where $\mathrm{pa}(\boldsymbol{\theta})$ and $\mathrm{ch}(\boldsymbol{\theta})$ are the set of parents and children of $\boldsymbol{\theta}$, respectively. Thus, the posterior approximation of a node is updated by taking a sum of the expectations of all log densities in which the node variable appears. The expectations are over the approximate distribution of all other variables than $\boldsymbol{\theta}$. Actually, not all the variables are needed, because the non-constant part depends only on the Markov blanket of $\boldsymbol{\theta}$. This leads to a local optimization scheme, which uses messages from neighbouring nodes.

The messages are simple for conjugate exponential family models. An exponential family distribution has the following log probability density function:

$$\log p(\mathbf{x}|\boldsymbol{\Theta}) = \mathbf{u}_{\mathbf{x}}(\mathbf{x})^{\mathrm{T}}\boldsymbol{\phi}_{\mathbf{x}}(\boldsymbol{\Theta}) + g_{\mathbf{x}}(\boldsymbol{\Theta}) + f_{\mathbf{x}}(\mathbf{x}), \tag{4.2}$$

where $\boldsymbol{\Theta} = \{\boldsymbol{\theta}_j\}$ is the set of parents, $\mathbf{u}$ is the sufficient statistic vector, $\boldsymbol{\phi}$ is the natural parameter vector, g is the negative log normalizer, and f is the log base function. Note that the log density is linear with respect to the terms that are functions of $\mathbf{x}$: $\mathbf{u}$ and f. If a parent has a conjugate prior, (4.2) is also linear with respect to the parent's sufficient statistic vector. Thus, (4.2) can be re-organized with respect to a parent $\boldsymbol{\theta}_j$ as

$$\log p(\mathbf{x}|\boldsymbol{\Theta}) = \mathbf{u}_{\boldsymbol{\theta}_j}(\boldsymbol{\theta}_j)^{\mathrm{T}}\boldsymbol{\phi}_{\mathbf{x}\to\boldsymbol{\theta}_j}(\mathbf{x}, \{\boldsymbol{\theta}_k\}_{k\neq j}) + \mathrm{const},$$

where $\mathbf{u}_{\boldsymbol{\theta}_j}$ is the sufficient statistic vector of $\boldsymbol{\theta}_j$ and the constant part is constant with respect to $\boldsymbol{\theta}_j$. Thus, the update equation (4.1) for $\boldsymbol{\theta}_j$ can be written as

$$\begin{aligned}
\log q(\boldsymbol{\theta}_j) &= \mathbf{u}_{\boldsymbol{\theta}_j}(\boldsymbol{\theta}_j)^{\mathrm{T}}\langle\boldsymbol{\phi}_{\boldsymbol{\theta}_j}\rangle + f_{\boldsymbol{\theta}_j}(\boldsymbol{\theta}_j) + \mathbf{u}_{\boldsymbol{\theta}_j}(\boldsymbol{\theta}_j)^{\mathrm{T}} \sum_{\mathbf{x}\in\mathrm{ch}(\boldsymbol{\theta}_j)} \langle\boldsymbol{\phi}_{\mathbf{x}\to\boldsymbol{\theta}_j}\rangle + \mathrm{const}, \\
&= \mathbf{u}_{\boldsymbol{\theta}_j}(\boldsymbol{\theta}_j)^{\mathrm{T}} \left(\langle\boldsymbol{\phi}_{\boldsymbol{\theta}_j}\rangle + \sum_{\mathbf{x}\in\mathrm{ch}(\boldsymbol{\theta}_j)} \langle\boldsymbol{\phi}_{\mathbf{x}\to\boldsymbol{\theta}_j}\rangle \right) + f_{\boldsymbol{\theta}_j}(\boldsymbol{\theta}_j) + \mathrm{const},
\end{aligned}$$

where the summation is over all the child nodes of $\boldsymbol{\theta}_j$. Because of the conjugacy, $\langle\boldsymbol{\phi}_{\boldsymbol{\theta}_j}\rangle$ depends (multi)linearly on the parents' sufficient statistic vector. Similarly, $\langle\boldsymbol{\phi}_{\mathbf{x}\to\boldsymbol{\theta}_j}\rangle$ depends (multi)linearly on the expectations of the children's and co-parents' sufficient statistics. This gives the following update equation for the natural parameter vector of the posterior approximation $q(\boldsymbol{\phi}_j)$:

$$\tilde{\boldsymbol{\phi}}_j = \langle\boldsymbol{\phi}_{\boldsymbol{\theta}_j}\rangle + \sum_{\mathbf{x}\in\mathrm{ch}(\boldsymbol{\theta}_j)} \langle\boldsymbol{\phi}_{\mathbf{x}\to\boldsymbol{\theta}_j}\rangle. \tag{4.3}$$

4.2.2 Variational messages

The update equation (4.3) leads to a message passing scheme: the term $\langle\boldsymbol{\phi}_{\boldsymbol{\theta}_j}\rangle$ is a function of the parents' sufficient statistic vector and the term $\langle\boldsymbol{\phi}_{\mathbf{x}\to\boldsymbol{\theta}_j}\rangle$ can be interpreted as a message from the child node $\mathbf{x}$. Thus, the message from the child node $\mathbf{x}$ to the parent node $\boldsymbol{\theta}$ is

$$\mathbf{m}_{\mathbf{x}\to\boldsymbol{\theta}} \equiv \langle\boldsymbol{\phi}_{\mathbf{x}\to\boldsymbol{\theta}}\rangle,$$

which can be computed as a function of the sufficient statistic vector of the co-parent nodes of $\boldsymbol{\theta}$ and the sufficient statistic vector of the child node $\mathbf{x}$. The message from the parent node $\boldsymbol{\theta}$ to the child node $\mathbf{x}$ is simply the expectation of the sufficient statistic vector:

$$\mathbf{m}_{\boldsymbol{\theta}\to\mathbf{x}} \equiv \langle \mathbf{u}_{\boldsymbol{\theta}} \rangle.$$

In order to compute the expectation of the sufficient statistic vector we need to write $q(\boldsymbol{\theta})$ as

$$\log q(\boldsymbol{\theta}) = \mathbf{u}(\boldsymbol{\theta})^{\mathrm{T}} \tilde{\boldsymbol{\phi}} + \tilde{g}(\tilde{\boldsymbol{\phi}}) + f(\boldsymbol{\theta}),$$

where $\tilde{\boldsymbol{\phi}}$ is the natural parameter vector of $q(\boldsymbol{\theta})$. Now, the expectation of the sufficient statistic vector is defined as

$$\langle \mathbf{u}_{\boldsymbol{\theta}} \rangle = - \frac{\partial \tilde{g}}{\partial \tilde{\boldsymbol{\phi}}_{\boldsymbol{\theta}}} (\tilde{\boldsymbol{\phi}}_{\boldsymbol{\theta}}). \tag{4.4}$$

We call this expectation of the sufficient statistic vector as the moments vector.

4.2.3 Lower bound

Computing the VB lower bound is not necessary in order to find the posterior approximation, although it is extremely useful in monitoring convergence and possible bugs. The VB lower bound can be written as

$$\mathcal{L} = \langle \log p(\mathbf{Y}, \mathbf{X}) \rangle - \langle \log q(\mathbf{X}) \rangle,$$

where $\mathbf{Y}$ is the set of all observed variables and $\mathbf{X}$ is the set of all latent variables. It can also be written as

$$\mathcal{L} = \sum_{\mathbf{y}\in\mathbf{Y}} \langle \log p(\mathbf{y}|\mathrm{pa}(\mathbf{y})) \rangle + \sum_{\mathbf{x}\in\mathbf{X}} \left[\langle \log p(\mathbf{x}|\mathrm{pa}(\mathbf{x})) \rangle - \langle \log q(\mathbf{x}) \right],$$

which shows that observed and latent variables contribute differently to the lower bound. These contributions have simple forms for exponential family nodes. Observed exponential family nodes contribute to the lower bound as follows:

$$\langle \log p(\mathbf{y}|\mathrm{pa}(\mathbf{y})) \rangle = \mathbf{u}(\mathbf{y})^T \langle \boldsymbol{\phi} \rangle + \langle g \rangle + f(\mathbf{x}),$$

where $\mathbf{y}$ is the observed data. On the other hand, latent exponential family nodes contribute to the lower bound as follows:

$$\langle \log p(\mathbf{x}|\boldsymbol{\theta}) \rangle - \langle \log q(\mathbf{x}) \rangle = \langle \mathbf{u} \rangle^T (\langle \boldsymbol{\phi} \rangle - \tilde{\boldsymbol{\phi}}) + \langle g \rangle - \tilde{g}.$$

If a node is partially observed and partially unobserved, these formulas are applied plate-wise appropriately.

4.2.4 Terms

To summarize, implementing VMP requires one to write for each stochastic exponential family node:

$\langle \boldsymbol{\phi} \rangle$: the expectation of the prior natural parameter vector

Computed as a function of the messages from parents.

$\tilde{\boldsymbol{\phi}}$: natural parameter vector of the posterior approximation

Computed as a sum of $\langle \boldsymbol{\phi} \rangle$ and the messages from children.

$\langle \mathbf{u} \rangle$: the posterior moments vector

Computed as a function of $\tilde{\boldsymbol{\phi}}$ as defined in (4.4).

$\mathbf{u}(\mathbf{x})$: the moments vector for given data

Computed as a function of of the observed data $\mathbf{x}$.

$\langle g \rangle$: the expectation of the negative log normalizer of the prior

Computed as a function of parent moments.

$\tilde{g}$: the negative log normalizer of the posterior approximation

Computed as a function of $\tilde{\boldsymbol{\phi}}$.

$f(\mathbf{x})$: the log base measure for given data

Computed as a function of the observed data $\mathbf{x}$.

$\langle \boldsymbol{\phi}_{\mathbf{x}\rightarrow\boldsymbol{\theta}} \rangle$: the message to parent $\boldsymbol{\theta}$

Computed as a function of the moments of this node and the other parents.

Deterministic nodes require only the following terms:

$\langle \mathbf{u} \rangle$: the posterior moments vector

Computed as a function of the messages from the parents.

$\mathbf{m}$: the message to a parent

Computed as a function of the messages from the other parents and all children.

4.3 Implementing inference engines

Currently, only variational Bayesian inference engine is implemented. This implementation is not very modular, that is, the inference engine is not well separated from the model construction. Thus, it is not straightforward to implement other inference engines at the moment. Improving the modularity of the inference engine and model construction is future work with high priority. In any case, BayesPy aims to be an efficient, simple and modular Bayesian package for variational inference at least.

4.4 Implementing nodes

The main goal of BayesPy is to provide a package which enables easy and flexible construction of simple and complex models with efficient inference. However, users may sometimes be unable to construct their models because the built-in nodes do not implement some specific features. Thus, one may need to implement new nodes in order to construct the model. BayesPy aims to make the implementation of new nodes both simple and fast. Probably, a large complex model can be constructed almost completely with the built-in nodes and the user needs to implement only a few nodes.

4.4.1 Moments

In order to implement nodes, it is important to understand the messaging framework of the nodes. A node is a unit of calculation which communicates to its parent and child nodes using messages. These messages have types that need to match between nodes, that is, the child node needs to understand the messages its parents are sending and vice versa. Thus, a node defines which message type it requires from each of its parents, and only nodes that have that type of output message (i.e., the message to a child node) are valid parent nodes for that node.

The message type is defined by the moments of the parent node. The moments are a collection of expectations: $\{\langle f_1(X)\rangle, \ldots, \langle f_N(X)\rangle\}$. The functions $f_1, \ldots, f_N$ (and the number of the functions) define the message type and they are the sufficient statistic as discussed in the previous section. Different message types are represented by `Moments` class hierarchy. For instance, `GaussianMoments` represents a message type with parent moments $\{\langle \mathbf{x} \rangle, \langle \mathbf{x}\mathbf{x}^T \rangle\}$ and `WishartMoments` a message type with parent moments $\{\langle \boldsymbol{\Lambda} \rangle, \langle \log |\boldsymbol{\Lambda}| \rangle\}$.

Let us give an example: `Gaussian` node outputs `GaussianMoments` messages and `Wishart` node outputs `WishartMoments` messages. `Gaussian` node requires that it receives `GaussianMoments` messages from the mean parent node and `WishartMoments` messages from the precision parent node. Thus, `Gaussian` and `Wishart` are valid node classes as the mean and precision parent nodes of `Gaussian` node.

Note that several nodes may have the same output message type and some message types can be transformed to other message types using deterministic converter nodes. For instance, `Gaussian` and `GaussianARD` nodes both output `GaussianMoments` messages, deterministic `SumMultiply` also outputs `GaussianMoments` messages, and deterministic converter `_MarkovChainToGaussian` converts `GaussianMarkovChainMoments` to `GaussianMoments`.

Each node specifies the message type requirements of its parents by `Node._parent_moments` attribute which is a list of `Moments` sub-class instances. These moments objects have a few purpose when creating the node: 1) check that parents are sending proper messages; 2) if parents use different message type, try to add a converter which converts the messages to the correct type if possible; 3) if given parents are not nodes but numeric arrays, convert them to constant nodes with correct output message type.

When implementing a new node, it is not always necessary to implement a new moments class. If another node has the same sufficient statistic vector, thus the same moments, that moments class can be used. Otherwise, one must implement a simple moments class which has the following methods:

- `Moments.compute_fixed_moments()`

  Computes the moments for a known value. This is used to compute the moments of constant numeric arrays and wrap them into constant nodes.

- `Moments.compute_dims_from_values()`

  Given a known value of the variable, return the shape of the variable dimensions in the moments. This is used to solve the shape of the moments array for constant nodes.

4.4.2 Distributions

In order to implement a stochastic exponential family node, one must first write down the log probability density function of the node and derive the terms discussed in section *Terms*. These terms are implemented and collected as a class which is a subclass of `Distribution`. The main reason to implement these methods in another class instead of the node class itself is that these methods can be used without creating a node, for instance, in `Mixture` class.

For exponential family distributions, the distribution class is a subclass of `ExponentialFamilyDistribution`, and the relation between the terms in section *Terms* and the methods is as follows:

- `ExponentialFamilyDistribution.compute_phi_from_parents()`

  Computes the expectation of the natural parameters $\langle \boldsymbol{\phi} \rangle$ in the prior distribution given the moments of the parents.

- `ExponentialFamilyDistribution.compute_cgf_from_parents()`

  Computes the expectation of the negative log normalizer $\langle g \rangle$ of the prior distribution given the moments of the parents.

- `ExponentialFamilyDistribution.compute_moments_and_cgf()`

  Computes the moments $\langle \mathbf{u} \rangle$ and the negative log normalizer $\tilde{g}$ of the posterior distribution given the natural parameters $\tilde{\boldsymbol{\phi}}$.

- `ExponentialFamilyDistribution.compute_message_to_parent()`

  Computes the message $\langle \boldsymbol{\phi}_{\mathbf{x}\rightarrow\boldsymbol{\theta}} \rangle$ from the node $\mathbf{x}$ to its parent node $\boldsymbol{\theta}$ given the moments of the node and the other parents.

- `ExponentialFamilyDistribution.compute_fixed_moments_and_f()`

Computes $\mathbf{u}(\mathbf{x})$ and $f(\mathbf{x})$ for given observed value $\mathbf{x}$. Without this method, variables from this distribution cannot be observed.

For each stochastic exponential family node, one must write a distribution class which implements these methods. After that, the node class is basically a simple wrapper and it also stores the moments and the natural parameters of the current posterior approximation. Note that the distribution classes do not store node-specific information, they are more like static collections of methods. However, sometimes the implementations depend on some information, such as the dimensionality of the variable, and this information must be provided, if needed, when constructing the distribution object.

In addition to the methods listed above, it is necessary to implement a few more methods in some cases. This happens when the plates of the parent do not map to the plates directly as discussed in section *Irregular plates*. Then, one must write methods that implement this plate mapping and apply the same mapping to the mask array:

- `ExponentialFamilyDistribution.plates_from_parent()`

  Given the plates of the parent, return the resulting plates of the child.

- `ExponentialFamilyDistribution.plates_to_parent()`

  Given the plates of the child, return the plates of the parent that would have resulted them.

- `ExponentialFamilyDistribution.compute_mask_to_parent()`

  Given the mask array of the child, apply the plate mapping.

It is important to understand when one must implement these methods, because the default implementations in the base class will lead to errors or weird results.

4.4.3 Stochastic exponential family nodes

After implementing the distribution class, the next task is to implement the node class. First, we need to explain a few important attributes before we can explain how to implement a node class.

Stochastic exponential family nodes have two attributes that store the state of the posterior distribution:

- `phi`

  The natural parameter vector $\tilde{\phi}$ of the posterior approximation.

- `u`

  The moments $\langle \mathbf{u} \rangle$ of the posterior approximation.

Instead of storing these two variables as vectors (as in the mathematical formulas), they are stored as lists of arrays with convenient shapes. For instance, `Gaussian` node stores the moments as a list consisting of a vector $\langle \mathbf{x} \rangle$ and a matrix $\langle \mathbf{x}\mathbf{x}^T \rangle$ instead of reshaping and concatenating these into a single vector. The same applies for the natural parameters `phi` because it has the same shape as `u`.

The shapes of the arrays in the lists `u` and `phi` consist of the shape caused by the plates and the shape caused by the variable itself. For instance, the moments of `Gaussian` node have shape `(D,)` and `(D, D)`, where `D` is the dimensionality of the Gaussian vector. In addition, if the node has plates, they are added to these shapes. Thus, for instance, if the `Gaussian` node has plates `(3, 7)` and `D` is 5, the shape of `u[0]` and `phi[0]` would be `(3, 7, 5)` and the shape of `u[1]` and `phi[1]` would be `(3, 7, 5, 5)`. This shape information is stored in the following attributes:

- `plates` : a tuple

  The plates of the node. In our example, `(3, 7)`.

- `dims` : a list of tuples

The shape of each of the moments arrays (or natural parameter arrays) without plates. In our example, `[ (5,), (5, 5) ]`.

Finally, three attributes define VMP for the node:

- `_moments` : `Moments` sub-class instance

  An object defining the moments of the node.

- `_parent_moments` : list of `Moments` sub-class instances

  A list defining the moments requirements for each parent.

- `_distribution` : `Distribution` sub-class instance

  An object implementing the VMP formulas.

Basically, a node class is a collection of the above attributes. When a node is created, these attributes are defined. The base class for exponential family nodes, `ExponentialFamily`, provides a simple default constructor which does not need to be overwritten if `dims`, `_moments`, `_parent_moments` and `_distribution` can be provided as static class attributes. For instance, `Gamma` node defines these attributes statically. However, usually at least one of these attributes cannot be defined statically in the class. In that case, one must implement a class method which overloads `ExponentialFamily._constructor()`. The purpose of this method is to define all the attributes given the parent nodes. These are defined using a class method instead of `__init__` method in order to be able to use the class constructors statically, for instance, in `Mixture` class. This construction allows users to create mixtures of any exponential family distribution with simple syntax.

The parents of a node must be converted so that they have a correct message type, because the user may have provided numeric arrays or nodes with incorrect message type. Numeric arrays should be converted to constant nodes with correct message type. Incorrect message type nodes should be converted to correct message type nodes if possible. Thus, the constructor should use `Node._ensure_moments` method to make sure the parent is a node with correct message type. Instead of calling this method for each parent node in the constructor, one can use `ensureparents` decorator to do this automatically. However, the decorator requires that `_parent_moments` attribute has already been defined statically. If this is not possible, the parent nodes must be converted manually in the constructor, because one should never assume that the parent nodes given to the constructor are nodes with correct message type or even nodes at all.

4.4.4 Deterministic nodes

Deterministic nodes are nodes that do not correspond to any probability distribution but rather a deterministic function. It does not have any moments or natural parameters to store. A deterministic node is implemented as a subclass of `Deterministic` base class. The new node class must implement the following methods:

- `Deterministic._compute_moments()`

  Computes the moments given the moments of the parents.

- `Deterministic._compute_message_to_parent()`

  Computes the message to a parent node given the message from the children and the moments of the other parents. In some cases, one may want to implement `Deterministic._compute_message_and_mask_to_parent()` or `Deterministic._message_to_parent()` instead in order to gain more control over efficient computation.

Similarly as in `Distribution` class, if the node handles plates irregularly, it is important to implement the following methods:

- `Deterministic._plates_from_parent()`

  Given the plates of the parent, return the resulting plates of the child.

- `Deterministic._plates_to_parent()`

  Given the plates of the child, return the plates of the parent that would have resulted them.

- `Deterministic._compute_mask_to_parent()`

  Given the mask array, convert it to a plate mask of the parent.

Converter nodes

Sometimes a node has incorrect message type but the message can be converted into a correct type. For instance, `GaussianMarkovChain` has `GaussianMarkovChainMoments` message type, which means moments $\{\langle \mathbf{x}_n \rangle, \langle \mathbf{x}_n \mathbf{x}_n^T \rangle, \langle \mathbf{x}_n \mathbf{x}_{n-1}^T \rangle\}_{n=1}^N$. These moments can be converted to `GaussianMoments` by ignoring the third element and considering the time axis as a plate axis. Thus, if a node requires `GaussianMoments` message from its parent, `GaussianMarkovChain` is a valid parent if its messages are modified properly. This conversion is implemented in `_MarkovChainToGaussian` converter class. Converter nodes are simple deterministic nodes that have one parent node and they convert the messages to another message type.

For the user, it is not convenient if the exact message type has to be known and an explicit converter node needs to be created. Thus, the conversions are done automatically and the user will be unaware of them. In order to enable this automatization, when writing a converter node, one should register the converter to the moments class using `Moments.add_converter()`. For instance, a class `X` which converts moments `A` to moments `B` is registered as `A.add_conveter(B, X)`. After that, `Node._ensure_moments()` and `Node._convert()` methods are used to perform the conversions automatically. The conversion can consist of several consecutive converter nodes, and the least number of conversions is used.

CHAPTER

FIVE

USER API

5.1 bayespy.nodes

Package for nodes used to construct the model.

5.1.1 Stochastic nodes

Nodes for Gaussian variables:

Gaussian(mu, Lambda, **kwargs)	Node for Gaussian variables.
GaussianARD(mu, alpha[, ndim, shape])	Node for Gaussian variables with ARD prior.

bayespy.nodes.Gaussian

class bayespy.nodes.**Gaussian**(*mu*, *Lambda*, ***kwargs*)

Node for Gaussian variables.

The node represents a D-dimensional vector from the Gaussian distribution:

$$\mathbf{x} \sim \mathcal{N}(\boldsymbol{\mu}, \mathbf{\Lambda}),$$

where $\boldsymbol{\mu}$ is the mean vector and $\mathbf{\Lambda}$ is the precision matrix (i.e., inverse of the covariance matrix).

$$\mathbf{x}, \boldsymbol{\mu} \in \mathbb{R}^D, \quad \mathbf{\Lambda} \in \mathbb{R}^{D \times D}, \quad \mathbf{\Lambda} \text{ symmetric positive definite}$$

Parameters **mu** : Gaussian-like node or GaussianGammaISO-like node or GaussianWishart-like node or array

Mean vector

Lambda : Wishart-like node or array

Precision matrix

See also:

Wishart, GaussianARD, GaussianWishart, GaussianGammaARD, GaussianGammaISO

__init__(*mu*, *Lambda*, ***kwargs*)

Create Gaussian node

Methods

__init__(mu, Lambda, **kwargs)	Create Gaussian node
add_plate_axis(to_plate)	
broadcasting_multiplier(plates, *args)	
delete()	Delete this node and the children
get_gradient(rg)	Computes gradient with respect to the natural parameters.
get_mask()	
get_moments()	
get_parameters()	Return parameters of the VB distribution.
get_riemannian_gradient()	Computes the Riemannian/natural gradient.
get_shape(ind)	
has_plotter()	Return True if the node has a plotter
initialize_from_parameters(mu, Lambda)	
initialize_from_prior()	
initialize_from_random()	Set the variable to a random sample from the current distribution.
initialize_from_value(x, *args)	
load(group)	Load the state of the node from a HDF5 file.
logpdf(X[, mask])	Compute the log probability density function Q(X) of this node.
lower_bound_contribution([gradient, ...])	Compute E[log p(X|parents) - log q(X)]
lowerbound()	
move_plates(from_plate, to_plate)	
observe(x, *args[, mask])	Fix moments, compute f and propagate mask.
pdf(X[, mask])	Compute the probability density function of this node.
plot([fig])	Plot the node distribution using the plotter of the node
random()	Draw a random sample from the distribution.
rotate(R[, inv, logdet, Q])	
rotate_matrix(R1, R2[, inv1, logdet1, inv2, ...])	The vector is reshaped into a matrix by stacking the row vectors.
save(group)	Save the state of the node into a HDF5 file.
set_parameters(x)	Set the parameters of the VB distribution.
set_plotter(plotter)	
show()	
unobserve()	
update([annealing])	

bayespy.nodes.Gaussian.__init__

Gaussian.**__init__**(*mu*, *Lambda*, ***kwargs*)

Create Gaussian node

bayespy.nodes.Gaussian.add_plate_axis

Gaussian.**add_plate_axis**(*to_plate*)

bayespy.nodes.Gaussian.broadcasting_multiplier

Gaussian.**broadcasting_multiplier**(*plates*, **args*)

bayespy.nodes.Gaussian.delete

`Gaussian.delete()`
Delete this node and the children

bayespy.nodes.Gaussian.get_gradient

`Gaussian.get_gradient(rg)`
Computes gradient with respect to the natural parameters.

The function takes the Riemannian gradient as an input. This is for three reasons: 1) You probably want to use the Riemannian gradient anyway so this helps avoiding accidental use of this function. 2) The gradient is computed by using the Riemannian gradient and chain rules. 3) Probably you need both Riemannian and normal gradients anyway so you can provide it to this function to avoid re-computing it.

bayespy.nodes.Gaussian.get_mask

`Gaussian.get_mask()`

bayespy.nodes.Gaussian.get_moments

`Gaussian.get_moments()`

bayespy.nodes.Gaussian.get_parameters

`Gaussian.get_parameters()`
Return parameters of the VB distribution.

The parameters should be such that they can be used for optimization, that is, use log transformation for positive parameters.

bayespy.nodes.Gaussian.get_riemannian_gradient

`Gaussian.get_riemannian_gradient()`
Computes the Riemannian/natural gradient.

bayespy.nodes.Gaussian.get_shape

`Gaussian.get_shape(ind)`

bayespy.nodes.Gaussian.has_plotter

`Gaussian.has_plotter()`
Return True if the node has a plotter

bayespy.nodes.Gaussian.initialize_from_parameters

`Gaussian.initialize_from_parameters(mu, Lambda)`

bayespy.nodes.Gaussian.initialize_from_prior

Gaussian.**initialize_from_prior**()

bayespy.nodes.Gaussian.initialize_from_random

Gaussian.**initialize_from_random**()
Set the variable to a random sample from the current distribution.

bayespy.nodes.Gaussian.initialize_from_value

Gaussian.**initialize_from_value**(*x*, **args*)

bayespy.nodes.Gaussian.load

Gaussian.**load**(*group*)
Load the state of the node from a HDF5 file.

bayespy.nodes.Gaussian.logpdf

Gaussian.**logpdf**(*X*, *mask=True*)
Compute the log probability density function Q(X) of this node.

bayespy.nodes.Gaussian.lower_bound_contribution

Gaussian.**lower_bound_contribution**(*gradient=False*, *ignore_masked=True*)
Compute E[log p(X|parents) - log q(X)]

If deterministic annealing is used, the term E[-log q(X)] is divided by the anneling coefficient. That is, phi and cgf of q are multiplied by the temperature (inverse annealing coefficient).

bayespy.nodes.Gaussian.lowerbound

Gaussian.**lowerbound**()

bayespy.nodes.Gaussian.move_plates

Gaussian.**move_plates**(*from_plate*, *to_plate*)

bayespy.nodes.Gaussian.observe

Gaussian.**observe**(*x*, **args*, *mask=True*)
Fix moments, compute f and propagate mask.

bayespy.nodes.Gaussian.pdf

`Gaussian.pdf(X, mask=True)`
Compute the probability density function of this node.

bayespy.nodes.Gaussian.plot

`Gaussian.plot(fig=None, **kwargs)`
Plot the node distribution using the plotter of the node

Because the distributions are in general very difficult to plot, the user must specify some functions which performs the plotting as wanted. See, for instance, bayespy.plot.plotting for available plotters, that is, functions that perform plotting for a node.

bayespy.nodes.Gaussian.random

`Gaussian.random()`
Draw a random sample from the distribution.

bayespy.nodes.Gaussian.rotate

`Gaussian.rotate(R, inv=None, logdet=None, Q=None)`

bayespy.nodes.Gaussian.rotate_matrix

`Gaussian.rotate_matrix(R1, R2, inv1=None, logdet1=None, inv2=None, logdet2=None, Q=None)`
The vector is reshaped into a matrix by stacking the row vectors.

Computes R1*X*R2', which is identical to kron(R1,R2)*x (??)

Note that this is slightly different from the standard Kronecker product definition because Numpy stacks row vectors instead of column vectors.

Parameters **R1** : ndarray

A matrix from the left

R2 : ndarray

A matrix from the right

bayespy.nodes.Gaussian.save

`Gaussian.save(group)`
Save the state of the node into a HDF5 file.

group can be the root

bayespy.nodes.Gaussian.set_parameters

`Gaussian.set_parameters(x)`

Set the parameters of the VB distribution.

The parameters should be such that they can be used for optimization, that is, use log transformation for positive parameters.

bayespy.nodes.Gaussian.set_plotter

`Gaussian.set_plotter(plotter)`

bayespy.nodes.Gaussian.show

`Gaussian.show()`

bayespy.nodes.Gaussian.unobserve

`Gaussian.unobserve()`

bayespy.nodes.Gaussian.update

`Gaussian.update(annealing=1.0)`

Attributes

dims	
plates	
plates_multiplier	Plate multiplier is applied to messages to parents

bayespy.nodes.Gaussian.dims

`Gaussian.dims = None`

bayespy.nodes.Gaussian.plates

`Gaussian.plates = None`

bayespy.nodes.Gaussian.plates_multiplier

`Gaussian.plates_multiplier`

Plate multiplier is applied to messages to parents

bayespy.nodes.GaussianARD

class `bayespy.nodes.GaussianARD`(*mu*, *alpha*, *ndim=None*, *shape=None*, ***kwargs*)

Node for Gaussian variables with ARD prior.

The node represents a D-dimensional vector from the Gaussian distribution:

$$\mathbf{x} \sim \mathcal{N}(\boldsymbol{\mu}, \mathrm{diag}(\boldsymbol{\alpha})),$$

where $\boldsymbol{\mu}$ is the mean vector and $\mathrm{diag}(\boldsymbol{\alpha})$ is the diagonal precision matrix (i.e., inverse of the covariance matrix).

$$\mathbf{x}, \boldsymbol{\mu} \in \mathbb{R}^D, \quad \alpha_d > 0 \text{ for } d = 0, \ldots, D-1$$

Note: The form of the posterior approximation is a Gaussian distribution with full covariance matrix instead of a diagonal matrix.

Parameters **mu** : Gaussian-like node or GaussianGammaISO-like node or GaussianGammaARD-like node or array

Mean vector

alpha : gamma-like node or array

Diagonal elements of the precision matrix

See also:

`Gamma`, `Gaussian`, `GaussianGammaARD`, `GaussianGammaISO`, `GaussianWishart`

`__init__`(*mu*, *alpha*, *ndim=None*, *shape=None*, ***kwargs*)

Create GaussianARD node.

Methods

`__init__`(mu, alpha[, ndim, shape])	Create GaussianARD node.
`add_plate_axis`(to_plate)	
`broadcasting_multiplier`(plates, *args)	
`delete`()	Delete this node and the children
`get_gradient`(rg)	Computes gradient with respect to the natural parameters.
`get_mask`()	
`get_moments`()	
`get_parameters`()	Return parameters of the VB distribution.
`get_riemannian_gradient`()	Computes the Riemannian/natural gradient.
`get_shape`(ind)	
`has_plotter`()	Return True if the node has a plotter
`initialize_from_mean_and_covariance`(mu, Cov)	
`initialize_from_parameters`(mu, alpha)	
`initialize_from_prior`()	
`initialize_from_random`()	Set the variable to a random sample from the current distribution.
`initialize_from_value`(x, *args)	
`load`(group)	Load the state of the node from a HDF5 file.
`logpdf`(X[, mask])	Compute the log probability density function Q(X) of this node.
`lower_bound_contribution`([gradient, ...])	Compute E[log p(X|parents) - log q(X)]
`lowerbound`()	
`move_plates`(from_plate, to_plate)	
`observe`(x, *args[, mask])	Fix moments, compute f and propagate mask.

Continued on next page

Table 5.5 – continued from previous page

pdf(X[, mask])	Compute the probability density function of this node.
plot([fig])	Plot the node distribution using the plotter of the node
random()	Draw a random sample from the distribution.
rotate(R[, inv, logdet, axis, Q, subset])	
rotate_plates(Q[, plate_axis])	Approximate rotation of a plate axis.
save(group)	Save the state of the node into a HDF5 file.
set_parameters(x)	Set the parameters of the VB distribution.
set_plotter(plotter)	
show()	
unobserve()	
update([annealing])	

bayespy.nodes.GaussianARD.__init__

GaussianARD.**__init__**(*mu*, *alpha*, *ndim=None*, *shape=None*, ***kwargs*)

Create GaussianARD node.

bayespy.nodes.GaussianARD.add_plate_axis

GaussianARD.**add_plate_axis**(*to_plate*)

bayespy.nodes.GaussianARD.broadcasting_multiplier

GaussianARD.**broadcasting_multiplier**(*plates*, **args*)

bayespy.nodes.GaussianARD.delete

GaussianARD.**delete**()

Delete this node and the children

bayespy.nodes.GaussianARD.get_gradient

GaussianARD.**get_gradient**(*rg*)

Computes gradient with respect to the natural parameters.

The function takes the Riemannian gradient as an input. This is for three reasons: 1) You probably want to use the Riemannian gradient anyway so this helps avoiding accidental use of this function. 2) The gradient is computed by using the Riemannian gradient and chain rules. 3) Probably you need both Riemannian and normal gradients anyway so you can provide it to this function to avoid re-computing it.

bayespy.nodes.GaussianARD.get_mask

GaussianARD.**get_mask**()

bayespy.nodes.GaussianARD.get_moments

GaussianARD.**get_moments**()

bayespy.nodes.GaussianARD.get_parameters

`GaussianARD.get_parameters()`
Return parameters of the VB distribution.

The parameters should be such that they can be used for optimization, that is, use log transformation for positive parameters.

bayespy.nodes.GaussianARD.get_riemannian_gradient

`GaussianARD.get_riemannian_gradient()`
Computes the Riemannian/natural gradient.

bayespy.nodes.GaussianARD.get_shape

`GaussianARD.get_shape(ind)`

bayespy.nodes.GaussianARD.has_plotter

`GaussianARD.has_plotter()`
Return True if the node has a plotter

bayespy.nodes.GaussianARD.initialize_from_mean_and_covariance

`GaussianARD.initialize_from_mean_and_covariance(mu, Cov)`

bayespy.nodes.GaussianARD.initialize_from_parameters

`GaussianARD.initialize_from_parameters(mu, alpha)`

bayespy.nodes.GaussianARD.initialize_from_prior

`GaussianARD.initialize_from_prior()`

bayespy.nodes.GaussianARD.initialize_from_random

`GaussianARD.initialize_from_random()`
Set the variable to a random sample from the current distribution.

bayespy.nodes.GaussianARD.initialize_from_value

`GaussianARD.initialize_from_value(x, *args)`

bayespy.nodes.GaussianARD.load

GaussianARD.**load**(*group*)
Load the state of the node from a HDF5 file.

bayespy.nodes.GaussianARD.logpdf

GaussianARD.**logpdf**(*X*, *mask=True*)
Compute the log probability density function Q(X) of this node.

bayespy.nodes.GaussianARD.lower_bound_contribution

GaussianARD.**lower_bound_contribution**(*gradient=False*, *ignore_masked=True*)
Compute E[log p(X|parents) - log q(X)]

If deterministic annealing is used, the term E[-log q(X)] is divided by the anneling coefficient. That is, phi and cgf of q are multiplied by the temperature (inverse annealing coefficient).

bayespy.nodes.GaussianARD.lowerbound

GaussianARD.**lowerbound**()

bayespy.nodes.GaussianARD.move_plates

GaussianARD.**move_plates**(*from_plate*, *to_plate*)

bayespy.nodes.GaussianARD.observe

GaussianARD.**observe**(*x*, **args*, *mask=True*)
Fix moments, compute f and propagate mask.

bayespy.nodes.GaussianARD.pdf

GaussianARD.**pdf**(*X*, *mask=True*)
Compute the probability density function of this node.

bayespy.nodes.GaussianARD.plot

GaussianARD.**plot**(*fig=None*, ***kwargs*)
Plot the node distribution using the plotter of the node

Because the distributions are in general very difficult to plot, the user must specify some functions which performs the plotting as wanted. See, for instance, bayespy.plot.plotting for available plotters, that is, functions that perform plotting for a node.

bayespy.nodes.GaussianARD.random

`GaussianARD.random()`
Draw a random sample from the distribution.

bayespy.nodes.GaussianARD.rotate

`GaussianARD.rotate(R, inv=None, logdet=None, axis=-1, Q=None, subset=None)`

bayespy.nodes.GaussianARD.rotate_plates

`GaussianARD.rotate_plates(Q, plate_axis=-1)`
Approximate rotation of a plate axis.

Mean is rotated exactly but covariance/precision matrix is rotated approximately.

bayespy.nodes.GaussianARD.save

`GaussianARD.save(group)`
Save the state of the node into a HDF5 file.

group can be the root

bayespy.nodes.GaussianARD.set_parameters

`GaussianARD.set_parameters(x)`
Set the parameters of the VB distribution.

The parameters should be such that they can be used for optimization, that is, use log transformation for positive parameters.

bayespy.nodes.GaussianARD.set_plotter

`GaussianARD.set_plotter(plotter)`

bayespy.nodes.GaussianARD.show

`GaussianARD.show()`

bayespy.nodes.GaussianARD.unobserve

`GaussianARD.unobserve()`

bayespy.nodes.GaussianARD.update

`GaussianARD.update(annealing=1.0)`

Attributes

dims	
plates	
plates_multiplier	Plate multiplier is applied to messages to parents

bayespy.nodes.GaussianARD.dims

GaussianARD.**dims** = None

bayespy.nodes.GaussianARD.plates

GaussianARD.**plates** = None

bayespy.nodes.GaussianARD.plates_multiplier

GaussianARD.**plates_multiplier**

Plate multiplier is applied to messages to parents

Nodes for precision and scale variables:

Gamma(a, b, **kwargs)	Node for gamma random variables.
Wishart(n, V, **kwargs)	Node for Wishart random variables.
Exponential(l, **kwargs)	Node for exponential random variables.

bayespy.nodes.Gamma

class bayespy.nodes.**Gamma**(*a*, *b*, ***kwargs*)

Node for gamma random variables.

Parameters **a** : scalar or array

Shape parameter

b : gamma-like node or scalar or array

Rate parameter

__init__(*a*, *b*, ***kwargs*)

Create gamma random variable node

Methods

__init__(a, b, **kwargs)	Create gamma random variable node
add_plate_axis(to_plate)	
as_diagonal_wishart()	
broadcasting_multiplier(plates, *args)	
delete()	Delete this node and the children
get_gradient(rg)	Computes gradient with respect to the natural parameters.
get_mask()	

Continued on next page

Table 5.8 – continued from previous page

get_moments()	
get_parameters()	Return parameters of the VB distribution.
get_riemannian_gradient()	Computes the Riemannian/natural gradient.
get_shape(ind)	
has_plotter()	Return True if the node has a plotter
initialize_from_parameters(*args)	
initialize_from_prior()	
initialize_from_random()	Set the variable to a random sample from the current distribution.
initialize_from_value(x, *args)	
load(group)	Load the state of the node from a HDF5 file.
logpdf(X[, mask])	Compute the log probability density function Q(X) of this node.
lower_bound_contribution([gradient, ...])	Compute E[log p(X|parents) - log q(X)]
lowerbound()	
move_plates(from_plate, to_plate)	
observe(x, *args[, mask])	Fix moments, compute f and propagate mask.
pdf(X[, mask])	Compute the probability density function of this node.
plot([fig])	Plot the node distribution using the plotter of the node
random()	Draw a random sample from the distribution.
save(group)	Save the state of the node into a HDF5 file.
set_parameters(x)	Set the parameters of the VB distribution.
set_plotter(plotter)	
show()	Print the distribution using standard parameterization.
unobserve()	
update([annealing])	

bayespy.nodes.Gamma.__init__

Gamma.**__init__**(*a*, *b*, ***kwargs*)

Create gamma random variable node

bayespy.nodes.Gamma.add_plate_axis

Gamma.**add_plate_axis**(*to_plate*)

bayespy.nodes.Gamma.as_diagonal_wishart

Gamma.**as_diagonal_wishart**()

bayespy.nodes.Gamma.broadcasting_multiplier

Gamma.**broadcasting_multiplier**(*plates*, **args*)

bayespy.nodes.Gamma.delete

Gamma.**delete**()

Delete this node and the children

bayespy.nodes.Gamma.get_gradient

Gamma.**get_gradient**(*rg*)
Computes gradient with respect to the natural parameters.

The function takes the Riemannian gradient as an input. This is for three reasons: 1) You probably want to use the Riemannian gradient anyway so this helps avoiding accidental use of this function. 2) The gradient is computed by using the Riemannian gradient and chain rules. 3) Probably you need both Riemannian and normal gradients anyway so you can provide it to this function to avoid re-computing it.

bayespy.nodes.Gamma.get_mask

Gamma.**get_mask**()

bayespy.nodes.Gamma.get_moments

Gamma.**get_moments**()

bayespy.nodes.Gamma.get_parameters

Gamma.**get_parameters**()
Return parameters of the VB distribution.

The parameters should be such that they can be used for optimization, that is, use log transformation for positive parameters.

bayespy.nodes.Gamma.get_riemannian_gradient

Gamma.**get_riemannian_gradient**()
Computes the Riemannian/natural gradient.

bayespy.nodes.Gamma.get_shape

Gamma.**get_shape**(*ind*)

bayespy.nodes.Gamma.has_plotter

Gamma.**has_plotter**()
Return True if the node has a plotter

bayespy.nodes.Gamma.initialize_from_parameters

Gamma.**initialize_from_parameters**(**args*)

bayespy.nodes.Gamma.initialize_from_prior

Gamma.**initialize_from_prior**()

bayespy.nodes.Gamma.initialize_from_random

Gamma.**initialize_from_random**()
Set the variable to a random sample from the current distribution.

bayespy.nodes.Gamma.initialize_from_value

Gamma.**initialize_from_value**(*x*, **args*)

bayespy.nodes.Gamma.load

Gamma.**load**(*group*)
Load the state of the node from a HDF5 file.

bayespy.nodes.Gamma.logpdf

Gamma.**logpdf**(*X*, *mask=True*)
Compute the log probability density function Q(X) of this node.

bayespy.nodes.Gamma.lower_bound_contribution

Gamma.**lower_bound_contribution**(*gradient=False*, *ignore_masked=True*)
Compute E[log p(X|parents) - log q(X)]

If deterministic annealing is used, the term E[-log q(X)] is divided by the anneling coefficient. That is, phi and cgf of q are multiplied by the temperature (inverse annealing coefficient).

bayespy.nodes.Gamma.lowerbound

Gamma.**lowerbound**()

bayespy.nodes.Gamma.move_plates

Gamma.**move_plates**(*from_plate*, *to_plate*)

bayespy.nodes.Gamma.observe

Gamma.**observe**(*x*, **args*, *mask=True*)
Fix moments, compute f and propagate mask.

bayespy.nodes.Gamma.pdf

Gamma.**pdf**(*X*, *mask=True*)
Compute the probability density function of this node.

bayespy.nodes.Gamma.plot

`Gamma.plot(fig=None, **kwargs)`

Plot the node distribution using the plotter of the node

Because the distributions are in general very difficult to plot, the user must specify some functions which performs the plotting as wanted. See, for instance, bayespy.plot.plotting for available plotters, that is, functions that perform plotting for a node.

bayespy.nodes.Gamma.random

`Gamma.random()`

Draw a random sample from the distribution.

bayespy.nodes.Gamma.save

`Gamma.save(group)`

Save the state of the node into a HDF5 file.

group can be the root

bayespy.nodes.Gamma.set_parameters

`Gamma.set_parameters(x)`

Set the parameters of the VB distribution.

The parameters should be such that they can be used for optimization, that is, use log transformation for positive parameters.

bayespy.nodes.Gamma.set_plotter

`Gamma.set_plotter(plotter)`

bayespy.nodes.Gamma.show

`Gamma.show()`

Print the distribution using standard parameterization.

bayespy.nodes.Gamma.unobserve

`Gamma.unobserve()`

bayespy.nodes.Gamma.update

`Gamma.update(annealing=1.0)`

Attributes

dims	
plates	
plates_multiplier	Plate multiplier is applied to messages to parents

bayespy.nodes.Gamma.dims

Gamma.**dims** = ((), ())

bayespy.nodes.Gamma.plates

Gamma.**plates** = None

bayespy.nodes.Gamma.plates_multiplier

Gamma.**plates_multiplier**
Plate multiplier is applied to messages to parents

bayespy.nodes.Wishart

class bayespy.nodes.**Wishart**(*n*, *V*, ***kwargs*)
Node for Wishart random variables.

The random variable $\mathbf{\Lambda}$ is a $D \times D$ positive-definite symmetric matrix.

$$p(\mathbf{\Lambda}) = \mathrm{Wishart}(\mathbf{\Lambda}|N, \mathbf{V})$$

Parameters **n** : scalar or array

N, degrees of freedom, $N > D - 1$.

V : Wishart-like node or (...,D,D)-array

$\mathbf{V}$, scale matrix.

__init__(*n*, *V*, ***kwargs*)
Create Wishart node.

Methods

__init__(n, V, **kwargs)	Create Wishart node.
add_plate_axis(to_plate)	
broadcasting_multiplier(plates, *args)	
delete()	Delete this node and the children
get_gradient(rg)	Computes gradient with respect to the natural parameters.
get_mask()	
get_moments()	
get_parameters()	Return parameters of the VB distribution.
get_riemannian_gradient()	Computes the Riemannian/natural gradient.
get_shape(ind)	
has_plotter()	Return True if the node has a plotter

Continued on next page

Table 5.10 – continued from previous page

initialize_from_parameters(*args)	
initialize_from_prior()	
initialize_from_random()	Set the variable to a random sample from the current distribution.
initialize_from_value(x, *args)	
load(group)	Load the state of the node from a HDF5 file.
logpdf(X[, mask])	Compute the log probability density function Q(X) of this node.
lower_bound_contribution([gradient, ...])	Compute E[log p(X|parents) - log q(X)]
lowerbound()	
move_plates(from_plate, to_plate)	
observe(x, *args[, mask])	Fix moments, compute f and propagate mask.
pdf(X[, mask])	Compute the probability density function of this node.
plot([fig])	Plot the node distribution using the plotter of the node
random()	Draw a random sample from the distribution.
save(group)	Save the state of the node into a HDF5 file.
set_parameters(x)	Set the parameters of the VB distribution.
set_plotter(plotter)	
show()	
unobserve()	
update([annealing])	

bayespy.nodes.Wishart.__init__

Wishart.**__init__**(*n*, *V*, ***kwargs*)

Create Wishart node.

bayespy.nodes.Wishart.add_plate_axis

Wishart.**add_plate_axis**(*to_plate*)

bayespy.nodes.Wishart.broadcasting_multiplier

Wishart.**broadcasting_multiplier**(*plates*, **args*)

bayespy.nodes.Wishart.delete

Wishart.**delete**()

Delete this node and the children

bayespy.nodes.Wishart.get_gradient

Wishart.**get_gradient**(*rg*)

Computes gradient with respect to the natural parameters.

The function takes the Riemannian gradient as an input. This is for three reasons: 1) You probably want to use the Riemannian gradient anyway so this helps avoiding accidental use of this function. 2) The gradient is computed by using the Riemannian gradient and chain rules. 3) Probably you need both Riemannian and normal gradients anyway so you can provide it to this function to avoid re-computing it.

bayespy.nodes.Wishart.get_mask

Wishart.**get_mask**()

bayespy.nodes.Wishart.get_moments

Wishart.**get_moments**()

bayespy.nodes.Wishart.get_parameters

Wishart.**get_parameters**()

Return parameters of the VB distribution.

The parameters should be such that they can be used for optimization, that is, use log transformation for positive parameters.

bayespy.nodes.Wishart.get_riemannian_gradient

Wishart.**get_riemannian_gradient**()

Computes the Riemannian/natural gradient.

bayespy.nodes.Wishart.get_shape

Wishart.**get_shape**(*ind*)

bayespy.nodes.Wishart.has_plotter

Wishart.**has_plotter**()

Return True if the node has a plotter

bayespy.nodes.Wishart.initialize_from_parameters

Wishart.**initialize_from_parameters**(**args*)

bayespy.nodes.Wishart.initialize_from_prior

Wishart.**initialize_from_prior**()

bayespy.nodes.Wishart.initialize_from_random

Wishart.**initialize_from_random**()

Set the variable to a random sample from the current distribution.

bayespy.nodes.Wishart.initialize_from_value

Wishart.**initialize_from_value**(*x*, **args*)

bayespy.nodes.Wishart.load

`Wishart.load(group)`
Load the state of the node from a HDF5 file.

bayespy.nodes.Wishart.logpdf

`Wishart.logpdf(X, mask=True)`
Compute the log probability density function Q(X) of this node.

bayespy.nodes.Wishart.lower_bound_contribution

`Wishart.lower_bound_contribution(gradient=False, ignore_masked=True)`
Compute E[log p(X|parents) - log q(X)]

If deterministic annealing is used, the term E[-log q(X)] is divided by the anneling coefficient. That is, phi and cgf of q are multiplied by the temperature (inverse annealing coefficient).

bayespy.nodes.Wishart.lowerbound

`Wishart.lowerbound()`

bayespy.nodes.Wishart.move_plates

`Wishart.move_plates(from_plate, to_plate)`

bayespy.nodes.Wishart.observe

`Wishart.observe(x, *args, mask=True)`
Fix moments, compute f and propagate mask.

bayespy.nodes.Wishart.pdf

`Wishart.pdf(X, mask=True)`
Compute the probability density function of this node.

bayespy.nodes.Wishart.plot

`Wishart.plot(fig=None, **kwargs)`
Plot the node distribution using the plotter of the node

Because the distributions are in general very difficult to plot, the user must specify some functions which performs the plotting as wanted. See, for instance, bayespy.plot.plotting for available plotters, that is, functions that perform plotting for a node.

bayespy.nodes.Wishart.random

`Wishart.random()`
Draw a random sample from the distribution.

bayespy.nodes.Wishart.save

`Wishart.save(group)`
Save the state of the node into a HDF5 file.

group can be the root

bayespy.nodes.Wishart.set_parameters

`Wishart.set_parameters(x)`
Set the parameters of the VB distribution.

The parameters should be such that they can be used for optimization, that is, use log transformation for positive parameters.

bayespy.nodes.Wishart.set_plotter

`Wishart.set_plotter(plotter)`

bayespy.nodes.Wishart.show

`Wishart.show()`

bayespy.nodes.Wishart.unobserve

`Wishart.unobserve()`

bayespy.nodes.Wishart.update

`Wishart.update(annealing=1.0)`

Attributes

dims	
plates	
plates_multiplier	Plate multiplier is applied to messages to parents

bayespy.nodes.Wishart.dims

`Wishart.dims = None`

bayespy.nodes.Wishart.plates

Wishart.**plates** = None

bayespy.nodes.Wishart.plates_multiplier

Wishart.**plates_multiplier**
Plate multiplier is applied to messages to parents

bayespy.nodes.Exponential

class bayespy.nodes.**Exponential**(*l*, ***kwargs*)
Node for exponential random variables.

> **Warning:** Use Gamma instead of this. *Exponential(l)* is equivalent to *Gamma(1, l)*.

Parameters l : gamma-like node or scalar or array

Rate parameter

See also:

Gamma, Poisson

Notes

For simplicity, this is just a gamma node with the first parent fixed to one. Note that this is a bit inconsistent with the BayesPy philosophy which states that the node does not only define the form of the prior distribution but more importantly the form of the posterior approximation. Thus, one might expect that this node would have exponential posterior distribution approximation. However, it has a gamma distribution. Also, the moments are gamma moments although only E[x] would be the moment of a exponential random variable. All this was done because: a) gamma was already implemented, so there was no need to implement anything, and b) people might easily use Exponential node as a prior definition and expect to get gamma posterior (which is what happens now). Maybe some day a pure Exponential node is implemented and the users are advised to use Gamma(1,b) if they want to use an exponential prior distribution but gamma posterior approximation.

__init__(*l*, ***kwargs*)

Methods

__init__(l, **kwargs)	
add_plate_axis(to_plate)	
as_diagonal_wishart()	
broadcasting_multiplier(plates, *args)	
delete()	Delete this node and the children
get_gradient(rg)	Computes gradient with respect to the natural parameters.
get_mask()	
get_moments()	
get_parameters()	Return parameters of the VB distribution.
get_riemannian_gradient()	Computes the Riemannian/natural gradient.
get_shape(ind)	

Continued on next page

Table 5.12 – continued from previous page

has_plotter()	Return True if the node has a plotter
initialize_from_parameters(*args)	
initialize_from_prior()	
initialize_from_random()	Set the variable to a random sample from the current distribution.
initialize_from_value(x, *args)	
load(group)	Load the state of the node from a HDF5 file.
logpdf(X[, mask])	Compute the log probability density function Q(X) of this node.
lower_bound_contribution([gradient, ...])	Compute E[log p(X|parents) - log q(X)]
lowerbound()	
move_plates(from_plate, to_plate)	
observe(x, *args[, mask])	Fix moments, compute f and propagate mask.
pdf(X[, mask])	Compute the probability density function of this node.
plot([fig])	Plot the node distribution using the plotter of the node
random()	Draw a random sample from the distribution.
save(group)	Save the state of the node into a HDF5 file.
set_parameters(x)	Set the parameters of the VB distribution.
set_plotter(plotter)	
show()	Print the distribution using standard parameterization.
unobserve()	
update([annealing])	

bayespy.nodes.Exponential.__init__

Exponential.**__init__**(*l*, ***kwargs*)

bayespy.nodes.Exponential.add_plate_axis

Exponential.**add_plate_axis**(*to_plate*)

bayespy.nodes.Exponential.as_diagonal_wishart

Exponential.**as_diagonal_wishart**()

bayespy.nodes.Exponential.broadcasting_multiplier

Exponential.**broadcasting_multiplier**(*plates*, **args*)

bayespy.nodes.Exponential.delete

Exponential.**delete**()

Delete this node and the children

bayespy.nodes.Exponential.get_gradient

Exponential.**get_gradient**(*rg*)

Computes gradient with respect to the natural parameters.

The function takes the Riemannian gradient as an input. This is for three reasons: 1) You probably want to use the Riemannian gradient anyway so this helps avoiding accidental use of this function. 2) The gradient is computed by using the Riemannian gradient and chain rules. 3) Probably you need both Riemannian and normal gradients anyway so you can provide it to this function to avoid re-computing it.

bayespy.nodes.Exponential.get_mask

Exponential.**get_mask**()

bayespy.nodes.Exponential.get_moments

Exponential.**get_moments**()

bayespy.nodes.Exponential.get_parameters

Exponential.**get_parameters**()

Return parameters of the VB distribution.

The parameters should be such that they can be used for optimization, that is, use log transformation for positive parameters.

bayespy.nodes.Exponential.get_riemannian_gradient

Exponential.**get_riemannian_gradient**()

Computes the Riemannian/natural gradient.

bayespy.nodes.Exponential.get_shape

Exponential.**get_shape**(*ind*)

bayespy.nodes.Exponential.has_plotter

Exponential.**has_plotter**()

Return True if the node has a plotter

bayespy.nodes.Exponential.initialize_from_parameters

Exponential.**initialize_from_parameters**(**args*)

bayespy.nodes.Exponential.initialize_from_prior

Exponential.**initialize_from_prior**()

bayespy.nodes.Exponential.initialize_from_random

Exponential.**initialize_from_random**()

Set the variable to a random sample from the current distribution.

bayespy.nodes.Exponential.initialize_from_value

Exponential.**initialize_from_value**(*x*, **args*)

bayespy.nodes.Exponential.load

Exponential.**load**(*group*)
Load the state of the node from a HDF5 file.

bayespy.nodes.Exponential.logpdf

Exponential.**logpdf**(*X*, *mask=True*)
Compute the log probability density function Q(X) of this node.

bayespy.nodes.Exponential.lower_bound_contribution

Exponential.**lower_bound_contribution**(*gradient=False*, *ignore_masked=True*)
Compute E[log p(X|parents) - log q(X)]

If deterministic annealing is used, the term E[-log q(X)] is divided by the anneling coefficient. That is, phi and cgf of q are multiplied by the temperature (inverse annealing coefficient).

bayespy.nodes.Exponential.lowerbound

Exponential.**lowerbound**()

bayespy.nodes.Exponential.move_plates

Exponential.**move_plates**(*from_plate*, *to_plate*)

bayespy.nodes.Exponential.observe

Exponential.**observe**(*x*, **args*, *mask=True*)
Fix moments, compute f and propagate mask.

bayespy.nodes.Exponential.pdf

Exponential.**pdf**(*X*, *mask=True*)
Compute the probability density function of this node.

bayespy.nodes.Exponential.plot

Exponential.**plot**(*fig=None*, ***kwargs*)
Plot the node distribution using the plotter of the node

Because the distributions are in general very difficult to plot, the user must specify some functions which performs the plotting as wanted. See, for instance, bayespy.plot.plotting for available plotters, that is, functions that perform plotting for a node.

bayespy.nodes.Exponential.random

`Exponential.random()`
Draw a random sample from the distribution.

bayespy.nodes.Exponential.save

`Exponential.save(group)`
Save the state of the node into a HDF5 file.

group can be the root

bayespy.nodes.Exponential.set_parameters

`Exponential.set_parameters(x)`
Set the parameters of the VB distribution.

The parameters should be such that they can be used for optimization, that is, use log transformation for positive parameters.

bayespy.nodes.Exponential.set_plotter

`Exponential.set_plotter(plotter)`

bayespy.nodes.Exponential.show

`Exponential.show()`
Print the distribution using standard parameterization.

bayespy.nodes.Exponential.unobserve

`Exponential.unobserve()`

bayespy.nodes.Exponential.update

`Exponential.update(annealing=1.0)`

Attributes

dims	
plates	
plates_multiplier	Plate multiplier is applied to messages to parents

bayespy.nodes.Exponential.dims

`Exponential.dims = ((), ())`

bayespy.nodes.Exponential.plates

`Exponential.plates` = None

bayespy.nodes.Exponential.plates_multiplier

`Exponential.plates_multiplier`
Plate multiplier is applied to messages to parents

Nodes for modelling Gaussian and precision variables jointly (useful as prior for Gaussian nodes):

GaussianGammaISO(*args, **kwargs)	Node for Gaussian-gamma (isotropic) random variables.
GaussianGammaARD(mu, alpha, a, b, **kwargs)	Node for Gaussian and gamma random variables with ARD form.
GaussianWishart(*args, **kwargs)	Node for Gaussian-Wishart random variables.

bayespy.nodes.GaussianGammaISO

class `bayespy.nodes.GaussianGammaISO`(**args*, ***kwargs*)
Node for Gaussian-gamma (isotropic) random variables.

The prior:

$$p(x, \alpha|\mu, \Lambda, a, b)$$
$$p(x|\alpha, \mu, \Lambda) = \mathcal{N}(x|\mu, \alpha Lambda)$$
$$p(\alpha|a, b) = \mathcal{G}(\alpha|a, b)$$

The posterior approximation $q(x, \alpha)$ has the same Gaussian-gamma form.

Currently, supports only vector variables.

`__init__`(**args*, ***kwargs*)

Methods

__init__(*args, **kwargs)	
add_plate_axis(to_plate)	
broadcasting_multiplier(plates, *args)	
delete()	Delete this node and the children
get_gaussian_mean_and_variance()	Return the mean and variance of the distribution
get_gradient(rg)	Computes gradient with respect to the natural parameters.
get_marginal_logpdf([gaussian, gamma])	Get the (marginal) log pdf of a subset of the variables
get_mask()	
get_moments()	
get_parameters()	Return parameters of the VB distribution.
get_riemannian_gradient()	Computes the Riemannian/natural gradient.
get_shape(ind)	
has_plotter()	Return True if the node has a plotter
initialize_from_parameters(*args)	
initialize_from_prior()	
initialize_from_random()	Set the variable to a random sample from the current distribution.
initialize_from_value(x, *args)	

Continued on next page

Table 5.15 – continued from previous page

load(group)	Load the state of the node from a HDF5 file.
logpdf(X[, mask])	Compute the log probability density function Q(X) of this node.
lower_bound_contribution([gradient, ...])	Compute E[log p(X|parents) - log q(X)]
lowerbound()	
move_plates(from_plate, to_plate)	
observe(x, *args[, mask])	Fix moments, compute f and propagate mask.
pdf(X[, mask])	Compute the probability density function of this node.
plot([fig])	Plot the node distribution using the plotter of the node
plotmatrix()	Creates a matrix of marginal plots.
random()	Draw a random sample from the distribution.
save(group)	Save the state of the node into a HDF5 file.
set_parameters(x)	Set the parameters of the VB distribution.
set_plotter(plotter)	
show()	Print the distribution using standard parameterization.
unobserve()	
update([annealing])	

bayespy.nodes.GaussianGammaISO.__init__

GaussianGammaISO.**__init__**(**args*, ***kwargs*)

bayespy.nodes.GaussianGammaISO.add_plate_axis

GaussianGammaISO.**add_plate_axis**(*to_plate*)

bayespy.nodes.GaussianGammaISO.broadcasting_multiplier

GaussianGammaISO.**broadcasting_multiplier**(*plates*, **args*)

bayespy.nodes.GaussianGammaISO.delete

GaussianGammaISO.**delete**()

Delete this node and the children

bayespy.nodes.GaussianGammaISO.get_gaussian_mean_and_variance

GaussianGammaISO.**get_gaussian_mean_and_variance**()

Return the mean and variance of the distribution

bayespy.nodes.GaussianGammaISO.get_gradient

GaussianGammaISO.**get_gradient**(*rg*)

Computes gradient with respect to the natural parameters.

The function takes the Riemannian gradient as an input. This is for three reasons: 1) You probably want to use the Riemannian gradient anyway so this helps avoiding accidental use of this function. 2) The gradient is computed by using the Riemannian gradient and chain rules. 3) Probably you need both Riemannian and normal gradients anyway so you can provide it to this function to avoid re-computing it.

bayespy.nodes.GaussianGammaISO.get_marginal_logpdf

`GaussianGammaISO.get_marginal_logpdf(gaussian=None, gamma=None)`

Get the (marginal) log pdf of a subset of the variables

Parameters gaussian : list or None

Indices of the Gaussian variables to keep or None

gamma : bool or None

True if keep the gamma variable, otherwise False or None

Returns function

A function which computes log-pdf

bayespy.nodes.GaussianGammaISO.get_mask

`GaussianGammaISO.get_mask()`

bayespy.nodes.GaussianGammaISO.get_moments

`GaussianGammaISO.get_moments()`

bayespy.nodes.GaussianGammaISO.get_parameters

`GaussianGammaISO.get_parameters()`

Return parameters of the VB distribution.

The parameters should be such that they can be used for optimization, that is, use log transformation for positive parameters.

bayespy.nodes.GaussianGammaISO.get_riemannian_gradient

`GaussianGammaISO.get_riemannian_gradient()`

Computes the Riemannian/natural gradient.

bayespy.nodes.GaussianGammaISO.get_shape

`GaussianGammaISO.get_shape(ind)`

bayespy.nodes.GaussianGammaISO.has_plotter

`GaussianGammaISO.has_plotter()`

Return True if the node has a plotter

bayespy.nodes.GaussianGammaISO.initialize_from_parameters

`GaussianGammaISO.initialize_from_parameters(*args)`

bayespy.nodes.GaussianGammaISO.initialize_from_prior

`GaussianGammaISO.initialize_from_prior()`

bayespy.nodes.GaussianGammaISO.initialize_from_random

`GaussianGammaISO.initialize_from_random()`
Set the variable to a random sample from the current distribution.

bayespy.nodes.GaussianGammaISO.initialize_from_value

`GaussianGammaISO.initialize_from_value(x, *args)`

bayespy.nodes.GaussianGammaISO.load

`GaussianGammaISO.load(group)`
Load the state of the node from a HDF5 file.

bayespy.nodes.GaussianGammaISO.logpdf

`GaussianGammaISO.logpdf(X, mask=True)`
Compute the log probability density function Q(X) of this node.

bayespy.nodes.GaussianGammaISO.lower_bound_contribution

`GaussianGammaISO.lower_bound_contribution(gradient=False, ignore_masked=True)`
Compute E[log p(X|parents) - log q(X)]

If deterministic annealing is used, the term E[-log q(X)] is divided by the anneling coefficient. That is, phi and cgf of q are multiplied by the temperature (inverse annealing coefficient).

bayespy.nodes.GaussianGammaISO.lowerbound

`GaussianGammaISO.lowerbound()`

bayespy.nodes.GaussianGammaISO.move_plates

`GaussianGammaISO.move_plates(from_plate, to_plate)`

bayespy.nodes.GaussianGammaISO.observe

`GaussianGammaISO.observe(x, *args, mask=True)`
Fix moments, compute f and propagate mask.

bayespy.nodes.GaussianGammaISO.pdf

GaussianGammaISO.**pdf**(*X*, *mask=True*)
Compute the probability density function of this node.

bayespy.nodes.GaussianGammaISO.plot

GaussianGammaISO.**plot**(*fig=None*, ***kwargs*)
Plot the node distribution using the plotter of the node

Because the distributions are in general very difficult to plot, the user must specify some functions which performs the plotting as wanted. See, for instance, bayespy.plot.plotting for available plotters, that is, functions that perform plotting for a node.

bayespy.nodes.GaussianGammaISO.plotmatrix

GaussianGammaISO.**plotmatrix**()
Creates a matrix of marginal plots.

On diagonal, are marginal plots of each variable. Off-diagonal plot (i,j) shows the joint marginal density of x_i and x_j.

bayespy.nodes.GaussianGammaISO.random

GaussianGammaISO.**random**()
Draw a random sample from the distribution.

bayespy.nodes.GaussianGammaISO.save

GaussianGammaISO.**save**(*group*)
Save the state of the node into a HDF5 file.

group can be the root

bayespy.nodes.GaussianGammaISO.set_parameters

GaussianGammaISO.**set_parameters**(*x*)
Set the parameters of the VB distribution.

The parameters should be such that they can be used for optimization, that is, use log transformation for positive parameters.

bayespy.nodes.GaussianGammaISO.set_plotter

GaussianGammaISO.**set_plotter**(*plotter*)

bayespy.nodes.GaussianGammaISO.show

GaussianGammaISO.**show**()
Print the distribution using standard parameterization.

bayespy.nodes.GaussianGammaISO.unobserve

GaussianGammaISO.**unobserve**()

bayespy.nodes.GaussianGammaISO.update

GaussianGammaISO.**update**(*annealing=1.0*)

Attributes

dims	
plates	
plates_multiplier	Plate multiplier is applied to messages to parents

bayespy.nodes.GaussianGammaISO.dims

GaussianGammaISO.**dims** = None

bayespy.nodes.GaussianGammaISO.plates

GaussianGammaISO.**plates** = None

bayespy.nodes.GaussianGammaISO.plates_multiplier

GaussianGammaISO.**plates_multiplier**
Plate multiplier is applied to messages to parents

bayespy.nodes.GaussianGammaARD

class bayespy.nodes.**GaussianGammaARD**(*mu*, *alpha*, *a*, *b*, ***kwargs*)
Node for Gaussian and gamma random variables with ARD form.

The prior:

$$p(x, \tau|\mu, \alpha, a, b) = p(x|\tau, \mu, \alpha)p(\tau|a, b)$$
$$p(x|\alpha, \mu, \alpha) = \mathcal{N}(x|\mu, \mathrm{diag}(\boldsymbol{\alpha\tau}))$$
$$p(\tau|a, b) = \mathcal{G}(\tau|a, b)$$

The posterior approximation $q(x, \tau)$ has the same Gaussian-gamma form.

Warning: Not yet implemented.

See also:

`Gaussian`, `GaussianARD`, `Gamma`, `GaussianGammaISO`, `GaussianWishart`

`__init__`(*mu*, *alpha*, *a*, *b*, ***kwargs*)

Methods

`__init__`(mu, alpha, a, b, **kwargs)	
`add_plate_axis`(to_plate)	
`broadcasting_multiplier`(plates, *args)	
`delete`()	Delete this node and the children
`get_gradient`(rg)	Computes gradient with respect to the natural parameters.
`get_mask`()	
`get_moments`()	
`get_parameters`()	Return parameters of the VB distribution.
`get_riemannian_gradient`()	Computes the Riemannian/natural gradient.
`get_shape`(ind)	
`has_plotter`()	Return True if the node has a plotter
`initialize_from_parameters`(*args)	
`initialize_from_prior`()	
`initialize_from_random`()	Set the variable to a random sample from the current distribution.
`initialize_from_value`(x, *args)	
`load`(group)	Load the state of the node from a HDF5 file.
`logpdf`(X[, mask])	Compute the log probability density function Q(X) of this node.
`lower_bound_contribution`([gradient, ...])	Compute E[log p(X|parents) - log q(X)]
`lowerbound`()	
`move_plates`(from_plate, to_plate)	
`observe`(x, *args[, mask])	Fix moments, compute f and propagate mask.
`pdf`(X[, mask])	Compute the probability density function of this node.
`plot`([fig])	Plot the node distribution using the plotter of the node
`random`()	Draw a random sample from the distribution.
`save`(group)	Save the state of the node into a HDF5 file.
`set_parameters`(x)	Set the parameters of the VB distribution.
`set_plotter`(plotter)	
`unobserve`()	
`update`([annealing])	

bayespy.nodes.GaussianGammaARD.__init__

`GaussianGammaARD.__init__`(*mu*, *alpha*, *a*, *b*, ***kwargs*)

bayespy.nodes.GaussianGammaARD.add_plate_axis

`GaussianGammaARD.add_plate_axis`(*to_plate*)

bayespy.nodes.GaussianGammaARD.broadcasting_multiplier

`GaussianGammaARD.broadcasting_multiplier`(*plates*, **args*)

bayespy.nodes.GaussianGammaARD.delete

`GaussianGammaARD.delete()`
Delete this node and the children

bayespy.nodes.GaussianGammaARD.get_gradient

`GaussianGammaARD.get_gradient(rg)`
Computes gradient with respect to the natural parameters.

The function takes the Riemannian gradient as an input. This is for three reasons: 1) You probably want to use the Riemannian gradient anyway so this helps avoiding accidental use of this function. 2) The gradient is computed by using the Riemannian gradient and chain rules. 3) Probably you need both Riemannian and normal gradients anyway so you can provide it to this function to avoid re-computing it.

bayespy.nodes.GaussianGammaARD.get_mask

`GaussianGammaARD.get_mask()`

bayespy.nodes.GaussianGammaARD.get_moments

`GaussianGammaARD.get_moments()`

bayespy.nodes.GaussianGammaARD.get_parameters

`GaussianGammaARD.get_parameters()`
Return parameters of the VB distribution.

The parameters should be such that they can be used for optimization, that is, use log transformation for positive parameters.

bayespy.nodes.GaussianGammaARD.get_riemannian_gradient

`GaussianGammaARD.get_riemannian_gradient()`
Computes the Riemannian/natural gradient.

bayespy.nodes.GaussianGammaARD.get_shape

`GaussianGammaARD.get_shape(ind)`

bayespy.nodes.GaussianGammaARD.has_plotter

`GaussianGammaARD.has_plotter()`
Return True if the node has a plotter

bayespy.nodes.GaussianGammaARD.initialize_from_parameters

`GaussianGammaARD.initialize_from_parameters(*args)`

bayespy.nodes.GaussianGammaARD.initialize_from_prior

GaussianGammaARD.**initialize_from_prior**()

bayespy.nodes.GaussianGammaARD.initialize_from_random

GaussianGammaARD.**initialize_from_random**()
Set the variable to a random sample from the current distribution.

bayespy.nodes.GaussianGammaARD.initialize_from_value

GaussianGammaARD.**initialize_from_value**(*x*, **args*)

bayespy.nodes.GaussianGammaARD.load

GaussianGammaARD.**load**(*group*)
Load the state of the node from a HDF5 file.

bayespy.nodes.GaussianGammaARD.logpdf

GaussianGammaARD.**logpdf**(*X*, *mask=True*)
Compute the log probability density function Q(X) of this node.

bayespy.nodes.GaussianGammaARD.lower_bound_contribution

GaussianGammaARD.**lower_bound_contribution**(*gradient=False*, *ignore_masked=True*)
Compute E[log p(X|parents) - log q(X)]

If deterministic annealing is used, the term E[-log q(X)] is divided by the anneling coefficient. That is, phi and cgf of q are multiplied by the temperature (inverse annealing coefficient).

bayespy.nodes.GaussianGammaARD.lowerbound

GaussianGammaARD.**lowerbound**()

bayespy.nodes.GaussianGammaARD.move_plates

GaussianGammaARD.**move_plates**(*from_plate*, *to_plate*)

bayespy.nodes.GaussianGammaARD.observe

GaussianGammaARD.**observe**(*x*, **args*, *mask=True*)
Fix moments, compute f and propagate mask.

bayespy.nodes.GaussianGammaARD.pdf

`GaussianGammaARD.pdf(X, mask=True)`

Compute the probability density function of this node.

bayespy.nodes.GaussianGammaARD.plot

`GaussianGammaARD.plot(fig=None, **kwargs)`

Plot the node distribution using the plotter of the node

Because the distributions are in general very difficult to plot, the user must specify some functions which performs the plotting as wanted. See, for instance, bayespy.plot.plotting for available plotters, that is, functions that perform plotting for a node.

bayespy.nodes.GaussianGammaARD.random

`GaussianGammaARD.random()`

Draw a random sample from the distribution.

bayespy.nodes.GaussianGammaARD.save

`GaussianGammaARD.save(group)`

Save the state of the node into a HDF5 file.

group can be the root

bayespy.nodes.GaussianGammaARD.set_parameters

`GaussianGammaARD.set_parameters(x)`

Set the parameters of the VB distribution.

The parameters should be such that they can be used for optimization, that is, use log transformation for positive parameters.

bayespy.nodes.GaussianGammaARD.set_plotter

`GaussianGammaARD.set_plotter(plotter)`

bayespy.nodes.GaussianGammaARD.unobserve

`GaussianGammaARD.unobserve()`

bayespy.nodes.GaussianGammaARD.update

`GaussianGammaARD.update(annealing=1.0)`

Attributes

dims	
plates	
plates_multiplier	Plate multiplier is applied to messages to parents

bayespy.nodes.GaussianGammaARD.dims

GaussianGammaARD.**dims** = None

bayespy.nodes.GaussianGammaARD.plates

GaussianGammaARD.**plates** = None

bayespy.nodes.GaussianGammaARD.plates_multiplier

GaussianGammaARD.**plates_multiplier**
Plate multiplier is applied to messages to parents

bayespy.nodes.GaussianWishart

class bayespy.nodes.**GaussianWishart**(**args*, ***kwargs*)
Node for Gaussian-Wishart random variables.

The prior:

$$p(x, \Lambda|\mu, \alpha, V, n)$$
$$p(x|\Lambda, \mu, \alpha) = (N)(x|\mu, \alpha^{-1} Lambda^{-1})$$
$$p(\Lambda|V, n) = (W)(\Lambda|n, V)$$

The posterior approximation $q(x, \Lambda)$ has the same Gaussian-Wishart form.

Currently, supports only vector variables.

__init__(**args*, ***kwargs*)

Methods

__init__(*args, **kwargs)	
add_plate_axis(to_plate)	
broadcasting_multiplier(plates, *args)	
delete()	Delete this node and the children
get_gradient(rg)	Computes gradient with respect to the natural parameters.
get_mask()	
get_moments()	
get_parameters()	Return parameters of the VB distribution.
get_riemannian_gradient()	Computes the Riemannian/natural gradient.
get_shape(ind)	
has_plotter()	Return True if the node has a plotter
initialize_from_parameters(*args)	

Continued on next page

Table 5.19 – continued from previous page

initialize_from_prior()	
initialize_from_random()	Set the variable to a random sample from the current distribution.
initialize_from_value(x, *args)	
load(group)	Load the state of the node from a HDF5 file.
logpdf(X[, mask])	Compute the log probability density function Q(X) of this node.
lower_bound_contribution([gradient, ...])	Compute E[log p(X|parents) - log q(X)]
lowerbound()	
move_plates(from_plate, to_plate)	
observe(x, *args[, mask])	Fix moments, compute f and propagate mask.
pdf(X[, mask])	Compute the probability density function of this node.
plot([fig])	Plot the node distribution using the plotter of the node
random()	Draw a random sample from the distribution.
save(group)	Save the state of the node into a HDF5 file.
set_parameters(x)	Set the parameters of the VB distribution.
set_plotter(plotter)	
show()	Print the distribution using standard parameterization.
unobserve()	
update([annealing])	

bayespy.nodes.GaussianWishart.__init__

GaussianWishart.**__init__**(**args*, ***kwargs*)

bayespy.nodes.GaussianWishart.add_plate_axis

GaussianWishart.**add_plate_axis**(*to_plate*)

bayespy.nodes.GaussianWishart.broadcasting_multiplier

GaussianWishart.**broadcasting_multiplier**(*plates*, **args*)

bayespy.nodes.GaussianWishart.delete

GaussianWishart.**delete**()

Delete this node and the children

bayespy.nodes.GaussianWishart.get_gradient

GaussianWishart.**get_gradient**(*rg*)

Computes gradient with respect to the natural parameters.

The function takes the Riemannian gradient as an input. This is for three reasons: 1) You probably want to use the Riemannian gradient anyway so this helps avoiding accidental use of this function. 2) The gradient is computed by using the Riemannian gradient and chain rules. 3) Probably you need both Riemannian and normal gradients anyway so you can provide it to this function to avoid re-computing it.

bayespy.nodes.GaussianWishart.get_mask

GaussianWishart.**get_mask**()

bayespy.nodes.GaussianWishart.get_moments

GaussianWishart.**get_moments**()

bayespy.nodes.GaussianWishart.get_parameters

GaussianWishart.**get_parameters**()

Return parameters of the VB distribution.

The parameters should be such that they can be used for optimization, that is, use log transformation for positive parameters.

bayespy.nodes.GaussianWishart.get_riemannian_gradient

GaussianWishart.**get_riemannian_gradient**()

Computes the Riemannian/natural gradient.

bayespy.nodes.GaussianWishart.get_shape

GaussianWishart.**get_shape**(*ind*)

bayespy.nodes.GaussianWishart.has_plotter

GaussianWishart.**has_plotter**()

Return True if the node has a plotter

bayespy.nodes.GaussianWishart.initialize_from_parameters

GaussianWishart.**initialize_from_parameters**(**args*)

bayespy.nodes.GaussianWishart.initialize_from_prior

GaussianWishart.**initialize_from_prior**()

bayespy.nodes.GaussianWishart.initialize_from_random

GaussianWishart.**initialize_from_random**()

Set the variable to a random sample from the current distribution.

bayespy.nodes.GaussianWishart.initialize_from_value

GaussianWishart.**initialize_from_value**(*x*, **args*)

bayespy.nodes.GaussianWishart.load

GaussianWishart.**load**(*group*)
Load the state of the node from a HDF5 file.

bayespy.nodes.GaussianWishart.logpdf

GaussianWishart.**logpdf**(*X*, *mask=True*)
Compute the log probability density function Q(X) of this node.

bayespy.nodes.GaussianWishart.lower_bound_contribution

GaussianWishart.**lower_bound_contribution**(*gradient=False*, *ignore_masked=True*)
Compute E[log p(X|parents) - log q(X)]

If deterministic annealing is used, the term E[-log q(X)] is divided by the anneling coefficient. That is, phi and cgf of q are multiplied by the temperature (inverse annealing coefficient).

bayespy.nodes.GaussianWishart.lowerbound

GaussianWishart.**lowerbound**()

bayespy.nodes.GaussianWishart.move_plates

GaussianWishart.**move_plates**(*from_plate*, *to_plate*)

bayespy.nodes.GaussianWishart.observe

GaussianWishart.**observe**(*x*, **args*, *mask=True*)
Fix moments, compute f and propagate mask.

bayespy.nodes.GaussianWishart.pdf

GaussianWishart.**pdf**(*X*, *mask=True*)
Compute the probability density function of this node.

bayespy.nodes.GaussianWishart.plot

GaussianWishart.**plot**(*fig=None*, ***kwargs*)
Plot the node distribution using the plotter of the node

Because the distributions are in general very difficult to plot, the user must specify some functions which performs the plotting as wanted. See, for instance, bayespy.plot.plotting for available plotters, that is, functions that perform plotting for a node.

bayespy.nodes.GaussianWishart.random

`GaussianWishart.random()`
Draw a random sample from the distribution.

bayespy.nodes.GaussianWishart.save

`GaussianWishart.save(group)`
Save the state of the node into a HDF5 file.

group can be the root

bayespy.nodes.GaussianWishart.set_parameters

`GaussianWishart.set_parameters(x)`
Set the parameters of the VB distribution.

The parameters should be such that they can be used for optimization, that is, use log transformation for positive parameters.

bayespy.nodes.GaussianWishart.set_plotter

`GaussianWishart.set_plotter(plotter)`

bayespy.nodes.GaussianWishart.show

`GaussianWishart.show()`
Print the distribution using standard parameterization.

bayespy.nodes.GaussianWishart.unobserve

`GaussianWishart.unobserve()`

bayespy.nodes.GaussianWishart.update

`GaussianWishart.update(annealing=1.0)`

Attributes

dims	
plates	
plates_multiplier	Plate multiplier is applied to messages to parents

bayespy.nodes.GaussianWishart.dims

`GaussianWishart.dims = None`

bayespy.nodes.GaussianWishart.plates

GaussianWishart.**plates** = **None**

bayespy.nodes.GaussianWishart.plates_multiplier

GaussianWishart.**plates_multiplier**

Plate multiplier is applied to messages to parents

Nodes for discrete count variables:

Bernoulli(p, **kwargs)	Node for Bernoulli random variables.
Binomial(n, p, **kwargs)	Node for binomial random variables.
Categorical(p, **kwargs)	Node for categorical random variables.
Multinomial(n, p, **kwargs)	Node for multinomial random variables.
Poisson(l, **kwargs)	Node for Poisson random variables.

bayespy.nodes.Bernoulli

class bayespy.nodes.**Bernoulli**(*p*, ***kwargs*)

Node for Bernoulli random variables.

The node models a binary random variable $z \in \{0, 1\}$ with prior probability $p \in [0, 1]$ for value one:

$$z \sim \mathrm{Bernoulli}(p).$$

Parameters p : beta-like node

Probability of a successful trial

Examples

```
from bayespy.nodes import Bernoulli, Beta
p = Beta([1e-3, 1e-3])
z = Bernoulli(p, plates=(10,))
z.observe([0, 1, 1, 1, 0, 1, 1, 1, 0, 1])
p.update()
import bayespy.plot as bpplt
import numpy as np
bpplt.pdf(p, np.linspace(0, 1, num=100))
```

__init__(*p*, ***kwargs*)

Create Bernoulli node.

Methods

__init__(p, **kwargs)	Create Bernoulli node.
add_plate_axis(to_plate)	
broadcasting_multiplier(plates, *args)	
delete()	Delete this node and the children
get_gradient(rg)	Computes gradient with respect to the natural parameters.

Continued on next page

Table 5.22 – continued from previous page

get_mask()	
get_moments()	
get_parameters()	Return parameters of the VB distribution.
get_riemannian_gradient()	Computes the Riemannian/natural gradient.
get_shape(ind)	
has_plotter()	Return True if the node has a plotter
initialize_from_parameters(*args)	
initialize_from_prior()	
initialize_from_random()	Set the variable to a random sample from the current distribution.
initialize_from_value(x, *args)	
load(group)	Load the state of the node from a HDF5 file.
logpdf(X[, mask])	Compute the log probability density function Q(X) of this node.
lower_bound_contribution([gradient, ...])	Compute E[log p(X|parents) - log q(X)]
lowerbound()	
move_plates(from_plate, to_plate)	
observe(x, *args[, mask])	Fix moments, compute f and propagate mask.
pdf(X[, mask])	Compute the probability density function of this node.
plot([fig])	Plot the node distribution using the plotter of the node
random()	Draw a random sample from the distribution.
save(group)	Save the state of the node into a HDF5 file.
set_parameters(x)	Set the parameters of the VB distribution.
set_plotter(plotter)	
show()	Print the distribution using standard parameterization.
unobserve()	
update([annealing])	

bayespy.nodes.Bernoulli.__init__

Bernoulli.**__init__**(*p*, ***kwargs*)

Create Bernoulli node.

bayespy.nodes.Bernoulli.add_plate_axis

Bernoulli.**add_plate_axis**(*to_plate*)

bayespy.nodes.Bernoulli.broadcasting_multiplier

Bernoulli.**broadcasting_multiplier**(*plates*, **args*)

bayespy.nodes.Bernoulli.delete

Bernoulli.**delete**()

Delete this node and the children

bayespy.nodes.Bernoulli.get_gradient

Bernoulli.**get_gradient**(*rg*)

Computes gradient with respect to the natural parameters.

The function takes the Riemannian gradient as an input. This is for three reasons: 1) You probably want to use the Riemannian gradient anyway so this helps avoiding accidental use of this function. 2) The gradient is computed by using the Riemannian gradient and chain rules. 3) Probably you need both Riemannian and normal gradients anyway so you can provide it to this function to avoid re-computing it.

bayespy.nodes.Bernoulli.get_mask

`Bernoulli.get_mask()`

bayespy.nodes.Bernoulli.get_moments

`Bernoulli.get_moments()`

bayespy.nodes.Bernoulli.get_parameters

`Bernoulli.get_parameters()`

Return parameters of the VB distribution.

The parameters should be such that they can be used for optimization, that is, use log transformation for positive parameters.

bayespy.nodes.Bernoulli.get_riemannian_gradient

`Bernoulli.get_riemannian_gradient()`

Computes the Riemannian/natural gradient.

bayespy.nodes.Bernoulli.get_shape

`Bernoulli.get_shape(ind)`

bayespy.nodes.Bernoulli.has_plotter

`Bernoulli.has_plotter()`

Return True if the node has a plotter

bayespy.nodes.Bernoulli.initialize_from_parameters

`Bernoulli.initialize_from_parameters(*args)`

bayespy.nodes.Bernoulli.initialize_from_prior

`Bernoulli.initialize_from_prior()`

bayespy.nodes.Bernoulli.initialize_from_random

`Bernoulli.initialize_from_random()`

Set the variable to a random sample from the current distribution.

bayespy.nodes.Bernoulli.initialize_from_value

`Bernoulli.initialize_from_value(x, *args)`

bayespy.nodes.Bernoulli.load

`Bernoulli.load(group)`
Load the state of the node from a HDF5 file.

bayespy.nodes.Bernoulli.logpdf

`Bernoulli.logpdf(X, mask=True)`
Compute the log probability density function Q(X) of this node.

bayespy.nodes.Bernoulli.lower_bound_contribution

`Bernoulli.lower_bound_contribution(gradient=False, ignore_masked=True)`
Compute E[log p(X|parents) - log q(X)]

If deterministic annealing is used, the term E[-log q(X)] is divided by the anneling coefficient. That is, phi and cgf of q are multiplied by the temperature (inverse annealing coefficient).

bayespy.nodes.Bernoulli.lowerbound

`Bernoulli.lowerbound()`

bayespy.nodes.Bernoulli.move_plates

`Bernoulli.move_plates(from_plate, to_plate)`

bayespy.nodes.Bernoulli.observe

`Bernoulli.observe(x, *args, mask=True)`
Fix moments, compute f and propagate mask.

bayespy.nodes.Bernoulli.pdf

`Bernoulli.pdf(X, mask=True)`
Compute the probability density function of this node.

bayespy.nodes.Bernoulli.plot

`Bernoulli.plot(fig=None, **kwargs)`
Plot the node distribution using the plotter of the node

Because the distributions are in general very difficult to plot, the user must specify some functions which performs the plotting as wanted. See, for instance, bayespy.plot.plotting for available plotters, that is, functions that perform plotting for a node.

bayespy.nodes.Bernoulli.random

`Bernoulli.random()`
Draw a random sample from the distribution.

bayespy.nodes.Bernoulli.save

`Bernoulli.save(group)`
Save the state of the node into a HDF5 file.

group can be the root

bayespy.nodes.Bernoulli.set_parameters

`Bernoulli.set_parameters(x)`
Set the parameters of the VB distribution.

The parameters should be such that they can be used for optimization, that is, use log transformation for positive parameters.

bayespy.nodes.Bernoulli.set_plotter

`Bernoulli.set_plotter(plotter)`

bayespy.nodes.Bernoulli.show

`Bernoulli.show()`
Print the distribution using standard parameterization.

bayespy.nodes.Bernoulli.unobserve

`Bernoulli.unobserve()`

bayespy.nodes.Bernoulli.update

`Bernoulli.update(annealing=1.0)`

Attributes

dims	
plates	
plates_multiplier	Plate multiplier is applied to messages to parents

bayespy.nodes.Bernoulli.dims

`Bernoulli.dims` = None

bayespy.nodes.Bernoulli.plates

`Bernoulli.plates = None`

bayespy.nodes.Bernoulli.plates_multiplier

`Bernoulli.plates_multiplier`
Plate multiplier is applied to messages to parents

bayespy.nodes.Binomial

class `bayespy.nodes.Binomial(n, p, **kwargs)`
Node for binomial random variables.

The node models the number of successes $x \in \{0, \ldots, n\}$ in n trials with probability p for success:

$$x \sim \mathrm{Binomial}(n, p).$$

Parameters **n** : scalar or array

Number of trials

p : beta-like node or scalar or array

Probability of a success in a trial

See also:

Bernoulli, Multinomial, Beta

Examples

```
from bayespy.nodes import Binomial, Beta
p = Beta([1e-3, 1e-3])
x = Binomial(10, p)
x.observe(7)
p.update()
import bayespy.plot as bpplt
import numpy as np
bpplt.pdf(p, np.linspace(0, 1, num=100))
```

`__init__(n, p, **kwargs)`
Create binomial node

Methods

`__init__`(n, p, **kwargs)	Create binomial node
`add_plate_axis`(to_plate)	
`broadcasting_multiplier`(plates, *args)	
`delete`()	Delete this node and the children
`get_gradient`(rg)	Computes gradient with respect to the natural parameters.
`get_mask`()	
`get_moments`()	

Continued on next page

Table 5.24 – continued from previous page

get_parameters()	Return parameters of the VB distribution.
get_riemannian_gradient()	Computes the Riemannian/natural gradient.
get_shape(ind)	
has_plotter()	Return True if the node has a plotter
initialize_from_parameters(*args)	
initialize_from_prior()	
initialize_from_random()	Set the variable to a random sample from the current distribution.
initialize_from_value(x, *args)	
load(group)	Load the state of the node from a HDF5 file.
logpdf(X[, mask])	Compute the log probability density function Q(X) of this node.
lower_bound_contribution([gradient, ...])	Compute E[log p(X|parents) - log q(X)]
lowerbound()	
move_plates(from_plate, to_plate)	
observe(x, *args[, mask])	Fix moments, compute f and propagate mask.
pdf(X[, mask])	Compute the probability density function of this node.
plot([fig])	Plot the node distribution using the plotter of the node
random()	Draw a random sample from the distribution.
save(group)	Save the state of the node into a HDF5 file.
set_parameters(x)	Set the parameters of the VB distribution.
set_plotter(plotter)	
show()	Print the distribution using standard parameterization.
unobserve()	
update([annealing])	

bayespy.nodes.Binomial.__init__

Binomial.**__init__**(*n*, *p*, ***kwargs*)
Create binomial node

bayespy.nodes.Binomial.add_plate_axis

Binomial.**add_plate_axis**(*to_plate*)

bayespy.nodes.Binomial.broadcasting_multiplier

Binomial.**broadcasting_multiplier**(*plates*, **args*)

bayespy.nodes.Binomial.delete

Binomial.**delete**()
Delete this node and the children

bayespy.nodes.Binomial.get_gradient

Binomial.**get_gradient**(*rg*)
Computes gradient with respect to the natural parameters.

The function takes the Riemannian gradient as an input. This is for three reasons: 1) You probably want to use the Riemannian gradient anyway so this helps avoiding accidental use of this function. 2) The gradient is computed by using the Riemannian gradient and chain rules. 3) Probably you need both Riemannian and normal gradients anyway so you can provide it to this function to avoid re-computing it.

bayespy.nodes.Binomial.get_mask

`Binomial.get_mask()`

bayespy.nodes.Binomial.get_moments

`Binomial.get_moments()`

bayespy.nodes.Binomial.get_parameters

`Binomial.get_parameters()`

Return parameters of the VB distribution.

The parameters should be such that they can be used for optimization, that is, use log transformation for positive parameters.

bayespy.nodes.Binomial.get_riemannian_gradient

`Binomial.get_riemannian_gradient()`

Computes the Riemannian/natural gradient.

bayespy.nodes.Binomial.get_shape

`Binomial.get_shape(ind)`

bayespy.nodes.Binomial.has_plotter

`Binomial.has_plotter()`

Return True if the node has a plotter

bayespy.nodes.Binomial.initialize_from_parameters

`Binomial.initialize_from_parameters(*args)`

bayespy.nodes.Binomial.initialize_from_prior

`Binomial.initialize_from_prior()`

bayespy.nodes.Binomial.initialize_from_random

`Binomial.initialize_from_random()`

Set the variable to a random sample from the current distribution.

bayespy.nodes.Binomial.initialize_from_value

`Binomial.initialize_from_value(x, *args)`

bayespy.nodes.Binomial.load

`Binomial.load(group)`

Load the state of the node from a HDF5 file.

bayespy.nodes.Binomial.logpdf

`Binomial.logpdf(X, mask=True)`

Compute the log probability density function Q(X) of this node.

bayespy.nodes.Binomial.lower_bound_contribution

`Binomial.lower_bound_contribution(gradient=False, ignore_masked=True)`

Compute E[log p(X|parents) - log q(X)]

If deterministic annealing is used, the term E[-log q(X)] is divided by the anneling coefficient. That is, phi and cgf of q are multiplied by the temperature (inverse annealing coefficient).

bayespy.nodes.Binomial.lowerbound

`Binomial.lowerbound()`

bayespy.nodes.Binomial.move_plates

`Binomial.move_plates(from_plate, to_plate)`

bayespy.nodes.Binomial.observe

`Binomial.observe(x, *args, mask=True)`

Fix moments, compute f and propagate mask.

bayespy.nodes.Binomial.pdf

`Binomial.pdf(X, mask=True)`

Compute the probability density function of this node.

bayespy.nodes.Binomial.plot

`Binomial.plot(fig=None, **kwargs)`

Plot the node distribution using the plotter of the node

Because the distributions are in general very difficult to plot, the user must specify some functions which performs the plotting as wanted. See, for instance, bayespy.plot.plotting for available plotters, that is, functions that perform plotting for a node.

bayespy.nodes.Binomial.random

`Binomial.random()`

Draw a random sample from the distribution.

bayespy.nodes.Binomial.save

`Binomial.save(group)`

Save the state of the node into a HDF5 file.

group can be the root

bayespy.nodes.Binomial.set_parameters

`Binomial.set_parameters(x)`

Set the parameters of the VB distribution.

The parameters should be such that they can be used for optimization, that is, use log transformation for positive parameters.

bayespy.nodes.Binomial.set_plotter

`Binomial.set_plotter(plotter)`

bayespy.nodes.Binomial.show

`Binomial.show()`

Print the distribution using standard parameterization.

bayespy.nodes.Binomial.unobserve

`Binomial.unobserve()`

bayespy.nodes.Binomial.update

`Binomial.update(annealing=1.0)`

Attributes

dims	
plates	
plates_multiplier	Plate multiplier is applied to messages to parents

bayespy.nodes.Binomial.dims

`Binomial.dims = None`

bayespy.nodes.Binomial.plates

`Binomial.plates` = **None**

bayespy.nodes.Binomial.plates_multiplier

`Binomial.plates_multiplier`
Plate multiplier is applied to messages to parents

bayespy.nodes.Categorical

class `bayespy.nodes.Categorical`(*p*, ***kwargs*)
Node for categorical random variables.

The node models a categorical random variable $x \in \{0,\ldots,K-1\}$ with prior probabilities $\{p_0,\ldots,p_{K-1}\}$ for each category:

$$p(x = k) = p_k \quad \text{for } k \in \{0,\ldots,K-1\}.$$

Parameters p : Dirichlet-like node or (...,K)-array

Probabilities for each category

See also:

`Bernoulli`, `Multinomial`, `Dirichlet`

`__init__`(*p*, ***kwargs*)
Create Categorical node.

Methods

`__init__`(p, **kwargs)	Create Categorical node.
`add_plate_axis`(to_plate)	
`broadcasting_multiplier`(plates, *args)	
`delete`()	Delete this node and the children
`get_gradient`(rg)	Computes gradient with respect to the natural parameters.
`get_mask`()	
`get_moments`()	
`get_parameters`()	Return parameters of the VB distribution.
`get_riemannian_gradient`()	Computes the Riemannian/natural gradient.
`get_shape`(ind)	
`has_plotter`()	Return True if the node has a plotter
`initialize_from_parameters`(*args)	
`initialize_from_prior`()	
`initialize_from_random`()	Set the variable to a random sample from the current distribution.
`initialize_from_value`(x, *args)	
`load`(group)	Load the state of the node from a HDF5 file.
`logpdf`(X[, mask])	Compute the log probability density function Q(X) of this node.
`lower_bound_contribution`([gradient, ...])	Compute E[log p(X|parents) - log q(X)]
`lowerbound`()	
`move_plates`(from_plate, to_plate)	
`observe`(x, *args[, mask])	Fix moments, compute f and propagate mask.

Continued on next page

Table 5.26 – continued from previous page

pdf(X[, mask])	Compute the probability density function of this node.
plot([fig])	Plot the node distribution using the plotter of the node
random()	Draw a random sample from the distribution.
save(group)	Save the state of the node into a HDF5 file.
set_parameters(x)	Set the parameters of the VB distribution.
set_plotter(plotter)	
show()	Print the distribution using standard parameterization.
unobserve()	
update([annealing])	

bayespy.nodes.Categorical.__init__

Categorical.**__init__**(*p*, ***kwargs*)

Create Categorical node.

bayespy.nodes.Categorical.add_plate_axis

Categorical.**add_plate_axis**(*to_plate*)

bayespy.nodes.Categorical.broadcasting_multiplier

Categorical.**broadcasting_multiplier**(*plates*, **args*)

bayespy.nodes.Categorical.delete

Categorical.**delete**()

Delete this node and the children

bayespy.nodes.Categorical.get_gradient

Categorical.**get_gradient**(*rg*)

Computes gradient with respect to the natural parameters.

The function takes the Riemannian gradient as an input. This is for three reasons: 1) You probably want to use the Riemannian gradient anyway so this helps avoiding accidental use of this function. 2) The gradient is computed by using the Riemannian gradient and chain rules. 3) Probably you need both Riemannian and normal gradients anyway so you can provide it to this function to avoid re-computing it.

bayespy.nodes.Categorical.get_mask

Categorical.**get_mask**()

bayespy.nodes.Categorical.get_moments

Categorical.**get_moments**()

bayespy.nodes.Categorical.get_parameters

`Categorical.get_parameters()`

Return parameters of the VB distribution.

The parameters should be such that they can be used for optimization, that is, use log transformation for positive parameters.

bayespy.nodes.Categorical.get_riemannian_gradient

`Categorical.get_riemannian_gradient()`

Computes the Riemannian/natural gradient.

bayespy.nodes.Categorical.get_shape

`Categorical.get_shape(ind)`

bayespy.nodes.Categorical.has_plotter

`Categorical.has_plotter()`

Return True if the node has a plotter

bayespy.nodes.Categorical.initialize_from_parameters

`Categorical.initialize_from_parameters(*args)`

bayespy.nodes.Categorical.initialize_from_prior

`Categorical.initialize_from_prior()`

bayespy.nodes.Categorical.initialize_from_random

`Categorical.initialize_from_random()`

Set the variable to a random sample from the current distribution.

bayespy.nodes.Categorical.initialize_from_value

`Categorical.initialize_from_value(x, *args)`

bayespy.nodes.Categorical.load

`Categorical.load(group)`

Load the state of the node from a HDF5 file.

bayespy.nodes.Categorical.logpdf

Categorical.**logpdf**(*X*, *mask=True*)

Compute the log probability density function Q(X) of this node.

bayespy.nodes.Categorical.lower_bound_contribution

Categorical.**lower_bound_contribution**(*gradient=False*, *ignore_masked=True*)

Compute E[log p(X|parents) - log q(X)]

If deterministic annealing is used, the term E[-log q(X)] is divided by the anneling coefficient. That is, phi and cgf of q are multiplied by the temperature (inverse annealing coefficient).

bayespy.nodes.Categorical.lowerbound

Categorical.**lowerbound**()

bayespy.nodes.Categorical.move_plates

Categorical.**move_plates**(*from_plate*, *to_plate*)

bayespy.nodes.Categorical.observe

Categorical.**observe**(*x*, **args*, *mask=True*)

Fix moments, compute f and propagate mask.

bayespy.nodes.Categorical.pdf

Categorical.**pdf**(*X*, *mask=True*)

Compute the probability density function of this node.

bayespy.nodes.Categorical.plot

Categorical.**plot**(*fig=None*, ***kwargs*)

Plot the node distribution using the plotter of the node

Because the distributions are in general very difficult to plot, the user must specify some functions which performs the plotting as wanted. See, for instance, bayespy.plot.plotting for available plotters, that is, functions that perform plotting for a node.

bayespy.nodes.Categorical.random

Categorical.**random**()

Draw a random sample from the distribution.

bayespy.nodes.Categorical.save

`Categorical.save(group)`
Save the state of the node into a HDF5 file.

group can be the root

bayespy.nodes.Categorical.set_parameters

`Categorical.set_parameters(x)`
Set the parameters of the VB distribution.

The parameters should be such that they can be used for optimization, that is, use log transformation for positive parameters.

bayespy.nodes.Categorical.set_plotter

`Categorical.set_plotter(plotter)`

bayespy.nodes.Categorical.show

`Categorical.show()`
Print the distribution using standard parameterization.

bayespy.nodes.Categorical.unobserve

`Categorical.unobserve()`

bayespy.nodes.Categorical.update

`Categorical.update(annealing=1.0)`

Attributes

dims	
plates	
plates_multiplier	Plate multiplier is applied to messages to parents

bayespy.nodes.Categorical.dims

`Categorical.dims = None`

bayespy.nodes.Categorical.plates

`Categorical.plates = None`

bayespy.nodes.Categorical.plates_multiplier

Categorical.**plates_multiplier**
Plate multiplier is applied to messages to parents

bayespy.nodes.Multinomial

class bayespy.nodes.**Multinomial**(*n*, *p*, ***kwargs*)
Node for multinomial random variables.

Assume there are K categories and N trials each of which leads a success for exactly one of the categories. Given the probabilities $p_0, \ldots, p_{K-1}$ for the categories, multinomial distribution is gives the probability of any combination of numbers of successes for the categories.

The node models the number of successes $x_k \in \{0, \ldots, n\}$ in n trials with probability p_k for success in K categories.

$$\mathrm{Multinomial}(\mathbf{x}|N, \mathbf{p}) = \frac{N!}{x_0! \cdots x_{K-1}!} p_0^{x_0} \cdots p_{K-1}^{x_{K-1}}$$

Parameters **n** : scalar or array

N, number of trials

p : Dirichlet-like node or (...,K)-array

$\mathbf{p}$, probabilities of successes for the categories

See also:

Dirichlet, Binomial, Categorical

__init__(*n*, *p*, ***kwargs*)
Create Multinomial node.

Methods

__init__(n, p, **kwargs)	Create Multinomial node.
add_plate_axis(to_plate)	
broadcasting_multiplier(plates, *args)	
delete()	Delete this node and the children
get_gradient(rg)	Computes gradient with respect to the natural parameters.
get_mask()	
get_moments()	
get_parameters()	Return parameters of the VB distribution.
get_riemannian_gradient()	Computes the Riemannian/natural gradient.
get_shape(ind)	
has_plotter()	Return True if the node has a plotter
initialize_from_parameters(*args)	
initialize_from_prior()	
initialize_from_random()	Set the variable to a random sample from the current distribution.
initialize_from_value(x, *args)	
load(group)	Load the state of the node from a HDF5 file.
logpdf(X[, mask])	Compute the log probability density function Q(X) of this node.
lower_bound_contribution([gradient, ...])	Compute E[log p(X|parents) - log q(X)]

Continued on next page

Table 5.28 – continued from previous page

lowerbound()	
move_plates(from_plate, to_plate)	
observe(x, *args[, mask])	Fix moments, compute f and propagate mask.
pdf(X[, mask])	Compute the probability density function of this node.
plot([fig])	Plot the node distribution using the plotter of the node
random()	Draw a random sample from the distribution.
save(group)	Save the state of the node into a HDF5 file.
set_parameters(x)	Set the parameters of the VB distribution.
set_plotter(plotter)	
show()	Print the distribution using standard parameterization.
unobserve()	
update([annealing])	

bayespy.nodes.Multinomial.__init__

Multinomial.**__init__**(*n*, *p*, ***kwargs*)
Create Multinomial node.

bayespy.nodes.Multinomial.add_plate_axis

Multinomial.**add_plate_axis**(*to_plate*)

bayespy.nodes.Multinomial.broadcasting_multiplier

Multinomial.**broadcasting_multiplier**(*plates*, **args*)

bayespy.nodes.Multinomial.delete

Multinomial.**delete**()
Delete this node and the children

bayespy.nodes.Multinomial.get_gradient

Multinomial.**get_gradient**(*rg*)
Computes gradient with respect to the natural parameters.

The function takes the Riemannian gradient as an input. This is for three reasons: 1) You probably want to use the Riemannian gradient anyway so this helps avoiding accidental use of this function. 2) The gradient is computed by using the Riemannian gradient and chain rules. 3) Probably you need both Riemannian and normal gradients anyway so you can provide it to this function to avoid re-computing it.

bayespy.nodes.Multinomial.get_mask

Multinomial.**get_mask**()

bayespy.nodes.Multinomial.get_moments

Multinomial.**get_moments**()

bayespy.nodes.Multinomial.get_parameters

Multinomial.**get_parameters**()

Return parameters of the VB distribution.

The parameters should be such that they can be used for optimization, that is, use log transformation for positive parameters.

bayespy.nodes.Multinomial.get_riemannian_gradient

Multinomial.**get_riemannian_gradient**()

Computes the Riemannian/natural gradient.

bayespy.nodes.Multinomial.get_shape

Multinomial.**get_shape**(*ind*)

bayespy.nodes.Multinomial.has_plotter

Multinomial.**has_plotter**()

Return True if the node has a plotter

bayespy.nodes.Multinomial.initialize_from_parameters

Multinomial.**initialize_from_parameters**(**args*)

bayespy.nodes.Multinomial.initialize_from_prior

Multinomial.**initialize_from_prior**()

bayespy.nodes.Multinomial.initialize_from_random

Multinomial.**initialize_from_random**()

Set the variable to a random sample from the current distribution.

bayespy.nodes.Multinomial.initialize_from_value

Multinomial.**initialize_from_value**(*x*, **args*)

bayespy.nodes.Multinomial.load

Multinomial.**load**(*group*)

Load the state of the node from a HDF5 file.

bayespy.nodes.Multinomial.logpdf

Multinomial.**logpdf**(*X*, *mask=True*)
Compute the log probability density function Q(X) of this node.

bayespy.nodes.Multinomial.lower_bound_contribution

Multinomial.**lower_bound_contribution**(*gradient=False*, *ignore_masked=True*)
Compute E[log p(X|parents) - log q(X)]

If deterministic annealing is used, the term E[-log q(X)] is divided by the anneling coefficient. That is, phi and cgf of q are multiplied by the temperature (inverse annealing coefficient).

bayespy.nodes.Multinomial.lowerbound

Multinomial.**lowerbound**()

bayespy.nodes.Multinomial.move_plates

Multinomial.**move_plates**(*from_plate*, *to_plate*)

bayespy.nodes.Multinomial.observe

Multinomial.**observe**(*x*, **args*, *mask=True*)
Fix moments, compute f and propagate mask.

bayespy.nodes.Multinomial.pdf

Multinomial.**pdf**(*X*, *mask=True*)
Compute the probability density function of this node.

bayespy.nodes.Multinomial.plot

Multinomial.**plot**(*fig=None*, ***kwargs*)
Plot the node distribution using the plotter of the node

Because the distributions are in general very difficult to plot, the user must specify some functions which performs the plotting as wanted. See, for instance, bayespy.plot.plotting for available plotters, that is, functions that perform plotting for a node.

bayespy.nodes.Multinomial.random

Multinomial.**random**()
Draw a random sample from the distribution.

bayespy.nodes.Multinomial.save

`Multinomial.save(group)`
Save the state of the node into a HDF5 file.

group can be the root

bayespy.nodes.Multinomial.set_parameters

`Multinomial.set_parameters(x)`
Set the parameters of the VB distribution.

The parameters should be such that they can be used for optimization, that is, use log transformation for positive parameters.

bayespy.nodes.Multinomial.set_plotter

`Multinomial.set_plotter(plotter)`

bayespy.nodes.Multinomial.show

`Multinomial.show()`
Print the distribution using standard parameterization.

bayespy.nodes.Multinomial.unobserve

`Multinomial.unobserve()`

bayespy.nodes.Multinomial.update

`Multinomial.update(annealing=1.0)`

Attributes

dims	
plates	
plates_multiplier	Plate multiplier is applied to messages to parents

bayespy.nodes.Multinomial.dims

`Multinomial.dims = None`

bayespy.nodes.Multinomial.plates

`Multinomial.plates = None`

bayespy.nodes.Multinomial.plates_multiplier

`Multinomial.plates_multiplier`
Plate multiplier is applied to messages to parents

bayespy.nodes.Poisson

class `bayespy.nodes.Poisson`(*l*, ***kwargs*)
Node for Poisson random variables.

The node uses Poisson distribution:

$$p(x) = \mathrm{Poisson}(x|\lambda)$$

where λ is the rate parameter.

Parameters **l** : gamma-like node or scalar or array

λ, rate parameter

See also:

`Gamma`, `Exponential`

`__init__`(*l*, ***kwargs*)
Create Poisson random variable node

Methods

`__init__`(l, **kwargs)	Create Poisson random variable node
`add_plate_axis`(to_plate)	
`broadcasting_multiplier`(plates, *args)	
`delete`()	Delete this node and the children
`get_gradient`(rg)	Computes gradient with respect to the natural parameters.
`get_mask`()	
`get_moments`()	
`get_parameters`()	Return parameters of the VB distribution.
`get_riemannian_gradient`()	Computes the Riemannian/natural gradient.
`get_shape`(ind)	
`has_plotter`()	Return True if the node has a plotter
`initialize_from_parameters`(*args)	
`initialize_from_prior`()	
`initialize_from_random`()	Set the variable to a random sample from the current distribution.
`initialize_from_value`(x, *args)	
`load`(group)	Load the state of the node from a HDF5 file.
`logpdf`(X[, mask])	Compute the log probability density function Q(X) of this node.
`lower_bound_contribution`([gradient, ...])	Compute E[log p(X|parents) - log q(X)]
`lowerbound`()	
`move_plates`(from_plate, to_plate)	
`observe`(x, *args[, mask])	Fix moments, compute f and propagate mask.
`pdf`(X[, mask])	Compute the probability density function of this node.
`plot`([fig])	Plot the node distribution using the plotter of the node
`random`()	Draw a random sample from the distribution.
`save`(group)	Save the state of the node into a HDF5 file.

Continued on next page

Table 5.30 – continued from previous page

set_parameters(x)	Set the parameters of the VB distribution.
set_plotter(plotter)	
show()	Print the distribution using standard parameterization.
unobserve()	
update([annealing])	

bayespy.nodes.Poisson.__init__

Poisson.**__init__**(*l*, ***kwargs*)
Create Poisson random variable node

bayespy.nodes.Poisson.add_plate_axis

Poisson.**add_plate_axis**(*to_plate*)

bayespy.nodes.Poisson.broadcasting_multiplier

Poisson.**broadcasting_multiplier**(*plates*, **args*)

bayespy.nodes.Poisson.delete

Poisson.**delete**()
Delete this node and the children

bayespy.nodes.Poisson.get_gradient

Poisson.**get_gradient**(*rg*)
Computes gradient with respect to the natural parameters.

The function takes the Riemannian gradient as an input. This is for three reasons: 1) You probably want to use the Riemannian gradient anyway so this helps avoiding accidental use of this function. 2) The gradient is computed by using the Riemannian gradient and chain rules. 3) Probably you need both Riemannian and normal gradients anyway so you can provide it to this function to avoid re-computing it.

bayespy.nodes.Poisson.get_mask

Poisson.**get_mask**()

bayespy.nodes.Poisson.get_moments

Poisson.**get_moments**()

bayespy.nodes.Poisson.get_parameters

`Poisson.get_parameters()`

Return parameters of the VB distribution.

The parameters should be such that they can be used for optimization, that is, use log transformation for positive parameters.

bayespy.nodes.Poisson.get_riemannian_gradient

`Poisson.get_riemannian_gradient()`

Computes the Riemannian/natural gradient.

bayespy.nodes.Poisson.get_shape

`Poisson.get_shape(`*ind*`)`

bayespy.nodes.Poisson.has_plotter

`Poisson.has_plotter()`

Return True if the node has a plotter

bayespy.nodes.Poisson.initialize_from_parameters

`Poisson.initialize_from_parameters(`**args*`)`

bayespy.nodes.Poisson.initialize_from_prior

`Poisson.initialize_from_prior()`

bayespy.nodes.Poisson.initialize_from_random

`Poisson.initialize_from_random()`

Set the variable to a random sample from the current distribution.

bayespy.nodes.Poisson.initialize_from_value

`Poisson.initialize_from_value(`*x*, **args*`)`

bayespy.nodes.Poisson.load

`Poisson.load(`*group*`)`

Load the state of the node from a HDF5 file.

bayespy.nodes.Poisson.logpdf

`Poisson.logpdf(X, mask=True)`

Compute the log probability density function Q(X) of this node.

bayespy.nodes.Poisson.lower_bound_contribution

`Poisson.lower_bound_contribution(gradient=False, ignore_masked=True)`

Compute E[log p(X|parents) - log q(X)]

If deterministic annealing is used, the term E[-log q(X)] is divided by the anneling coefficient. That is, phi and cgf of q are multiplied by the temperature (inverse annealing coefficient).

bayespy.nodes.Poisson.lowerbound

`Poisson.lowerbound()`

bayespy.nodes.Poisson.move_plates

`Poisson.move_plates(from_plate, to_plate)`

bayespy.nodes.Poisson.observe

`Poisson.observe(x, *args, mask=True)`

Fix moments, compute f and propagate mask.

bayespy.nodes.Poisson.pdf

`Poisson.pdf(X, mask=True)`

Compute the probability density function of this node.

bayespy.nodes.Poisson.plot

`Poisson.plot(fig=None, **kwargs)`

Plot the node distribution using the plotter of the node

Because the distributions are in general very difficult to plot, the user must specify some functions which performs the plotting as wanted. See, for instance, bayespy.plot.plotting for available plotters, that is, functions that perform plotting for a node.

bayespy.nodes.Poisson.random

`Poisson.random()`

Draw a random sample from the distribution.

bayespy.nodes.Poisson.save

`Poisson.save(group)`

Save the state of the node into a HDF5 file.

group can be the root

bayespy.nodes.Poisson.set_parameters

`Poisson.set_parameters(x)`

Set the parameters of the VB distribution.

The parameters should be such that they can be used for optimization, that is, use log transformation for positive parameters.

bayespy.nodes.Poisson.set_plotter

`Poisson.set_plotter(plotter)`

bayespy.nodes.Poisson.show

`Poisson.show()`

Print the distribution using standard parameterization.

bayespy.nodes.Poisson.unobserve

`Poisson.unobserve()`

bayespy.nodes.Poisson.update

`Poisson.update(annealing=1.0)`

Attributes

dims	
plates	
plates_multiplier	Plate multiplier is applied to messages to parents

bayespy.nodes.Poisson.dims

`Poisson.dims = ((),)`

bayespy.nodes.Poisson.plates

`Poisson.plates = None`

bayespy.nodes.Poisson.plates_multiplier

`Poisson.plates_multiplier`
Plate multiplier is applied to messages to parents

Nodes for probabilities:

`Beta`(alpha, **kwargs)	Node for beta random variables.
`Dirichlet`(*args, **kwargs)	Node for Dirichlet random variables.

bayespy.nodes.Beta

class `bayespy.nodes.Beta`(*alpha*, ***kwargs*)
Node for beta random variables.

The node models a probability variable $p \in [0, 1]$ as

$$p \sim \mathrm{Beta}(a, b)$$

where a and b are prior counts for success and failure, respectively.

Parameters **alpha** : (...,2)-shaped array

Two-element vector containing a and b

Examples

```
from bayespy.nodes import Bernoulli, Beta
p = Beta([1e-3, 1e-3])
z = Bernoulli(p, plates=(10,))
z.observe([0, 1, 1, 1, 0, 1, 1, 1, 0, 1])
p.update()
import bayespy.plot as bpplt
import numpy as np
bpplt.pdf(p, np.linspace(0, 1, num=100))
```

`__init__`(*alpha*, ***kwargs*)
Create beta node

Methods

`__init__`(alpha, **kwargs)	Create beta node
`add_plate_axis`(to_plate)	
`broadcasting_multiplier`(plates, *args)	
`delete`()	Delete this node and the children
`get_gradient`(rg)	Computes gradient with respect to the natural parameters.
`get_mask`()	
`get_moments`()	
`get_parameters`()	Return parameters of the VB distribution.
`get_riemannian_gradient`()	Computes the Riemannian/natural gradient.
`get_shape`(ind)	
`has_plotter`()	Return True if the node has a plotter

Continued on next page

Table 5.33 – continued from previous page

initialize_from_parameters(*args)	
initialize_from_prior()	
initialize_from_random()	Set the variable to a random sample from the current distribution.
initialize_from_value(x, *args)	
load(group)	Load the state of the node from a HDF5 file.
logpdf(X[, mask])	Compute the log probability density function Q(X) of this node.
lower_bound_contribution([gradient, ...])	Compute E[log p(X|parents) - log q(X)]
lowerbound()	
move_plates(from_plate, to_plate)	
observe(x, *args[, mask])	Fix moments, compute f and propagate mask.
pdf(X[, mask])	Compute the probability density function of this node.
plot([fig])	Plot the node distribution using the plotter of the node
random()	Draw a random sample from the distribution.
save(group)	Save the state of the node into a HDF5 file.
set_parameters(x)	Set the parameters of the VB distribution.
set_plotter(plotter)	
show()	Print the distribution using standard parameterization.
unobserve()	
update([annealing])	

bayespy.nodes.Beta.__init__

Beta.**__init__**(*alpha*, ***kwargs*)
Create beta node

bayespy.nodes.Beta.add_plate_axis

Beta.**add_plate_axis**(*to_plate*)

bayespy.nodes.Beta.broadcasting_multiplier

Beta.**broadcasting_multiplier**(*plates*, **args*)

bayespy.nodes.Beta.delete

Beta.**delete**()
Delete this node and the children

bayespy.nodes.Beta.get_gradient

Beta.**get_gradient**(*rg*)
Computes gradient with respect to the natural parameters.

The function takes the Riemannian gradient as an input. This is for three reasons: 1) You probably want to use the Riemannian gradient anyway so this helps avoiding accidental use of this function. 2) The gradient is computed by using the Riemannian gradient and chain rules. 3) Probably you need both Riemannian and normal gradients anyway so you can provide it to this function to avoid re-computing it.

bayespy.nodes.Beta.get_mask

`Beta.get_mask()`

bayespy.nodes.Beta.get_moments

`Beta.get_moments()`

bayespy.nodes.Beta.get_parameters

`Beta.get_parameters()`

Return parameters of the VB distribution.

The parameters should be such that they can be used for optimization, that is, use log transformation for positive parameters.

bayespy.nodes.Beta.get_riemannian_gradient

`Beta.get_riemannian_gradient()`

Computes the Riemannian/natural gradient.

bayespy.nodes.Beta.get_shape

`Beta.get_shape(ind)`

bayespy.nodes.Beta.has_plotter

`Beta.has_plotter()`

Return True if the node has a plotter

bayespy.nodes.Beta.initialize_from_parameters

`Beta.initialize_from_parameters(*args)`

bayespy.nodes.Beta.initialize_from_prior

`Beta.initialize_from_prior()`

bayespy.nodes.Beta.initialize_from_random

`Beta.initialize_from_random()`

Set the variable to a random sample from the current distribution.

bayespy.nodes.Beta.initialize_from_value

`Beta.initialize_from_value(x, *args)`

bayespy.nodes.Beta.load

`Beta.load(group)`
Load the state of the node from a HDF5 file.

bayespy.nodes.Beta.logpdf

`Beta.logpdf(X, mask=True)`
Compute the log probability density function Q(X) of this node.

bayespy.nodes.Beta.lower_bound_contribution

`Beta.lower_bound_contribution(gradient=False, ignore_masked=True)`
Compute E[log p(X|parents) - log q(X)]

If deterministic annealing is used, the term E[-log q(X)] is divided by the anneling coefficient. That is, phi and cgf of q are multiplied by the temperature (inverse annealing coefficient).

bayespy.nodes.Beta.lowerbound

`Beta.lowerbound()`

bayespy.nodes.Beta.move_plates

`Beta.move_plates(from_plate, to_plate)`

bayespy.nodes.Beta.observe

`Beta.observe(x, *args, mask=True)`
Fix moments, compute f and propagate mask.

bayespy.nodes.Beta.pdf

`Beta.pdf(X, mask=True)`
Compute the probability density function of this node.

bayespy.nodes.Beta.plot

`Beta.plot(fig=None, **kwargs)`
Plot the node distribution using the plotter of the node

Because the distributions are in general very difficult to plot, the user must specify some functions which performs the plotting as wanted. See, for instance, bayespy.plot.plotting for available plotters, that is, functions that perform plotting for a node.

bayespy.nodes.Beta.random

`Beta.random()`
Draw a random sample from the distribution.

bayespy.nodes.Beta.save

`Beta.save(group)`
Save the state of the node into a HDF5 file.

group can be the root

bayespy.nodes.Beta.set_parameters

`Beta.set_parameters(x)`
Set the parameters of the VB distribution.

The parameters should be such that they can be used for optimization, that is, use log transformation for positive parameters.

bayespy.nodes.Beta.set_plotter

`Beta.set_plotter(plotter)`

bayespy.nodes.Beta.show

`Beta.show()`
Print the distribution using standard parameterization.

bayespy.nodes.Beta.unobserve

`Beta.unobserve()`

bayespy.nodes.Beta.update

`Beta.update(annealing=1.0)`

Attributes

dims	
plates	
plates_multiplier	Plate multiplier is applied to messages to parents

bayespy.nodes.Beta.dims

`Beta.dims = None`

bayespy.nodes.Beta.plates

`Beta.plates` = None

bayespy.nodes.Beta.plates_multiplier

`Beta.plates_multiplier`
Plate multiplier is applied to messages to parents

bayespy.nodes.Dirichlet

class `bayespy.nodes.Dirichlet`(*args, **kwargs)
Node for Dirichlet random variables.

The node models a set of probabilities $\{\pi_0,\dots,\pi_{K-1}\}$ which satisfy $\sum_{k=0}^{K-1}\pi_k = 1$ and $\pi_k \in [0,1]\ \forall k = 0,\dots,K-1$.

$$p(\pi_0,\dots,\pi_{K-1}) = \mathrm{Dirichlet}(\alpha_0,\dots,\alpha_{K-1})$$

where α_k are concentration parameters.

The posterior approximation has the same functional form but with different concentration parameters.

Parameters **alpha** : (...,K)-shaped array

Prior counts α_k

See also:

`Beta`, `Categorical`, `Multinomial`, `CategoricalMarkovChain`

`__init__`(*args, **kwargs)

Methods

`__init__`(*args, **kwargs)	
`add_plate_axis`(to_plate)	
`broadcasting_multiplier`(plates, *args)	
`delete`()	Delete this node and the children
`get_gradient`(rg)	Computes gradient with respect to the natural parameters.
`get_mask`()	
`get_moments`()	
`get_parameters`()	Return parameters of the VB distribution.
`get_riemannian_gradient`()	Computes the Riemannian/natural gradient.
`get_shape`(ind)	
`has_plotter`()	Return True if the node has a plotter
`initialize_from_parameters`(*args)	
`initialize_from_prior`()	
`initialize_from_random`()	Set the variable to a random sample from the current distribution.
`initialize_from_value`(x, *args)	
`load`(group)	Load the state of the node from a HDF5 file.
`logpdf`(X[, mask])	Compute the log probability density function Q(X) of this node.
`lower_bound_contribution`([gradient, ...])	Compute E[log p(X|parents) - log q(X)]
`lowerbound`()	

Continued on next page

Table 5.35 – continued from previous page

move_plates(from_plate, to_plate)	
observe(x, *args[, mask])	Fix moments, compute f and propagate mask.
pdf(X[, mask])	Compute the probability density function of this node.
plot([fig])	Plot the node distribution using the plotter of the node
random()	Draw a random sample from the distribution.
save(group)	Save the state of the node into a HDF5 file.
set_parameters(x)	Set the parameters of the VB distribution.
set_plotter(plotter)	
show()	Print the distribution using standard parameterization.
unobserve()	
update([annealing])	

bayespy.nodes.Dirichlet.__init__

`Dirichlet.__init__(*args, **kwargs)`

bayespy.nodes.Dirichlet.add_plate_axis

`Dirichlet.add_plate_axis(to_plate)`

bayespy.nodes.Dirichlet.broadcasting_multiplier

`Dirichlet.broadcasting_multiplier(plates, *args)`

bayespy.nodes.Dirichlet.delete

`Dirichlet.delete()`

Delete this node and the children

bayespy.nodes.Dirichlet.get_gradient

`Dirichlet.get_gradient(rg)`

Computes gradient with respect to the natural parameters.

The function takes the Riemannian gradient as an input. This is for three reasons: 1) You probably want to use the Riemannian gradient anyway so this helps avoiding accidental use of this function. 2) The gradient is computed by using the Riemannian gradient and chain rules. 3) Probably you need both Riemannian and normal gradients anyway so you can provide it to this function to avoid re-computing it.

bayespy.nodes.Dirichlet.get_mask

`Dirichlet.get_mask()`

bayespy.nodes.Dirichlet.get_moments

`Dirichlet.get_moments()`

bayespy.nodes.Dirichlet.get_parameters

Dirichlet.**get_parameters**()
Return parameters of the VB distribution.

The parameters should be such that they can be used for optimization, that is, use log transformation for positive parameters.

bayespy.nodes.Dirichlet.get_riemannian_gradient

Dirichlet.**get_riemannian_gradient**()
Computes the Riemannian/natural gradient.

bayespy.nodes.Dirichlet.get_shape

Dirichlet.**get_shape**(*ind*)

bayespy.nodes.Dirichlet.has_plotter

Dirichlet.**has_plotter**()
Return True if the node has a plotter

bayespy.nodes.Dirichlet.initialize_from_parameters

Dirichlet.**initialize_from_parameters**(**args*)

bayespy.nodes.Dirichlet.initialize_from_prior

Dirichlet.**initialize_from_prior**()

bayespy.nodes.Dirichlet.initialize_from_random

Dirichlet.**initialize_from_random**()
Set the variable to a random sample from the current distribution.

bayespy.nodes.Dirichlet.initialize_from_value

Dirichlet.**initialize_from_value**(*x*, **args*)

bayespy.nodes.Dirichlet.load

Dirichlet.**load**(*group*)
Load the state of the node from a HDF5 file.

bayespy.nodes.Dirichlet.logpdf

`Dirichlet.logpdf(X, mask=True)`
Compute the log probability density function Q(X) of this node.

bayespy.nodes.Dirichlet.lower_bound_contribution

`Dirichlet.lower_bound_contribution(gradient=False, ignore_masked=True)`
Compute E[log p(X|parents) - log q(X)]

If deterministic annealing is used, the term E[-log q(X)] is divided by the anneling coefficient. That is, phi and cgf of q are multiplied by the temperature (inverse annealing coefficient).

bayespy.nodes.Dirichlet.lowerbound

`Dirichlet.lowerbound()`

bayespy.nodes.Dirichlet.move_plates

`Dirichlet.move_plates(from_plate, to_plate)`

bayespy.nodes.Dirichlet.observe

`Dirichlet.observe(x, *args, mask=True)`
Fix moments, compute f and propagate mask.

bayespy.nodes.Dirichlet.pdf

`Dirichlet.pdf(X, mask=True)`
Compute the probability density function of this node.

bayespy.nodes.Dirichlet.plot

`Dirichlet.plot(fig=None, **kwargs)`
Plot the node distribution using the plotter of the node

Because the distributions are in general very difficult to plot, the user must specify some functions which performs the plotting as wanted. See, for instance, bayespy.plot.plotting for available plotters, that is, functions that perform plotting for a node.

bayespy.nodes.Dirichlet.random

`Dirichlet.random()`
Draw a random sample from the distribution.

bayespy.nodes.Dirichlet.save

`Dirichlet.save(group)`

Save the state of the node into a HDF5 file.

group can be the root

bayespy.nodes.Dirichlet.set_parameters

`Dirichlet.set_parameters(x)`

Set the parameters of the VB distribution.

The parameters should be such that they can be used for optimization, that is, use log transformation for positive parameters.

bayespy.nodes.Dirichlet.set_plotter

`Dirichlet.set_plotter(plotter)`

bayespy.nodes.Dirichlet.show

`Dirichlet.show()`

Print the distribution using standard parameterization.

bayespy.nodes.Dirichlet.unobserve

`Dirichlet.unobserve()`

bayespy.nodes.Dirichlet.update

`Dirichlet.update(annealing=1.0)`

Attributes

dims	
plates	
plates_multiplier	Plate multiplier is applied to messages to parents

bayespy.nodes.Dirichlet.dims

`Dirichlet.dims` = None

bayespy.nodes.Dirichlet.plates

`Dirichlet.plates` = None

bayespy.nodes.Dirichlet.plates_multiplier

Dirichlet.**plates_multiplier**
Plate multiplier is applied to messages to parents

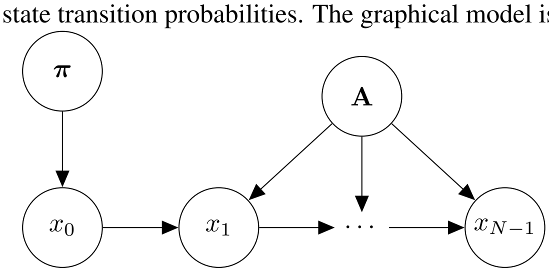
Nodes for dynamic variables:

CategoricalMarkovChain(pi, A[, states])	Node for categorical Markov chain random variables.
GaussianMarkovChain(mu, Lambda, A, nu[, n, ...])	Node for Gaussian Markov chain random variables.
SwitchingGaussianMarkovChain(mu, Lambda, B, ...)	Node for Gaussian Markov chain random variables with switching d
VaryingGaussianMarkovChain(mu, Lambda, B, S, nu)	Node for Gaussian Markov chain random variables with time-varyin

bayespy.nodes.CategoricalMarkovChain

class bayespy.nodes.**CategoricalMarkovChain**(*pi*, *A*, *states=None*, ***kwargs*)
Node for categorical Markov chain random variables.

The node models a Markov chain which has a discrete set of K possible states and the next state depends only on the previous state and the state transition probabilities. The graphical model is shown below:



where $\boldsymbol{\pi}$ contains the probabilities for the initial state and $\mathbf{A}$ is the state transition probability matrix. It is possible to have $\mathbf{A}$ varying in time.

$$p(x_0, \ldots, x_{N-1}) = p(x_0) \prod_{n=1}^{N-1} p(x_n | x_{n-1}),$$

where

$$\begin{aligned} p(x_0 = k) &= \pi_k, \quad \text{for } k \in \{0, \ldots, K-1\}, \\ p(x_n = j | x_{n-1} = i) &= a_{ij}^{(n-1)} \quad \text{for } n = 1, \ldots, N-1,\, i \in \{1, \ldots, K-1\},\, j \in \{1, \ldots, K-1\} \\ a_{ij}^{(n)} &= [\mathbf{A}_n]_{ij} \end{aligned}$$

This node can be used to construct hidden Markov models by using Mixture for the emission distribution.

Parameters **pi** : Dirichlet-like node or (...,K)-array

$\boldsymbol{\pi}$, probabilities for the first state. K-dimensional Dirichlet.

A : Dirichlet-like node or (K,K)-array or (...,1,K,K)-array or (...,N-1,K,K)-array

$\mathbf{A}$, probabilities for state transitions. K-dimensional Dirichlet with plates (K,) or (...,1,K) or (...,N-1,K).

states : int, optional

N, the length of the chain.

See also:

Categorical, Dirichlet, GaussianMarkovChain, Mixture, SwitchingGaussianMarkovChain

`__init__`(*pi*, *A*, *states=None*, ***kwargs*)
Create categorical Markov chain

Methods

`__init__`(pi, A[, states])	Create categorical Markov chain
`add_plate_axis`(to_plate)	
`broadcasting_multiplier`(plates, *args)	
`delete`()	Delete this node and the children
`get_gradient`(rg)	Computes gradient with respect to the natural parameters.
`get_mask`()	
`get_moments`()	
`get_parameters`()	Return parameters of the VB distribution.
`get_riemannian_gradient`()	Computes the Riemannian/natural gradient.
`get_shape`(ind)	
`has_plotter`()	Return True if the node has a plotter
`initialize_from_parameters`(*args)	
`initialize_from_prior`()	
`initialize_from_random`()	Set the variable to a random sample from the current distribution.
`initialize_from_value`(x, *args)	
`load`(group)	Load the state of the node from a HDF5 file.
`logpdf`(X[, mask])	Compute the log probability density function Q(X) of this node.
`lower_bound_contribution`([gradient, ...])	Compute E[log p(X|parents) - log q(X)]
`lowerbound`()	
`move_plates`(from_plate, to_plate)	
`observe`(x, *args[, mask])	Fix moments, compute f and propagate mask.
`pdf`(X[, mask])	Compute the probability density function of this node.
`plot`([fig])	Plot the node distribution using the plotter of the node
`random`()	Draw a random sample from the distribution.
`save`(group)	Save the state of the node into a HDF5 file.
`set_parameters`(x)	Set the parameters of the VB distribution.
`set_plotter`(plotter)	
`show`()	Print the distribution using standard parameterization.
`unobserve`()	
`update`([annealing])	

bayespy.nodes.CategoricalMarkovChain.__init__

`CategoricalMarkovChain.__init__`(*pi*, *A*, *states=None*, ***kwargs*)
Create categorical Markov chain

bayespy.nodes.CategoricalMarkovChain.add_plate_axis

`CategoricalMarkovChain.add_plate_axis`(*to_plate*)

bayespy.nodes.CategoricalMarkovChain.broadcasting_multiplier

`CategoricalMarkovChain.broadcasting_multiplier`(*plates*, **args*)

bayespy.nodes.CategoricalMarkovChain.delete

`CategoricalMarkovChain.delete()`

Delete this node and the children

bayespy.nodes.CategoricalMarkovChain.get_gradient

`CategoricalMarkovChain.get_gradient(rg)`

Computes gradient with respect to the natural parameters.

The function takes the Riemannian gradient as an input. This is for three reasons: 1) You probably want to use the Riemannian gradient anyway so this helps avoiding accidental use of this function. 2) The gradient is computed by using the Riemannian gradient and chain rules. 3) Probably you need both Riemannian and normal gradients anyway so you can provide it to this function to avoid re-computing it.

bayespy.nodes.CategoricalMarkovChain.get_mask

`CategoricalMarkovChain.get_mask()`

bayespy.nodes.CategoricalMarkovChain.get_moments

`CategoricalMarkovChain.get_moments()`

bayespy.nodes.CategoricalMarkovChain.get_parameters

`CategoricalMarkovChain.get_parameters()`

Return parameters of the VB distribution.

The parameters should be such that they can be used for optimization, that is, use log transformation for positive parameters.

bayespy.nodes.CategoricalMarkovChain.get_riemannian_gradient

`CategoricalMarkovChain.get_riemannian_gradient()`

Computes the Riemannian/natural gradient.

bayespy.nodes.CategoricalMarkovChain.get_shape

`CategoricalMarkovChain.get_shape(ind)`

bayespy.nodes.CategoricalMarkovChain.has_plotter

`CategoricalMarkovChain.has_plotter()`

Return True if the node has a plotter

bayespy.nodes.CategoricalMarkovChain.initialize_from_parameters

`CategoricalMarkovChain.initialize_from_parameters(*args)`

bayespy.nodes.CategoricalMarkovChain.initialize_from_prior

CategoricalMarkovChain.**initialize_from_prior**()

bayespy.nodes.CategoricalMarkovChain.initialize_from_random

CategoricalMarkovChain.**initialize_from_random**()
Set the variable to a random sample from the current distribution.

bayespy.nodes.CategoricalMarkovChain.initialize_from_value

CategoricalMarkovChain.**initialize_from_value**(*x*, **args*)

bayespy.nodes.CategoricalMarkovChain.load

CategoricalMarkovChain.**load**(*group*)
Load the state of the node from a HDF5 file.

bayespy.nodes.CategoricalMarkovChain.logpdf

CategoricalMarkovChain.**logpdf**(*X*, *mask=True*)
Compute the log probability density function Q(X) of this node.

bayespy.nodes.CategoricalMarkovChain.lower_bound_contribution

CategoricalMarkovChain.**lower_bound_contribution**(*gradient=False*, *ignore_masked=True*)
Compute E[log p(X|parents) - log q(X)]

If deterministic annealing is used, the term E[-log q(X)] is divided by the anneling coefficient. That is, phi and cgf of q are multiplied by the temperature (inverse annealing coefficient).

bayespy.nodes.CategoricalMarkovChain.lowerbound

CategoricalMarkovChain.**lowerbound**()

bayespy.nodes.CategoricalMarkovChain.move_plates

CategoricalMarkovChain.**move_plates**(*from_plate*, *to_plate*)

bayespy.nodes.CategoricalMarkovChain.observe

CategoricalMarkovChain.**observe**(*x*, **args*, *mask=True*)
Fix moments, compute f and propagate mask.

bayespy.nodes.CategoricalMarkovChain.pdf

CategoricalMarkovChain.**pdf**(*X*, *mask=True*)
Compute the probability density function of this node.

bayespy.nodes.CategoricalMarkovChain.plot

CategoricalMarkovChain.**plot**(*fig=None*, ***kwargs*)
Plot the node distribution using the plotter of the node

Because the distributions are in general very difficult to plot, the user must specify some functions which performs the plotting as wanted. See, for instance, bayespy.plot.plotting for available plotters, that is, functions that perform plotting for a node.

bayespy.nodes.CategoricalMarkovChain.random

CategoricalMarkovChain.**random**()
Draw a random sample from the distribution.

bayespy.nodes.CategoricalMarkovChain.save

CategoricalMarkovChain.**save**(*group*)
Save the state of the node into a HDF5 file.

group can be the root

bayespy.nodes.CategoricalMarkovChain.set_parameters

CategoricalMarkovChain.**set_parameters**(*x*)
Set the parameters of the VB distribution.

The parameters should be such that they can be used for optimization, that is, use log transformation for positive parameters.

bayespy.nodes.CategoricalMarkovChain.set_plotter

CategoricalMarkovChain.**set_plotter**(*plotter*)

bayespy.nodes.CategoricalMarkovChain.show

CategoricalMarkovChain.**show**()
Print the distribution using standard parameterization.

bayespy.nodes.CategoricalMarkovChain.unobserve

CategoricalMarkovChain.**unobserve**()

bayespy.nodes.CategoricalMarkovChain.update

CategoricalMarkovChain.**update**(*annealing=1.0*)

Attributes

dims	
plates	
plates_multiplier	Plate multiplier is applied to messages to parents

bayespy.nodes.CategoricalMarkovChain.dims

CategoricalMarkovChain.**dims** = None

bayespy.nodes.CategoricalMarkovChain.plates

CategoricalMarkovChain.**plates** = None

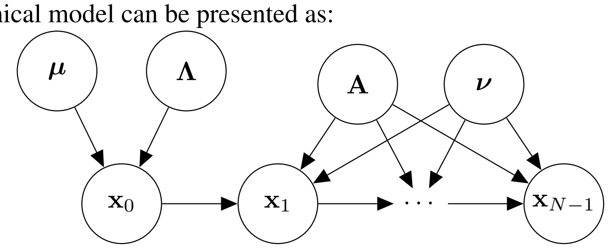
bayespy.nodes.CategoricalMarkovChain.plates_multiplier

CategoricalMarkovChain.**plates_multiplier**
Plate multiplier is applied to messages to parents

bayespy.nodes.GaussianMarkovChain

class bayespy.nodes.**GaussianMarkovChain**(*mu*, *Lambda*, *A*, *nu*, *n=None*, *inputs=None*, ***kwargs*)
Node for Gaussian Markov chain random variables.

In a simple case, the graphical model can be presented as:



where $\boldsymbol{\mu}$ and $\boldsymbol{\Lambda}$ are the mean and the precision matrix of the initial state, $\mathbf{A}$ is the state dynamics matrix and $\boldsymbol{\nu}$ is the precision of the innovation noise. It is possible that $\mathbf{A}$ and/or $\boldsymbol{\nu}$ are different for each transition instead of being constant.

The probability distribution is

$$p(\mathbf{x}_0, \dots, \mathbf{x}_{N-1}) = p(\mathbf{x}_0) \prod^{N-1}_{n=1} p(\mathbf{x}_n | \mathbf{x}_{n-1})$$

where

$$\begin{aligned} p(\mathbf{x}_0) &= \mathcal{N}(\mathbf{x}_0 | \boldsymbol{\mu}, \boldsymbol{\Lambda}) \\ p(\mathbf{x}_n | \mathbf{x}_{n-1}) &= \mathcal{N}(\mathbf{x}_n | \mathbf{A}_{n-1}\mathbf{x}_{n-1}, \mathrm{diag}(\boldsymbol{\nu}_{n-1})). \end{aligned}$$

Parameters **mu** : Gaussian-like node or (...,D)-array

$\boldsymbol{\mu}$, mean of x_0, D-dimensional with plates (...)

Lambda : Wishart-like node or (...,D,D)-array

$\boldsymbol{\Lambda}$, precision matrix of x_0, $D \times D$ -dimensional with plates (...)

A : Gaussian-like node or (D,D)-array or (...,1,D,D)-array or (...,N-1,D,D)-array

$\mathbf{A}$, state dynamics matrix, D-dimensional with plates (D,) or (...,1,D) or (...,N-1,D)

nu : gamma-like node or (D,)-array or (...,1,D)-array or (...,N-1,D)-array

$\boldsymbol{\nu}$, diagonal elements of the precision of the innovation process, plates (D,) or (...,1,D) or (...,N-1,D)

n : int, optional

N, the length of the chain. Must be given if $\mathbf{A}$ and $\boldsymbol{\nu}$ are constant over time.

See also:

`Gaussian`, `GaussianARD`, `Wishart`, `Gamma`, `SwitchingGaussianMarkovChain`, `VaryingGaussianMarkovChain`, `CategoricalMarkovChain`

`__init__`(*mu*, *Lambda*, *A*, *nu*, *n=None*, *inputs=None*, ***kwargs*)
Create GaussianMarkovChain node.

Methods

`__init__`(mu, Lambda, A, nu[, n, inputs])	Create GaussianMarkovChain node.
`add_plate_axis`(to_plate)	
`broadcasting_multiplier`(plates, *args)	
`delete`()	Delete this node and the children
`get_gradient`(rg)	Computes gradient with respect to the natural parameters.
`get_mask`()	
`get_moments`()	
`get_parameters`()	Return parameters of the VB distribution.
`get_riemannian_gradient`()	Computes the Riemannian/natural gradient.
`get_shape`(ind)	
`has_plotter`()	Return True if the node has a plotter
`initialize_from_parameters`(*args)	
`initialize_from_prior`()	
`initialize_from_random`()	Set the variable to a random sample from the current distribution.
`initialize_from_value`(x, *args)	
`load`(group)	Load the state of the node from a HDF5 file.
`logpdf`(X[, mask])	Compute the log probability density function Q(X) of this node.
`lower_bound_contribution`([gradient, ...])	Compute E[log p(X|parents) - log q(X)]
`lowerbound`()	
`move_plates`(from_plate, to_plate)	
`observe`(x, *args[, mask])	Fix moments, compute f and propagate mask.
`pdf`(X[, mask])	Compute the probability density function of this node.
`plot`([fig])	Plot the node distribution using the plotter of the node
`random`()	Draw a random sample from the distribution.
`rotate`(R[, inv, logdet])	
`save`(group)	Save the state of the node into a HDF5 file.
`set_parameters`(x)	Set the parameters of the VB distribution.

Continued on next page

Table 5.40 – continued from previous page

set_plotter(plotter)	
show()	
unobserve()	
update([annealing])	

bayespy.nodes.GaussianMarkovChain.__init__

GaussianMarkovChain.**__init__**(*mu, Lambda, A, nu, n=None, inputs=None, **kwargs*)
Create GaussianMarkovChain node.

bayespy.nodes.GaussianMarkovChain.add_plate_axis

GaussianMarkovChain.**add_plate_axis**(*to_plate*)

bayespy.nodes.GaussianMarkovChain.broadcasting_multiplier

GaussianMarkovChain.**broadcasting_multiplier**(*plates, *args*)

bayespy.nodes.GaussianMarkovChain.delete

GaussianMarkovChain.**delete**()
Delete this node and the children

bayespy.nodes.GaussianMarkovChain.get_gradient

GaussianMarkovChain.**get_gradient**(*rg*)
Computes gradient with respect to the natural parameters.

The function takes the Riemannian gradient as an input. This is for three reasons: 1) You probably want to use the Riemannian gradient anyway so this helps avoiding accidental use of this function. 2) The gradient is computed by using the Riemannian gradient and chain rules. 3) Probably you need both Riemannian and normal gradients anyway so you can provide it to this function to avoid re-computing it.

bayespy.nodes.GaussianMarkovChain.get_mask

GaussianMarkovChain.**get_mask**()

bayespy.nodes.GaussianMarkovChain.get_moments

GaussianMarkovChain.**get_moments**()

bayespy.nodes.GaussianMarkovChain.get_parameters

GaussianMarkovChain.**get_parameters**()
Return parameters of the VB distribution.

The parameters should be such that they can be used for optimization, that is, use log transformation for positive parameters.

bayespy.nodes.GaussianMarkovChain.get_riemannian_gradient

`GaussianMarkovChain.get_riemannian_gradient()`
Computes the Riemannian/natural gradient.

bayespy.nodes.GaussianMarkovChain.get_shape

`GaussianMarkovChain.get_shape(ind)`

bayespy.nodes.GaussianMarkovChain.has_plotter

`GaussianMarkovChain.has_plotter()`
Return True if the node has a plotter

bayespy.nodes.GaussianMarkovChain.initialize_from_parameters

`GaussianMarkovChain.initialize_from_parameters(*args)`

bayespy.nodes.GaussianMarkovChain.initialize_from_prior

`GaussianMarkovChain.initialize_from_prior()`

bayespy.nodes.GaussianMarkovChain.initialize_from_random

`GaussianMarkovChain.initialize_from_random()`
Set the variable to a random sample from the current distribution.

bayespy.nodes.GaussianMarkovChain.initialize_from_value

`GaussianMarkovChain.initialize_from_value(x, *args)`

bayespy.nodes.GaussianMarkovChain.load

`GaussianMarkovChain.load(group)`
Load the state of the node from a HDF5 file.

bayespy.nodes.GaussianMarkovChain.logpdf

`GaussianMarkovChain.logpdf(X, mask=True)`
Compute the log probability density function Q(X) of this node.

bayespy.nodes.GaussianMarkovChain.lower_bound_contribution

GaussianMarkovChain.**lower_bound_contribution**(*gradient=False*, *ignore_masked=True*)
Compute E[log p(X|parents) - log q(X)]

If deterministic annealing is used, the term E[-log q(X)] is divided by the anneling coefficient. That is, phi and cgf of q are multiplied by the temperature (inverse annealing coefficient).

bayespy.nodes.GaussianMarkovChain.lowerbound

GaussianMarkovChain.**lowerbound**()

bayespy.nodes.GaussianMarkovChain.move_plates

GaussianMarkovChain.**move_plates**(*from_plate*, *to_plate*)

bayespy.nodes.GaussianMarkovChain.observe

GaussianMarkovChain.**observe**(*x*, **args*, *mask=True*)
Fix moments, compute f and propagate mask.

bayespy.nodes.GaussianMarkovChain.pdf

GaussianMarkovChain.**pdf**(*X*, *mask=True*)
Compute the probability density function of this node.

bayespy.nodes.GaussianMarkovChain.plot

GaussianMarkovChain.**plot**(*fig=None*, ***kwargs*)
Plot the node distribution using the plotter of the node

Because the distributions are in general very difficult to plot, the user must specify some functions which performs the plotting as wanted. See, for instance, bayespy.plot.plotting for available plotters, that is, functions that perform plotting for a node.

bayespy.nodes.GaussianMarkovChain.random

GaussianMarkovChain.**random**()
Draw a random sample from the distribution.

bayespy.nodes.GaussianMarkovChain.rotate

GaussianMarkovChain.**rotate**(*R*, *inv=None*, *logdet=None*)

bayespy.nodes.GaussianMarkovChain.save

GaussianMarkovChain.**save**(*group*)

Save the state of the node into a HDF5 file.

group can be the root

bayespy.nodes.GaussianMarkovChain.set_parameters

GaussianMarkovChain.**set_parameters**(*x*)

Set the parameters of the VB distribution.

The parameters should be such that they can be used for optimization, that is, use log transformation for positive parameters.

bayespy.nodes.GaussianMarkovChain.set_plotter

GaussianMarkovChain.**set_plotter**(*plotter*)

bayespy.nodes.GaussianMarkovChain.show

GaussianMarkovChain.**show**()

bayespy.nodes.GaussianMarkovChain.unobserve

GaussianMarkovChain.**unobserve**()

bayespy.nodes.GaussianMarkovChain.update

GaussianMarkovChain.**update**(*annealing=1.0*)

Attributes

dims	
plates	
plates_multiplier	Plate multiplier is applied to messages to parents

bayespy.nodes.GaussianMarkovChain.dims

GaussianMarkovChain.**dims** = None

bayespy.nodes.GaussianMarkovChain.plates

GaussianMarkovChain.**plates** = None

bayespy.nodes.GaussianMarkovChain.plates_multiplier

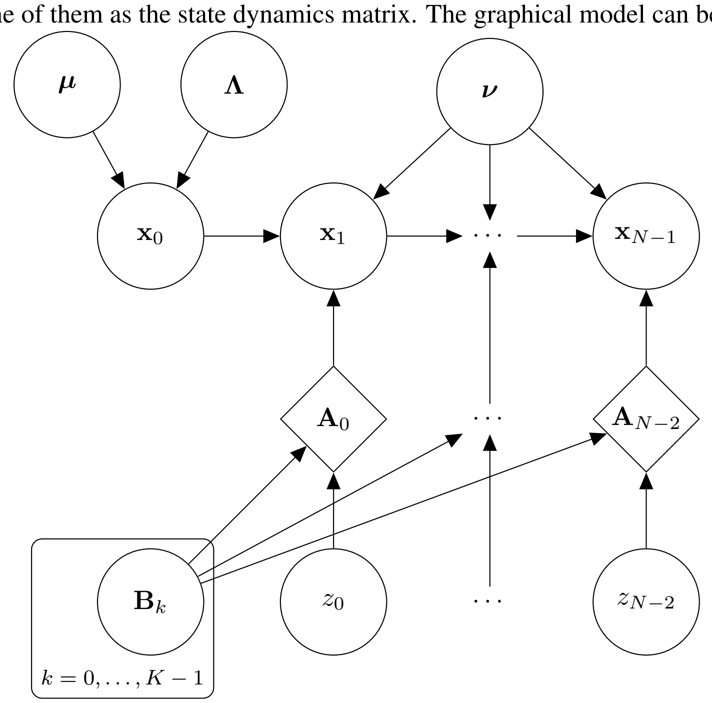
`GaussianMarkovChain.`**`plates_multiplier`**
Plate multiplier is applied to messages to parents

bayespy.nodes.SwitchingGaussianMarkovChain

class `bayespy.nodes.`**`SwitchingGaussianMarkovChain`**(*mu*, *Lambda*, *B*, *Z*, *nu*, *n=None*, ***kwargs*)

Node for Gaussian Markov chain random variables with switching dynamics.

The node models a sequence of Gaussian variables :math:'mathbf{x}_0,ldots,mathbf{x}_{N-1}$ with linear Markovian dynamics. The dynamics may change in time, which is obtained by having a set of matrices and at each time selecting one of them as the state dynamics matrix. The graphical model can be presented as:



where $\boldsymbol{\mu}$ and $\boldsymbol{\Lambda}$ are the mean and the precision matrix of the initial state, $\boldsymbol{\nu}$ is the precision of the innovation noise, and $\mathbf{A}_n$ are the state dynamics matrix obtained by selecting one of the matrices $\{\mathbf{B}_k\}_{k=0}^{K-1}$ at each time. The selections are provided by $z_n \in \{0, \ldots, K-1\}$. The probability distribution is

$$p(\mathbf{x}_0, \ldots, \mathbf{x}_{N-1}) = p(\mathbf{x}_0) \prod_{n=1}^{N-1} p(\mathbf{x}_n | \mathbf{x}_{n-1})$$

where

$$\begin{aligned} p(\mathbf{x}_0) &= \mathcal{N}(\mathbf{x}_0 | \boldsymbol{\mu}, \boldsymbol{\Lambda}) \\ p(\mathbf{x}_n | \mathbf{x}_{n-1}) &= \mathcal{N}(\mathbf{x}_n | \mathbf{A}_{n-1}\mathbf{x}_{n-1}, \mathrm{diag}(\boldsymbol{\nu})), \quad \text{for } n = 1, \ldots, N-1, \\ \mathbf{A}_n &= \mathbf{B}_{z_n}, \quad \text{for } n = 0, \ldots, N-2. \end{aligned}$$

Parameters **mu** : Gaussian-like node or (...,D)-array

$\boldsymbol{\mu}$, mean of x_0, D-dimensional with plates (...)

Lambda : Wishart-like node or (...,D,D)-array

$\boldsymbol{\Lambda}$, precision matrix of x_0, $D \times D$ -dimensional with plates (...)

B : Gaussian-like node or (...,D,D,K)-array

$\{\mathbf{B}_k\}_{k=0}^{K-1}$, a set of state dynamics matrix, $D \times K$-dimensional with plates (...,D)

Z : categorical-like node or (...,N-1)-array

$\{z_0, \ldots, z_{N-2}\}$, time-dependent selection, K-categorical with plates (...,N-1)

nu : gamma-like node or (...,D)-array

$\boldsymbol{\nu}$, diagonal elements of the precision of the innovation process, plates (...,D)

n : int, optional

N, the length of the chain. Must be given if **Z** does not have plates over the time domain (which would not make sense).

See also:

Gaussian, GaussianARD, Wishart, Gamma, GaussianMarkovChain, VaryingGaussianMarkovChain, Categorical, CategoricalMarkovChain

Notes

Equivalent model block can be constructed with GaussianMarkovChain by explicitly using Gate to select the state dynamics matrix. However, that approach is not very efficient for large datasets because it does not utilize the structure of $\mathbf{A}_n$, thus it explicitly computes huge moment arrays.

__init__(*mu*, *Lambda*, *B*, *Z*, *nu*, *n=None*, ***kwargs*)
Create SwitchingGaussianMarkovChain node.

Methods

__init__(mu, Lambda, B, Z, nu[, n])	Create SwitchingGaussianMarkovChain node.
add_plate_axis(to_plate)	
broadcasting_multiplier(plates, *args)	
delete()	Delete this node and the children
get_gradient(rg)	Computes gradient with respect to the natural parameters.
get_mask()	
get_moments()	
get_parameters()	Return parameters of the VB distribution.
get_riemannian_gradient()	Computes the Riemannian/natural gradient.
get_shape(ind)	
has_plotter()	Return True if the node has a plotter
initialize_from_parameters(*args)	
initialize_from_prior()	
initialize_from_random()	Set the variable to a random sample from the current distribution.
initialize_from_value(x, *args)	
load(group)	Load the state of the node from a HDF5 file.
logpdf(X[, mask])	Compute the log probability density function Q(X) of this node.
lower_bound_contribution([gradient, ...])	Compute E[log p(X|parents) - log q(X)]
lowerbound()	
move_plates(from_plate, to_plate)	
observe(x, *args[, mask])	Fix moments, compute f and propagate mask.
pdf(X[, mask])	Compute the probability density function of this node.
plot([fig])	Plot the node distribution using the plotter of the node

Continued on next page

Table 5.42 – continued from previous page

random()	Draw a random sample from the distribution.
rotate(R[, inv, logdet])	
save(group)	Save the state of the node into a HDF5 file.
set_parameters(x)	Set the parameters of the VB distribution.
set_plotter(plotter)	
show()	
unobserve()	
update([annealing])	

bayespy.nodes.SwitchingGaussianMarkovChain.__init__

SwitchingGaussianMarkovChain.**__init__**(*mu*, *Lambda*, *B*, *Z*, *nu*, *n=None*, ***kwargs*)
Create SwitchingGaussianMarkovChain node.

bayespy.nodes.SwitchingGaussianMarkovChain.add_plate_axis

SwitchingGaussianMarkovChain.**add_plate_axis**(*to_plate*)

bayespy.nodes.SwitchingGaussianMarkovChain.broadcasting_multiplier

SwitchingGaussianMarkovChain.**broadcasting_multiplier**(*plates*, **args*)

bayespy.nodes.SwitchingGaussianMarkovChain.delete

SwitchingGaussianMarkovChain.**delete**()
Delete this node and the children

bayespy.nodes.SwitchingGaussianMarkovChain.get_gradient

SwitchingGaussianMarkovChain.**get_gradient**(*rg*)
Computes gradient with respect to the natural parameters.

The function takes the Riemannian gradient as an input. This is for three reasons: 1) You probably want to use the Riemannian gradient anyway so this helps avoiding accidental use of this function. 2) The gradient is computed by using the Riemannian gradient and chain rules. 3) Probably you need both Riemannian and normal gradients anyway so you can provide it to this function to avoid re-computing it.

bayespy.nodes.SwitchingGaussianMarkovChain.get_mask

SwitchingGaussianMarkovChain.**get_mask**()

bayespy.nodes.SwitchingGaussianMarkovChain.get_moments

SwitchingGaussianMarkovChain.**get_moments**()

bayespy.nodes.SwitchingGaussianMarkovChain.get_parameters

`SwitchingGaussianMarkovChain.get_parameters()`

Return parameters of the VB distribution.

The parameters should be such that they can be used for optimization, that is, use log transformation for positive parameters.

bayespy.nodes.SwitchingGaussianMarkovChain.get_riemannian_gradient

`SwitchingGaussianMarkovChain.get_riemannian_gradient()`

Computes the Riemannian/natural gradient.

bayespy.nodes.SwitchingGaussianMarkovChain.get_shape

`SwitchingGaussianMarkovChain.get_shape(ind)`

bayespy.nodes.SwitchingGaussianMarkovChain.has_plotter

`SwitchingGaussianMarkovChain.has_plotter()`

Return True if the node has a plotter

bayespy.nodes.SwitchingGaussianMarkovChain.initialize_from_parameters

`SwitchingGaussianMarkovChain.initialize_from_parameters(*args)`

bayespy.nodes.SwitchingGaussianMarkovChain.initialize_from_prior

`SwitchingGaussianMarkovChain.initialize_from_prior()`

bayespy.nodes.SwitchingGaussianMarkovChain.initialize_from_random

`SwitchingGaussianMarkovChain.initialize_from_random()`

Set the variable to a random sample from the current distribution.

bayespy.nodes.SwitchingGaussianMarkovChain.initialize_from_value

`SwitchingGaussianMarkovChain.initialize_from_value(x, *args)`

bayespy.nodes.SwitchingGaussianMarkovChain.load

`SwitchingGaussianMarkovChain.load(group)`

Load the state of the node from a HDF5 file.

bayespy.nodes.SwitchingGaussianMarkovChain.logpdf

SwitchingGaussianMarkovChain.**logpdf**(*X*, *mask=True*)
Compute the log probability density function Q(X) of this node.

bayespy.nodes.SwitchingGaussianMarkovChain.lower_bound_contribution

SwitchingGaussianMarkovChain.**lower_bound_contribution**(*gradient=False*, *ignore_masked=True*)
Compute E[log p(X|parents) - log q(X)]

If deterministic annealing is used, the term E[-log q(X)] is divided by the anneling coefficient. That is, phi and cgf of q are multiplied by the temperature (inverse annealing coefficient).

bayespy.nodes.SwitchingGaussianMarkovChain.lowerbound

SwitchingGaussianMarkovChain.**lowerbound**()

bayespy.nodes.SwitchingGaussianMarkovChain.move_plates

SwitchingGaussianMarkovChain.**move_plates**(*from_plate*, *to_plate*)

bayespy.nodes.SwitchingGaussianMarkovChain.observe

SwitchingGaussianMarkovChain.**observe**(*x*, **args*, *mask=True*)
Fix moments, compute f and propagate mask.

bayespy.nodes.SwitchingGaussianMarkovChain.pdf

SwitchingGaussianMarkovChain.**pdf**(*X*, *mask=True*)
Compute the probability density function of this node.

bayespy.nodes.SwitchingGaussianMarkovChain.plot

SwitchingGaussianMarkovChain.**plot**(*fig=None*, ***kwargs*)
Plot the node distribution using the plotter of the node

Because the distributions are in general very difficult to plot, the user must specify some functions which performs the plotting as wanted. See, for instance, bayespy.plot.plotting for available plotters, that is, functions that perform plotting for a node.

bayespy.nodes.SwitchingGaussianMarkovChain.random

SwitchingGaussianMarkovChain.**random**()
Draw a random sample from the distribution.

bayespy.nodes.SwitchingGaussianMarkovChain.rotate

SwitchingGaussianMarkovChain.**rotate**(*R*, *inv=None*, *logdet=None*)

bayespy.nodes.SwitchingGaussianMarkovChain.save

SwitchingGaussianMarkovChain.**save**(*group*)

Save the state of the node into a HDF5 file.

group can be the root

bayespy.nodes.SwitchingGaussianMarkovChain.set_parameters

SwitchingGaussianMarkovChain.**set_parameters**(*x*)

Set the parameters of the VB distribution.

The parameters should be such that they can be used for optimization, that is, use log transformation for positive parameters.

bayespy.nodes.SwitchingGaussianMarkovChain.set_plotter

SwitchingGaussianMarkovChain.**set_plotter**(*plotter*)

bayespy.nodes.SwitchingGaussianMarkovChain.show

SwitchingGaussianMarkovChain.**show**()

bayespy.nodes.SwitchingGaussianMarkovChain.unobserve

SwitchingGaussianMarkovChain.**unobserve**()

bayespy.nodes.SwitchingGaussianMarkovChain.update

SwitchingGaussianMarkovChain.**update**(*annealing=1.0*)

Attributes

dims	
plates	
plates_multiplier	Plate multiplier is applied to messages to parents

bayespy.nodes.SwitchingGaussianMarkovChain.dims

SwitchingGaussianMarkovChain.**dims** = None

bayespy.nodes.SwitchingGaussianMarkovChain.plates

`SwitchingGaussianMarkovChain.`**`plates`** = **None**

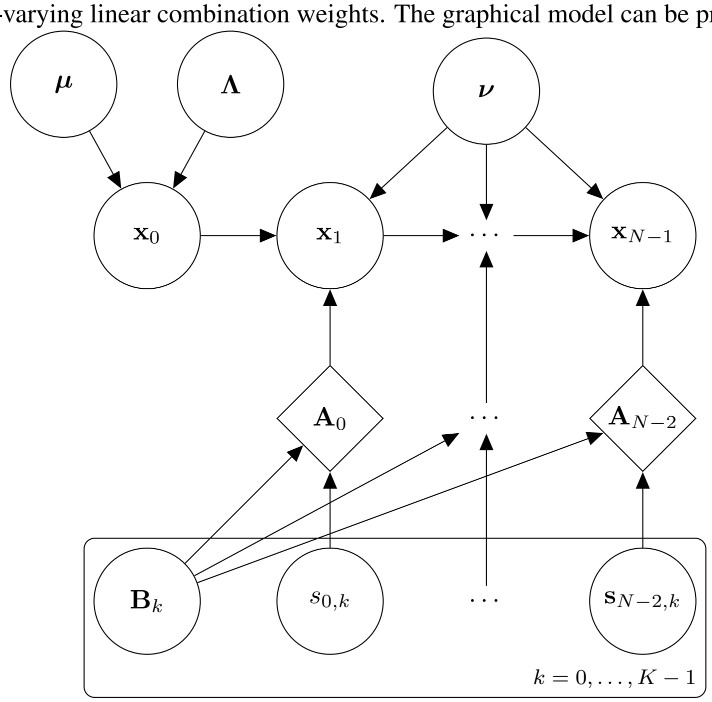
bayespy.nodes.SwitchingGaussianMarkovChain.plates_multiplier

`SwitchingGaussianMarkovChain.`**`plates_multiplier`**
Plate multiplier is applied to messages to parents

bayespy.nodes.VaryingGaussianMarkovChain

class `bayespy.nodes.`**`VaryingGaussianMarkovChain`**(*mu*, *Lambda*, *B*, *S*, *nu*, *n=None*, ***kwargs*)
Node for Gaussian Markov chain random variables with time-varying dynamics.

The node models a sequence of Gaussian variables $\mathbf{x}_0, \ldots, \mathbf{x}_{N-1}$ with linear Markovian dynamics. The time variability of the dynamics is obtained by modelling the state dynamics matrix as a linear combination of a set of matrices with time-varying linear combination weights. The graphical model can be presented as:



where $\boldsymbol{\mu}$ and $\boldsymbol{\Lambda}$ are the mean and the precision matrix of the initial state, $\boldsymbol{\nu}$ is the precision of the innovation noise, and $\mathbf{A}_n$ are the state dynamics matrix obtained by mixing matrices $\mathbf{B}_k$ with weights $s_{n,k}$.

The probability distribution is

$$p(\mathbf{x}_0, \ldots, \mathbf{x}_{N-1}) = p(\mathbf{x}_0) \prod_{n=1}^{N-1} p(\mathbf{x}_n | \mathbf{x}_{n-1})$$

where

$$\begin{aligned} p(\mathbf{x}_0) &= \mathcal{N}(\mathbf{x}_0 | \boldsymbol{\mu}, \boldsymbol{\Lambda}) \\ p(\mathbf{x}_n | \mathbf{x}_{n-1}) &= \mathcal{N}(\mathbf{x}_n | \mathbf{A}_{n-1}\mathbf{x}_{n-1}, \mathrm{diag}(\boldsymbol{\nu})), \quad \text{for } n = 1, \ldots, N-1, \\ \mathbf{A}_n &= \sum_{k=0}^{K-1} s_{n,k} \mathbf{B}_k, \quad \text{for } n = 0, \ldots, N-2. \end{aligned}$$

Parameters **mu** : Gaussian-like node or (...,D)-array

$\boldsymbol{\mu}$, mean of x_0, D-dimensional with plates (...)

Lambda : Wishart-like node or (...,D,D)-array

$\mathbf{\Lambda}$, precision matrix of x_0, $D \times D$ -dimensional with plates (...)

B : Gaussian-like node or (...,D,D,K)-array

$\{\mathbf{B}_k\}_{k=0}^{K-1}$, a set of state dynamics matrix, $D \times K$-dimensional with plates (...,D)

S : Gaussian-like node or (...,N-1,K)-array

$\{\mathbf{s}_0, \dots, \mathbf{s}_{N-2}\}$, time-varying weights of the linear combination, K-dimensional with plates (...,N-1)

nu : gamma-like node or (...,D)-array

$\boldsymbol{\nu}$, diagonal elements of the precision of the innovation process, plates (...,D)

n : int, optional

N, the length of the chain. Must be given if $\mathbf{S}$ does not have plates over the time domain (which would not make sense).

See also:

Gaussian, GaussianARD, Wishart, Gamma, GaussianMarkovChain, SwitchingGaussianMarkovChain

Notes

Equivalent model block can be constructed with GaussianMarkovChain by explicitly using SumMultiply to compute the linear combination. However, that approach is not very efficient for large datasets because it does not utilize the structure of $\mathbf{A}_n$, thus it explicitly computes huge moment arrays.

References

[8]

__init__(*mu, Lambda, B, S, nu, n=None, **kwargs*)
Create VaryingGaussianMarkovChain node.

Methods

__init__(mu, Lambda, B, S, nu[, n])	Create VaryingGaussianMarkovChain node.
add_plate_axis(to_plate)	
broadcasting_multiplier(plates, *args)	
delete()	Delete this node and the children
get_gradient(rg)	Computes gradient with respect to the natural parameters.
get_mask()	
get_moments()	
get_parameters()	Return parameters of the VB distribution.
get_riemannian_gradient()	Computes the Riemannian/natural gradient.
get_shape(ind)	
has_plotter()	Return True if the node has a plotter

Continued on next page

Table 5.44 – continued from previous page

initialize_from_parameters(*args)	
initialize_from_prior()	
initialize_from_random()	Set the variable to a random sample from the current distribution.
initialize_from_value(x, *args)	
load(group)	Load the state of the node from a HDF5 file.
logpdf(X[, mask])	Compute the log probability density function Q(X) of this node.
lower_bound_contribution([gradient, ...])	Compute E[log p(X|parents) - log q(X)]
lowerbound()	
move_plates(from_plate, to_plate)	
observe(x, *args[, mask])	Fix moments, compute f and propagate mask.
pdf(X[, mask])	Compute the probability density function of this node.
plot([fig])	Plot the node distribution using the plotter of the node
random()	Draw a random sample from the distribution.
rotate(R[, inv, logdet])	
save(group)	Save the state of the node into a HDF5 file.
set_parameters(x)	Set the parameters of the VB distribution.
set_plotter(plotter)	
show()	
unobserve()	
update([annealing])	

bayespy.nodes.VaryingGaussianMarkovChain.__init__

`VaryingGaussianMarkovChain.__init__(mu, Lambda, B, S, nu, n=None, **kwargs)`

Create VaryingGaussianMarkovChain node.

bayespy.nodes.VaryingGaussianMarkovChain.add_plate_axis

`VaryingGaussianMarkovChain.add_plate_axis(to_plate)`

bayespy.nodes.VaryingGaussianMarkovChain.broadcasting_multiplier

`VaryingGaussianMarkovChain.broadcasting_multiplier(plates, *args)`

bayespy.nodes.VaryingGaussianMarkovChain.delete

`VaryingGaussianMarkovChain.delete()`

Delete this node and the children

bayespy.nodes.VaryingGaussianMarkovChain.get_gradient

`VaryingGaussianMarkovChain.get_gradient(rg)`

Computes gradient with respect to the natural parameters.

The function takes the Riemannian gradient as an input. This is for three reasons: 1) You probably want to use the Riemannian gradient anyway so this helps avoiding accidental use of this function. 2) The gradient is computed by using the Riemannian gradient and chain rules. 3) Probably you need both Riemannian and normal gradients anyway so you can provide it to this function to avoid re-computing it.

bayespy.nodes.VaryingGaussianMarkovChain.get_mask

`VaryingGaussianMarkovChain.get_mask()`

bayespy.nodes.VaryingGaussianMarkovChain.get_moments

`VaryingGaussianMarkovChain.get_moments()`

bayespy.nodes.VaryingGaussianMarkovChain.get_parameters

`VaryingGaussianMarkovChain.get_parameters()`

Return parameters of the VB distribution.

The parameters should be such that they can be used for optimization, that is, use log transformation for positive parameters.

bayespy.nodes.VaryingGaussianMarkovChain.get_riemannian_gradient

`VaryingGaussianMarkovChain.get_riemannian_gradient()`

Computes the Riemannian/natural gradient.

bayespy.nodes.VaryingGaussianMarkovChain.get_shape

`VaryingGaussianMarkovChain.get_shape(ind)`

bayespy.nodes.VaryingGaussianMarkovChain.has_plotter

`VaryingGaussianMarkovChain.has_plotter()`

Return True if the node has a plotter

bayespy.nodes.VaryingGaussianMarkovChain.initialize_from_parameters

`VaryingGaussianMarkovChain.initialize_from_parameters(*args)`

bayespy.nodes.VaryingGaussianMarkovChain.initialize_from_prior

`VaryingGaussianMarkovChain.initialize_from_prior()`

bayespy.nodes.VaryingGaussianMarkovChain.initialize_from_random

`VaryingGaussianMarkovChain.initialize_from_random()`

Set the variable to a random sample from the current distribution.

bayespy.nodes.VaryingGaussianMarkovChain.initialize_from_value

`VaryingGaussianMarkovChain.initialize_from_value(x, *args)`

bayespy.nodes.VaryingGaussianMarkovChain.load

VaryingGaussianMarkovChain.**load**(*group*)

Load the state of the node from a HDF5 file.

bayespy.nodes.VaryingGaussianMarkovChain.logpdf

VaryingGaussianMarkovChain.**logpdf**(*X*, *mask=True*)

Compute the log probability density function Q(X) of this node.

bayespy.nodes.VaryingGaussianMarkovChain.lower_bound_contribution

VaryingGaussianMarkovChain.**lower_bound_contribution**(*gradient=False*, *ignore_masked=True*)

Compute E[log p(X|parents) - log q(X)]

If deterministic annealing is used, the term E[-log q(X)] is divided by the anneling coefficient. That is, phi and cgf of q are multiplied by the temperature (inverse annealing coefficient).

bayespy.nodes.VaryingGaussianMarkovChain.lowerbound

VaryingGaussianMarkovChain.**lowerbound**()

bayespy.nodes.VaryingGaussianMarkovChain.move_plates

VaryingGaussianMarkovChain.**move_plates**(*from_plate*, *to_plate*)

bayespy.nodes.VaryingGaussianMarkovChain.observe

VaryingGaussianMarkovChain.**observe**(*x*, **args*, *mask=True*)

Fix moments, compute f and propagate mask.

bayespy.nodes.VaryingGaussianMarkovChain.pdf

VaryingGaussianMarkovChain.**pdf**(*X*, *mask=True*)

Compute the probability density function of this node.

bayespy.nodes.VaryingGaussianMarkovChain.plot

VaryingGaussianMarkovChain.**plot**(*fig=None*, ***kwargs*)

Plot the node distribution using the plotter of the node

Because the distributions are in general very difficult to plot, the user must specify some functions which performs the plotting as wanted. See, for instance, bayespy.plot.plotting for available plotters, that is, functions that perform plotting for a node.

bayespy.nodes.VaryingGaussianMarkovChain.random

`VaryingGaussianMarkovChain.random()`

Draw a random sample from the distribution.

bayespy.nodes.VaryingGaussianMarkovChain.rotate

`VaryingGaussianMarkovChain.rotate(R, inv=None, logdet=None)`

bayespy.nodes.VaryingGaussianMarkovChain.save

`VaryingGaussianMarkovChain.save(group)`

Save the state of the node into a HDF5 file.

group can be the root

bayespy.nodes.VaryingGaussianMarkovChain.set_parameters

`VaryingGaussianMarkovChain.set_parameters(x)`

Set the parameters of the VB distribution.

The parameters should be such that they can be used for optimization, that is, use log transformation for positive parameters.

bayespy.nodes.VaryingGaussianMarkovChain.set_plotter

`VaryingGaussianMarkovChain.set_plotter(plotter)`

bayespy.nodes.VaryingGaussianMarkovChain.show

`VaryingGaussianMarkovChain.show()`

bayespy.nodes.VaryingGaussianMarkovChain.unobserve

`VaryingGaussianMarkovChain.unobserve()`

bayespy.nodes.VaryingGaussianMarkovChain.update

`VaryingGaussianMarkovChain.update(annealing=1.0)`

Attributes

`dims`	
`plates`	
`plates_multiplier`	Plate multiplier is applied to messages to parents

bayespy.nodes.VaryingGaussianMarkovChain.dims

VaryingGaussianMarkovChain.**dims** = None

bayespy.nodes.VaryingGaussianMarkovChain.plates

VaryingGaussianMarkovChain.**plates** = None

bayespy.nodes.VaryingGaussianMarkovChain.plates_multiplier

VaryingGaussianMarkovChain.**plates_multiplier**
Plate multiplier is applied to messages to parents

Other stochastic nodes:

Mixture(z, node_class, *params[, cluster_plate])	Node for exponential family mixture variables.

bayespy.nodes.Mixture

class bayespy.nodes.**Mixture**(*z*, *node_class*, **params*, *cluster_plate=-1*, ***kwargs*)
Node for exponential family mixture variables.

The node represents a random variable which is sampled from a mixture distribution. It is possible to mix any exponential family distribution. The probability density function is

$$p(x|z=k,\boldsymbol{\theta}_0,\dots,\boldsymbol{\theta}_{K-1}) = \phi(x|\boldsymbol{\theta}_k),$$

where ϕ is the probability density function of the mixed exponential family distribution and $\boldsymbol{\theta}_0,\dots,\boldsymbol{\theta}_{K-1}$ are the parameters of each cluster. For instance, ϕ could be the Gaussian probability density function $\mathcal{N}$ and $\boldsymbol{\theta}_k = \{\boldsymbol{\mu}_k, \boldsymbol{\Lambda}_k\}$ where $\boldsymbol{\mu}_k$ and $\boldsymbol{\Lambda}_k$ are the mean vector and precision matrix for cluster k.

Parameters **z** : categorical-like node or array

z, cluster assignment

node_class : stochastic exponential family node class

Mixed distribution

params : types specified by the mixed distribution

Parameters of the mixed distribution. If some parameters should vary between clusters, those parameters' plate axis *cluster_plate* should have a size which equals the number of clusters. For parameters with shared values, that plate axis should have length 1. At least one parameter should vary between clusters.

cluster_plate : int, optional

Negative integer defining which plate axis is used for the clusters in the parameters. That plate axis is ignored from the parameters when considering the plates for this node. By default, mix over the last plate axis.

See also:

Categorical, CategoricalMarkovChain

Examples

A simple 2-dimensional Gaussian mixture model with three clusters for 100 samples can be constructed, for instance, as:

```
from bayespy.nodes import (Dirichlet, Categorical, Mixture,
                           Gaussian, Wishart)
alpha = Dirichlet([1e-3, 1e-3, 1e-3])
Z = Categorical(alpha, plates=(100,))
mu = Gaussian(np.zeros(2), 1e-6*np.identity(2), plates=(3,))
Lambda = Wishart(2, 1e-6*np.identity(2), plates=(3,))
X = Mixture(Z, Gaussian, mu, Lambda)
```

`__init__`(*z, node_class, *params, cluster_plate=-1, **kwargs*)

Methods

`__init__`(z, node_class, *params[, cluster_plate])	
`add_plate_axis`(to_plate)	
`broadcasting_multiplier`(plates, *args)	
`delete`()	Delete this node and the children
`get_gradient`(rg)	Computes gradient with respect to the natural parameters.
`get_mask`()	
`get_moments`()	
`get_parameters`()	Return parameters of the VB distribution.
`get_riemannian_gradient`()	Computes the Riemannian/natural gradient.
`get_shape`(ind)	
`has_plotter`()	Return True if the node has a plotter
`initialize_from_parameters`(*args)	
`initialize_from_prior`()	
`initialize_from_random`()	Set the variable to a random sample from the current distribution.
`initialize_from_value`(x, *args)	
`integrated_logpdf_from_parents`(x, index)	Approximates the posterior predictive pdf int p(x|parents) q(parents) dparents
`load`(group)	Load the state of the node from a HDF5 file.
`logpdf`(X[, mask])	Compute the log probability density function Q(X) of this node.
`lower_bound_contribution`([gradient, ...])	Compute E[log p(X|parents) - log q(X)]
`lowerbound`()	
`move_plates`(from_plate, to_plate)	
`observe`(x, *args[, mask])	Fix moments, compute f and propagate mask.
`pdf`(X[, mask])	Compute the probability density function of this node.
`plot`([fig])	Plot the node distribution using the plotter of the node
`random`()	Draw a random sample from the distribution.
`save`(group)	Save the state of the node into a HDF5 file.
`set_parameters`(x)	Set the parameters of the VB distribution.
`set_plotter`(plotter)	
`unobserve`()	
`update`([annealing])	

bayespy.nodes.Mixture.__init__

`Mixture.__init__`(*z, node_class, *params, cluster_plate=-1, **kwargs*)

bayespy.nodes.Mixture.add_plate_axis

Mixture.**add_plate_axis**(*to_plate*)

bayespy.nodes.Mixture.broadcasting_multiplier

Mixture.**broadcasting_multiplier**(*plates*, **args*)

bayespy.nodes.Mixture.delete

Mixture.**delete**()

Delete this node and the children

bayespy.nodes.Mixture.get_gradient

Mixture.**get_gradient**(*rg*)

Computes gradient with respect to the natural parameters.

The function takes the Riemannian gradient as an input. This is for three reasons: 1) You probably want to use the Riemannian gradient anyway so this helps avoiding accidental use of this function. 2) The gradient is computed by using the Riemannian gradient and chain rules. 3) Probably you need both Riemannian and normal gradients anyway so you can provide it to this function to avoid re-computing it.

bayespy.nodes.Mixture.get_mask

Mixture.**get_mask**()

bayespy.nodes.Mixture.get_moments

Mixture.**get_moments**()

bayespy.nodes.Mixture.get_parameters

Mixture.**get_parameters**()

Return parameters of the VB distribution.

The parameters should be such that they can be used for optimization, that is, use log transformation for positive parameters.

bayespy.nodes.Mixture.get_riemannian_gradient

Mixture.**get_riemannian_gradient**()

Computes the Riemannian/natural gradient.

bayespy.nodes.Mixture.get_shape

Mixture.**get_shape**(*ind*)

bayespy.nodes.Mixture.has_plotter

Mixture.**has_plotter**()
Return True if the node has a plotter

bayespy.nodes.Mixture.initialize_from_parameters

Mixture.**initialize_from_parameters**(**args*)

bayespy.nodes.Mixture.initialize_from_prior

Mixture.**initialize_from_prior**()

bayespy.nodes.Mixture.initialize_from_random

Mixture.**initialize_from_random**()
Set the variable to a random sample from the current distribution.

bayespy.nodes.Mixture.initialize_from_value

Mixture.**initialize_from_value**(*x*, **args*)

bayespy.nodes.Mixture.integrated_logpdf_from_parents

Mixture.**integrated_logpdf_from_parents**(*x*, *index*)
Approximates the posterior predictive pdf int p(x|parents) q(parents) dparents in log-scale as int q(parents_i) exp(int q(**parents**_i) log p(x|parents) **dparents**_i) dparents_i.

bayespy.nodes.Mixture.load

Mixture.**load**(*group*)
Load the state of the node from a HDF5 file.

bayespy.nodes.Mixture.logpdf

Mixture.**logpdf**(*X*, *mask=True*)
Compute the log probability density function Q(X) of this node.

bayespy.nodes.Mixture.lower_bound_contribution

Mixture.**lower_bound_contribution**(*gradient=False*, *ignore_masked=True*)
Compute E[log p(X|parents) - log q(X)]

If deterministic annealing is used, the term E[-log q(X)] is divided by the anneling coefficient. That is, phi and cgf of q are multiplied by the temperature (inverse annealing coefficient).

bayespy.nodes.Mixture.lowerbound

`Mixture.lowerbound()`

bayespy.nodes.Mixture.move_plates

`Mixture.move_plates(from_plate, to_plate)`

bayespy.nodes.Mixture.observe

`Mixture.observe(x, *args, mask=True)`
Fix moments, compute f and propagate mask.

bayespy.nodes.Mixture.pdf

`Mixture.pdf(X, mask=True)`
Compute the probability density function of this node.

bayespy.nodes.Mixture.plot

`Mixture.plot(fig=None, **kwargs)`
Plot the node distribution using the plotter of the node

Because the distributions are in general very difficult to plot, the user must specify some functions which performs the plotting as wanted. See, for instance, bayespy.plot.plotting for available plotters, that is, functions that perform plotting for a node.

bayespy.nodes.Mixture.random

`Mixture.random()`
Draw a random sample from the distribution.

bayespy.nodes.Mixture.save

`Mixture.save(group)`
Save the state of the node into a HDF5 file.

group can be the root

bayespy.nodes.Mixture.set_parameters

`Mixture.set_parameters(x)`
Set the parameters of the VB distribution.

The parameters should be such that they can be used for optimization, that is, use log transformation for positive parameters.

bayespy.nodes.Mixture.set_plotter

Mixture.**set_plotter**(*plotter*)

bayespy.nodes.Mixture.unobserve

Mixture.**unobserve**()

bayespy.nodes.Mixture.update

Mixture.**update**(*annealing=1.0*)

Attributes

dims	
plates	
plates_multiplier	Plate multiplier is applied to messages to parents

bayespy.nodes.Mixture.dims

Mixture.**dims** = None

bayespy.nodes.Mixture.plates

Mixture.**plates** = None

bayespy.nodes.Mixture.plates_multiplier

Mixture.**plates_multiplier**
Plate multiplier is applied to messages to parents

5.1.2 Deterministic nodes

Dot(*args, **kwargs)	Node for computing inner product of several Gaussian vectors.
SumMultiply(*args[, iterator_axis])	Node for computing general products and sums of Gaussian nodes.
Gate(Z, X[, gated_plate, moments])	Deterministic gating of one node.

bayespy.nodes.Dot

bayespy.nodes.**Dot**(**args*, ***kwargs*)
Node for computing inner product of several Gaussian vectors.

This is a simple wrapper of the much more general SumMultiply. For now, it is here for backward compatibility.

bayespy.nodes.SumMultiply

class `bayespy.nodes.SumMultiply`(**args*, *iterator_axis=None*, ***kwargs*)

Node for computing general products and sums of Gaussian nodes.

The node is similar to *numpy.einsum*, which is a very general function for computing dot products, sums, products and other sums of products of arrays.

For instance, the equivalent of

```
np.einsum('abc,bd,ca->da', X, Y, Z)
```

would be given as

```
SumMultiply('abc,bd,ca->da', X, Y, Z)
```

or

```
SumMultiply(X, [0,1,2], Y, [1,3], Z, [2,0], [3,0])
```

which is similar to the other syntax of numpy.einsum.

This node operates similarly as numpy.einsum. However, you must use all the elements of each node, that is, an operation like np.einsum('ii->i',X) is not allowed. Thus, for each node, each axis must be given unique id. The id identifies which axes correspond to which axes between the different nodes. Also, Ellipsis ('...') is not yet supported for simplicity. It would also have some problems with constant inputs (because how to determine ndim), so let us just forget it for now.

Each output axis must appear in the input mappings.

The keys must refer to variable dimension axes only, not plate axes.

The input nodes may be Gaussian-gamma (isotropic) nodes.

The output message is Gaussian-gamma (isotropic) if any of the input nodes is Gaussian-gamma.

Notes

This operation can be extremely slow if not used wisely. For large and complex operations, it is sometimes more efficient to split the operation into multiple nodes. For instance, the example above could probably be computed faster by

```
XZ = SumMultiply(X, [0,1,2], Z, [2,0], [0,1])
F = SumMultiply(XZ, [0,1], Y, [1,2], [2,0])
```

because the third axis ('c') could be summed out already in the first operation. This same effect applies also to numpy.einsum in general.

Examples

Sum over the rows: 'ij->j'

Inner product of three vectors: 'i,i,i'

Matrix-vector product: 'ij,j->i'

Matrix-matrix product: 'ik,kj->ij'

Outer product: 'i,j->ij'

Vector-matrix-vector product: 'i,ij,j'

`__init__(Node1, map1, Node2, map2, ..., NodeN, mapN[, map_out ])`

Methods

`__init__`(Node1, map1, Node2, map2, ..., ...)	
`add_plate_axis`(to_plate)	
`broadcasting_multiplier`(plates, *args)	
`delete`()	Delete this node and the children
`get_mask`()	
`get_moments`()	
`get_parameters`()	
`get_shape`(ind)	
`has_plotter`()	Return True if the node has a plotter
`lower_bound_contribution`([gradient])	
`move_plates`(from_plate, to_plate)	
`plot`([fig])	Plot the node distribution using the plotter of the node
`set_plotter`(plotter)	

bayespy.nodes.SumMultiply.__init__

`SumMultiply.__init__(Node1, map1, Node2, map2, ..., NodeN, mapN[, map_out ])`

bayespy.nodes.SumMultiply.add_plate_axis

`SumMultiply.add_plate_axis(to_plate)`

bayespy.nodes.SumMultiply.broadcasting_multiplier

`SumMultiply.broadcasting_multiplier(plates, *args)`

bayespy.nodes.SumMultiply.delete

`SumMultiply.delete()`

Delete this node and the children

bayespy.nodes.SumMultiply.get_mask

`SumMultiply.get_mask()`

bayespy.nodes.SumMultiply.get_moments

`SumMultiply.get_moments()`

bayespy.nodes.SumMultiply.get_parameters

`SumMultiply.get_parameters()`

bayespy.nodes.SumMultiply.get_shape

SumMultiply.**get_shape**(*ind*)

bayespy.nodes.SumMultiply.has_plotter

SumMultiply.**has_plotter**()
Return True if the node has a plotter

bayespy.nodes.SumMultiply.lower_bound_contribution

SumMultiply.**lower_bound_contribution**(*gradient=False*, ***kwargs*)

bayespy.nodes.SumMultiply.move_plates

SumMultiply.**move_plates**(*from_plate*, *to_plate*)

bayespy.nodes.SumMultiply.plot

SumMultiply.**plot**(*fig=None*, ***kwargs*)
Plot the node distribution using the plotter of the node

Because the distributions are in general very difficult to plot, the user must specify some functions which performs the plotting as wanted. See, for instance, bayespy.plot.plotting for available plotters, that is, functions that perform plotting for a node.

bayespy.nodes.SumMultiply.set_plotter

SumMultiply.**set_plotter**(*plotter*)

Attributes

plates	
plates_multiplier	Plate multiplier is applied to messages to parents

bayespy.nodes.SumMultiply.plates

SumMultiply.**plates** = None

bayespy.nodes.SumMultiply.plates_multiplier

SumMultiply.**plates_multiplier**
Plate multiplier is applied to messages to parents

bayespy.nodes.Gate

class bayespy.nodes.**Gate**(*Z*, *X*, *gated_plate=-1*, *moments=None*, ***kwargs*)

Deterministic gating of one node.

Gating is performed over one plate axis.

Note: You should not use gating for several variables which parents of a same node if the gates use the same gate assignments. In such case, the results will be wrong. The reason is a general one: A stochastic node may not be a parent of another node via several paths unless at most one path has no other stochastic nodes between them.

__init__(*Z*, *X*, *gated_plate=-1*, *moments=None*, ***kwargs*)

Methods

`__init__`(Z, X[, gated_plate, moments])	
`add_plate_axis`(to_plate)	
`broadcasting_multiplier`(plates, *args)	
`delete`()	Delete this node and the children
`get_mask`()	
`get_moments`()	
`get_shape`(ind)	
`has_plotter`()	Return True if the node has a plotter
`lower_bound_contribution`([gradient])	
`move_plates`(from_plate, to_plate)	
`plot`([fig])	Plot the node distribution using the plotter of the node
`set_plotter`(plotter)	

bayespy.nodes.Gate.__init__

Gate.**__init__**(*Z*, *X*, *gated_plate=-1*, *moments=None*, ***kwargs*)

bayespy.nodes.Gate.add_plate_axis

Gate.**add_plate_axis**(*to_plate*)

bayespy.nodes.Gate.broadcasting_multiplier

Gate.**broadcasting_multiplier**(*plates*, **args*)

bayespy.nodes.Gate.delete

Gate.**delete**()

Delete this node and the children

bayespy.nodes.Gate.get_mask

Gate.**get_mask**()

bayespy.nodes.Gate.get_moments

`Gate.get_moments()`

bayespy.nodes.Gate.get_shape

`Gate.get_shape(ind)`

bayespy.nodes.Gate.has_plotter

`Gate.has_plotter()`

Return True if the node has a plotter

bayespy.nodes.Gate.lower_bound_contribution

`Gate.lower_bound_contribution(gradient=False, **kwargs)`

bayespy.nodes.Gate.move_plates

`Gate.move_plates(from_plate, to_plate)`

bayespy.nodes.Gate.plot

`Gate.plot(fig=None, **kwargs)`

Plot the node distribution using the plotter of the node

Because the distributions are in general very difficult to plot, the user must specify some functions which performs the plotting as wanted. See, for instance, bayespy.plot.plotting for available plotters, that is, functions that perform plotting for a node.

bayespy.nodes.Gate.set_plotter

`Gate.set_plotter(plotter)`

Attributes

plates	
plates_multiplier	Plate multiplier is applied to messages to parents

bayespy.nodes.Gate.plates

`Gate.plates = None`

bayespy.nodes.Gate.plates_multiplier

Gate.**plates_multiplier**
Plate multiplier is applied to messages to parents

5.2 bayespy.inference

Package for Bayesian inference engines

5.2.1 Inference engines

VB(*nodes[, tol, autosave_filename, ...])	Variational Bayesian (VB) inference engine

bayespy.inference.VB

class bayespy.inference.**VB**(**nodes*, *tol=1e-05*, *autosave_filename=None*, *autosave_iterations=0*, *callback=None*)
Variational Bayesian (VB) inference engine

Parameters **nodes** : nodes

Nodes that form the model. Must include all at least all stochastic nodes of the model.

tol : double, optional

Convergence criterion. Tolerance for the relative change in the VB lower bound.

autosave_filename : string, optional

Filename for automatic saving

autosave_iterations : int, optional

Iteration interval between each automatic saving

callback : callable, optional

Function which is called after each update iteration step

__init__(**nodes*, *tol=1e-05*, *autosave_filename=None*, *autosave_iterations=0*, *callback=None*)

Methods

__init__(*nodes[, tol, autosave_filename, ...])	
add(x1, x2[, scale])	Add two vectors (in parameter format)
compute_lowerbound([ignore_masked])	
compute_lowerbound_terms(*nodes)	
dot(x1, x2)	Computes dot products of given vectors (in parameter format)
get_gradients(*nodes[, euclidian])	Computes gradients (both Riemannian and normal)
get_iteration_by_nodes()	
get_parameters(*nodes)	Get parameters of the nodes
gradient_step(*nodes[, scale])	Update nodes by taking a gradient ascent step
has_converged([tol])	

Continued on next page

Table 5.55 – continued from previous page

load(*nodes[, filename])	
loglikelihood_lowerbound()	
optimize(*nodes[, maxiter, verbose, method, ...])	Optimize nodes using Riemannian conjugate gradient
pattern_search(*nodes[, collapsed, maxiter])	Perform simple pattern search [4].
plot(*nodes)	Plot the distribution of the given nodes (or all nodes)
plot_iteration_by_nodes([axes, diff])	Plot the cost function per node during the iteration.
save(*nodes[, filename])	
set_annealing(annealing)	Set deterministic annealing from range (0, 1].
set_autosave(filename[, iterations])	
set_callback(callback)	
set_parameters(x, *nodes)	Set parameters of the nodes
update(*nodes[, repeat, plot, tol, verbose])	

bayespy.inference.VB.__init__

VB.**__init__**(**nodes*, *tol=1e-05*, *autosave_filename=None*, *autosave_iterations=0*, *callback=None*)

bayespy.inference.VB.add

VB.**add**(*x1*, *x2*, *scale=1*)

Add two vectors (in parameter format)

bayespy.inference.VB.compute_lowerbound

VB.**compute_lowerbound**(*ignore_masked=True*)

bayespy.inference.VB.compute_lowerbound_terms

VB.**compute_lowerbound_terms**(**nodes*)

bayespy.inference.VB.dot

VB.**dot**(*x1*, *x2*)

Computes dot products of given vectors (in parameter format)

bayespy.inference.VB.get_gradients

VB.**get_gradients**(**nodes*, *euclidian=False*)

Computes gradients (both Riemannian and normal)

bayespy.inference.VB.get_iteration_by_nodes

VB.**get_iteration_by_nodes**()

bayespy.inference.VB.get_parameters

VB.**get_parameters**(**nodes*)

Get parameters of the nodes

bayespy.inference.VB.gradient_step

VB.**gradient_step**(**nodes*, *scale=1.0*)

Update nodes by taking a gradient ascent step

bayespy.inference.VB.has_converged

VB.**has_converged**(*tol=None*)

bayespy.inference.VB.load

VB.**load**(**nodes*, *filename=None*)

bayespy.inference.VB.loglikelihood_lowerbound

VB.**loglikelihood_lowerbound**()

bayespy.inference.VB.optimize

VB.**optimize**(**nodes*, *maxiter=10*, *verbose=True*, *method='fletcher-reeves'*, *riemannian=True*, *collapsed=None*, *tol=None*)

Optimize nodes using Riemannian conjugate gradient

bayespy.inference.VB.pattern_search

VB.**pattern_search**(**nodes*, *collapsed=None*, *maxiter=3*)

Perform simple pattern search [4].

Some of the variables can be collapsed.

bayespy.inference.VB.plot

VB.**plot**(**nodes*)

Plot the distribution of the given nodes (or all nodes)

bayespy.inference.VB.plot_iteration_by_nodes

VB.**plot_iteration_by_nodes**(*axes=None*, *diff=False*)

Plot the cost function per node during the iteration.

Handy tool for debugging.

bayespy.inference.VB.save

VB.**save**(**nodes*, *filename=None*)

bayespy.inference.VB.set_annealing

VB.**set_annealing**(*annealing*)

Set deterministic annealing from range (0, 1].

With 1, no annealing, standard updates.

With smaller values, entropy has more weight and model probability equations less. With 0, one would obtain improper uniform distributions.

bayespy.inference.VB.set_autosave

VB.**set_autosave**(*filename*, *iterations=None*)

bayespy.inference.VB.set_callback

VB.**set_callback**(*callback*)

bayespy.inference.VB.set_parameters

VB.**set_parameters**(*x*, **nodes*)

Set parameters of the nodes

bayespy.inference.VB.update

VB.**update**(**nodes*, *repeat=1*, *plot=False*, *tol=None*, *verbose=True*)

Attributes

ignore_bound_checks

bayespy.inference.VB.ignore_bound_checks

VB.**ignore_bound_checks**

5.2.2 Parameter expansions

vmp.transformations.RotationOptimizer(...)	Optimizer for rotation parameter expansion in state-spa
vmp.transformations.RotateGaussian(X)	Rotation parameter expansion for bayespy.nodes.(
vmp.transformations.RotateGaussianARD(X, *alpha)	Rotation parameter expansion for bayespy.nodes.(
vmp.transformations.RotateGaussianMarkovChain(X, ...)	Rotation parameter expansion for bayespy.nodes.(

Table 5.57 – continued from previous page

vmp.transformations.RotateSwitchingMarkovChain(X, ...)	Rotation for bayespy.nodes.VaryingGaussian
vmp.transformations.RotateVaryingMarkovChain(X, ...)	Rotation for bayespy.nodes.SwitchingGauss
vmp.transformations.RotateMultiple(*rotators)	Identical parameter expansion for several nodes simulta

bayespy.inference.vmp.transformations.RotationOptimizer

class bayespy.inference.vmp.transformations.**RotationOptimizer**(*block1*, *block2*, *D*)

Optimizer for rotation parameter expansion in state-space models

Rotates one model block with $\mathbf{R}$ and one model block with $\mathbf{R}^{-1}$.

Parameters **block1** : rotator object

The first rotation parameter expansion object

block2 : rotator object

The second rotation parameter expansion object

D : int

Dimensionality of the latent space

References

[7], [6]

__init__(*block1*, *block2*, *D*)

Methods

__init__(block1, block2, D)	
rotate([maxiter, check_gradient, verbose, ...])	Optimize the rotation of two separate model blocks jointly.

bayespy.inference.vmp.transformations.RotationOptimizer.__init__

RotationOptimizer.**__init__**(*block1*, *block2*, *D*)

bayespy.inference.vmp.transformations.RotationOptimizer.rotate

RotationOptimizer.**rotate**(*maxiter=10*, *check_gradient=False*, *verbose=False*, *check_bound=False*)

Optimize the rotation of two separate model blocks jointly.

If some variable is the dot product of two Gaussians, rotating the two Gaussians optimally can make the inference algorithm orders of magnitude faster.

First block is rotated with $\mathbf{R}$ and the second with $\mathbf{R}^{-T}$.

Blocks must have methods: *bound(U,s,V)* and *rotate(R)*.

bayespy.inference.vmp.transformations.RotateGaussian

class bayespy.inference.vmp.transformations.**RotateGaussian**(*X*)
Rotation parameter expansion for bayespy.nodes.Gaussian

__init__(*X*)

Methods

__init__(X)	
bound(R[, logdet, inv])	
get_bound_terms(R[, logdet, inv])	
nodes()	
rotate(R[, inv, logdet])	
setup()	This method should be called just before optimization.

bayespy.inference.vmp.transformations.RotateGaussian.__init__

RotateGaussian.**__init__**(*X*)

bayespy.inference.vmp.transformations.RotateGaussian.bound

RotateGaussian.**bound**(*R*, *logdet=None*, *inv=None*)

bayespy.inference.vmp.transformations.RotateGaussian.get_bound_terms

RotateGaussian.**get_bound_terms**(*R*, *logdet=None*, *inv=None*)

bayespy.inference.vmp.transformations.RotateGaussian.nodes

RotateGaussian.**nodes**()

bayespy.inference.vmp.transformations.RotateGaussian.rotate

RotateGaussian.**rotate**(*R*, *inv=None*, *logdet=None*)

bayespy.inference.vmp.transformations.RotateGaussian.setup

RotateGaussian.**setup**()
This method should be called just before optimization.

bayespy.inference.vmp.transformations.RotateGaussianARD

class bayespy.inference.vmp.transformations.**RotateGaussianARD**(*X*, **alpha*, *axis=-1*, *precompute=False*, *subset=None*)
Rotation parameter expansion for bayespy.nodes.GaussianARD

The model:

alpha ~ N(a, b) X ~ N(mu, alpha)

X can be an array (e.g., GaussianARD).

Transform q(X) and q(alpha) by rotating X.

Requirements: * X and alpha do not contain any observed values

__init__(*X*, **alpha*, *axis=-1*, *precompute=False*, *subset=None*)
Precompute tells whether to compute some moments once in the setup function instead of every time in the bound function. However, they are computed a bit differently in the bound function so it can be useful too. Precomputation is probably beneficial only when there are large axes that are not rotated (by R nor Q) and they are not contained in the plates of alpha, and the dimensions for R and Q are quite small.

Methods

__init__(X, *alpha[, axis, precompute, subset])	Precompute tells whether to compute some moments once in the setup function ins
bound(R[, logdet, inv, Q])	
get_bound_terms(R[, logdet, inv, Q])	
nodes()	
rotate(R[, inv, logdet, Q])	
setup([plate_axis])	This method should be called just before optimization.

bayespy.inference.vmp.transformations.RotateGaussianARD.__init__

RotateGaussianARD.**__init__**(*X*, **alpha*, *axis=-1*, *precompute=False*, *subset=None*)
Precompute tells whether to compute some moments once in the setup function instead of every time in the bound function. However, they are computed a bit differently in the bound function so it can be useful too. Precomputation is probably beneficial only when there are large axes that are not rotated (by R nor Q) and they are not contained in the plates of alpha, and the dimensions for R and Q are quite small.

bayespy.inference.vmp.transformations.RotateGaussianARD.bound

RotateGaussianARD.**bound**(*R*, *logdet=None*, *inv=None*, *Q=None*)

bayespy.inference.vmp.transformations.RotateGaussianARD.get_bound_terms

RotateGaussianARD.**get_bound_terms**(*R*, *logdet=None*, *inv=None*, *Q=None*)

bayespy.inference.vmp.transformations.RotateGaussianARD.nodes

RotateGaussianARD.**nodes**()

bayespy.inference.vmp.transformations.RotateGaussianARD.rotate

RotateGaussianARD.**rotate**(*R*, *inv=None*, *logdet=None*, *Q=None*)

bayespy.inference.vmp.transformations.RotateGaussianARD.setup

RotateGaussianARD.**setup**(*plate_axis=None*)

This method should be called just before optimization.

For efficiency, sum over axes that are not in mu, alpha nor rotation.

If using Q, set rotate_plates to True.

bayespy.inference.vmp.transformations.RotateGaussianMarkovChain

class bayespy.inference.vmp.transformations.**RotateGaussianMarkovChain**(*X*, **args*)

Rotation parameter expansion for bayespy.nodes.GaussianMarkovChain

Assume the following model.

Constant, unit isotropic innovation noise. Unit variance only?

Maybe: Assume innovation noise with unit variance? Would it help make this function more general with respect to A.

TODO: Allow constant A or not rotating A.

A may vary in time.

Shape of A: (N,D,D) Shape of AA: (N,D,D,D)

No plates for X.

__init__(*X*, **args*)

Methods

__init__(X, *args)	
bound(R[, logdet, inv])	
get_bound_terms(R[, logdet, inv])	
nodes()	
rotate(R[, inv, logdet])	
setup()	This method should be called just before optimization.

bayespy.inference.vmp.transformations.RotateGaussianMarkovChain.__init__

RotateGaussianMarkovChain.**__init__**(*X*, **args*)

bayespy.inference.vmp.transformations.RotateGaussianMarkovChain.bound

RotateGaussianMarkovChain.**bound**(*R*, *logdet=None*, *inv=None*)

bayespy.inference.vmp.transformations.RotateGaussianMarkovChain.get_bound_terms

RotateGaussianMarkovChain.**get_bound_terms**(*R*, *logdet=None*, *inv=None*)

bayespy.inference.vmp.transformations.RotateGaussianMarkovChain.nodes

RotateGaussianMarkovChain.**nodes**()

bayespy.inference.vmp.transformations.RotateGaussianMarkovChain.rotate

RotateGaussianMarkovChain.**rotate**(*R*, *inv=None*, *logdet=None*)

bayespy.inference.vmp.transformations.RotateGaussianMarkovChain.setup

RotateGaussianMarkovChain.**setup**()

This method should be called just before optimization.

bayespy.inference.vmp.transformations.RotateSwitchingMarkovChain

class bayespy.inference.vmp.transformations.**RotateSwitchingMarkovChain**(*X*, *B*, *Z*, *B_rotator*)

Rotation for bayespy.nodes.VaryingGaussianMarkovChain

Assume the following model.

Constant, unit isotropic innovation noise.

$A_n = B_{z_n}$

Gaussian B: (..., K, D) x (D) Categorical Z: (..., N-1) x (K) GaussianMarkovChain X: (...) x (N,D)

No plates for X.

__init__(*X*, *B*, *Z*, *B_rotator*)

Methods

__init__(X, B, Z, B_rotator)	
bound(R[, logdet, inv])	
get_bound_terms(R[, logdet, inv])	
nodes()	
rotate(R[, inv, logdet])	
setup()	This method should be called just before optimization.

bayespy.inference.vmp.transformations.RotateSwitchingMarkovChain.__init__

RotateSwitchingMarkovChain.**__init__**(*X*, *B*, *Z*, *B_rotator*)

bayespy.inference.vmp.transformations.RotateSwitchingMarkovChain.bound

RotateSwitchingMarkovChain.**bound**(*R*, *logdet=None*, *inv=None*)

bayespy.inference.vmp.transformations.RotateSwitchingMarkovChain.get_bound_terms

`RotateSwitchingMarkovChain.get_bound_terms(R, logdet=None, inv=None)`

bayespy.inference.vmp.transformations.RotateSwitchingMarkovChain.nodes

`RotateSwitchingMarkovChain.nodes()`

bayespy.inference.vmp.transformations.RotateSwitchingMarkovChain.rotate

`RotateSwitchingMarkovChain.rotate(R, inv=None, logdet=None)`

bayespy.inference.vmp.transformations.RotateSwitchingMarkovChain.setup

`RotateSwitchingMarkovChain.setup()`

This method should be called just before optimization.

bayespy.inference.vmp.transformations.RotateVaryingMarkovChain

class `bayespy.inference.vmp.transformations.RotateVaryingMarkovChain(X, B, S, B_rotator)`

Rotation for `bayespy.nodes.SwitchingGaussianMarkovChain`

Assume the following model.

Constant, unit isotropic innovation noise.

$A_n = \sum_k B_k s_{kn}$

Gaussian B: (1,D) x (D,K) Gaussian S: (N,1) x (K) MC X: () x (N+1,D)

No plates for X.

`__init__(X, B, S, B_rotator)`

Methods

`__init__`(X, B, S, B_rotator)	
`bound`(R[, logdet, inv])	
`get_bound_terms`(R[, logdet, inv])	
`nodes`()	
`rotate`(R[, inv, logdet])	
`setup`()	This method should be called just before optimization.

bayespy.inference.vmp.transformations.RotateVaryingMarkovChain.__init__

`RotateVaryingMarkovChain.__init__(X, B, S, B_rotator)`

bayespy.inference.vmp.transformations.RotateVaryingMarkovChain.bound

RotateVaryingMarkovChain.**bound**(*R*, *logdet=None*, *inv=None*)

bayespy.inference.vmp.transformations.RotateVaryingMarkovChain.get_bound_terms

RotateVaryingMarkovChain.**get_bound_terms**(*R*, *logdet=None*, *inv=None*)

bayespy.inference.vmp.transformations.RotateVaryingMarkovChain.nodes

RotateVaryingMarkovChain.**nodes**()

bayespy.inference.vmp.transformations.RotateVaryingMarkovChain.rotate

RotateVaryingMarkovChain.**rotate**(*R*, *inv=None*, *logdet=None*)

bayespy.inference.vmp.transformations.RotateVaryingMarkovChain.setup

RotateVaryingMarkovChain.**setup**()

This method should be called just before optimization.

bayespy.inference.vmp.transformations.RotateMultiple

class bayespy.inference.vmp.transformations.**RotateMultiple**(**rotators*)

Identical parameter expansion for several nodes simultaneously

Performs the same rotation for multiple nodes and combines the cost effect.

__init__(**rotators*)

Methods

__init__(*rotators)
bound(R[, logdet, inv])
get_bound_terms(R[, logdet, inv])
nodes()
rotate(R[, inv, logdet])
setup()

bayespy.inference.vmp.transformations.RotateMultiple.__init__

RotateMultiple.**__init__**(**rotators*)

bayespy.inference.vmp.transformations.RotateMultiple.bound

RotateMultiple.**bound**(*R*, *logdet=None*, *inv=None*)

bayespy.inference.vmp.transformations.RotateMultiple.get_bound_terms

`RotateMultiple.get_bound_terms(R, logdet=None, inv=None)`

bayespy.inference.vmp.transformations.RotateMultiple.nodes

`RotateMultiple.nodes()`

bayespy.inference.vmp.transformations.RotateMultiple.rotate

`RotateMultiple.rotate(R, inv=None, logdet=None)`

bayespy.inference.vmp.transformations.RotateMultiple.setup

`RotateMultiple.setup()`

5.3 bayespy.plot

Functions for plotting nodes.

5.3.1 Functions

pdf(Z, x, *args[, name, axes, fig])	Plot probability density function of a scalar variable.
contour(Z, x, y[, n, axes, fig])	Plot 2-D probability density function of a 2-D variable.
plot(Y[, axis, scale, center])	Plot a variable or an array as 1-D function with errorbars
hinton(X, **kwargs)	Plot the Hinton diagram of a node
gaussian_mixture(X[, scale, fill, axes])	Plot Gaussian mixture as ellipses in 2-D

bayespy.plot.pdf

`bayespy.plot.pdf(Z, x, *args, name=None, axes=None, fig=None, **kwargs)`

Plot probability density function of a scalar variable.

Parameters **Z** : node or function

Stochastic node or log pdf function

x : array

Grid points

bayespy.plot.contour

`bayespy.plot.contour(Z, x, y, n=None, axes=None, fig=None, **kwargs)`

Plot 2-D probability density function of a 2-D variable.

Parameters **Z** : node or function

Stochastic node or log pdf function

x : array

Grid points on x axis

y : array

Grid points on y axis

bayespy.plot.plot

`bayespy.plot.plot(Y, axis=-1, scale=2, center=False, **kwargs)`

Plot a variable or an array as 1-D function with errorbars

bayespy.plot.hinton

`bayespy.plot.hinton(X, **kwargs)`

Plot the Hinton diagram of a node

The keyword arguments depend on the node type. For some node types, the diagram also shows uncertainty with non-filled rectangles. Currently, beta-like, Gaussian-like and Dirichlet-like nodes are supported.

Parameters **X** : node

bayespy.plot.gaussian_mixture

`bayespy.plot.gaussian_mixture(X, scale=1, fill=False, axes=None, **kwargs)`

Plot Gaussian mixture as ellipses in 2-D

5.3.2 Plotters

Plotter(plotter, *args, **kwargs)	Wrapper for plotting functions and base class for node plotters
PDFPlotter(x_grid, **kwargs)	Plotter of probability density function of a scalar node
ContourPlotter(x1_grid, x2_grid, **kwargs)	Plotter of probability density function of a two-dimensional node
HintonPlotter(**kwargs)	Plotter of the Hinton diagram of a node
FunctionPlotter(**kwargs)	Plotter of a node as a 1-dimensional function
GaussianTimeseriesPlotter(**kwargs)	Plotter of a Gaussian node as a timeseries
CategoricalMarkovChainPlotter(**kwargs)	Plotter of a Categorical timeseries

bayespy.plot.Plotter

class `bayespy.plot.Plotter(plotter, *args, **kwargs)`

Wrapper for plotting functions and base class for node plotters

The purpose of this class is to collect all the parameters needed by a plotting function and provide a callable interface which needs only the node as the input.

Plotter instances are callable objects that plot a given node using a specified plotting function.

Parameters **plotter** : function

Plotting function to use

args : defined by the plotting function

Additional inputs needed by the plotting function

kwargs : defined by the plotting function

Additional keyword arguments supported by the plotting function

Examples

First, create a gamma variable:

```
>>> import numpy as np
>>> from bayespy.nodes import Gamma
>>> x = Gamma(4, 5)
```

The probability density function can be plotted as:

```
>>> import bayespy.plot as bpplt
>>> bpplt.pdf(x, np.linspace(0.1, 10, num=100))
[<matplotlib.lines.Line2D object at 0x...>]
```

However, this can be problematic when one needs to provide a plotting function for the inference engine as the inference engine gives only the node as input. Thus, we need to create a simple plotter wrapper:

```
>>> p = bpplt.Plotter(bpplt.pdf, np.linspace(0.1, 10, num=100))
```

Now, this callable object `p` needs only the node as the input:

```
>>> p(x)
[<matplotlib.lines.Line2D object at 0x...>]
```

Thus, it can be given to the inference engine to use as a plotting function:

```
>>> x = Gamma(4, 5, plotter=p)
>>> x.plot()
[<matplotlib.lines.Line2D object at 0x...>]
```

`__init__`(*plotter*, **args*, ***kwargs*)

Methods

`__init__`(plotter, *args, **kwargs)	

bayespy.plot.Plotter.__init__

`Plotter.__init__`(*plotter*, **args*, ***kwargs*)

bayespy.plot.PDFPlotter

class `bayespy.plot.PDFPlotter`(*x_grid*, ***kwargs*)

Plotter of probability density function of a scalar node

Parameters **x_grid** : array

Numerical grid on which the density function is computed and plotted

See also:

`pdf`

`__init__`(*x_grid*, ***kwargs*)

Methods

`__init__`(x_grid, **kwargs)	

bayespy.plot.PDFPlotter.__init__

`PDFPlotter.__init__`(*x_grid*, ***kwargs*)

bayespy.plot.ContourPlotter

class `bayespy.plot.ContourPlotter`(*x1_grid*, *x2_grid*, ***kwargs*)

Plotter of probability density function of a two-dimensional node

Parameters **x1_grid** : array

Grid for the first dimension

x2_grid : array

Grid for the second dimension

See also:

`contour`

`__init__`(*x1_grid*, *x2_grid*, ***kwargs*)

Methods

`__init__`(x1_grid, x2_grid, **kwargs)	

bayespy.plot.ContourPlotter.__init__

`ContourPlotter.__init__`(*x1_grid*, *x2_grid*, ***kwargs*)

bayespy.plot.HintonPlotter

class `bayespy.plot.HintonPlotter`(***kwargs*)

Plotter of the Hinton diagram of a node

See also:

`hinton`

`__init__`(***kwargs*)

Methods

`__init__`(**kwargs)

bayespy.plot.HintonPlotter.__init__

HintonPlotter.**__init__**(***kwargs*)

bayespy.plot.FunctionPlotter

class bayespy.plot.**FunctionPlotter**(***kwargs*)

Plotter of a node as a 1-dimensional function

See also:

`plot`

__init__(***kwargs*)

Methods

`__init__`(**kwargs)

bayespy.plot.FunctionPlotter.__init__

FunctionPlotter.**__init__**(***kwargs*)

bayespy.plot.GaussianTimeseriesPlotter

class bayespy.plot.**GaussianTimeseriesPlotter**(***kwargs*)

Plotter of a Gaussian node as a timeseries

__init__(***kwargs*)

Methods

`__init__`(**kwargs)

bayespy.plot.GaussianTimeseriesPlotter.__init__

GaussianTimeseriesPlotter.**__init__**(***kwargs*)

bayespy.plot.CategoricalMarkovChainPlotter

class bayespy.plot.**CategoricalMarkovChainPlotter**(***kwargs*)

Plotter of a Categorical timeseries

__init__(***kwargs*)

Methods

`__init__`(**kwargs)	

bayespy.plot.CategoricalMarkovChainPlotter.__init__

CategoricalMarkovChainPlotter.**__init__**(***kwargs*)

CHAPTER

SIX

DEVELOPER API

This chapter contains API specifications which are relevant to BayesPy developers and contributors.

6.1 Developer nodes

The following base classes are useful if writing new nodes:

node.Node(*parents, **kwargs)	Base class for all nodes.
stochastic.Stochastic(*args[, initialize, dims])	Base class for nodes that are stochastic.
expfamily.ExponentialFamily(*args, **kwargs)	A base class for nodes using natural parameterization *phi*.
deterministic.Deterministic(*args, **kwargs)	Base class for deterministic nodes.

6.1.1 bayespy.inference.vmp.nodes.node.Node

class `bayespy.inference.vmp.nodes.node.`**`Node`**(**parents*, ***kwargs*)

Base class for all nodes.

mask dims plates parents children name

Sub-classes must implement: 1. For computing the message to children:

get_moments(self):

2.For computing the message to parents: _get_message_and_mask_to_parent(self, index)

Sub-classes may need to re-implement: 1. If they manipulate plates:

_compute_mask_to_parent(index, mask) _plates_to_parent(self, index) _plates_from_parent(self, index)

`__init__`(**parents*, ***kwargs*)

Methods

__init__(*parents, **kwargs)	
add_plate_axis(to_plate)	
broadcasting_multiplier(plates, *args)	
delete()	Delete this node and the children
get_mask()	

Continued on next page

Table 6.2 – continued from previous page

get_moments()	
get_shape(ind)	
has_plotter()	Return True if the node has a plotter
move_plates(from_plate, to_plate)	
plot([fig])	Plot the node distribution using the plotter of the node
set_plotter(plotter)	

bayespy.inference.vmp.nodes.node.Node.__init__

Node.**__init__**(**parents*, ***kwargs*)

bayespy.inference.vmp.nodes.node.Node.add_plate_axis

Node.**add_plate_axis**(*to_plate*)

bayespy.inference.vmp.nodes.node.Node.broadcasting_multiplier

static Node.**broadcasting_multiplier**(*plates*, **args*)

bayespy.inference.vmp.nodes.node.Node.delete

Node.**delete**()

Delete this node and the children

bayespy.inference.vmp.nodes.node.Node.get_mask

Node.**get_mask**()

bayespy.inference.vmp.nodes.node.Node.get_moments

Node.**get_moments**()

bayespy.inference.vmp.nodes.node.Node.get_shape

Node.**get_shape**(*ind*)

bayespy.inference.vmp.nodes.node.Node.has_plotter

Node.**has_plotter**()

Return True if the node has a plotter

bayespy.inference.vmp.nodes.node.Node.move_plates

Node.**move_plates**(*from_plate*, *to_plate*)

bayespy.inference.vmp.nodes.node.Node.plot

Node.**plot**(*fig=None*, ***kwargs*)

Plot the node distribution using the plotter of the node

Because the distributions are in general very difficult to plot, the user must specify some functions which performs the plotting as wanted. See, for instance, bayespy.plot.plotting for available plotters, that is, functions that perform plotting for a node.

bayespy.inference.vmp.nodes.node.Node.set_plotter

Node.**set_plotter**(*plotter*)

Attributes

plates	
plates_multiplier	Plate multiplier is applied to messages to parents

bayespy.inference.vmp.nodes.node.Node.plates

Node.**plates** = None

bayespy.inference.vmp.nodes.node.Node.plates_multiplier

Node.**plates_multiplier**

Plate multiplier is applied to messages to parents

6.1.2 bayespy.inference.vmp.nodes.stochastic.Stochastic

class bayespy.inference.vmp.nodes.stochastic.**Stochastic**(**args*, *initialize=True*, *dims=None*, ***kwargs*)

Base class for nodes that are stochastic.

u observed

Sub-classes must implement: _compute_message_to_parent(parent, index, u_self, *u_parents) _update_distribution_and_lowerbound(self, m, *u) lowerbound(self) _compute_dims initialize_from_prior()

If you want to be able to observe the variable: _compute_fixed_moments_and_f

Sub-classes may need to re-implement: 1. If they manipulate plates:

_compute_mask_to_parent(index, mask) _compute_plates_to_parent(self, index, plates) _compute_plates_from_parent(self, index, plates)

__init__(**args*, *initialize=True*, *dims=None*, ***kwargs*)

Methods

__init__(*args[, initialize, dims])	

Continued on next page

Table 6.4 – continued from previous page

add_plate_axis(to_plate)	
broadcasting_multiplier(plates, *args)	
delete()	Delete this node and the children
get_mask()	
get_moments()	
get_shape(ind)	
has_plotter()	Return True if the node has a plotter
load(group)	Load the state of the node from a HDF5 file.
lowerbound()	
move_plates(from_plate, to_plate)	
observe(x[, mask])	Fix moments, compute f and propagate mask.
plot([fig])	Plot the node distribution using the plotter of the node
random()	Draw a random sample from the distribution.
save(group)	Save the state of the node into a HDF5 file.
set_plotter(plotter)	
unobserve()	
update([annealing])	

bayespy.inference.vmp.nodes.stochastic.Stochastic.__init__

Stochastic.**__init__**(**args*, *initialize=True*, *dims=None*, ***kwargs*)

bayespy.inference.vmp.nodes.stochastic.Stochastic.add_plate_axis

Stochastic.**add_plate_axis**(*to_plate*)

bayespy.inference.vmp.nodes.stochastic.Stochastic.broadcasting_multiplier

Stochastic.**broadcasting_multiplier**(*plates*, **args*)

bayespy.inference.vmp.nodes.stochastic.Stochastic.delete

Stochastic.**delete**()

Delete this node and the children

bayespy.inference.vmp.nodes.stochastic.Stochastic.get_mask

Stochastic.**get_mask**()

bayespy.inference.vmp.nodes.stochastic.Stochastic.get_moments

Stochastic.**get_moments**()

bayespy.inference.vmp.nodes.stochastic.Stochastic.get_shape

Stochastic.**get_shape**(*ind*)

bayespy.inference.vmp.nodes.stochastic.Stochastic.has_plotter

`Stochastic.has_plotter()`
Return True if the node has a plotter

bayespy.inference.vmp.nodes.stochastic.Stochastic.load

`Stochastic.load(group)`
Load the state of the node from a HDF5 file.

bayespy.inference.vmp.nodes.stochastic.Stochastic.lowerbound

`Stochastic.lowerbound()`

bayespy.inference.vmp.nodes.stochastic.Stochastic.move_plates

`Stochastic.move_plates(from_plate, to_plate)`

bayespy.inference.vmp.nodes.stochastic.Stochastic.observe

`Stochastic.observe(x, mask=True)`
Fix moments, compute f and propagate mask.

bayespy.inference.vmp.nodes.stochastic.Stochastic.plot

`Stochastic.plot(fig=None, **kwargs)`
Plot the node distribution using the plotter of the node

Because the distributions are in general very difficult to plot, the user must specify some functions which performs the plotting as wanted. See, for instance, bayespy.plot.plotting for available plotters, that is, functions that perform plotting for a node.

bayespy.inference.vmp.nodes.stochastic.Stochastic.random

`Stochastic.random()`
Draw a random sample from the distribution.

bayespy.inference.vmp.nodes.stochastic.Stochastic.save

`Stochastic.save(group)`
Save the state of the node into a HDF5 file.

group can be the root

bayespy.inference.vmp.nodes.stochastic.Stochastic.set_plotter

`Stochastic.set_plotter(plotter)`

bayespy.inference.vmp.nodes.stochastic.Stochastic.unobserve

`Stochastic.unobserve()`

bayespy.inference.vmp.nodes.stochastic.Stochastic.update

`Stochastic.update(annealing=1.0)`

Attributes

plates	
plates_multiplier	Plate multiplier is applied to messages to parents

bayespy.inference.vmp.nodes.stochastic.Stochastic.plates

`Stochastic.plates` = None

bayespy.inference.vmp.nodes.stochastic.Stochastic.plates_multiplier

`Stochastic.plates_multiplier`
Plate multiplier is applied to messages to parents

6.1.3 bayespy.inference.vmp.nodes.expfamily.ExponentialFamily

class `bayespy.inference.vmp.nodes.expfamily.ExponentialFamily(*args, **kwargs)`
A base class for nodes using natural parameterization *phi*.

phi

Sub-classes must implement the following static methods: _compute_message_to_parent(index, u_self, *u_parents) _compute_phi_from_parents(*u_parents, mask) _compute_moments_and_cgf(phi, mask) _compute_fixed_moments_and_f(x, mask=True)

Sub-classes may need to re-implement: 1. If they manipulate plates:

_compute_mask_to_parent(index, mask) _compute_plates_to_parent(self, index, plates) _compute_plates_from_parent(self, index, plates)

`__init__(*args, **kwargs)`

Methods

`__init__`(*args, **kwargs)	
`add_plate_axis`(to_plate)	
`broadcasting_multiplier`(plates, *args)	
`delete`()	Delete this node and the children
`get_gradient`(rg)	Computes gradient with respect to the natural parameters.
`get_mask`()	
`get_moments`()	

Continued on next page

Table 6.6 – continued from previous page

get_parameters()	Return parameters of the VB distribution.
get_riemannian_gradient()	Computes the Riemannian/natural gradient.
get_shape(ind)	
has_plotter()	Return True if the node has a plotter
initialize_from_parameters(*args)	
initialize_from_prior()	
initialize_from_random()	Set the variable to a random sample from the current distribution.
initialize_from_value(x, *args)	
load(group)	Load the state of the node from a HDF5 file.
logpdf(X[, mask])	Compute the log probability density function Q(X) of this node.
lower_bound_contribution([gradient, ...])	Compute E[log p(X|parents) - log q(X)]
lowerbound()	
move_plates(from_plate, to_plate)	
observe(x, *args[, mask])	Fix moments, compute f and propagate mask.
pdf(X[, mask])	Compute the probability density function of this node.
plot([fig])	Plot the node distribution using the plotter of the node
random()	Draw a random sample from the distribution.
save(group)	Save the state of the node into a HDF5 file.
set_parameters(x)	Set the parameters of the VB distribution.
set_plotter(plotter)	
unobserve()	
update([annealing])	

bayespy.inference.vmp.nodes.expfamily.ExponentialFamily.__init__

ExponentialFamily.**__init__**(**args*, ***kwargs*)

bayespy.inference.vmp.nodes.expfamily.ExponentialFamily.add_plate_axis

ExponentialFamily.**add_plate_axis**(*to_plate*)

bayespy.inference.vmp.nodes.expfamily.ExponentialFamily.broadcasting_multiplier

ExponentialFamily.**broadcasting_multiplier**(*plates*, **args*)

bayespy.inference.vmp.nodes.expfamily.ExponentialFamily.delete

ExponentialFamily.**delete**()

Delete this node and the children

bayespy.inference.vmp.nodes.expfamily.ExponentialFamily.get_gradient

ExponentialFamily.**get_gradient**(*rg*)

Computes gradient with respect to the natural parameters.

The function takes the Riemannian gradient as an input. This is for three reasons: 1) You probably want to use the Riemannian gradient anyway so this helps avoiding accidental use of this function. 2) The gradient is computed by using the Riemannian gradient and chain rules. 3) Probably you need both Riemannian and normal gradients anyway so you can provide it to this function to avoid re-computing it.

bayespy.inference.vmp.nodes.expfamily.ExponentialFamily.get_mask

`ExponentialFamily.get_mask()`

bayespy.inference.vmp.nodes.expfamily.ExponentialFamily.get_moments

`ExponentialFamily.get_moments()`

bayespy.inference.vmp.nodes.expfamily.ExponentialFamily.get_parameters

`ExponentialFamily.get_parameters()`

Return parameters of the VB distribution.

The parameters should be such that they can be used for optimization, that is, use log transformation for positive parameters.

bayespy.inference.vmp.nodes.expfamily.ExponentialFamily.get_riemannian_gradient

`ExponentialFamily.get_riemannian_gradient()`

Computes the Riemannian/natural gradient.

bayespy.inference.vmp.nodes.expfamily.ExponentialFamily.get_shape

`ExponentialFamily.get_shape(ind)`

bayespy.inference.vmp.nodes.expfamily.ExponentialFamily.has_plotter

`ExponentialFamily.has_plotter()`

Return True if the node has a plotter

bayespy.inference.vmp.nodes.expfamily.ExponentialFamily.initialize_from_parameters

`ExponentialFamily.initialize_from_parameters(*args)`

bayespy.inference.vmp.nodes.expfamily.ExponentialFamily.initialize_from_prior

`ExponentialFamily.initialize_from_prior()`

bayespy.inference.vmp.nodes.expfamily.ExponentialFamily.initialize_from_random

`ExponentialFamily.initialize_from_random()`

Set the variable to a random sample from the current distribution.

bayespy.inference.vmp.nodes.expfamily.ExponentialFamily.initialize_from_value

`ExponentialFamily.initialize_from_value(x, *args)`

bayespy.inference.vmp.nodes.expfamily.ExponentialFamily.load

`ExponentialFamily.load(group)`
Load the state of the node from a HDF5 file.

bayespy.inference.vmp.nodes.expfamily.ExponentialFamily.logpdf

`ExponentialFamily.logpdf(X, mask=True)`
Compute the log probability density function Q(X) of this node.

bayespy.inference.vmp.nodes.expfamily.ExponentialFamily.lower_bound_contribution

`ExponentialFamily.lower_bound_contribution(gradient=False, ignore_masked=True)`
Compute E[log p(X|parents) - log q(X)]

If deterministic annealing is used, the term E[-log q(X)] is divided by the anneling coefficient. That is, phi and cgf of q are multiplied by the temperature (inverse annealing coefficient).

bayespy.inference.vmp.nodes.expfamily.ExponentialFamily.lowerbound

`ExponentialFamily.lowerbound()`

bayespy.inference.vmp.nodes.expfamily.ExponentialFamily.move_plates

`ExponentialFamily.move_plates(from_plate, to_plate)`

bayespy.inference.vmp.nodes.expfamily.ExponentialFamily.observe

`ExponentialFamily.observe(x, *args, mask=True)`
Fix moments, compute f and propagate mask.

bayespy.inference.vmp.nodes.expfamily.ExponentialFamily.pdf

`ExponentialFamily.pdf(X, mask=True)`
Compute the probability density function of this node.

bayespy.inference.vmp.nodes.expfamily.ExponentialFamily.plot

`ExponentialFamily.plot(fig=None, **kwargs)`
Plot the node distribution using the plotter of the node

Because the distributions are in general very difficult to plot, the user must specify some functions which performs the plotting as wanted. See, for instance, bayespy.plot.plotting for available plotters, that is, functions that perform plotting for a node.

bayespy.inference.vmp.nodes.expfamily.ExponentialFamily.random

`ExponentialFamily.random()`
Draw a random sample from the distribution.

bayespy.inference.vmp.nodes.expfamily.ExponentialFamily.save

ExponentialFamily.**save**(*group*)

Save the state of the node into a HDF5 file.

group can be the root

bayespy.inference.vmp.nodes.expfamily.ExponentialFamily.set_parameters

ExponentialFamily.**set_parameters**(*x*)

Set the parameters of the VB distribution.

The parameters should be such that they can be used for optimization, that is, use log transformation for positive parameters.

bayespy.inference.vmp.nodes.expfamily.ExponentialFamily.set_plotter

ExponentialFamily.**set_plotter**(*plotter*)

bayespy.inference.vmp.nodes.expfamily.ExponentialFamily.unobserve

ExponentialFamily.**unobserve**()

bayespy.inference.vmp.nodes.expfamily.ExponentialFamily.update

ExponentialFamily.**update**(*annealing=1.0*)

Attributes

dims	
plates	
plates_multiplier	Plate multiplier is applied to messages to parents

bayespy.inference.vmp.nodes.expfamily.ExponentialFamily.dims

ExponentialFamily.**dims** = None

bayespy.inference.vmp.nodes.expfamily.ExponentialFamily.plates

ExponentialFamily.**plates** = None

bayespy.inference.vmp.nodes.expfamily.ExponentialFamily.plates_multiplier

ExponentialFamily.**plates_multiplier**

Plate multiplier is applied to messages to parents

6.1.4 bayespy.inference.vmp.nodes.deterministic.Deterministic

class `bayespy.inference.vmp.nodes.deterministic.`**`Deterministic`**(**args*, ***kwargs*)

Base class for deterministic nodes.

Sub-classes must implement: 1. For implementing the deterministic function:

_compute_moments(self, *u)

2.One of the following options: a) Simple methods:

_compute_message_to_parent(self, index, m, *u) not? _compute_mask_to_parent(self, index, mask)

(a)More control with: _compute_message_and_mask_to_parent(self, index, m, *u)

Sub-classes may need to re-implement: 1. If they manipulate plates:

_compute_mask_to_parent(index, mask) _compute_plates_to_parent(self, index, plates) _compute_plates_from_parent(self, index, plates)

`__init__`(**args*, ***kwargs*)

Methods

`__init__`(*args, **kwargs)	
`add_plate_axis`(to_plate)	
`broadcasting_multiplier`(plates, *args)	
`delete`()	Delete this node and the children
`get_mask`()	
`get_moments`()	
`get_shape`(ind)	
`has_plotter`()	Return True if the node has a plotter
`lower_bound_contribution`([gradient])	
`move_plates`(from_plate, to_plate)	
`plot`([fig])	Plot the node distribution using the plotter of the node
`set_plotter`(plotter)	

bayespy.inference.vmp.nodes.deterministic.Deterministic.__init__

`Deterministic.`**`__init__`**(**args*, ***kwargs*)

bayespy.inference.vmp.nodes.deterministic.Deterministic.add_plate_axis

`Deterministic.`**`add_plate_axis`**(*to_plate*)

bayespy.inference.vmp.nodes.deterministic.Deterministic.broadcasting_multiplier

`Deterministic.`**`broadcasting_multiplier`**(*plates*, **args*)

bayespy.inference.vmp.nodes.deterministic.Deterministic.delete

`Deterministic.delete()`
Delete this node and the children

bayespy.inference.vmp.nodes.deterministic.Deterministic.get_mask

`Deterministic.get_mask()`

bayespy.inference.vmp.nodes.deterministic.Deterministic.get_moments

`Deterministic.get_moments()`

bayespy.inference.vmp.nodes.deterministic.Deterministic.get_shape

`Deterministic.get_shape(ind)`

bayespy.inference.vmp.nodes.deterministic.Deterministic.has_plotter

`Deterministic.has_plotter()`
Return True if the node has a plotter

bayespy.inference.vmp.nodes.deterministic.Deterministic.lower_bound_contribution

`Deterministic.lower_bound_contribution(gradient=False, **kwargs)`

bayespy.inference.vmp.nodes.deterministic.Deterministic.move_plates

`Deterministic.move_plates(from_plate, to_plate)`

bayespy.inference.vmp.nodes.deterministic.Deterministic.plot

`Deterministic.plot(fig=None, **kwargs)`
Plot the node distribution using the plotter of the node

Because the distributions are in general very difficult to plot, the user must specify some functions which performs the plotting as wanted. See, for instance, bayespy.plot.plotting for available plotters, that is, functions that perform plotting for a node.

bayespy.inference.vmp.nodes.deterministic.Deterministic.set_plotter

`Deterministic.set_plotter(plotter)`

Attributes

plates	
plates_multiplier	Plate multiplier is applied to messages to parents

bayespy.inference.vmp.nodes.deterministic.Deterministic.plates

Deterministic.**plates** = None

bayespy.inference.vmp.nodes.deterministic.Deterministic.plates_multiplier

Deterministic.**plates_multiplier**
Plate multiplier is applied to messages to parents

The following nodes are examples of special nodes that remain hidden for the user although they are often implicitly used:

constant.Constant(moments, x, **kwargs)	Node for presenting constant values.
gaussian.GaussianToGaussianGammaISO(X, **kwargs)	Converter for Gaussian moments to Gaussian-gamma isotro
gaussian.GaussianGammaISOToGaussianGammaARD(X, ...)	Converter for Gaussian-gamma ISO moments to Gaussian-g
gaussian.GaussianGammaARDToGaussianWishart(...)	
gaussian.WrapToGaussianGammaISO(*parents, ...)	
gaussian.WrapToGaussianGammaARD(mu_alpha, ...)	
gaussian.WrapToGaussianWishart(X, Lambda, ...)	Wraps Gaussian and Wishart nodes into a Gaussian-Wishart

6.1.5 bayespy.inference.vmp.nodes.constant.Constant

class bayespy.inference.vmp.nodes.constant.**Constant** (*moments*, *x*, ***kwargs*)
Node for presenting constant values.

The node wraps arrays into proper node type.

__init__ (*moments*, *x*, ***kwargs*)

Methods

__init__(moments, x, **kwargs)	
add_plate_axis(to_plate)	
broadcasting_multiplier(plates, *args)	
delete()	Delete this node and the children
get_mask()	
get_moments()	
get_shape(ind)	
has_plotter()	Return True if the node has a plotter
lower_bound_contribution([gradient])	
move_plates(from_plate, to_plate)	
plot([fig])	Plot the node distribution using the plotter of the node
set_plotter(plotter)	
set_value(x)	

bayespy.inference.vmp.nodes.constant.Constant.__init__

Constant.**__init__** (*moments*, *x*, ***kwargs*)

bayespy.inference.vmp.nodes.constant.Constant.add_plate_axis

`Constant.add_plate_axis(to_plate)`

bayespy.inference.vmp.nodes.constant.Constant.broadcasting_multiplier

`Constant.broadcasting_multiplier(plates, *args)`

bayespy.inference.vmp.nodes.constant.Constant.delete

`Constant.delete()`
Delete this node and the children

bayespy.inference.vmp.nodes.constant.Constant.get_mask

`Constant.get_mask()`

bayespy.inference.vmp.nodes.constant.Constant.get_moments

`Constant.get_moments()`

bayespy.inference.vmp.nodes.constant.Constant.get_shape

`Constant.get_shape(ind)`

bayespy.inference.vmp.nodes.constant.Constant.has_plotter

`Constant.has_plotter()`
Return True if the node has a plotter

bayespy.inference.vmp.nodes.constant.Constant.lower_bound_contribution

`Constant.lower_bound_contribution(gradient=False, **kwargs)`

bayespy.inference.vmp.nodes.constant.Constant.move_plates

`Constant.move_plates(from_plate, to_plate)`

bayespy.inference.vmp.nodes.constant.Constant.plot

`Constant.plot(fig=None, **kwargs)`
Plot the node distribution using the plotter of the node

Because the distributions are in general very difficult to plot, the user must specify some functions which performs the plotting as wanted. See, for instance, bayespy.plot.plotting for available plotters, that is, functions that perform plotting for a node.

bayespy.inference.vmp.nodes.constant.Constant.set_plotter

Constant.**set_plotter**(*plotter*)

bayespy.inference.vmp.nodes.constant.Constant.set_value

Constant.**set_value**(*x*)

Attributes

plates	
plates_multiplier	Plate multiplier is applied to messages to parents

bayespy.inference.vmp.nodes.constant.Constant.plates

Constant.**plates** = None

bayespy.inference.vmp.nodes.constant.Constant.plates_multiplier

Constant.**plates_multiplier**
Plate multiplier is applied to messages to parents

6.1.6 bayespy.inference.vmp.nodes.gaussian.GaussianToGaussianGammaISO

class bayespy.inference.vmp.nodes.gaussian.**GaussianToGaussianGammaISO**(*X*, ***kwargs*)

Converter for Gaussian moments to Gaussian-gamma isotropic moments

Combines the Gaussian moments with gamma moments for a fixed value 1.

__init__(*X*, ***kwargs*)

Methods

__init__(X, **kwargs)	
add_plate_axis(to_plate)	
broadcasting_multiplier(plates, *args)	
delete()	Delete this node and the children
get_mask()	
get_moments()	
get_shape(ind)	
has_plotter()	Return True if the node has a plotter
lower_bound_contribution([gradient])	
move_plates(from_plate, to_plate)	
plot([fig])	Plot the node distribution using the plotter of the node
set_plotter(plotter)	

bayespy.inference.vmp.nodes.gaussian.GaussianToGaussianGammaISO.__init__

GaussianToGaussianGammaISO.**__init__**(*X*, ***kwargs*)

bayespy.inference.vmp.nodes.gaussian.GaussianToGaussianGammaISO.add_plate_axis

GaussianToGaussianGammaISO.**add_plate_axis**(*to_plate*)

bayespy.inference.vmp.nodes.gaussian.GaussianToGaussianGammaISO.broadcasting_multiplier

GaussianToGaussianGammaISO.**broadcasting_multiplier**(*plates*, **args*)

bayespy.inference.vmp.nodes.gaussian.GaussianToGaussianGammaISO.delete

GaussianToGaussianGammaISO.**delete**()

Delete this node and the children

bayespy.inference.vmp.nodes.gaussian.GaussianToGaussianGammaISO.get_mask

GaussianToGaussianGammaISO.**get_mask**()

bayespy.inference.vmp.nodes.gaussian.GaussianToGaussianGammaISO.get_moments

GaussianToGaussianGammaISO.**get_moments**()

bayespy.inference.vmp.nodes.gaussian.GaussianToGaussianGammaISO.get_shape

GaussianToGaussianGammaISO.**get_shape**(*ind*)

bayespy.inference.vmp.nodes.gaussian.GaussianToGaussianGammaISO.has_plotter

GaussianToGaussianGammaISO.**has_plotter**()

Return True if the node has a plotter

bayespy.inference.vmp.nodes.gaussian.GaussianToGaussianGammaISO.lower_bound_contribution

GaussianToGaussianGammaISO.**lower_bound_contribution**(*gradient=False*, ***kwargs*)

bayespy.inference.vmp.nodes.gaussian.GaussianToGaussianGammaISO.move_plates

GaussianToGaussianGammaISO.**move_plates**(*from_plate*, *to_plate*)

bayespy.inference.vmp.nodes.gaussian.GaussianToGaussianGammaISO.plot

GaussianToGaussianGammaISO.**plot** (*fig=None*, ***kwargs*)

Plot the node distribution using the plotter of the node

Because the distributions are in general very difficult to plot, the user must specify some functions which performs the plotting as wanted. See, for instance, bayespy.plot.plotting for available plotters, that is, functions that perform plotting for a node.

bayespy.inference.vmp.nodes.gaussian.GaussianToGaussianGammaISO.set_plotter

GaussianToGaussianGammaISO.**set_plotter** (*plotter*)

Attributes

plates	
plates_multiplier	Plate multiplier is applied to messages to parents

bayespy.inference.vmp.nodes.gaussian.GaussianToGaussianGammaISO.plates

GaussianToGaussianGammaISO.**plates** = None

bayespy.inference.vmp.nodes.gaussian.GaussianToGaussianGammaISO.plates_multiplier

GaussianToGaussianGammaISO.**plates_multiplier**

Plate multiplier is applied to messages to parents

6.1.7 bayespy.inference.vmp.nodes.gaussian.GaussianGammaISOToGaussianGammaARD

class bayespy.inference.vmp.nodes.gaussian.**GaussianGammaISOToGaussianGammaARD** (*X*, ***kwargs*)

Converter for Gaussian-gamma ISO moments to Gaussian-gamma ARD moments

__init__ (*X*, ***kwargs*)

Methods

__init__(X, **kwargs)	
add_plate_axis(to_plate)	
broadcasting_multiplier(plates, *args)	
delete()	Delete this node and the children
get_mask()	
get_moments()	
get_shape(ind)	
has_plotter()	Return True if the node has a plotter
lower_bound_contribution([gradient])	
move_plates(from_plate, to_plate)	
plot([fig])	Plot the node distribution using the plotter of the node

Continued on next page

Table 6.15 – continued from previous page

set_plotter(plotter)

bayespy.inference.vmp.nodes.gaussian.GaussianGammaISOToGaussianGammaARD.__init__

GaussianGammaISOToGaussianGammaARD.**__init__**(*X*, ***kwargs*)

bayespy.inference.vmp.nodes.gaussian.GaussianGammaISOToGaussianGammaARD.add_plate_axis

GaussianGammaISOToGaussianGammaARD.**add_plate_axis**(*to_plate*)

bayespy.inference.vmp.nodes.gaussian.GaussianGammaISOToGaussianGammaARD.broadcasting_multiplier

GaussianGammaISOToGaussianGammaARD.**broadcasting_multiplier**(*plates*, **args*)

bayespy.inference.vmp.nodes.gaussian.GaussianGammaISOToGaussianGammaARD.delete

GaussianGammaISOToGaussianGammaARD.**delete**()

Delete this node and the children

bayespy.inference.vmp.nodes.gaussian.GaussianGammaISOToGaussianGammaARD.get_mask

GaussianGammaISOToGaussianGammaARD.**get_mask**()

bayespy.inference.vmp.nodes.gaussian.GaussianGammaISOToGaussianGammaARD.get_moments

GaussianGammaISOToGaussianGammaARD.**get_moments**()

bayespy.inference.vmp.nodes.gaussian.GaussianGammaISOToGaussianGammaARD.get_shape

GaussianGammaISOToGaussianGammaARD.**get_shape**(*ind*)

bayespy.inference.vmp.nodes.gaussian.GaussianGammaISOToGaussianGammaARD.has_plotter

GaussianGammaISOToGaussianGammaARD.**has_plotter**()

Return True if the node has a plotter

bayespy.inference.vmp.nodes.gaussian.GaussianGammaISOToGaussianGammaARD.lower_bound_contributi

GaussianGammaISOToGaussianGammaARD.**lower_bound_contribution**(*gradient=False*, ***kwargs*)

bayespy.inference.vmp.nodes.gaussian.GaussianGammaISOToGaussianGammaARD.move_plates

GaussianGammaISOToGaussianGammaARD.**move_plates**(*from_plate*, *to_plate*)

bayespy.inference.vmp.nodes.gaussian.GaussianGammaISOToGaussianGammaARD.plot

GaussianGammaISOToGaussianGammaARD.**plot**(*fig=None*, ***kwargs*)

Plot the node distribution using the plotter of the node

Because the distributions are in general very difficult to plot, the user must specify some functions which performs the plotting as wanted. See, for instance, bayespy.plot.plotting for available plotters, that is, functions that perform plotting for a node.

bayespy.inference.vmp.nodes.gaussian.GaussianGammaISOToGaussianGammaARD.set_plotter

GaussianGammaISOToGaussianGammaARD.**set_plotter**(*plotter*)

Attributes

plates	
plates_multiplier	Plate multiplier is applied to messages to parents

bayespy.inference.vmp.nodes.gaussian.GaussianGammaISOToGaussianGammaARD.plates

GaussianGammaISOToGaussianGammaARD.**plates** = None

bayespy.inference.vmp.nodes.gaussian.GaussianGammaISOToGaussianGammaARD.plates_multiplier

GaussianGammaISOToGaussianGammaARD.**plates_multiplier**

Plate multiplier is applied to messages to parents

6.1.8 bayespy.inference.vmp.nodes.gaussian.GaussianGammaARDToGaussianWishart

class bayespy.inference.vmp.nodes.gaussian.**GaussianGammaARDToGaussianWishart**(*X_alpha*, ***kwargs*)

__init__(*X_alpha*, ***kwargs*)

Methods

__init__(X_alpha, **kwargs)	
add_plate_axis(to_plate)	
broadcasting_multiplier(plates, *args)	
delete()	Delete this node and the children
get_mask()	
get_moments()	
get_shape(ind)	
has_plotter()	Return True if the node has a plotter
lower_bound_contribution([gradient])	
move_plates(from_plate, to_plate)	
plot([fig])	Plot the node distribution using the plotter of the node
set_plotter(plotter)	

bayespy.inference.vmp.nodes.gaussian.GaussianGammaARDToGaussianWishart.__init__

GaussianGammaARDToGaussianWishart.**__init__**(*X_alpha*, ***kwargs*)

bayespy.inference.vmp.nodes.gaussian.GaussianGammaARDToGaussianWishart.add_plate_axis

GaussianGammaARDToGaussianWishart.**add_plate_axis**(*to_plate*)

bayespy.inference.vmp.nodes.gaussian.GaussianGammaARDToGaussianWishart.broadcasting_multiplier

GaussianGammaARDToGaussianWishart.**broadcasting_multiplier**(*plates*, **args*)

bayespy.inference.vmp.nodes.gaussian.GaussianGammaARDToGaussianWishart.delete

GaussianGammaARDToGaussianWishart.**delete**()

Delete this node and the children

bayespy.inference.vmp.nodes.gaussian.GaussianGammaARDToGaussianWishart.get_mask

GaussianGammaARDToGaussianWishart.**get_mask**()

bayespy.inference.vmp.nodes.gaussian.GaussianGammaARDToGaussianWishart.get_moments

GaussianGammaARDToGaussianWishart.**get_moments**()

bayespy.inference.vmp.nodes.gaussian.GaussianGammaARDToGaussianWishart.get_shape

GaussianGammaARDToGaussianWishart.**get_shape**(*ind*)

bayespy.inference.vmp.nodes.gaussian.GaussianGammaARDToGaussianWishart.has_plotter

GaussianGammaARDToGaussianWishart.**has_plotter**()

Return True if the node has a plotter

bayespy.inference.vmp.nodes.gaussian.GaussianGammaARDToGaussianWishart.lower_bound_contribution

GaussianGammaARDToGaussianWishart.**lower_bound_contribution**(*gradient=False*, ***kwargs*)

bayespy.inference.vmp.nodes.gaussian.GaussianGammaARDToGaussianWishart.move_plates

GaussianGammaARDToGaussianWishart.**move_plates**(*from_plate*, *to_plate*)

bayespy.inference.vmp.nodes.gaussian.GaussianGammaARDToGaussianWishart.plot

GaussianGammaARDToGaussianWishart.**plot**(*fig=None*, ***kwargs*)

Plot the node distribution using the plotter of the node

Because the distributions are in general very difficult to plot, the user must specify some functions which performs the plotting as wanted. See, for instance, bayespy.plot.plotting for available plotters, that is, functions that perform plotting for a node.

bayespy.inference.vmp.nodes.gaussian.GaussianGammaARDToGaussianWishart.set_plotter

GaussianGammaARDToGaussianWishart.**set_plotter**(*plotter*)

Attributes

plates	
plates_multiplier	Plate multiplier is applied to messages to parents

bayespy.inference.vmp.nodes.gaussian.GaussianGammaARDToGaussianWishart.plates

GaussianGammaARDToGaussianWishart.**plates** = None

bayespy.inference.vmp.nodes.gaussian.GaussianGammaARDToGaussianWishart.plates_multiplier

GaussianGammaARDToGaussianWishart.**plates_multiplier**

Plate multiplier is applied to messages to parents

6.1.9 bayespy.inference.vmp.nodes.gaussian.WrapToGaussianGammaISO

class bayespy.inference.vmp.nodes.gaussian.**WrapToGaussianGammaISO**(**parents*, ***kwargs*)

__init__(**parents*, ***kwargs*)

Methods

__init__(*parents, **kwargs)	
add_plate_axis(to_plate)	
broadcasting_multiplier(plates, *args)	
delete()	Delete this node and the children
get_mask()	
get_moments()	
get_shape(ind)	
has_plotter()	Return True if the node has a plotter
lower_bound_contribution([gradient])	
move_plates(from_plate, to_plate)	
plot([fig])	Plot the node distribution using the plotter of the node
set_plotter(plotter)	

bayespy.inference.vmp.nodes.gaussian.WrapToGaussianGammaISO.__init__

WrapToGaussianGammaISO.**__init__**(**parents*, ***kwargs*)

bayespy.inference.vmp.nodes.gaussian.WrapToGaussianGammaISO.add_plate_axis

WrapToGaussianGammaISO.**add_plate_axis**(*to_plate*)

bayespy.inference.vmp.nodes.gaussian.WrapToGaussianGammaISO.broadcasting_multiplier

WrapToGaussianGammaISO.**broadcasting_multiplier**(*plates*, **args*)

bayespy.inference.vmp.nodes.gaussian.WrapToGaussianGammaISO.delete

WrapToGaussianGammaISO.**delete**()
Delete this node and the children

bayespy.inference.vmp.nodes.gaussian.WrapToGaussianGammaISO.get_mask

WrapToGaussianGammaISO.**get_mask**()

bayespy.inference.vmp.nodes.gaussian.WrapToGaussianGammaISO.get_moments

WrapToGaussianGammaISO.**get_moments**()

bayespy.inference.vmp.nodes.gaussian.WrapToGaussianGammaISO.get_shape

WrapToGaussianGammaISO.**get_shape**(*ind*)

bayespy.inference.vmp.nodes.gaussian.WrapToGaussianGammaISO.has_plotter

WrapToGaussianGammaISO.**has_plotter**()
Return True if the node has a plotter

bayespy.inference.vmp.nodes.gaussian.WrapToGaussianGammaISO.lower_bound_contribution

WrapToGaussianGammaISO.**lower_bound_contribution**(*gradient=False*, ***kwargs*)

bayespy.inference.vmp.nodes.gaussian.WrapToGaussianGammaISO.move_plates

WrapToGaussianGammaISO.**move_plates**(*from_plate*, *to_plate*)

bayespy.inference.vmp.nodes.gaussian.WrapToGaussianGammaISO.plot

WrapToGaussianGammaISO.**plot**(*fig=None*, ***kwargs*)
Plot the node distribution using the plotter of the node

Because the distributions are in general very difficult to plot, the user must specify some functions which performs the plotting as wanted. See, for instance, bayespy.plot.plotting for available plotters, that is, functions that perform plotting for a node.

bayespy.inference.vmp.nodes.gaussian.WrapToGaussianGammaISO.set_plotter

WrapToGaussianGammaISO.**set_plotter**(*plotter*)

Attributes

plates	
plates_multiplier	Plate multiplier is applied to messages to parents

bayespy.inference.vmp.nodes.gaussian.WrapToGaussianGammaISO.plates

WrapToGaussianGammaISO.**plates** = None

bayespy.inference.vmp.nodes.gaussian.WrapToGaussianGammaISO.plates_multiplier

WrapToGaussianGammaISO.**plates_multiplier**
Plate multiplier is applied to messages to parents

6.1.10 bayespy.inference.vmp.nodes.gaussian.WrapToGaussianGammaARD

class bayespy.inference.vmp.nodes.gaussian.**WrapToGaussianGammaARD**(*mu_alpha*, *tau*, ***kwargs*)

__init__(*mu_alpha*, *tau*, ***kwargs*)

Methods

__init__(mu_alpha, tau, **kwargs)	
add_plate_axis(to_plate)	
broadcasting_multiplier(plates, *args)	
delete()	Delete this node and the children
get_mask()	
get_moments()	
get_shape(ind)	
has_plotter()	Return True if the node has a plotter
lower_bound_contribution([gradient])	
move_plates(from_plate, to_plate)	
plot([fig])	Plot the node distribution using the plotter of the node
set_plotter(plotter)	

bayespy.inference.vmp.nodes.gaussian.WrapToGaussianGammaARD.__init__

WrapToGaussianGammaARD.**__init__**(*mu_alpha*, *tau*, ***kwargs*)

bayespy.inference.vmp.nodes.gaussian.WrapToGaussianGammaARD.add_plate_axis

WrapToGaussianGammaARD.**add_plate_axis**(*to_plate*)

bayespy.inference.vmp.nodes.gaussian.WrapToGaussianGammaARD.broadcasting_multiplier

WrapToGaussianGammaARD.**broadcasting_multiplier**(*plates*, **args*)

bayespy.inference.vmp.nodes.gaussian.WrapToGaussianGammaARD.delete

WrapToGaussianGammaARD.**delete**()

Delete this node and the children

bayespy.inference.vmp.nodes.gaussian.WrapToGaussianGammaARD.get_mask

WrapToGaussianGammaARD.**get_mask**()

bayespy.inference.vmp.nodes.gaussian.WrapToGaussianGammaARD.get_moments

WrapToGaussianGammaARD.**get_moments**()

bayespy.inference.vmp.nodes.gaussian.WrapToGaussianGammaARD.get_shape

WrapToGaussianGammaARD.**get_shape**(*ind*)

bayespy.inference.vmp.nodes.gaussian.WrapToGaussianGammaARD.has_plotter

WrapToGaussianGammaARD.**has_plotter**()

Return True if the node has a plotter

bayespy.inference.vmp.nodes.gaussian.WrapToGaussianGammaARD.lower_bound_contribution

WrapToGaussianGammaARD.**lower_bound_contribution**(*gradient=False*, ***kwargs*)

bayespy.inference.vmp.nodes.gaussian.WrapToGaussianGammaARD.move_plates

WrapToGaussianGammaARD.**move_plates**(*from_plate*, *to_plate*)

bayespy.inference.vmp.nodes.gaussian.WrapToGaussianGammaARD.plot

WrapToGaussianGammaARD.**plot**(*fig=None*, ***kwargs*)

Plot the node distribution using the plotter of the node

Because the distributions are in general very difficult to plot, the user must specify some functions which performs the plotting as wanted. See, for instance, bayespy.plot.plotting for available plotters, that is, functions that perform plotting for a node.

bayespy.inference.vmp.nodes.gaussian.WrapToGaussianGammaARD.set_plotter

WrapToGaussianGammaARD.**set_plotter**(*plotter*)

Attributes

plates	
plates_multiplier	Plate multiplier is applied to messages to parents

bayespy.inference.vmp.nodes.gaussian.WrapToGaussianGammaARD.plates

WrapToGaussianGammaARD.**plates** = None

bayespy.inference.vmp.nodes.gaussian.WrapToGaussianGammaARD.plates_multiplier

WrapToGaussianGammaARD.**plates_multiplier**

Plate multiplier is applied to messages to parents

6.1.11 bayespy.inference.vmp.nodes.gaussian.WrapToGaussianWishart

class bayespy.inference.vmp.nodes.gaussian.**WrapToGaussianWishart**(*X*, *Lambda*, ***kwargs*)

Wraps Gaussian and Wishart nodes into a Gaussian-Wishart node.

The following node combinations can be wrapped:

- Gaussian and Wishart
- Gaussian-gamma and Wishart
- Gaussian-Wishart and gamma

__init__(*X*, *Lambda*, ***kwargs*)

Methods

__init__(X, Lambda, **kwargs)	
add_plate_axis(to_plate)	
broadcasting_multiplier(plates, *args)	
delete()	Delete this node and the children
get_mask()	

Continued on next page

Table 6.23 – continued from previous page

get_moments()	
get_shape(ind)	
has_plotter()	Return True if the node has a plotter
lower_bound_contribution([gradient])	
move_plates(from_plate, to_plate)	
plot([fig])	Plot the node distribution using the plotter of the node
set_plotter(plotter)	

bayespy.inference.vmp.nodes.gaussian.WrapToGaussianWishart.__init__

WrapToGaussianWishart.**__init__**(*X*, *Lambda*, ***kwargs*)

bayespy.inference.vmp.nodes.gaussian.WrapToGaussianWishart.add_plate_axis

WrapToGaussianWishart.**add_plate_axis**(*to_plate*)

bayespy.inference.vmp.nodes.gaussian.WrapToGaussianWishart.broadcasting_multiplier

WrapToGaussianWishart.**broadcasting_multiplier**(*plates*, **args*)

bayespy.inference.vmp.nodes.gaussian.WrapToGaussianWishart.delete

WrapToGaussianWishart.**delete**()

Delete this node and the children

bayespy.inference.vmp.nodes.gaussian.WrapToGaussianWishart.get_mask

WrapToGaussianWishart.**get_mask**()

bayespy.inference.vmp.nodes.gaussian.WrapToGaussianWishart.get_moments

WrapToGaussianWishart.**get_moments**()

bayespy.inference.vmp.nodes.gaussian.WrapToGaussianWishart.get_shape

WrapToGaussianWishart.**get_shape**(*ind*)

bayespy.inference.vmp.nodes.gaussian.WrapToGaussianWishart.has_plotter

WrapToGaussianWishart.**has_plotter**()

Return True if the node has a plotter

bayespy.inference.vmp.nodes.gaussian.WrapToGaussianWishart.lower_bound_contribution

WrapToGaussianWishart.**lower_bound_contribution**(*gradient=False*, ***kwargs*)

bayespy.inference.vmp.nodes.gaussian.WrapToGaussianWishart.move_plates

`WrapToGaussianWishart.move_plates(from_plate, to_plate)`

bayespy.inference.vmp.nodes.gaussian.WrapToGaussianWishart.plot

`WrapToGaussianWishart.plot(fig=None, **kwargs)`
Plot the node distribution using the plotter of the node

Because the distributions are in general very difficult to plot, the user must specify some functions which performs the plotting as wanted. See, for instance, bayespy.plot.plotting for available plotters, that is, functions that perform plotting for a node.

bayespy.inference.vmp.nodes.gaussian.WrapToGaussianWishart.set_plotter

`WrapToGaussianWishart.set_plotter(plotter)`

Attributes

`plates`	
`plates_multiplier`	Plate multiplier is applied to messages to parents

bayespy.inference.vmp.nodes.gaussian.WrapToGaussianWishart.plates

`WrapToGaussianWishart.plates = None`

bayespy.inference.vmp.nodes.gaussian.WrapToGaussianWishart.plates_multiplier

`WrapToGaussianWishart.plates_multiplier`
Plate multiplier is applied to messages to parents

6.2 Moments

`node.Moments`	Base class for defining the expectation of the su
`gaussian.GaussianMoments(ndim)`	Class for the moments of Gaussian variables.
`gaussian_markov_chain.GaussianMarkovChainMoments`	
`gaussian.GaussianGammaISOMoments(ndim)`	Class for the moments of Gaussian-gamma-ISO
`gaussian.GaussianGammaARDMoments(ndim)`	Class for the moments of Gaussian-gamma-AR
`gaussian.GaussianWishartMoments`	Class for the moments of Gaussian-Wishart vari
`gamma.GammaMoments`	Class for the moments of gamma variables.
`wishart.WishartMoments`	
`beta.BetaMoments`	Class for the moments of beta variables.
`dirichlet.DirichletMoments`	Class for the moments of Dirichlet variables.
`bernoulli.BernoulliMoments()`	Class for the moments of Bernoulli variables.
`binomial.BinomialMoments(N)`	Class for the moments of binomial variables
`categorical.CategoricalMoments(categories)`	Class for the moments of categorical variables.
`categorical_markov_chain.CategoricalMarkovChainMoments(...)`	Class for the moments of categorical Markov ch

Continue

Table 6.25 – continued from previous page

multinomial.MultinomialMoments	Class for the moments of multinomial variables
poisson.PoissonMoments	Class for the moments of Poisson variables

6.2.1 bayespy.inference.vmp.nodes.node.Moments

class bayespy.inference.vmp.nodes.node.**Moments**

Base class for defining the expectation of the sufficient statistics.

The benefits:

- Write statistic-specific features in one place only. For instance, covariance from Gaussian message.
- Different nodes may have identically defined statistic so you need to implement related features only once. For instance, Gaussian and GaussianARD differ on the prior but the moments are the same.
- General processing nodes which do not change the type of the moments may "inherit" the features from the parent node. For instance, slicing operator.
- Conversions can be done easily in both of the above cases if the message conversion is defined in the moments class. For instance, GaussianMarkovChain to Gaussian and VaryingGaussianMarkovChain to Gaussian.

__init__()

Initialize self. See help(type(self)) for accurate signature.

Methods

add_converter(moments_to, converter)	
compute_dims_from_values(x)	
compute_fixed_moments(x)	
get_converter(moments_to)	Finds conversion to another moments type if possible.

bayespy.inference.vmp.nodes.node.Moments.add_converter

classmethod Moments.**add_converter**(*moments_to*, *converter*)

bayespy.inference.vmp.nodes.node.Moments.compute_dims_from_values

Moments.**compute_dims_from_values**(*x*)

bayespy.inference.vmp.nodes.node.Moments.compute_fixed_moments

Moments.**compute_fixed_moments**(*x*)

bayespy.inference.vmp.nodes.node.Moments.get_converter

Moments.**get_converter**(*moments_to*)

Finds conversion to another moments type if possible.

Note that a conversion from moments A to moments B may require intermediate conversions. For instance: A->C->D->B. This method finds the path which uses the least amount of conversions and returns that path as a single conversion. If no conversion path is available, an error is raised.

The search algorithm starts from the original moments class and applies all possible converters to get a new list of moments classes. This list is extended by adding recursively all parent classes because their converters are applicable. Then, all possible converters are applied to this list to get a new list of current moments classes. This is iterated until the algorithm hits the target moments class or its subclass.

6.2.2 bayespy.inference.vmp.nodes.gaussian.GaussianMoments

class `bayespy.inference.vmp.nodes.gaussian.`**`GaussianMoments`**(*ndim*)

Class for the moments of Gaussian variables.

`__init__`(*ndim*)

Methods

`__init__`(ndim)	
`add_converter`(moments_to, converter)	
`compute_dims_from_values`(x)	Return the shape of the moments for a fixed value.
`compute_fixed_moments`(x)	Compute the moments for a fixed value
`get_converter`(moments_to)	Finds conversion to another moments type if possible.

bayespy.inference.vmp.nodes.gaussian.GaussianMoments.__init__

`GaussianMoments.`**`__init__`**(*ndim*)

bayespy.inference.vmp.nodes.gaussian.GaussianMoments.add_converter

`GaussianMoments.`**`add_converter`**(*moments_to*, *converter*)

bayespy.inference.vmp.nodes.gaussian.GaussianMoments.compute_dims_from_values

`GaussianMoments.`**`compute_dims_from_values`**(*x*)

Return the shape of the moments for a fixed value.

bayespy.inference.vmp.nodes.gaussian.GaussianMoments.compute_fixed_moments

`GaussianMoments.`**`compute_fixed_moments`**(*x*)

Compute the moments for a fixed value

bayespy.inference.vmp.nodes.gaussian.GaussianMoments.get_converter

`GaussianMoments.`**`get_converter`**(*moments_to*)

Finds conversion to another moments type if possible.

Note that a conversion from moments A to moments B may require intermediate conversions. For instance: A->C->D->B. This method finds the path which uses the least amount of conversions and returns that path as a single conversion. If no conversion path is available, an error is raised.

The search algorithm starts from the original moments class and applies all possible converters to get a new list of moments classes. This list is extended by adding recursively all parent classes because their converters are applicable. Then, all possible converters are applied to this list to get a new list of current moments classes. This is iterated until the algorithm hits the target moments class or its subclass.

6.2.3 bayespy.inference.vmp.nodes.gaussian_markov_chain.GaussianMarkovChainMoments

class `bayespy.inference.vmp.nodes.gaussian_markov_chain.`**`GaussianMarkovChainMoments`**

`__init__``()`
Initialize self. See help(type(self)) for accurate signature.

Methods

`add_converter`(moments_to, converter)	
`compute_dims_from_values`(x)	
`compute_fixed_moments`(x)	
`get_converter`(moments_to)	Finds conversion to another moments type if possible.

bayespy.inference.vmp.nodes.gaussian_markov_chain.GaussianMarkovChainMoments.add_converter

`GaussianMarkovChainMoments.`**`add_converter`**(*moments_to*, *converter*)

bayespy.inference.vmp.nodes.gaussian_markov_chain.GaussianMarkovChainMoments.compute_dims_from_va

`GaussianMarkovChainMoments.`**`compute_dims_from_values`**(*x*)

bayespy.inference.vmp.nodes.gaussian_markov_chain.GaussianMarkovChainMoments.compute_fixed_momen

`GaussianMarkovChainMoments.`**`compute_fixed_moments`**(*x*)

bayespy.inference.vmp.nodes.gaussian_markov_chain.GaussianMarkovChainMoments.get_converter

`GaussianMarkovChainMoments.`**`get_converter`**(*moments_to*)
Finds conversion to another moments type if possible.

Note that a conversion from moments A to moments B may require intermediate conversions. For instance: A->C->D->B. This method finds the path which uses the least amount of conversions and returns that path as a single conversion. If no conversion path is available, an error is raised.

The search algorithm starts from the original moments class and applies all possible converters to get a new list of moments classes. This list is extended by adding recursively all parent classes because their converters are applicable. Then, all possible converters are applied to this list to get a new list of current moments classes. This is iterated until the algorithm hits the target moments class or its subclass.

6.2.4 bayespy.inference.vmp.nodes.gaussian.GaussianGammaISOMoments

class `bayespy.inference.vmp.nodes.gaussian.`**`GaussianGammaISOMoments`**(*ndim*)
Class for the moments of Gaussian-gamma-ISO variables.

`__init__(ndim)`
Create moments object for Gaussian-gamma isotropic variables

ndim=0: scalar ndim=1: vector ndim=2: matrix ...

Methods

`__init__`(ndim)	Create moments object for Gaussian-gamma isotropic variables
`add_converter`(moments_to, converter)	
`compute_dims_from_values`(x, alpha)	Return the shape of the moments for a fixed value.
`compute_fixed_moments`(x, alpha)	Compute the moments for a fixed value
`get_converter`(moments_to)	Finds conversion to another moments type if possible.

bayespy.inference.vmp.nodes.gaussian.GaussianGammaISOMoments.__init__

`GaussianGammaISOMoments.__init__(ndim)`
Create moments object for Gaussian-gamma isotropic variables

ndim=0: scalar ndim=1: vector ndim=2: matrix ...

bayespy.inference.vmp.nodes.gaussian.GaussianGammaISOMoments.add_converter

`GaussianGammaISOMoments.add_converter(moments_to, converter)`

bayespy.inference.vmp.nodes.gaussian.GaussianGammaISOMoments.compute_dims_from_values

`GaussianGammaISOMoments.compute_dims_from_values(x, alpha)`
Return the shape of the moments for a fixed value.

bayespy.inference.vmp.nodes.gaussian.GaussianGammaISOMoments.compute_fixed_moments

`GaussianGammaISOMoments.compute_fixed_moments(x, alpha)`
Compute the moments for a fixed value

x is a mean vector. *alpha* is a precision scale

bayespy.inference.vmp.nodes.gaussian.GaussianGammaISOMoments.get_converter

`GaussianGammaISOMoments.get_converter(moments_to)`
Finds conversion to another moments type if possible.

Note that a conversion from moments A to moments B may require intermediate conversions. For instance: A->C->D->B. This method finds the path which uses the least amount of conversions and returns that path as a single conversion. If no conversion path is available, an error is raised.

The search algorithm starts from the original moments class and applies all possible converters to get a new list of moments classes. This list is extended by adding recursively all parent classes because their converters are applicable. Then, all possible converters are applied to this list to get a new list of current moments classes. This is iterated until the algorithm hits the target moments class or its subclass.

6.2.5 bayespy.inference.vmp.nodes.gaussian.GaussianGammaARDMoments

class `bayespy.inference.vmp.nodes.gaussian.GaussianGammaARDMoments(ndim)`
Class for the moments of Gaussian-gamma-ARD variables.

`__init__(ndim)`
Create moments object for Gaussian-gamma isotropic variables

ndim=0: scalar ndim=1: vector ndim=2: matrix ...

Methods

`__init__(ndim)`	Create moments object for Gaussian-gamma isotropic variables
`add_converter(moments_to, converter)`	
`compute_dims_from_values(x, alpha)`	Return the shape of the moments for a fixed value.
`compute_fixed_moments(x, alpha)`	Compute the moments for a fixed value
`get_converter(moments_to)`	Finds conversion to another moments type if possible.

bayespy.inference.vmp.nodes.gaussian.GaussianGammaARDMoments.__init__

`GaussianGammaARDMoments.__init__(ndim)`
Create moments object for Gaussian-gamma isotropic variables

ndim=0: scalar ndim=1: vector ndim=2: matrix ...

bayespy.inference.vmp.nodes.gaussian.GaussianGammaARDMoments.add_converter

`GaussianGammaARDMoments.add_converter(moments_to, converter)`

bayespy.inference.vmp.nodes.gaussian.GaussianGammaARDMoments.compute_dims_from_values

`GaussianGammaARDMoments.compute_dims_from_values(x, alpha)`
Return the shape of the moments for a fixed value.

bayespy.inference.vmp.nodes.gaussian.GaussianGammaARDMoments.compute_fixed_moments

`GaussianGammaARDMoments.compute_fixed_moments(x, alpha)`
Compute the moments for a fixed value

x is a mean vector. *alpha* is a precision scale

bayespy.inference.vmp.nodes.gaussian.GaussianGammaARDMoments.get_converter

`GaussianGammaARDMoments.get_converter(moments_to)`
Finds conversion to another moments type if possible.

Note that a conversion from moments A to moments B may require intermediate conversions. For instance: A->C->D->B. This method finds the path which uses the least amount of conversions and returns that path as a single conversion. If no conversion path is available, an error is raised.

The search algorithm starts from the original moments class and applies all possible converters to get a new list of moments classes. This list is extended by adding recursively all parent classes because their converters are applicable. Then, all possible converters are applied to this list to get a new list of current moments classes. This is iterated until the algorithm hits the target moments class or its subclass.

6.2.6 bayespy.inference.vmp.nodes.gaussian.GaussianWishartMoments

class `bayespy.inference.vmp.nodes.gaussian.`**`GaussianWishartMoments`**

Class for the moments of Gaussian-Wishart variables.

`__init__``()`

Initialize self. See help(type(self)) for accurate signature.

Methods

`add_converter`(moments_to, converter)	
`compute_dims_from_values`(x, Lambda)	Return the shape of the moments for a fixed value.
`compute_fixed_moments`(x, Lambda)	Compute the moments for a fixed value
`get_converter`(moments_to)	Finds conversion to another moments type if possible.

bayespy.inference.vmp.nodes.gaussian.GaussianWishartMoments.add_converter

`GaussianWishartMoments.`**`add_converter`**`(`*moments_to*, *converter*`)`

bayespy.inference.vmp.nodes.gaussian.GaussianWishartMoments.compute_dims_from_values

`GaussianWishartMoments.`**`compute_dims_from_values`**`(`*x*, *Lambda*`)`

Return the shape of the moments for a fixed value.

bayespy.inference.vmp.nodes.gaussian.GaussianWishartMoments.compute_fixed_moments

`GaussianWishartMoments.`**`compute_fixed_moments`**`(`*x*, *Lambda*`)`

Compute the moments for a fixed value

x is a vector. *Lambda* is a precision matrix

bayespy.inference.vmp.nodes.gaussian.GaussianWishartMoments.get_converter

`GaussianWishartMoments.`**`get_converter`**`(`*moments_to*`)`

Finds conversion to another moments type if possible.

Note that a conversion from moments A to moments B may require intermediate conversions. For instance: A->C->D->B. This method finds the path which uses the least amount of conversions and returns that path as a single conversion. If no conversion path is available, an error is raised.

The search algorithm starts from the original moments class and applies all possible converters to get a new list of moments classes. This list is extended by adding recursively all parent classes because their converters are applicable. Then, all possible converters are applied to this list to get a new list of current moments classes. This is iterated until the algorithm hits the target moments class or its subclass.

6.2.7 bayespy.inference.vmp.nodes.gamma.GammaMoments

class `bayespy.inference.vmp.nodes.gamma.`**`GammaMoments`**

Class for the moments of gamma variables.

`__init__``()`

Initialize self. See help(type(self)) for accurate signature.

Methods

`add_converter`(moments_to, converter)	
`compute_dims_from_values`(x)	Return the shape of the moments for a fixed value.
`compute_fixed_moments`(x)	Compute the moments for a fixed value
`get_converter`(moments_to)	Finds conversion to another moments type if possible.

bayespy.inference.vmp.nodes.gamma.GammaMoments.add_converter

`GammaMoments.`**`add_converter`**`(`*moments_to*, *converter*`)`

bayespy.inference.vmp.nodes.gamma.GammaMoments.compute_dims_from_values

`GammaMoments.`**`compute_dims_from_values`**`(`*x*`)`

Return the shape of the moments for a fixed value.

bayespy.inference.vmp.nodes.gamma.GammaMoments.compute_fixed_moments

`GammaMoments.`**`compute_fixed_moments`**`(`*x*`)`

Compute the moments for a fixed value

bayespy.inference.vmp.nodes.gamma.GammaMoments.get_converter

`GammaMoments.`**`get_converter`**`(`*moments_to*`)`

Finds conversion to another moments type if possible.

Note that a conversion from moments A to moments B may require intermediate conversions. For instance: A->C->D->B. This method finds the path which uses the least amount of conversions and returns that path as a single conversion. If no conversion path is available, an error is raised.

The search algorithm starts from the original moments class and applies all possible converters to get a new list of moments classes. This list is extended by adding recursively all parent classes because their converters are applicable. Then, all possible converters are applied to this list to get a new list of current moments classes. This is iterated until the algorithm hits the target moments class or its subclass.

6.2.8 bayespy.inference.vmp.nodes.wishart.WishartMoments

class `bayespy.inference.vmp.nodes.wishart.`**`WishartMoments`**

`__init__``()`

Initialize self. See help(type(self)) for accurate signature.

Methods

add_converter(moments_to, converter)	
compute_dims_from_values(x)	Compute the dimensions of phi and u.
compute_fixed_moments(Lambda)	Compute moments for fixed x.
get_converter(moments_to)	Finds conversion to another moments type if possible.

bayespy.inference.vmp.nodes.wishart.WishartMoments.add_converter

WishartMoments.**add_converter**(*moments_to*, *converter*)

bayespy.inference.vmp.nodes.wishart.WishartMoments.compute_dims_from_values

WishartMoments.**compute_dims_from_values**(*x*)

Compute the dimensions of phi and u.

bayespy.inference.vmp.nodes.wishart.WishartMoments.compute_fixed_moments

WishartMoments.**compute_fixed_moments**(*Lambda*)

Compute moments for fixed x.

bayespy.inference.vmp.nodes.wishart.WishartMoments.get_converter

WishartMoments.**get_converter**(*moments_to*)

Finds conversion to another moments type if possible.

Note that a conversion from moments A to moments B may require intermediate conversions. For instance: A->C->D->B. This method finds the path which uses the least amount of conversions and returns that path as a single conversion. If no conversion path is available, an error is raised.

The search algorithm starts from the original moments class and applies all possible converters to get a new list of moments classes. This list is extended by adding recursively all parent classes because their converters are applicable. Then, all possible converters are applied to this list to get a new list of current moments classes. This is iterated until the algorithm hits the target moments class or its subclass.

6.2.9 bayespy.inference.vmp.nodes.beta.BetaMoments

class bayespy.inference.vmp.nodes.beta.**BetaMoments**

Class for the moments of beta variables.

__init__()

Initialize self. See help(type(self)) for accurate signature.

Methods

add_converter(moments_to, converter)	
compute_dims_from_values(p)	Return the shape of the moments for a fixed value.
compute_fixed_moments(p)	Compute the moments for a fixed value
get_converter(moments_to)	Finds conversion to another moments type if possible.

bayespy.inference.vmp.nodes.beta.BetaMoments.add_converter

BetaMoments.**add_converter**(*moments_to*, *converter*)

bayespy.inference.vmp.nodes.beta.BetaMoments.compute_dims_from_values

BetaMoments.**compute_dims_from_values**(*p*)
Return the shape of the moments for a fixed value.

bayespy.inference.vmp.nodes.beta.BetaMoments.compute_fixed_moments

BetaMoments.**compute_fixed_moments**(*p*)
Compute the moments for a fixed value

bayespy.inference.vmp.nodes.beta.BetaMoments.get_converter

BetaMoments.**get_converter**(*moments_to*)
Finds conversion to another moments type if possible.

Note that a conversion from moments A to moments B may require intermediate conversions. For instance: A->C->D->B. This method finds the path which uses the least amount of conversions and returns that path as a single conversion. If no conversion path is available, an error is raised.

The search algorithm starts from the original moments class and applies all possible converters to get a new list of moments classes. This list is extended by adding recursively all parent classes because their converters are applicable. Then, all possible converters are applied to this list to get a new list of current moments classes. This is iterated until the algorithm hits the target moments class or its subclass.

6.2.10 bayespy.inference.vmp.nodes.dirichlet.DirichletMoments

class bayespy.inference.vmp.nodes.dirichlet.**DirichletMoments**
Class for the moments of Dirichlet variables.

__init__()
Initialize self. See help(type(self)) for accurate signature.

Methods

add_converter(moments_to, converter)	
compute_dims_from_values(x)	Return the shape of the moments for a fixed value.
compute_fixed_moments(p)	Compute the moments for a fixed value
get_converter(moments_to)	Finds conversion to another moments type if possible.

bayespy.inference.vmp.nodes.dirichlet.DirichletMoments.add_converter

DirichletMoments.**add_converter**(*moments_to*, *converter*)

bayespy.inference.vmp.nodes.dirichlet.DirichletMoments.compute_dims_from_values

DirichletMoments.**compute_dims_from_values**(*x*)
Return the shape of the moments for a fixed value.

bayespy.inference.vmp.nodes.dirichlet.DirichletMoments.compute_fixed_moments

DirichletMoments.**compute_fixed_moments**(*p*)
Compute the moments for a fixed value

bayespy.inference.vmp.nodes.dirichlet.DirichletMoments.get_converter

DirichletMoments.**get_converter**(*moments_to*)
Finds conversion to another moments type if possible.

Note that a conversion from moments A to moments B may require intermediate conversions. For instance: A->C->D->B. This method finds the path which uses the least amount of conversions and returns that path as a single conversion. If no conversion path is available, an error is raised.

The search algorithm starts from the original moments class and applies all possible converters to get a new list of moments classes. This list is extended by adding recursively all parent classes because their converters are applicable. Then, all possible converters are applied to this list to get a new list of current moments classes. This is iterated until the algorithm hits the target moments class or its subclass.

6.2.11 bayespy.inference.vmp.nodes.bernoulli.BernoulliMoments

class bayespy.inference.vmp.nodes.bernoulli.**BernoulliMoments**
Class for the moments of Bernoulli variables.

__init__()

Methods

__init__()	
add_converter(moments_to, converter)	
compute_dims_from_values(x)	Return the shape of the moments for a fixed value.
compute_fixed_moments(x)	Compute the moments for a fixed value
get_converter(moments_to)	Finds conversion to another moments type if possible.

bayespy.inference.vmp.nodes.bernoulli.BernoulliMoments.__init__

BernoulliMoments.**__init__**()

bayespy.inference.vmp.nodes.bernoulli.BernoulliMoments.add_converter

BernoulliMoments.**add_converter**(*moments_to*, *converter*)

bayespy.inference.vmp.nodes.bernoulli.BernoulliMoments.compute_dims_from_values

BernoulliMoments.**compute_dims_from_values**(*x*)

Return the shape of the moments for a fixed value.

The realizations are scalars, thus the shape of the moment is ().

bayespy.inference.vmp.nodes.bernoulli.BernoulliMoments.compute_fixed_moments

BernoulliMoments.**compute_fixed_moments**(*x*)

Compute the moments for a fixed value

bayespy.inference.vmp.nodes.bernoulli.BernoulliMoments.get_converter

BernoulliMoments.**get_converter**(*moments_to*)

Finds conversion to another moments type if possible.

Note that a conversion from moments A to moments B may require intermediate conversions. For instance: A->C->D->B. This method finds the path which uses the least amount of conversions and returns that path as a single conversion. If no conversion path is available, an error is raised.

The search algorithm starts from the original moments class and applies all possible converters to get a new list of moments classes. This list is extended by adding recursively all parent classes because their converters are applicable. Then, all possible converters are applied to this list to get a new list of current moments classes. This is iterated until the algorithm hits the target moments class or its subclass.

6.2.12 bayespy.inference.vmp.nodes.binomial.BinomialMoments

class bayespy.inference.vmp.nodes.binomial.**BinomialMoments**(*N*)

Class for the moments of binomial variables

__init__(*N*)

Methods

__init__(N)	
add_converter(moments_to, converter)	
compute_dims_from_values(x)	Return the shape of the moments for a fixed value.
compute_fixed_moments(x)	Compute the moments for a fixed value
get_converter(moments_to)	Finds conversion to another moments type if possible.

bayespy.inference.vmp.nodes.binomial.BinomialMoments.__init__

BinomialMoments.**__init__**(*N*)

bayespy.inference.vmp.nodes.binomial.BinomialMoments.add_converter

BinomialMoments.**add_converter**(*moments_to*, *converter*)

bayespy.inference.vmp.nodes.binomial.BinomialMoments.compute_dims_from_values

BinomialMoments.**compute_dims_from_values**(*x*)
Return the shape of the moments for a fixed value.

The realizations are scalars, thus the shape of the moment is ().

bayespy.inference.vmp.nodes.binomial.BinomialMoments.compute_fixed_moments

BinomialMoments.**compute_fixed_moments**(*x*)
Compute the moments for a fixed value

bayespy.inference.vmp.nodes.binomial.BinomialMoments.get_converter

BinomialMoments.**get_converter**(*moments_to*)
Finds conversion to another moments type if possible.

Note that a conversion from moments A to moments B may require intermediate conversions. For instance: A->C->D->B. This method finds the path which uses the least amount of conversions and returns that path as a single conversion. If no conversion path is available, an error is raised.

The search algorithm starts from the original moments class and applies all possible converters to get a new list of moments classes. This list is extended by adding recursively all parent classes because their converters are applicable. Then, all possible converters are applied to this list to get a new list of current moments classes. This is iterated until the algorithm hits the target moments class or its subclass.

6.2.13 bayespy.inference.vmp.nodes.categorical.CategoricalMoments

class bayespy.inference.vmp.nodes.categorical.**CategoricalMoments**(*categories*)
Class for the moments of categorical variables.

__init__(*categories*)
Create moments object for categorical variables

Methods

__init__(categories)	Create moments object for categorical variables
add_converter(moments_to, converter)	
compute_dims_from_values(x)	Return the shape of the moments for a fixed value.
compute_fixed_moments(x)	Compute the moments for a fixed value
get_converter(moments_to)	Finds conversion to another moments type if possible.

bayespy.inference.vmp.nodes.categorical.CategoricalMoments.__init__

CategoricalMoments.**__init__**(*categories*)
Create moments object for categorical variables

bayespy.inference.vmp.nodes.categorical.CategoricalMoments.add_converter

CategoricalMoments.**add_converter**(*moments_to*, *converter*)

bayespy.inference.vmp.nodes.categorical.CategoricalMoments.compute_dims_from_values

CategoricalMoments.**compute_dims_from_values**(*x*)
Return the shape of the moments for a fixed value.

The observations are scalar.

bayespy.inference.vmp.nodes.categorical.CategoricalMoments.compute_fixed_moments

CategoricalMoments.**compute_fixed_moments**(*x*)
Compute the moments for a fixed value

bayespy.inference.vmp.nodes.categorical.CategoricalMoments.get_converter

CategoricalMoments.**get_converter**(*moments_to*)
Finds conversion to another moments type if possible.

Note that a conversion from moments A to moments B may require intermediate conversions. For instance: A->C->D->B. This method finds the path which uses the least amount of conversions and returns that path as a single conversion. If no conversion path is available, an error is raised.

The search algorithm starts from the original moments class and applies all possible converters to get a new list of moments classes. This list is extended by adding recursively all parent classes because their converters are applicable. Then, all possible converters are applied to this list to get a new list of current moments classes. This is iterated until the algorithm hits the target moments class or its subclass.

6.2.14 bayespy.inference.vmp.nodes.categorical_markov_chain.CategoricalMarkovChainMomen

class bayespy.inference.vmp.nodes.categorical_markov_chain.**CategoricalMarkovChainMoments**(*catego*
Class for the moments of categorical Markov chain variables.

__init__(*categories*)
Create moments object for categorical Markov chain variables.

Methods

__init__(categories)	Create moments object for categorical Markov chain variables.
add_converter(moments_to, converter)	
compute_dims_from_values(x)	Return the shape of the moments for a fixed value.
compute_fixed_moments(x)	Compute the moments for a fixed value
get_converter(moments_to)	Finds conversion to another moments type if possible.

bayespy.inference.vmp.nodes.categorical_markov_chain.CategoricalMarkovChainMoments.__init__

CategoricalMarkovChainMoments.**__init__**(*categories*)
Create moments object for categorical Markov chain variables.

bayespy.inference.vmp.nodes.categorical_markov_chain.CategoricalMarkovChainMoments.add_converter

CategoricalMarkovChainMoments.**add_converter**(*moments_to*, *converter*)

bayespy.inference.vmp.nodes.categorical_markov_chain.CategoricalMarkovChainMoments.compute_dims_fro

CategoricalMarkovChainMoments.**compute_dims_from_values**(*x*)
Return the shape of the moments for a fixed value.

bayespy.inference.vmp.nodes.categorical_markov_chain.CategoricalMarkovChainMoments.compute_fixed_mo

CategoricalMarkovChainMoments.**compute_fixed_moments**(*x*)
Compute the moments for a fixed value

bayespy.inference.vmp.nodes.categorical_markov_chain.CategoricalMarkovChainMoments.get_converter

CategoricalMarkovChainMoments.**get_converter**(*moments_to*)
Finds conversion to another moments type if possible.

Note that a conversion from moments A to moments B may require intermediate conversions. For instance: A->C->D->B. This method finds the path which uses the least amount of conversions and returns that path as a single conversion. If no conversion path is available, an error is raised.

The search algorithm starts from the original moments class and applies all possible converters to get a new list of moments classes. This list is extended by adding recursively all parent classes because their converters are applicable. Then, all possible converters are applied to this list to get a new list of current moments classes. This is iterated until the algorithm hits the target moments class or its subclass.

6.2.15 bayespy.inference.vmp.nodes.multinomial.MultinomialMoments

class bayespy.inference.vmp.nodes.multinomial.**MultinomialMoments**
Class for the moments of multinomial variables.

__init__()
Initialize self. See help(type(self)) for accurate signature.

Methods

add_converter(moments_to, converter)	
compute_dims_from_values(x)	Return the shape of the moments for a fixed value.
compute_fixed_moments(x)	Compute the moments for a fixed value
get_converter(moments_to)	Finds conversion to another moments type if possible.

bayespy.inference.vmp.nodes.multinomial.MultinomialMoments.add_converter

MultinomialMoments.**add_converter**(*moments_to*, *converter*)

bayespy.inference.vmp.nodes.multinomial.MultinomialMoments.compute_dims_from_values

MultinomialMoments.**compute_dims_from_values**(*x*)
Return the shape of the moments for a fixed value.

bayespy.inference.vmp.nodes.multinomial.MultinomialMoments.compute_fixed_moments

MultinomialMoments.**compute_fixed_moments**(*x*)

Compute the moments for a fixed value

x must be a vector of counts.

bayespy.inference.vmp.nodes.multinomial.MultinomialMoments.get_converter

MultinomialMoments.**get_converter**(*moments_to*)

Finds conversion to another moments type if possible.

Note that a conversion from moments A to moments B may require intermediate conversions. For instance: A->C->D->B. This method finds the path which uses the least amount of conversions and returns that path as a single conversion. If no conversion path is available, an error is raised.

The search algorithm starts from the original moments class and applies all possible converters to get a new list of moments classes. This list is extended by adding recursively all parent classes because their converters are applicable. Then, all possible converters are applied to this list to get a new list of current moments classes. This is iterated until the algorithm hits the target moments class or its subclass.

6.2.16 bayespy.inference.vmp.nodes.poisson.PoissonMoments

class bayespy.inference.vmp.nodes.poisson.**PoissonMoments**

Class for the moments of Poisson variables

__init__()

Initialize self. See help(type(self)) for accurate signature.

Methods

add_converter(moments_to, converter)	
compute_dims_from_values(x)	Return the shape of the moments for a fixed value.
compute_fixed_moments(x)	Compute the moments for a fixed value
get_converter(moments_to)	Finds conversion to another moments type if possible.

bayespy.inference.vmp.nodes.poisson.PoissonMoments.add_converter

PoissonMoments.**add_converter**(*moments_to*, *converter*)

bayespy.inference.vmp.nodes.poisson.PoissonMoments.compute_dims_from_values

PoissonMoments.**compute_dims_from_values**(*x*)

Return the shape of the moments for a fixed value.

The realizations are scalars, thus the shape of the moment is ().

bayespy.inference.vmp.nodes.poisson.PoissonMoments.compute_fixed_moments

PoissonMoments.**compute_fixed_moments**(*x*)

Compute the moments for a fixed value

bayespy.inference.vmp.nodes.poisson.PoissonMoments.get_converter

PoissonMoments.**get_converter**(*moments_to*)
Finds conversion to another moments type if possible.

Note that a conversion from moments A to moments B may require intermediate conversions. For instance: A->C->D->B. This method finds the path which uses the least amount of conversions and returns that path as a single conversion. If no conversion path is available, an error is raised.

The search algorithm starts from the original moments class and applies all possible converters to get a new list of moments classes. This list is extended by adding recursively all parent classes because their converters are applicable. Then, all possible converters are applied to this list to get a new list of current moments classes. This is iterated until the algorithm hits the target moments class or its subclass.

6.3 Distributions

stochastic.Distribution	A base class for the VMP forr
expfamily.ExponentialFamilyDistribution	Sub-classes implement distrib
gaussian.GaussianDistribution	Class for the VMP formulas c
gaussian.GaussianARDDistribution(shape, ndim_mu)	...
gaussian.GaussianGammaISODistribution	Class for the VMP formulas c
gaussian.GaussianGammaARDDistribution()	
gaussian.GaussianWishartDistribution	Class for the VMP formulas c
gaussian_markov_chain.GaussianMarkovChainDistribution(N, D)	Sub-classes implement distrib
gaussian_markov_chain.SwitchingGaussianMarkovChainDistribution(N, D, K)	Sub-classes implement distrib
gaussian_markov_chain.VaryingGaussianMarkovChainDistribution(N, D)	Sub-classes implement distrib
gamma.GammaDistribution	Class for the VMP formulas c
wishart.WishartDistribution	Sub-classes implement distrib
beta.BetaDistribution	Class for the VMP formulas c
dirichlet.DirichletDistribution	Class for the VMP formulas c
bernoulli.BernoulliDistribution()	Class for the VMP formulas c
binomial.BinomialDistribution(N)	Class for the VMP formulas c
categorical.CategoricalDistribution(categories)	Class for the VMP formulas c
categorical_markov_chain.CategoricalMarkovChainDistribution(...)	Class for the VMP formulas c
multinomial.MultinomialDistribution(trials)	Class for the VMP formulas c
poisson.PoissonDistribution	Class for the VMP formulas c

6.3.1 bayespy.inference.vmp.nodes.stochastic.Distribution

class bayespy.inference.vmp.nodes.stochastic.**Distribution**
A base class for the VMP formulas of variables.

Sub-classes implement distribution specific computations.

If a sub-class maps the plates differently, it needs to overload the following methods:

- compute_mask_to_parent
- plates_to_parent
- plates_from_parent

__init__()
Initialize self. See help(type(self)) for accurate signature.

Methods

compute_mask_to_parent(index, mask)	Maps the mask to the plates of a parent.
compute_message_to_parent(parent, index, ...)	Compute the message to a parent node.
plates_from_parent(index, plates)	Resolve the plate mapping from a parent.
plates_to_parent(index, plates)	Resolves the plate mapping to a parent.
random(*params[, plates])	Draw a random sample from the distribution.

bayespy.inference.vmp.nodes.stochastic.Distribution.compute_mask_to_parent

Distribution.**compute_mask_to_parent**(*index*, *mask*)
Maps the mask to the plates of a parent.

bayespy.inference.vmp.nodes.stochastic.Distribution.compute_message_to_parent

Distribution.**compute_message_to_parent**(*parent*, *index*, *u_self*, **u_parents*)
Compute the message to a parent node.

bayespy.inference.vmp.nodes.stochastic.Distribution.plates_from_parent

Distribution.**plates_from_parent**(*index*, *plates*)
Resolve the plate mapping from a parent.

Given the plates of a parent's moments, this method returns the plates that the moments has for this distribution.

bayespy.inference.vmp.nodes.stochastic.Distribution.plates_to_parent

Distribution.**plates_to_parent**(*index*, *plates*)
Resolves the plate mapping to a parent.

Given the plates of the node's moments, this method returns the plates that the message to a parent has for the parent's distribution.

bayespy.inference.vmp.nodes.stochastic.Distribution.random

Distribution.**random**(**params*, *plates=None*)
Draw a random sample from the distribution.

6.3.2 bayespy.inference.vmp.nodes.expfamily.ExponentialFamilyDistribution

class bayespy.inference.vmp.nodes.expfamily.**ExponentialFamilyDistribution**
Sub-classes implement distribution specific computations.

__init__()
Initialize self. See help(type(self)) for accurate signature.

Methods

compute_cgf_from_parents(*u_parents)	
compute_fixed_moments_and_f(x[, mask])	
compute_gradient(g, u, phi)	Compute the standard gradient with respect to the natural parameters.
compute_logpdf(u, phi, g, f, ndims)	Compute E[log p(X)] given E[u], E[phi], E[g] and E[f].
compute_mask_to_parent(index, mask)	Maps the mask to the plates of a parent.
compute_message_to_parent(parent, index, ...)	
compute_moments_and_cgf(phi[, mask])	
compute_phi_from_parents(*u_parents[, mask])	
plates_from_parent(index, plates)	Resolve the plate mapping from a parent.
plates_to_parent(index, plates)	Resolves the plate mapping to a parent.
random(*params[, plates])	Draw a random sample from the distribution.

bayespy.inference.vmp.nodes.expfamily.ExponentialFamilyDistribution.compute_cgf_from_parents

ExponentialFamilyDistribution.**compute_cgf_from_parents**(**u_parents*)

bayespy.inference.vmp.nodes.expfamily.ExponentialFamilyDistribution.compute_fixed_moments_and_f

ExponentialFamilyDistribution.**compute_fixed_moments_and_f**(*x*, *mask=True*)

bayespy.inference.vmp.nodes.expfamily.ExponentialFamilyDistribution.compute_gradient

ExponentialFamilyDistribution.**compute_gradient**(*g*, *u*, *phi*)
Compute the standard gradient with respect to the natural parameters.

bayespy.inference.vmp.nodes.expfamily.ExponentialFamilyDistribution.compute_logpdf

ExponentialFamilyDistribution.**compute_logpdf**(*u*, *phi*, *g*, *f*, *ndims*)
Compute E[log p(X)] given E[u], E[phi], E[g] and E[f]. Does not sum over plates.

bayespy.inference.vmp.nodes.expfamily.ExponentialFamilyDistribution.compute_mask_to_parent

ExponentialFamilyDistribution.**compute_mask_to_parent**(*index*, *mask*)
Maps the mask to the plates of a parent.

bayespy.inference.vmp.nodes.expfamily.ExponentialFamilyDistribution.compute_message_to_parent

ExponentialFamilyDistribution.**compute_message_to_parent**(*parent*, *index*, *u_self*, **u_parents*)

bayespy.inference.vmp.nodes.expfamily.ExponentialFamilyDistribution.compute_moments_and_cgf

ExponentialFamilyDistribution.**compute_moments_and_cgf**(*phi*, *mask=True*)

bayespy.inference.vmp.nodes.expfamily.ExponentialFamilyDistribution.compute_phi_from_parents

ExponentialFamilyDistribution.**compute_phi_from_parents**(**u_parents*, *mask=True*)

bayespy.inference.vmp.nodes.expfamily.ExponentialFamilyDistribution.plates_from_parent

`ExponentialFamilyDistribution.plates_from_parent(index, plates)`

Resolve the plate mapping from a parent.

Given the plates of a parent's moments, this method returns the plates that the moments has for this distribution.

bayespy.inference.vmp.nodes.expfamily.ExponentialFamilyDistribution.plates_to_parent

`ExponentialFamilyDistribution.plates_to_parent(index, plates)`

Resolves the plate mapping to a parent.

Given the plates of the node's moments, this method returns the plates that the message to a parent has for the parent's distribution.

bayespy.inference.vmp.nodes.expfamily.ExponentialFamilyDistribution.random

`ExponentialFamilyDistribution.random(*params, plates=None)`

Draw a random sample from the distribution.

6.3.3 bayespy.inference.vmp.nodes.gaussian.GaussianDistribution

class `bayespy.inference.vmp.nodes.gaussian.GaussianDistribution`

Class for the VMP formulas of Gaussian variables.

Currently, supports only vector variables.

Notes

Message passing equations:

$$\mathbf{x} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Lambda}),$$

$$\mathbf{x}, \boldsymbol{\mu} \in \mathbb{R}^D, \quad \boldsymbol{\Lambda} \in \mathbb{R}^{D \times D}, \quad \boldsymbol{\Lambda} \text{ symmetric positive definite}$$

$$\log \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Lambda}) = -\frac{1}{2}\mathbf{x}^{\mathrm{T}}\boldsymbol{\Lambda}\mathbf{x} + \mathbf{x}^{\mathrm{T}}\boldsymbol{\Lambda}\boldsymbol{\mu} - \frac{1}{2}\boldsymbol{\mu}^{\mathrm{T}}\boldsymbol{\Lambda}\boldsymbol{\mu} + \frac{1}{2}\log|\boldsymbol{\Lambda}| - \frac{D}{2}\log(2\pi)$$

`__init__()`

Initialize self. See help(type(self)) for accurate signature.

Methods

`compute_cgf_from_parents`(u_mu_Lambda)	Compute $\mathrm{E}_{q(p)}[g(p)]$
`compute_fixed_moments_and_f`(x[, mask])	Compute the moments and $f(x)$ for a fixed value.
`compute_gradient`(g, u, phi)	Compute the standard gradient with respect to the natural parameters.
`compute_logpdf`(u, phi, g, f, ndims)	Compute E[log p(X)] given E[u], E[phi], E[g] and E[f].

Continued on next page

Table 6.45 – continued from previous page

compute_mask_to_parent(index, mask)	Maps the mask to the plates of a parent.
compute_message_to_parent(parent, index, u, ...)	Compute the message to a parent node.
compute_moments_and_cgf(phi[, mask])	Compute the moments and $g(\phi)$.
compute_phi_from_parents(u_mu_Lambda[, mask])	Compute the natural parameter vector given parent moments.
plates_from_parent(index, plates)	Resolve the plate mapping from a parent.
plates_to_parent(index, plates)	Resolves the plate mapping to a parent.
random(*phi[, plates])	Draw a random sample from the distribution.

bayespy.inference.vmp.nodes.gaussian.GaussianDistribution.compute_cgf_from_parents

GaussianDistribution.**compute_cgf_from_parents**(*u_mu_Lambda*)

Compute $\mathrm{E}_{q(p)}[g(p)]$

$$g(\boldsymbol{\mu}, \mathbf{\Lambda}) = -\frac{1}{2}\operatorname{tr}(\boldsymbol{\mu}\boldsymbol{\mu}^{\mathrm{T}}\mathbf{\Lambda}) + \frac{1}{2}\log|\mathbf{\Lambda}|$$

bayespy.inference.vmp.nodes.gaussian.GaussianDistribution.compute_fixed_moments_and_f

GaussianDistribution.**compute_fixed_moments_and_f**(*x*, *mask=True*)

Compute the moments and $f(x)$ for a fixed value.

$$\mathbf{u}(\mathbf{x}) = \begin{bmatrix} \mathbf{x} \\ \mathbf{x}\mathbf{x}^{\mathrm{T}} \end{bmatrix}$$
$$f(\mathbf{x}) = -\frac{D}{2}\log(2\pi)$$

bayespy.inference.vmp.nodes.gaussian.GaussianDistribution.compute_gradient

GaussianDistribution.**compute_gradient**(*g*, *u*, *phi*)

Compute the standard gradient with respect to the natural parameters.

Gradient of the moments:

$$\begin{aligned} \mathrm{d}\overline{\mathbf{u}} &= \begin{bmatrix} \frac{1}{2}\phi_2^{-1}\mathrm{d}\phi_2\phi_2^{-1}\phi_1 - \frac{1}{2}\phi_2^{-1}\mathrm{d}\phi_1 \\ -\frac{1}{4}\phi_2^{-1}\mathrm{d}\phi_2\phi_2^{-1}\phi_1\phi_1^{\mathrm{T}}\phi_2^{-1} - \frac{1}{4}\phi_2^{-1}\phi_1\phi_1^{\mathrm{T}}\phi_2^{-1}\mathrm{d}\phi_2\phi_2^{-1} + \frac{1}{2}\phi_2^{-1}\mathrm{d}\phi_2\phi_2^{-1} + \frac{1}{4}\phi_2^{-1}\mathrm{d}\phi_1\phi_1^{\mathrm{T}}\phi_2^{-1} + \frac{1}{4}\phi_2^{-1}\phi_1\mathrm{d}\phi_1^{\mathrm{T}}\phi_2^{-1} \end{bmatrix} \\ &= \begin{bmatrix} 2(\overline{u}_2 - \overline{u}_1\overline{u}_1^{\mathrm{T}})\mathrm{d}\phi_2\overline{u}_1 + (\overline{u}_2 - \overline{u}_1\overline{u}_1^{\mathrm{T}})\mathrm{d}\phi_1 \\ u_2 d\phi_2 u_2 - 2u_1u_1^T d\phi_2 u_1u_1^T + 2(u_2 - u_1u_1^T)d\phi_1 u_1^T \end{bmatrix} \end{aligned}$$

Standard gradient given the gradient with respect to the moments, that is, given the Riemannian gradient $\tilde{\nabla}$:

$$\nabla = \begin{bmatrix} (\overline{u}_2 - \overline{u}_1\overline{u}_1^{\mathrm{T}})\tilde{\nabla}_1 + 2(u_2 - u_1u_1^T)\tilde{\nabla}_2 u_1 \\ (u_2 - u_1u_1^T)\tilde{\nabla}_1 u_1^T + u_1\tilde{\nabla}_1^T(u_2 - u_1u_1^T) + 2u_2\tilde{\nabla}_2 u_2 - 2u_1u_1^T\tilde{\nabla}_2 u_1u_1^T \end{bmatrix}$$

bayespy.inference.vmp.nodes.gaussian.GaussianDistribution.compute_logpdf

GaussianDistribution.**compute_logpdf**(*u*, *phi*, *g*, *f*, *ndims*)

Compute E[log p(X)] given E[u], E[phi], E[g] and E[f]. Does not sum over plates.

bayespy.inference.vmp.nodes.gaussian.GaussianDistribution.compute_mask_to_parent

GaussianDistribution.**compute_mask_to_parent**(*index*, *mask*)

Maps the mask to the plates of a parent.

bayespy.inference.vmp.nodes.gaussian.GaussianDistribution.compute_message_to_parent

GaussianDistribution.**compute_message_to_parent**(*parent*, *index*, *u*, *u_mu_Lambda*)
Compute the message to a parent node.

$$\phi_{\boldsymbol{\mu}}(\mathbf{x}, \mathbf{\Lambda}) = \begin{bmatrix} \mathbf{\Lambda}\mathbf{x} \\ -\frac{1}{2}\mathbf{\Lambda} \end{bmatrix}$$
$$\phi_{\mathbf{\Lambda}}(\mathbf{x}, \boldsymbol{\mu}) = \begin{bmatrix} -\frac{1}{2}\mathbf{x}\mathbf{x}^{\mathrm{T}} + \frac{1}{2}\mathbf{x}\boldsymbol{\mu}^{\mathrm{T}} + \frac{1}{2}\boldsymbol{\mu}\mathbf{x}^{\mathrm{T}} - \frac{1}{2}\boldsymbol{\mu}\boldsymbol{\mu}^{\mathrm{T}} \\ \frac{1}{2} \end{bmatrix}$$

bayespy.inference.vmp.nodes.gaussian.GaussianDistribution.compute_moments_and_cgf

GaussianDistribution.**compute_moments_and_cgf**(*phi*, *mask=True*)
Compute the moments and $g(\phi)$.

$$\overline{\mathbf{u}}(\boldsymbol{\phi}) = \begin{bmatrix} -\frac{1}{2}\boldsymbol{\phi}_2^{-1}\boldsymbol{\phi}_1 \\ \frac{1}{4}\boldsymbol{\phi}_2^{-1}\boldsymbol{\phi}_1\boldsymbol{\phi}_1^{\mathrm{T}}\boldsymbol{\phi}_2^{-1} - \frac{1}{2}\boldsymbol{\phi}_2^{-1} \end{bmatrix}$$
$$g_{\boldsymbol{\phi}}(\boldsymbol{\phi}) = \frac{1}{4}\boldsymbol{\phi}_1^{\mathrm{T}}\boldsymbol{\phi}_2^{-1}\boldsymbol{\phi}_1 + \frac{1}{2}\log|-2\boldsymbol{\phi}_2|$$

bayespy.inference.vmp.nodes.gaussian.GaussianDistribution.compute_phi_from_parents

GaussianDistribution.**compute_phi_from_parents**(*u_mu_Lambda*, *mask=True*)
Compute the natural parameter vector given parent moments.

$$\boldsymbol{\phi}(\boldsymbol{\mu}, \mathbf{\Lambda}) = \begin{bmatrix} \mathbf{\Lambda}\boldsymbol{\mu} \\ -\frac{1}{2}\mathbf{\Lambda} \end{bmatrix}$$

bayespy.inference.vmp.nodes.gaussian.GaussianDistribution.plates_from_parent

GaussianDistribution.**plates_from_parent**(*index*, *plates*)
Resolve the plate mapping from a parent.

Given the plates of a parent's moments, this method returns the plates that the moments has for this distribution.

bayespy.inference.vmp.nodes.gaussian.GaussianDistribution.plates_to_parent

GaussianDistribution.**plates_to_parent**(*index*, *plates*)
Resolves the plate mapping to a parent.

Given the plates of the node's moments, this method returns the plates that the message to a parent has for the parent's distribution.

bayespy.inference.vmp.nodes.gaussian.GaussianDistribution.random

GaussianDistribution.**random**(**phi*, *plates=None*)
Draw a random sample from the distribution.

6.3.4 bayespy.inference.vmp.nodes.gaussian.GaussianARDDistribution

class bayespy.inference.vmp.nodes.gaussian.**GaussianARDDistribution**(*shape*, *ndim_mu*)

...

Log probability density function:

$$\log p(x|\mu,\alpha) = -\frac{1}{2}x^T \operatorname{diag}(\alpha) x + x^T \operatorname{diag}(\alpha)\mu - \frac{1}{2}\mu^T \operatorname{diag}(\alpha)\mu + \frac{1}{2}\sum_i \log\alpha_i - \frac{D}{2}\log(2\pi)$$

Parent has moments:

$$\begin{bmatrix} \alpha \circ \mu \\ \alpha \circ \mu \circ \mu \\ \alpha \\ \log(\alpha) \end{bmatrix}$$

__init__(*shape*, *ndim_mu*)

Methods

__init__(shape, ndim_mu)	
compute_cgf_from_parents(u_mu_alpha)	Compute the value of the cumulant generating function.
compute_fixed_moments_and_f(x[, mask])	Compute u(x) and f(x) for given x.
compute_gradient(g, u, phi)	Compute the standard gradient with respect to the natural parameters.
compute_logpdf(u, phi, g, f, ndims)	Compute E[log p(X)] given E[u], E[phi], E[g] and E[f].
compute_mask_to_parent(index, mask)	Maps the mask to the plates of a parent.
compute_message_to_parent(parent, index, u, ...)	...
compute_moments_and_cgf(phi[, mask])	
compute_phi_from_parents(u_mu_alpha[, mask])	
plates_from_parent(index, plates)	Resolve the plate mapping from a parent.
plates_to_parent(index, plates)	Resolves the plate mapping to a parent.
random(*phi[, plates])	Draw a random sample from the Gaussian distribution.

bayespy.inference.vmp.nodes.gaussian.GaussianARDDistribution.__init__

GaussianARDDistribution.**__init__**(*shape*, *ndim_mu*)

bayespy.inference.vmp.nodes.gaussian.GaussianARDDistribution.compute_cgf_from_parents

GaussianARDDistribution.**compute_cgf_from_parents**(*u_mu_alpha*)

Compute the value of the cumulant generating function.

bayespy.inference.vmp.nodes.gaussian.GaussianARDDistribution.compute_fixed_moments_and_f

GaussianARDDistribution.**compute_fixed_moments_and_f**(*x*, *mask=True*)

Compute u(x) and f(x) for given x.

bayespy.inference.vmp.nodes.gaussian.GaussianARDDistribution.compute_gradient

GaussianARDDistribution.**compute_gradient**(*g*, *u*, *phi*)

Compute the standard gradient with respect to the natural parameters.

Gradient of the moments:

$$\mathrm{d}\overline{\mathbf{u}} = \begin{bmatrix} \frac{1}{2}\phi_2^{-1}\mathrm{d}\phi_2\phi_2^{-1}\phi_1 - \frac{1}{2}\phi_2^{-1}\mathrm{d}\phi_1 \\ -\frac{1}{4}\phi_2^{-1}\mathrm{d}\phi_2\phi_2^{-1}\phi_1\phi_1^{\mathrm{T}}\phi_2^{-1} - \frac{1}{4}\phi_2^{-1}\phi_1\phi_1^{\mathrm{T}}\phi_2^{-1}\mathrm{d}\phi_2\phi_2^{-1} + \frac{1}{2}\phi_2^{-1}\mathrm{d}\phi_2\phi_2^{-1} + \frac{1}{4}\phi_2^{-1}\mathrm{d}\phi_1\phi_1^{\mathrm{T}}\phi_2^{-1} + \frac{1}{4}\phi_2^{-1}\phi_1\mathrm{d}\phi_1^{\mathrm{T}}\phi_2^{-1} \end{bmatrix}$$
$$= \begin{bmatrix} 2(\overline{u}_2 - \overline{u}_1\overline{u}_1^{\mathrm{T}})\mathrm{d}\phi_2\overline{u}_1 + (\overline{u}_2 - \overline{u}_1\overline{u}_1^{\mathrm{T}})\mathrm{d}\phi_1 \\ u_2 d\phi_2 u_2 - 2u_1u_1^T d\phi_2 u_1 u_1^T + 2(u_2 - u_1u_1^T)d\phi_1 u_1^T \end{bmatrix}$$

Standard gradient given the gradient with respect to the moments, that is, given the Riemannian gradient $\tilde{\nabla}$:

$$\nabla = \begin{bmatrix} (\overline{u}_2 - \overline{u}_1\overline{u}_1^{\mathrm{T}})\tilde{\nabla}_1 + 2(u_2 - u_1u_1^T)\tilde{\nabla}_2 u_1 \\ (u_2 - u_1u_1^T)\tilde{\nabla}_1 u_1^T + u_1\tilde{\nabla}_1^T(u_2 - u_1u_1^T) + 2u_2\tilde{\nabla}_2 u_2 - 2u_1u_1^T\tilde{\nabla}_2 u_1 u_1^T \end{bmatrix}$$

bayespy.inference.vmp.nodes.gaussian.GaussianARDDistribution.compute_logpdf

GaussianARDDistribution.**compute_logpdf**(*u*, *phi*, *g*, *f*, *ndims*)

Compute E[log p(X)] given E[u], E[phi], E[g] and E[f]. Does not sum over plates.

bayespy.inference.vmp.nodes.gaussian.GaussianARDDistribution.compute_mask_to_parent

GaussianARDDistribution.**compute_mask_to_parent**(*index*, *mask*)

Maps the mask to the plates of a parent.

bayespy.inference.vmp.nodes.gaussian.GaussianARDDistribution.compute_message_to_parent

GaussianARDDistribution.**compute_message_to_parent**(*parent*, *index*, *u*, *u_mu_alpha*)

...

$$m = \begin{bmatrix} x \\ [-\frac{1}{2}, \ldots, -\frac{1}{2}] \\ -\frac{1}{2}\mathrm{diag}(xx^T) \\ [\frac{1}{2}, \ldots, \frac{1}{2}] \end{bmatrix}$$

bayespy.inference.vmp.nodes.gaussian.GaussianARDDistribution.compute_moments_and_cgf

GaussianARDDistribution.**compute_moments_and_cgf**(*phi*, *mask=True*)

bayespy.inference.vmp.nodes.gaussian.GaussianARDDistribution.compute_phi_from_parents

GaussianARDDistribution.**compute_phi_from_parents**(*u_mu_alpha*, *mask=True*)

bayespy.inference.vmp.nodes.gaussian.GaussianARDDistribution.plates_from_parent

GaussianARDDistribution.**plates_from_parent**(*index*, *plates*)

Resolve the plate mapping from a parent.

Given the plates of a parent's moments, this method returns the plates that the moments has for this distribution.

bayespy.inference.vmp.nodes.gaussian.GaussianARDDistribution.plates_to_parent

GaussianARDDistribution.**plates_to_parent**(*index*, *plates*)
Resolves the plate mapping to a parent.

Given the plates of the node's moments, this method returns the plates that the message to a parent has for the parent's distribution.

bayespy.inference.vmp.nodes.gaussian.GaussianARDDistribution.random

GaussianARDDistribution.**random**(**phi*, *plates=None*)
Draw a random sample from the Gaussian distribution.

6.3.5 bayespy.inference.vmp.nodes.gaussian.GaussianGammaISODistribution

class bayespy.inference.vmp.nodes.gaussian.**GaussianGammaISODistribution**
Class for the VMP formulas of Gaussian-Gamma-ISO variables.

Currently, supports only vector variables.

__init__()
Initialize self. See help(type(self)) for accurate signature.

Methods

compute_cgf_from_parents(u_mu_Lambda, u_a, u_b)	Compute $\mathrm{E}_{q(p)}[g(p)]$
compute_fixed_moments_and_f(x, alpha[, mask])	Compute the moments and $f(x)$ for a fixed value.
compute_gradient(g, u, phi)	Compute the standard gradient with respect to the natural parameters.
compute_logpdf(u, phi, g, f, ndims)	Compute E[log p(X)] given E[u], E[phi], E[g] and E[f].
compute_mask_to_parent(index, mask)	Maps the mask to the plates of a parent.
compute_message_to_parent(parent, index, u, ...)	Compute the message to a parent node.
compute_moments_and_cgf(phi[, mask])	Compute the moments and $g(\phi)$.
compute_phi_from_parents(u_mu_Lambda, u_a, u_b)	Compute the natural parameter vector given parent moments.
plates_from_parent(index, plates)	Resolve the plate mapping from a parent.
plates_to_parent(index, plates)	Resolves the plate mapping to a parent.
random(*params[, plates])	Draw a random sample from the distribution.

bayespy.inference.vmp.nodes.gaussian.GaussianGammaISODistribution.compute_cgf_from_parents

GaussianGammaISODistribution.**compute_cgf_from_parents**(*u_mu_Lambda*, *u_a*, *u_b*)
Compute $\mathrm{E}_{q(p)}[g(p)]$

bayespy.inference.vmp.nodes.gaussian.GaussianGammaISODistribution.compute_fixed_moments_and_f

GaussianGammaISODistribution.**compute_fixed_moments_and_f**(*x*, *alpha*, *mask=True*)
Compute the moments and $f(x)$ for a fixed value.

bayespy.inference.vmp.nodes.gaussian.GaussianGammaISODistribution.compute_gradient

GaussianGammaISODistribution.**compute_gradient**(*g*, *u*, *phi*)
Compute the standard gradient with respect to the natural parameters.

bayespy.inference.vmp.nodes.gaussian.GaussianGammaISODistribution.compute_logpdf

GaussianGammaISODistribution.**compute_logpdf**(*u*, *phi*, *g*, *f*, *ndims*)
Compute E[log p(X)] given E[u], E[phi], E[g] and E[f]. Does not sum over plates.

bayespy.inference.vmp.nodes.gaussian.GaussianGammaISODistribution.compute_mask_to_parent

GaussianGammaISODistribution.**compute_mask_to_parent**(*index*, *mask*)
Maps the mask to the plates of a parent.

bayespy.inference.vmp.nodes.gaussian.GaussianGammaISODistribution.compute_message_to_parent

GaussianGammaISODistribution.**compute_message_to_parent**(*parent*, *index*, *u*, *u_mu_Lambda*, *u_a*, *u_b*)
Compute the message to a parent node.

bayespy.inference.vmp.nodes.gaussian.GaussianGammaISODistribution.compute_moments_and_cgf

GaussianGammaISODistribution.**compute_moments_and_cgf**(*phi*, *mask=True*)
Compute the moments and $g(\phi)$.

bayespy.inference.vmp.nodes.gaussian.GaussianGammaISODistribution.compute_phi_from_parents

GaussianGammaISODistribution.**compute_phi_from_parents**(*u_mu_Lambda*, *u_a*, *u_b*, *mask=True*)
Compute the natural parameter vector given parent moments.

bayespy.inference.vmp.nodes.gaussian.GaussianGammaISODistribution.plates_from_parent

GaussianGammaISODistribution.**plates_from_parent**(*index*, *plates*)
Resolve the plate mapping from a parent.

Given the plates of a parent's moments, this method returns the plates that the moments has for this distribution.

bayespy.inference.vmp.nodes.gaussian.GaussianGammaISODistribution.plates_to_parent

GaussianGammaISODistribution.**plates_to_parent**(*index*, *plates*)
Resolves the plate mapping to a parent.

Given the plates of the node's moments, this method returns the plates that the message to a parent has for the parent's distribution.

bayespy.inference.vmp.nodes.gaussian.GaussianGammaISODistribution.random

GaussianGammaISODistribution.**random**(**params*, *plates=None*)
Draw a random sample from the distribution.

6.3.6 bayespy.inference.vmp.nodes.gaussian.GaussianGammaARDDistribution

class bayespy.inference.vmp.nodes.gaussian.**GaussianGammaARDDistribution**

__init__()

Methods

__init__()	
compute_cgf_from_parents(*u_parents)	
compute_fixed_moments_and_f(x[, mask])	
compute_gradient(g, u, phi)	Compute the standard gradient with respect to the natural parameters.
compute_logpdf(u, phi, g, f, ndims)	Compute E[log p(X)] given E[u], E[phi], E[g] and E[f].
compute_mask_to_parent(index, mask)	Maps the mask to the plates of a parent.
compute_message_to_parent(parent, index, ...)	
compute_moments_and_cgf(phi[, mask])	
compute_phi_from_parents(*u_parents[, mask])	
plates_from_parent(index, plates)	Resolve the plate mapping from a parent.
plates_to_parent(index, plates)	Resolves the plate mapping to a parent.
random(*params[, plates])	Draw a random sample from the distribution.

bayespy.inference.vmp.nodes.gaussian.GaussianGammaARDDistribution.__init__

GaussianGammaARDDistribution.**__init__**()

bayespy.inference.vmp.nodes.gaussian.GaussianGammaARDDistribution.compute_cgf_from_parents

GaussianGammaARDDistribution.**compute_cgf_from_parents**(**u_parents*)

bayespy.inference.vmp.nodes.gaussian.GaussianGammaARDDistribution.compute_fixed_moments_and_f

GaussianGammaARDDistribution.**compute_fixed_moments_and_f**(*x*, *mask=True*)

bayespy.inference.vmp.nodes.gaussian.GaussianGammaARDDistribution.compute_gradient

GaussianGammaARDDistribution.**compute_gradient**(*g*, *u*, *phi*)
Compute the standard gradient with respect to the natural parameters.

bayespy.inference.vmp.nodes.gaussian.GaussianGammaARDDistribution.compute_logpdf

GaussianGammaARDDistribution.**compute_logpdf**(*u*, *phi*, *g*, *f*, *ndims*)
Compute E[log p(X)] given E[u], E[phi], E[g] and E[f]. Does not sum over plates.

bayespy.inference.vmp.nodes.gaussian.GaussianGammaARDDistribution.compute_mask_to_parent

GaussianGammaARDDistribution.**compute_mask_to_parent**(*index*, *mask*)
Maps the mask to the plates of a parent.

bayespy.inference.vmp.nodes.gaussian.GaussianGammaARDDistribution.compute_message_to_parent

GaussianGammaARDDistribution.**compute_message_to_parent**(*parent*, *index*, *u_self*, **u_parents*)

bayespy.inference.vmp.nodes.gaussian.GaussianGammaARDDistribution.compute_moments_and_cgf

GaussianGammaARDDistribution.**compute_moments_and_cgf**(*phi*, *mask=True*)

bayespy.inference.vmp.nodes.gaussian.GaussianGammaARDDistribution.compute_phi_from_parents

GaussianGammaARDDistribution.**compute_phi_from_parents**(**u_parents*, *mask=True*)

bayespy.inference.vmp.nodes.gaussian.GaussianGammaARDDistribution.plates_from_parent

GaussianGammaARDDistribution.**plates_from_parent**(*index*, *plates*)
Resolve the plate mapping from a parent.

Given the plates of a parent's moments, this method returns the plates that the moments has for this distribution.

bayespy.inference.vmp.nodes.gaussian.GaussianGammaARDDistribution.plates_to_parent

GaussianGammaARDDistribution.**plates_to_parent**(*index*, *plates*)
Resolves the plate mapping to a parent.

Given the plates of the node's moments, this method returns the plates that the message to a parent has for the parent's distribution.

bayespy.inference.vmp.nodes.gaussian.GaussianGammaARDDistribution.random

GaussianGammaARDDistribution.**random**(**params*, *plates=None*)
Draw a random sample from the distribution.

6.3.7 bayespy.inference.vmp.nodes.gaussian.GaussianWishartDistribution

class bayespy.inference.vmp.nodes.gaussian.**GaussianWishartDistribution**
Class for the VMP formulas of Gaussian-Wishart variables.

Currently, supports only vector variables.

__init__()
Initialize self. See help(type(self)) for accurate signature.

Methods

compute_cgf_from_parents(u_mu_alpha, u_V, u_n)	Compute $\mathrm{E}_{q(p)}[g(p)]$
compute_fixed_moments_and_f(x, Lambda[, mask])	Compute the moments and $f(x)$ for a fixed value.
compute_gradient(g, u, phi)	Compute the standard gradient with respect to the natural parameters.
compute_logpdf(u, phi, g, f, ndims)	Compute E[log p(X)] given E[u], E[phi], E[g] and E[f].
compute_mask_to_parent(index, mask)	Maps the mask to the plates of a parent.
compute_message_to_parent(parent, index, u, ...)	Compute the message to a parent node.
compute_moments_and_cgf(phi[, mask])	Compute the moments and $g(\phi)$.
compute_phi_from_parents(u_mu_alpha, u_V, u_n)	Compute the natural parameter vector given parent moments.
plates_from_parent(index, plates)	Resolve the plate mapping from a parent.
plates_to_parent(index, plates)	Resolves the plate mapping to a parent.
random(*params[, plates])	Draw a random sample from the distribution.

bayespy.inference.vmp.nodes.gaussian.GaussianWishartDistribution.compute_cgf_from_parents

GaussianWishartDistribution.**compute_cgf_from_parents**(*u_mu_alpha*, *u_V*, *u_n*)

Compute $\mathrm{E}_{q(p)}[g(p)]$

bayespy.inference.vmp.nodes.gaussian.GaussianWishartDistribution.compute_fixed_moments_and_f

GaussianWishartDistribution.**compute_fixed_moments_and_f**(*x*, *Lambda*, *mask=True*)

Compute the moments and $f(x)$ for a fixed value.

bayespy.inference.vmp.nodes.gaussian.GaussianWishartDistribution.compute_gradient

GaussianWishartDistribution.**compute_gradient**(*g*, *u*, *phi*)

Compute the standard gradient with respect to the natural parameters.

bayespy.inference.vmp.nodes.gaussian.GaussianWishartDistribution.compute_logpdf

GaussianWishartDistribution.**compute_logpdf**(*u*, *phi*, *g*, *f*, *ndims*)

Compute E[log p(X)] given E[u], E[phi], E[g] and E[f]. Does not sum over plates.

bayespy.inference.vmp.nodes.gaussian.GaussianWishartDistribution.compute_mask_to_parent

GaussianWishartDistribution.**compute_mask_to_parent**(*index*, *mask*)

Maps the mask to the plates of a parent.

bayespy.inference.vmp.nodes.gaussian.GaussianWishartDistribution.compute_message_to_parent

GaussianWishartDistribution.**compute_message_to_parent**(*parent*, *index*, *u*, *u_mu_alpha*, *u_V*, *u_n*)

Compute the message to a parent node.

bayespy.inference.vmp.nodes.gaussian.GaussianWishartDistribution.compute_moments_and_cgf

GaussianWishartDistribution.**compute_moments_and_cgf**(*phi*, *mask=True*)

Compute the moments and $g(\phi)$.

bayespy.inference.vmp.nodes.gaussian.GaussianWishartDistribution.compute_phi_from_parents

GaussianWishartDistribution.**compute_phi_from_parents**(*u_mu_alpha*, *u_V*, *u_n*, *mask=True*)

Compute the natural parameter vector given parent moments.

bayespy.inference.vmp.nodes.gaussian.GaussianWishartDistribution.plates_from_parent

GaussianWishartDistribution.**plates_from_parent**(*index*, *plates*)

Resolve the plate mapping from a parent.

Given the plates of a parent's moments, this method returns the plates that the moments has for this distribution.

bayespy.inference.vmp.nodes.gaussian.GaussianWishartDistribution.plates_to_parent

GaussianWishartDistribution.**plates_to_parent**(*index*, *plates*)

Resolves the plate mapping to a parent.

Given the plates of the node's moments, this method returns the plates that the message to a parent has for the parent's distribution.

bayespy.inference.vmp.nodes.gaussian.GaussianWishartDistribution.random

GaussianWishartDistribution.**random**(**params*, *plates=None*)

Draw a random sample from the distribution.

6.3.8 bayespy.inference.vmp.nodes.gaussian_markov_chain.GaussianMarkovChainDistribution

class bayespy.inference.vmp.nodes.gaussian_markov_chain.**GaussianMarkovChainDistribution**(*N*, *D*)

Sub-classes implement distribution specific computations.

__init__(*N*, *D*)

Methods

__init__(N, D)	
compute_cgf_from_parents(u_mu, u_Lambda, ...)	Compute CGF using the moments of the parents.
compute_fixed_moments_and_f(x[, mask])	Compute u(x) and f(x) for given x.
compute_gradient(g, u, phi)	Compute the standard gradient with respect to the natural parameters.
compute_logpdf(u, phi, g, f, ndims)	Compute E[log p(X)] given E[u], E[phi], E[g] and E[f].
compute_mask_to_parent(index, mask)	
compute_message_to_parent(parent, index, u, ...)	Compute a message to a parent.
compute_moments_and_cgf(phi[, mask])	Compute the moments and the cumulant-generating function.
compute_phi_from_parents(u_mu, u_Lambda, ...)	Compute the natural parameters using parents' moments.
plates_from_parent(index, plates)	Compute the plates using information of a parent node.
plates_to_parent(index, plates)	Computes the plates of this node with respect to a parent.
random(*params[, plates])	Draw a random sample from the distribution.

bayespy.inference.vmp.nodes.gaussian_markov_chain.GaussianMarkovChainDistribution.__init__

GaussianMarkovChainDistribution.**__init__**(*N*, *D*)

bayespy.inference.vmp.nodes.gaussian_markov_chain.GaussianMarkovChainDistribution.compute_cgf_from_p

GaussianMarkovChainDistribution.**compute_cgf_from_parents**(*u_mu*, *u_Lambda*, *u_A*, *u_v*, **u_inputs*)

Compute CGF using the moments of the parents.

bayespy.inference.vmp.nodes.gaussian_markov_chain.GaussianMarkovChainDistribution.compute_fixed_mon

GaussianMarkovChainDistribution.**compute_fixed_moments_and_f**(*x*, *mask=True*)

Compute u(x) and f(x) for given x.

bayespy.inference.vmp.nodes.gaussian_markov_chain.GaussianMarkovChainDistribution.compute_gradient

GaussianMarkovChainDistribution.**compute_gradient**(*g*, *u*, *phi*)

Compute the standard gradient with respect to the natural parameters.

bayespy.inference.vmp.nodes.gaussian_markov_chain.GaussianMarkovChainDistribution.compute_logpdf

GaussianMarkovChainDistribution.**compute_logpdf**(*u*, *phi*, *g*, *f*, *ndims*)

Compute E[log p(X)] given E[u], E[phi], E[g] and E[f]. Does not sum over plates.

bayespy.inference.vmp.nodes.gaussian_markov_chain.GaussianMarkovChainDistribution.compute_mask_to_p

GaussianMarkovChainDistribution.**compute_mask_to_parent**(*index*, *mask*)

bayespy.inference.vmp.nodes.gaussian_markov_chain.GaussianMarkovChainDistribution.compute_message_

GaussianMarkovChainDistribution.**compute_message_to_parent**(*parent*, *index*, *u*, *u_mu*, *u_Lambda*, *u_A*, *u_v*, **u_inputs*)

Compute a message to a parent.

bayespy.inference.vmp.nodes.gaussian_markov_chain.GaussianMarkovChainDistribution.compute_moments_

GaussianMarkovChainDistribution.**compute_moments_and_cgf**(*phi*, *mask=True*)

Compute the moments and the cumulant-generating function.

This basically performs the filtering and smoothing for the variable.

Parameters **phi**

Returns u

g

bayespy.inference.vmp.nodes.gaussian_markov_chain.GaussianMarkovChainDistribution.compute_phi_from_p

GaussianMarkovChainDistribution.**compute_phi_from_parents**(*u_mu*, *u_Lambda*, *u_A*, *u_v*, **u_inputs*, *mask=True*)

Compute the natural parameters using parents' moments.

Parameters u_parents : list of list of arrays

List of parents' lists of moments.

Returns phi : list of arrays

Natural parameters.

dims : tuple

Shape of the variable part of phi.

bayespy.inference.vmp.nodes.gaussian_markov_chain.GaussianMarkovChainDistribution.plates_from_parent

GaussianMarkovChainDistribution.**plates_from_parent**(*index*, *plates*)

Compute the plates using information of a parent node.

If the plates of the parents are: mu: (...) Lambda: (...) A: (...,N-1,D) v: (...,N-1,D) N: ()

the resulting plates of this node are (...)

Parameters index : int

Index of the parent to use.

bayespy.inference.vmp.nodes.gaussian_markov_chain.GaussianMarkovChainDistribution.plates_to_parent

GaussianMarkovChainDistribution.**plates_to_parent**(*index*, *plates*)

Computes the plates of this node with respect to a parent.

If this node has plates (...), the latent dimensionality is D and the number of time instances is N, the plates with respect to the parents are:

mu: (...) Lambda: (...) A: (...,N-1,D) v: (...,N-1,D)

bayespy.inference.vmp.nodes.gaussian_markov_chain.GaussianMarkovChainDistribution.random

GaussianMarkovChainDistribution.**random**(**params*, *plates=None*)

Draw a random sample from the distribution.

6.3.9 bayespy.inference.vmp.nodes.gaussian_markov_chain.SwitchingGaussianMarkovChainDis

class bayespy.inference.vmp.nodes.gaussian_markov_chain.**SwitchingGaussianMarkovChainDistributi**

Sub-classes implement distribution specific computations.

__init__(*N*, *D*, *K*)

Methods

`__init__`(N, D, K)	
`compute_cgf_from_parents`(u_mu, u_Lambda, ...)	Compute CGF using the moments of the parents.
`compute_fixed_moments_and_f`(x[, mask])	Compute u(x) and f(x) for given x.
`compute_gradient`(g, u, phi)	Compute the standard gradient with respect to the natural parameters.
`compute_logpdf`(u, phi, g, f, ndims)	Compute E[log p(X)] given E[u], E[phi], E[g] and E[f].
`compute_mask_to_parent`(index, mask)	
`compute_message_to_parent`(parent, index, u, ...)	Compute a message to a parent.
`compute_moments_and_cgf`(phi[, mask])	Compute the moments and the cumulant-generating function.
`compute_phi_from_parents`(u_mu, u_Lambda, ...)	Compute the natural parameters using parents' moments.
`plates_from_parent`(index, plates)	Compute the plates using information of a parent node.
`plates_to_parent`(index, plates)	Computes the plates of this node with respect to a parent.
`random`(*params[, plates])	Draw a random sample from the distribution.

bayespy.inference.vmp.nodes.gaussian_markov_chain.SwitchingGaussianMarkovChainDistribution.__init__

SwitchingGaussianMarkovChainDistribution.**__init__**(*N*, *D*, *K*)

bayespy.inference.vmp.nodes.gaussian_markov_chain.SwitchingGaussianMarkovChainDistribution.compute_

SwitchingGaussianMarkovChainDistribution.**compute_cgf_from_parents**(*u_mu*, *u_Lambda*, *u_B*, *u_Z*, *u_v*)

Compute CGF using the moments of the parents.

bayespy.inference.vmp.nodes.gaussian_markov_chain.SwitchingGaussianMarkovChainDistribution.compute_

SwitchingGaussianMarkovChainDistribution.**compute_fixed_moments_and_f**(*x*, *mask=True*)

Compute u(x) and f(x) for given x.

bayespy.inference.vmp.nodes.gaussian_markov_chain.SwitchingGaussianMarkovChainDistribution.compute_

SwitchingGaussianMarkovChainDistribution.**compute_gradient**(*g*, *u*, *phi*)

Compute the standard gradient with respect to the natural parameters.

bayespy.inference.vmp.nodes.gaussian_markov_chain.SwitchingGaussianMarkovChainDistribution.compute_

SwitchingGaussianMarkovChainDistribution.**compute_logpdf**(*u*, *phi*, *g*, *f*, *ndims*)

Compute E[log p(X)] given E[u], E[phi], E[g] and E[f]. Does not sum over plates.

bayespy.inference.vmp.nodes.gaussian_markov_chain.SwitchingGaussianMarkovChainDistribution.compute_

SwitchingGaussianMarkovChainDistribution.**compute_mask_to_parent**(*index*, *mask*)

bayespy.inference.vmp.nodes.gaussian_markov_chain.SwitchingGaussianMarkovChainDistribution.compute_

`SwitchingGaussianMarkovChainDistribution.`**`compute_message_to_parent`**(*parent*, *index*, *u*, *u_mu*, *u_Lambda*, *u_B*, *u_Z*, *u_v*)

Compute a message to a parent.

bayespy.inference.vmp.nodes.gaussian_markov_chain.SwitchingGaussianMarkovChainDistribution.compute_

`SwitchingGaussianMarkovChainDistribution.`**`compute_moments_and_cgf`**(*phi*, *mask=True*)

Compute the moments and the cumulant-generating function.

This basically performs the filtering and smoothing for the variable.

Parameters phi

Returns u

g

bayespy.inference.vmp.nodes.gaussian_markov_chain.SwitchingGaussianMarkovChainDistribution.compute_

`SwitchingGaussianMarkovChainDistribution.`**`compute_phi_from_parents`**(*u_mu*, *u_Lambda*, *u_B*, *u_Z*, *u_v*, *mask=True*)

Compute the natural parameters using parents' moments.

Parameters u_parents : list of list of arrays

List of parents' lists of moments.

Returns phi : list of arrays

Natural parameters.

dims : tuple

Shape of the variable part of phi.

bayespy.inference.vmp.nodes.gaussian_markov_chain.SwitchingGaussianMarkovChainDistribution.plates_fro

`SwitchingGaussianMarkovChainDistribution.`**`plates_from_parent`**(*index*, *plates*)

Compute the plates using information of a parent node.

If the plates of the parents are: mu: (...) Lambda: (...) B: (...,D) S: (...,N-1) v: (...,N-1,D) N: ()

the resulting plates of this node are (...)

Parameters index : int

Index of the parent to use.

bayespy.inference.vmp.nodes.gaussian_markov_chain.SwitchingGaussianMarkovChainDistribution.plates_to_

SwitchingGaussianMarkovChainDistribution.**plates_to_parent**(*index*, *plates*)
Computes the plates of this node with respect to a parent.

If this node has plates (...), the latent dimensionality is D and the number of time instances is N, the plates with respect to the parents are:

mu: (...) Lambda: (...) A: (...,N-1,D) v: (...,N-1,D)

bayespy.inference.vmp.nodes.gaussian_markov_chain.SwitchingGaussianMarkovChainDistribution.random

SwitchingGaussianMarkovChainDistribution.**random**(**params*, *plates=None*)
Draw a random sample from the distribution.

6.3.10 bayespy.inference.vmp.nodes.gaussian_markov_chain.VaryingGaussianMarkovChainDis

class bayespy.inference.vmp.nodes.gaussian_markov_chain.**VaryingGaussianMarkovChainDistributio**

Sub-classes implement distribution specific computations.

__init__(*N*, *D*)

Methods

__init__(N, D)	
compute_cgf_from_parents(u_mu, u_Lambda, ...)	Compute CGF using the moments of the parents.
compute_fixed_moments_and_f(x[, mask])	Compute u(x) and f(x) for given x.
compute_gradient(g, u, phi)	Compute the standard gradient with respect to the natural parameters.
compute_logpdf(u, phi, g, f, ndims)	Compute E[log p(X)] given E[u], E[phi], E[g] and E[f].
compute_mask_to_parent(index, mask)	
compute_message_to_parent(parent, index, u, ...)	Compute a message to a parent.
compute_moments_and_cgf(phi[, mask])	Compute the moments and the cumulant-generating function.
compute_phi_from_parents(u_mu, u_Lambda, ...)	Compute the natural parameters using parents' moments.
plates_from_parent(index, plates)	Compute the plates using information of a parent node.
plates_to_parent(index, plates)	Computes the plates of this node with respect to a parent.
random(*params[, plates])	Draw a random sample from the distribution.

bayespy.inference.vmp.nodes.gaussian_markov_chain.VaryingGaussianMarkovChainDistribution.__init__

VaryingGaussianMarkovChainDistribution.**__init__**(*N*, *D*)

bayespy.inference.vmp.nodes.gaussian_markov_chain.VaryingGaussianMarkovChainDistribution.compute_cg

VaryingGaussianMarkovChainDistribution.**compute_cgf_from_parents**(*u_mu*, *u_Lambda*, *u_B*, *u_S*, *u_v*)
Compute CGF using the moments of the parents.

bayespy.inference.vmp.nodes.gaussian_markov_chain.VaryingGaussianMarkovChainDistribution.compute_fix

VaryingGaussianMarkovChainDistribution.**compute_fixed_moments_and_f**(*x*, *mask=True*)

Compute u(x) and f(x) for given x.

bayespy.inference.vmp.nodes.gaussian_markov_chain.VaryingGaussianMarkovChainDistribution.compute_gr

VaryingGaussianMarkovChainDistribution.**compute_gradient**(*g*, *u*, *phi*)

Compute the standard gradient with respect to the natural parameters.

bayespy.inference.vmp.nodes.gaussian_markov_chain.VaryingGaussianMarkovChainDistribution.compute_log

VaryingGaussianMarkovChainDistribution.**compute_logpdf**(*u*, *phi*, *g*, *f*, *ndims*)

Compute E[log p(X)] given E[u], E[phi], E[g] and E[f]. Does not sum over plates.

bayespy.inference.vmp.nodes.gaussian_markov_chain.VaryingGaussianMarkovChainDistribution.compute_ma

VaryingGaussianMarkovChainDistribution.**compute_mask_to_parent**(*index*, *mask*)

bayespy.inference.vmp.nodes.gaussian_markov_chain.VaryingGaussianMarkovChainDistribution.compute_me

VaryingGaussianMarkovChainDistribution.**compute_message_to_parent**(*parent*, *index*, *u*, *u_mu*, *u_Lambda*, *u_B*, *u_S*, *u_v*)

Compute a message to a parent.

bayespy.inference.vmp.nodes.gaussian_markov_chain.VaryingGaussianMarkovChainDistribution.compute_mo

VaryingGaussianMarkovChainDistribution.**compute_moments_and_cgf**(*phi*, *mask=True*)

Compute the moments and the cumulant-generating function.

This basically performs the filtering and smoothing for the variable.

Parameters **phi**

Returns u

g

bayespy.inference.vmp.nodes.gaussian_markov_chain.VaryingGaussianMarkovChainDistribution.compute_ph

VaryingGaussianMarkovChainDistribution.**compute_phi_from_parents**(*u_mu*, *u_Lambda*, *u_B*, *u_S*, *u_v*, *mask=True*)

Compute the natural parameters using parents' moments.

Parameters **u_parents** : list of list of arrays

List of parents' lists of moments.

Returns **phi** : list of arrays

Natural parameters.

dims : tuple

Shape of the variable part of phi.

bayespy.inference.vmp.nodes.gaussian_markov_chain.VaryingGaussianMarkovChainDistribution.plates_from_

`VaryingGaussianMarkovChainDistribution.`**`plates_from_parent`**(*index*, *plates*)

Compute the plates using information of a parent node.

If the plates of the parents are: mu: (...) Lambda: (...) B: (...,D) S: (...,N-1) v: (...,N-1,D) N: ()

the resulting plates of this node are (...)

Parameters **index** : int

Index of the parent to use.

bayespy.inference.vmp.nodes.gaussian_markov_chain.VaryingGaussianMarkovChainDistribution.plates_to_pa

`VaryingGaussianMarkovChainDistribution.`**`plates_to_parent`**(*index*, *plates*)

Computes the plates of this node with respect to a parent.

If this node has plates (...), the latent dimensionality is D and the number of time instances is N, the plates with respect to the parents are:

mu: (...) Lambda: (...) A: (...,N-1,D) v: (...,N-1,D)

bayespy.inference.vmp.nodes.gaussian_markov_chain.VaryingGaussianMarkovChainDistribution.random

`VaryingGaussianMarkovChainDistribution.`**`random`**(**params*, *plates=None*)

Draw a random sample from the distribution.

6.3.11 bayespy.inference.vmp.nodes.gamma.GammaDistribution

class `bayespy.inference.vmp.nodes.gamma.`**`GammaDistribution`**

Class for the VMP formulas of gamma variables.

`__init__`()

Initialize self. See help(type(self)) for accurate signature.

Methods

`compute_cgf_from_parents`(*u_parents)	Compute $\mathrm{E}_{q(p)}[g(p)]$
`compute_fixed_moments_and_f`(x[, mask])	Compute the moments and $f(x)$ for a fixed value.
`compute_gradient`(g, u, phi)	Compute the moments and $g(\phi)$.
`compute_logpdf`(u, phi, g, f, ndims)	Compute E[log p(X)] given E[u], E[phi], E[g] and E[f].
`compute_mask_to_parent`(index, mask)	Maps the mask to the parent.

Continued on next page

Table 6.53 – continued from previous page

compute_message_to_parent(parent, index, ...)	Compute the message to a parent node.
compute_moments_and_cgf(phi[, mask])	Compute the moments and $g(\phi)$.
compute_phi_from_parents(*u_parents[, mask])	Compute the natural parameter vector given parent moments.
plates_from_parent(index, plates)	Resolve the plate mapping from a parent.
plates_to_parent(index, plates)	Resolves the plate mapping to a parent.
random(*phi[, plates])	Draw a random sample from the distribution.

bayespy.inference.vmp.nodes.gamma.GammaDistribution.compute_cgf_from_parents

GammaDistribution.**compute_cgf_from_parents**(**u_parents*)

Compute $\mathrm{E}_{q(p)}[g(p)]$

bayespy.inference.vmp.nodes.gamma.GammaDistribution.compute_fixed_moments_and_f

GammaDistribution.**compute_fixed_moments_and_f**(*x*, *mask=True*)

Compute the moments and $f(x)$ for a fixed value.

bayespy.inference.vmp.nodes.gamma.GammaDistribution.compute_gradient

GammaDistribution.**compute_gradient**(*g*, *u*, *phi*)

Compute the moments and $g(\phi)$.

$$\mathrm{d}\overline{\mathbf{u}} = \begin{bmatrix} -\frac{\mathrm{d}\phi_2}{phi_1} + \frac{\phi_2}{\phi_1^2}\mathrm{d}\phi_1 \\ \psi^{(1)}(\phi_2)\mathrm{d}\phi_2 - \frac{1}{\phi_1}\mathrm{d}\phi_1 \end{bmatrix}$$

Standard gradient given the gradient with respect to the moments, that is, given the Riemannian gradient $\tilde{\nabla}$:

$$\nabla = \begin{bmatrix} \nabla_1\frac{\phi_2}{\phi_1^2} - \nabla_2\frac{1}{\phi_1} \\ \nabla_2\psi^{(1)}(\phi_2) - \nabla_1\frac{1}{\phi_1} \end{bmatrix}$$

bayespy.inference.vmp.nodes.gamma.GammaDistribution.compute_logpdf

GammaDistribution.**compute_logpdf**(*u*, *phi*, *g*, *f*, *ndims*)

Compute E[log p(X)] given E[u], E[phi], E[g] and E[f]. Does not sum over plates.

bayespy.inference.vmp.nodes.gamma.GammaDistribution.compute_mask_to_parent

GammaDistribution.**compute_mask_to_parent**(*index*, *mask*)

Maps the mask to the plates of a parent.

bayespy.inference.vmp.nodes.gamma.GammaDistribution.compute_message_to_parent

GammaDistribution.**compute_message_to_parent**(*parent*, *index*, *u_self*, **u_parents*)

Compute the message to a parent node.

bayespy.inference.vmp.nodes.gamma.GammaDistribution.compute_moments_and_cgf

GammaDistribution.**compute_moments_and_cgf**(*phi*, *mask=True*)
Compute the moments and $g(\phi)$.

$$\overline{\mathbf{u}}(\boldsymbol{\phi}) = \begin{bmatrix} -\frac{\phi_2}{\phi_1} \\ \psi(\phi_2) - \log(-\phi_1) \end{bmatrix}$$
$$g_{\boldsymbol{\phi}}(\boldsymbol{\phi}) = TODO$$

bayespy.inference.vmp.nodes.gamma.GammaDistribution.compute_phi_from_parents

GammaDistribution.**compute_phi_from_parents**(**u_parents*, *mask=True*)
Compute the natural parameter vector given parent moments.

bayespy.inference.vmp.nodes.gamma.GammaDistribution.plates_from_parent

GammaDistribution.**plates_from_parent**(*index*, *plates*)
Resolve the plate mapping from a parent.

Given the plates of a parent's moments, this method returns the plates that the moments has for this distribution.

bayespy.inference.vmp.nodes.gamma.GammaDistribution.plates_to_parent

GammaDistribution.**plates_to_parent**(*index*, *plates*)
Resolves the plate mapping to a parent.

Given the plates of the node's moments, this method returns the plates that the message to a parent has for the parent's distribution.

bayespy.inference.vmp.nodes.gamma.GammaDistribution.random

GammaDistribution.**random**(**phi*, *plates=None*)
Draw a random sample from the distribution.

6.3.12 bayespy.inference.vmp.nodes.wishart.WishartDistribution

class bayespy.inference.vmp.nodes.wishart.**WishartDistribution**
Sub-classes implement distribution specific computations.

__init__()
Initialize self. See help(type(self)) for accurate signature.

Methods

compute_cgf_from_parents(*u_parents)	
compute_fixed_moments_and_f(Lambda[, mask])	Compute u(x) and f(x) for given x.
compute_gradient(g, u, phi)	Compute the standard gradient with respect to the natural parameters.
compute_logpdf(u, phi, g, f, ndims)	Compute E[log p(X)] given E[u], E[phi], E[g] and E[f].

Continued on next page

Table 6.54 – continued from previous page

compute_mask_to_parent(index, mask)	Maps the mask to the plates of a parent.
compute_message_to_parent(parent, index, ...)	
compute_moments_and_cgf(phi[, mask])	
compute_phi_from_parents(*u_parents[, mask])	
plates_from_parent(index, plates)	Resolve the plate mapping from a parent.
plates_to_parent(index, plates)	Resolves the plate mapping to a parent.
random(*params[, plates])	Draw a random sample from the distribution.

bayespy.inference.vmp.nodes.wishart.WishartDistribution.compute_cgf_from_parents

WishartDistribution.**compute_cgf_from_parents**(**u_parents*)

bayespy.inference.vmp.nodes.wishart.WishartDistribution.compute_fixed_moments_and_f

WishartDistribution.**compute_fixed_moments_and_f**(*Lambda*, *mask=True*)

Compute u(x) and f(x) for given x.

bayespy.inference.vmp.nodes.wishart.WishartDistribution.compute_gradient

WishartDistribution.**compute_gradient**(*g*, *u*, *phi*)

Compute the standard gradient with respect to the natural parameters.

bayespy.inference.vmp.nodes.wishart.WishartDistribution.compute_logpdf

WishartDistribution.**compute_logpdf**(*u*, *phi*, *g*, *f*, *ndims*)

Compute E[log p(X)] given E[u], E[phi], E[g] and E[f]. Does not sum over plates.

bayespy.inference.vmp.nodes.wishart.WishartDistribution.compute_mask_to_parent

WishartDistribution.**compute_mask_to_parent**(*index*, *mask*)

Maps the mask to the plates of a parent.

bayespy.inference.vmp.nodes.wishart.WishartDistribution.compute_message_to_parent

WishartDistribution.**compute_message_to_parent**(*parent*, *index*, *u_self*, **u_parents*)

bayespy.inference.vmp.nodes.wishart.WishartDistribution.compute_moments_and_cgf

WishartDistribution.**compute_moments_and_cgf**(*phi*, *mask=True*)

bayespy.inference.vmp.nodes.wishart.WishartDistribution.compute_phi_from_parents

WishartDistribution.**compute_phi_from_parents**(**u_parents*, *mask=True*)

bayespy.inference.vmp.nodes.wishart.WishartDistribution.plates_from_parent

WishartDistribution.**plates_from_parent**(*index*, *plates*)
Resolve the plate mapping from a parent.

Given the plates of a parent's moments, this method returns the plates that the moments has for this distribution.

bayespy.inference.vmp.nodes.wishart.WishartDistribution.plates_to_parent

WishartDistribution.**plates_to_parent**(*index*, *plates*)
Resolves the plate mapping to a parent.

Given the plates of the node's moments, this method returns the plates that the message to a parent has for the parent's distribution.

bayespy.inference.vmp.nodes.wishart.WishartDistribution.random

WishartDistribution.**random**(**params*, *plates=None*)
Draw a random sample from the distribution.

6.3.13 bayespy.inference.vmp.nodes.beta.BetaDistribution

class bayespy.inference.vmp.nodes.beta.**BetaDistribution**
Class for the VMP formulas of beta variables.

Although the realizations are scalars (probability p), the moments is a two-dimensional vector: [log(p), log(1-p)].

__init__()
Initialize self. See help(type(self)) for accurate signature.

Methods

compute_cgf_from_parents(u_alpha)	Compute $\mathrm{E}_{q(p)}[g(p)]$
compute_fixed_moments_and_f(p[, mask])	Compute the moments and $f(x)$ for a fixed value.
compute_gradient(g, u, phi)	Compute the moments and $g(\phi)$.
compute_logpdf(u, phi, g, f, ndims)	Compute E[log p(X)] given E[u], E[phi], E[g] and E[f].
compute_mask_to_parent(index, mask)	Maps the mask to the plates of a parent.
compute_message_to_parent(parent, index, ...)	Compute the message to a parent node.
compute_moments_and_cgf(phi[, mask])	Compute the moments and $g(\phi)$.
compute_phi_from_parents(u_alpha[, mask])	Compute the natural parameter vector given parent moments.
plates_from_parent(index, plates)	Resolve the plate mapping from a parent.
plates_to_parent(index, plates)	Resolves the plate mapping to a parent.
random(*phi[, plates])	Draw a random sample from the distribution.

bayespy.inference.vmp.nodes.beta.BetaDistribution.compute_cgf_from_parents

BetaDistribution.**compute_cgf_from_parents**(*u_alpha*)
Compute $\mathrm{E}_{q(p)}[g(p)]$

bayespy.inference.vmp.nodes.beta.BetaDistribution.compute_fixed_moments_and_f

BetaDistribution.**compute_fixed_moments_and_f**(*p*, *mask=True*)
Compute the moments and $f(x)$ for a fixed value.

bayespy.inference.vmp.nodes.beta.BetaDistribution.compute_gradient

BetaDistribution.**compute_gradient**(*g*, *u*, *phi*)
Compute the moments and $g(\phi)$.

psi(phi_1) - psi(sum_d phi_{1,d})

Standard gradient given the gradient with respect to the moments, that is, given the Riemannian gradient $\tilde{\nabla}$:

$$\nabla = \left[(\psi^{(1)}(\phi_1) - \psi^{(1)}(\sum_d \phi_{1,d})\nabla_1\right]$$

bayespy.inference.vmp.nodes.beta.BetaDistribution.compute_logpdf

BetaDistribution.**compute_logpdf**(*u*, *phi*, *g*, *f*, *ndims*)
Compute E[log p(X)] given E[u], E[phi], E[g] and E[f]. Does not sum over plates.

bayespy.inference.vmp.nodes.beta.BetaDistribution.compute_mask_to_parent

BetaDistribution.**compute_mask_to_parent**(*index*, *mask*)
Maps the mask to the plates of a parent.

bayespy.inference.vmp.nodes.beta.BetaDistribution.compute_message_to_parent

BetaDistribution.**compute_message_to_parent**(*parent*, *index*, *u_self*, *u_alpha*)
Compute the message to a parent node.

bayespy.inference.vmp.nodes.beta.BetaDistribution.compute_moments_and_cgf

BetaDistribution.**compute_moments_and_cgf**(*phi*, *mask=True*)
Compute the moments and $g(\phi)$.

bayespy.inference.vmp.nodes.beta.BetaDistribution.compute_phi_from_parents

BetaDistribution.**compute_phi_from_parents**(*u_alpha*, *mask=True*)
Compute the natural parameter vector given parent moments.

bayespy.inference.vmp.nodes.beta.BetaDistribution.plates_from_parent

BetaDistribution.**plates_from_parent**(*index*, *plates*)
Resolve the plate mapping from a parent.

Given the plates of a parent's moments, this method returns the plates that the moments has for this distribution.

bayespy.inference.vmp.nodes.beta.BetaDistribution.plates_to_parent

BetaDistribution.**plates_to_parent**(*index*, *plates*)

Resolves the plate mapping to a parent.

Given the plates of the node's moments, this method returns the plates that the message to a parent has for the parent's distribution.

bayespy.inference.vmp.nodes.beta.BetaDistribution.random

BetaDistribution.**random**(**phi*, *plates=None*)

Draw a random sample from the distribution.

6.3.14 bayespy.inference.vmp.nodes.dirichlet.DirichletDistribution

class bayespy.inference.vmp.nodes.dirichlet.**DirichletDistribution**

Class for the VMP formulas of Dirichlet variables.

__init__()

Initialize self. See help(type(self)) for accurate signature.

Methods

compute_cgf_from_parents(u_alpha)	Compute $\mathrm{E}_{q(p)}[g(p)]$
compute_fixed_moments_and_f(x[, mask])	Compute the moments and $f(x)$ for a fixed value.
compute_gradient(g, u, phi)	Compute the moments and $g(\phi)$.
compute_logpdf(u, phi, g, f, ndims)	Compute E[log p(X)] given E[u], E[phi], E[g] and E[f].
compute_mask_to_parent(index, mask)	Maps the mask to the plates of a parent.
compute_message_to_parent(parent, index, ...)	Compute the message to a parent node.
compute_moments_and_cgf(phi[, mask])	Compute the moments and $g(\phi)$.
compute_phi_from_parents(u_alpha[, mask])	Compute the natural parameter vector given parent moments.
plates_from_parent(index, plates)	Resolve the plate mapping from a parent.
plates_to_parent(index, plates)	Resolves the plate mapping to a parent.
random(*phi[, plates])	Draw a random sample from the distribution.

bayespy.inference.vmp.nodes.dirichlet.DirichletDistribution.compute_cgf_from_parents

DirichletDistribution.**compute_cgf_from_parents**(*u_alpha*)

Compute $\mathrm{E}_{q(p)}[g(p)]$

bayespy.inference.vmp.nodes.dirichlet.DirichletDistribution.compute_fixed_moments_and_f

DirichletDistribution.**compute_fixed_moments_and_f**(*x*, *mask=True*)

Compute the moments and $f(x)$ for a fixed value.

bayespy.inference.vmp.nodes.dirichlet.DirichletDistribution.compute_gradient

DirichletDistribution.**compute_gradient**(*g*, *u*, *phi*)

Compute the moments and $g(\phi)$.

psi(phi_1) - psi(sum_d phi_{1,d})

Standard gradient given the gradient with respect to the moments, that is, given the Riemannian gradient $\tilde{\nabla}$:

$$\nabla = \left[(\psi^{(1)}(\phi_1) - \psi^{(1)}(\sum_d \phi_{1,d}) \nabla_1 \right]$$

bayespy.inference.vmp.nodes.dirichlet.DirichletDistribution.compute_logpdf

DirichletDistribution.**compute_logpdf**(*u*, *phi*, *g*, *f*, *ndims*)
Compute E[log p(X)] given E[u], E[phi], E[g] and E[f]. Does not sum over plates.

bayespy.inference.vmp.nodes.dirichlet.DirichletDistribution.compute_mask_to_parent

DirichletDistribution.**compute_mask_to_parent**(*index*, *mask*)
Maps the mask to the plates of a parent.

bayespy.inference.vmp.nodes.dirichlet.DirichletDistribution.compute_message_to_parent

DirichletDistribution.**compute_message_to_parent**(*parent*, *index*, *u_self*, *u_alpha*)
Compute the message to a parent node.

bayespy.inference.vmp.nodes.dirichlet.DirichletDistribution.compute_moments_and_cgf

DirichletDistribution.**compute_moments_and_cgf**(*phi*, *mask=True*)
Compute the moments and $g(\phi)$.

$$\overline{\mathbf{u}}(\boldsymbol{\phi}) = \left[\psi(\phi_1) - \psi(\sum_d \phi_{1,d}) \right]$$
$$g_{\boldsymbol{\phi}}(\boldsymbol{\phi}) = TODO$$

bayespy.inference.vmp.nodes.dirichlet.DirichletDistribution.compute_phi_from_parents

DirichletDistribution.**compute_phi_from_parents**(*u_alpha*, *mask=True*)
Compute the natural parameter vector given parent moments.

bayespy.inference.vmp.nodes.dirichlet.DirichletDistribution.plates_from_parent

DirichletDistribution.**plates_from_parent**(*index*, *plates*)
Resolve the plate mapping from a parent.

Given the plates of a parent's moments, this method returns the plates that the moments has for this distribution.

bayespy.inference.vmp.nodes.dirichlet.DirichletDistribution.plates_to_parent

DirichletDistribution.**plates_to_parent**(*index*, *plates*)
Resolves the plate mapping to a parent.

Given the plates of the node's moments, this method returns the plates that the message to a parent has for the parent's distribution.

bayespy.inference.vmp.nodes.dirichlet.DirichletDistribution.random

DirichletDistribution.**random**(**phi*, *plates=None*)
Draw a random sample from the distribution.

6.3.15 bayespy.inference.vmp.nodes.bernoulli.BernoulliDistribution

class bayespy.inference.vmp.nodes.bernoulli.**BernoulliDistribution**
Class for the VMP formulas of Bernoulli variables.

__init__()

Methods

__init__()	
compute_cgf_from_parents(u_p)	Compute $\mathrm{E}_{q(p)}[g(p)]$
compute_fixed_moments_and_f(x[, mask])	Compute the moments and $f(x)$ for a fixed value.
compute_gradient(g, u, phi)	Compute the standard gradient with respect to the natural parameters.
compute_logpdf(u, phi, g, f, ndims)	Compute E[log p(X)] given E[u], E[phi], E[g] and E[f].
compute_mask_to_parent(index, mask)	Maps the mask to the plates of a parent.
compute_message_to_parent(parent, index, ...)	Compute the message to a parent node.
compute_moments_and_cgf(phi[, mask])	Compute the moments and $g(\phi)$.
compute_phi_from_parents(u_p[, mask])	Compute the natural parameter vector given parent moments.
plates_from_parent(index, plates)	Resolve the plate mapping from a parent.
plates_to_parent(index, plates)	Resolves the plate mapping to a parent.
random(*phi[, plates])	Draw a random sample from the distribution.

bayespy.inference.vmp.nodes.bernoulli.BernoulliDistribution.__init__

BernoulliDistribution.**__init__**()

bayespy.inference.vmp.nodes.bernoulli.BernoulliDistribution.compute_cgf_from_parents

BernoulliDistribution.**compute_cgf_from_parents**(*u_p*)
Compute $\mathrm{E}_{q(p)}[g(p)]$

bayespy.inference.vmp.nodes.bernoulli.BernoulliDistribution.compute_fixed_moments_and_f

BernoulliDistribution.**compute_fixed_moments_and_f**(*x*, *mask=True*)
Compute the moments and $f(x)$ for a fixed value.

bayespy.inference.vmp.nodes.bernoulli.BernoulliDistribution.compute_gradient

BernoulliDistribution.**compute_gradient**(*g*, *u*, *phi*)
Compute the standard gradient with respect to the natural parameters.

bayespy.inference.vmp.nodes.bernoulli.BernoulliDistribution.compute_logpdf

`BernoulliDistribution.compute_logpdf(u, phi, g, f, ndims)`

Compute E[log p(X)] given E[u], E[phi], E[g] and E[f]. Does not sum over plates.

bayespy.inference.vmp.nodes.bernoulli.BernoulliDistribution.compute_mask_to_parent

`BernoulliDistribution.compute_mask_to_parent(index, mask)`

Maps the mask to the plates of a parent.

bayespy.inference.vmp.nodes.bernoulli.BernoulliDistribution.compute_message_to_parent

`BernoulliDistribution.compute_message_to_parent(parent, index, u_self, u_p)`

Compute the message to a parent node.

bayespy.inference.vmp.nodes.bernoulli.BernoulliDistribution.compute_moments_and_cgf

`BernoulliDistribution.compute_moments_and_cgf(phi, mask=True)`

Compute the moments and $g(\phi)$.

bayespy.inference.vmp.nodes.bernoulli.BernoulliDistribution.compute_phi_from_parents

`BernoulliDistribution.compute_phi_from_parents(u_p, mask=True)`

Compute the natural parameter vector given parent moments.

bayespy.inference.vmp.nodes.bernoulli.BernoulliDistribution.plates_from_parent

`BernoulliDistribution.plates_from_parent(index, plates)`

Resolve the plate mapping from a parent.

Given the plates of a parent's moments, this method returns the plates that the moments has for this distribution.

bayespy.inference.vmp.nodes.bernoulli.BernoulliDistribution.plates_to_parent

`BernoulliDistribution.plates_to_parent(index, plates)`

Resolves the plate mapping to a parent.

Given the plates of the node's moments, this method returns the plates that the message to a parent has for the parent's distribution.

bayespy.inference.vmp.nodes.bernoulli.BernoulliDistribution.random

`BernoulliDistribution.random(*phi, plates=None)`

Draw a random sample from the distribution.

6.3.16 bayespy.inference.vmp.nodes.binomial.BinomialDistribution

class `bayespy.inference.vmp.nodes.binomial.`**`BinomialDistribution`**(*N*)

Class for the VMP formulas of binomial variables.

`__init__`(*N*)

Methods

`__init__`(N)	
`compute_cgf_from_parents`(u_p)	Compute $\mathrm{E}_{q(p)}[g(p)]$
`compute_fixed_moments_and_f`(x[, mask])	Compute the moments and $f(x)$ for a fixed value.
`compute_gradient`(g, u, phi)	Compute the standard gradient with respect to the natural parameters.
`compute_logpdf`(u, phi, g, f, ndims)	Compute E[log p(X)] given E[u], E[phi], E[g] and E[f].
`compute_mask_to_parent`(index, mask)	Maps the mask to the plates of a parent.
`compute_message_to_parent`(parent, index, ...)	Compute the message to a parent node.
`compute_moments_and_cgf`(phi[, mask])	Compute the moments and $g(\phi)$.
`compute_phi_from_parents`(u_p[, mask])	Compute the natural parameter vector given parent moments.
`plates_from_parent`(index, plates)	Resolve the plate mapping from a parent.
`plates_to_parent`(index, plates)	Resolves the plate mapping to a parent.
`random`(*phi[, plates])	Draw a random sample from the distribution.

bayespy.inference.vmp.nodes.binomial.BinomialDistribution.__init__

`BinomialDistribution.`**`__init__`**(*N*)

bayespy.inference.vmp.nodes.binomial.BinomialDistribution.compute_cgf_from_parents

`BinomialDistribution.`**`compute_cgf_from_parents`**(*u_p*)

Compute $\mathrm{E}_{q(p)}[g(p)]$

bayespy.inference.vmp.nodes.binomial.BinomialDistribution.compute_fixed_moments_and_f

`BinomialDistribution.`**`compute_fixed_moments_and_f`**(*x*, *mask=True*)

Compute the moments and $f(x)$ for a fixed value.

bayespy.inference.vmp.nodes.binomial.BinomialDistribution.compute_gradient

`BinomialDistribution.`**`compute_gradient`**(*g*, *u*, *phi*)

Compute the standard gradient with respect to the natural parameters.

bayespy.inference.vmp.nodes.binomial.BinomialDistribution.compute_logpdf

`BinomialDistribution.`**`compute_logpdf`**(*u*, *phi*, *g*, *f*, *ndims*)

Compute E[log p(X)] given E[u], E[phi], E[g] and E[f]. Does not sum over plates.

bayespy.inference.vmp.nodes.binomial.BinomialDistribution.compute_mask_to_parent

BinomialDistribution.**compute_mask_to_parent**(*index*, *mask*)
Maps the mask to the plates of a parent.

bayespy.inference.vmp.nodes.binomial.BinomialDistribution.compute_message_to_parent

BinomialDistribution.**compute_message_to_parent**(*parent*, *index*, *u_self*, *u_p*)
Compute the message to a parent node.

bayespy.inference.vmp.nodes.binomial.BinomialDistribution.compute_moments_and_cgf

BinomialDistribution.**compute_moments_and_cgf**(*phi*, *mask=True*)
Compute the moments and $g(\phi)$.

bayespy.inference.vmp.nodes.binomial.BinomialDistribution.compute_phi_from_parents

BinomialDistribution.**compute_phi_from_parents**(*u_p*, *mask=True*)
Compute the natural parameter vector given parent moments.

bayespy.inference.vmp.nodes.binomial.BinomialDistribution.plates_from_parent

BinomialDistribution.**plates_from_parent**(*index*, *plates*)
Resolve the plate mapping from a parent.

Given the plates of a parent's moments, this method returns the plates that the moments has for this distribution.

bayespy.inference.vmp.nodes.binomial.BinomialDistribution.plates_to_parent

BinomialDistribution.**plates_to_parent**(*index*, *plates*)
Resolves the plate mapping to a parent.

Given the plates of the node's moments, this method returns the plates that the message to a parent has for the parent's distribution.

bayespy.inference.vmp.nodes.binomial.BinomialDistribution.random

BinomialDistribution.**random**(**phi*, *plates=None*)
Draw a random sample from the distribution.

6.3.17 bayespy.inference.vmp.nodes.categorical.CategoricalDistribution

class bayespy.inference.vmp.nodes.categorical.**CategoricalDistribution**(*categories*)
Class for the VMP formulas of categorical variables.

__init__(*categories*)
Create VMP formula node for a categorical variable

categories is the total number of categories.

Methods

`__init__`(categories)	Create VMP formula node for a categorical variable
`compute_cgf_from_parents`(u_p)	Compute $\mathrm{E}_{q(p)}[g(p)]$
`compute_fixed_moments_and_f`(x[, mask])	Compute the moments and $f(x)$ for a fixed value.
`compute_gradient`(g, u, phi)	Compute the standard gradient with respect to the natural parameters.
`compute_logpdf`(u, phi, g, f, ndims)	Compute E[log p(X)] given E[u], E[phi], E[g] and E[f].
`compute_mask_to_parent`(index, mask)	Maps the mask to the plates of a parent.
`compute_message_to_parent`(parent, index, u, u_p)	Compute the message to a parent node.
`compute_moments_and_cgf`(phi[, mask])	Compute the moments and $g(\phi)$.
`compute_phi_from_parents`(u_p[, mask])	Compute the natural parameter vector given parent moments.
`plates_from_parent`(index, plates)	Resolve the plate mapping from a parent.
`plates_to_parent`(index, plates)	Resolves the plate mapping to a parent.
`random`(*phi[, plates])	Draw a random sample from the distribution.

bayespy.inference.vmp.nodes.categorical.CategoricalDistribution.__init__

`CategoricalDistribution.`**`__init__`**(*categories*)

Create VMP formula node for a categorical variable

categories is the total number of categories.

bayespy.inference.vmp.nodes.categorical.CategoricalDistribution.compute_cgf_from_parents

`CategoricalDistribution.`**`compute_cgf_from_parents`**(*u_p*)

Compute $\mathrm{E}_{q(p)}[g(p)]$

bayespy.inference.vmp.nodes.categorical.CategoricalDistribution.compute_fixed_moments_and_f

`CategoricalDistribution.`**`compute_fixed_moments_and_f`**(*x*, *mask=True*)

Compute the moments and $f(x)$ for a fixed value.

bayespy.inference.vmp.nodes.categorical.CategoricalDistribution.compute_gradient

`CategoricalDistribution.`**`compute_gradient`**(*g*, *u*, *phi*)

Compute the standard gradient with respect to the natural parameters.

bayespy.inference.vmp.nodes.categorical.CategoricalDistribution.compute_logpdf

`CategoricalDistribution.`**`compute_logpdf`**(*u*, *phi*, *g*, *f*, *ndims*)

Compute E[log p(X)] given E[u], E[phi], E[g] and E[f]. Does not sum over plates.

bayespy.inference.vmp.nodes.categorical.CategoricalDistribution.compute_mask_to_parent

`CategoricalDistribution.`**`compute_mask_to_parent`**(*index*, *mask*)

Maps the mask to the plates of a parent.

bayespy.inference.vmp.nodes.categorical.CategoricalDistribution.compute_message_to_parent

CategoricalDistribution.**compute_message_to_parent**(*parent*, *index*, *u*, *u_p*)
Compute the message to a parent node.

bayespy.inference.vmp.nodes.categorical.CategoricalDistribution.compute_moments_and_cgf

CategoricalDistribution.**compute_moments_and_cgf**(*phi*, *mask=True*)
Compute the moments and $g(\phi)$.

bayespy.inference.vmp.nodes.categorical.CategoricalDistribution.compute_phi_from_parents

CategoricalDistribution.**compute_phi_from_parents**(*u_p*, *mask=True*)
Compute the natural parameter vector given parent moments.

bayespy.inference.vmp.nodes.categorical.CategoricalDistribution.plates_from_parent

CategoricalDistribution.**plates_from_parent**(*index*, *plates*)
Resolve the plate mapping from a parent.

Given the plates of a parent's moments, this method returns the plates that the moments has for this distribution.

bayespy.inference.vmp.nodes.categorical.CategoricalDistribution.plates_to_parent

CategoricalDistribution.**plates_to_parent**(*index*, *plates*)
Resolves the plate mapping to a parent.

Given the plates of the node's moments, this method returns the plates that the message to a parent has for the parent's distribution.

bayespy.inference.vmp.nodes.categorical.CategoricalDistribution.random

CategoricalDistribution.**random**(**phi*, *plates=None*)
Draw a random sample from the distribution.

6.3.18 bayespy.inference.vmp.nodes.categorical_markov_chain.CategoricalMarkovChainDistribu

class bayespy.inference.vmp.nodes.categorical_markov_chain.**CategoricalMarkovChainDistribution**

Class for the VMP formulas of categorical Markov chain variables.

__init__(*categories*, *states*)
Create VMP formula node for a categorical variable

categories is the total number of categories. *states* is the length of the chain.

Continued on next page

Table 6.60 – continued from previous page

Methods

`__init__`(categories, states)	Create VMP formula node for a categorical variable
`compute_cgf_from_parents`(u_p0, u_P)	Compute $\mathrm{E}_{q(p)}[g(p)]$
`compute_fixed_moments_and_f`(x[, mask])	Compute the moments and $f(x)$ for a fixed value.
`compute_gradient`(g, u, phi)	Compute the standard gradient with respect to the natural parameters.
`compute_logpdf`(u, phi, g, f, ndims)	Compute E[log p(X)] given E[u], E[phi], E[g] and E[f].
`compute_mask_to_parent`(index, mask)	Maps the mask to the plates of a parent.
`compute_message_to_parent`(parent, index, u, ...)	Compute the message to a parent node.
`compute_moments_and_cgf`(phi[, mask])	Compute the moments and $g(\phi)$.
`compute_phi_from_parents`(u_p0, u_P[, mask])	Compute the natural parameter vector given parent moments.
`plates_from_parent`(index, plates)	Resolve the plate mapping from a parent.
`plates_to_parent`(index, plates)	Resolves the plate mapping to a parent.
`random`(*phi[, plates])	Draw a random sample from the distribution.

bayespy.inference.vmp.nodes.categorical_markov_chain.CategoricalMarkovChainDistribution.__init__

`CategoricalMarkovChainDistribution.`**`__init__`**(*categories*, *states*)

Create VMP formula node for a categorical variable

categories is the total number of categories. *states* is the length of the chain.

bayespy.inference.vmp.nodes.categorical_markov_chain.CategoricalMarkovChainDistribution.compute_cgf_fro

`CategoricalMarkovChainDistribution.`**`compute_cgf_from_parents`**(*u_p0*, *u_P*)

Compute $\mathrm{E}_{q(p)}[g(p)]$

bayespy.inference.vmp.nodes.categorical_markov_chain.CategoricalMarkovChainDistribution.compute_fixed_

`CategoricalMarkovChainDistribution.`**`compute_fixed_moments_and_f`**(*x*, *mask=True*)

Compute the moments and $f(x)$ for a fixed value.

bayespy.inference.vmp.nodes.categorical_markov_chain.CategoricalMarkovChainDistribution.compute_gradie

`CategoricalMarkovChainDistribution.`**`compute_gradient`**(*g*, *u*, *phi*)

Compute the standard gradient with respect to the natural parameters.

bayespy.inference.vmp.nodes.categorical_markov_chain.CategoricalMarkovChainDistribution.compute_logpd

`CategoricalMarkovChainDistribution.`**`compute_logpdf`**(*u*, *phi*, *g*, *f*, *ndims*)

Compute E[log p(X)] given E[u], E[phi], E[g] and E[f]. Does not sum over plates.

bayespy.inference.vmp.nodes.categorical_markov_chain.CategoricalMarkovChainDistribution.compute_mask_

`CategoricalMarkovChainDistribution.`**`compute_mask_to_parent`**(*index*, *mask*)

Maps the mask to the plates of a parent.

bayespy.inference.vmp.nodes.categorical_markov_chain.CategoricalMarkovChainDistribution.compute_messa

CategoricalMarkovChainDistribution.**compute_message_to_parent**(*parent*, *index*, *u*, *u_p0*, *u_P*)

Compute the message to a parent node.

bayespy.inference.vmp.nodes.categorical_markov_chain.CategoricalMarkovChainDistribution.compute_mome

CategoricalMarkovChainDistribution.**compute_moments_and_cgf**(*phi*, *mask=True*)

Compute the moments and $g(\phi)$.

bayespy.inference.vmp.nodes.categorical_markov_chain.CategoricalMarkovChainDistribution.compute_phi_fro

CategoricalMarkovChainDistribution.**compute_phi_from_parents**(*u_p0*, *u_P*, *mask=True*)

Compute the natural parameter vector given parent moments.

bayespy.inference.vmp.nodes.categorical_markov_chain.CategoricalMarkovChainDistribution.plates_from_par

CategoricalMarkovChainDistribution.**plates_from_parent**(*index*, *plates*)

Resolve the plate mapping from a parent.

Given the plates of a parent's moments, this method returns the plates that the moments has for this distribution.

bayespy.inference.vmp.nodes.categorical_markov_chain.CategoricalMarkovChainDistribution.plates_to_paren

CategoricalMarkovChainDistribution.**plates_to_parent**(*index*, *plates*)

Resolves the plate mapping to a parent.

Given the plates of the node's moments, this method returns the plates that the message to a parent has for the parent's distribution.

bayespy.inference.vmp.nodes.categorical_markov_chain.CategoricalMarkovChainDistribution.random

CategoricalMarkovChainDistribution.**random**(**phi*, *plates=None*)

Draw a random sample from the distribution.

6.3.19 bayespy.inference.vmp.nodes.multinomial.MultinomialDistribution

class bayespy.inference.vmp.nodes.multinomial.**MultinomialDistribution**(*trials*)

Class for the VMP formulas of multinomial variables.

__init__(*trials*)

Create VMP formula node for a multinomial variable

trials is the total number of trials.

Methods

`__init__`(trials)	Create VMP formula node for a multinomial variable
`compute_cgf_from_parents`(u_p)	Compute $\mathrm{E}_{q(p)}[g(p)]$
`compute_fixed_moments_and_f`(x[, mask])	Compute the moments and $f(x)$ for a fixed value.
`compute_gradient`(g, u, phi)	Compute the standard gradient with respect to the natural parameters.
`compute_logpdf`(u, phi, g, f, ndims)	Compute E[log p(X)] given E[u], E[phi], E[g] and E[f].
`compute_mask_to_parent`(index, mask)	Maps the mask to the plates of a parent.
`compute_message_to_parent`(parent, index, u, u_p)	Compute the message to a parent node.
`compute_moments_and_cgf`(phi[, mask])	Compute the moments and $g(\phi)$.
`compute_phi_from_parents`(u_p[, mask])	Compute the natural parameter vector given parent moments.
`plates_from_parent`(index, plates)	Resolve the plate mapping from a parent.
`plates_to_parent`(index, plates)	Resolves the plate mapping to a parent.
`random`(*phi)	Draw a random sample from the distribution.

bayespy.inference.vmp.nodes.multinomial.MultinomialDistribution.__init__

`MultinomialDistribution.`**`__init__`**(*trials*)

Create VMP formula node for a multinomial variable

trials is the total number of trials.

bayespy.inference.vmp.nodes.multinomial.MultinomialDistribution.compute_cgf_from_parents

`MultinomialDistribution.`**`compute_cgf_from_parents`**(*u_p*)

Compute $\mathrm{E}_{q(p)}[g(p)]$

bayespy.inference.vmp.nodes.multinomial.MultinomialDistribution.compute_fixed_moments_and_f

`MultinomialDistribution.`**`compute_fixed_moments_and_f`**(*x*, *mask=True*)

Compute the moments and $f(x)$ for a fixed value.

bayespy.inference.vmp.nodes.multinomial.MultinomialDistribution.compute_gradient

`MultinomialDistribution.`**`compute_gradient`**(*g*, *u*, *phi*)

Compute the standard gradient with respect to the natural parameters.

bayespy.inference.vmp.nodes.multinomial.MultinomialDistribution.compute_logpdf

`MultinomialDistribution.`**`compute_logpdf`**(*u*, *phi*, *g*, *f*, *ndims*)

Compute E[log p(X)] given E[u], E[phi], E[g] and E[f]. Does not sum over plates.

bayespy.inference.vmp.nodes.multinomial.MultinomialDistribution.compute_mask_to_parent

`MultinomialDistribution.`**`compute_mask_to_parent`**(*index*, *mask*)

Maps the mask to the plates of a parent.

bayespy.inference.vmp.nodes.multinomial.MultinomialDistribution.compute_message_to_parent

`MultinomialDistribution.`**`compute_message_to_parent`**(*parent*, *index*, *u*, *u_p*)

Compute the message to a parent node.

bayespy.inference.vmp.nodes.multinomial.MultinomialDistribution.compute_moments_and_cgf

MultinomialDistribution.**compute_moments_and_cgf**(*phi*, *mask=True*)
Compute the moments and $g(\phi)$.

bayespy.inference.vmp.nodes.multinomial.MultinomialDistribution.compute_phi_from_parents

MultinomialDistribution.**compute_phi_from_parents**(*u_p*, *mask=True*)
Compute the natural parameter vector given parent moments.

bayespy.inference.vmp.nodes.multinomial.MultinomialDistribution.plates_from_parent

MultinomialDistribution.**plates_from_parent**(*index*, *plates*)
Resolve the plate mapping from a parent.

Given the plates of a parent's moments, this method returns the plates that the moments has for this distribution.

bayespy.inference.vmp.nodes.multinomial.MultinomialDistribution.plates_to_parent

MultinomialDistribution.**plates_to_parent**(*index*, *plates*)
Resolves the plate mapping to a parent.

Given the plates of the node's moments, this method returns the plates that the message to a parent has for the parent's distribution.

bayespy.inference.vmp.nodes.multinomial.MultinomialDistribution.random

MultinomialDistribution.**random**(**phi*)
Draw a random sample from the distribution.

6.3.20 bayespy.inference.vmp.nodes.poisson.PoissonDistribution

class bayespy.inference.vmp.nodes.poisson.**PoissonDistribution**
Class for the VMP formulas of Poisson variables.

__init__()
Initialize self. See help(type(self)) for accurate signature.

Methods

compute_cgf_from_parents(u_lambda)	Compute $\mathrm{E}_{q(p)}[g(p)]$
compute_fixed_moments_and_f(x[, mask])	Compute the moments and $f(x)$ for a fixed value.
compute_gradient(g, u, phi)	Compute the standard gradient with respect to the natural parameters.
compute_logpdf(u, phi, g, f, ndims)	Compute E[log p(X)] given E[u], E[phi], E[g] and E[f].
compute_mask_to_parent(index, mask)	Maps the mask to the plates of a parent.
compute_message_to_parent(parent, index, u, ...)	Compute the message to a parent node.
compute_moments_and_cgf(phi[, mask])	Compute the moments and $g(\phi)$.
compute_phi_from_parents(u_lambda[, mask])	Compute the natural parameter vector given parent moments.

Continued on next page

Table 6.62 – continued from previous page

plates_from_parent(index, plates)	Resolve the plate mapping from a parent.
plates_to_parent(index, plates)	Resolves the plate mapping to a parent.
random(*phi)	Draw a random sample from the distribution.

bayespy.inference.vmp.nodes.poisson.PoissonDistribution.compute_cgf_from_parents

PoissonDistribution.**compute_cgf_from_parents**(*u_lambda*)
Compute $\mathrm{E}_{q(p)}[g(p)]$

bayespy.inference.vmp.nodes.poisson.PoissonDistribution.compute_fixed_moments_and_f

PoissonDistribution.**compute_fixed_moments_and_f**(*x*, *mask=True*)
Compute the moments and $f(x)$ for a fixed value.

bayespy.inference.vmp.nodes.poisson.PoissonDistribution.compute_gradient

PoissonDistribution.**compute_gradient**(*g*, *u*, *phi*)
Compute the standard gradient with respect to the natural parameters.

bayespy.inference.vmp.nodes.poisson.PoissonDistribution.compute_logpdf

PoissonDistribution.**compute_logpdf**(*u*, *phi*, *g*, *f*, *ndims*)
Compute E[log p(X)] given E[u], E[phi], E[g] and E[f]. Does not sum over plates.

bayespy.inference.vmp.nodes.poisson.PoissonDistribution.compute_mask_to_parent

PoissonDistribution.**compute_mask_to_parent**(*index*, *mask*)
Maps the mask to the plates of a parent.

bayespy.inference.vmp.nodes.poisson.PoissonDistribution.compute_message_to_parent

PoissonDistribution.**compute_message_to_parent**(*parent*, *index*, *u*, *u_lambda*)
Compute the message to a parent node.

bayespy.inference.vmp.nodes.poisson.PoissonDistribution.compute_moments_and_cgf

PoissonDistribution.**compute_moments_and_cgf**(*phi*, *mask=True*)
Compute the moments and $g(\phi)$.

bayespy.inference.vmp.nodes.poisson.PoissonDistribution.compute_phi_from_parents

PoissonDistribution.**compute_phi_from_parents**(*u_lambda*, *mask=True*)
Compute the natural parameter vector given parent moments.

bayespy.inference.vmp.nodes.poisson.PoissonDistribution.plates_from_parent

PoissonDistribution.**plates_from_parent**(*index*, *plates*)
Resolve the plate mapping from a parent.

Given the plates of a parent's moments, this method returns the plates that the moments has for this distribution.

bayespy.inference.vmp.nodes.poisson.PoissonDistribution.plates_to_parent

PoissonDistribution.**plates_to_parent**(*index*, *plates*)
Resolves the plate mapping to a parent.

Given the plates of the node's moments, this method returns the plates that the message to a parent has for the parent's distribution.

bayespy.inference.vmp.nodes.poisson.PoissonDistribution.random

PoissonDistribution.**random**(**phi*)
Draw a random sample from the distribution.

6.4 Utility functions

linalg	
random	Random variable generators.
optimize	
misc	

6.4.1 bayespy.utils.linalg

General numerical functions and methods.

Functions

block_banded_solve(A, B, y)	Invert symmetric, banded, positive-definite matrix.
chol(C)	
chol_inv(U)	
chol_logdet(U)	
chol_solve(U, b[, out, matrix])	
dot(*arrays)	Compute matrix-matrix product.
inner(*args[, ndim])	Compute inner product.
inv(A[, ndim])	General array inversion.
logdet_chol(U)	
logdet_cov(C)	
logdet_tri(R)	Logarithm of the absolute value of the determinant of a triangular matrix.
m_dot(A, b)	
mmdot(A, B[, ndim])	Compute matrix-matrix product.
mvdot(A, b[, ndim])	Compute matrix-vector product.

Continued on next page

Table 6.64 – continued from previous page

outer(A, B[, ndim])	Computes outer product over the last axes of A and B.
solve_triangular(U, B, **kwargs)	
tracedot(A, B)	Computes trace(A*B).
transpose(X[, ndim])	Transpose the matrix.

bayespy.utils.linalg.block_banded_solve

bayespy.utils.linalg.**block_banded_solve**(*A*, *B*, *y*)

Invert symmetric, banded, positive-definite matrix.

A contains the diagonal blocks.

B contains the superdiagonal blocks (their transposes are the subdiagonal blocks).

Shapes: A: (..., N, D, D) B: (..., N-1, D, D) y: (..., N, D)

The algorithm is basically LU decomposition.

Computes only the diagonal and super-diagonal blocks of the inverse. The true inverse is dense, in general.

Assume each block has the same size.

Return: * inverse blocks * solution to the system * log-determinant

bayespy.utils.linalg.chol

bayespy.utils.linalg.**chol**(*C*)

bayespy.utils.linalg.chol_inv

bayespy.utils.linalg.**chol_inv**(*U*)

bayespy.utils.linalg.chol_logdet

bayespy.utils.linalg.**chol_logdet**(*U*)

bayespy.utils.linalg.chol_solve

bayespy.utils.linalg.**chol_solve**(*U*, *b*, *out=None*, *matrix=False*)

bayespy.utils.linalg.dot

bayespy.utils.linalg.**dot**(**arrays*)

Compute matrix-matrix product.

You can give multiple arrays, the dot product is computed from left to right: A1*A2*A3*...*AN. The dot product is computed over the last two axes of each arrays. All other axes must be broadcastable.

bayespy.utils.linalg.inner

bayespy.utils.linalg.**inner**(**args*, *ndim=1*)
Compute inner product.

The number of arrays is arbitrary. The number of dimensions is arbitrary.

bayespy.utils.linalg.inv

bayespy.utils.linalg.**inv**(*A*, *ndim=1*)
General array inversion.

Supports broadcasting and inversion of multidimensional arrays. For instance, an array with shape (4,3,2,3,2) could mean that there are four (3*2) x (3*2) matrices to be inverted. This can be done by inv(A, ndim=2). For inverting scalars, ndim=0. For inverting matrices, ndim=1.

bayespy.utils.linalg.logdet_chol

bayespy.utils.linalg.**logdet_chol**(*U*)

bayespy.utils.linalg.logdet_cov

bayespy.utils.linalg.**logdet_cov**(*C*)

bayespy.utils.linalg.logdet_tri

bayespy.utils.linalg.**logdet_tri**(*R*)
Logarithm of the absolute value of the determinant of a triangular matrix.

bayespy.utils.linalg.m_dot

bayespy.utils.linalg.**m_dot**(*A*, *b*)

bayespy.utils.linalg.mmdot

bayespy.utils.linalg.**mmdot**(*A*, *B*, *ndim=1*)
Compute matrix-matrix product.

Applies broadcasting.

bayespy.utils.linalg.mvdot

bayespy.utils.linalg.**mvdot**(*A*, *b*, *ndim=1*)
Compute matrix-vector product.

Applies broadcasting.

bayespy.utils.linalg.outer

`bayespy.utils.linalg.outer(A, B, ndim=1)`

Computes outer product over the last axes of A and B.

The other axes are broadcasted. Thus, if A has shape (..., N) and B has shape (..., M), then the result has shape (..., N, M).

Using the argument *ndim* it is possible to change that how many axes trailing axes are used for the outer product. For instance, if ndim=3, A and B have shapes (...,N1,N2,N3) and (...,M1,M2,M3), the result has shape (...,N1,M1,N2,M2,N3,M3).

bayespy.utils.linalg.solve_triangular

`bayespy.utils.linalg.solve_triangular(U, B, **kwargs)`

bayespy.utils.linalg.tracedot

`bayespy.utils.linalg.tracedot(A, B)`

Computes trace(A*B).

bayespy.utils.linalg.transpose

`bayespy.utils.linalg.transpose(X, ndim=1)`

Transpose the matrix.

6.4.2 bayespy.utils.random

General functions random sampling and distributions.

Functions

alpha_beta_recursion(logp0, logP)	Compute alpha-beta recursion for Markov chain
bernoulli(p[, size])	Draw random samples from the Bernoulli distribution.
categorical(p[, size])	Draw random samples from a categorical distribution.
correlation(D)	Draw a random correlation matrix.
covariance(D[, size])	Draw a random covariance matrix.
dirichlet(alpha[, size])	Draw random samples from the Dirichlet distribution.
gamma_entropy(a, log_b, gammaln_a, psi_a, ...)	Entropy of $\mathcal{G}(a, b)$.
gamma_logpdf(bx, logx, a_logx, a_logb, gammaln_a)	Log-density of $\mathcal{G}(x\|a, b)$.
gaussian_entropy(logdet_V, D)	Compute the entropy of a Gaussian distribution.
gaussian_gamma_to_t(mu, Cov, a, b[, ndim])	Integrates gamma distribution to obtain parameters of t distribution
gaussian_logpdf(yVy, yVmu, muVmu, logdet_V, D)	Log-density of a Gaussian distribution.
intervals(N, length[, amount, gap])	Return random non-overlapping parts of a sequence.
invwishart_rand(nu, V)	
logodds_to_probability(x)	Solves p from log(p/(1-p))
mask(*shape[, p])	Return a boolean array of the given shape.
orth(D)	Draw random orthogonal matrix.
sphere([N])	Draw random points uniformly on a unit sphere.
svd(s)	Draw a random matrix given its singular values.

Continued on next page

Table 6.65 – continued from previous page

t_logpdf(z2, logdet_cov, nu, D)	
wishart_rand(nu, V)	Draw a random sample from the Wishart distribution.

bayespy.utils.random.alpha_beta_recursion

bayespy.utils.random.**alpha_beta_recursion**(*logp0*, *logP*)

Compute alpha-beta recursion for Markov chain

Initial state log-probabilities are in *p0* and state transition log-probabilities are in P. The probabilities do not need to be scaled to sum to one, but they are interpreted as below:

logp0 = log P(z_0) + log P(y_0|z_0) logP[...,n,:,:] = log P(z_{n+1}|z_n) + log P(y_{n+1}|z_{n+1})

bayespy.utils.random.bernoulli

bayespy.utils.random.**bernoulli**(*p*, *size=None*)

Draw random samples from the Bernoulli distribution.

bayespy.utils.random.categorical

bayespy.utils.random.**categorical**(*p*, *size=None*)

Draw random samples from a categorical distribution.

bayespy.utils.random.correlation

bayespy.utils.random.**correlation**(*D*)

Draw a random correlation matrix.

bayespy.utils.random.covariance

bayespy.utils.random.**covariance**(*D*, *size=()*)

Draw a random covariance matrix.

Draws from inverse-Wishart distribution. The distribution of each element is independent of the dimensionality of the matrix.

C ~ Inv-W(I, D)

Parameters **D** : int

Dimensionality of the covariance matrix.

bayespy.utils.random.dirichlet

bayespy.utils.random.**dirichlet**(*alpha*, *size=None*)

Draw random samples from the Dirichlet distribution.

bayespy.utils.random.gamma_entropy

bayespy.utils.random.**gamma_entropy**(*a*, *log_b*, *gammaln_a*, *psi_a*, *a_psi_a*)
Entropy of $\mathcal{G}(a, b)$.

If you want to get the gradient, just let each parameter be a gradient of that term.

Parameters **a** : ndarray

a

log_b : ndarray

$\log(b)$

gammaln_a : ndarray

$\log \Gamma(a)$

psi_a : ndarray

$\psi(a)$

a_psi_a : ndarray

$a\psi(a)$

bayespy.utils.random.gamma_logpdf

bayespy.utils.random.**gamma_logpdf**(*bx*, *logx*, *a_logx*, *a_logb*, *gammaln_a*)
Log-density of $\mathcal{G}(x|a, b)$.

If you want to get the gradient, just let each parameter be a gradient of that term.

Parameters **bx** : ndarray

bx

logx : ndarray

$\log(x)$

a_logx : ndarray

$a \log(x)$

a_logb : ndarray

$a \log(b)$

gammaln_a : ndarray

$\log \Gamma(a)$

bayespy.utils.random.gaussian_entropy

bayespy.utils.random.**gaussian_entropy**(*logdet_V*, *D*)
Compute the entropy of a Gaussian distribution.

If you want to get the gradient, just let each parameter be a gradient of that term.

Parameters **logdet_V** : ndarray or double

The log-determinant of the precision matrix.

D : int

The dimensionality of the distribution.

bayespy.utils.random.gaussian_gamma_to_t

`bayespy.utils.random.gaussian_gamma_to_t(mu, Cov, a, b, ndim=1)`

Integrates gamma distribution to obtain parameters of t distribution

bayespy.utils.random.gaussian_logpdf

`bayespy.utils.random.gaussian_logpdf(yVy, yVmu, muVmu, logdet_V, D)`

Log-density of a Gaussian distribution.

$\mathcal{G}(\mathbf{y}|\boldsymbol{\mu}, \mathbf{V}^{-1})$

Parameters **yVy** : ndarray or double

$\mathbf{y}^T\mathbf{V}\mathbf{y}$

yVmu : ndarray or double

$\mathbf{y}^T\mathbf{V}\boldsymbol{\mu}$

muVmu : ndarray or double

$\boldsymbol{\mu}^T\mathbf{V}\boldsymbol{\mu}$

logdet_V : ndarray or double

Log-determinant of the precision matrix, $\log|\mathbf{V}|$.

D : int

Dimensionality of the distribution.

bayespy.utils.random.intervals

`bayespy.utils.random.intervals(N, length, amount=1, gap=0)`

Return random non-overlapping parts of a sequence.

For instance, N=16, length=2 and amount=4: [0, **|1, 2|**, 3, 4, 5, **|6, 7|**, 8, 9, **|10, 11|**, **|12, 13|**, 14, 15] that is, [1,2,6,7,10,11,12,13]

However, the function returns only the indices of the beginning of the sequences, that is, in the example: [1,6,10,12]

bayespy.utils.random.invwishart_rand

`bayespy.utils.random.invwishart_rand(nu, V)`

bayespy.utils.random.logodds_to_probability

`bayespy.utils.random.logodds_to_probability(x)`

Solves p from log(p/(1-p))

bayespy.utils.random.mask

`bayespy.utils.random.mask(*shape, p=0.5)`
Return a boolean array of the given shape.

Parameters d0, d1, ..., dn : int

Shape of the output.

p : value in range [0,1]

A probability that the elements are *True*.

bayespy.utils.random.orth

`bayespy.utils.random.orth(D)`
Draw random orthogonal matrix.

bayespy.utils.random.sphere

`bayespy.utils.random.sphere(N=1)`
Draw random points uniformly on a unit sphere.

Returns (latitude,longitude) in degrees.

bayespy.utils.random.svd

`bayespy.utils.random.svd(s)`
Draw a random matrix given its singular values.

bayespy.utils.random.t_logpdf

`bayespy.utils.random.t_logpdf(z2, logdet_cov, nu, D)`

bayespy.utils.random.wishart_rand

`bayespy.utils.random.wishart_rand(nu, V)`
Draw a random sample from the Wishart distribution.

Parameters nu : int

6.4.3 bayespy.utils.optimize

Functions

check_gradient(f, x0[, verbose])	Simple wrapper for SciPy's gradient checker.
minimize(f, x0[, maxiter, verbose])	Simple wrapper for SciPy's optimize.

bayespy.utils.optimize.check_gradient

`bayespy.utils.optimize.check_gradient(f, x0, verbose=True)`

Simple wrapper for SciPy's gradient checker.

The given function must return a tuple: (value, gradient).

Returns relative

bayespy.utils.optimize.minimize

`bayespy.utils.optimize.minimize(f, x0, maxiter=None, verbose=False)`

Simple wrapper for SciPy's optimize.

The given function must return a tuple: (value, gradient).

6.4.4 bayespy.utils.misc

General numerical functions and methods.

Functions

T(X)	Transpose the matrix.
add_axes(X[, num, axis])	
add_leading_axes(x, n)	
add_trailing_axes(x, n)	
array_to_scalar(x)	
atleast_nd(X, d)	
axes_to_collapse(shape_x, shape_to)	
block_banded(D, B)	Construct a symmetric block-banded matrix.
broadcasted_shape(*shapes)	Computes the resulting broadcasted shape for a given set of shapes.
broadcasted_shape_from_arrays(*arrays)	Computes the resulting broadcasted shape for a given set of arrays.
broadcasting_multiplier(plates, *args)	Compute the plate multiplier for given shapes.
ceildiv(a, b)	Compute a divided by b and rounded up.
check_gradient(x0, f, df, eps)	
chol(C)	
chol_inv(U)	
chol_logdet(U)	
chol_solve(U, b)	
cholesky(K)	
composite_function(function_list)	Construct a function composition from a list of functions.
diag(X[, ndim])	Create a diagonal array given the diagonal elements.
diagonal(A)	
dist_haversine(c1, c2[, radius])	
first(L)	
gaussian_logpdf(y_invcov_y, y_invcov_mu, ...)	
get_diag(X[, ndim])	Get the diagonal of an array.
grid(x1, x2)	Returns meshgrid as a (M*N,2)-shape array.
identity(*shape)	
is_callable(f)	
is_numeric(a)	

Continued on next page

Table 6.67 – continued from previous page

is_shape_subset(sub_shape, full_shape)	
is_string(s)	
isinteger(x)	
kalman_filter(y, U, A, V, mu0, Cov0[, out])	Perform Kalman filtering to obtain filtered mean and covariance.
logdet_chol(U)	
logsumexp(X[, axis, keepdims])	Compute log(sum(exp(X)) in a numerically stable way
m_chol(C)	
m_chol_inv(U)	
m_chol_logdet(U)	
m_chol_solve(U, B[, out])	
m_digamma(a, d)	
m_dot(A, b)	
m_outer(A, B)	
m_solve_triangular(U, B, **kwargs)	
make_equal_length(*shapes)	Make tuples equal length.
make_equal_ndim(*arrays)	Add trailing unit axes so that arrays have equal ndim
mean(X[, axis, keepdims])	Compute the mean, ignoring NaNs.
moveaxis(A, axis_from, axis_to)	Move the axis *axis_from* to position *axis_to*.
multiply_shapes(*shapes)	Compute element-wise product of lists/tuples.
nans([size])	
nested_iterator(max_inds)	
remove_whitespace(s)	
repeat_to_shape(A, s)	
rmse(y1, y2[, axis])	
rts_smoother(mu, Cov, A, V[, removethis])	Perform Rauch-Tung-Striebel smoothing to obtain the posterior.
safe_indices(inds, shape)	Makes sure that indices are valid for given shape.
squeeze(X)	Remove leading axes that have unit length.
squeeze_to_dim(X, dim)	
sum_multiply(*args[, axis, sumaxis, keepdims])	
sum_multiply_to_plates(*arrays[, to_plates, ...])	Compute the product of the arguments and sum to the target shape.
sum_product(*args[, axes_to_keep, ...])	
sum_to_dim(A, dim)	Sum leading axes of A such that A has dim dimensions.
sum_to_shape(X, s)	Sum axes of the array such that the resulting shape is as given.
symm(X)	Make X symmetric.
tempfile([prefix, suffix])	
trues(shape)	
unique(l)	Remove duplicate items from a list while preserving order.
vb_optimize(x0, set_values, lowerbound[, ...])	
vb_optimize_nodes(*nodes)	
write_to_hdf5(group, data, name)	Writes the given array into the HDF5 file.
zipper_merge(*lists)	Combines lists by alternating elements from them.

bayespy.utils.misc.T

`bayespy.utils.misc.`**`T`**`(`*X*`)`

Transpose the matrix.

bayespy.utils.misc.add_axes

`bayespy.utils.misc.`**`add_axes`**`(`*X*, *num=1*, *axis=0*`)`

bayespy.utils.misc.add_leading_axes

`bayespy.utils.misc.`**`add_leading_axes`**(*x*, *n*)

bayespy.utils.misc.add_trailing_axes

`bayespy.utils.misc.`**`add_trailing_axes`**(*x*, *n*)

bayespy.utils.misc.array_to_scalar

`bayespy.utils.misc.`**`array_to_scalar`**(*x*)

bayespy.utils.misc.atleast_nd

`bayespy.utils.misc.`**`atleast_nd`**(*X*, *d*)

bayespy.utils.misc.axes_to_collapse

`bayespy.utils.misc.`**`axes_to_collapse`**(*shape_x*, *shape_to*)

bayespy.utils.misc.block_banded

`bayespy.utils.misc.`**`block_banded`**(*D*, *B*)

Construct a symmetric block-banded matrix.

D contains square diagonal blocks. *B* contains super-diagonal blocks.

The resulting matrix is:

D[0], B[0], 0, 0, ..., 0, 0, 0 B[0].T, D[1], B[1], 0, ..., 0, 0, 0 0, B[1].T, D[2], B[2], ..., ..., ..., ..., B[N-2].T, D[N-1], B[N-1] 0, 0, 0, 0, ..., 0, B[N-1].T, D[N]

bayespy.utils.misc.broadcasted_shape

`bayespy.utils.misc.`**`broadcasted_shape`**(**shapes*)

Computes the resulting broadcasted shape for a given set of shapes.

Uses the broadcasting rules of NumPy. Raises an exception if the shapes do not broadcast.

bayespy.utils.misc.broadcasted_shape_from_arrays

`bayespy.utils.misc.`**`broadcasted_shape_from_arrays`**(**arrays*)

Computes the resulting broadcasted shape for a given set of arrays.

Raises an exception if the shapes do not broadcast.

bayespy.utils.misc.broadcasting_multiplier

`bayespy.utils.misc.`**`broadcasting_multiplier`**(*plates*, **args*)

Compute the plate multiplier for given shapes.

The first shape is compared to all other shapes (using NumPy broadcasting rules). All the elements which are non-unit in the first shape but 1 in all other shapes are multiplied together.

This method is used, for instance, for computing a correction factor for messages to parents: If this node has non-unit plates that are unit plates in the parent, those plates are summed. However, if the message has unit axis for that plate, it should be first broadcasted to the plates of this node and then summed to the plates of the parent. In order to avoid this broadcasting and summing, it is more efficient to just multiply by the correct factor. This method computes that factor. The first argument is the full plate shape of this node (with respect to the parent). The other arguments are the shape of the message array and the plates of the parent (with respect to this node).

bayespy.utils.misc.ceildiv

`bayespy.utils.misc.`**`ceildiv`**(*a*, *b*)

Compute a divided by b and rounded up.

bayespy.utils.misc.check_gradient

`bayespy.utils.misc.`**`check_gradient`**(*x0*, *f*, *df*, *eps*)

bayespy.utils.misc.chol

`bayespy.utils.misc.`**`chol`**(*C*)

bayespy.utils.misc.chol_inv

`bayespy.utils.misc.`**`chol_inv`**(*U*)

bayespy.utils.misc.chol_logdet

`bayespy.utils.misc.`**`chol_logdet`**(*U*)

bayespy.utils.misc.chol_solve

`bayespy.utils.misc.`**`chol_solve`**(*U*, *b*)

bayespy.utils.misc.cholesky

`bayespy.utils.misc.`**`cholesky`**(*K*)

bayespy.utils.misc.composite_function

bayespy.utils.misc.**composite_function**(*function_list*)

Construct a function composition from a list of functions.

Given a list of functions [f,g,h], constructs a function $h \circ g \circ f$. That is, returns a function z, for which $z(x) = h(g(f(x)))$.

bayespy.utils.misc.diag

bayespy.utils.misc.**diag**(*X*, *ndim=1*)

Create a diagonal array given the diagonal elements.

The diagonal array can be multi-dimensional. By default, the last axis is transformed to two axes (diagonal matrix) but this can be changed using ndim keyword. For instance, an array with shape (K,L,M,N) can be transformed to a set of diagonal 4-D tensors with shape (K,L,M,N,M,N) by giving ndim=2. If ndim=3, the result has shape (K,L,M,N,L,M,N), and so on.

Diagonality means that for the resulting array Y holds: Y[...,i_1,i_2,..,i_ndim,j_1,j_2,..,j_ndim] is zero if i_n!=j_n for any n.

bayespy.utils.misc.diagonal

bayespy.utils.misc.**diagonal**(*A*)

bayespy.utils.misc.dist_haversine

bayespy.utils.misc.**dist_haversine**(*c1*, *c2*, *radius=6372795*)

bayespy.utils.misc.first

bayespy.utils.misc.**first**(*L*)

bayespy.utils.misc.gaussian_logpdf

bayespy.utils.misc.**gaussian_logpdf**(*y_invcov_y*, *y_invcov_mu*, *mu_invcov_mu*, *logdetcov*, *D*)

bayespy.utils.misc.get_diag

bayespy.utils.misc.**get_diag**(*X*, *ndim=1*)

Get the diagonal of an array.

If ndim>1, take the diagonal of the last 2*ndim axes.

bayespy.utils.misc.grid

bayespy.utils.misc.**grid**(*x1*, *x2*)

Returns meshgrid as a (M*N,2)-shape array.

bayespy.utils.misc.identity

bayespy.utils.misc.**identity**(**shape*)

bayespy.utils.misc.is_callable

bayespy.utils.misc.**is_callable**(*f*)

bayespy.utils.misc.is_numeric

bayespy.utils.misc.**is_numeric**(*a*)

bayespy.utils.misc.is_shape_subset

bayespy.utils.misc.**is_shape_subset**(*sub_shape*, *full_shape*)

bayespy.utils.misc.is_string

bayespy.utils.misc.**is_string**(*s*)

bayespy.utils.misc.isinteger

bayespy.utils.misc.**isinteger**(*x*)

bayespy.utils.misc.kalman_filter

bayespy.utils.misc.**kalman_filter**(*y*, *U*, *A*, *V*, *mu0*, *Cov0*, *out=None*)

Perform Kalman filtering to obtain filtered mean and covariance.

The parameters of the process may vary in time, thus they are given as iterators instead of fixed values.

Parameters **y** : (N,D) array

“Normalized” noisy observations of the states, that is, the observations multiplied by the precision matrix U (and possibly other transformation matrices).

U : (N,D,D) array or N-list of (D,D) arrays

Precision matrix (i.e., inverse covariance matrix) of the observation noise for each time instance.

A : (N-1,D,D) array or (N-1)-list of (D,D) arrays

Dynamic matrix for each time instance.

V : (N-1,D,D) array or (N-1)-list of (D,D) arrays

Covariance matrix of the innovation noise for each time instance.

Returns **mu** : array

Filtered mean of the states.

Cov : array

Filtered covariance of the states.

See also:

rts_smoother

bayespy.utils.misc.logdet_chol

bayespy.utils.misc.**logdet_chol**(*U*)

bayespy.utils.misc.logsumexp

bayespy.utils.misc.**logsumexp**(*X*, *axis=None*, *keepdims=False*)
Compute log(sum(exp(X)) in a numerically stable way

bayespy.utils.misc.m_chol

bayespy.utils.misc.**m_chol**(*C*)

bayespy.utils.misc.m_chol_inv

bayespy.utils.misc.**m_chol_inv**(*U*)

bayespy.utils.misc.m_chol_logdet

bayespy.utils.misc.**m_chol_logdet**(*U*)

bayespy.utils.misc.m_chol_solve

bayespy.utils.misc.**m_chol_solve**(*U*, *B*, *out=None*)

bayespy.utils.misc.m_digamma

bayespy.utils.misc.**m_digamma**(*a*, *d*)

bayespy.utils.misc.m_dot

bayespy.utils.misc.**m_dot**(*A*, *b*)

bayespy.utils.misc.m_outer

bayespy.utils.misc.**m_outer**(*A*, *B*)

bayespy.utils.misc.m_solve_triangular

bayespy.utils.misc.**m_solve_triangular**(*U*, *B*, ***kwargs*)

bayespy.utils.misc.make_equal_length

`bayespy.utils.misc.make_equal_length(*shapes)`

Make tuples equal length.

Add leading 1s to shorter tuples.

bayespy.utils.misc.make_equal_ndim

`bayespy.utils.misc.make_equal_ndim(*arrays)`

Add trailing unit axes so that arrays have equal ndim

bayespy.utils.misc.mean

`bayespy.utils.misc.mean(X, axis=None, keepdims=False)`

Compute the mean, ignoring NaNs.

bayespy.utils.misc.moveaxis

`bayespy.utils.misc.moveaxis(A, axis_from, axis_to)`

Move the axis *axis_from* to position *axis_to*.

bayespy.utils.misc.multiply_shapes

`bayespy.utils.misc.multiply_shapes(*shapes)`

Compute element-wise product of lists/tuples.

Shorter lists are concatenated with leading 1s in order to get lists with the same length.

bayespy.utils.misc.nans

`bayespy.utils.misc.nans(size=())`

bayespy.utils.misc.nested_iterator

`bayespy.utils.misc.nested_iterator(max_inds)`

bayespy.utils.misc.remove_whitespace

`bayespy.utils.misc.remove_whitespace(s)`

bayespy.utils.misc.repeat_to_shape

`bayespy.utils.misc.repeat_to_shape(A, s)`

bayespy.utils.misc.rmse

`bayespy.utils.misc.rmse(y1, y2, axis=None)`

bayespy.utils.misc.rts_smoother

`bayespy.utils.misc.rts_smoother(mu, Cov, A, V, removethis=None)`

Perform Rauch-Tung-Striebel smoothing to obtain the posterior.

The function returns the posterior mean and covariance of each state. The parameters of the process may vary in time, thus they are given as iterators instead of fixed values.

Parameters **mu** : (N,D) array

Mean of the states from Kalman filter.

Cov : (N,D,D) array

Covariance of the states from Kalman filter.

A : (N-1,D,D) array or (N-1)-list of (D,D) arrays

Dynamic matrix for each time instance.

V : (N-1,D,D) array or (N-1)-list of (D,D) arrays

Covariance matrix of the innovation noise for each time instance.

Returns **mu** : array

Posterior mean of the states.

Cov : array

Posterior covariance of the states.

See also:

`kalman_filter`

bayespy.utils.misc.safe_indices

`bayespy.utils.misc.safe_indices(inds, shape)`

Makes sure that indices are valid for given shape.

The shorter shape determines the length.

For instance,

```
>>> safe_indices( (3, 4, 5), (1, 6) )
(0, 5)
```

bayespy.utils.misc.squeeze

`bayespy.utils.misc.squeeze(X)`

Remove leading axes that have unit length.

For instance, a shape (1,1,4,1,3) will be reshaped to (4,1,3).

bayespy.utils.misc.squeeze_to_dim

`bayespy.utils.misc.squeeze_to_dim(X, dim)`

bayespy.utils.misc.sum_multiply

`bayespy.utils.misc.`**`sum_multiply`**(**args*, *axis=None*, *sumaxis=True*, *keepdims=False*)

bayespy.utils.misc.sum_multiply_to_plates

`bayespy.utils.misc.`**`sum_multiply_to_plates`**(**arrays*, *to_plates=()*, *from_plates=None*, *ndim=0*)

Compute the product of the arguments and sum to the target shape.

bayespy.utils.misc.sum_product

`bayespy.utils.misc.`**`sum_product`**(**args*, *axes_to_keep=None*, *axes_to_sum=None*, *keepdims=False*)

bayespy.utils.misc.sum_to_dim

`bayespy.utils.misc.`**`sum_to_dim`**(*A*, *dim*)

Sum leading axes of A such that A has dim dimensions.

bayespy.utils.misc.sum_to_shape

`bayespy.utils.misc.`**`sum_to_shape`**(*X*, *s*)

Sum axes of the array such that the resulting shape is as given.

Thus, the shape of the result will be s or an error is raised.

bayespy.utils.misc.symm

`bayespy.utils.misc.`**`symm`**(*X*)

Make X symmetric.

bayespy.utils.misc.tempfile

`bayespy.utils.misc.`**`tempfile`**(*prefix='‘*, *suffix='‘*)

bayespy.utils.misc.trues

`bayespy.utils.misc.`**`trues`**(*shape*)

bayespy.utils.misc.unique

`bayespy.utils.misc.`**`unique`**(*l*)

Remove duplicate items from a list while preserving order.

bayespy.utils.misc.vb_optimize

`bayespy.utils.misc.`**`vb_optimize`**(*x0*, *set_values*, *lowerbound*, *gradient=None*)

bayespy.utils.misc.vb_optimize_nodes

`bayespy.utils.misc.vb_optimize_nodes(*nodes)`

bayespy.utils.misc.write_to_hdf5

`bayespy.utils.misc.write_to_hdf5(group, data, name)`

Writes the given array into the HDF5 file.

bayespy.utils.misc.zipper_merge

`bayespy.utils.misc.zipper_merge(*lists)`

Combines lists by alternating elements from them.

Combining lists [1,2,3], ['a','b','c'] and [42,666,99] results in [1,'a',42,2,'b',666,3,'c',99]

The lists should have equal length or they are assumed to have the length of the shortest list.

This is known as alternating merge or zipper merge.

Classes

CholeskyDense(K)	
CholeskySparse(K)	
TestCase([methodName])	Simple base class for unit testing.

bayespy.utils.misc.CholeskyDense

class `bayespy.utils.misc.CholeskyDense(K)`

`__init__(K)`

Methods

__init__(K)
logdet()
solve(b)
trace_solve_gradient(dK)

bayespy.utils.misc.CholeskyDense.__init__

`CholeskyDense.__init__(K)`

bayespy.utils.misc.CholeskyDense.logdet

`CholeskyDense.logdet()`

bayespy.utils.misc.CholeskyDense.solve

CholeskyDense.**solve**(*b*)

bayespy.utils.misc.CholeskyDense.trace_solve_gradient

CholeskyDense.**trace_solve_gradient**(*dK*)

bayespy.utils.misc.CholeskySparse

class bayespy.utils.misc.**CholeskySparse**(*K*)

__init__(*K*)

Methods

__init__(K)	
logdet()	
solve(b)	
trace_solve_gradient(dK)	

bayespy.utils.misc.CholeskySparse.__init__

CholeskySparse.**__init__**(*K*)

bayespy.utils.misc.CholeskySparse.logdet

CholeskySparse.**logdet**()

bayespy.utils.misc.CholeskySparse.solve

CholeskySparse.**solve**(*b*)

bayespy.utils.misc.CholeskySparse.trace_solve_gradient

CholeskySparse.**trace_solve_gradient**(*dK*)

bayespy.utils.misc.TestCase

class bayespy.utils.misc.**TestCase**(*methodName='runTest'*)

Simple base class for unit testing.

Adds NumPy's features to Python's unittest.

__init__(*methodName='runTest'*)

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

Methods

`__init__`([methodName])	Create an instance of the class that will use the named test method when e
`addCleanup`(function, *args, **kwargs)	Add a function, with arguments, to be called when the test is completed.
`addTypeEqualityFunc`(typeobj, function)	Add a type specific assertEqual style function to compare a type.
`assertAllClose`(A, B[, msg, rtol, atol])	
`assertAlmostEqual`(first, second[, places, ...])	Fail if the two objects are unequal as determined by their difference round
`assertAlmostEquals`(*args, **kwargs)	
`assertArrayEqual`(A, B[, msg])	
`assertCountEqual`(first, second[, msg])	An unordered sequence comparison asserting that the same elements, rega
`assertDictContainsSubset`(subset, dictionary)	Checks whether dictionary is a superset of subset.
`assertDictEqual`(d1, d2[, msg])	
`assertEqual`(first, second[, msg])	Fail if the two objects are unequal as determined by the '==' operator.
`assertEquals`(*args, **kwargs)	
`assertFalse`(expr[, msg])	Check that the expression is false.
`assertGreater`(a, b[, msg])	Just like self.assertTrue(a > b), but with a nicer default message.
`assertGreaterEqual`(a, b[, msg])	Just like self.assertTrue(a >= b), but with a nicer default message.
`assertIn`(member, container[, msg])	Just like self.assertTrue(a in b), but with a nicer default message.
`assertIs`(expr1, expr2[, msg])	Just like self.assertTrue(a is b), but with a nicer default message.
`assertIsInstance`(obj, cls[, msg])	Same as self.assertTrue(isinstance(obj, cls)), with a nicer default message
`assertIsNone`(obj[, msg])	Same as self.assertTrue(obj is None), with a nicer default message.
`assertIsNot`(expr1, expr2[, msg])	Just like self.assertTrue(a is not b), but with a nicer default message.
`assertIsNotNone`(obj[, msg])	Included for symmetry with assertIsNone.
`assertLess`(a, b[, msg])	Just like self.assertTrue(a < b), but with a nicer default message.
`assertLessEqual`(a, b[, msg])	Just like self.assertTrue(a <= b), but with a nicer default message.
`assertListEqual`(list1, list2[, msg])	A list-specific equality assertion.
`assertLogs`([logger, level])	Fail unless a log message of level *level* or higher is emitted on *logger_nam*
`assertMessage`(M1, M2)	
`assertMessageToChild`(X, u)	
`assertMultiLineEqual`(first, second[, msg])	Assert that two multi-line strings are equal.
`assertNotAlmostEqual`(first, second[, ...])	Fail if the two objects are equal as determined by their difference rounded
`assertNotAlmostEquals`(*args, **kwargs)	
`assertNotEqual`(first, second[, msg])	Fail if the two objects are equal as determined by the '!=' operator.
`assertNotEquals`(*args, **kwargs)	
`assertNotIn`(member, container[, msg])	Just like self.assertTrue(a not in b), but with a nicer default message.
`assertNotIsInstance`(obj, cls[, msg])	Included for symmetry with assertIsInstance.
`assertNotRegex`(text, unexpected_regex[, msg])	Fail the test if the text matches the regular expression.
`assertRaises`(excClass[, callableObj])	Fail unless an exception of class excClass is raised by callableObj when in
`assertRaisesRegex`(expected_exception, ...[, ...])	Asserts that the message in a raised exception matches a regex.
`assertRaisesRegexp`(*args, **kwargs)	
`assertRegex`(text, expected_regex[, msg])	Fail the test unless the text matches the regular expression.
`assertRegexpMatches`(*args, **kwargs)	
`assertSequenceEqual`(seq1, seq2[, msg, seq_type])	An equality assertion for ordered sequences (like lists and tuples).
`assertSetEqual`(set1, set2[, msg])	A set-specific equality assertion.
`assertTrue`(expr[, msg])	Check that the expression is true.
`assertTupleEqual`(tuple1, tuple2[, msg])	A tuple-specific equality assertion.
`assertWarns`(expected_warning[, callable_obj])	Fail unless a warning of class warnClass is triggered by callable_obj when
`assertWarnsRegex`(expected_warning, ...[, ...])	Asserts that the message in a triggered warning matches a regexp.
`assert_`(*args, **kwargs)	
`countTestCases`()	
`debug`()	Run the test without collecting errors in a TestResult
`defaultTestResult`()	

Table 6.71 –

doCleanups()	Execute all cleanup functions.
fail([msg])	Fail immediately, with the given message.
failIf(*args, **kwargs)	
failIfAlmostEqual(*args, **kwargs)	
failIfEqual(*args, **kwargs)	
failUnless(*args, **kwargs)	
failUnlessAlmostEqual(*args, **kwargs)	
failUnlessEqual(*args, **kwargs)	
failUnlessRaises(*args, **kwargs)	
id()	
run([result])	
setUp()	Hook method for setting up the test fixture before exercising it.
setUpClass()	Hook method for setting up class fixture before running tests in the class.
shortDescription()	Returns a one-line description of the test, or None if no description has be
skipTest(reason)	Skip this test.
subTest([msg])	Return a context manager that will return the enclosed block of code in a
tearDown()	Hook method for deconstructing the test fixture after testing it.
tearDownClass()	Hook method for deconstructing the class fixture after running all tests in

bayespy.utils.misc.TestCase.__init__

TestCase.**__init__**(*methodName='runTest'*)

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

bayespy.utils.misc.TestCase.addCleanup

TestCase.**addCleanup**(*function*, **args*, ***kwargs*)

Add a function, with arguments, to be called when the test is completed. Functions added are called on a LIFO basis and are called after tearDown on test failure or success.

Cleanup items are called even if setUp fails (unlike tearDown).

bayespy.utils.misc.TestCase.addTypeEqualityFunc

TestCase.**addTypeEqualityFunc**(*typeobj*, *function*)

Add a type specific assertEqual style function to compare a type.

This method is for use by TestCase subclasses that need to register their own type equality functions to provide nicer error messages.

Args:

typeobj: The data type to call this function on when both values are of the same type in assertEqual().

function: The callable taking two arguments and an optional msg= argument that raises self.failureException with a useful error message when the two arguments are not equal.

bayespy.utils.misc.TestCase.assertAllClose

TestCase.**assertAllClose**(*A*, *B*, *msg='Arrays not almost equal'*, *rtol=0.0001*, *atol=0*)

bayespy.utils.misc.TestCase.assertAlmostEqual

`TestCase.assertAlmostEqual` (*first*, *second*, *places=None*, *msg=None*, *delta=None*)

Fail if the two objects are unequal as determined by their difference rounded to the given number of decimal places (default 7) and comparing to zero, or by comparing that the between the two objects is more than the given delta.

Note that decimal places (from zero) are usually not the same as significant digits (measured from the most signficant digit).

If the two objects compare equal then they will automatically compare almost equal.

bayespy.utils.misc.TestCase.assertAlmostEquals

`TestCase.assertAlmostEquals` (**args*, ***kwargs*)

bayespy.utils.misc.TestCase.assertArrayEqual

`TestCase.assertArrayEqual` (*A*, *B*, *msg='Arrays not equal'*)

bayespy.utils.misc.TestCase.assertCountEqual

`TestCase.assertCountEqual` (*first*, *second*, *msg=None*)

An unordered sequence comparison asserting that the same elements, regardless of order. If the same element occurs more than once, it verifies that the elements occur the same number of times.

self.assertEqual(Counter(list(first)), Counter(list(second)))

Example:

- [0, 1, 1] and [1, 0, 1] compare equal.
- [0, 0, 1] and [0, 1] compare unequal.

bayespy.utils.misc.TestCase.assertDictContainsSubset

`TestCase.assertDictContainsSubset` (*subset*, *dictionary*, *msg=None*)

Checks whether dictionary is a superset of subset.

bayespy.utils.misc.TestCase.assertDictEqual

`TestCase.assertDictEqual` (*d1*, *d2*, *msg=None*)

bayespy.utils.misc.TestCase.assertEqual

`TestCase.assertEqual` (*first*, *second*, *msg=None*)

Fail if the two objects are unequal as determined by the '==' operator.

bayespy.utils.misc.TestCase.assertEquals

`TestCase.`**`assertEquals`**(**args*, ***kwargs*)

bayespy.utils.misc.TestCase.assertFalse

`TestCase.`**`assertFalse`**(*expr*, *msg=None*)
Check that the expression is false.

bayespy.utils.misc.TestCase.assertGreater

`TestCase.`**`assertGreater`**(*a*, *b*, *msg=None*)
Just like self.assertTrue(a > b), but with a nicer default message.

bayespy.utils.misc.TestCase.assertGreaterEqual

`TestCase.`**`assertGreaterEqual`**(*a*, *b*, *msg=None*)
Just like self.assertTrue(a >= b), but with a nicer default message.

bayespy.utils.misc.TestCase.assertIn

`TestCase.`**`assertIn`**(*member*, *container*, *msg=None*)
Just like self.assertTrue(a in b), but with a nicer default message.

bayespy.utils.misc.TestCase.assertIs

`TestCase.`**`assertIs`**(*expr1*, *expr2*, *msg=None*)
Just like self.assertTrue(a is b), but with a nicer default message.

bayespy.utils.misc.TestCase.assertIsInstance

`TestCase.`**`assertIsInstance`**(*obj*, *cls*, *msg=None*)
Same as self.assertTrue(isinstance(obj, cls)), with a nicer default message.

bayespy.utils.misc.TestCase.assertIsNone

`TestCase.`**`assertIsNone`**(*obj*, *msg=None*)
Same as self.assertTrue(obj is None), with a nicer default message.

bayespy.utils.misc.TestCase.assertIsNot

`TestCase.`**`assertIsNot`**(*expr1*, *expr2*, *msg=None*)
Just like self.assertTrue(a is not b), but with a nicer default message.

bayespy.utils.misc.TestCase.assertIsNotNone

`TestCase.assertIsNotNone(obj, msg=None)`
Included for symmetry with assertIsNone.

bayespy.utils.misc.TestCase.assertLess

`TestCase.assertLess(a, b, msg=None)`
Just like self.assertTrue(a < b), but with a nicer default message.

bayespy.utils.misc.TestCase.assertLessEqual

`TestCase.assertLessEqual(a, b, msg=None)`
Just like self.assertTrue(a <= b), but with a nicer default message.

bayespy.utils.misc.TestCase.assertListEqual

`TestCase.assertListEqual(list1, list2, msg=None)`
A list-specific equality assertion.

Args: list1: The first list to compare. list2: The second list to compare. msg: Optional message to use on failure instead of a list of

differences.

bayespy.utils.misc.TestCase.assertLogs

`TestCase.assertLogs(logger=None, level=None)`
Fail unless a log message of level *level* or higher is emitted on *logger_name* or its children. If omitted, *level* defaults to INFO and *logger* defaults to the root logger.

This method must be used as a context manager, and will yield a recording object with two attributes: *output* and *records*. At the end of the context manager, the *output* attribute will be a list of the matching formatted log messages and the *records* attribute will be a list of the corresponding LogRecord objects.

Example:

```
with self.assertLogs('foo', level='INFO') as cm:
    logging.getLogger('foo').info('first message')
    logging.getLogger('foo.bar').error('second message')
self.assertEqual(cm.output, ['INFO:foo:first message',
                             'ERROR:foo.bar:second message'])
```

bayespy.utils.misc.TestCase.assertMessage

`TestCase.assertMessage(M1, M2)`

bayespy.utils.misc.TestCase.assertMessageToChild

`TestCase.assertMessageToChild(X, u)`

bayespy.utils.misc.TestCase.assertMultiLineEqual

`TestCase.assertMultiLineEqual` (*first*, *second*, *msg=None*)
Assert that two multi-line strings are equal.

bayespy.utils.misc.TestCase.assertNotAlmostEqual

`TestCase.assertNotAlmostEqual` (*first*, *second*, *places=None*, *msg=None*, *delta=None*)
Fail if the two objects are equal as determined by their difference rounded to the given number of decimal places (default 7) and comparing to zero, or by comparing that the between the two objects is less than the given delta.

Note that decimal places (from zero) are usually not the same as significant digits (measured from the most signficant digit).

Objects that are equal automatically fail.

bayespy.utils.misc.TestCase.assertNotAlmostEquals

`TestCase.assertNotAlmostEquals` (**args*, ***kwargs*)

bayespy.utils.misc.TestCase.assertNotEqual

`TestCase.assertNotEqual` (*first*, *second*, *msg=None*)
Fail if the two objects are equal as determined by the '!=' operator.

bayespy.utils.misc.TestCase.assertNotEquals

`TestCase.assertNotEquals` (**args*, ***kwargs*)

bayespy.utils.misc.TestCase.assertNotIn

`TestCase.assertNotIn` (*member*, *container*, *msg=None*)
Just like self.assertTrue(a not in b), but with a nicer default message.

bayespy.utils.misc.TestCase.assertNotIsInstance

`TestCase.assertNotIsInstance` (*obj*, *cls*, *msg=None*)
Included for symmetry with assertIsInstance.

bayespy.utils.misc.TestCase.assertNotRegex

`TestCase.assertNotRegex` (*text*, *unexpected_regex*, *msg=None*)
Fail the test if the text matches the regular expression.

bayespy.utils.misc.TestCase.assertRaises

TestCase.**assertRaises**(*excClass*, *callableObj=None*, **args*, ***kwargs*)

Fail unless an exception of class excClass is raised by callableObj when invoked with arguments args and keyword arguments kwargs. If a different type of exception is raised, it will not be caught, and the test case will be deemed to have suffered an error, exactly as for an unexpected exception.

If called with callableObj omitted or None, will return a context object used like this:

```
with self.assertRaises(SomeException):
    do_something()
```

An optional keyword argument 'msg' can be provided when assertRaises is used as a context object.

The context manager keeps a reference to the exception as the 'exception' attribute. This allows you to inspect the exception after the assertion:

```
with self.assertRaises(SomeException) as cm:
    do_something()
the_exception = cm.exception
self.assertEqual(the_exception.error_code, 3)
```

bayespy.utils.misc.TestCase.assertRaisesRegex

TestCase.**assertRaisesRegex**(*expected_exception*, *expected_regex*, *callable_obj=None*, **args*, ***kwargs*)

Asserts that the message in a raised exception matches a regex.

Args: expected_exception: Exception class expected to be raised. expected_regex: Regex (re pattern object or string) expected

to be found in error message.

callable_obj: Function to be called. msg: Optional message used in case of failure. Can only be used

when assertRaisesRegex is used as a context manager.

args: Extra args. kwargs: Extra kwargs.

bayespy.utils.misc.TestCase.assertRaisesRegexp

TestCase.**assertRaisesRegexp**(**args*, ***kwargs*)

bayespy.utils.misc.TestCase.assertRegex

TestCase.**assertRegex**(*text*, *expected_regex*, *msg=None*)

Fail the test unless the text matches the regular expression.

bayespy.utils.misc.TestCase.assertRegexpMatches

TestCase.**assertRegexpMatches**(**args*, ***kwargs*)

bayespy.utils.misc.TestCase.assertSequenceEqual

TestCase.**assertSequenceEqual**(*seq1*, *seq2*, *msg=None*, *seq_type=None*)
An equality assertion for ordered sequences (like lists and tuples).

For the purposes of this function, a valid ordered sequence type is one which can be indexed, has a length, and has an equality operator.

Args: seq1: The first sequence to compare. seq2: The second sequence to compare. seq_type: The expected datatype of the sequences, or None if no

datatype should be enforced.

msg: Optional message to use on failure instead of a list of differences.

bayespy.utils.misc.TestCase.assertSetEqual

TestCase.**assertSetEqual**(*set1*, *set2*, *msg=None*)
A set-specific equality assertion.

Args: set1: The first set to compare. set2: The second set to compare. msg: Optional message to use on failure instead of a list of

differences.

assertSetEqual uses ducktyping to support different types of sets, and is optimized for sets specifically (parameters must support a difference method).

bayespy.utils.misc.TestCase.assertTrue

TestCase.**assertTrue**(*expr*, *msg=None*)
Check that the expression is true.

bayespy.utils.misc.TestCase.assertTupleEqual

TestCase.**assertTupleEqual**(*tuple1*, *tuple2*, *msg=None*)
A tuple-specific equality assertion.

Args: tuple1: The first tuple to compare. tuple2: The second tuple to compare. msg: Optional message to use on failure instead of a list of

differences.

bayespy.utils.misc.TestCase.assertWarns

TestCase.**assertWarns**(*expected_warning*, *callable_obj=None*, **args*, ***kwargs*)
Fail unless a warning of class warnClass is triggered by callable_obj when invoked with arguments args and keyword arguments kwargs. If a different type of warning is triggered, it will not be handled: depending on the other warning filtering rules in effect, it might be silenced, printed out, or raised as an exception.

If called with callable_obj omitted or None, will return a context object used like this:

```
with self.assertWarns(SomeWarning):
    do_something()
```

An optional keyword argument 'msg' can be provided when assertWarns is used as a context object.

The context manager keeps a reference to the first matching warning as the 'warning' attribute; similarly, the 'filename' and 'lineno' attributes give you information about the line of Python code from which the warning was triggered. This allows you to inspect the warning after the assertion:

```
with self.assertWarns(SomeWarning) as cm:
    do_something()
the_warning = cm.warning
self.assertEqual(the_warning.some_attribute, 147)
```

bayespy.utils.misc.TestCase.assertWarnsRegex

TestCase.**assertWarnsRegex**(*expected_warning*, *expected_regex*, *callable_obj=None*, **args*, ***kwargs*)

Asserts that the message in a triggered warning matches a regexp. Basic functioning is similar to assertWarns() with the addition that only warnings whose messages also match the regular expression are considered successful matches.

Args: expected_warning: Warning class expected to be triggered. expected_regex: Regex (re pattern object or string) expected

to be found in error message.

callable_obj: Function to be called. msg: Optional message used in case of failure. Can only be used

when assertWarnsRegex is used as a context manager.

args: Extra args. kwargs: Extra kwargs.

bayespy.utils.misc.TestCase.assert

TestCase.**assert_**(**args*, ***kwargs*)

bayespy.utils.misc.TestCase.countTestCases

TestCase.**countTestCases**()

bayespy.utils.misc.TestCase.debug

TestCase.**debug**()

Run the test without collecting errors in a TestResult

bayespy.utils.misc.TestCase.defaultTestResult

TestCase.**defaultTestResult**()

bayespy.utils.misc.TestCase.doCleanups

TestCase.**doCleanups**()

Execute all cleanup functions. Normally called for you after tearDown.

bayespy.utils.misc.TestCase.fail

`TestCase.fail(msg=None)`
Fail immediately, with the given message.

bayespy.utils.misc.TestCase.failIf

`TestCase.failIf(*args, **kwargs)`

bayespy.utils.misc.TestCase.failIfAlmostEqual

`TestCase.failIfAlmostEqual(*args, **kwargs)`

bayespy.utils.misc.TestCase.failIfEqual

`TestCase.failIfEqual(*args, **kwargs)`

bayespy.utils.misc.TestCase.failUnless

`TestCase.failUnless(*args, **kwargs)`

bayespy.utils.misc.TestCase.failUnlessAlmostEqual

`TestCase.failUnlessAlmostEqual(*args, **kwargs)`

bayespy.utils.misc.TestCase.failUnlessEqual

`TestCase.failUnlessEqual(*args, **kwargs)`

bayespy.utils.misc.TestCase.failUnlessRaises

`TestCase.failUnlessRaises(*args, **kwargs)`

bayespy.utils.misc.TestCase.id

`TestCase.id()`

bayespy.utils.misc.TestCase.run

`TestCase.run(result=None)`

bayespy.utils.misc.TestCase.setUp

`TestCase.setUp()`
Hook method for setting up the test fixture before exercising it.

bayespy.utils.misc.TestCase.setUpClass

`TestCase.setUpClass()`
Hook method for setting up class fixture before running tests in the class.

bayespy.utils.misc.TestCase.shortDescription

`TestCase.shortDescription()`
Returns a one-line description of the test, or None if no description has been provided.

The default implementation of this method returns the first line of the specified test method's docstring.

bayespy.utils.misc.TestCase.skipTest

`TestCase.skipTest(reason)`
Skip this test.

bayespy.utils.misc.TestCase.subTest

`TestCase.subTest(msg=None, **params)`
Return a context manager that will return the enclosed block of code in a subtest identified by the optional message and keyword parameters. A failure in the subtest marks the test case as failed but resumes execution at the end of the enclosed block, allowing further test code to be executed.

bayespy.utils.misc.TestCase.tearDown

`TestCase.tearDown()`
Hook method for deconstructing the test fixture after testing it.

bayespy.utils.misc.TestCase.tearDownClass

`TestCase.tearDownClass()`
Hook method for deconstructing the class fixture after running all tests in the class.

Attributes

`longMessage`
`maxDiff`

bayespy.utils.misc.TestCase.longMessage

`TestCase.longMessage = True`

bayespy.utils.misc.TestCase.maxDiff

`TestCase.maxDiff = 640`

Bibliography

[1] James Hensman, Magnus Rattray, and Neil D. Lawrence. Fast variational inference in the conjugate exponential family. In *Advances in Neural Information Processing Systems 25*. 2012.

[2] Matthew D. Hoffman, David M. Blei, Chong Wang, and John Paisley. Stochastic variational inference. *Journal of Machine Learning Research*, 14:1303–47, 2013.

[3] Antti Honkela, Tapani Raiko, Mikael Kuusela, Matti Tornio, and Juha Karhunen. Approximate Riemannian conjugate gradient learning for fixed-form variational Bayes. *Journal of Machine Learning Research*, 11:3235–3268, 2010.

[4] Antti Honkela, Harri Valpola, and Juha Karhunen. Accelerating cyclic update algorithms for parameter estimation by pattern searches. *Neural Processing Letters*, 17(2):191–203, 2003.

[5] K. Katahira, K. Watanabe, and M. Okada. Deterministic annealing variant of variational Bayes method. *Journal of Physics: Conference Series*, 2008.

[6] Jaakko Luttinen. Fast variational Bayesian linear state-space model. In Hendrik Blockeel, Kristian Kersting, Siegfried Nijssen, and Filip Železný, editors, *Machine Learning and Knowledge Discovery in Databases*, volume 8188 of Lecture Notes in Computer Science, pages 305–320. Springer, 2013.

[7] Jaakko Luttinen and Alexander Ilin. Transformations in variational Bayesian factor analysis to speed up learning. *Neurocomputing*, 73:1093–1102, 2010.

[8] Jaakko Luttinen, Tapani Raiko, and Alexander Ilin. Linear state-space model with time-varying dynamics. In Toon Calders, Floriana Esposito, Eyke Hüllermeier, and Rosa Meo, editors, *Machine Learning and Knowledge Discovery in Databases*, volume 8725 of Lecture Notes in Computer Science, pages 338–353. Springer, 2014.

PYTHON MODULE INDEX

b

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

Z