
BayesPy Documentation

Release 0.1

Jaakko Luttinen

July 22, 2014

CONTENTS

1	Introduction	1
1.1	Project information	1
1.2	Similar projects	2
2	User guide	3
2.1	Installation	3
2.2	Quick start guide	5
2.3	Constructing the model	7
2.4	Performing inference	14
2.5	Examining the results	19
3	Examples	25
3.1	Regression	25
3.2	Gaussian mixture model	27
3.3	Bernoulli mixture model	30
3.4	Hidden Markov model	35
3.5	Principal component analysis	42
3.6	Linear state-space model	45
4	Developer guide	55
4.1	Variational message passing	55
4.2	Implementing nodes	56
5	User API	57
5.1	bayespy.nodes	57
5.2	bayespy.inference	146
5.3	bayespy.plot	156
6	Developer API	163
6.1	Developer nodes	163
6.2	Moments	185
6.3	Distributions	201
6.4	Utility functions	239
	Bibliography	271
	Python Module Index	273
	Index	275

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.

It is recommended to use the latest version from the GitHub master branch. The version in PyPI is quite outdated.

1.1 Project information

Copyright (C) 2011-2014 Jaakko Luttinen, Aalto University

BayesPy including the documentation is licensed under Version 3.0 of the GNU General Public License. See LICENSE file for a text of the license or visit <http://www.gnu.org/copyleft/gpl.html>.

- 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.probplog.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.

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 requirements

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

- NumPy ($\geq 1.8.0$),
- SciPy ($\geq 0.13.0$)
- matplotlib (≥ 1.2)
- h5py

Ideally, a manual installation of these dependencies is not required and you can skip to the next section “Installing BayesPy”. However, there are several reasons why the installation of BayesPy as described in the next section won’t work because of your system. Thus, this section tries to give as detailed and robust a method of setting up your system such that the installation of BayesPy should work.

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. If your system has an older version of any of the packages (NumPy, SciPy or matplotlib) or it does not provide the packages for Python 3, you may set up a virtual environment and install the latest versions there. To create and activate a new virtual environment, run

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

If you have relevant system libraries installed (C compiler, Python development files, BLAS/LAPACK etc.), you may be able to install the Python packages from PyPI. 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/upgrade from PyPI should work:

```
pip install distribute --upgrade
pip install numpy --upgrade
pip install scipy --upgrade
pip install matplotlib --upgrade
pip install h5py
```

Note that Matplotlib requires a quite recent version of Distribute ($\geq 0.6.28$). If you have problems installing any of these packages, refer to the manual of that package.

2.1.2 Installing BayesPy

If the system has been properly set up and the virtual environment is activated (optional), 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
```

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
```

2.1.3 Compiling documentation

This documentation can be found at <http://bayespy.org/>. The documentation source files are 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 sphinxcontrib-tikz sphinxcontrib-bayesnet sphinxcontrib-bibtex
```

In addition, the `numpydoc` extension for Sphinx is required. However, the latest stable release (0.4) does not support Python 3, thus one needs to install the development version:

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

In order to visualize graphical models in HTML, you need to have `pnmcrop`. On Ubuntu, it can be installed as

```
sudo apt-get install netpbm
```

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:

$$p(\mathbf{y}|\mu, \tau) = \prod_{n=0}^9 \mathcal{N}(y_n|\mu, \tau)$$

$$p(\mu) = \mathcal{N}(\mu|0, 10^{-6})$$

$$p(\tau) = \mathcal{G}(\tau|10^{-6}, 10^{-6})$$

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

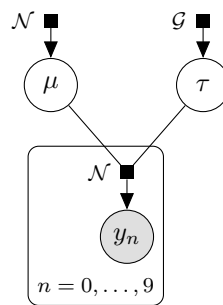


Figure 2.1: Directed factor graph of the example model.

constructed in BayesPy as follows:

```
>>> 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 $n = 0, \dots, 9$.

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):

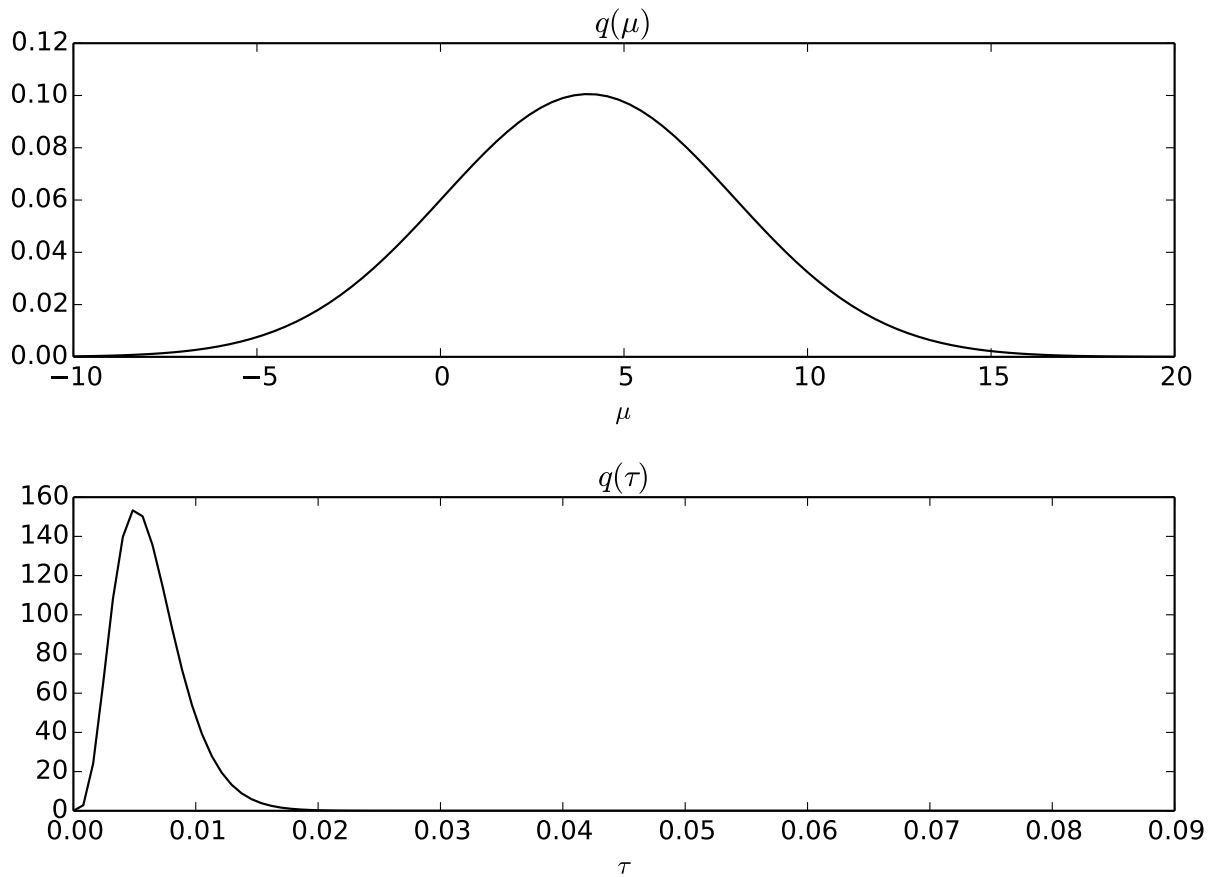
```
>>> Q.update(repeat=20)
Iteration 1: loglike=-6.020956e+01 (... seconds)
Iteration 2: loglike=-5.820527e+01 (... seconds)
Iteration 3: loglike=-5.820290e+01 (... seconds)
Iteration 4: loglike=-5.820288e+01 (... seconds)
Converged at iteration 4.
```

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.

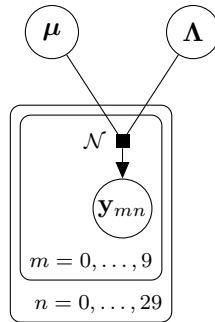
2.3.3 Using plate notation

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

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

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)
```

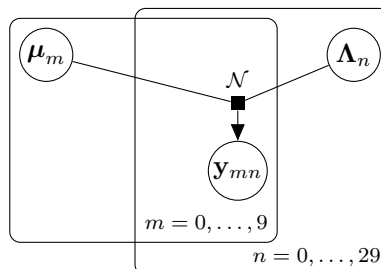
Note that this indexing is for the plates only, not for the random variable dimensions.

Sharing and broadcasting plates

Instead of having a common mean and precision matrix for all 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 :

$$y_{mn} \sim \mathcal{N}(\mu_m, \Lambda_n), \quad m = 0, \dots, 9, \quad n = 0, \dots, 29.$$

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 understand 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):

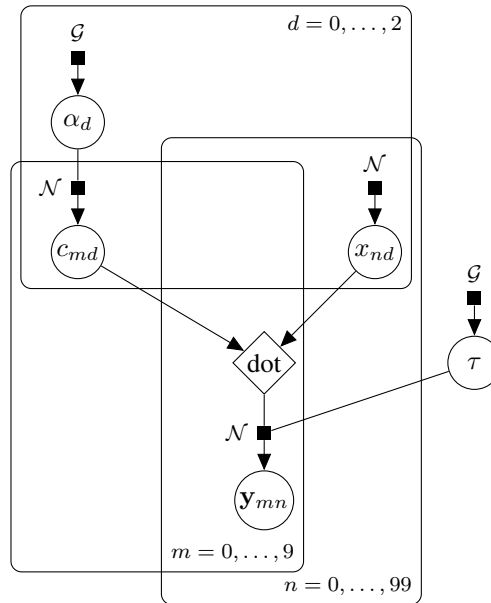
$$\mathbf{Y} = \mathbf{C}\mathbf{X}^T + \text{noise}$$

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 α of the loading matrix:

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

The prior for the loading matrix 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 α , $(D,)$, 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 is 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 \mathbf{C} provides the ARD parameters α 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 (\mathbf{X} , \mathbf{C} and α) 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 be 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:

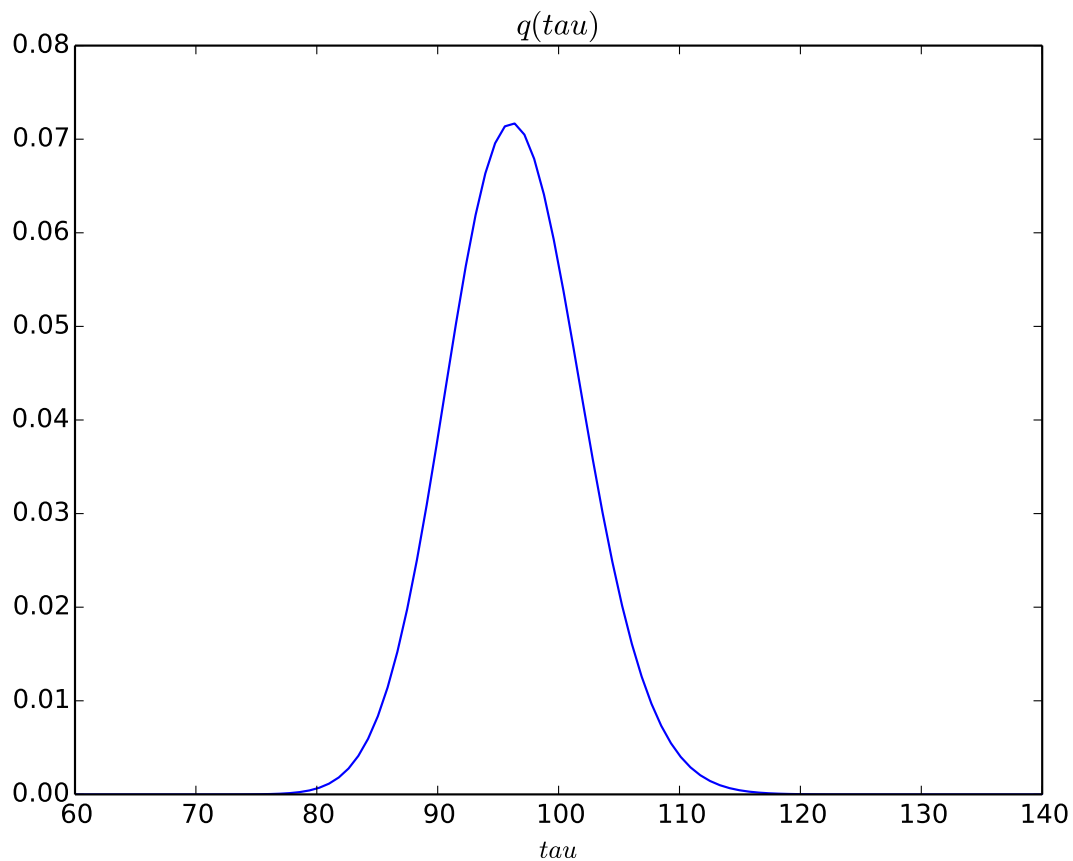
```
>>> bpplt.pyplot.ion()
```

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

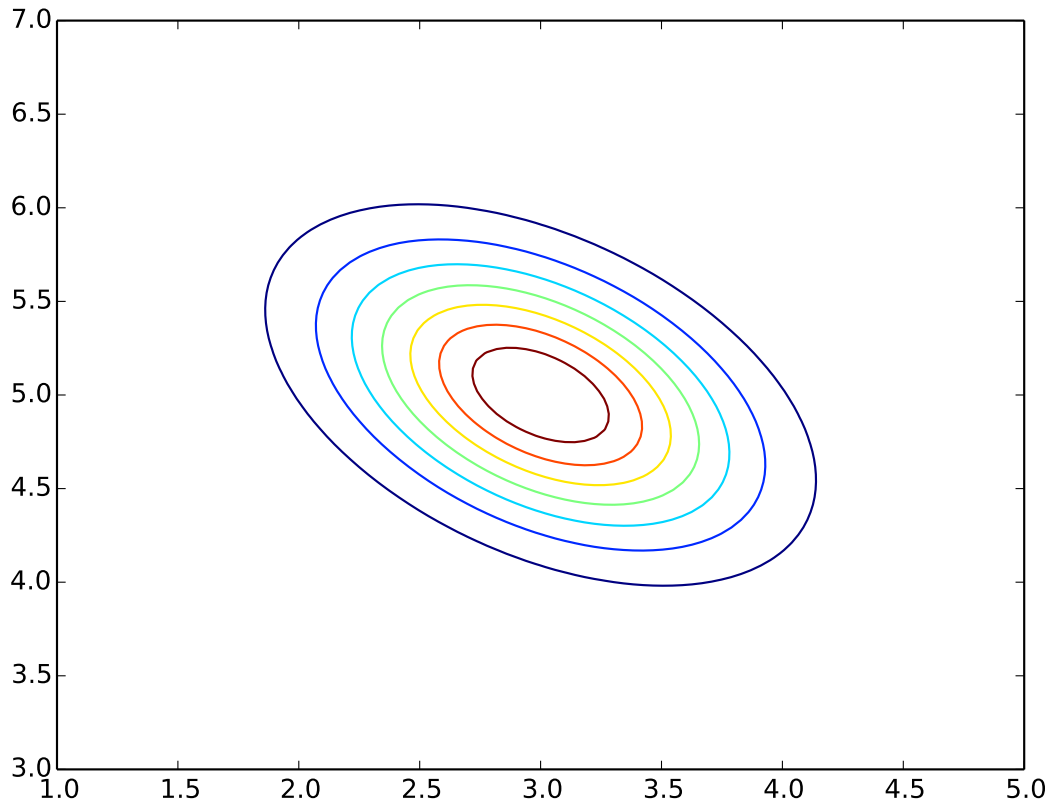
The probability density function of a scalar random variable can be plotted using the function `pdf()`:

```
>>> 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...>]
```



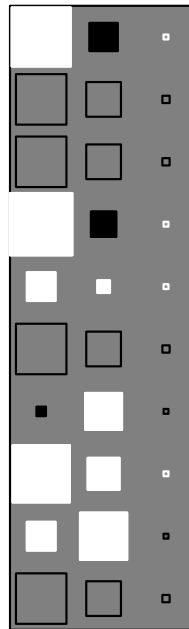
The variable `tau` models the inverse variance of the noise, for which the true value is $0.1^{-2} = 100$. Thus, the posterior captures the true value quite accurately. Similarly, the function `contour()` can be used to plot the probability density function of a 2-dimensional variable, for instance:

```
>>> 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:

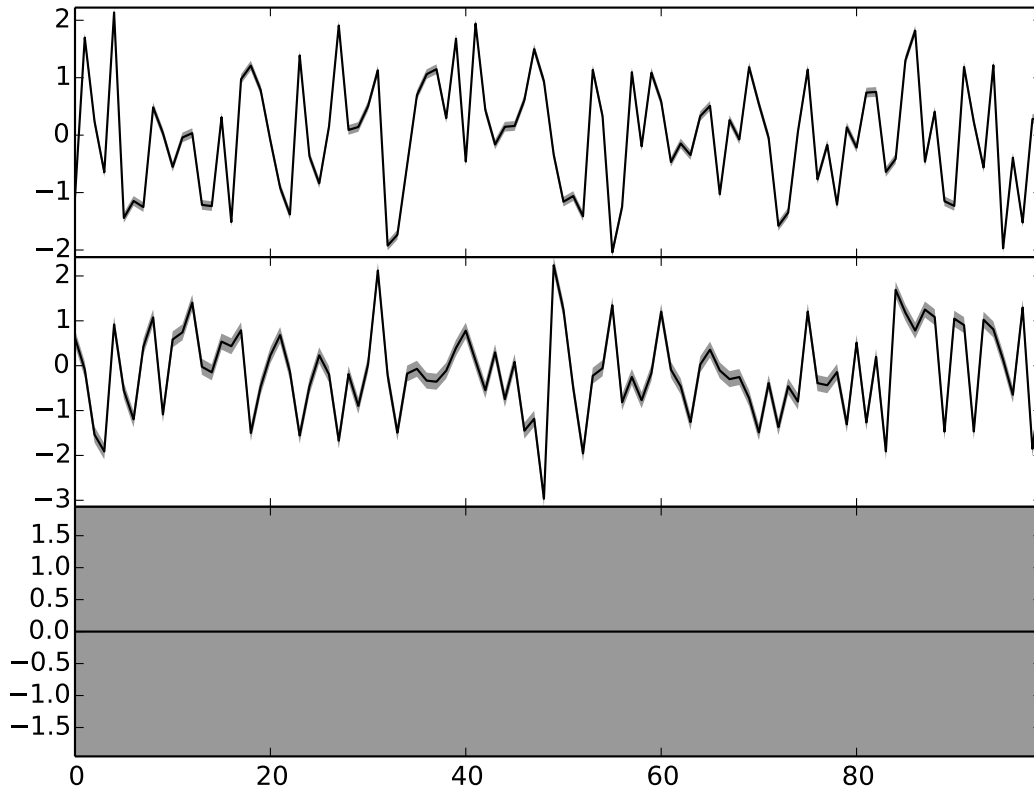
```
>>> bpplt.pyplot.figure()
<matplotlib.figure.Figure object at 0x...>
>>> bpplt.hinton(C)
```



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 $n = 0, \dots, N-1$. 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.

EXAMPLES

3.1 Regression

3.1.1 Linear regression

```
import numpy as np
k = 2
c = 5
s = 2

x = np.arange(10)
y = k*x + c + s*np.random.randn(10)

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

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

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

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

Y = GaussianARD(F, tau)
Y.observe(y)

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

Q.update(repeat=100)

import bayespy.plot as bpplt
# These two lines are needed to enable inline plotting IPython Notebooks
%matplotlib inline
bpplt.pyplot.plot([])

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

```
Iteration 1: loglike=-4.515537e+01 (0.000 seconds)
Iteration 2: loglike=-4.429472e+01 (0.010 seconds)
Iteration 3: loglike=-4.428241e+01 (0.000 seconds)
Iteration 4: loglike=-4.428197e+01 (0.000 seconds)
Iteration 5: loglike=-4.428195e+01 (0.010 seconds)
Converged.
```

```
bpplt.pdf(tau, np.linspace(1e-6,1,100), color='k')
bpplt.pyplot.axvline(s**(-2), color='r')
# Add labels
bpplt.pyplot.title(r'$q(\tau)$')
bpplt.pyplot.xlabel(r'$\tau$');

bpplt.contour(B, np.linspace(1,3,1000), np.linspace(1,9,1000), n=10, colors='k')
bpplt.plot(c, x=k, color='r', marker='x', linestyle='None', markersize=10, markedgewidth=2)
# Add labels
bpplt.pyplot.title(r'$q(k,c)$')
bpplt.pyplot.xlabel(r'$k$')
bpplt.pyplot.ylabel(r'$c$');
```

3.1.2 Improving accuracy

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

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

```
Y = GaussianARD(F_tau, 1)
Y.observe(y)
```

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

```
Q.update(repeat=10)
```

```
Iteration 1: loglike=-4.594957e+01 (0.000 seconds)
Iteration 2: loglike=-4.594957e+01 (0.000 seconds)
Converged.
```

```
bpplt.pdf(B_tau.get_marginal_logpdf(gaussian=None, gamma=True),
          np.linspace(1e-6,1,100), color='k')
bpplt.pyplot.axvline(s**(-2), color='r')
# Add labels
bpplt.pyplot.title(r'$q(\tau)$')
bpplt.pyplot.xlabel(r'$\tau$');

bpplt.contour(B_tau.get_marginal_logpdf(gaussian=[0,1], gamma=False),
              np.linspace(1,3,100), np.linspace(1,9,100),
              n=10, colors='k')
# Plot the true value
bpplt.plot(c, x=k, color='r', marker='x', linestyle='None', markersize=10, markedgewidth=2)
# Add labels
bpplt.pyplot.title(r'$q(k,c)$')
bpplt.pyplot.xlabel(r'$k$')
bpplt.pyplot.ylabel(r'$c$');
```

```

bpplt.contour(B_tau.get_marginal_logpdf(gaussian=[0], gamma=True),
              np.linspace(1,3,100), np.linspace(1e-6,1,100),
              n=10, colors='k')
bpplt.plot(s**(-2), x=k, color='r', marker='x', linestyle='None', markersize=10, markeredgewidth=2)
bpplt.pyplot.title(r'$q(k, \tau)$')
bpplt.pyplot.xlabel(r'$k$')
bpplt.pyplot.ylabel(r'$\tau$');

xh = np.linspace(-5, 15, 100)
Xh = np.vstack([xh, np.ones(len(xh))]).T
Fh_tau = SumMultiply('i,i', B_tau, Xh)
bpplt.plot(Fh_tau, x=xh, scale=2)
bpplt.plot(y, x=x, color='r', marker='x', linestyle='None')
bpplt.plot(k*xh+c, x=xh, color='r')

-----
AttributeError                                Traceback (most recent call last)

<ipython-input-8-bad1c68bbf3d> in <module>()
      2 Xh = np.vstack([xh, np.ones(len(xh))]).T
      3 Fh_tau = SumMultiply('i,i', B_tau, Xh)
----> 4 bpplt.plot(Fh_tau, x=xh, scale=2)
      5 bpplt.plot(y, x=x, color='r', marker='x', linestyle='None')
      6 bpplt.plot(k*xh+c, x=xh, color='r')

/home/jluttine/workspace/bayespy/bayespy/plot.py in plot(Y, axis, scale, center, **kwargs)
    125         return plot_gaussian(Y, axis=axis, scale=scale, center=center, **kwargs)
    126
--> 127     (mu, var) = Y.get_mean_and_variance()
    128     std = np.sqrt(var)
    129

AttributeError: 'SumMultiply' object has no attribute 'get_mean_and_variance'

```

3.1.3 Multivariate regression

3.1.4 Non-linear regression

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:

```

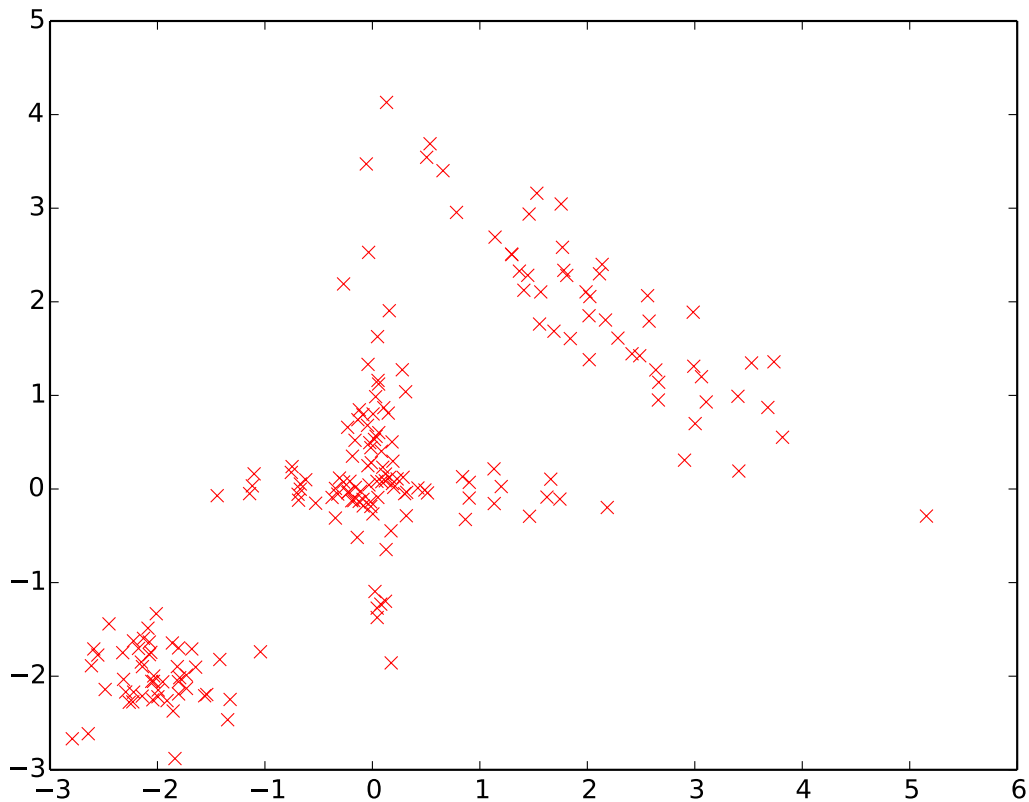
>>> import numpy as np
>>> y0 = np.random.multivariate_normal([0, 0], [[2, 0], [0, 0.1]], size=50)
>>> y1 = np.random.multivariate_normal([0, 0], [[0.1, 0], [0, 2]], size=50)
>>> y2 = np.random.multivariate_normal([2, 2], [[2, -1.5], [-1.5, 2]], size=50)

```

```
>>> 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 α :


```
>>> 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:

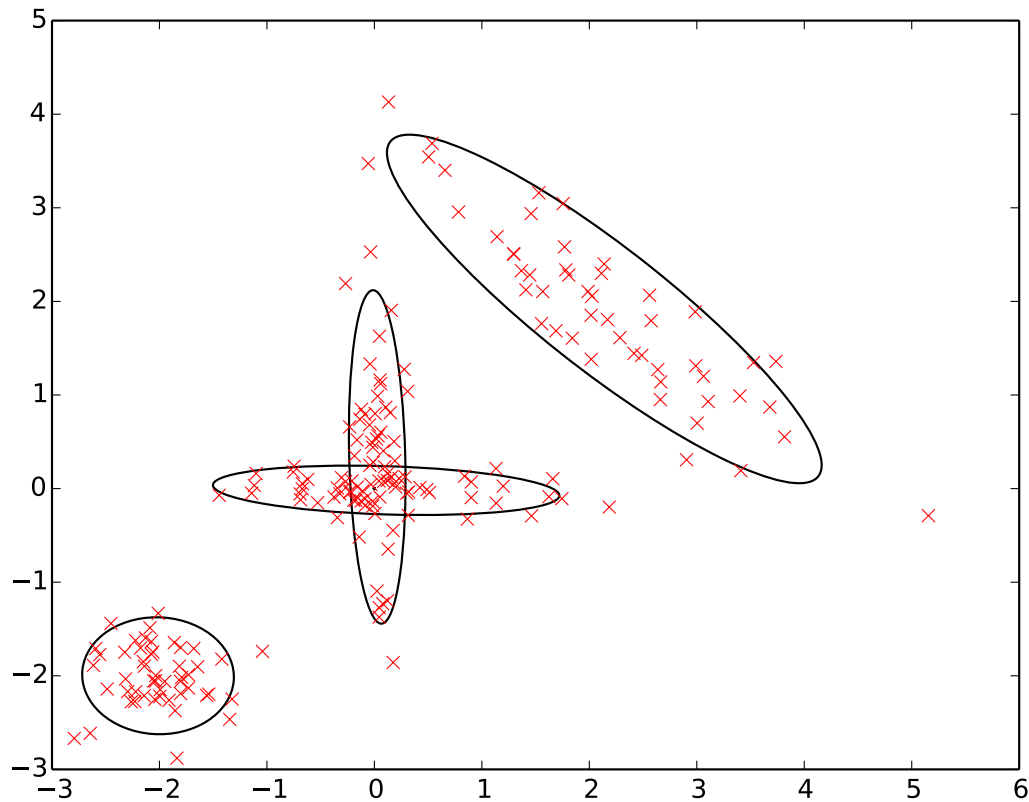
```
>>> Q.update(repeat=1000)
Iteration 1: loglike=-1.401968e+03 (... seconds)
...
Iteration 48: loglike=-1.017893e+03 (... seconds)
Converged at iteration 48.
```

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.

3.2.5 Next steps

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.

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

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

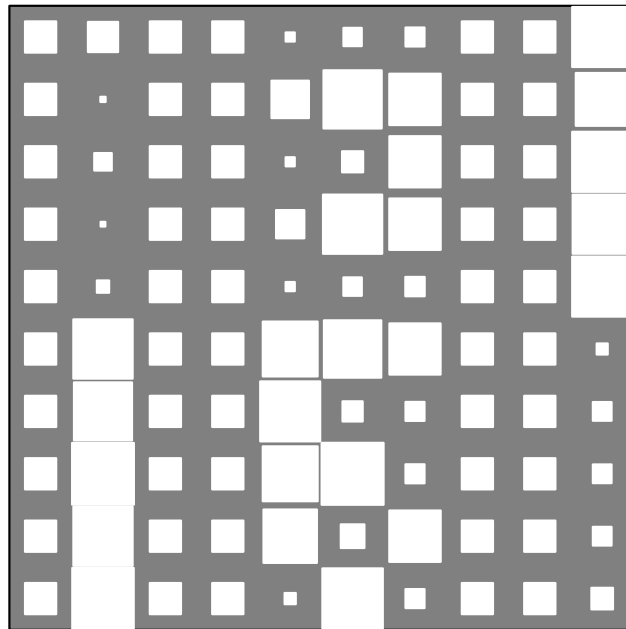
Now we can examine the approximate posterior distribution. First, let us plot the group assignment probabilities:

```
>>> 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:

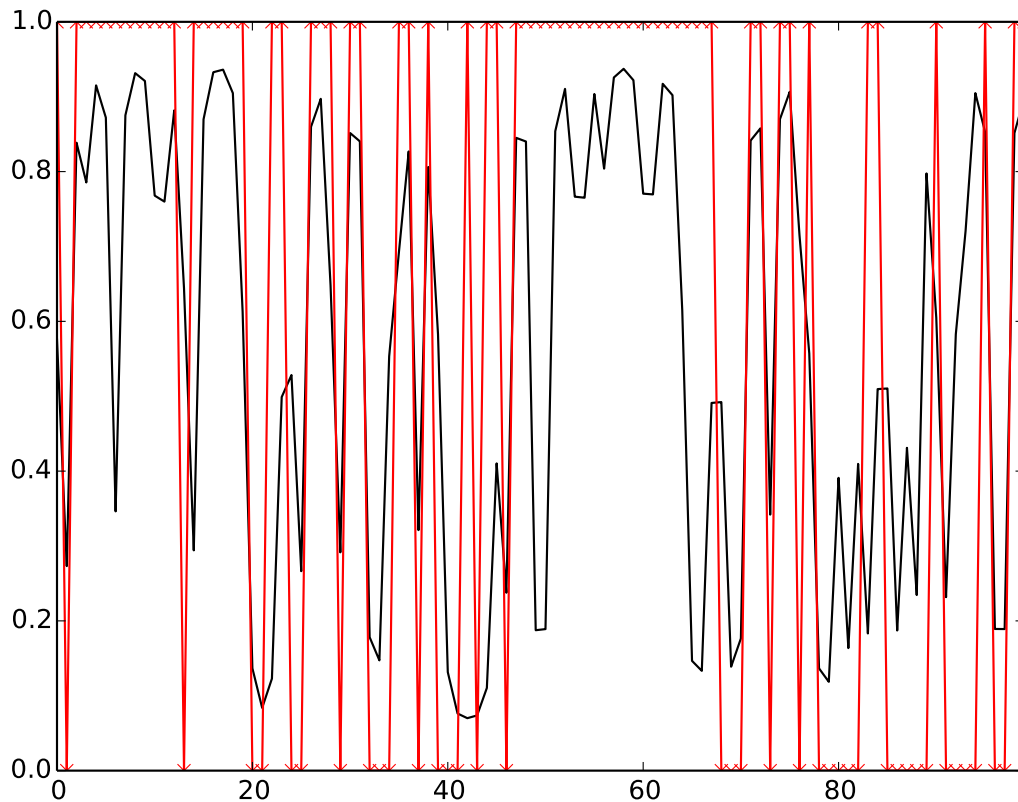
```
>>> Q.update()
Iteration 1: loglike=-1.095883e+02 (... seconds)
```

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 activities 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.

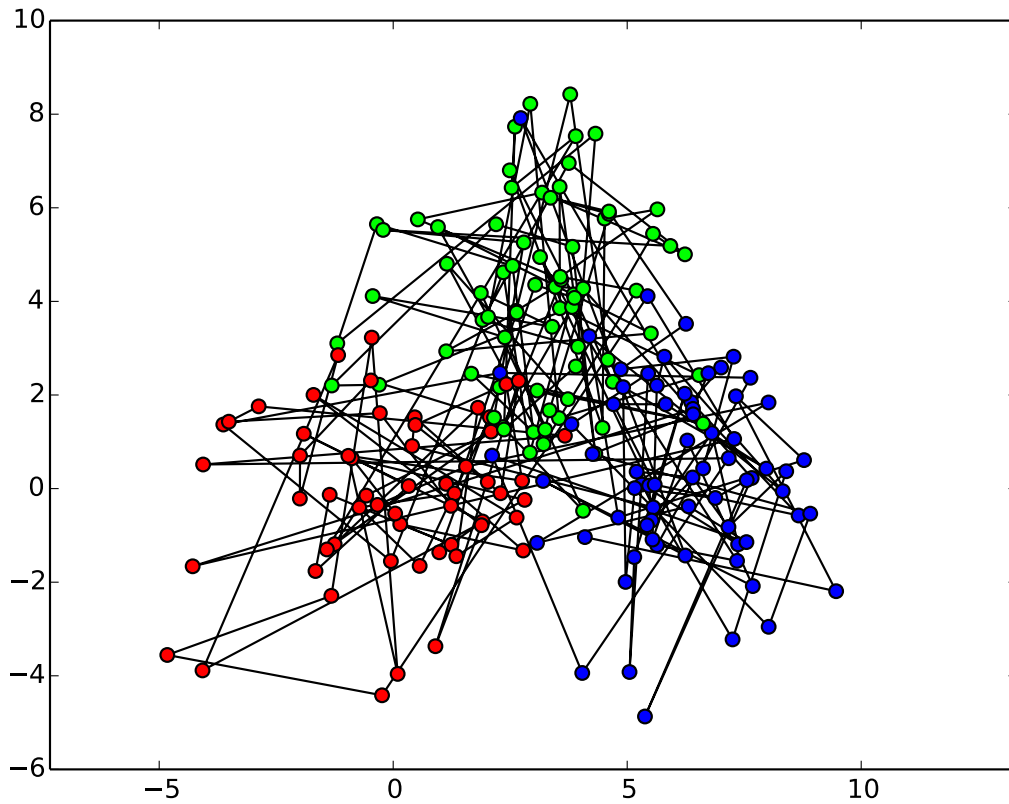
```
>>> q = 0.9
>>> r = (1-q) / (K-1)
>>> P = q*np.identity(K) + r*(np.ones((3,3))-np.identity(3))
```

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)
```

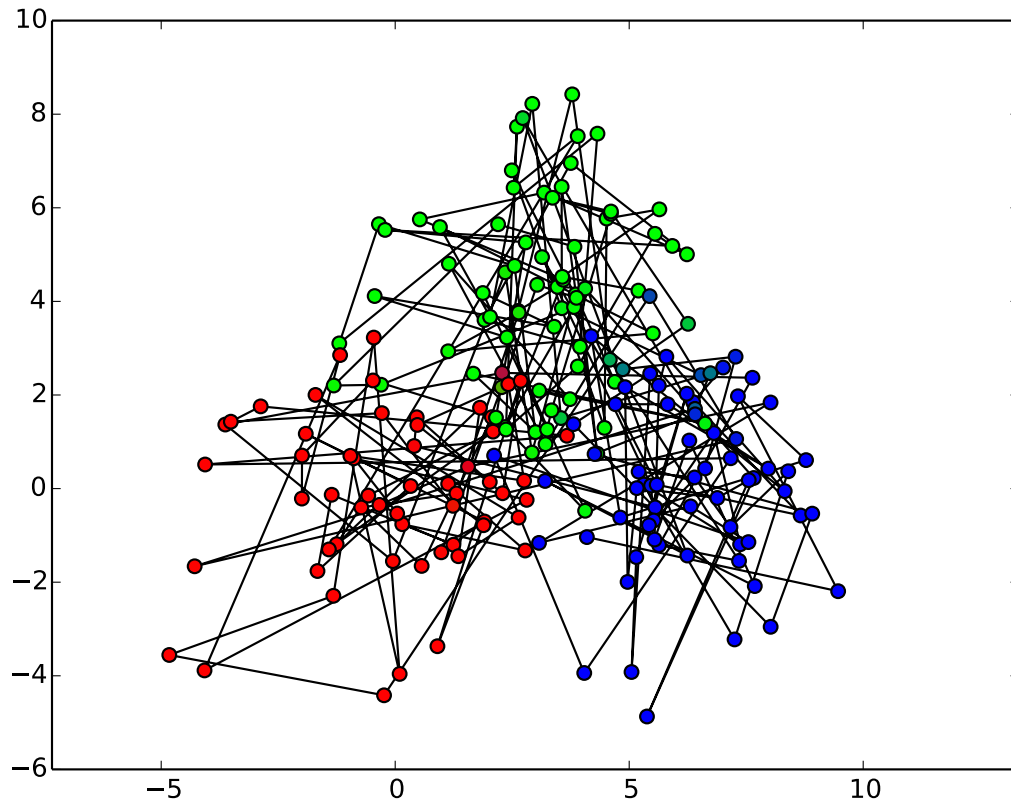
Then, run VB iteration until convergence:

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

Results

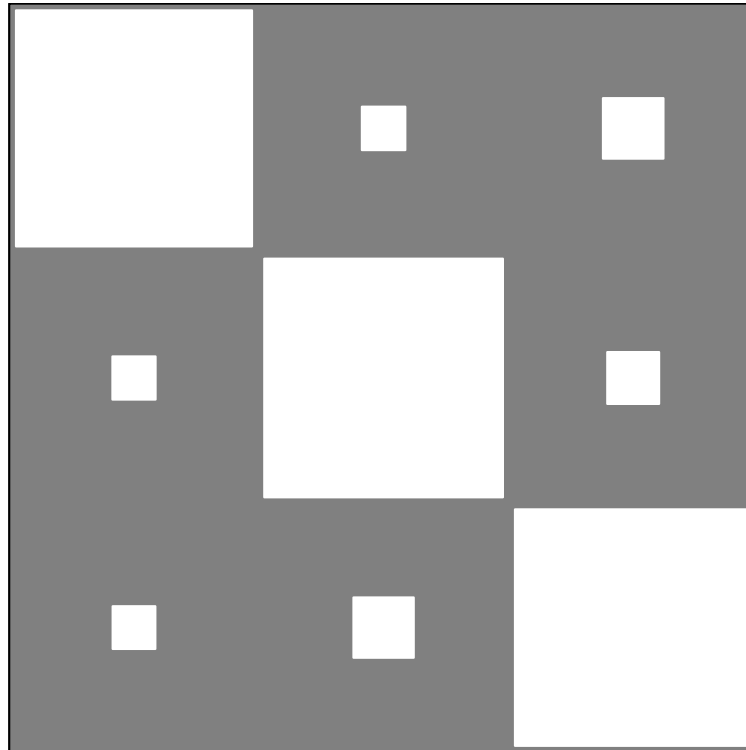
Plot the classification of the data similarly as the data:

```
>>> 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)
```

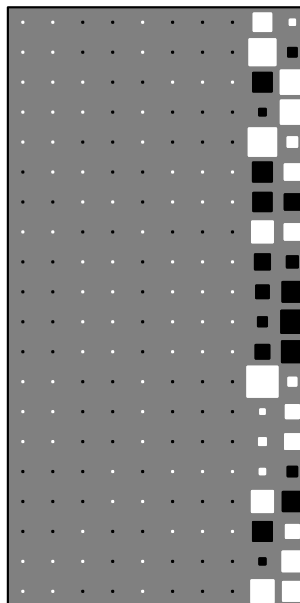
For more information about the rotation parameter expansion, see [2] and [1]. Now we can run the actual inference until convergence:

```
>>> Q.update(repeat=1000)
Iteration 1: loglike=-2.339710e+03 (... seconds)
...
Iteration 23: loglike=6.500608e+02 (... seconds)
Converged at iteration 23.
```

3.5.4 Results

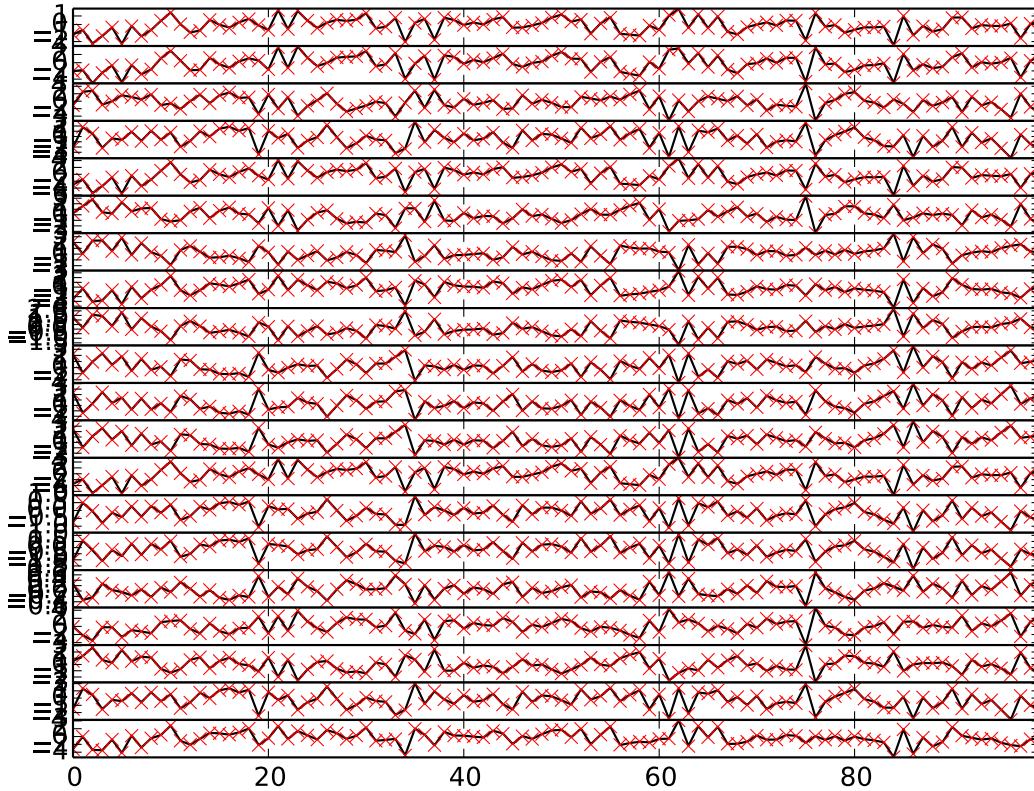
The results can be visualized, for instance, by plotting the Hinton diagram of the loading matrix:

```
>>> 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 $\mathbf{Y} = (\mathbf{y}_1, \dots, \mathbf{y}_N)$ is assumed to be generated from latent D -dimensional states $\mathbf{X} = (\mathbf{x}_1, \dots, \mathbf{x}_N)$ which follow a first-order Markov process:

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

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 **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 **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 constructed 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 [1] and [2]. 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 \mathbf{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
```

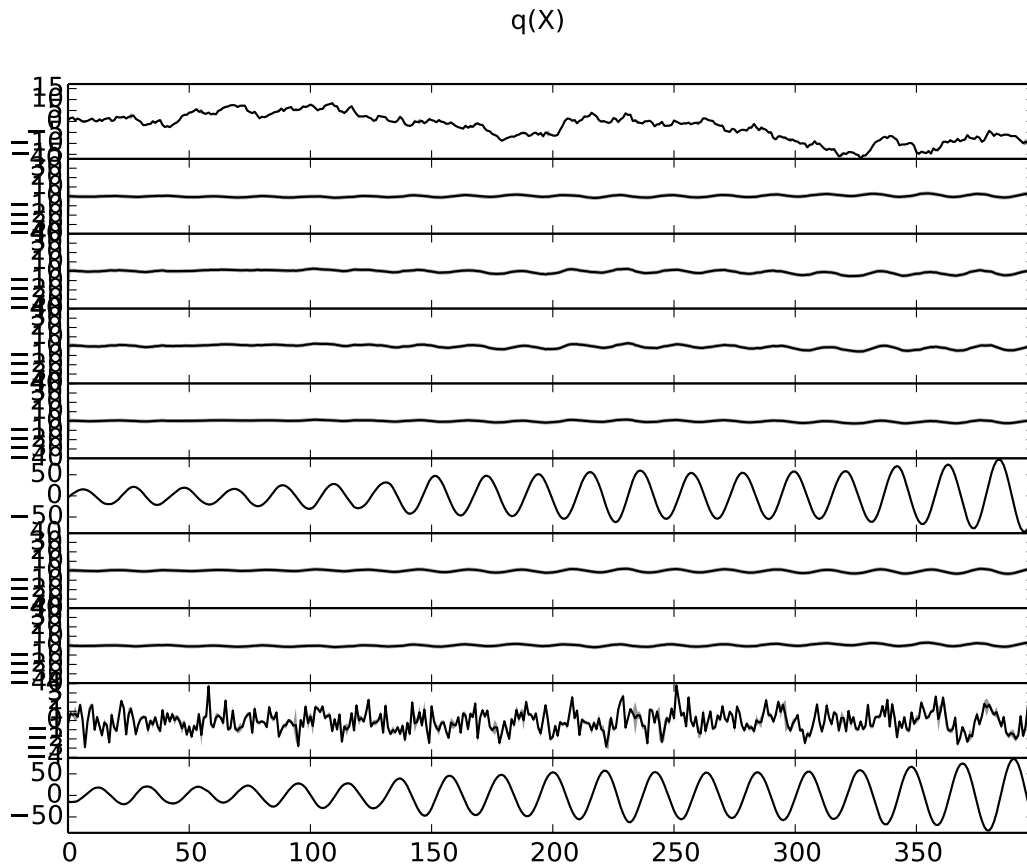
Now, run iterations until convergence:

```
>>> Q.update(repeat=1000)
Iteration 11: loglike=-1.010806e+04 (... seconds)
...
Iteration 60: loglike=-8.906224e+03 (... seconds)
Converged at iteration 60.
```

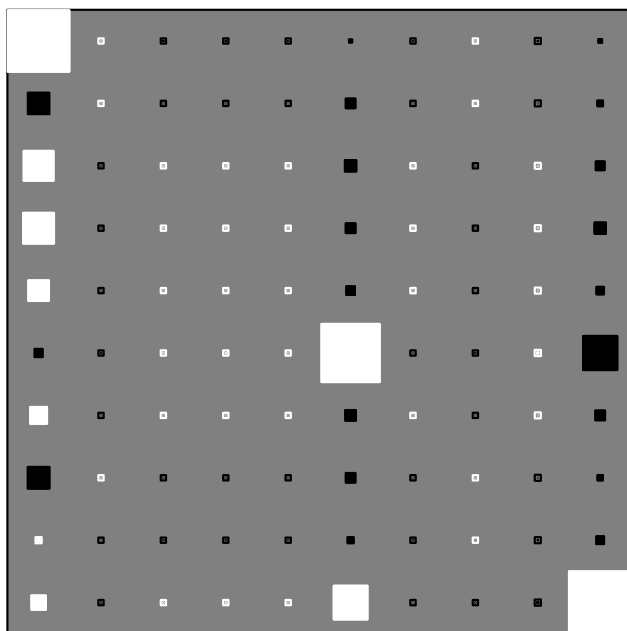
3.6.4 Results

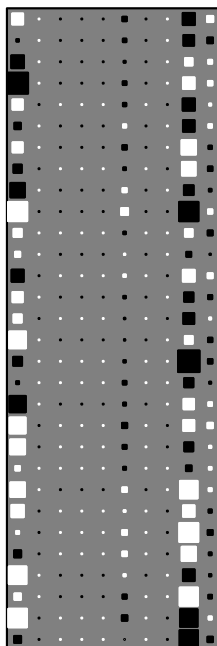
Because we have set the plotters, we can plot those nodes as:

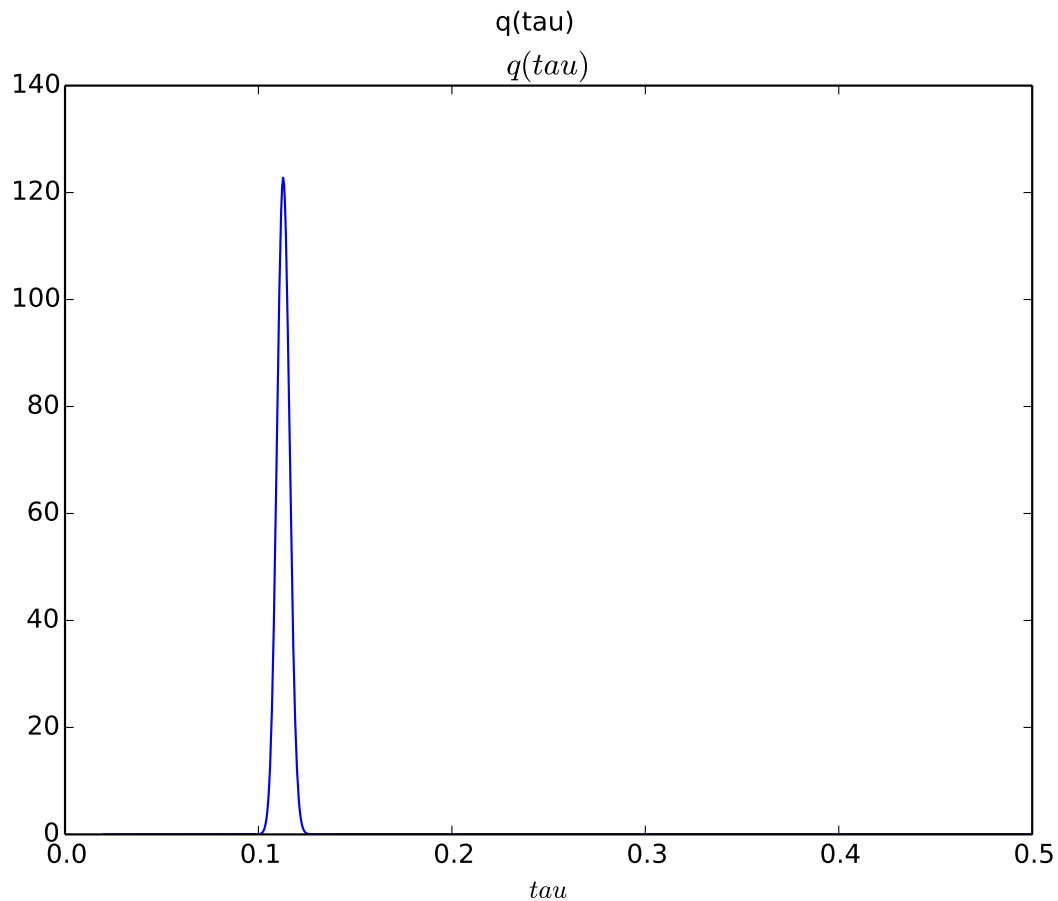
```
>>> Q.plot(X, A, C, tau)
```



$q(A)$

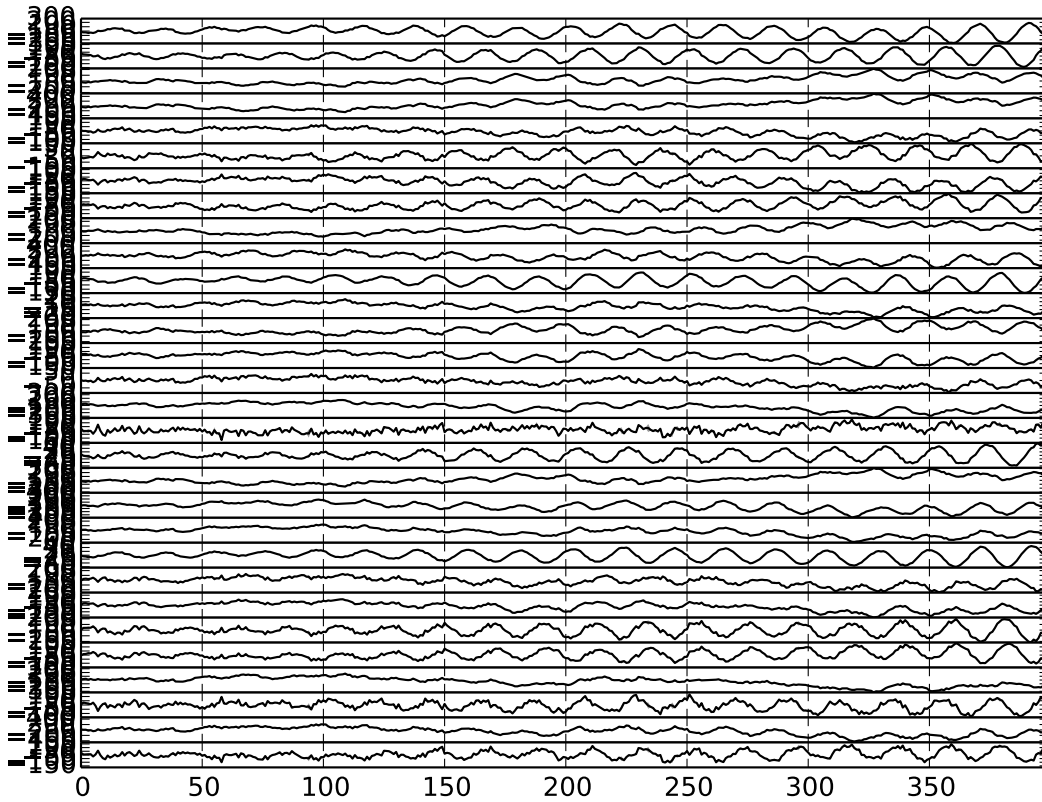


$q(C)$ 



There are clearly four effective components in \mathbf{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 \mathbf{C} shows only four effective components. The posterior of τ 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])
5.182592...
```

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.

DEVELOPER GUIDE

How to document: https://github.com/numpy/numpy/blob/master/doc/HOWTO_DOCUMENT.rst.txt

How to contribute: http://docs.scipy.org/doc/numpy/dev/gitwash/development_workflow.html

4.1 Variational message passing

The general update equation for factorized approximation:

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

where $\text{pa}(\boldsymbol{\theta})$ and $\text{ch}(\boldsymbol{\theta})$ are the set of parents and children of $\boldsymbol{\theta}$, respectively. 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 uses only the Markov blanket of $\boldsymbol{\theta}$. Thus, the optimization can be done locally using messages from neighbouring nodes.

The messages are simple for conjugate-exponential models. Exponential-family distributions have the form

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

where $\boldsymbol{\Theta} = \{\boldsymbol{\theta}_j\}$ is the set of parents. If a parent has a conjugate prior, (4.2) is linear with respect to the parent's natural statistics. Thus, (4.2) can be re-organized with respect to $\boldsymbol{\theta}_j$ as

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

where $\mathbf{u}_{\boldsymbol{\theta}_j}$ is the natural statistics of $\boldsymbol{\theta}_j$. Thus, the update equation (4.1) can be given as

$$\log q(\boldsymbol{\theta}_j) = \mathbf{u}_{\boldsymbol{\theta}_j}(\boldsymbol{\theta}_j)^T \left(\langle \boldsymbol{\phi}_{\boldsymbol{\theta}_j} \rangle + \sum_{\mathbf{x} \in \text{ch}(\boldsymbol{\theta}_j)} \langle \boldsymbol{\phi}_{\mathbf{x} \rightarrow \boldsymbol{\theta}_j} \rangle \right) + f_{\boldsymbol{\theta}_j}(\boldsymbol{\theta}_j) + \text{const},$$

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 expectations of the parents' natural statistics. Similarly, $\langle \boldsymbol{\phi}_{\mathbf{x} \rightarrow \boldsymbol{\theta}_j} \rangle$ depends (multi)linearly on the expectations of the children's and co-parents' natural statistics.

The required expectations can be computed locally by using messages from the parents and the children. The message from a parent node $\boldsymbol{\theta}_j$ to a child node \mathbf{x} is

$$\mathbf{m}_{\boldsymbol{\theta}_j \rightarrow \mathbf{x}} = \langle \mathbf{u}_{\boldsymbol{\theta}_j} \rangle = \tilde{\mathbf{u}}_{\boldsymbol{\theta}_j}(\tilde{\boldsymbol{\phi}}_{\boldsymbol{\theta}_j}),$$

and the message from a child node \mathbf{x} to a parent node $\boldsymbol{\theta}_j$ is

$$\mathbf{m}_{\mathbf{x} \rightarrow \boldsymbol{\theta}_j} = \langle \boldsymbol{\phi}_{\mathbf{x} \rightarrow \boldsymbol{\theta}_j} \rangle = \boldsymbol{\phi}_{\mathbf{x} \rightarrow \boldsymbol{\theta}_j}(\langle \mathbf{u}_{\mathbf{x}} \rangle, \{\mathbf{m}_{\boldsymbol{\theta}_k \rightarrow \mathbf{x}}\}_{k \neq j}).$$

Using the messages, the natural parameters of $q(\boldsymbol{\theta})$ can be updated as

$$\tilde{\boldsymbol{\phi}}_{\boldsymbol{\theta}} = \boldsymbol{\phi}_{\boldsymbol{\theta}}(\{\mathbf{m}_{\mathbf{z} \rightarrow \boldsymbol{\theta}}\}_{\mathbf{z} \in \text{pa}(\boldsymbol{\theta})}) + \sum_{\mathbf{x} \in \text{ch}(\boldsymbol{\theta})} \mathbf{m}_{\mathbf{x} \rightarrow \boldsymbol{\theta}}.$$

4.2 Implementing nodes

<code>bayespy.nodes</code>	Package for nodes used to construct the model.
<code>bayespy.inference</code>	Package for Bayesian inference engines
<code>bayespy.plot</code>	Functions for plotting nodes.

5.1 bayespy.nodes

Package for nodes used to construct the model.

5.1.1 Stochastic nodes

Nodes for Gaussian variables:

<code>Gaussian(mu, Lambda, **kwargs)</code>	Node for Gaussian variables.
<code>GaussianARD(mu, alpha[, ndim, shape])</code>	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:

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

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

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

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

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

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.delete

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

bayespy.nodes.Gaussian.get_mask

`Gaussian.get_mask()`

bayespy.nodes.Gaussian.get_moments

`Gaussian.get_moments()`

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)`

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 (**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 $\text{kron}(R1, R2) * x$ (??)

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_plotter

`Gaussian.set_plotter(plotter)`

bayespy.nodes.Gaussian.show

`Gaussian.show()`

bayespy.nodes.Gaussian.unobserve

`Gaussian.unobserve()`

bayespy.nodes.Gaussian.update

`Gaussian.update()`

Attributes

`dims`
`plates`

bayespy.nodes.Gaussian.dims

`Gaussian.dims = None`

bayespy.nodes.Gaussian.plates

`Gaussian.plates = None`

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:

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

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

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

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

<code>__init__(mu, alpha[, ndim, shape])</code>	Create GaussianARD node.
<code>add_plate_axis(to_plate)</code>	
<code>delete()</code>	Delete this node and the children
<code>get_mask()</code>	
<code>get_moments()</code>	
<code>get_shape(ind)</code>	
<code>has_plotter()</code>	Return True if the node has a plotter
<code>initialize_from_mean_and_covariance(mu, Cov)</code>	
<code>initialize_from_parameters(mu, alpha)</code>	
<code>initialize_from_prior()</code>	
<code>initialize_from_random()</code>	Set the variable to a random sample from the current distribution.
<code>initialize_from_value(x, *args)</code>	
<code>load(group)</code>	Load the state of the node from a HDF5 file.
<code>logpdf(X[, mask])</code>	Compute the log probability density function $Q(X)$ of this node.
<code>lower_bound_contribution([gradient])</code>	
<code>lowerbound()</code>	
<code>move_plates(from_plate, to_plate)</code>	
<code>observe(x, *args[, mask])</code>	Fix moments, compute f and propagate mask.
<code>pdf(X[, mask])</code>	Compute the probability density function of this node.
<code>plot(**kwargs)</code>	Plot the node distribution using the plotter of the node
<code>random()</code>	Draw a random sample from the distribution.
<code>rotate(R[, inv, logdet, axis, Q])</code>	
<code>rotate_plates(Q[, plate_axis])</code>	Approximate rotation of a plate axis.
<code>save(group)</code>	Save the state of the node into a HDF5 file.
<code>set_plotter(plotter)</code>	
<code>show()</code>	
<code>unobserve()</code>	
<code>update()</code>	

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.delete

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

bayespy.nodes.GaussianARD.get_mask

`GaussianARD.get_mask()`

bayespy.nodes.GaussianARD.get_moments

`GaussianARD.get_moments()`

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)`

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 (**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)`

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_plotter

`GaussianARD.set_plotter (plotter)`

bayespy.nodes.GaussianARD.show

`GaussianARD.show()`

bayespy.nodes.GaussianARD.unobserve

`GaussianARD.unobserve()`

bayespy.nodes.GaussianARD.update

`GaussianARD.update()`

Attributes

`dims`
`plates`

`bayespy.nodes.GaussianARD.dims`

`GaussianARD.dims = None`

`bayespy.nodes.GaussianARD.plates`

`GaussianARD.plates = None`

Nodes for precision and scale variables:

<code>Gamma(a, b, **kwargs)</code>	Node for gamma random variables.
<code>Wishart(n, V, **kwargs)</code>	Node for Wishart random variables.
<code>Exponential(l, **kwargs)</code>	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

<code>__init__(a, b, **kwargs)</code>	Create gamma random variable node
<code>add_plate_axis(to_plate)</code>	
<code>as_diagonal_wishart()</code>	
<code>delete()</code>	Delete this node and the children
<code>get_mask()</code>	
<code>get_moments()</code>	
<code>get_shape(ind)</code>	
<code>has_plotter()</code>	Return True if the node has a plotter
<code>initialize_from_parameters(*args)</code>	
<code>initialize_from_prior()</code>	
<code>initialize_from_random()</code>	Set the variable to a random sample from the current distribution.
<code>initialize_from_value(x, *args)</code>	
<code>load(group)</code>	Load the state of the node from a HDF5 file.

Continued on next page

Table 5.8 – continued from previous page

<code>logpdf(X[, mask])</code>	Compute the log probability density function $Q(X)$ of this node.
<code>lower_bound_contribution([gradient])</code>	
<code>lowerbound()</code>	
<code>move_plates(from_plate, to_plate)</code>	
<code>observe(x, *args[, mask])</code>	Fix moments, compute f and propagate mask.
<code>pdf(X[, mask])</code>	Compute the probability density function of this node.
<code>plot(**kwargs)</code>	Plot the node distribution using the plotter of the node
<code>random()</code>	Draw a random sample from the distribution.
<code>save(group)</code>	Save the state of the node into a HDF5 file.
<code>set_plotter(plotter)</code>	
<code>show()</code>	Print the distribution using standard parameterization.
<code>unobserve()</code>	
<code>update()</code>	

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.delete

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

bayespy.nodes.Gamma.get_mask

`Gamma.get_mask ()`

bayespy.nodes.Gamma.get_moments

`Gamma.get_moments ()`

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)`

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 (**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_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 ()`

Attributes

<code>dims</code>	tuple() -> empty tuple
<code>plates</code>	

`bayespy.nodes.Gamma.dims`

`Gamma.dims = (), ()`

`bayespy.nodes.Gamma.plates`

`Gamma.plates = None`

`bayespy.nodes.Wishart`

class `bayespy.nodes.Wishart` (*n*, *V*, ***kwargs*)

Node for Wishart random variables.

The random variable Λ is a $D \times D$ positive-definite symmetric matrix.

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

Parameters *n* : scalar or array

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

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

V, scale matrix.

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

Create Wishart node.

Methods

<code>__init__(n, V, **kwargs)</code>	Create Wishart node.
<code>add_plate_axis(to_plate)</code>	
<code>delete()</code>	Delete this node and the children
<code>get_mask()</code>	
<code>get_moments()</code>	
<code>get_shape(ind)</code>	
<code>has_plotter()</code>	Return True if the node has a plotter
<code>initialize_from_parameters(*args)</code>	
<code>initialize_from_prior()</code>	
<code>initialize_from_random()</code>	Set the variable to a random sample from the current distribution.
<code>initialize_from_value(x, *args)</code>	
<code>load(group)</code>	Load the state of the node from a HDF5 file.
<code>logpdf(X[, mask])</code>	Compute the log probability density function $Q(X)$ of this node.
<code>lower_bound_contribution([gradient])</code>	
<code>lowerbound()</code>	
<code>move_plates(from_plate, to_plate)</code>	

Continued on next page

Table 5.10 – continued from previous page

<code>observe(x, *args[, mask])</code>	Fix moments, compute f and propagate mask.
<code>pdf(X[, mask])</code>	Compute the probability density function of this node.
<code>plot(**kwargs)</code>	Plot the node distribution using the plotter of the node
<code>random()</code>	Draw a random sample from the distribution.
<code>save(group)</code>	Save the state of the node into a HDF5 file.
<code>set_plotter(plotter)</code>	
<code>show()</code>	
<code>unobserve()</code>	
<code>update()</code>	

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.delete

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

bayespy.nodes.Wishart.get_mask

`Wishart.get_mask()`

bayespy.nodes.Wishart.get_moments

`Wishart.get_moments()`

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)`

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(**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_plotter

`Wishart.set_plotter(plotter)`

bayespy.nodes.Wishart.show

`Wishart.show()`

bayespy.nodes.Wishart.unobserve

`Wishart.unobserve()`

bayespy.nodes.Wishart.update

`Wishart.update()`

Attributes

<code>dims</code>
<code>plates</code>

bayespy.nodes.Wishart.dims

`Wishart.dims = None`

bayespy.nodes.Wishart.plates

Wishart.plates = None

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

<code>__init__(l, **kwargs)</code>	
<code>add_plate_axis(to_plate)</code>	
<code>as_diagonal_wishart()</code>	
<code>delete()</code>	Delete this node and the children
<code>get_mask()</code>	
<code>get_moments()</code>	
<code>get_shape(ind)</code>	
<code>has_plotter()</code>	Return True if the node has a plotter
<code>initialize_from_parameters(*args)</code>	
<code>initialize_from_prior()</code>	
<code>initialize_from_random()</code>	Set the variable to a random sample from the current distribution.
<code>initialize_from_value(x, *args)</code>	
<code>load(group)</code>	Load the state of the node from a HDF5 file.
<code>logpdf(X[, mask])</code>	Compute the log probability density function $Q(X)$ of this node.
<code>lower_bound_contribution([gradient])</code>	
<code>lowerbound()</code>	
<code>move_plates(from_plate, to_plate)</code>	

Continued on next page

Table 5.12 – continued from previous page

<code>observe(x, *args[, mask])</code>	Fix moments, compute f and propagate mask.
<code>pdf(X[, mask])</code>	Compute the probability density function of this node.
<code>plot(**kwargs)</code>	Plot the node distribution using the plotter of the node
<code>random()</code>	Draw a random sample from the distribution.
<code>save(group)</code>	Save the state of the node into a HDF5 file.
<code>set_plotter(plotter)</code>	
<code>show()</code>	Print the distribution using standard parameterization.
<code>unobserve()</code>	
<code>update()</code>	

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.delete

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

bayespy.nodes.Exponential.get_mask

`Exponential.get_mask()`

bayespy.nodes.Exponential.get_moments

`Exponential.get_moments()`

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)`

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(**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_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()`

Continued on next page

Table 5.13 – continued from previous page

Attributes

<code>dims</code>	tuple() -> empty tuple
<code>plates</code>	

`bayespy.nodes.Exponential.dims`

`Exponential.dims = (), ()`

`bayespy.nodes.Exponential.plates`

`Exponential.plates = None`

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

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

<code>__init__(*args, **kwargs)</code>	
<code>add_plate_axis(to_plate)</code>	
<code>delete()</code>	Delete this node and the children
<code>get_gaussian_mean_and_variance()</code>	Return the mean and variance of the distribution
<code>get_marginal_logpdf([gaussian, gamma])</code>	Get the (marginal) log pdf of a subset of the variables
<code>get_mask()</code>	
<code>get_moments()</code>	
<code>get_shape(ind)</code>	
<code>has_plotter()</code>	Return True if the node has a plotter

Continued on next page

Table 5.15 – continued from previous page

<code>initialize_from_parameters(*args)</code>	
<code>initialize_from_prior()</code>	
<code>initialize_from_random()</code>	Set the variable to a random sample from the current distribution.
<code>initialize_from_value(x, *args)</code>	
<code>load(group)</code>	Load the state of the node from a HDF5 file.
<code>logpdf(X[, mask])</code>	Compute the log probability density function $Q(X)$ of this node.
<code>lower_bound_contribution([gradient])</code>	
<code>lowerbound()</code>	
<code>move_plates(from_plate, to_plate)</code>	
<code>observe(x, *args[, mask])</code>	Fix moments, compute f and propagate mask.
<code>pdf(X[, mask])</code>	Compute the probability density function of this node.
<code>plot(**kwargs)</code>	Plot the node distribution using the plotter of the node
<code>plotmatrix()</code>	Creates a matrix of marginal plots.
<code>random()</code>	Draw a random sample from the distribution.
<code>save(group)</code>	Save the state of the node into a HDF5 file.
<code>set_plotter(plotter)</code>	
<code>show()</code>	Print the distribution using standard parameterization.
<code>unobserve()</code>	
<code>update()</code>	

bayespy.nodes.GaussianGammaISO.__init__

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

bayespy.nodes.GaussianGammaISO.add_plate_axis

`GaussianGammaISO.add_plate_axis (to_plate)`

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_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_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*)

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** (***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_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()`

Attributes

`dims`
`plates`

bayespy.nodes.GaussianGammaISO.dims

`GaussianGammaISO.dims = None`

bayespy.nodes.GaussianGammaISO.plates

`GaussianGammaISO.plates = None`

bayespy.nodes.GaussianGammaARD

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

The prior:

$$\begin{aligned}
 p(x, \tau | \mu, \alpha, a, b) &= p(x | \tau, \mu, \alpha) p(\tau | a, b) \\
 p(x | \alpha, \mu, \alpha) &= \mathcal{N}(x | \mu, \text{diag}(\alpha \tau)) \\
 p(\tau | a, b) &= \mathcal{G}(\tau | a, b)
 \end{aligned}$$

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

<code>__init__</code> (<i>mu</i> , <i>alpha</i> , <i>a</i> , <i>b</i> , <i>**kwargs</i>)	
<code>add_plate_axis</code> (<i>to_plate</i>)	
<code>delete</code> ()	Delete this node and the children
<code>get_mask</code> ()	
<code>get_moments</code> ()	
<code>get_shape</code> (<i>ind</i>)	
<code>has_plotter</code> ()	Return True if the node has a plotter
<code>initialize_from_parameters</code> (* <i>args</i>)	
<code>initialize_from_prior</code> ()	
<code>initialize_from_random</code> ()	Set the variable to a random sample from the current distribution.
<code>initialize_from_value</code> (<i>x</i> , * <i>args</i>)	
<code>load</code> (<i>group</i>)	Load the state of the node from a HDF5 file.
<code>logpdf</code> (<i>X</i> [, <i>mask</i>])	Compute the log probability density function $Q(X)$ of this node.
<code>lower_bound_contribution</code> ([<i>gradient</i>])	
<code>lowerbound</code> ()	
<code>move_plates</code> (<i>from_plate</i> , <i>to_plate</i>)	
<code>observe</code> (<i>x</i> , * <i>args</i> [, <i>mask</i>])	Fix moments, compute f and propagate mask.
<code>pdf</code> (<i>X</i> [, <i>mask</i>])	Compute the probability density function of this node.
<code>plot</code> (* <i>kwargs</i>)	Plot the node distribution using the plotter of the node
<code>random</code> ()	Draw a random sample from the distribution.
<code>save</code> (<i>group</i>)	Save the state of the node into a HDF5 file.
<code>set_plotter</code> (<i>plotter</i>)	
<code>unobserve</code> ()	
<code>update</code> ()	

`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.delete`

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

bayespy.nodes.GaussianGammaARD.get_mask

GaussianGammaARD.get_mask()

bayespy.nodes.GaussianGammaARD.get_moments

GaussianGammaARD.get_moments()

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*)

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** (***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_plotter

GaussianGammaARD.**set_plotter** (*plotter*)

bayespy.nodes.GaussianGammaARD.unobserve

GaussianGammaARD.unobserve()

bayespy.nodes.GaussianGammaARD.update

GaussianGammaARD.update()

Attributes

dims
plates

bayespy.nodes.GaussianGammaARD.dims

GaussianGammaARD.dims = None

bayespy.nodes.GaussianGammaARD.plates

GaussianGammaARD.plates = None

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)
delete()
get_mask()
get_moments()
get_shape(ind)
has_plotter()
```

Delete this node and the children

Return True if the node has a plotter

Continued on next page

Table 5.19 – continued from previous page

<code>initialize_from_parameters(*args)</code>	
<code>initialize_from_prior()</code>	
<code>initialize_from_random()</code>	Set the variable to a random sample from the current distribution.
<code>initialize_from_value(x, *args)</code>	
<code>load(group)</code>	Load the state of the node from a HDF5 file.
<code>logpdf(X[, mask])</code>	Compute the log probability density function $Q(X)$ of this node.
<code>lower_bound_contribution([gradient])</code>	
<code>lowerbound()</code>	
<code>move_plates(from_plate, to_plate)</code>	
<code>observe(x, *args[, mask])</code>	Fix moments, compute f and propagate mask.
<code>pdf(X[, mask])</code>	Compute the probability density function of this node.
<code>plot(**kwargs)</code>	Plot the node distribution using the plotter of the node
<code>random()</code>	Draw a random sample from the distribution.
<code>save(group)</code>	Save the state of the node into a HDF5 file.
<code>set_plotter(plotter)</code>	
<code>show()</code>	Print the distribution using standard parameterization.
<code>unobserve()</code>	
<code>update()</code>	

bayespy.nodes.GaussianWishart.__init__

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

bayespy.nodes.GaussianWishart.add_plate_axis

`GaussianWishart.add_plate_axis(to_plate)`

bayespy.nodes.GaussianWishart.delete

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

bayespy.nodes.GaussianWishart.get_mask

`GaussianWishart.get_mask()`

bayespy.nodes.GaussianWishart.get_moments

`GaussianWishart.get_moments()`

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)`

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(**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_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()`

Attributes

```
dims
plates
```

bayespy.nodes.GaussianWishart.dims

GaussianWishart.**dims** = None

bayespy.nodes.GaussianWishart.plates

GaussianWishart.**plates** = None

Nodes for discrete count variables:

<code>Bernoulli(p, **kwargs)</code>	Node for Bernoulli random variables.
<code>Binomial(n, p, **kwargs)</code>	Node for binomial random variables.
<code>Categorical(p, **kwargs)</code>	Node for categorical random variables.
<code>Multinomial(n, p, **kwargs)</code>	Node for multinomial random variables.
<code>Poisson(l, **kwargs)</code>	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:

$$z \sim \text{Bernoulli}(p).$$

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

<code>__init__(p, **kwargs)</code>	Create Bernoulli node.
<code>add_plate_axis(to_plate)</code>	
<code>delete()</code>	Delete this node and the children
<code>get_mask()</code>	
<code>get_moments()</code>	
<code>get_shape(ind)</code>	
<code>has_plotter()</code>	Return True if the node has a plotter
<code>initialize_from_parameters(*args)</code>	
<code>initialize_from_prior()</code>	
<code>initialize_from_random()</code>	Set the variable to a random sample from the current distribution.
<code>initialize_from_value(x, *args)</code>	
<code>load(group)</code>	Load the state of the node from a HDF5 file.
<code>logpdf(X[, mask])</code>	Compute the log probability density function $Q(X)$ of this node.
<code>lower_bound_contribution([gradient])</code>	
<code>lowerbound()</code>	
<code>move_plates(from_plate, to_plate)</code>	
<code>observe(x, *args[, mask])</code>	Fix moments, compute f and propagate mask.
<code>pdf(X[, mask])</code>	Compute the probability density function of this node.
<code>plot(**kwargs)</code>	Plot the node distribution using the plotter of the node
<code>random()</code>	Draw a random sample from the distribution.
<code>save(group)</code>	Save the state of the node into a HDF5 file.
<code>set_plotter(plotter)</code>	
<code>show()</code>	Print the distribution using standard parameterization.
<code>unobserve()</code>	
<code>update()</code>	

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.delete

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

bayespy.nodes.Bernoulli.get_mask

`Bernoulli.get_mask()`

bayespy.nodes.Bernoulli.get_moments

`Bernoulli.get_moments()`

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)`

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` (***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_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()`

Attributes

<code>dims</code>
<code>plates</code>

`bayespy.nodes.Bernoulli.dims`

`Bernoulli.dims = None`

`bayespy.nodes.Bernoulli.plates`

`Bernoulli.plates = None`

`bayespy.nodes.Binomial`

class `bayespy.nodes.Binomial` (*n*, *p*, ***kwargs*)

Node for binomial random variables.

The node models the number of successes $x \in \{0, \dots, n\}$ in n trials with probability p for success:

$$x \sim \text{Binomial}(n, p).$$

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

<code>__init__(n, p, **kwargs)</code>	Create binomial node
<code>add_plate_axis(to_plate)</code>	
<code>delete()</code>	Delete this node and the children
<code>get_mask()</code>	
<code>get_moments()</code>	
<code>get_shape(ind)</code>	
<code>has_plotter()</code>	Return True if the node has a plotter
<code>initialize_from_parameters(*args)</code>	
<code>initialize_from_prior()</code>	
<code>initialize_from_random()</code>	Set the variable to a random sample from the current distribution.
<code>initialize_from_value(x, *args)</code>	
<code>load(group)</code>	Load the state of the node from a HDF5 file.
<code>logpdf(X[, mask])</code>	Compute the log probability density function $Q(X)$ of this node.
<code>lower_bound_contribution([gradient])</code>	
<code>lowerbound()</code>	
<code>move_plates(from_plate, to_plate)</code>	
<code>observe(x, *args[, mask])</code>	Fix moments, compute f and propagate mask.
<code>pdf(X[, mask])</code>	Compute the probability density function of this node.
<code>plot(**kwargs)</code>	Plot the node distribution using the plotter of the node
<code>random()</code>	Draw a random sample from the distribution.
<code>save(group)</code>	Save the state of the node into a HDF5 file.
<code>set_plotter(plotter)</code>	
<code>show()</code>	Print the distribution using standard parameterization.
<code>unobserve()</code>	
<code>update()</code>	

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.delete

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

bayespy.nodes.Binomial.get_mask

`Binomial.get_mask()`

bayespy.nodes.Binomial.get_moments

`Binomial.get_moments()`

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)`

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` (***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_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()`

Attributes

<code>dims</code>
<code>plates</code>

bayespy.nodes.Binomial.dims

`Binomial.dims = None`

bayespy.nodes.Binomial.plates

`Binomial.plates = None`

bayespy.nodes.Categorical

class `bayespy.nodes.Categorical(p, **kwargs)`

Node for categorical random variables.

The node models a categorical random variable $x \in \{0, \dots, K - 1\}$ with prior probabilities $\{p_0, \dots, p_{K-1}\}$ for each category:

$$p(x = k) = p_k \quad \text{for } k \in \{0, \dots, K - 1\}.$$

Parameters `p` : Dirichlet-like node or (...,K)-array

Probabilities for each category

See also:

`Bernoulli`, `Multinomial`, `Dirichlet`

__init__(`p`, **`kwargs`)
Create Categorical node.

Methods

<code>__init__(p, **kwargs)</code>	Create Categorical node.
<code>add_plate_axis(to_plate)</code>	
<code>delete()</code>	Delete this node and the children
<code>get_mask()</code>	
<code>get_moments()</code>	
<code>get_shape(ind)</code>	
<code>has_plotter()</code>	Return True if the node has a plotter
<code>initialize_from_parameters(*args)</code>	
<code>initialize_from_prior()</code>	
<code>initialize_from_random()</code>	Set the variable to a random sample from the current distribution.

Continued on next page

Table 5.26 – continued from previous page

<code>initialize_from_value(x, *args)</code>	
<code>load(group)</code>	Load the state of the node from a HDF5 file.
<code>logpdf(X[, mask])</code>	Compute the log probability density function $Q(X)$ of this node.
<code>lower_bound_contribution([gradient])</code>	
<code>lowerbound()</code>	
<code>move_plates(from_plate, to_plate)</code>	
<code>observe(x, *args[, mask])</code>	Fix moments, compute f and propagate mask.
<code>pdf(X[, mask])</code>	Compute the probability density function of this node.
<code>plot(**kwargs)</code>	Plot the node distribution using the plotter of the node
<code>random()</code>	Draw a random sample from the distribution.
<code>save(group)</code>	Save the state of the node into a HDF5 file.
<code>set_plotter(plotter)</code>	
<code>show()</code>	Print the distribution using standard parameterization.
<code>unobserve()</code>	
<code>update()</code>	

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.delete

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

bayespy.nodes.Categorical.get_mask

`Categorical.get_mask()`

bayespy.nodes.Categorical.get_moments

`Categorical.get_moments()`

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)`

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` (***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_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` ()

Continued on next page

Table 5.27 – continued from previous page

Attributes

`dims`
`plates`

`bayespy.nodes.Categorical.dims`

`Categorical.dims = None`

`bayespy.nodes.Categorical.plates`

`Categorical.plates = None`

`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, \dots, 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 $x_k \in \{0, \dots, n\}$ in n trials with probability p_k for success in K categories.

$$\text{Multinomial}(\mathbf{x}|N, \mathbf{p}) = \frac{N!}{x_0! \dots x_{K-1}!} p_0^{x_0} \dots p_{K-1}^{x_{K-1}}$$

Parameters *n* : scalar or array

N , number of trials

p : Dirichlet-like node or (...*K*)-array

p, probabilities of successes for the categories

See also:

`Dirichlet`, `Binomial`, `Categorical`

__init__ (*n*, *p*, ***kwargs*)

Create Multinomial node.

Methods

<code>__init__(n, p, **kwargs)</code>	Create Multinomial node.
<code>add_plate_axis(to_plate)</code>	
<code>delete()</code>	Delete this node and the children
<code>get_mask()</code>	
<code>get_moments()</code>	

Continued on next page

Table 5.28 – continued from previous page

<code>get_shape(ind)</code>	
<code>has_plotter()</code>	Return True if the node has a plotter
<code>initialize_from_parameters(*args)</code>	
<code>initialize_from_prior()</code>	
<code>initialize_from_random()</code>	Set the variable to a random sample from the current distribution.
<code>initialize_from_value(x, *args)</code>	
<code>load(group)</code>	Load the state of the node from a HDF5 file.
<code>logpdf(X[, mask])</code>	Compute the log probability density function $Q(X)$ of this node.
<code>lower_bound_contribution([gradient])</code>	
<code>lowerbound()</code>	
<code>move_plates(from_plate, to_plate)</code>	
<code>observe(x, *args[, mask])</code>	Fix moments, compute f and propagate mask.
<code>pdf(X[, mask])</code>	Compute the probability density function of this node.
<code>plot(**kwargs)</code>	Plot the node distribution using the plotter of the node
<code>random()</code>	Draw a random sample from the distribution.
<code>save(group)</code>	Save the state of the node into a HDF5 file.
<code>set_plotter(plotter)</code>	
<code>show()</code>	Print the distribution using standard parameterization.
<code>unobserve()</code>	
<code>update()</code>	

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.delete

`Multinomial.delete()`

Delete this node and the children

bayespy.nodes.Multinomial.get_mask

`Multinomial.get_mask()`

bayespy.nodes.Multinomial.get_moments

`Multinomial.get_moments()`

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)`

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(**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_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()`

Attributes

`dims`
`plates`

`bayespy.nodes.Multinomial.dims`

`Multinomial.dims = None`

`bayespy.nodes.Multinomial.plates`

`Multinomial.plates = None`

`bayespy.nodes.Poisson`

class `bayespy.nodes.Poisson` (*l*, ***kwargs*)

Node for Poisson random variables.

The node uses Poisson distribution:

$$p(x) = \text{Poisson}(x|\lambda)$$

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

<code>__init__(l, **kwargs)</code>	Create Poisson random variable node
<code>add_plate_axis(to_plate)</code>	
<code>delete()</code>	Delete this node and the children
<code>get_mask()</code>	
<code>get_moments()</code>	
<code>get_shape(ind)</code>	
<code>has_plotter()</code>	Return True if the node has a plotter
<code>initialize_from_parameters(*args)</code>	
<code>initialize_from_prior()</code>	
<code>initialize_from_random()</code>	Set the variable to a random sample from the current distribution.
<code>initialize_from_value(x, *args)</code>	
<code>load(group)</code>	Load the state of the node from a HDF5 file.
<code>logpdf(X[, mask])</code>	Compute the log probability density function Q(X) of this node.
<code>lower_bound_contribution([gradient])</code>	

Continued on next page

Table 5.30 – continued from previous page

<code>lowerbound()</code>	
<code>move_plates(from_plate, to_plate)</code>	
<code>observe(x, *args[, mask])</code>	Fix moments, compute f and propagate mask.
<code>pdf(X[, mask])</code>	Compute the probability density function of this node.
<code>plot(**kwargs)</code>	Plot the node distribution using the plotter of the node
<code>random()</code>	Draw a random sample from the distribution.
<code>save(group)</code>	Save the state of the node into a HDF5 file.
<code>set_plotter(plotter)</code>	
<code>show()</code>	Print the distribution using standard parameterization.
<code>unobserve()</code>	
<code>update()</code>	

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.delete

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

bayespy.nodes.Poisson.get_mask

`Poisson.get_mask()`

bayespy.nodes.Poisson.get_moments

`Poisson.get_moments()`

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)`

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 (**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_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()`

Attributes

<code>dims</code>	tuple() -> empty tuple
<code>plates</code>	

bayespy.nodes.Poisson.dims

`Poisson.dims = (),`

bayespy.nodes.Poisson.plates

Poisson.plates = None

Nodes for probabilities:

<code>Beta(alpha, **kwargs)</code>	Node for beta random variables.
<code>Dirichlet(*args, **kwargs)</code>	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

$$p \sim \text{Beta}(a, b)$$

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

<code>__init__(alpha, **kwargs)</code>	Create beta node
<code>add_plate_axis(to_plate)</code>	
<code>delete()</code>	Delete this node and the children
<code>get_mask()</code>	
<code>get_moments()</code>	
<code>get_shape(ind)</code>	
<code>has_plotter()</code>	Return True if the node has a plotter
<code>initialize_from_parameters(*args)</code>	
<code>initialize_from_prior()</code>	
<code>initialize_from_random()</code>	Set the variable to a random sample from the current distribution.
<code>initialize_from_value(x, *args)</code>	
<code>load(group)</code>	Load the state of the node from a HDF5 file.

Continued on next page

Table 5.33 – continued from previous page

<code>logpdf(X[, mask])</code>	Compute the log probability density function $Q(X)$ of this node.
<code>lower_bound_contribution([gradient])</code>	
<code>lowerbound()</code>	
<code>move_plates(from_plate, to_plate)</code>	
<code>observe(x, *args[, mask])</code>	Fix moments, compute f and propagate mask.
<code>pdf(X[, mask])</code>	Compute the probability density function of this node.
<code>plot(**kwargs)</code>	Plot the node distribution using the plotter of the node
<code>random()</code>	Draw a random sample from the distribution.
<code>save(group)</code>	Save the state of the node into a HDF5 file.
<code>set_plotter(plotter)</code>	
<code>show()</code>	Print the distribution using standard parameterization.
<code>unobserve()</code>	
<code>update()</code>	

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.delete

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

bayespy.nodes.Beta.get_mask

`Beta.get_mask()`

bayespy.nodes.Beta.get_moments

`Beta.get_moments()`

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)`

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` (***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_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` ()

Continued on next page

Table 5.34 – continued from previous page

Attributes

`dims`
`plates`

`bayespy.nodes.Beta.dims`

`Beta.dims = None`

`bayespy.nodes.Beta.plates`

`Beta.plates = None`

`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 $\pi_k \in [0, 1] \forall k = 0, \dots, K-1$.

$$p(\pi_0, \dots, \pi_{K-1}) = \text{Dirichlet}(\alpha_0, \dots, \alpha_{K-1})$$

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

<code>__init__(*args, **kwargs)</code>	
<code>add_plate_axis(to_plate)</code>	
<code>delete()</code>	Delete this node and the children
<code>get_mask()</code>	
<code>get_moments()</code>	
<code>get_shape(ind)</code>	
<code>has_plotter()</code>	Return True if the node has a plotter
<code>initialize_from_parameters(*args)</code>	
<code>initialize_from_prior()</code>	
<code>initialize_from_random()</code>	Set the variable to a random sample from the current distribution.

Continued on next page

Table 5.35 – continued from previous page

<code>initialize_from_value(x, *args)</code>	
<code>load(group)</code>	Load the state of the node from a HDF5 file.
<code>logpdf(X[, mask])</code>	Compute the log probability density function $Q(X)$ of this node.
<code>lower_bound_contribution([gradient])</code>	
<code>lowerbound()</code>	
<code>move_plates(from_plate, to_plate)</code>	
<code>observe(x, *args[, mask])</code>	Fix moments, compute f and propagate mask.
<code>pdf(X[, mask])</code>	Compute the probability density function of this node.
<code>plot(**kwargs)</code>	Plot the node distribution using the plotter of the node
<code>random()</code>	Draw a random sample from the distribution.
<code>save(group)</code>	Save the state of the node into a HDF5 file.
<code>set_plotter(plotter)</code>	
<code>show()</code>	Print the distribution using standard parameterization.
<code>unobserve()</code>	
<code>update()</code>	

bayespy.nodes.Dirichlet.__init__

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

bayespy.nodes.Dirichlet.add_plate_axis

`Dirichlet.add_plate_axis(to_plate)`

bayespy.nodes.Dirichlet.delete

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

bayespy.nodes.Dirichlet.get_mask

`Dirichlet.get_mask()`

bayespy.nodes.Dirichlet.get_moments

`Dirichlet.get_moments()`

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)`

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(**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_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()`

Continued on next page

Table 5.36 – continued from previous page

Attributes

`dims`
`plates`

`bayespy.nodes.Dirichlet.dims`

`Dirichlet.dims = None`

`bayespy.nodes.Dirichlet.plates`

`Dirichlet.plates = None`

Nodes for dynamic variables:

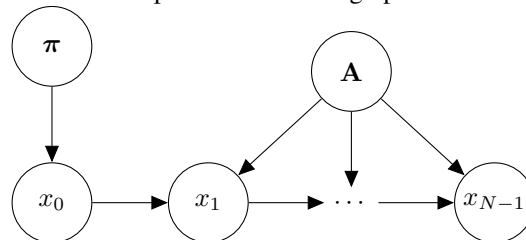
<code>CategoricalMarkovChain(pi, A[, states])</code>	Node for categorical Markov chain random variables.
<code>GaussianMarkovChain(mu, Lambda, A, nu[, n])</code>	Node for Gaussian Markov chain random variables.
<code>SwitchingGaussianMarkovChain(mu, Lambda, B, ...)</code>	Node for Gaussian Markov chain random variables with switching d
<code>VaryingGaussianMarkovChain(mu, Lambda, B, S, nu)</code>	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 π 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.

$$p(x_0, \dots, x_{N-1}) = p(x_0) \prod_{n=1}^{N-1} p(x_n | x_{n-1}),$$

where

$$\begin{aligned} p(x_0 = k) &= \pi_k, \quad \text{for } k \in \{0, \dots, K-1\}, \\ p(x_n = j | x_{n-1} = i) &= a_{ij}^{(n-1)} \quad \text{for } n = 1, \dots, N-1, i \in \{1, \dots, K-1\}, j \in \{1, \dots, K-1\} \\ a_{ij}^{(n)} &= [\mathbf{A}_n]_{ij} \end{aligned}$$

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

π , 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

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

<code>__init__(pi, A[, states])</code>	Create categorical Markov chain
<code>add_plate_axis(to_plate)</code>	
<code>delete()</code>	Delete this node and the children
<code>get_mask()</code>	
<code>get_moments()</code>	
<code>get_shape(ind)</code>	
<code>has_plotter()</code>	Return True if the node has a plotter
<code>initialize_from_parameters(*args)</code>	
<code>initialize_from_prior()</code>	
<code>initialize_from_random()</code>	Set the variable to a random sample from the current distribution.
<code>initialize_from_value(x, *args)</code>	
<code>load(group)</code>	Load the state of the node from a HDF5 file.
<code>logpdf(X[, mask])</code>	Compute the log probability density function $Q(X)$ of this node.
<code>lower_bound_contribution([gradient])</code>	
<code>lowerbound()</code>	
<code>move_plates(from_plate, to_plate)</code>	
<code>observe(x, *args[, mask])</code>	Fix moments, compute f and propagate mask.
<code>pdf(X[, mask])</code>	Compute the probability density function of this node.
<code>plot(**kwargs)</code>	Plot the node distribution using the plotter of the node
<code>random()</code>	Draw a random sample from the distribution.
<code>save(group)</code>	Save the state of the node into a HDF5 file.
<code>set_plotter(plotter)</code>	
<code>show()</code>	Print the distribution using standard parameterization.
<code>unobserve()</code>	
<code>update()</code>	

`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.delete

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

bayespy.nodes.CategoricalMarkovChain.get_mask

`CategoricalMarkovChain.get_mask()`

bayespy.nodes.CategoricalMarkovChain.get_moments

`CategoricalMarkovChain.get_moments()`

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)`

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(**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_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()`

Attributes

`dims`
`plates`

bayespy.nodes.CategoricalMarkovChain.dims

`CategoricalMarkovChain.dims = None`

bayespy.nodes.CategoricalMarkovChain.plates

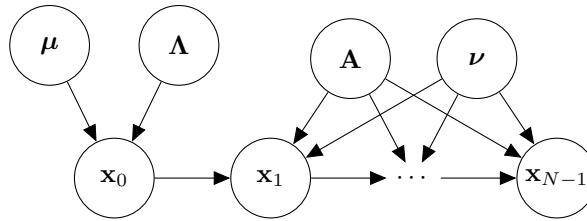
`CategoricalMarkovChain.plates = None`

bayespy.nodes.GaussianMarkovChain

class `bayespy.nodes.GaussianMarkovChain(mu, Lambda, A, nu, n=None, **kwargs)`

Node for Gaussian Markov chain random variables.

In a simple case, the graphical model can be presented as:



where μ and Λ are the mean and the precision matrix of the initial state, \mathbf{A} is the state dynamics matrix and ν is the precision of the innovation noise. It is possible that \mathbf{A} and/or ν 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

$$p(\mathbf{x}_0) = \mathcal{N}(\mathbf{x}_0 | \mu, \Lambda)$$

$$p(\mathbf{x}_n | \mathbf{x}_{n-1}) = \mathcal{N}(\mathbf{x}_n | \mathbf{A}_{n-1} \mathbf{x}_{n-1}, \text{diag}(\nu_{n-1})).$$

Parameters **mu** : Gaussian-like node or (... ,D)-array

μ , mean of x_0 , D -dimensional with plates (...)

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

Λ , 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

ν , 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 ν are constant over time.

See also:

`Gaussian`, `GaussianARD`, `Wishart`, `Gamma`, `SwitchingGaussianMarkovChain`, `VaryingGaussianMarkovChain`, `CategoricalMarkovChain`

__init__ (*mu*, *Lambda*, *A*, *nu*, *n=None*, ***kwargs*)

Create GaussianMarkovChain node.

Methods

<code>__init__(mu, Lambda, A, nu[, n])</code>	Create GaussianMarkovChain node.
<code>add_plate_axis(to_plate)</code>	
<code>delete()</code>	Delete this node and the children
<code>get_mask()</code>	
<code>get_moments()</code>	
<code>get_shape(ind)</code>	
<code>has_plotter()</code>	Return True if the node has a plotter

Continued on next page

Table 5.40 – continued from previous page

<code>initialize_from_parameters(*args)</code>	
<code>initialize_from_prior()</code>	
<code>initialize_from_random()</code>	Set the variable to a random sample from the current distribution.
<code>initialize_from_value(x, *args)</code>	
<code>load(group)</code>	Load the state of the node from a HDF5 file.
<code>logpdf(X[, mask])</code>	Compute the log probability density function $Q(X)$ of this node.
<code>lower_bound_contribution([gradient])</code>	
<code>lowerbound()</code>	
<code>move_plates(from_plate, to_plate)</code>	
<code>observe(x, *args[, mask])</code>	Fix moments, compute f and propagate mask.
<code>pdf(X[, mask])</code>	Compute the probability density function of this node.
<code>plot(**kwargs)</code>	Plot the node distribution using the plotter of the node
<code>random()</code>	Draw a random sample from the distribution.
<code>rotate(R[, inv, logdet])</code>	
<code>save(group)</code>	Save the state of the node into a HDF5 file.
<code>set_plotter(plotter)</code>	
<code>show()</code>	
<code>unobserve()</code>	
<code>update()</code>	

bayespy.nodes.GaussianMarkovChain.__init__

`GaussianMarkovChain.__init__ (mu, Lambda, A, nu, n=None, **kwargs)`
 Create GaussianMarkovChain node.

bayespy.nodes.GaussianMarkovChain.add_plate_axis

`GaussianMarkovChain.add_plate_axis (to_plate)`

bayespy.nodes.GaussianMarkovChain.delete

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

bayespy.nodes.GaussianMarkovChain.get_mask

`GaussianMarkovChain.get_mask ()`

bayespy.nodes.GaussianMarkovChain.get_moments

`GaussianMarkovChain.get_moments ()`

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)`

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(**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_plotter

`GaussianMarkovChain.set_plotter(plotter)`

bayespy.nodes.GaussianMarkovChain.show

`GaussianMarkovChain.show()`

bayespy.nodes.GaussianMarkovChain.unobserve

`GaussianMarkovChain.unobserve()`

bayespy.nodes.GaussianMarkovChain.update

GaussianMarkovChain.update()

Attributes

```
dims
plates
```

bayespy.nodes.GaussianMarkovChain.dims

GaussianMarkovChain.dims = None

bayespy.nodes.GaussianMarkovChain.plates

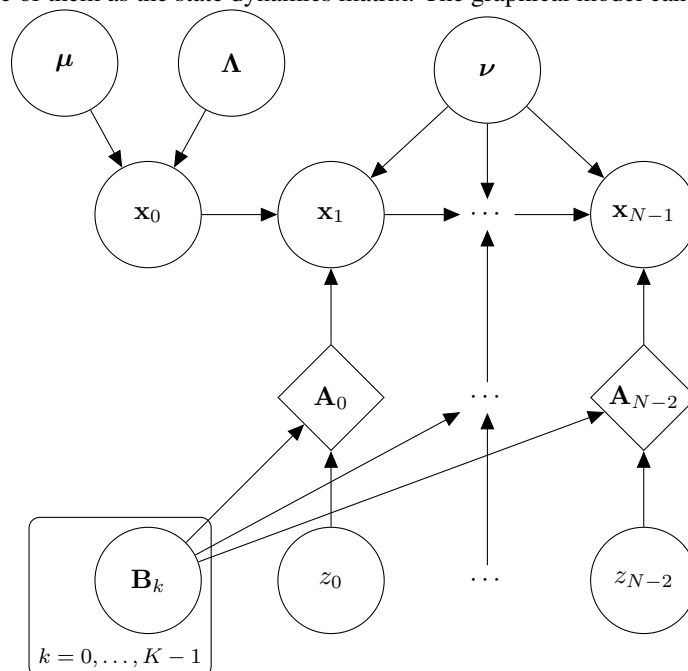
GaussianMarkovChain.plates = None

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 $\mathbf{x}_0, \dots, \mathbf{x}_{N-1}$ 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 μ and Λ are the mean and the precision matrix of the initial state, ν 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 $z_n \in \{0, \dots, K-1\}$. 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

$$\begin{aligned} p(\mathbf{x}_0) &= \mathcal{N}(\mathbf{x}_0 | \boldsymbol{\mu}, \boldsymbol{\Lambda}) \\ p(\mathbf{x}_n | \mathbf{x}_{n-1}) &= \mathcal{N}(\mathbf{x}_n | \mathbf{A}_{n-1} \mathbf{x}_{n-1}, \text{diag}(\boldsymbol{\nu})), \quad \text{for } n = 1, \dots, N-1, \\ \mathbf{A}_n &= \mathbf{B}_{z_n}, \quad \text{for } n = 0, \dots, N-2. \end{aligned}$$

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

$\{z_0, \dots, z_{N-2}\}$, 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

<code>__init__</code> (mu, Lambda, B, Z, nu[, n])	Create SwitchingGaussianMarkovChain node.
<code>add_plate_axis</code> (to_plate)	
<code>delete</code> ()	Delete this node and the children
<code>get_mask</code> ()	
<code>get_moments</code> ()	
<code>get_shape</code> (ind)	

Continued on next page

Table 5.42 – continued from previous page

<code>has_plotter()</code>	Return True if the node has a plotter
<code>initialize_from_parameters(*args)</code>	
<code>initialize_from_prior()</code>	
<code>initialize_from_random()</code>	Set the variable to a random sample from the current distribution.
<code>initialize_from_value(x, *args)</code>	
<code>load(group)</code>	Load the state of the node from a HDF5 file.
<code>logpdf(X[, mask])</code>	Compute the log probability density function $Q(X)$ of this node.
<code>lower_bound_contribution([gradient])</code>	
<code>lowerbound()</code>	
<code>move_plates(from_plate, to_plate)</code>	
<code>observe(x, *args[, mask])</code>	Fix moments, compute f and propagate mask.
<code>pdf(X[, mask])</code>	Compute the probability density function of this node.
<code>plot(**kwargs)</code>	Plot the node distribution using the plotter of the node
<code>random()</code>	Draw a random sample from the distribution.
<code>rotate(R[, inv, logdet])</code>	
<code>save(group)</code>	Save the state of the node into a HDF5 file.
<code>set_plotter(plotter)</code>	
<code>show()</code>	
<code>unobserve()</code>	
<code>update()</code>	

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.delete

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

bayespy.nodes.SwitchingGaussianMarkovChain.get_mask

`SwitchingGaussianMarkovChain.get_mask()`

bayespy.nodes.SwitchingGaussianMarkovChain.get_moments

`SwitchingGaussianMarkovChain.get_moments()`

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)`

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(**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_plotter

`SwitchingGaussianMarkovChain.set_plotter(plotter)`

bayespy.nodes.SwitchingGaussianMarkovChain.show

`SwitchingGaussianMarkovChain.show()`

bayespy.nodes.SwitchingGaussianMarkovChain.unobserve

`SwitchingGaussianMarkovChain.unobserve()`

bayespy.nodes.SwitchingGaussianMarkovChain.update

SwitchingGaussianMarkovChain.update()

Attributes

dims
plates

bayespy.nodes.SwitchingGaussianMarkovChain.dims

SwitchingGaussianMarkovChain.dims = None

bayespy.nodes.SwitchingGaussianMarkovChain.plates

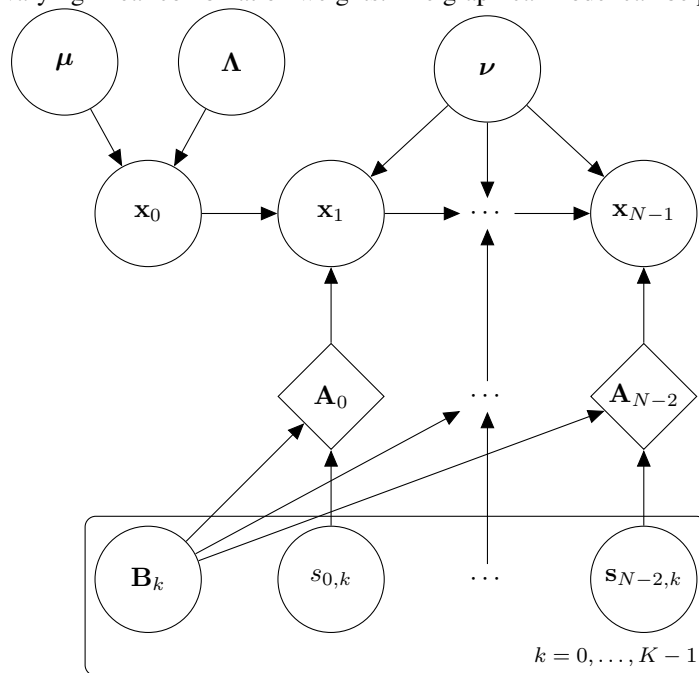
SwitchingGaussianMarkovChain.plates = None

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 $\mathbf{x}_0, \dots, \mathbf{x}_{N-1}$ 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 μ and Λ are the mean and the precision matrix of the initial state, ν 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

$$p(\mathbf{x}_0, \dots, \mathbf{x}_{N-1}) = p(\mathbf{x}_0) \prod_{n=1}^{N-1} p(\mathbf{x}_n | \mathbf{x}_{n-1})$$

where

$$\begin{aligned} p(\mathbf{x}_0) &= \mathcal{N}(\mathbf{x}_0 | \boldsymbol{\mu}, \boldsymbol{\Lambda}) \\ p(\mathbf{x}_n | \mathbf{x}_{n-1}) &= \mathcal{N}(\mathbf{x}_n | \mathbf{A}_{n-1} \mathbf{x}_{n-1}, \text{diag}(\boldsymbol{\nu})), \quad \text{for } n = 1, \dots, N-1, \\ \mathbf{A}_n &= \sum_{k=0}^{K-1} s_{n,k} \mathbf{B}_k, \quad \text{for } n = 0, \dots, N-2. \end{aligned}$$

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)

S : Gaussian-like node or (... ,N-1,K)-array

$\{s_0, \dots, s_{N-2}\}$, 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 **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

[3]

`__init__` (*mu*, *Lambda*, *B*, *S*, *nu*, *n=None*, ***kwargs*)
Create VaryingGaussianMarkovChain node.

Methods

<code>__init__(mu, Lambda, B, S, nu[, n])</code>	Create VaryingGaussianMarkovChain node.
<code>add_plate_axis(to_plate)</code>	
<code>delete()</code>	Delete this node and the children
<code>get_mask()</code>	
<code>get_moments()</code>	
<code>get_shape(ind)</code>	
<code>has_plotter()</code>	Return True if the node has a plotter
<code>initialize_from_parameters(*args)</code>	
<code>initialize_from_prior()</code>	
<code>initialize_from_random()</code>	Set the variable to a random sample from the current distribution.
<code>initialize_from_value(x, *args)</code>	
<code>load(group)</code>	Load the state of the node from a HDF5 file.
<code>logpdf(X[, mask])</code>	Compute the log probability density function Q(X) of this node.
<code>lower_bound_contribution([gradient])</code>	
<code>lowerbound()</code>	
<code>move_plates(from_plate, to_plate)</code>	
<code>observe(x, *args[, mask])</code>	Fix moments, compute f and propagate mask.
<code>pdf(X[, mask])</code>	Compute the probability density function of this node.
<code>plot(**kwargs)</code>	Plot the node distribution using the plotter of the node
<code>random()</code>	Draw a random sample from the distribution.
<code>rotate(R[, inv, logdet])</code>	
<code>save(group)</code>	Save the state of the node into a HDF5 file.
<code>set_plotter(plotter)</code>	
<code>show()</code>	
<code>unobserve()</code>	
<code>update()</code>	

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.delete

VaryingGaussianMarkovChain.**delete** ()
 Delete this node and the children

bayespy.nodes.VaryingGaussianMarkovChain.get_mask

VaryingGaussianMarkovChain.**get_mask** ()

bayespy.nodes.VaryingGaussianMarkovChain.get_moments

VaryingGaussianMarkovChain.**get_moments** ()

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)

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** (***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_plotter

VaryingGaussianMarkovChain.**set_plotter** (*plotter*)

bayespy.nodes.VaryingGaussianMarkovChain.show

VaryingGaussianMarkovChain.**show**()

bayespy.nodes.VaryingGaussianMarkovChain.unobserve

VaryingGaussianMarkovChain.unobserve()

bayespy.nodes.VaryingGaussianMarkovChain.update

VaryingGaussianMarkovChain.update()

Attributes

dims plates

bayespy.nodes.VaryingGaussianMarkovChain.dims

VaryingGaussianMarkovChain.dims = None

bayespy.nodes.VaryingGaussianMarkovChain.plates

VaryingGaussianMarkovChain.plates = None

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

$$p(x|z = k, \theta_0, \dots, \theta_{K-1}) = \phi(x|\theta_k),$$

where ϕ is the probability density function of the mixed exponential family distribution and $\theta_0, \dots, \theta_{K-1}$ are the parameters of each cluster. For instance, ϕ could be the Gaussian probability density function \mathcal{N} and $\theta_k = \{\mu_k, \Lambda_k\}$ where μ_k and Λ_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

<code>__init__(z, node_class, *params[, cluster_plate])</code>	
<code>add_plate_axis(to_plate)</code>	
<code>delete()</code>	Delete this node and the children
<code>get_mask()</code>	
<code>get_moments()</code>	
<code>get_shape(ind)</code>	
<code>has_plotter()</code>	Return True if the node has a plotter
<code>initialize_from_parameters(*args)</code>	
<code>initialize_from_prior()</code>	
<code>initialize_from_random()</code>	Set the variable to a random sample from the current distribution.
<code>initialize_from_value(x, *args)</code>	
<code>integrated_logpdf_from_parents(x, index)</code>	Approximates the posterior predictive pdf $\int p(x parents) q(parents) dparents$
<code>load(group)</code>	Load the state of the node from a HDF5 file.
<code>logpdf(X[, mask])</code>	Compute the log probability density function $Q(X)$ of this node.
<code>lower_bound_contribution([gradient])</code>	
<code>lowerbound()</code>	
<code>move_plates(from_plate, to_plate)</code>	
<code>observe(x, *args[, mask])</code>	Fix moments, compute f and propagate mask.
<code>pdf(X[, mask])</code>	Compute the probability density function of this node.
<code>plot(**kwargs)</code>	Plot the node distribution using the plotter of the node
<code>random()</code>	Draw a random sample from the distribution.
<code>save(group)</code>	Save the state of the node into a HDF5 file.
<code>set_plotter(plotter)</code>	
<code>unobserve()</code>	
<code>update()</code>	

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.delete

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

bayespy.nodes.Mixture.get_mask

`Mixture.get_mask()`

bayespy.nodes.Mixture.get_moments

`Mixture.get_moments()`

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 $\int q(parents_i) \exp(\int q(\text{parents}_i) \log p(x|parents_i) dparents_i)$.

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)`

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(**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_plotter

`Mixture.set_plotter(plotter)`

bayespy.nodes.Mixture.unobserve

`Mixture.unobserve()`

bayespy.nodes.Mixture.update

`Mixture.update()`

Attributes

<code>dims</code>
<code>plates</code>

bayespy.nodes.Mixture.dims

`Mixture.dims = None`

bayespy.nodes.Mixture.plates

`Mixture.plates = None`

5.1.2 Deterministic nodes

<code>Dot(*args, **kwargs)</code>	Node for computing inner product of several Gaussian vectors.
<code>SumMultiply(*args[, iterator_axis])</code>	Node for computing general products and sums of Gaussian nodes.
<code>Gate(Z, X[, gated_plate, moments])</code>	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

<code>__init__(Node1, map1, Node2, map2, ..., ...)</code>	
<code>add_plate_axis(to_plate)</code>	
<code>delete()</code>	Delete this node and the children
<code>get_mask()</code>	
<code>get_moments()</code>	
<code>get_parameters()</code>	
<code>get_shape(ind)</code>	
<code>has_plotter()</code>	Return True if the node has a plotter
<code>lower_bound_contribution([gradient])</code>	
<code>move_plates(from_plate, to_plate)</code>	
<code>plot(**kwargs)</code>	Plot the node distribution using the plotter of the node
<code>set_plotter(plotter)</code>	

`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.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)`

bayespy.nodes.SumMultiply.move_plates

`SumMultiply.move_plates(from_plate, to_plate)`

bayespy.nodes.SumMultiply.plot

`SumMultiply.plot(**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`

bayespy.nodes.SumMultiply.plates

`SumMultiply.plates = None`

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

<code>__init__(Z, X[, gated_plate, moments])</code>	
<code>add_plate_axis(to_plate)</code>	
<code>delete()</code>	Delete this node and the children
<code>get_mask()</code>	
<code>get_moments()</code>	
<code>get_shape(ind)</code>	
<code>has_plotter()</code>	Return True if the node has a plotter
<code>lower_bound_contribution([gradient])</code>	
<code>move_plates(from_plate, to_plate)</code>	
<code>plot(**kwargs)</code>	Plot the node distribution using the plotter of the node
<code>set_plotter(plotter)</code>	

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.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)`

bayespy.nodes.Gate.move_plates

`Gate.move_plates(from_plate, to_plate)`

bayespy.nodes.Gate.plot

`Gate.plot(**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`

bayespy.nodes.Gate.plates

`Gate.plates = None`

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

<code>__init__(*nodes[, tol, autosave_filename, ...])</code>	
<code>compute_lowerbound()</code>	
<code>compute_lowerbound_terms(*nodes)</code>	
<code>get_iteration_by_nodes()</code>	
<code>load(*nodes[, filename])</code>	
<code>loglikelihood_lowerbound()</code>	
<code>plot(*nodes)</code>	Plot the distribution of the given nodes (or all nodes)
<code>plot_iteration_by_nodes()</code>	Plot the cost function per node during the iteration.
<code>save([filename])</code>	
<code>set_autosave(filename[, iterations])</code>	
<code>update(*nodes[, repeat, plot, tol, verbose])</code>	

bayespy.inference.VB.__init__

VB.__init__ (*nodes, tol=1e-05, autosave_filename=None, autosave_iterations=0, callback=None)

bayespy.inference.VB.compute_lowerbound

VB.compute_lowerbound()

bayespy.inference.VB.compute_lowerbound_terms

VB.compute_lowerbound_terms (*nodes)

bayespy.inference.VB.get_iteration_by_nodes

VB.get_iteration_by_nodes ()

bayespy.inference.VB.load

VB.load (*nodes, filename=None)

bayespy.inference.VB.loglikelihood_lowerbound

VB.loglikelihood_lowerbound ()

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 ()
Plot the cost function per node during the iteration.
Handy tool for debugging.

bayespy.inference.VB.save

VB.save (filename=None)

bayespy.inference.VB.set_autosave

VB.set_autosave (filename, iterations=None)

bayespy.inference.VB.update

VB.update (*nodes, repeat=1, plot=False, tol=None, verbose=True)

Table 5.56 – continued from previous page

5.2.2 Parameter expansions

<code>vmp.transformations.RotationOptimizer(...)</code>	Optimizer for rotation parameter expansion in state-space models
<code>vmp.transformations.RotateGaussian(X)</code>	Rotation parameter expansion for <code>bayespy.nodes.VaryingGaussian</code>
<code>vmp.transformations.RotateGaussianARD(X, *alpha)</code>	Rotation parameter expansion for <code>bayespy.nodes.VaryingGaussian</code>
<code>vmp.transformations.RotateGaussianMarkovChain(X, ...)</code>	Rotation parameter expansion for <code>bayespy.nodes.VaryingGaussian</code>
<code>vmp.transformations.RotateSwitchingMarkovChain(X, ...)</code>	Rotation for <code>bayespy.nodes.SwitchingGaussian</code>
<code>vmp.transformations.RotateVaryingMarkovChain(X, ...)</code>	Rotation for <code>bayespy.nodes.SwitchingGaussian</code>
<code>vmp.transformations.RotateMultiple(*rotators)</code>	Identical parameter expansion for several nodes simultaneously

`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

[2], [1]

`__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*)

Rotation parameter expansion for bayespy.nodes.GaussianARD

The model:

$\alpha \sim N(a, b)$ $X \sim N(\mu, \alpha)$

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*)

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__ (<i>X</i> , * <i>alpha</i> [, <i>axis</i> , <i>precompute</i>])	Precompute tells whether to compute some moments once in the setup function instead
bound (<i>R</i> [, <i>logdet</i> , <i>inv</i> , <i>Q</i>])	
get_bound_terms (<i>R</i> [, <i>logdet</i> , <i>inv</i> , <i>Q</i>])	
nodes ()	
rotate (<i>R</i> [, <i>inv</i> , <i>logdet</i> , <i>Q</i>])	
setup ([<i>plate_axis</i>])	This method should be called just before optimization.

bayespy.inference.vmp.transformations.RotateGaussianARD.__init__

RotateGaussianARD.__init__ (*X*, **alpha*, *axis=-1*, *precompute=False*)

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

<code>__init__</code> (<i>X</i> , * <i>args</i>)	
<code>bound</code> (<i>R</i> [, <i>logdet</i> , <i>inv</i>])	
<code>get_bound_terms</code> (<i>R</i> [, <i>logdet</i> , <i>inv</i>])	
<code>nodes</code> ()	
<code>rotate</code> (<i>R</i> [, <i>inv</i> , <i>logdet</i>])	
<code>setup</code> ()	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

<code>__init__(X, B, Z, B_rotator)</code>	
<code>bound(R[, logdet, inv])</code>	
<code>get_bound_terms(R[, logdet, inv])</code>	
<code>nodes()</code>	
<code>rotate(R[, inv, logdet])</code>	
<code>setup()</code>	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

<code>__init__(X, B, S, B_rotator)</code>	
<code>bound(R[, logdet, inv])</code>	
<code>get_bound_terms(R[, logdet, inv])</code>	
<code>nodes()</code>	
<code>rotate(R[, inv, logdet])</code>	
<code>setup()</code>	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

<code>pdf</code> (<i>Z</i> , <i>x</i> , * <i>args</i> [, <i>name</i>])	Plot probability density function of a scalar variable.
<code>contour</code> (<i>Z</i> , <i>x</i> , <i>y</i> [, <i>n</i>])	Plot 2-D probability density function of a 2-D variable.
<code>plot</code> (<i>Y</i> [, <i>axis</i> , <i>scale</i> , <i>center</i>])	Plot a variable or an array as 1-D function with errorbars
<code>hinton</code> (<i>X</i> , ** <i>kwargs</i>)	Plot the Hinton diagram of a node
<code>gaussian_mixture</code> (<i>X</i> [, <i>scale</i> , <i>fill</i>])	Plot Gaussian mixture as ellipses in 2-D

`bayespy.plot.pdf`

`bayespy.plot.pdf` (*Z*, *x*, **args*, *name=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*, ***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*, ***kwargs*)
 Plot Gaussian mixture as ellipses in 2-D

5.3.2 Plotters

<code>Plotter</code> (<i>plotter</i> , <i>*args</i> , <i>**kwargs</i>)	Wrapper for plotting functions and base class for node plotters
<code>PDFPlotter</code> (<i>x_grid</i> , <i>**kwargs</i>)	Plotter of probability density function of a scalar node
<code>ContourPlotter</code> (<i>x1_grid</i> , <i>x2_grid</i> , <i>**kwargs</i>)	Plotter of probability density function of a two-dimensional node
<code>HintonPlotter</code> (<i>**kwargs</i>)	Plotter of the Hinton diagram of a node
<code>FunctionPlotter</code> (<i>**kwargs</i>)	Plotter of a node as a 1-dimensional function
<code>GaussianTimeseriesPlotter</code> (<i>**kwargs</i>)	Plotter of a Gaussian node as a timeseries
<code>CategoricalMarkovChainPlotter</code> (<i>**kwargs</i>)	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)

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:

<code>node.Node(*parents, **kwargs)</code>	Base class for all nodes.
<code>stochastic.Stochastic(*args[, initialize, dims])</code>	Base class for nodes that are stochastic.
<code>expfamily.ExponentialFamily(*args, **kwargs)</code>	A base class for nodes using natural parameterization ϕ .
<code>deterministic.Deterministic(*args, **kwargs)</code>	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

<code>__init__(*parents, **kwargs)</code>	
<code>add_plate_axis(to_plate)</code>	
<code>delete()</code>	Delete this node and the children
<code>get_mask()</code>	
<code>get_moments()</code>	

Continued on next page

Table 6.2 – continued from previous page

<code>get_shape(ind)</code>	
<code>has_plotter()</code>	Return True if the node has a plotter
<code>move_plates(from_plate, to_plate)</code>	
<code>plot(**kwargs)</code>	Plot the node distribution using the plotter of the node
<code>set_plotter(plotter)</code>	

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.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 (**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

bayespy.inference.vmp.nodes.node.Node.plates

Node.**plates** = None

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) _plates_to_parent(self, index) _plates_from_parent(self, index)

__init__ (*args, initialize=True, dims=None, **kwargs)

Methods

__init__(*args[, initialize, dims])	
add_plate_axis(to_plate)	
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(**kwargs)	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()	

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.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(**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()`

Attributes

`plates`

bayespy.inference.vmp.nodes.stochastic.Stochastic.plates

`Stochastic.plates = None`

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 ϕ .

ϕ

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) _plates_to_parent(self, index) _plates_from_parent(self, index)

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

Methods

<code>__init__(*args, **kwargs)</code>	
<code>add_plate_axis(to_plate)</code>	
<code>delete()</code>	Delete this node and the children
<code>get_mask()</code>	
<code>get_moments()</code>	
<code>get_shape(ind)</code>	
<code>has_plotter()</code>	Return True if the node has a plotter
<code>initialize_from_parameters(*args)</code>	
<code>initialize_from_prior()</code>	
<code>initialize_from_random()</code>	Set the variable to a random sample from the current distribution.
<code>initialize_from_value(x, *args)</code>	
<code>load(group)</code>	Load the state of the node from a HDF5 file.
<code>logpdf(X[, mask])</code>	Compute the log probability density function Q(X) of this node.
<code>lower_bound_contribution([gradient])</code>	
<code>lowerbound()</code>	
<code>move_plates(from_plate, to_plate)</code>	
<code>observe(x, *args[, mask])</code>	Fix moments, compute f and propagate mask.
<code>pdf(X[, mask])</code>	Compute the probability density function of this node.
<code>plot(**kwargs)</code>	Plot the node distribution using the plotter of the node
<code>random()</code>	Draw a random sample from the distribution.
<code>save(group)</code>	Save the state of the node into a HDF5 file.
<code>set_plotter(plotter)</code>	
<code>unobserve()</code>	
<code>update()</code>	

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.delete

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

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_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)`

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 (**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_plotter

`ExponentialFamily.set_plotter (plotter)`

bayespy.inference.vmp.nodes.expfamily.ExponentialFamily.unobserve

`ExponentialFamily.unobserve()`

bayespy.inference.vmp.nodes.expfamily.ExponentialFamily.update

ExponentialFamily.update()

Attributes

dims
plates

bayespy.inference.vmp.nodes.expfamily.ExponentialFamily.dims

ExponentialFamily.dims = None

bayespy.inference.vmp.nodes.expfamily.ExponentialFamily.plates

ExponentialFamily.plates = None

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) _plates_to_parent(self, index) _plates_from_parent(self, index)

__init__(*args, **kwargs)

Methods

__init__(*args, **kwargs)	
add_plate_axis(to_plate)	
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])	

Continued on next page

Table 6.8 – continued from previous page

<code>move_plates(from_plate, to_plate)</code>	
<code>plot(**kwargs)</code>	Plot the node distribution using the plotter of the node
<code>set_plotter(plotter)</code>	

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.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)`

bayespy.inference.vmp.nodes.deterministic.Deterministic.move_plates

`Deterministic.move_plates(from_plate, to_plate)`

bayespy.inference.vmp.nodes.deterministic.Deterministic.plot

`Deterministic.plot` (***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`

bayespy.inference.vmp.nodes.deterministic.Deterministic.plates

`Deterministic.plates` = `None`

The following nodes are examples of special nodes that remain hidden for the user although they are often implicitly used:

<code>constant.Constant</code> (<i>moments</i> , <i>x</i> , <i>**kwargs</i>)	Node for presenting constant values.
<code>gaussian.GaussianToGaussianGammaISO</code> (<i>X</i> , <i>**kwargs</i>)	Converter for Gaussian moments to Gaussian-gamma isotropic
<code>gaussian.GaussianGammaISOToGaussianGammaARD</code> (<i>X</i> , ...)	Converter for Gaussian-gamma ISO moments to Gaussian-gamma ARD
<code>gaussian.GaussianGammaARDToGaussianWishart</code> (...)	
<code>gaussian.WrapToGaussianGammaISO</code> (<i>*parents</i> , ...)	
<code>gaussian.WrapToGaussianGammaARD</code> (<i>mu_alpha</i> , ...)	
<code>gaussian.WrapToGaussianWishart</code> (<i>X</i> , <i>Lambda</i> , ...)	Wraps Gaussian and Wishart nodes into a Gaussian-Wishart node

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

<code>__init__</code> (<i>moments</i> , <i>x</i> , <i>**kwargs</i>)	
<code>add_plate_axis</code> (<i>to_plate</i>)	
<code>delete</code> ()	Delete this node and the children
<code>get_mask</code> ()	
<code>get_moments</code> ()	
<code>get_shape</code> (<i>ind</i>)	

Continued on next page

Table 6.11 – continued from previous page

<code>has_plotter()</code>	Return True if the node has a plotter
<code>move_plates(from_plate, to_plate)</code>	
<code>plot(**kwargs)</code>	Plot the node distribution using the plotter of the node
<code>set_plotter(plotter)</code>	

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.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.move_plates

`Constant.move_plates` (*from_plate, to_plate*)

bayespy.inference.vmp.nodes.constant.Constant.plot

`Constant.plot` (***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)`

Attributes

`plates`

bayespy.inference.vmp.nodes.constant.Constant.plates

`Constant.plates = None`

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

<code>__init__(X, **kwargs)</code>	
<code>add_plate_axis(to_plate)</code>	
<code>delete()</code>	Delete this node and the children
<code>get_mask()</code>	
<code>get_moments()</code>	
<code>get_shape(ind)</code>	
<code>has_plotter()</code>	Return True if the node has a plotter
<code>lower_bound_contribution([gradient])</code>	
<code>move_plates(from_plate, to_plate)</code>	
<code>plot(**kwargs)</code>	Plot the node distribution using the plotter of the node
<code>set_plotter(plotter)</code>	

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.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)`

bayespy.inference.vmp.nodes.gaussian.GaussianToGaussianGammaISO.move_plates

`GaussianToGaussianGammaISO.move_plates(from_plate, to_plate)`

bayespy.inference.vmp.nodes.gaussian.GaussianToGaussianGammaISO.plot

`GaussianToGaussianGammaISO.plot(**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`

bayespy.inference.vmp.nodes.gaussian.GaussianToGaussianGammaISO.plates

`GaussianToGaussianGammaISO.plates = None`

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__ (<i>X</i> , <i>**kwargs</i>)	
add_plate_axis (<i>to_plate</i>)	
delete ()	Delete this node and the children
get_mask ()	
get_moments ()	
get_shape (<i>ind</i>)	
has_plotter ()	Return True if the node has a plotter
lower_bound_contribution ([<i>gradient</i>])	
move_plates (<i>from_plate</i> , <i>to_plate</i>)	
plot (<i>**kwargs</i>)	Plot the node distribution using the plotter of the node
set_plotter (<i>plotter</i>)	

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.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_contribution

`GaussianGammaISOToGaussianGammaARD.lower_bound_contribution(gradient=False)`

bayespy.inference.vmp.nodes.gaussian.GaussianGammaISOToGaussianGammaARD.move_plates

`GaussianGammaISOToGaussianGammaARD.move_plates(from_plate, to_plate)`

bayespy.inference.vmp.nodes.gaussian.GaussianGammaISOToGaussianGammaARD.plot

`GaussianGammaISOToGaussianGammaARD.plot(**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`

bayespy.inference.vmp.nodes.gaussian.GaussianGammaISOToGaussianGammaARD.plates

`GaussianGammaISOToGaussianGammaARD.plates = None`

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)`
`delete()`

Delete this node and the children

Continued on next page

Table 6.17 – continued from previous page

<code>get_mask()</code>	
<code>get_moments()</code>	
<code>get_shape(ind)</code>	
<code>has_plotter()</code>	Return True if the node has a plotter
<code>lower_bound_contribution([gradient])</code>	
<code>move_plates(from_plate, to_plate)</code>	
<code>plot(**kwargs)</code>	Plot the node distribution using the plotter of the node
<code>set_plotter(plotter)</code>	

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.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)`

bayespy.inference.vmp.nodes.gaussian.GaussianGammaARDToGaussianWishart.move_plates

`GaussianGammaARDToGaussianWishart.move_plates(from_plate, to_plate)`

bayespy.inference.vmp.nodes.gaussian.GaussianGammaARDToGaussianWishart.plot

GaussianGammaARDToGaussianWishart.**plot** (***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`

bayespy.inference.vmp.nodes.gaussian.GaussianGammaARDToGaussianWishart.plates

GaussianGammaARDToGaussianWishart.**plates** = None

6.1.9 bayespy.inference.vmp.nodes.gaussian.WrapToGaussianGammaISO

class bayespy.inference.vmp.nodes.gaussian.**WrapToGaussianGammaISO** (**parents*, ***kwargs*)

__init__ (**parents*, ***kwargs*)

Methods

<code>__init__(*parents, **kwargs)</code>	
<code>add_plate_axis(to_plate)</code>	
<code>delete()</code>	Delete this node and the children
<code>get_mask()</code>	
<code>get_moments()</code>	
<code>get_shape(ind)</code>	
<code>has_plotter()</code>	Return True if the node has a plotter
<code>lower_bound_contribution([gradient])</code>	
<code>move_plates(from_plate, to_plate)</code>	
<code>plot(**kwargs)</code>	Plot the node distribution using the plotter of the node
<code>set_plotter(plotter)</code>	

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.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)`

bayespy.inference.vmp.nodes.gaussian.WrapToGaussianGammaISO.move_plates

`WrapToGaussianGammaISO.move_plates(from_plate, to_plate)`

bayespy.inference.vmp.nodes.gaussian.WrapToGaussianGammaISO.plot

`WrapToGaussianGammaISO.plot(**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`

bayespy.inference.vmp.nodes.gaussian.WrapToGaussianGammaISO.plates

WrapToGaussianGammaISO.**plates** = None

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

<code>__init__(mu_alpha, tau, **kwargs)</code>	
<code>add_plate_axis(to_plate)</code>	
<code>delete()</code>	Delete this node and the children
<code>get_mask()</code>	
<code>get_moments()</code>	
<code>get_shape(ind)</code>	
<code>has_plotter()</code>	Return True if the node has a plotter
<code>lower_bound_contribution([gradient])</code>	
<code>move_plates(from_plate, to_plate)</code>	
<code>plot(**kwargs)</code>	Plot the node distribution using the plotter of the node
<code>set_plotter(plotter)</code>	

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.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)`

bayespy.inference.vmp.nodes.gaussian.WrapToGaussianGammaARD.move_plates

`WrapToGaussianGammaARD.move_plates(from_plate, to_plate)`

bayespy.inference.vmp.nodes.gaussian.WrapToGaussianGammaARD.plot

`WrapToGaussianGammaARD.plot(**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`

bayespy.inference.vmp.nodes.gaussian.WrapToGaussianGammaARD.plates

`WrapToGaussianGammaARD.plates = None`

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

<code>__init__(X, Lambda, **kwargs)</code>	
<code>add_plate_axis(to_plate)</code>	
<code>delete()</code>	Delete this node and the children
<code>get_mask()</code>	
<code>get_moments()</code>	
<code>get_shape(ind)</code>	
<code>has_plotter()</code>	Return True if the node has a plotter
<code>lower_bound_contribution([gradient])</code>	
<code>move_plates(from_plate, to_plate)</code>	
<code>plot(**kwargs)</code>	Plot the node distribution using the plotter of the node
<code>set_plotter(plotter)</code>	

`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.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)`

bayespy.inference.vmp.nodes.gaussian.WrapToGaussianWishart.move_plates

`WrapToGaussianWishart.move_plates(from_plate, to_plate)`

bayespy.inference.vmp.nodes.gaussian.WrapToGaussianWishart.plot

`WrapToGaussianWishart.plot(**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`

bayespy.inference.vmp.nodes.gaussian.WrapToGaussianWishart.plates

`WrapToGaussianWishart.plates = None`

6.2 Moments

<code>node.Moments</code>	Base class for defining the expectation of the s
<code>gaussian.GaussianMoments(ndim)</code>	Class for the moments of Gaussian variables.
<code>gaussian_markov_chain.GaussianMarkovChainMoments</code>	
<code>gaussian.GaussianGammaISOMoments(ndim)</code>	Class for the moments of Gaussian-gamma-IS
<code>gaussian.GaussianGammaARDMoments(ndim)</code>	Class for the moments of Gaussian-gamma-AR
<code>gaussian.GaussianWishartMoments</code>	Class for the moments of Gaussian-Wishart va
<code>gamma.GammaMoments</code>	Class for the moments of gamma variables.
<code>wishart.WishartMoments</code>	
<code>beta.BetaMoments</code>	Class for the moments of beta variables.
<code>dirichlet.DirichletMoments</code>	Class for the moments of Dirichlet variables.

Continu

Table 6.25 – continued from previous page

<code>bernoulli.BernoulliMoments()</code>	Class for the moments of Bernoulli variables.
<code>binomial.BinomialMoments(N)</code>	Class for the moments of binomial variables
<code>categorical.CategoricalMoments(categories)</code>	Class for the moments of categorical variables
<code>categorical_markov_chain.CategoricalMarkovChainMoments(...)</code>	Class for the moments of categorical Markov c
<code>multinomial.MultinomialMoments</code>	Class for the moments of multinomial variable
<code>poisson.PoissonMoments</code>	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

<code>add_converter(moments_to, converter)</code>	
<code>compute_dims_from_values(x)</code>	
<code>compute_fixed_moments(x)</code>	
<code>get_converter(moments_to)</code>	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

<code>__init__(ndim)</code>	
<code>add_converter(moments_to, converter)</code>	
<code>compute_dims_from_values(x)</code>	Return the shape of the moments for a fixed value.
<code>compute_fixed_moments(x)</code>	Compute the moments for a fixed value
<code>get_converter(moments_to)</code>	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

`GaussianMarkovChainMoments.compute_dims_from_values(x)`

bayespy.inference.vmp.nodes.gaussian_markov_chain.GaussianMarkovChainMoments.compute_fixed_mom

`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__ (<i>ndim</i>)	Create moments object for Gaussian-gamma isotropic variables
add_converter (<i>moments_to</i> , <i>converter</i>)	
compute_dims_from_values (<i>x</i> , <i>alpha</i>)	Return the shape of the moments for a fixed value.
compute_fixed_moments (<i>x</i> , <i>alpha</i>)	Compute the moments for a fixed value
get_converter (<i>moments_to</i>)	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__ (<i>ndim</i>)	Create moments object for Gaussian-gamma isotropic variables
add_converter (<i>moments_to</i> , <i>converter</i>)	
compute_dims_from_values (<i>x</i> , <i>alpha</i>)	Return the shape of the moments for a fixed value.
compute_fixed_moments (<i>x</i> , <i>alpha</i>)	Compute the moments for a fixed value
get_converter (<i>moments_to</i>)	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

<code>add_converter(moments_to, converter)</code>	
<code>compute_dims_from_values(x, Lambda)</code>	Return the shape of the moments for a fixed value.
<code>compute_fixed_moments(x, Lambda)</code>	Compute the moments for a fixed value
<code>get_converter(moments_to)</code>	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

<code>add_converter(moments_to, converter)</code>	
<code>compute_dims_from_values(x)</code>	Return the shape of the moments for a fixed value.
<code>compute_fixed_moments(x)</code>	Compute the moments for a fixed value
<code>get_converter(moments_to)</code>	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

<code>add_converter(moments_to, converter)</code>	
<code>compute_dims_from_values(x)</code>	Compute the dimensions of phi and u.
<code>compute_fixed_moments(Lambda)</code>	Compute moments for fixed x.
<code>get_converter(moments_to)</code>	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

<code>add_converter(moments_to, converter)</code>	
<code>compute_dims_from_values(p)</code>	Return the shape of the moments for a fixed value.
<code>compute_fixed_moments(p)</code>	Compute the moments for a fixed value
<code>get_converter(moments_to)</code>	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

<code>add_converter(moments_to, converter)</code>	
<code>compute_dims_from_values(x)</code>	Return the shape of the moments for a fixed value.
<code>compute_fixed_moments(p)</code>	Compute the moments for a fixed value
<code>get_converter(moments_to)</code>	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

<code>__init__()</code>	
<code>add_converter(moments_to, converter)</code>	
<code>compute_dims_from_values(x)</code>	Return the shape of the moments for a fixed value.
<code>compute_fixed_moments(x)</code>	Compute the moments for a fixed value
<code>get_converter(moments_to)</code>	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

<code>__init__(N)</code>	
<code>add_converter(moments_to, converter)</code>	
<code>compute_dims_from_values(x)</code>	Return the shape of the moments for a fixed value.
<code>compute_fixed_moments(x)</code>	Compute the moments for a fixed value
<code>get_converter(moments_to)</code>	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

<code>__init__(categories)</code>	Create moments object for categorical variables
<code>add_converter(moments_to, converter)</code>	
<code>compute_dims_from_values(x)</code>	Return the shape of the moments for a fixed value.
<code>compute_fixed_moments(x)</code>	Compute the moments for a fixed value
<code>get_converter(moments_to)</code>	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.CategoricalMarkovChainMoments

class `bayespy.inference.vmp.nodes.categorical_markov_chain.CategoricalMarkovChainMoments(categories)`

Class for the moments of categorical Markov chain variables.

__init__(categories)

Create moments object for categorical Markov chain variables.

Methods

<code>__init__(categories)</code>	Create moments object for categorical Markov chain variables.
<code>add_converter(moments_to, converter)</code>	
<code>compute_dims_from_values(x)</code>	Return the shape of the moments for a fixed value.
<code>compute_fixed_moments(x)</code>	Compute the moments for a fixed value
<code>get_converter(moments_to)</code>	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_from_values(x)

`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_moments(x)

`CategoricalMarkovChainMoments.compute_fixed_moments(x)`

Compute the moments for a fixed value

bayespy.inference.vmp.nodes.categorical_markov_chain.CategoricalMarkovChainMoments.get_converter(moments_to)

`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

<code>add_converter(moments_to, converter)</code>	
<code>compute_dims_from_values(x)</code>	Return the shape of the moments for a fixed value.
<code>compute_fixed_moments(x)</code>	Compute the moments for a fixed value
<code>get_converter(moments_to)</code>	Finds conversion to another moments type if possible.

bayespy.inference.vmp.nodes.multinomial.MultinomialMoments.add_converter(moments_to, converter)

`MultinomialMoments.add_converter(moments_to, converter)`

bayespy.inference.vmp.nodes.multinomial.MultinomialMoments.compute_dims_from_values(x)

`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

<code>add_converter(moments_to, converter)</code>	
<code>compute_dims_from_values(x)</code>	Return the shape of the moments for a fixed value.
<code>compute_fixed_moments(x)</code>	Compute the moments for a fixed value
<code>get_converter(moments_to)</code>	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

<code>stochastic.Distribution</code>	A base class for the VMP for
<code>expfamily.ExponentialFamilyDistribution</code>	Sub-classes implement distri
<code>gaussian.GaussianDistribution</code>	Class for the VMP formulas
<code>gaussian.GaussianARDDistribution(shape, ndim_mu)</code>	...
<code>gaussian.GaussianGammaISODistribution</code>	Class for the VMP formulas
<code>gaussian.GaussianGammaARDDistribution()</code>	
<code>gaussian.GaussianWishartDistribution</code>	Class for the VMP formulas
<code>gaussian_markov_chain.GaussianMarkovChainDistribution(N, D)</code>	Sub-classes implement distri
<code>gaussian_markov_chain.SwitchingGaussianMarkovChainDistribution(N, D, K)</code>	Sub-classes implement distri
<code>gaussian_markov_chain.VaryingGaussianMarkovChainDistribution(N, D)</code>	Sub-classes implement distri
<code>gamma.GammaDistribution</code>	Class for the VMP formulas
<code>wishart.WishartDistribution</code>	Sub-classes implement distri
<code>beta.BetaDistribution</code>	Class for the VMP formulas
<code>dirichlet.DirichletDistribution</code>	Class for the VMP formulas
<code>bernoulli.BernoulliDistribution()</code>	Class for the VMP formulas
<code>binomial.BinomialDistribution(N)</code>	Class for the VMP formulas
<code>categorical.CategoricalDistribution(categories)</code>	Class for the VMP formulas
<code>categorical_markov_chain.CategoricalMarkovChainDistribution(...)</code>	Class for the VMP formulas
<code>multinomial.MultinomialDistribution(trials)</code>	Class for the VMP formulas
<code>poisson.PoissonDistribution</code>	Class for the VMP formulas

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

<code>compute_mask_to_parent(index, mask)</code>	Maps the mask to the plates of a parent.
<code>compute_message_to_parent(parent, index, ...)</code>	Compute the message to a parent node.
<code>plates_from_parent(index, plates)</code>	Resolve the plate mapping from a parent.
<code>plates_to_parent(index, plates)</code>	Resolves the plate mapping to a parent.
<code>random(*params[, plates])</code>	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

<code>compute_cgf_from_parents(*u_parents)</code>	
<code>compute_fixed_moments_and_f(x[, mask])</code>	
<code>compute_logpdf(u, phi, g, f, ndims)</code>	Compute $E[\log p(X)]$ given $E[u]$, $E[\phi]$, $E[g]$ and $E[f]$.
<code>compute_mask_to_parent(index, mask)</code>	Maps the mask to the plates of a parent.
<code>compute_message_to_parent(parent, index, ...)</code>	
<code>compute_moments_and_cgf(phi[, mask])</code>	
<code>compute_phi_from_parents(*u_parents[, mask])</code>	
<code>plates_from_parent(index, plates)</code>	Resolve the plate mapping from a parent.
<code>plates_to_parent(index, plates)</code>	Resolves the plate mapping to a parent.
<code>random(*params[, plates])</code>	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_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:

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

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

$$\log \mathcal{N}(\mathbf{x} | \boldsymbol{\mu}, \boldsymbol{\Lambda}) = -\frac{1}{2} \mathbf{x}^T \boldsymbol{\Lambda} \mathbf{x} + \mathbf{x}^T \boldsymbol{\Lambda} \boldsymbol{\mu} - \frac{1}{2} \boldsymbol{\mu}^T \boldsymbol{\Lambda} \boldsymbol{\mu} + \frac{1}{2} \log |\boldsymbol{\Lambda}| - \frac{D}{2} \log(2\pi)$$

$$\begin{aligned}
 \mathbf{u}(\mathbf{x}) &= \begin{bmatrix} \mathbf{x} \\ \mathbf{x}\mathbf{x}^T \end{bmatrix} \\
 \phi(\boldsymbol{\mu}, \boldsymbol{\Lambda}) &= \begin{bmatrix} \boldsymbol{\Lambda}\boldsymbol{\mu} \\ -\frac{1}{2}\boldsymbol{\Lambda} \end{bmatrix} \\
 \phi_{\boldsymbol{\mu}}(\mathbf{x}, \boldsymbol{\Lambda}) &= \begin{bmatrix} \boldsymbol{\Lambda}\mathbf{x} \\ -\frac{1}{2}\boldsymbol{\Lambda} \end{bmatrix} \\
 \phi_{\boldsymbol{\Lambda}}(\mathbf{x}, \boldsymbol{\mu}) &= \begin{bmatrix} -\frac{1}{2}\mathbf{x}\mathbf{x}^T + \frac{1}{2}\mathbf{x}\boldsymbol{\mu}^T + \frac{1}{2}\boldsymbol{\mu}\mathbf{x}^T - \frac{1}{2}\boldsymbol{\mu}\boldsymbol{\mu}^T \\ \frac{1}{2} \end{bmatrix} \\
 g(\boldsymbol{\mu}, \boldsymbol{\Lambda}) &= -\frac{1}{2} \text{tr}(\boldsymbol{\mu}\boldsymbol{\mu}^T \boldsymbol{\Lambda}) + \frac{1}{2} \log |\boldsymbol{\Lambda}| \\
 g_{\phi}(\phi) &= \frac{1}{4} \phi_1^T \phi_2^{-1} \phi_1 + \frac{1}{2} \log | -2\phi_2 | \\
 f(\mathbf{x}) &= -\frac{D}{2} \log(2\pi) \\
 \bar{\mathbf{u}}(\phi) &= \begin{bmatrix} -\frac{1}{2}\phi_2^{-1}\phi_1 \\ \frac{1}{4}\phi_2^{-1}\phi_1\phi_1^T\phi_2^{-1} - \frac{1}{2}\phi_2^{-1} \end{bmatrix}
 \end{aligned}$$

`__init__()`
Initialize self. See help(type(self)) for accurate signature.

Methods

<code>compute_cgf_from_parents(u_mu_Lambda)</code>	Compute $E_{q(p)}[g(p)]$
<code>compute_fixed_moments_and_f(x[, mask])</code>	Compute the moments and $f(x)$ for a fixed value.
<code>compute_logpdf(u, phi, g, f, ndims)</code>	Compute $E[\log p(X)]$ given $E[u]$, $E[\phi]$, $E[g]$ and $E[f]$.
<code>compute_mask_to_parent(index, mask)</code>	Maps the mask to the plates of a parent.
<code>compute_message_to_parent(parent, index, u, ...)</code>	Compute the message to a parent node.
<code>compute_moments_and_cgf(phi[, mask])</code>	Compute the moments and $g(\phi)$.
<code>compute_phi_from_parents(u_mu_Lambda[, mask])</code>	Compute the natural parameter vector given parent moments.
<code>plates_from_parent(index, plates)</code>	Resolve the plate mapping from a parent.
<code>plates_to_parent(index, plates)</code>	Resolves the plate mapping to a parent.
<code>random(*phi[, plates])</code>	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 $E_{q(p)}[g(p)]$

`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.

`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.

bayespy.inference.vmp.nodes.gaussian.GaussianDistribution.compute_moments_and_cgf

`GaussianDistribution.compute_moments_and_cgf` (*phi, mask=True*)
 Compute the moments and $g(\phi)$.

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.

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:

$$\log p(x|\mu, \alpha) = -\frac{1}{2}x^T \text{diag}(\alpha)x + x^T \text{diag}(\alpha)\mu - \frac{1}{2}\mu^T \text{diag}(\alpha)\mu + \frac{1}{2} \sum_i \log \alpha_i - \frac{D}{2} \log(2\pi)$$

Parent has moments:

$$\begin{bmatrix} \alpha \circ \mu \\ \alpha \circ \mu \circ \mu \\ \alpha \\ \log(\alpha) \end{bmatrix}$$

`__init__` (*shape*, *ndim_mu*)

Methods

<code>__init__</code> (<i>shape</i> , <i>ndim_mu</i>)	
<code>compute_cgf_from_parents</code> (<i>u_mu_alpha</i>)	Compute the value of the cumulant generating function.
<code>compute_fixed_moments_and_f</code> (<i>x</i> [, <i>mask</i>])	Compute <i>u</i> (<i>x</i>) and <i>f</i> (<i>x</i>) for given <i>x</i> .
<code>compute_logpdf</code> (<i>u</i> , <i>phi</i> , <i>g</i> , <i>f</i> , <i>ndims</i>)	Compute <i>E</i> [log <i>p</i> (<i>X</i>)] given <i>E</i> [<i>u</i>], <i>E</i> [<i>phi</i>], <i>E</i> [<i>g</i>] and <i>E</i> [<i>f</i>].
<code>compute_mask_to_parent</code> (<i>index</i> , <i>mask</i>)	Maps the mask to the plates of a parent.
<code>compute_message_to_parent</code> (<i>parent</i> , <i>index</i> , <i>u</i> , ...)	...
<code>compute_moments_and_cgf</code> (<i>phi</i> [, <i>mask</i>])	
<code>compute_phi_from_parents</code> (<i>u_mu_alpha</i> [, <i>mask</i>])	
<code>plates_from_parent</code> (<i>index</i> , <i>plates</i>)	Resolve the plate mapping from a parent.
<code>plates_to_parent</code> (<i>index</i> , <i>plates</i>)	Resolves the plate mapping to a parent.
<code>random</code> (* <i>phi</i> [, <i>plates</i>])	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_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*)
...

$$m = \begin{bmatrix} x \\ [-\frac{1}{2}, \dots, -\frac{1}{2}] \\ -\frac{1}{2} \text{diag}(xx^T) \\ [\frac{1}{2}, \dots, \frac{1}{2}] \end{bmatrix}$$

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

<code>compute_cgf_from_parents(u_mu_Lambda, u_a, u_b)</code>	Compute $E_{q(p)}[g(p)]$
<code>compute_fixed_moments_and_f(x, alpha[, mask])</code>	Compute the moments and $f(x)$ for a fixed value.
<code>compute_logpdf(u, phi, g, f, ndims)</code>	Compute $E[\log p(X)]$ given $E[u]$, $E[\phi]$, $E[g]$ and $E[f]$.
<code>compute_mask_to_parent(index, mask)</code>	Maps the mask to the plates of a parent.
<code>compute_message_to_parent(parent, index, u, ...)</code>	Compute the message to a parent node.
<code>compute_moments_and_cgf(phi[, mask])</code>	Compute the moments and $g(\phi)$.
<code>compute_phi_from_parents(u_mu_Lambda, u_a, u_b)</code>	Compute the natural parameter vector given parent moments.
<code>plates_from_parent(index, plates)</code>	Resolve the plate mapping from a parent.
<code>plates_to_parent(index, plates)</code>	Resolves the plate mapping to a parent.
<code>random(*params[, plates])</code>	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 $E_{q(p)}[g(p)]$

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_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

<code>__init__()</code>	
<code>compute_cgf_from_parents(*u_parents)</code>	
<code>compute_fixed_moments_and_f(x[, mask])</code>	
<code>compute_logpdf(u, phi, g, f, ndims)</code>	Compute $E[\log p(X)]$ given $E[u]$, $E[\phi]$, $E[g]$ and $E[f]$.
<code>compute_mask_to_parent(index, mask)</code>	Maps the mask to the plates of a parent.
<code>compute_message_to_parent(parent, index, ...)</code>	
<code>compute_moments_and_cgf(phi[, mask])</code>	
<code>compute_phi_from_parents(*u_parents[, mask])</code>	
<code>plates_from_parent(index, plates)</code>	Resolve the plate mapping from a parent.
<code>plates_to_parent(index, plates)</code>	Resolves the plate mapping to a parent.
<code>random(*params[, plates])</code>	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_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

<code>compute_cgf_from_parents(u_mu_alpha, u_V, u_n)</code>	Compute $E_{q(p)}[g(p)]$
<code>compute_fixed_moments_and_f(x, Lambda[, mask])</code>	Compute the moments and $f(x)$ for a fixed value.
<code>compute_logpdf(u, phi, g, f, ndims)</code>	Compute $E[\log p(X)]$ given $E[u]$, $E[\phi]$, $E[g]$ and $E[f]$.
<code>compute_mask_to_parent(index, mask)</code>	Maps the mask to the plates of a parent.
<code>compute_message_to_parent(parent, index, u, ...)</code>	Compute the message to a parent node.
<code>compute_moments_and_cgf(phi[, mask])</code>	Compute the moments and $g(\phi)$.
<code>compute_phi_from_parents(u_mu_alpha, u_V, u_n)</code>	Compute the natural parameter vector given parent moments.
<code>plates_from_parent(index, plates)</code>	Resolve the plate mapping from a parent.
<code>plates_to_parent(index, plates)</code>	Resolves the plate mapping to a parent.
<code>random(*params[, plates])</code>	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 $E_{q(p)}[g(p)]$

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_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

<code>__init__(N, D)</code>	
<code>compute_cgf_from_parents(u_mu, u_Lambda, ...)</code>	Compute CGF using the moments of the parents.
<code>compute_fixed_moments_and_f(x[, mask])</code>	Compute $u(x)$ and $f(x)$ for given x .
<code>compute_logpdf(u, phi, g, f, ndims)</code>	Compute $E[\log p(X)]$ given $E[u]$, $E[\phi]$, $E[g]$ and $E[f]$.
<code>compute_mask_to_parent(index, mask)</code>	
<code>compute_message_to_parent(parent, index, u, ...)</code>	Compute a message to a parent.
<code>compute_moments_and_cgf(phi[, mask])</code>	Compute the moments and the cumulant-generating function.
<code>compute_phi_from_parents(u_mu, u_Lambda, ...)</code>	Compute the natural parameters using parents' moments.

Continued on next page

Table 6.50 – continued from previous page

<code>plates_from_parent(index, plates)</code>	Compute the plates using information of a parent node.
<code>plates_to_parent(index, plates)</code>	Computes the plates of this node with respect to a parent.
<code>random(*params[, plates])</code>	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

`GaussianMarkovChainDistribution.compute_cgf_from_parents(u_mu, u_Lambda, u_A, u_v)`
 Compute CGF using the moments of the parents.

bayespy.inference.vmp.nodes.gaussian_markov_chain.GaussianMarkovChainDistribution.compute_fixed_moments_and_f

`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_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

`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)`
 Compute a message to a parent.

bayespy.inference.vmp.nodes.gaussian_markov_chain.GaussianMarkovChainDistribution.compute_moments_and_cgf

`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

GaussianMarkovChainDistribution.**compute_phi_from_parents** (*u_mu, u_Lambda, u_A, 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.GaussianMarkovChainDistribution.plates_from_paren

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.SwitchingGaussianMarkovChainD

class bayespy.inference.vmp.nodes.gaussian_markov_chain.**SwitchingGaussianMarkovChainDistribut**

Sub-classes implement distribution specific computations.

__init__ (*N, D, K*)

Methods

<code>__init__(N, D, K)</code>	
<code>compute_cgf_from_parents(u_mu, u_Lambda, ...)</code>	Compute CGF using the moments of the parents.
<code>compute_fixed_moments_and_f(x[, mask])</code>	Compute $u(x)$ and $f(x)$ for given x .
<code>compute_logpdf(u, phi, g, f, ndims)</code>	Compute $E[\log p(X)]$ given $E[u]$, $E[\phi]$, $E[g]$ and $E[f]$.
<code>compute_mask_to_parent(index, mask)</code>	
<code>compute_message_to_parent(parent, index, u, ...)</code>	Compute a message to a parent.
<code>compute_moments_and_cgf(phi[, mask])</code>	Compute the moments and the cumulant-generating function.
<code>compute_phi_from_parents(u_mu, u_Lambda, ...)</code>	Compute the natural parameters using parents' moments.
<code>plates_from_parent(index, plates)</code>	Compute the plates using information of a parent node.
<code>plates_to_parent(index, plates)</code>	Computes the plates of this node with respect to a parent.
<code>random(*params[, plates])</code>	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_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_fr

```
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.VaryingGaussianMarkovChainDistribution`

Sub-classes implement distribution specific computations.

`__init__` (*N, D*)

Methods

<code>__init__(N, D)</code>	
<code>compute_cgf_from_parents(u_mu, u_Lambda, ...)</code>	Compute CGF using the moments of the parents.
<code>compute_fixed_moments_and_f(x[, mask])</code>	Compute u(x) and f(x) for given x.
<code>compute_logpdf(u, phi, g, f, ndims)</code>	Compute $E[\log p(X)]$ given $E[u]$, $E[\phi]$, $E[g]$ and $E[f]$.
<code>compute_mask_to_parent(index, mask)</code>	
<code>compute_message_to_parent(parent, index, u, ...)</code>	Compute a message to a parent.
<code>compute_moments_and_cgf(phi[, mask])</code>	Compute the moments and the cumulant-generating function.
<code>compute_phi_from_parents(u_mu, u_Lambda, ...)</code>	Compute the natural parameters using parents' moments.
<code>plates_from_parent(index, plates)</code>	Compute the plates using information of a parent node.
<code>plates_to_parent(index, plates)</code>	Computes the plates of this node with respect to a parent.
<code>random(*params[, plates])</code>	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_c

`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_f

`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_logpdf

`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_mask_to_parent

`VaryingGaussianMarkovChainDistribution.compute_mask_to_parent(index, mask)`

bayespy.inference.vmp.nodes.gaussian_markov_chain.VaryingGaussianMarkovChainDistribution.compute_message_to_parent

`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_moments_and_cgf

`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_phi_from_parents

`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

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

<code>compute_cgf_from_parents(*u_parents)</code>	Compute $E_{q(p)}[g(p)]$
<code>compute_fixed_moments_and_f(x[, mask])</code>	Compute the moments and $f(x)$ for a fixed value.
<code>compute_logpdf(u, phi, g, f, ndims)</code>	Compute $E[\log p(X)]$ given $E[u]$, $E[\phi]$, $E[g]$ and $E[f]$.
<code>compute_mask_to_parent(index, mask)</code>	Maps the mask to the plates of a parent.
<code>compute_message_to_parent(parent, index, ...)</code>	Compute the message to a parent node.
<code>compute_moments_and_cgf(phi[, mask])</code>	Compute the moments and $g(\phi)$.
<code>compute_phi_from_parents(*u_parents[, mask])</code>	Compute the natural parameter vector given parent moments.
<code>plates_from_parent(index, plates)</code>	Resolve the plate mapping from a parent.

Continued on next page

Table 6.53 – continued from previous page

<code>plates_to_parent(index, plates)</code>	Resolves the plate mapping to a parent.
<code>random(*phi[, plates])</code>	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 $E_{q(p)}[g(p)]$

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_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 $g(\phi)$.

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

<code>compute_cgf_from_parents(*u_parents)</code>	
<code>compute_fixed_moments_and_f(Lambda[, mask])</code>	Compute $u(x)$ and $f(x)$ for given x .
<code>compute_logpdf(u, phi, g, f, ndims)</code>	Compute $E[\log p(X)]$ given $E[u]$, $E[\phi]$, $E[g]$ and $E[f]$.
<code>compute_mask_to_parent(index, mask)</code>	Maps the mask to the plates of a parent.
<code>compute_message_to_parent(parent, index, ...)</code>	
<code>compute_moments_and_cgf(phi[, mask])</code>	
<code>compute_phi_from_parents(*u_parents[, mask])</code>	
<code>plates_from_parent(index, plates)</code>	Resolve the plate mapping from a parent.
<code>plates_to_parent(index, plates)</code>	Resolves the plate mapping to a parent.
<code>random(*params[, plates])</code>	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_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

<code>compute_cgf_from_parents(u_alpha)</code>	Compute $E_{q(p)}[g(p)]$
<code>compute_fixed_moments_and_f(p[, mask])</code>	Compute the moments and $f(x)$ for a fixed value.
<code>compute_logpdf(u, phi, g, f, ndims)</code>	Compute $E[\log p(X)]$ given $E[u]$, $E[\phi]$, $E[g]$ and $E[f]$.
<code>compute_mask_to_parent(index, mask)</code>	Maps the mask to the plates of a parent.
<code>compute_message_to_parent(parent, index, ...)</code>	Compute the message to a parent node.
<code>compute_moments_and_cgf(phi[, mask])</code>	Compute the moments and $g(\phi)$.
<code>compute_phi_from_parents(u_alpha[, mask])</code>	Compute the natural parameter vector given parent moments.
<code>plates_from_parent(index, plates)</code>	Resolve the plate mapping from a parent.
<code>plates_to_parent(index, plates)</code>	Resolves the plate mapping to a parent.
<code>random(*phi[, plates])</code>	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 $E_{q(p)}[g(p)]$

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_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 $E_{q(p)}[g(p)]$
compute_fixed_moments_and_f(x[, mask])	Compute the moments and $f(x)$ for a fixed value.
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 $E_{q(p)}[g(p)]$

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_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 $g(\phi)$.

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 $E_{q(p)}[g(p)]$
compute_fixed_moments_and_f(x[, mask])	Compute the moments and $f(x)$ for a fixed value.
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)	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 $E_{q(p)}[g(p)]$

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_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)`
 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

<code>__init__(N)</code>	
<code>compute_cgf_from_parents(u_p)</code>	Compute $E_{q(p)}[g(p)]$
<code>compute_fixed_moments_and_f(x[, mask])</code>	Compute the moments and $f(x)$ for a fixed value.
<code>compute_logpdf(u, phi, g, f, ndims)</code>	Compute $E[\log p(X)]$ given $E[u]$, $E[\phi]$, $E[g]$ and $E[f]$.
<code>compute_mask_to_parent(index, mask)</code>	Maps the mask to the plates of a parent.
<code>compute_message_to_parent(parent, index, ...)</code>	Compute the message to a parent node.
<code>compute_moments_and_cgf(phi[, mask])</code>	Compute the moments and $g(\phi)$.
<code>compute_phi_from_parents(u_p[, mask])</code>	Compute the natural parameter vector given parent moments.
<code>plates_from_parent(index, plates)</code>	Resolve the plate mapping from a parent.

Continued on next page

Table 6.58 – continued from previous page

<code>plates_to_parent(index, plates)</code>	Resolves the plate mapping to a parent.
<code>random(*phi)</code>	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 $E_{q(p)}[g(p)]$

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_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)`

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

<code>__init__(categories)</code>	Create VMP formula node for a categorical variable
<code>compute_cgf_from_parents(u_p)</code>	Compute $E_{q(p)}[g(p)]$
<code>compute_fixed_moments_and_f(x[, mask])</code>	Compute the moments and $f(x)$ for a fixed value.
<code>compute_logpdf(u, phi, g, f, ndims)</code>	Compute $E[\log p(X)]$ given $E[u]$, $E[\phi]$, $E[g]$ and $E[f]$.
<code>compute_mask_to_parent(index, mask)</code>	Maps the mask to the plates of a parent.
<code>compute_message_to_parent(parent, index, u, u_p)</code>	Compute the message to a parent node.
<code>compute_moments_and_cgf(phi[, mask])</code>	Compute the moments and $g(\phi)$.
<code>compute_phi_from_parents(u_p[, mask])</code>	Compute the natural parameter vector given parent moments.
<code>plates_from_parent(index, plates)</code>	Resolve the plate mapping from a parent.
<code>plates_to_parent(index, plates)</code>	Resolves the plate mapping to a parent.
<code>random(*phi[, plates])</code>	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 $E_{q(p)}[g(p)]$

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_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.CategoricalMarkovChainDistribution

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.

Methods

<code>__init__(categories, states)</code>	Create VMP formula node for a categorical variable
<code>compute_cgf_from_parents(u_p0, u_P)</code>	Compute $E_{q(p)}[g(p)]$
<code>compute_fixed_moments_and_f(x[, mask])</code>	Compute the moments and $f(x)$ for a fixed value.
<code>compute_logpdf(u, phi, g, f, ndims)</code>	Compute $E[\log p(X)]$ given $E[u]$, $E[\phi]$, $E[g]$ and $E[f]$.
<code>compute_mask_to_parent(index, mask)</code>	Maps the mask to the plates of a parent.
<code>compute_message_to_parent(parent, index, u, ...)</code>	Compute the message to a parent node.
<code>compute_moments_and_cgf(phi[, mask])</code>	Compute the moments and $g(\phi)$.
<code>compute_phi_from_parents(u_p0, u_P[, mask])</code>	Compute the natural parameter vector given parent moments.
<code>plates_from_parent(index, plates)</code>	Resolve the plate mapping from a parent.
<code>plates_to_parent(index, plates)</code>	Resolves the plate mapping to a parent.
<code>random(*phi[, plates])</code>	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_t

`CategoricalMarkovChainDistribution.compute_cgf_from_parents(u_p0, u_P)`
 Compute $E_{q(p)}[g(p)]$

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_logp

`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_message

`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_moments

`CategoricalMarkovChainDistribution.compute_moments_and_cgf` (*phi, mask=True*)
 Compute the moments and $g(\phi)$.

bayespy.inference.vmp.nodes.categorical_markov_chain.CategoricalMarkovChainDistribution.compute_phi

`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_parents

`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_parents

`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__ (<i>trials</i>)	Create VMP formula node for a multinomial variable
compute_cgf_from_parents (<i>u_p</i>)	Compute $E_{q(p)}[g(p)]$
compute_fixed_moments_and_f (<i>x</i> , <i>mask</i>)	Compute the moments and $f(x)$ for a fixed value.
compute_logpdf (<i>u</i> , <i>phi</i> , <i>g</i> , <i>f</i> , <i>ndims</i>)	Compute $E[\log p(X)]$ given $E[u]$, $E[\phi]$, $E[g]$ and $E[f]$.
compute_mask_to_parent (<i>index</i> , <i>mask</i>)	Maps the mask to the plates of a parent.
compute_message_to_parent (<i>parent</i> , <i>index</i> , <i>u</i> , <i>u_p</i>)	Compute the message to a parent node.
compute_moments_and_cgf (<i>phi</i> , <i>mask</i>)	Compute the moments and $g(\phi)$.
compute_phi_from_parents (<i>u_p</i> , <i>mask</i>)	Compute the natural parameter vector given parent moments.
plates_from_parent (<i>index</i> , <i>plates</i>)	Resolve the plate mapping from a parent.
plates_to_parent (<i>index</i> , <i>plates</i>)	Resolves the plate mapping to a parent.
random (* <i>phi</i>)	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 $E_{q(p)}[g(p)]$

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_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

<code>compute_cgf_from_parents(u_lambda)</code>	Compute $E_{q(p)}[g(p)]$
<code>compute_fixed_moments_and_f(x[, mask])</code>	Compute the moments and $f(x)$ for a fixed value.
<code>compute_logpdf(u, phi, g, f, ndims)</code>	Compute $E[\log p(X)]$ given $E[u]$, $E[\phi]$, $E[g]$ and $E[f]$.

Continued on next page

Table 6.62 – continued from previous page

<code>compute_mask_to_parent(index, mask)</code>	Maps the mask to the plates of a parent.
<code>compute_message_to_parent(parent, index, u, ...)</code>	Compute the message to a parent node.
<code>compute_moments_and_cgf(phi[, mask])</code>	Compute the moments and $g(\phi)$.
<code>compute_phi_from_parents(u_lambda[, mask])</code>	Compute the natural parameter vector given parent moments.
<code>plates_from_parent(index, plates)</code>	Resolve the plate mapping from a parent.
<code>plates_to_parent(index, plates)</code>	Resolves the plate mapping to a parent.
<code>random(*phi)</code>	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 $E_{q(p)}[g(p)]$

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_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

<code>linalg</code>	General numerical functions and methods.
<code>random</code>	General functions random sampling and distributions.
<code>optimize</code>	
<code>misc</code>	General numerical functions and methods.

6.4.1 bayespy.utils.linalg

General numerical functions and methods.

Functions

<code>block_banded_solve(A, B, y)</code>	Invert symmetric, banded, positive-definite matrix.
<code>chol(C)</code>	
<code>chol_inv(U)</code>	
<code>chol_logdet(U)</code>	
<code>chol_solve(U, b[, out, matrix])</code>	
<code>dot(*arrays)</code>	Compute matrix-matrix product.
<code>inner(*args[, ndim])</code>	Compute inner product.
<code>inv(A[, ndim])</code>	General array inversion.
<code>logdet_chol(U)</code>	
<code>logdet_cov(C)</code>	
<code>logdet_tri(R)</code>	Logarithm of the absolute value of the determinant of a triangular matrix.
<code>m_dot(A, b)</code>	
<code>mmdot(A, B)</code>	Compute matrix-matrix product.
<code>mvdot(A, b)</code>	Compute matrix-vector product.

Continued on next page

Table 6.64 – continued from previous page

<code>outer(A, B[, ndim])</code>	Computes outer product over the last axes of A and B.
<code>solve_triangular(U, B, **kwargs)</code>	
<code>tracedot(A, B)</code>	Computes trace(A*B).

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: $A_1 * A_2 * A_3 * \dots * A_N$. 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)`

Compute matrix-matrix product.

Applies broadcasting.

bayespy.utils.linalg.mvdot

`bayespy.utils.linalg.mvdot(A, b)`

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).

6.4.2 bayespy.utils.random

General functions random sampling and distributions.

Functions

<code>alpha_beta_recursion(logp0, logP)</code>	Compute alpha-beta recursion for Markov chain
<code>bernoulli(p[, size])</code>	Draw random samples from the Bernoulli distribution.
<code>categorical(p[, size])</code>	Draw random samples from a categorical distribution.
<code>correlation(D)</code>	Draw a random correlation matrix.
<code>covariance(D[, size])</code>	Draw a random covariance matrix.
<code>dirichlet(alpha[, size])</code>	Draw random samples from the Dirichlet distribution.
<code>gamma_entropy(a, log_b, gammaln_a, psi_a, ...)</code>	Entropy of $\mathcal{G}(a, b)$.
<code>gamma_logpdf(bx, logx, a_logx, a_logb, gammaln_a)</code>	Log-density of $\mathcal{G}(x a, b)$.
<code>gaussian_entropy(logdet_V, D)</code>	Compute the entropy of a Gaussian distribution.
<code>gaussian_gamma_to_t(mu, Cov, a, b[, ndim])</code>	Integrates gamma distribution to obtain parameters of t distribution
<code>gaussian_logpdf(yVy, yVmu, muVmu, logdet_V, D)</code>	Log-density of a Gaussian distribution.
<code>intervals(N, length[, amount, gap])</code>	Return random non-overlapping parts of a sequence.
<code>invwishart_rand(nu, V)</code>	
<code>mask(*shape[, p])</code>	Return a boolean array of the given shape.
<code>orth(D)</code>	Draw random orthogonal matrix.
<code>sphere([N])</code>	Draw random points uniformly on a unit sphere.
<code>svd(s)</code>	Draw a random matrix given its singular values.
<code>t_logpdf(z2, logdet_cov, nu, D)</code>	
<code>wishart_rand(nu, V)</code>	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:

$$\log p0 = \log P(z_0) + \log P(y_0|z_0) \quad \log P[...n,:]= \log P(z_{n+1}|z_n) + \log P(y_{n+1}|z_{n+1})$$

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.

$C \sim \text{Inv-W}(I, D)$

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.

$\mathcal{G}(y|\mu, V^{-1})$

Parameters *yVy* : ndarray or double

$y^T V y$

yVmu : ndarray or double

$y^T V \mu$

muVmu : ndarray or double

$$\mu^T \mathbf{V} \mu$$

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, **11**, **21**, 3, 4, 5, **16**, **71**, 8, 9, **110**, **111**, **112**, **131**, 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.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

<code>check_gradient(f, x0[, verbose])</code>	Simple wrapper for SciPy's gradient checker.
<code>minimize(f, x0[, maxiter, verbose])</code>	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

<code>T(X)</code>	Transpose the matrix.
<code>add_axes(X[, num, axis])</code>	
<code>add_leading_axes(x, n)</code>	
<code>add_trailing_axes(x, n)</code>	
<code>array_to_scalar(x)</code>	
<code>atleast_nd(X, d)</code>	
<code>axes_to_collapse(shape_x, shape_to)</code>	
<code>block_banded(D, B)</code>	Construct a symmetric block-banded matrix.

Continued on next page

Table 6.67 – continued from previous page

<code>broadcasted_shape(*shapes)</code>	Computes the resulting broadcasted shape for a given set of shapes.
<code>broadcasted_shape_from_arrays(*arrays)</code>	Computes the resulting broadcasted shape for a given set of arrays.
<code>ceildiv(a, b)</code>	Compute a divided by b and rounded up.
<code>check_gradient(x0, f, df, eps)</code>	
<code>chol(C)</code>	
<code>chol_inv(U)</code>	
<code>chol_logdet(U)</code>	
<code>chol_solve(U, b)</code>	
<code>cholesky(K)</code>	
<code>composite_function(function_list)</code>	Construct a function composition from a list of functions.
<code>diag(X[, ndim])</code>	Create a diagonal array given the diagonal elements.
<code>diagonal(A)</code>	
<code>dist_haversine(c1, c2[, radius])</code>	
<code>first(L)</code>	
<code>gaussian_logpdf(y_invcov_y, y_invcov_mu, ...)</code>	
<code>get_diag(X[, ndim])</code>	Get the diagonal of an array.
<code>grid(x1, x2)</code>	Returns meshgrid as a (M*N,2)-shape array.
<code>identity(*shape)</code>	
<code>is_callable(f)</code>	
<code>is_numeric(a)</code>	
<code>is_shape_subset(sub_shape, full_shape)</code>	
<code>is_string(s)</code>	
<code>isinteger(x)</code>	
<code>kalman_filter(y, U, A, V, mu0, Cov0[, out])</code>	Perform Kalman filtering to obtain filtered mean and covariance.
<code>logdet_chol(U)</code>	
<code>logsumexp(X[, axis, keepdims])</code>	Compute $\log(\sum(\exp(X)))$ in a numerically stable way
<code>m_chol(C)</code>	
<code>m_chol_inv(U)</code>	
<code>m_chol_logdet(U)</code>	
<code>m_chol_solve(U, B[, out])</code>	
<code>m_digamma(a, d)</code>	
<code>m_dot(A, b)</code>	
<code>m_outer(A, B)</code>	
<code>m_solve_triangular(U, B, **kwargs)</code>	
<code>make_equal_length(*shapes)</code>	Make tuples equal length.
<code>make_equal_ndim(*arrays)</code>	Add trailing unit axes so that arrays have equal ndim
<code>mean(X[, axis, keepdims])</code>	Compute the mean, ignoring NaNs.
<code>moveaxis(A, axis_from, axis_to)</code>	Move the axis <i>axis_from</i> to position <i>axis_to</i> .
<code>multiply_shapes(*shapes)</code>	Compute element-wise product of lists/tuples.
<code>nans([size])</code>	
<code>nested_iterator(max_inds)</code>	
<code>remove_whitespace(s)</code>	
<code>repeat_to_shape(A, s)</code>	
<code>rmse(y1, y2[, axis])</code>	
<code>rts_smoother(mu, Cov, A, V[, removethis])</code>	Perform Rauch-Tung-Striebel smoothing to obtain the posterior.
<code>squeeze(X)</code>	Remove leading axes that have unit length.
<code>squeeze_to_dim(X, dim)</code>	
<code>sum_multiply(*args[, axis, sumaxis, keepdims])</code>	
<code>sum_product(*args[, axes_to_keep, ...])</code>	
<code>sum_to_dim(A, dim)</code>	Sum leading axes of A such that A has dim dimensions.
<code>sum_to_shape(X, s)</code>	Sum axes of the array such that the resulting shape is as given.
<code>symm(X)</code>	Make X symmetric.

Continued on next page

Table 6.67 – continued from previous page

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.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 $i_n \neq j_n$ 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 $2 \times \text{ndim}$ axes.

bayespy.utils.misc.grid

`bayespy.utils.misc.grid(x1, x2)`

Returns meshgrid as a $(M \times N, 2)$ -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.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_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

<code>CholeskyDense(K)</code>	
<code>CholeskySparse(K)</code>	
<code>TestCase([methodName])</code>	Simple base class for unit testing.

bayespy.utils.misc.CholeskyDense

`class bayespy.utils.misc.CholeskyDense(K)`

`__init__(K)`

Methods

<code>__init__(K)</code>
<code>logdet()</code>
<code>solve(b)</code>
<code>trace_solve_gradient(dK)</code>

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__ ([<i>methodName</i>])	Create an instance of the class that will use the named test method when
addCleanup (<i>function</i> , * <i>args</i> , ** <i>kwargs</i>)	Add a function, with arguments, to be called when the test is completed.
addTypeEqualityFunc (<i>typeobj</i> , <i>function</i>)	Add a type specific assertEquals style function to compare a type.
assertAllClose (<i>A</i> , <i>B</i> [, <i>msg</i> , <i>rtol</i> , <i>atol</i>])	

Table 6.71 -

<code>assertAlmostEqual(first, second[, places, ...])</code>	Fail if the two objects are unequal as determined by their difference rounded to the given number of places.
<code>assertAlmostEquals(*args, **kwargs)</code>	
<code>assertArrayEqual(A, B[, msg])</code>	An unordered sequence comparison asserting that the same elements, regardless of order, are present in both sequences. Checks whether dictionary is a superset of subset.
<code>assertCountEqual(first, second[, msg])</code>	
<code>assertDictContainsSubset(subset, dictionary)</code>	
<code>assertDictEqual(d1, d2[, msg])</code>	Fail if the two objects are unequal as determined by the '==' operator.
<code>assertEqual(first, second[, msg])</code>	
<code>assertEquals(*args, **kwargs)</code>	
<code>assertFalse(expr[, msg])</code>	Check that the expression is false.
<code>assertGreater(a, b[, msg])</code>	Just like <code>self.assertTrue(a > b)</code> , but with a nicer default message.
<code>assertGreaterEqual(a, b[, msg])</code>	Just like <code>self.assertTrue(a >= b)</code> , but with a nicer default message.
<code>assertIn(member, container[, msg])</code>	Just like <code>self.assertTrue(a in b)</code> , but with a nicer default message.
<code>assertIs(expr1, expr2[, msg])</code>	Just like <code>self.assertTrue(a is b)</code> , but with a nicer default message.
<code>assertIsInstance(obj, cls[, msg])</code>	Same as <code>self.assertTrue(isinstance(obj, cls))</code> , with a nicer default message.
<code>assertIsNone(obj[, msg])</code>	Same as <code>self.assertTrue(obj is None)</code> , with a nicer default message.
<code>assertIsNot(expr1, expr2[, msg])</code>	Just like <code>self.assertTrue(a is not b)</code> , but with a nicer default message.
<code>assertIsNotNone(obj[, msg])</code>	Included for symmetry with <code>assertIsNone</code> .
<code>assertLess(a, b[, msg])</code>	Just like <code>self.assertTrue(a < b)</code> , but with a nicer default message.
<code>assertLessEqual(a, b[, msg])</code>	Just like <code>self.assertTrue(a <= b)</code> , but with a nicer default message.
<code>assertListEqual(list1, list2[, msg])</code>	A list-specific equality assertion.
<code>assertLogs([logger, level])</code>	Fail unless a log message of level <i>level</i> or higher is emitted on <i>logger_name</i> .
<code>assertMessage(M1, M2)</code>	
<code>assertMessageToChild(X, u)</code>	
<code>assertMultiLineEqual(first, second[, msg])</code>	Assert that two multi-line strings are equal.
<code>assertNotAlmostEqual(first, second[, ...])</code>	Fail if the two objects are equal as determined by their difference rounded to the given number of places.
<code>assertNotAlmostEquals(*args, **kwargs)</code>	
<code>assertNotEqual(first, second[, msg])</code>	Fail if the two objects are equal as determined by the '!=' operator.
<code>assertNotEquals(*args, **kwargs)</code>	
<code>assertNotIn(member, container[, msg])</code>	Just like <code>self.assertTrue(a not in b)</code> , but with a nicer default message.
<code>assertNotIsInstance(obj, cls[, msg])</code>	Included for symmetry with <code>assertIsInstance</code> .
<code>assertNotRegex(text, unexpected_regex[, msg])</code>	Fail the test if the text matches the regular expression.
<code>assertRaises(excClass[, callableObj])</code>	Fail unless an exception of class <i>excClass</i> is raised by <i>callableObj</i> when invoked.
<code>assertRaisesRegex(expected_exception, ...[, ...])</code>	Asserts that the message in a raised exception matches a regex.
<code>assertRaisesRegexp(*args, **kwargs)</code>	
<code>assertRegex(text, expected_regex[, msg])</code>	Fail the test unless the text matches the regular expression.
<code>assertRegexpMatches(*args, **kwargs)</code>	
<code>assertSequenceEqual(seq1, seq2[, msg, seq_type])</code>	An equality assertion for ordered sequences (like lists and tuples).
<code>assertSetEqual(set1, set2[, msg])</code>	A set-specific equality assertion.
<code>assertTrue(expr[, msg])</code>	Check that the expression is true.
<code>assertTupleEqual(tuple1, tuple2[, msg])</code>	A tuple-specific equality assertion.
<code>assertWarns(expected_warning[, callable_obj])</code>	Fail unless a warning of class <i>warnClass</i> is triggered by <i>callable_obj</i> when invoked.
<code>assertWarnsRegex(expected_warning, ...[, ...])</code>	Asserts that the message in a triggered warning matches a regex.
<code>assert_(*args, **kwargs)</code>	
<code>countTestCases()</code>	
<code>debug()</code>	Run the test without collecting errors in a <code>TestResult</code> .
<code>defaultTestResult()</code>	
<code>doCleanups()</code>	Execute all cleanup functions.
<code>fail([msg])</code>	Fail immediately, with the given message.
<code>failIf(*args, **kwargs)</code>	
<code>failIfAlmostEqual(*args, **kwargs)</code>	
<code>failIfEqual(*args, **kwargs)</code>	
<code>failUnless(*args, **kwargs)</code>	

<code>failUnlessAlmostEqual(*args, **kwargs)</code>	
<code>failUnlessEqual(*args, **kwargs)</code>	
<code>failUnlessRaises(*args, **kwargs)</code>	
<code>id()</code>	
<code>run([result])</code>	
<code>setUp()</code>	Hook method for setting up the test fixture before exercising it.
<code>setUpClass()</code>	Hook method for setting up class fixture before running tests in the class.
<code>shortDescription()</code>	Returns a one-line description of the test, or None if no description has been set.
<code>skipTest(reason)</code>	Skip this test.
<code>subTest([msg])</code>	Return a context manager that will return the enclosed block of code in a sub-test.
<code>tearDown()</code>	Hook method for deconstructing the test fixture after testing it.
<code>tearDownClass()</code>	Hook method for deconstructing the class fixture after running all tests in the class.

`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 significant 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 significant 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

<code>longMessage</code>	<code>bool(x) -> bool</code>
<code>maxDiff</code>	<code>int(x=0) -> integer</code>

`bayespy.utils.misc.TestCase.longMessage`

`TestCase.longMessage = True`

`bayespy.utils.misc.TestCase.maxDiff`

`TestCase.maxDiff = 640`

Bibliography

- *Bibliography*
- *genindex*

- *modindex*
- *search*

BIBLIOGRAPHY

- [1] 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. doi:[10.1007/978-3-642-40988-2_20](https://doi.org/10.1007/978-3-642-40988-2_20).
- [2] Jaakko Luttinen and Alexander Ilin. Transformations in variational Bayesian factor analysis to speed up learning. *Neurocomputing*, 73:1093–1102, 2010. doi:[10.1016/j.neucom.2009.11.018](https://doi.org/10.1016/j.neucom.2009.11.018).
- [3] 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 ??? of Lecture Notes in Computer Science, pages ???–??? Springer, 2014. doi:???

b

`bayespy.inference`, 146
`bayespy.nodes`, 57
`bayespy.plot`, 156
`bayespy.utils.linalg`, 239
`bayespy.utils.misc`, 246
`bayespy.utils.optimize`, 246
`bayespy.utils.random`, 242

Symbols

- `__init__()` (bayespy.inference.VB method), 147
- `__init__()` (bayespy.inference.vmp.nodes.bernoulli.BernoulliDistribution method), 229
- `__init__()` (bayespy.inference.vmp.nodes.bernoulli.BernoulliMoments method), 195
- `__init__()` (bayespy.inference.vmp.nodes.beta.BetaDistribution method), 224
- `__init__()` (bayespy.inference.vmp.nodes.beta.BetaMoments method), 193
- `__init__()` (bayespy.inference.vmp.nodes.binomial.BinomialDistribution method), 230, 231
- `__init__()` (bayespy.inference.vmp.nodes.binomial.BinomialMoments method), 196
- `__init__()` (bayespy.inference.vmp.nodes.categorical.CategoricalDistribution method), 232
- `__init__()` (bayespy.inference.vmp.nodes.categorical.CategoricalMoments method), 197
- `__init__()` (bayespy.inference.vmp.nodes.categorical_markov_chain.CategoricalMarkovChainDistribution method), 234
- `__init__()` (bayespy.inference.vmp.nodes.categorical_markov_chain.CategoricalMarkovChainMoments method), 198
- `__init__()` (bayespy.inference.vmp.nodes.constant.Constant method), 173, 174
- `__init__()` (bayespy.inference.vmp.nodes.deterministic.Deterministic method), 171, 172
- `__init__()` (bayespy.inference.vmp.nodes.dirichlet.DirichletDistribution method), 227
- `__init__()` (bayespy.inference.vmp.nodes.dirichlet.DirichletMoments method), 194
- `__init__()` (bayespy.inference.vmp.nodes.expfamily.ExponentialFamily method), 168
- `__init__()` (bayespy.inference.vmp.nodes.expfamily.ExponentialFamilyDistribution method), 202
- `__init__()` (bayespy.inference.vmp.nodes.gamma.GammaDistribution method), 221
- `__init__()` (bayespy.inference.vmp.nodes.gamma.GammaMoments method), 192
- `__init__()` (bayespy.inference.vmp.nodes.gaussian.GaussianARDDistribution method), 207
- `__init__()` (bayespy.inference.vmp.nodes.gaussian.GaussianDistribution method), 205
- `__init__()` (bayespy.inference.vmp.nodes.gaussian.GaussianGammaARDDistribution method), 210
- `__init__()` (bayespy.inference.vmp.nodes.gaussian.GaussianGammaARDMoments method), 190
- `__init__()` (bayespy.inference.vmp.nodes.gaussian.GaussianGammaARDToGaussianGamma method), 178, 179
- `__init__()` (bayespy.inference.vmp.nodes.gaussian.GaussianGammaISODistribution method), 208
- `__init__()` (bayespy.inference.vmp.nodes.gaussian.GaussianGammaISOMoments method), 189
- `__init__()` (bayespy.inference.vmp.nodes.gaussian.GaussianGammaISOTOGaussianGamma method), 177
- `__init__()` (bayespy.inference.vmp.nodes.gaussian.GaussianMoments method), 187
- `__init__()` (bayespy.inference.vmp.nodes.gaussian.GaussianToGaussianGamma method), 175
- `__init__()` (bayespy.inference.vmp.nodes.gaussian.GaussianWishartDistribution method), 212
- `__init__()` (bayespy.inference.vmp.nodes.gaussian.GaussianWishartMoments method), 191
- `__init__()` (bayespy.inference.vmp.nodes.gaussian.WrapToGaussianGamma method), 182
- `__init__()` (bayespy.inference.vmp.nodes.gaussian.WrapToGaussianGammaMoments method), 180
- `__init__()` (bayespy.inference.vmp.nodes.gaussian.WrapToGaussianWishart method), 184
- `__init__()` (bayespy.inference.vmp.nodes.gaussian_markov_chain.GaussianMarkovChainDistribution method), 213, 214
- `__init__()` (bayespy.inference.vmp.nodes.gaussian_markov_chain.GaussianMarkovChainMoments method), 188
- `__init__()` (bayespy.inference.vmp.nodes.gaussian_markov_chain.SwitchingGaussianMarkovChain method), 215, 217
- `__init__()` (bayespy.inference.vmp.nodes.gaussian_markov_chain.VaryingGaussianMarkovChain method), 219
- `__init__()` (bayespy.inference.vmp.nodes.multinomial.MultinomialDistribution method), 236
- `__init__()` (bayespy.inference.vmp.nodes.multinomial.MultinomialMoments method), 199
- `__init__()` (bayespy.inference.vmp.nodes.node.Moments method), 186
- `__init__()` (bayespy.inference.vmp.nodes.node.Node method), 163, 164

[__init__\(\)](#) (bayespy.inference.vmp.nodes.poisson.PoissonDistribution method), 237
[__init__\(\)](#) (bayespy.inference.vmp.nodes.poisson.PoissonMoments method), 200
[__init__\(\)](#) (bayespy.inference.vmp.nodes.stochastic.Distribution method), 201
[__init__\(\)](#) (bayespy.inference.vmp.nodes.stochastic.Stochastic method), 165, 166
[__init__\(\)](#) (bayespy.inference.vmp.nodes.wishart.WishartDistribution method), 223
[__init__\(\)](#) (bayespy.inference.vmp.nodes.wishart.WishartMoments method), 193
[__init__\(\)](#) (bayespy.inference.vmp.transformations.RotateGaussian method), 150
[__init__\(\)](#) (bayespy.inference.vmp.transformations.RotateGaussianARD method), 151
[__init__\(\)](#) (bayespy.inference.vmp.transformations.RotateGaussianMarkovChain method), 152
[__init__\(\)](#) (bayespy.inference.vmp.transformations.RotateMultiple method), 155
[__init__\(\)](#) (bayespy.inference.vmp.transformations.RotateSwitchingMarkovChain method), 153
[__init__\(\)](#) (bayespy.inference.vmp.transformations.RotateVaryingMarkovChain method), 154, 155
[__init__\(\)](#) (bayespy.inference.vmp.transformations.RotationOptimizer method), 149
[__init__\(\)](#) (bayespy.nodes.Bernoulli method), 90, 91
[__init__\(\)](#) (bayespy.nodes.Beta method), 110, 111
[__init__\(\)](#) (bayespy.nodes.Binomial method), 94, 95
[__init__\(\)](#) (bayespy.nodes.Categorical method), 98, 99
[__init__\(\)](#) (bayespy.nodes.CategoricalMarkovChain method), 119
[__init__\(\)](#) (bayespy.nodes.Dirichlet method), 114, 115
[__init__\(\)](#) (bayespy.nodes.Exponential method), 74, 75
[__init__\(\)](#) (bayespy.nodes.Gamma method), 66, 67
[__init__\(\)](#) (bayespy.nodes.Gate method), 145
[__init__\(\)](#) (bayespy.nodes.Gaussian method), 57, 58
[__init__\(\)](#) (bayespy.nodes.GaussianARD method), 62, 63
[__init__\(\)](#) (bayespy.nodes.GaussianGammaARD method), 83
[__init__\(\)](#) (bayespy.nodes.GaussianGammaISO method), 78, 79
[__init__\(\)](#) (bayespy.nodes.GaussianMarkovChain method), 123, 124
[__init__\(\)](#) (bayespy.nodes.GaussianWishart method), 86, 87
[__init__\(\)](#) (bayespy.nodes.Mixture method), 138, 139
[__init__\(\)](#) (bayespy.nodes.Multinomial method), 102, 103
[__init__\(\)](#) (bayespy.nodes.Poisson method), 106, 107
[__init__\(\)](#) (bayespy.nodes.SumMultiply method), 143
[__init__\(\)](#) (bayespy.nodes.SwitchingGaussianMarkovChain method), 128, 129
[__init__\(\)](#) (bayespy.nodes.VaryingGaussianMarkovChain method), 133, 134
[__init__\(\)](#) (bayespy.nodes.Wishart method), 70, 71
[__init__\(\)](#) (bayespy.plot.CategoricalMarkovChainPlotter method), 161
[__init__\(\)](#) (bayespy.plot.ContourPlotter method), 159
[__init__\(\)](#) (bayespy.plot.FunctionPlotter method), 160
[__init__\(\)](#) (bayespy.plot.GaussianTimeseriesPlotter method), 160
[__init__\(\)](#) (bayespy.plot.HintonPlotter method), 160
[__init__\(\)](#) (bayespy.plot.PDFPlotter method), 159
[__init__\(\)](#) (bayespy.plot.Plotter method), 158
[__init__\(\)](#) (bayespy.utils.misc.CholeskyDense method), 256
[__init__\(\)](#) (bayespy.utils.misc.CholeskySparse method), 257
[__init__\(\)](#) (bayespy.utils.misc.TestCase method), 257, 259

A

[add_converter\(\)](#) (in module bayespy.utils.misc), 248
[add_converter\(\)](#) (bayespy.inference.vmp.nodes.bernoulli.BernoulliMoments method), 95
[add_converter\(\)](#) (bayespy.inference.vmp.nodes.beta.BetaMoments method), 194
[add_converter\(\)](#) (bayespy.inference.vmp.nodes.binomial.BinomialMoments method), 196
[add_converter\(\)](#) (bayespy.inference.vmp.nodes.categorical.CategoricalMoments method), 197
[add_converter\(\)](#) (bayespy.inference.vmp.nodes.categorical_markov_chain.CategoricalMarkovChainMoments method), 198
[add_converter\(\)](#) (bayespy.inference.vmp.nodes.dirichlet.DirichletMoments method), 195
[add_converter\(\)](#) (bayespy.inference.vmp.nodes.gamma.GammaMoments method), 192
[add_converter\(\)](#) (bayespy.inference.vmp.nodes.gaussian.GaussianGammaARD method), 190
[add_converter\(\)](#) (bayespy.inference.vmp.nodes.gaussian.GaussianGammaISO method), 189
[add_converter\(\)](#) (bayespy.inference.vmp.nodes.gaussian.GaussianMoments method), 187
[add_converter\(\)](#) (bayespy.inference.vmp.nodes.gaussian.GaussianWishartMoments method), 191
[add_converter\(\)](#) (bayespy.inference.vmp.nodes.gaussian_markov_chain.GaussianMarkovChainMoments method), 188
[add_converter\(\)](#) (bayespy.inference.vmp.nodes.multinomial.MultinomialMoments method), 199
[add_converter\(\)](#) (bayespy.inference.vmp.nodes.node.Moments class method), 186
[add_converter\(\)](#) (bayespy.inference.vmp.nodes.poisson.PoissonMoments method), 200
[add_converter\(\)](#) (bayespy.inference.vmp.nodes.wishart.WishartMoments method), 193
[add_leading_axes\(\)](#) (in module bayespy.utils.misc), 248
[add_plate_axis\(\)](#) (bayespy.inference.vmp.nodes.constant.Constant method), 174

[add_plate_axis\(\) \(bayespy.inference.vmp.nodes.deterministic.Deterministic method\), 172](#)
[add_plate_axis\(\) \(bayespy.inference.vmp.nodes.expfamily.ExponentialFamily method\), 168](#)
[add_plate_axis\(\) \(bayespy.inference.vmp.nodes.gaussian.GaussianGammaARD method\), 179](#)
[add_plate_axis\(\) \(bayespy.inference.vmp.nodes.gaussian.GaussianGammaISO method\), 177](#)
[add_plate_axis\(\) \(bayespy.inference.vmp.nodes.gaussian.GaussianGammaWishart method\), 175](#)
[add_plate_axis\(\) \(bayespy.inference.vmp.nodes.gaussian.WrapToGaussianGammaISO method\), 182](#)
[add_plate_axis\(\) \(bayespy.inference.vmp.nodes.gaussian.WrapToGaussianGammaWishart method\), 184](#)
[add_plate_axis\(\) \(bayespy.inference.vmp.nodes.node.Node method\), 164](#)
[add_plate_axis\(\) \(bayespy.inference.vmp.nodes.stochastic.Stochastic method\), 166](#)
[add_plate_axis\(\) \(bayespy.nodes.Bernoulli method\), 91](#)
[add_plate_axis\(\) \(bayespy.nodes.Beta method\), 111](#)
[add_plate_axis\(\) \(bayespy.nodes.Binomial method\), 95](#)
[add_plate_axis\(\) \(bayespy.nodes.Categorical method\), 99](#)
[add_plate_axis\(\) \(bayespy.nodes.CategoricalMarkovChain method\), 120](#)
[add_plate_axis\(\) \(bayespy.nodes.Dirichlet method\), 115](#)
[add_plate_axis\(\) \(bayespy.nodes.Exponential method\), 75](#)
[add_plate_axis\(\) \(bayespy.nodes.Gamma method\), 67](#)
[add_plate_axis\(\) \(bayespy.nodes.Gate method\), 145](#)
[add_plate_axis\(\) \(bayespy.nodes.Gaussian method\), 58](#)
[add_plate_axis\(\) \(bayespy.nodes.GaussianARD method\), 63](#)
[add_plate_axis\(\) \(bayespy.nodes.GaussianGammaARD method\), 83](#)
[add_plate_axis\(\) \(bayespy.nodes.GaussianGammaISO method\), 79](#)
[add_plate_axis\(\) \(bayespy.nodes.GaussianMarkovChain method\), 124](#)
[add_plate_axis\(\) \(bayespy.nodes.GaussianWishart method\), 87](#)
[add_plate_axis\(\) \(bayespy.nodes.Mixture method\), 139](#)
[add_plate_axis\(\) \(bayespy.nodes.Multinomial method\), 103](#)
[add_plate_axis\(\) \(bayespy.nodes.Poisson method\), 107](#)
[add_plate_axis\(\) \(bayespy.nodes.SumMultiply method\), 143](#)
[add_plate_axis\(\) \(bayespy.nodes.SwitchingGaussianMarkovChain method\), 129](#)
[add_plate_axis\(\) \(bayespy.nodes.VaryingGaussianMarkovChain method\), 134](#)
[add_plate_axis\(\) \(bayespy.nodes.Wishart method\), 71](#)
[add_trailing_axes\(\) \(in module bayespy.utils.misc\), 248](#)
[addTypeEqualityFunc\(\) \(bayespy.utils.misc.TestCase method\), 259](#)
[alpha_beta_recursion\(\) \(in module bayespy.utils.random\), 242](#)
[array_to_scalar\(\) \(in module bayespy.utils.misc\), 248](#)
[assertAllClose\(\) \(bayespy.nodes.Exponential method\), 75](#)
[assertAllClose\(\) \(bayespy.nodes.Gamma method\), 67](#)
[assertAllClose\(\) \(bayespy.nodes.Wishart method\), 266](#)
[assertAllClose\(\) \(bayespy.utils.misc.TestCase method\), 266](#)
[assertAlmostEqual\(\) \(bayespy.utils.misc.TestCase method\), 260](#)
[assertAlmostEquals\(\) \(bayespy.utils.misc.TestCase method\), 260](#)
[assertArrayEqual\(\) \(bayespy.utils.misc.TestCase method\), 260](#)
[assertCountEqual\(\) \(bayespy.utils.misc.TestCase method\), 260](#)
[assertDictContainsSubset\(\) \(bayespy.utils.misc.TestCase method\), 260](#)
[assertDictEqual\(\) \(bayespy.utils.misc.TestCase method\), 260](#)
[assertEqual\(\) \(bayespy.utils.misc.TestCase method\), 260](#)
[assertEquals\(\) \(bayespy.utils.misc.TestCase method\), 260](#)
[assertFalse\(\) \(bayespy.utils.misc.TestCase method\), 261](#)
[assertGreater\(\) \(bayespy.utils.misc.TestCase method\), 261](#)
[assertGreaterEqual\(\) \(bayespy.utils.misc.TestCase method\), 261](#)
[assertIn\(\) \(bayespy.utils.misc.TestCase method\), 261](#)
[assertIs\(\) \(bayespy.utils.misc.TestCase method\), 261](#)
[assertIsInstance\(\) \(bayespy.utils.misc.TestCase method\), 261](#)
[assertIsNone\(\) \(bayespy.utils.misc.TestCase method\), 261](#)
[assertIsNot\(\) \(bayespy.utils.misc.TestCase method\), 261](#)
[assertIsNotNone\(\) \(bayespy.utils.misc.TestCase method\), 261](#)
[assertLess\(\) \(bayespy.utils.misc.TestCase method\), 262](#)
[assertLessEqual\(\) \(bayespy.utils.misc.TestCase method\), 262](#)
[assertListEqual\(\) \(bayespy.utils.misc.TestCase method\), 262](#)
[assertLogs\(\) \(bayespy.utils.misc.TestCase method\), 262](#)
[assertMessage\(\) \(bayespy.utils.misc.TestCase method\), 262](#)
[assertMessageToChild\(\) \(bayespy.utils.misc.TestCase method\), 262](#)
[assertMultiLineEqual\(\) \(bayespy.utils.misc.TestCase method\), 262](#)

assertNotAlmostEqual() (bayespy.utils.misc.TestCase method), 263

assertNotAlmostEquals() (bayespy.utils.misc.TestCase method), 263

assertNotEqual() (bayespy.utils.misc.TestCase method), 263

assertNotEquals() (bayespy.utils.misc.TestCase method), 263

assertNotIn() (bayespy.utils.misc.TestCase method), 263

assertNotIsInstance() (bayespy.utils.misc.TestCase method), 263

assertNotRegex() (bayespy.utils.misc.TestCase method), 263

assertRaises() (bayespy.utils.misc.TestCase method), 263

assertRaisesRegex() (bayespy.utils.misc.TestCase method), 264

assertRaisesRegexp() (bayespy.utils.misc.TestCase method), 264

assertRegex() (bayespy.utils.misc.TestCase method), 264

assertRegexpMatches() (bayespy.utils.misc.TestCase method), 264

assertSequenceEqual() (bayespy.utils.misc.TestCase method), 264

assertSetEqual() (bayespy.utils.misc.TestCase method), 265

assertTrue() (bayespy.utils.misc.TestCase method), 265

assertTupleEqual() (bayespy.utils.misc.TestCase method), 265

assertWarns() (bayespy.utils.misc.TestCase method), 265

assertWarnsRegex() (bayespy.utils.misc.TestCase method), 266

atleast_nd() (in module bayespy.utils.misc), 248

axes_to_collapse() (in module bayespy.utils.misc), 248

B

bayespy.inference (module), 146

bayespy.nodes (module), 57

bayespy.plot (module), 156

bayespy.utils.linalg (module), 239

bayespy.utils.misc (module), 246

bayespy.utils.optimize (module), 246

bayespy.utils.random (module), 242

Bernoulli (class in bayespy.nodes), 90

bernoulli() (in module bayespy.utils.random), 242

BernoulliDistribution (class in bayespy.inference.vmp.nodes.bernoulli), 229

BernoulliMoments (class in bayespy.inference.vmp.nodes.bernoulli), 195

Beta (class in bayespy.nodes), 110

BetaDistribution (class in bayespy.inference.vmp.nodes.beta), 224

BetaMoments (class in bayespy.inference.vmp.nodes.beta), 193

Binomial (class in bayespy.nodes), 94

BinomialDistribution (class in bayespy.inference.vmp.nodes.binomial), 230

BinomialMoments (class in bayespy.inference.vmp.nodes.binomial), 196

block_banded() (in module bayespy.utils.misc), 248

block_banded_solve() (in module bayespy.utils.linalg), 240

bound() (bayespy.inference.vmp.transformations.RotateGaussian method), 150

bound() (bayespy.inference.vmp.transformations.RotateGaussianARD method), 151

bound() (bayespy.inference.vmp.transformations.RotateGaussianMarkovChain method), 152

bound() (bayespy.inference.vmp.transformations.RotateMultiple method), 156

bound() (bayespy.inference.vmp.transformations.RotateSwitchingMarkovChain method), 154

bound() (bayespy.inference.vmp.transformations.RotateVaryingMarkovChain method), 155

broadcasted_shape() (in module bayespy.utils.misc), 249

broadcasted_shape_from_arrays() (in module bayespy.utils.misc), 249

C

Categorical (class in bayespy.nodes), 98

categorical() (in module bayespy.utils.random), 243

CategoricalDistribution (class in bayespy.inference.vmp.nodes.categorical), 232

CategoricalMarkovChain (class in bayespy.nodes), 118

CategoricalMarkovChainDistribution (class in bayespy.inference.vmp.nodes.categorical_markov_chain), 234

CategoricalMarkovChainMoments (class in bayespy.inference.vmp.nodes.categorical_markov_chain), 198

CategoricalMarkovChainPlotter (class in bayespy.plot), 161

CategoricalMoments (class in bayespy.inference.vmp.nodes.categorical), 197

ceildiv() (in module bayespy.utils.misc), 249

check_gradient() (in module bayespy.utils.misc), 249

check_gradient() (in module bayespy.utils.optimize), 246

chol() (in module bayespy.utils.linalg), 240

chol() (in module bayespy.utils.misc), 249

chol_inv() (in module bayespy.utils.linalg), 240

chol_inv() (in module bayespy.utils.misc), 249

chol_logdet() (in module bayespy.utils.linalg), 240

chol_logdet() (in module bayespy.utils.misc), 249

chol_solve() (in module bayespy.utils.linalg), 240

chol_solve() (in module bayespy.utils.misc), 249

cholesky() (in module bayespy.utils.misc), 249

CholeskyDense (class in bayespy.utils.misc), 256

CholeskySparse (class in bayespy.utils.misc), 257

composite_function() (in module bayespy.utils.misc), 250

compute_cgf_from_parents()
(bayespy.inference.vmp.nodes.bernoulli.BernoulliDistribution
method), 229

compute_cgf_from_parents()
(bayespy.inference.vmp.nodes.beta.BetaDistribution
method), 226

compute_cgf_from_parents()
(bayespy.inference.vmp.nodes.binomial.BinomialDistribution
method), 231

compute_cgf_from_parents()
(bayespy.inference.vmp.nodes.categorical.CategoricalDistribution
method), 233

compute_cgf_from_parents()
(bayespy.inference.vmp.nodes.categorical_markov_chain.CategoricalMarkovChainDistribution
method), 234

compute_cgf_from_parents()
(bayespy.inference.vmp.nodes.dirichlet.DirichletDistribution
method), 227

compute_cgf_from_parents()
(bayespy.inference.vmp.nodes.expfamily.ExponentialFamilyDistribution
method), 203

compute_cgf_from_parents()
(bayespy.inference.vmp.nodes.gamma.GammaDistribution
method), 222

compute_cgf_from_parents()
(bayespy.inference.vmp.nodes.gaussian.GaussianARDDistribution
method), 207

compute_cgf_from_parents()
(bayespy.inference.vmp.nodes.gaussian.GaussianDistribution
method), 205

compute_cgf_from_parents()
(bayespy.inference.vmp.nodes.gaussian.GaussianGammaARDDistribution
method), 210

compute_cgf_from_parents()
(bayespy.inference.vmp.nodes.gaussian.GaussianGammaISODistribution
method), 209

compute_cgf_from_parents()
(bayespy.inference.vmp.nodes.gaussian.GaussianWishartDistribution
method), 212

compute_cgf_from_parents()
(bayespy.inference.vmp.nodes.gaussian_markov_chain.GaussianMarkovChainDistribution
method), 214

compute_cgf_from_parents()
(bayespy.inference.vmp.nodes.gaussian_markov_chain.SwitchingGaussianMarkovChainDistribution
method), 217

compute_cgf_from_parents()
(bayespy.inference.vmp.nodes.gaussian_markov_chain.VaryingGaussianMarkovChainDistribution
method), 219

compute_cgf_from_parents()
(bayespy.inference.vmp.nodes.multinomial.MultinomialDistribution
method), 236

compute_cgf_from_parents()
(bayespy.inference.vmp.nodes.poisson.PoissonDistribution
method), 238

compute_cgf_from_parents()
(bayespy.inference.vmp.nodes.wishart.WishartDistribution
method), 223

compute_dims_from_values()
(bayespy.inference.vmp.nodes.bernoulli.BernoulliMoments
method), 196

compute_dims_from_values()
(bayespy.inference.vmp.nodes.beta.BetaMoments
method), 194

compute_dims_from_values()
(bayespy.inference.vmp.nodes.binomial.BinomialMoments
method), 197

compute_dims_from_values()
(bayespy.inference.vmp.nodes.categorical.CategoricalMoments
method), 198

compute_dims_from_values()
(bayespy.inference.vmp.nodes.categorical_markov_chain.CategoricalMarkovChainMoments
method), 199

compute_dims_from_values()
(bayespy.inference.vmp.nodes.dirichlet.DirichletMoments
method), 195

compute_dims_from_values()
(bayespy.inference.vmp.nodes.gamma.GammaMoments
method), 192

compute_dims_from_values()
(bayespy.inference.vmp.nodes.gaussian.GaussianGammaARDMoments
method), 190

compute_dims_from_values()
(bayespy.inference.vmp.nodes.gaussian.GaussianGammaISOMoments
method), 189

compute_dims_from_values()
(bayespy.inference.vmp.nodes.gaussian.GaussianMoments
method), 187

compute_dims_from_values()
(bayespy.inference.vmp.nodes.gaussian.GaussianWishartMoments
method), 191

compute_dims_from_values()
(bayespy.inference.vmp.nodes.gaussian_markov_chain.GaussianMarkovChainMoments
method), 188

compute_dims_from_values()
(bayespy.inference.vmp.nodes.multinomial.MultinomialMoments
method), 199

compute_dims_from_values()
(bayespy.inference.vmp.nodes.poisson.PoissonMoments
method), 186

method), 200

compute_dims_from_values()
(bayespy.inference.vmp.nodes.wishart.WishartMoments
method), 193

compute_fixed_moments()
(bayespy.inference.vmp.nodes.bernoulli.BernoulliMoments
method), 196

compute_fixed_moments()
(bayespy.inference.vmp.nodes.beta.BetaMoments
method), 194

compute_fixed_moments()
(bayespy.inference.vmp.nodes.binomial.BinomialMoments
method), 197

compute_fixed_moments()
(bayespy.inference.vmp.nodes.categorical.CategoricalMoments
method), 198

compute_fixed_moments()
(bayespy.inference.vmp.nodes.categorical_markov_chain.CategoricalMarkovChainMoments
method), 199

compute_fixed_moments()
(bayespy.inference.vmp.nodes.dirichlet.DirichletMoments
method), 195

compute_fixed_moments()
(bayespy.inference.vmp.nodes.gamma.GammaMoments
method), 192

compute_fixed_moments()
(bayespy.inference.vmp.nodes.gaussian.GaussianGammaARDDMoments
method), 190

compute_fixed_moments()
(bayespy.inference.vmp.nodes.gaussian.GaussianGammaISODMoments
method), 189

compute_fixed_moments()
(bayespy.inference.vmp.nodes.gaussian.GaussianMoments
method), 187

compute_fixed_moments()
(bayespy.inference.vmp.nodes.gaussian.GaussianWishartMoments
method), 191

compute_fixed_moments()
(bayespy.inference.vmp.nodes.gaussian_markov_chain.GaussianMarkovChainMoments
method), 188

compute_fixed_moments()
(bayespy.inference.vmp.nodes.multinomial.MultinomialMoments
method), 200

compute_fixed_moments()
(bayespy.inference.vmp.nodes.node.Moments
method), 186

compute_fixed_moments()
(bayespy.inference.vmp.nodes.poisson.PoissonMoments
method), 200

compute_fixed_moments()
(bayespy.inference.vmp.nodes.wishart.WishartMoments
method), 193

compute_fixed_moments_and_f()
(bayespy.inference.vmp.nodes.bernoulli.BernoulliDistribution
method), 229

compute_fixed_moments_and_f()
(bayespy.inference.vmp.nodes.beta.BetaDistribution
method), 226

compute_fixed_moments_and_f()
(bayespy.inference.vmp.nodes.binomial.BinomialDistribution
method), 231

compute_fixed_moments_and_f()
(bayespy.inference.vmp.nodes.categorical.CategoricalDistribution
method), 233

compute_fixed_moments_and_f()
(bayespy.inference.vmp.nodes.categorical_markov_chain.CategoricalMarkovChainDistribution
method), 234

compute_fixed_moments_and_f()
(bayespy.inference.vmp.nodes.dirichlet.DirichletDistribution
method), 228

compute_fixed_moments_and_f()
(bayespy.inference.vmp.nodes.exponential.ExponentialFamilyDistribution
method), 203

compute_fixed_moments_and_f()
(bayespy.inference.vmp.nodes.gamma.GammaDistribution
method), 222

compute_fixed_moments_and_f()
(bayespy.inference.vmp.nodes.gaussian.GaussianARDDistribution
method), 207

compute_fixed_moments_and_f()
(bayespy.inference.vmp.nodes.gaussian.GaussianDistribution
method), 205

compute_fixed_moments_and_f()
(bayespy.inference.vmp.nodes.gaussian.GaussianGammaARDDistribution
method), 211

compute_fixed_moments_and_f()
(bayespy.inference.vmp.nodes.gaussian.GaussianGammaISODistribution
method), 209

compute_fixed_moments_and_f()
(bayespy.inference.vmp.nodes.gaussian.GaussianWishartDistribution
method), 212

compute_fixed_moments_and_f()
(bayespy.inference.vmp.nodes.gaussian_markov_chain.GaussianMarkovChainDistribution
method), 214

compute_fixed_moments_and_f()
(bayespy.inference.vmp.nodes.gaussian_markov_chain.SwitchingGaussianMarkovChainDistribution
method), 217

compute_fixed_moments_and_f()
(bayespy.inference.vmp.nodes.gaussian_markov_chain.VaryingGaussianMarkovChainDistribution
method), 220

compute_fixed_moments_and_f()
(bayespy.inference.vmp.nodes.multinomial.MultinomialDistribution
method), 236

compute_fixed_moments_and_f()
(bayespy.inference.vmp.nodes.poisson.PoissonDistribution
method), 238

compute_fixed_moments_and_f()
(bayespy.inference.vmp.nodes.wishart.WishartDistribution
method), 238

method), 223

compute_logpdf() (bayespy.inference.vmp.nodes.bernoulli.BernoulliDistribution method), 229

compute_logpdf() (bayespy.inference.vmp.nodes.beta.BetaDistribution method), 226

compute_logpdf() (bayespy.inference.vmp.nodes.binomial.BinomialDistribution method), 231

compute_logpdf() (bayespy.inference.vmp.nodes.categorical.CategoricalDistribution method), 233

compute_logpdf() (bayespy.inference.vmp.nodes.categorical_markov_chain.CategoricalMarkovChainDistribution method), 235

compute_logpdf() (bayespy.inference.vmp.nodes.dirichlet.DirichletDistribution method), 228

compute_logpdf() (bayespy.inference.vmp.nodes.expfamily.ExponentialFamilyDistribution method), 203

compute_logpdf() (bayespy.inference.vmp.nodes.gamma.GammaDistribution method), 222

compute_logpdf() (bayespy.inference.vmp.nodes.gaussian.GaussianARDDistribution method), 207

compute_logpdf() (bayespy.inference.vmp.nodes.gaussian.GaussianDistribution method), 206

compute_logpdf() (bayespy.inference.vmp.nodes.gaussian.GaussianGammaARDDistribution method), 205

compute_logpdf() (bayespy.inference.vmp.nodes.gaussian.GaussianGammaISODistribution method), 211

compute_logpdf() (bayespy.inference.vmp.nodes.gaussian.GaussianGammaSODistribution method), 209

compute_logpdf() (bayespy.inference.vmp.nodes.gaussian.GaussianWishartDistribution method), 212

compute_logpdf() (bayespy.inference.vmp.nodes.gaussian_markov_chain.GaussianMarkovChainDistribution method), 214

compute_logpdf() (bayespy.inference.vmp.nodes.gaussian_markov_chain.SwitchingGaussianMarkovChainDistribution method), 217

compute_logpdf() (bayespy.inference.vmp.nodes.gaussian_markov_chain.VaryingGaussianMarkovChainDistribution method), 220

compute_logpdf() (bayespy.inference.vmp.nodes.multinomial.MultinomialDistribution method), 236

compute_logpdf() (bayespy.inference.vmp.nodes.poisson.PoissonDistribution method), 238

compute_logpdf() (bayespy.inference.vmp.nodes.wishart.WishartDistribution method), 223

compute_lowerbound() (bayespy.inference.VB method), 147

compute_lowerbound_terms() (bayespy.inference.VB method), 148

compute_mask_to_parent() (bayespy.inference.vmp.nodes.bernoulli.BernoulliDistribution method), 229

compute_mask_to_parent() (bayespy.inference.vmp.nodes.beta.BetaDistribution method), 226

compute_mask_to_parent() (bayespy.inference.vmp.nodes.binomial.BinomialDistribution method), 231

compute_mask_to_parent() (bayespy.inference.vmp.nodes.categorical.CategoricalDistribution method), 233

compute_mask_to_parent() (bayespy.inference.vmp.nodes.categorical_markov_chain.CategoricalMarkovChainDistribution method), 235

compute_mask_to_parent() (bayespy.inference.vmp.nodes.dirichlet.DirichletDistribution method), 228

compute_mask_to_parent() (bayespy.inference.vmp.nodes.expfamily.ExponentialFamilyDistribution method), 203

compute_mask_to_parent() (bayespy.inference.vmp.nodes.gamma.GammaDistribution method), 222

compute_mask_to_parent() (bayespy.inference.vmp.nodes.gaussian.GaussianARDDistribution method), 207

compute_mask_to_parent() (bayespy.inference.vmp.nodes.gaussian.GaussianDistribution method), 206

compute_mask_to_parent() (bayespy.inference.vmp.nodes.gaussian.GaussianGammaARDDistribution method), 205

compute_mask_to_parent() (bayespy.inference.vmp.nodes.gaussian.GaussianGammaISODistribution method), 211

compute_mask_to_parent() (bayespy.inference.vmp.nodes.gaussian.GaussianGammaSODistribution method), 209

compute_mask_to_parent() (bayespy.inference.vmp.nodes.gaussian.GaussianWishartDistribution method), 212

compute_mask_to_parent() (bayespy.inference.vmp.nodes.gaussian_markov_chain.GaussianMarkovChainDistribution method), 214

compute_mask_to_parent() (bayespy.inference.vmp.nodes.gaussian_markov_chain.SwitchingGaussianMarkovChainDistribution method), 217

compute_mask_to_parent() (bayespy.inference.vmp.nodes.gaussian_markov_chain.VaryingGaussianMarkovChainDistribution method), 220

compute_mask_to_parent() (bayespy.inference.vmp.nodes.multinomial.MultinomialDistribution method), 236

compute_mask_to_parent() (bayespy.inference.vmp.nodes.poisson.PoissonDistribution method), 238

compute_mask_to_parent() (bayespy.inference.vmp.nodes.wishart.WishartDistribution method), 223

compute_message_to_parent() (bayespy.inference.vmp.nodes.bernoulli.BernoulliDistribution method), 229

compute_message_to_parent() (bayespy.inference.vmp.nodes.beta.BetaDistribution method), 226

compute_message_to_parent() (bayespy.inference.vmp.nodes.binomial.BinomialDistribution method), 231

compute_message_to_parent() (bayespy.inference.vmp.nodes.categorical.CategoricalDistribution method), 233

compute_message_to_parent() (bayespy.inference.vmp.nodes.categorical_markov_chain.CategoricalMarkovChainDistribution method), 235

compute_message_to_parent() (bayespy.inference.vmp.nodes.dirichlet.DirichletDistribution method), 228

compute_message_to_parent() (bayespy.inference.vmp.nodes.expfamily.ExponentialFamilyDistribution method), 203

compute_message_to_parent() (bayespy.inference.vmp.nodes.gamma.GammaDistribution method), 222

compute_message_to_parent() (bayespy.inference.vmp.nodes.gaussian.GaussianARDDistribution method), 207

compute_message_to_parent() (bayespy.inference.vmp.nodes.gaussian.GaussianDistribution method), 206

compute_message_to_parent() (bayespy.inference.vmp.nodes.gaussian.GaussianGammaARDDistribution method), 205

compute_message_to_parent() (bayespy.inference.vmp.nodes.gaussian.GaussianGammaISODistribution method), 211

compute_message_to_parent() (bayespy.inference.vmp.nodes.gaussian.GaussianGammaSODistribution method), 209

compute_message_to_parent() (bayespy.inference.vmp.nodes.gaussian.GaussianWishartDistribution method), 212

compute_message_to_parent() (bayespy.inference.vmp.nodes.gaussian_markov_chain.GaussianMarkovChainDistribution method), 214

compute_message_to_parent() (bayespy.inference.vmp.nodes.gaussian_markov_chain.SwitchingGaussianMarkovChainDistribution method), 217

compute_message_to_parent() (bayespy.inference.vmp.nodes.gaussian_markov_chain.VaryingGaussianMarkovChainDistribution method), 220

compute_message_to_parent() (bayespy.inference.vmp.nodes.multinomial.MultinomialDistribution method), 236

compute_message_to_parent() (bayespy.inference.vmp.nodes.poisson.PoissonDistribution method), 238

compute_message_to_parent() (bayespy.inference.vmp.nodes.wishart.WishartDistribution method), 223

method), 226	method), 224
compute_message_to_parent() (bayespy.inference.vmp.nodes.binomial.BinomialDistribution method), 231	compute_moments_and_cgf() (bayespy.inference.vmp.nodes.bernoulli.BernoulliDistribution method), 230
compute_message_to_parent() (bayespy.inference.vmp.nodes.categorical.CategoricalDistribution method), 233	compute_moments_and_cgf() (bayespy.inference.vmp.nodes.beta.BetaDistribution method), 226
compute_message_to_parent() (bayespy.inference.vmp.nodes.categorical_markov_chain.CategoricalMarkovChainDistribution method), 235	compute_moments_and_cgf() (bayespy.inference.vmp.nodes.binomial.BinomialDistribution method), 231
compute_message_to_parent() (bayespy.inference.vmp.nodes.dirichlet.DirichletDistribution method), 228	compute_moments_and_cgf() (bayespy.inference.vmp.nodes.categorical.CategoricalDistribution method), 233
compute_message_to_parent() (bayespy.inference.vmp.nodes.expfamily.ExponentialFamilyDistribution method), 203	compute_moments_and_cgf() (bayespy.inference.vmp.nodes.categorical_markov_chain.CategoricalMarkovChainDistribution method), 235
compute_message_to_parent() (bayespy.inference.vmp.nodes.gamma.GammaDistribution method), 222	compute_moments_and_cgf() (bayespy.inference.vmp.nodes.dirichlet.DirichletDistribution method), 228
compute_message_to_parent() (bayespy.inference.vmp.nodes.gaussian.GaussianARDDistribution method), 208	compute_moments_and_cgf() (bayespy.inference.vmp.nodes.expfamily.ExponentialFamilyDistribution method), 203
compute_message_to_parent() (bayespy.inference.vmp.nodes.gaussian.GaussianDistribution method), 206	compute_moments_and_cgf() (bayespy.inference.vmp.nodes.gamma.GammaDistribution method), 222
compute_message_to_parent() (bayespy.inference.vmp.nodes.gaussian.GaussianGammaARDDistribution method), 211	compute_moments_and_cgf() (bayespy.inference.vmp.nodes.gaussian.GaussianARDDistribution method), 208
compute_message_to_parent() (bayespy.inference.vmp.nodes.gaussian.GaussianGammaSODistribution method), 209	compute_moments_and_cgf() (bayespy.inference.vmp.nodes.gaussian.GaussianDistribution method), 206
compute_message_to_parent() (bayespy.inference.vmp.nodes.gaussian.GaussianWishartDistribution method), 212	compute_moments_and_cgf() (bayespy.inference.vmp.nodes.gaussian.GaussianGammaARDDistribution method), 211
compute_message_to_parent() (bayespy.inference.vmp.nodes.gaussian_markov_chain.GaussianMarkovChainDistribution method), 214	compute_moments_and_cgf() (bayespy.inference.vmp.nodes.gaussian.GaussianGammaSODistribution method), 209
compute_message_to_parent() (bayespy.inference.vmp.nodes.gaussian_markov_chain.SwitchingGaussianMarkovChainDistribution method), 218	compute_moments_and_cgf() (bayespy.inference.vmp.nodes.gaussian.GaussianWishartDistribution method), 213
compute_message_to_parent() (bayespy.inference.vmp.nodes.gaussian_markov_chain.VaryingGaussianMarkovChainDistribution method), 220	compute_moments_and_cgf() (bayespy.inference.vmp.nodes.gaussian_markov_chain.GaussianMarkovChainDistribution method), 214
compute_message_to_parent() (bayespy.inference.vmp.nodes.multinomial.MultinomialDistribution method), 237	compute_moments_and_cgf() (bayespy.inference.vmp.nodes.gaussian_markov_chain.SwitchingGaussianMarkovChainDistribution method), 218
compute_message_to_parent() (bayespy.inference.vmp.nodes.poisson.PoissonDistribution method), 238	compute_moments_and_cgf() (bayespy.inference.vmp.nodes.gaussian_markov_chain.VaryingGaussianMarkovChainDistribution method), 220
compute_message_to_parent() (bayespy.inference.vmp.nodes.stochastic.Distribution method), 202	compute_moments_and_cgf() (bayespy.inference.vmp.nodes.multinomial.MultinomialDistribution method), 237
compute_message_to_parent() (bayespy.inference.vmp.nodes.wishart.WishartDistribution	compute_moments_and_cgf() (bayespy.inference.vmp.nodes.poisson.PoissonDistribution

method), 238
compute_moments_and_cgf()
(bayespy.inference.vmp.nodes.wishart.WishartDistribution method), 224
compute_phi_from_parents()
(bayespy.inference.vmp.nodes.bernoulli.BernoulliDistribution method), 230
compute_phi_from_parents()
(bayespy.inference.vmp.nodes.beta.BetaDistribution method), 226
compute_phi_from_parents()
(bayespy.inference.vmp.nodes.binomial.BinomialDistribution method), 231
compute_phi_from_parents()
(bayespy.inference.vmp.nodes.categorical.CategoricalDistribution method), 233
compute_phi_from_parents()
(bayespy.inference.vmp.nodes.categorical_markov_chain.CategoricalMarkovChainDistribution method), 235
compute_phi_from_parents()
(bayespy.inference.vmp.nodes.dirichlet.DirichletDistribution method), 228
compute_phi_from_parents()
(bayespy.inference.vmp.nodes.expfamily.ExponentialFamilyDistribution method), 203
compute_phi_from_parents()
(bayespy.inference.vmp.nodes.gamma.GammaDistribution method), 222
compute_phi_from_parents()
(bayespy.inference.vmp.nodes.gaussian.GaussianARDDistribution method), 208
compute_phi_from_parents()
(bayespy.inference.vmp.nodes.gaussian.GaussianDistribution method), 206
compute_phi_from_parents()
(bayespy.inference.vmp.nodes.gaussian.GaussianGammaARDDistribution method), 211
compute_phi_from_parents()
(bayespy.inference.vmp.nodes.gaussian.GaussianGammaISODistribution method), 209
compute_phi_from_parents()
(bayespy.inference.vmp.nodes.gaussian.GaussianWishartDistribution method), 213
compute_phi_from_parents()
(bayespy.inference.vmp.nodes.gaussian_markov_chain.GaussianMarkovChainDistribution method), 215
compute_phi_from_parents()
(bayespy.inference.vmp.nodes.gaussian_markov_chain.SwitchingGaussianMarkovChainDistribution method), 218
compute_phi_from_parents()
(bayespy.inference.vmp.nodes.gaussian_markov_chain.VaryingGaussianMarkovChainDistribution method), 220
compute_phi_from_parents()
(bayespy.inference.vmp.nodes.multinomial.MultinomialDistribution method), 237
compute_phi_from_parents()
(bayespy.inference.vmp.nodes.poisson.PoissonDistribution method), 238
compute_phi_from_parents()
(bayespy.inference.vmp.nodes.wishart.WishartDistribution method), 224
Constant (class in bayespy.inference.vmp.nodes.constant), 173
contour() (in module bayespy.plot), 157
ContourPlotter (class in bayespy.plot), 159
Distribution (in module bayespy.utils.random), 243
countTestCases() (bayespy.utils.misc.TestCase method), 266
createDistribution (in module bayespy.utils.random), 243

D

delete() (bayespy.inference.vmp.nodes.constant.Constant method), 174
delete() (bayespy.inference.vmp.nodes.deterministic.Deterministic method), 168
delete() (bayespy.inference.vmp.nodes.expfamily.ExponentialFamily method), 179
delete() (bayespy.inference.vmp.nodes.gaussian.GaussianGammaARDDistribution method), 177
delete() (bayespy.inference.vmp.nodes.gaussian.GaussianGammaISOTToGaussianGammaISODistribution method), 175
delete() (bayespy.inference.vmp.nodes.gaussian.GaussianToGaussianGammaARDDistribution method), 182
delete() (bayespy.inference.vmp.nodes.gaussian.WrapToGaussianGammaARDDistribution method), 184
delete() (bayespy.inference.vmp.nodes.gaussian.WrapToGaussianGammaISODistribution method), 164
delete() (bayespy.inference.vmp.nodes.stochastic.Stochastic method), 166
delete() (bayespy.nodes.Bernoulli method), 91
delete() (bayespy.nodes.Beta method), 111
delete() (bayespy.nodes.Binomial method), 95
delete() (bayespy.nodes.Categorical method), 99
delete() (bayespy.nodes.CategoricalMarkovChain method), 121
delete() (bayespy.nodes.Dirichlet method), 115
delete() (bayespy.nodes.Exponential method), 75
delete() (bayespy.nodes.Gamma method), 89
delete() (bayespy.nodes.Gate method), 145
delete() (bayespy.nodes.Gaussian method), 58
delete() (bayespy.nodes.GaussianARD method), 63

- ul style="list-style-type: none; padding-left: 0;">
- `delete()` (`bayespy.nodes.GaussianGammaARD` method), 83
- `delete()` (`bayespy.nodes.GaussianGammaISO` method), 79
- `delete()` (`bayespy.nodes.GaussianMarkovChain` method), 124
- `delete()` (`bayespy.nodes.GaussianWishart` method), 87
- `delete()` (`bayespy.nodes.Mixture` method), 139
- `delete()` (`bayespy.nodes.Multinomial` method), 103
- `delete()` (`bayespy.nodes.Poisson` method), 107
- `delete()` (`bayespy.nodes.SumMultiply` method), 143
- `delete()` (`bayespy.nodes.SwitchingGaussianMarkovChain` method), 129
- `delete()` (`bayespy.nodes.VaryingGaussianMarkovChain` method), 134
- `delete()` (`bayespy.nodes.Wishart` method), 71
- `Deterministic` (class in `bayespy.inference.vmp.nodes.deterministic`), 171
- `diag()` (in module `bayespy.utils.misc`), 250
- `diagonal()` (in module `bayespy.utils.misc`), 250
- `dims` (`bayespy.inference.vmp.nodes.expfamily.ExponentialFamily` attribute), 171
- `dims` (`bayespy.nodes.Bernoulli` attribute), 94
- `dims` (`bayespy.nodes.Beta` attribute), 114
- `dims` (`bayespy.nodes.Binomial` attribute), 98
- `dims` (`bayespy.nodes.Categorical` attribute), 102
- `dims` (`bayespy.nodes.CategoricalMarkovChain` attribute), 122
- `dims` (`bayespy.nodes.Dirichlet` attribute), 118
- `dims` (`bayespy.nodes.Exponential` attribute), 78
- `dims` (`bayespy.nodes.Gamma` attribute), 70
- `dims` (`bayespy.nodes.Gaussian` attribute), 61
- `dims` (`bayespy.nodes.GaussianARD` attribute), 66
- `dims` (`bayespy.nodes.GaussianGammaARD` attribute), 86
- `dims` (`bayespy.nodes.GaussianGammaISO` attribute), 82
- `dims` (`bayespy.nodes.GaussianMarkovChain` attribute), 127
- `dims` (`bayespy.nodes.GaussianWishart` attribute), 90
- `dims` (`bayespy.nodes.Mixture` attribute), 141
- `dims` (`bayespy.nodes.Multinomial` attribute), 106
- `dims` (`bayespy.nodes.Poisson` attribute), 109
- `dims` (`bayespy.nodes.SwitchingGaussianMarkovChain` attribute), 132
- `dims` (`bayespy.nodes.VaryingGaussianMarkovChain` attribute), 137
- `dims` (`bayespy.nodes.Wishart` attribute), 73
- `Dirichlet` (class in `bayespy.nodes`), 114
- `dirichlet()` (in module `bayespy.utils.random`), 243
- `DirichletDistribution` (class in `bayespy.inference.vmp.nodes.dirichlet`), 227
- `DirichletMoments` (class in `bayespy.inference.vmp.nodes.dirichlet`), 194
- `dist_haversine()` (in module `bayespy.utils.misc`), 250
- `Distribution` (class in `bayespy.inference.vmp.nodes.stochastic`), 201
- `doCleanups()` (`bayespy.utils.misc.TestCase` method), 266
- `Dot()` (in module `bayespy.nodes`), 142
- `dot()` (in module `bayespy.utils.linalg`), 240
- ## E
- `Exponential` (class in `bayespy.nodes`), 74
 - `ExponentialFamily` (class in `bayespy.inference.vmp.nodes.expfamily`), 167
 - `ExponentialFamilyDistribution` (class in `bayespy.inference.vmp.nodes.expfamily`), 202
- ## F
- `fail()` (`bayespy.utils.misc.TestCase` method), 266
 - `failIf()` (`bayespy.utils.misc.TestCase` method), 266
 - `failIfAlmostEqual()` (`bayespy.utils.misc.TestCase` method), 267
 - `failIfEqual()` (`bayespy.utils.misc.TestCase` method), 267
 - `failUnless()` (`bayespy.utils.misc.TestCase` method), 267
 - `failUnlessAlmostEqual()` (`bayespy.utils.misc.TestCase` method), 267
 - `failUnlessEqual()` (`bayespy.utils.misc.TestCase` method), 267
 - `failUnlessRaises()` (`bayespy.utils.misc.TestCase` method), 267
 - `first()` (in module `bayespy.utils.misc`), 250
 - `FunctionPlotter` (class in `bayespy.plot`), 160
- ## G
- `Gamma` (class in `bayespy.nodes`), 66
 - `gamma_entropy()` (in module `bayespy.utils.random`), 243
 - `gamma_logpdf()` (in module `bayespy.utils.random`), 244
 - `GammaDistribution` (class in `bayespy.inference.vmp.nodes.gamma`), 221
 - `GammaMoments` (class in `bayespy.inference.vmp.nodes.gamma`), 192
 - `Gate` (class in `bayespy.nodes`), 145
 - `Gaussian` (class in `bayespy.nodes`), 57
 - `gaussian_entropy()` (in module `bayespy.utils.random`), 244
 - `gaussian_gamma_to_t()` (in module `bayespy.utils.random`), 244
 - `gaussian_logpdf()` (in module `bayespy.utils.misc`), 250
 - `gaussian_logpdf()` (in module `bayespy.utils.random`), 244
 - `gaussian_mixture()` (in module `bayespy.plot`), 157
 - `GaussianARD` (class in `bayespy.nodes`), 61
 - `GaussianARDDistribution` (class in `bayespy.inference.vmp.nodes.gaussian`), 206
 - `GaussianDistribution` (class in `bayespy.inference.vmp.nodes.gaussian`), 204
 - `GaussianGammaARD` (class in `bayespy.nodes`), 82

GaussianGammaARDDistribution	(class in bayespy.inference.vmp.nodes.gaussian), 210	get_converter() (bayespy.inference.vmp.nodes.gaussian.GammaMoments method), 192
GaussianGammaARDMoments	(class in bayespy.inference.vmp.nodes.gaussian), 190	get_converter() (bayespy.inference.vmp.nodes.gaussian.GaussianGammaARD method), 191
GaussianGammaARDToGaussianWishart	(class in bayespy.inference.vmp.nodes.gaussian), 178	get_converter() (bayespy.inference.vmp.nodes.gaussian.GaussianGammaARD method), 189
GaussianGammaISO (class in bayespy.nodes), 78		get_converter() (bayespy.inference.vmp.nodes.gaussian.GaussianMoments method), 188
GaussianGammaISODistribution	(class in bayespy.inference.vmp.nodes.gaussian), 208	get_converter() (bayespy.inference.vmp.nodes.gaussian.GaussianWishartMoments method), 191
GaussianGammaISOMoments	(class in bayespy.inference.vmp.nodes.gaussian), 189	get_converter() (bayespy.inference.vmp.nodes.gaussian_markov_chain.GaussianMarkovChainMoments method), 188
GaussianGammaISOToGaussianGammaARD	(class in bayespy.inference.vmp.nodes.gaussian), 177	get_converter() (bayespy.inference.vmp.nodes.multinomial.MultinomialMoments method), 200
GaussianMarkovChain (class in bayespy.nodes), 122		get_converter() (bayespy.inference.vmp.nodes.node.Moments method), 187
GaussianMarkovChainDistribution	(class in bayespy.inference.vmp.nodes.gaussian_markov_chain), 213	get_converter() (bayespy.inference.vmp.nodes.poisson.PoissonMoments method), 201
GaussianMarkovChainMoments	(class in bayespy.inference.vmp.nodes.gaussian_markov_chain), 188	get_converter() (bayespy.inference.vmp.nodes.wishart.WishartMoments method), 193
GaussianMoments	(class in bayespy.inference.vmp.nodes.gaussian), 187	get_diag() (in module bayespy.utils.misc), 250
GaussianTimeseriesPlotter (class in bayespy.plot), 160		get_gaussian_mean_and_variance() (bayespy.nodes.GaussianGammaISO method), 79
GaussianToGaussianGammaISO	(class in bayespy.inference.vmp.nodes.gaussian), 175	get_iteration_by_nodes() (bayespy.inference.VB method), 148
GaussianWishart (class in bayespy.nodes), 86		get_marginal_logpdf() (bayespy.nodes.GaussianGammaISO method), 79
GaussianWishartDistribution	(class in bayespy.inference.vmp.nodes.gaussian), 212	get_mask() (bayespy.inference.vmp.nodes.constant.Constant method), 174
GaussianWishartMoments	(class in bayespy.inference.vmp.nodes.gaussian), 191	get_mask_terms() (bayespy.inference.vmp.nodes.deterministic.Deterministic method), 172
get_bound_terms() (bayespy.inference.vmp.transformations.PotentialMasking method), 150		get_mask_terms() (bayespy.inference.vmp.nodes.expfamily.ExponentialFamily method), 169
get_bound_terms() (bayespy.inference.vmp.transformations.PotentialMasking method), 151		get_mask_terms() (bayespy.inference.vmp.nodes.gaussian.GaussianGammaARD method), 179
get_bound_terms() (bayespy.inference.vmp.transformations.PotentialMasking method), 153		get_mask_terms() (bayespy.inference.vmp.nodes.gaussian.GaussianGammaISOToGaussianGammaARD method), 177
get_bound_terms() (bayespy.inference.vmp.transformations.PotentialMasking method), 154		get_mask_terms() (bayespy.inference.vmp.nodes.gaussian.GaussianToGaussianGamma method), 176
get_bound_terms() (bayespy.inference.vmp.transformations.PotentialMasking method), 155		get_mask_terms() (bayespy.inference.vmp.nodes.gaussian.WrapToGaussianGamma method), 182
get_converter() (bayespy.inference.vmp.nodes.bernoulli.BernoulliMoments method), 196		get_mask_terms() (bayespy.inference.vmp.nodes.gaussian.WrapToGaussianGamma method), 181
get_converter() (bayespy.inference.vmp.nodes.beta.BetaMoments method), 194		get_mask_terms() (bayespy.inference.vmp.nodes.gaussian.WrapToGaussianWishart method), 184
get_converter() (bayespy.inference.vmp.nodes.binomial.BinomialMoments method), 197		get_mask_terms() (bayespy.inference.vmp.nodes.node.Node method), 164
get_converter() (bayespy.inference.vmp.nodes.categorical.CategoricalMoments method), 198		get_mask_terms() (bayespy.inference.vmp.nodes.stochastic.Stochastic method), 166
get_converter() (bayespy.inference.vmp.nodes.categorical_multinomial.MultinomialMoments method), 199		get_mask_terms() (bayespy.inference.vmp.nodes.stochastic.Stochastic method), 166
get_converter() (bayespy.inference.vmp.nodes.dirichlet.DirichletMoments method), 195		get_mask_terms() (bayespy.inference.vmp.nodes.stochastic.Stochastic method), 166

[get_mask\(\) \(bayespy.nodes.CategoricalMarkovChain method\), 120](#)
[get_mask\(\) \(bayespy.nodes.Dirichlet method\), 115](#)
[get_mask\(\) \(bayespy.nodes.Exponential method\), 75](#)
[get_mask\(\) \(bayespy.nodes.Gamma method\), 67](#)
[get_mask\(\) \(bayespy.nodes.Gate method\), 145](#)
[get_mask\(\) \(bayespy.nodes.Gaussian method\), 58](#)
[get_mask\(\) \(bayespy.nodes.GaussianARD method\), 63](#)
[get_mask\(\) \(bayespy.nodes.GaussianGammaARD method\), 84](#)
[get_mask\(\) \(bayespy.nodes.GaussianGammaISO method\), 80](#)
[get_mask\(\) \(bayespy.nodes.GaussianMarkovChain method\), 124](#)
[get_mask\(\) \(bayespy.nodes.GaussianWishart method\), 87](#)
[get_mask\(\) \(bayespy.nodes.Mixture method\), 139](#)
[get_mask\(\) \(bayespy.nodes.Multinomial method\), 103](#)
[get_mask\(\) \(bayespy.nodes.Poisson method\), 107](#)
[get_mask\(\) \(bayespy.nodes.SumMultiply method\), 143](#)
[get_mask\(\) \(bayespy.nodes.SwitchingGaussianMarkovChain method\), 129](#)
[get_mask\(\) \(bayespy.nodes.VaryingGaussianMarkovChain method\), 134](#)
[get_mask\(\) \(bayespy.nodes.Wishart method\), 71](#)
[get_moments\(\) \(bayespy.inference.vmp.nodes.constant.Constant method\), 174](#)
[get_moments\(\) \(bayespy.inference.vmp.nodes.deterministic.Deterministic method\), 172](#)
[get_moments\(\) \(bayespy.inference.vmp.nodes.expfamily.ExponentialFamily method\), 169](#)
[get_moments\(\) \(bayespy.inference.vmp.nodes.gaussian.Gaussian method\), 179](#)
[get_moments\(\) \(bayespy.inference.vmp.nodes.gaussian.GaussianARD method\), 177](#)
[get_moments\(\) \(bayespy.inference.vmp.nodes.gaussian.GaussianGammaARD method\), 179](#)
[get_moments\(\) \(bayespy.inference.vmp.nodes.gaussian.GaussianGammaISO method\), 177](#)
[get_moments\(\) \(bayespy.inference.vmp.nodes.gaussian.GaussianGammaISOT method\), 176](#)
[get_moments\(\) \(bayespy.inference.vmp.nodes.gaussian.WrapToGaussianARD method\), 183](#)
[get_moments\(\) \(bayespy.inference.vmp.nodes.gaussian.WrapToGaussianGamma method\), 181](#)
[get_moments\(\) \(bayespy.inference.vmp.nodes.gaussian.WrapToGaussianWishart method\), 184](#)
[get_moments\(\) \(bayespy.inference.vmp.nodes.node.Node method\), 164](#)
[get_moments\(\) \(bayespy.inference.vmp.nodes.stochastic.Stochastic method\), 166](#)
[get_moments\(\) \(bayespy.nodes.Bernoulli method\), 91](#)
[get_moments\(\) \(bayespy.nodes.Beta method\), 111](#)
[get_moments\(\) \(bayespy.nodes.Binomial method\), 95](#)
[get_moments\(\) \(bayespy.nodes.Categorical method\), 99](#)
[get_moments\(\) \(bayespy.nodes.CategoricalMarkovChain method\), 120](#)
[get_moments\(\) \(bayespy.nodes.Dirichlet method\), 115](#)
[get_moments\(\) \(bayespy.nodes.Exponential method\), 75](#)
[get_moments\(\) \(bayespy.nodes.Gamma method\), 67](#)
[get_moments\(\) \(bayespy.nodes.Gate method\), 145](#)
[get_moments\(\) \(bayespy.nodes.Gaussian method\), 58](#)
[get_moments\(\) \(bayespy.nodes.GaussianARD method\), 59](#)
[get_moments\(\) \(bayespy.nodes.GaussianARD method\), 63](#)
[get_moments\(\) \(bayespy.nodes.GaussianGammaARD method\), 84](#)
[get_moments\(\) \(bayespy.nodes.GaussianGammaISO method\), 80](#)
[get_moments\(\) \(bayespy.nodes.GaussianMarkovChain method\), 124](#)
[get_moments\(\) \(bayespy.nodes.GaussianWishart method\), 87](#)
[get_moments\(\) \(bayespy.nodes.Mixture method\), 139](#)
[get_moments\(\) \(bayespy.nodes.Multinomial method\), 103](#)
[get_moments\(\) \(bayespy.nodes.Poisson method\), 107](#)
[get_moments\(\) \(bayespy.nodes.SumMultiply method\), 144](#)
[get_moments\(\) \(bayespy.nodes.SwitchingGaussianMarkovChain method\), 129](#)
[get_moments\(\) \(bayespy.nodes.VaryingGaussianMarkovChain method\), 134](#)
[get_moments\(\) \(bayespy.nodes.Wishart method\), 71](#)
[get_parameters\(\) \(bayespy.nodes.SumMultiply method\), 144](#)
[get_shape\(\) \(bayespy.inference.vmp.nodes.constant.Constant method\), 174](#)
[get_shape\(\) \(bayespy.inference.vmp.nodes.deterministic.Deterministic method\), 172](#)
[get_shape\(\) \(bayespy.inference.vmp.nodes.expfamily.ExponentialFamily method\), 169](#)
[get_shape\(\) \(bayespy.inference.vmp.nodes.gaussian.Gaussian method\), 179](#)
[get_shape\(\) \(bayespy.inference.vmp.nodes.gaussian.GaussianARD method\), 179](#)
[get_shape\(\) \(bayespy.inference.vmp.nodes.gaussian.GaussianGammaARD method\), 177](#)
[get_shape\(\) \(bayespy.inference.vmp.nodes.gaussian.GaussianGammaISO method\), 177](#)
[get_shape\(\) \(bayespy.inference.vmp.nodes.gaussian.GaussianGammaISOT method\), 176](#)
[get_shape\(\) \(bayespy.inference.vmp.nodes.gaussian.WrapToGaussianARD method\), 183](#)
[get_shape\(\) \(bayespy.inference.vmp.nodes.gaussian.WrapToGaussianGamma method\), 181](#)
[get_shape\(\) \(bayespy.inference.vmp.nodes.gaussian.WrapToGaussianWishart method\), 184](#)
[get_shape\(\) \(bayespy.inference.vmp.nodes.node.Node method\), 164](#)
[get_shape\(\) \(bayespy.inference.vmp.nodes.stochastic.Stochastic method\), 166](#)
[get_shape\(\) \(bayespy.nodes.Bernoulli method\), 92](#)
[get_shape\(\) \(bayespy.nodes.Beta method\), 111](#)
[get_shape\(\) \(bayespy.nodes.Binomial method\), 96](#)
[get_shape\(\) \(bayespy.nodes.Categorical method\), 99](#)
[get_shape\(\) \(bayespy.nodes.CategoricalMarkovChain method\), 120](#)

[get_shape\(\) \(bayespy.nodes.Dirichlet method\), 115](#)
[get_shape\(\) \(bayespy.nodes.Exponential method\), 75](#)
[get_shape\(\) \(bayespy.nodes.Gamma method\), 67](#)
[get_shape\(\) \(bayespy.nodes.Gate method\), 146](#)
[get_shape\(\) \(bayespy.nodes.Gaussian method\), 59](#)
[get_shape\(\) \(bayespy.nodes.GaussianARD method\), 63](#)
[get_shape\(\) \(bayespy.nodes.GaussianGammaARD method\), 84](#)
[get_shape\(\) \(bayespy.nodes.GaussianGammaISO method\), 80](#)
[get_shape\(\) \(bayespy.nodes.GaussianMarkovChain method\), 124](#)
[get_shape\(\) \(bayespy.nodes.GaussianWishart method\), 87](#)
[get_shape\(\) \(bayespy.nodes.Mixture method\), 139](#)
[get_shape\(\) \(bayespy.nodes.Multinomial method\), 103](#)
[get_shape\(\) \(bayespy.nodes.Poisson method\), 107](#)
[get_shape\(\) \(bayespy.nodes.SumMultiply method\), 144](#)
[get_shape\(\) \(bayespy.nodes.SwitchingGaussianMarkovChain method\), 129](#)
[get_shape\(\) \(bayespy.nodes.VaryingGaussianMarkovChain method\), 135](#)
[get_shape\(\) \(bayespy.nodes.Wishart method\), 71](#)
[grid\(\) \(in module bayespy.utils.misc\), 250](#)

H

[has_plotter\(\) \(bayespy.inference.vmp.nodes.constant.Constant method\), 174](#)
[has_plotter\(\) \(bayespy.inference.vmp.nodes.deterministic.Deterministic method\), 172](#)
[has_plotter\(\) \(bayespy.inference.vmp.nodes.expfamily.ExponentialFamily method\), 169](#)
[has_plotter\(\) \(bayespy.inference.vmp.nodes.gaussian.GaussianGammaARDToGaussianWishart method\), 179](#)
[has_plotter\(\) \(bayespy.inference.vmp.nodes.gaussian.GaussianGammaISOToGaussianGammaARD method\), 178](#)
[has_plotter\(\) \(bayespy.inference.vmp.nodes.gaussian.GaussianToGaussianGammaISO method\), 176](#)
[has_plotter\(\) \(bayespy.inference.vmp.nodes.gaussian.WrapToGaussianGammaARD method\), 183](#)
[has_plotter\(\) \(bayespy.inference.vmp.nodes.gaussian.WrapToGaussianGammaISO method\), 181](#)
[has_plotter\(\) \(bayespy.inference.vmp.nodes.gaussian.WrapToGaussianWishart method\), 185](#)
[has_plotter\(\) \(bayespy.inference.vmp.nodes.node.Node method\), 164](#)
[has_plotter\(\) \(bayespy.inference.vmp.nodes.stochastic.Stochastic method\), 166](#)
[has_plotter\(\) \(bayespy.nodes.Bernoulli method\), 92](#)
[has_plotter\(\) \(bayespy.nodes.Beta method\), 111](#)
[has_plotter\(\) \(bayespy.nodes.Binomial method\), 96](#)
[has_plotter\(\) \(bayespy.nodes.Categorical method\), 99](#)
[has_plotter\(\) \(bayespy.nodes.CategoricalMarkovChain method\), 120](#)
[has_plotter\(\) \(bayespy.nodes.Dirichlet method\), 115](#)
[has_plotter\(\) \(bayespy.nodes.Exponential method\), 75](#)
[has_plotter\(\) \(bayespy.nodes.Gamma method\), 68](#)
[has_plotter\(\) \(bayespy.nodes.Gate method\), 146](#)
[has_plotter\(\) \(bayespy.nodes.Gaussian method\), 59](#)
[has_plotter\(\) \(bayespy.nodes.GaussianARD method\), 63](#)
[has_plotter\(\) \(bayespy.nodes.GaussianGammaARD method\), 84](#)
[has_plotter\(\) \(bayespy.nodes.GaussianGammaISO method\), 80](#)
[has_plotter\(\) \(bayespy.nodes.GaussianMarkovChain method\), 125](#)
[has_plotter\(\) \(bayespy.nodes.GaussianWishart method\), 88](#)
[has_plotter\(\) \(bayespy.nodes.Mixture method\), 139](#)
[has_plotter\(\) \(bayespy.nodes.Multinomial method\), 104](#)
[has_plotter\(\) \(bayespy.nodes.Poisson method\), 107](#)
[has_plotter\(\) \(bayespy.nodes.SumMultiply method\), 144](#)
[has_plotter\(\) \(bayespy.nodes.SwitchingGaussianMarkovChain method\), 130](#)
[has_plotter\(\) \(bayespy.nodes.VaryingGaussianMarkovChain method\), 135](#)
[has_plotter\(\) \(bayespy.nodes.Wishart method\), 71](#)
[hinton\(\) \(in module bayespy.plot\), 157](#)
[HintonPlotter \(class in bayespy.plot\), 159](#)

<code>initialize_from_parameters()</code> (<code>bayespy.nodes.GaussianARD</code> method), 63	<code>initialize_from_prior()</code> (<code>bayespy.nodes.GaussianMarkovChain</code> method), 125
<code>initialize_from_parameters()</code> (<code>bayespy.nodes.GaussianGammaARD</code> method), 84	<code>initialize_from_prior()</code> (<code>bayespy.nodes.GaussianWishart</code> method), 88
<code>initialize_from_parameters()</code> (<code>bayespy.nodes.GaussianGammaISO</code> method), 80	<code>initialize_from_prior()</code> (<code>bayespy.nodes.Mixture</code> method), 139
<code>initialize_from_parameters()</code> (<code>bayespy.nodes.GaussianMarkovChain</code> method), 125	<code>initialize_from_prior()</code> (<code>bayespy.nodes.Multinomial</code> method), 104
<code>initialize_from_parameters()</code> (<code>bayespy.nodes.GaussianWishart</code> method), 88	<code>initialize_from_prior()</code> (<code>bayespy.nodes.Poisson</code> method), 108
<code>initialize_from_parameters()</code> (<code>bayespy.nodes.Mixture</code> method), 139	<code>initialize_from_prior()</code> (<code>bayespy.nodes.SwitchingGaussianMarkovChain</code> method), 130
<code>initialize_from_parameters()</code> (<code>bayespy.nodes.Multinomial</code> method), 104	<code>initialize_from_prior()</code> (<code>bayespy.nodes.VaryingGaussianMarkovChain</code> method), 135
<code>initialize_from_parameters()</code> (<code>bayespy.nodes.Poisson</code> method), 107	<code>initialize_from_prior()</code> (<code>bayespy.nodes.Wishart</code> method), 72
<code>initialize_from_parameters()</code> (<code>bayespy.nodes.SwitchingGaussianMarkovChain</code> method), 130	<code>initialize_from_random()</code> (<code>bayespy.inference.vmp.nodes.expfamily.ExponentialFamily</code> method), 169
<code>initialize_from_parameters()</code> (<code>bayespy.nodes.VaryingGaussianMarkovChain</code> method), 135	<code>initialize_from_random()</code> (<code>bayespy.nodes.Bernoulli</code> method), 92
<code>initialize_from_parameters()</code> (<code>bayespy.nodes.Wishart</code> method), 71	<code>initialize_from_random()</code> (<code>bayespy.nodes.Beta</code> method), 112
<code>initialize_from_prior()</code> (<code>bayespy.inference.vmp.nodes.expfamily.ExponentialFamily</code> method), 169	<code>initialize_from_random()</code> (<code>bayespy.nodes.Binomial</code> method), 96
<code>initialize_from_prior()</code> (<code>bayespy.nodes.Bernoulli</code> method), 92	<code>initialize_from_random()</code> (<code>bayespy.nodes.Categorical</code> method), 100
<code>initialize_from_prior()</code> (<code>bayespy.nodes.Beta</code> method), 112	<code>initialize_from_random()</code> (<code>bayespy.nodes.CategoricalMarkovChain</code> method), 110
<code>initialize_from_prior()</code> (<code>bayespy.nodes.Binomial</code> method), 96	<code>initialize_from_random()</code> (<code>bayespy.nodes.Dirichlet</code> method), 116
<code>initialize_from_prior()</code> (<code>bayespy.nodes.Categorical</code> method), 100	<code>initialize_from_random()</code> (<code>bayespy.nodes.Exponential</code> method), 76
<code>initialize_from_prior()</code> (<code>bayespy.nodes.CategoricalMarkovChain</code> method), 120	<code>initialize_from_random()</code> (<code>bayespy.nodes.Gamma</code> method), 68
<code>initialize_from_prior()</code> (<code>bayespy.nodes.Dirichlet</code> method), 116	<code>initialize_from_random()</code> (<code>bayespy.nodes.Gaussian</code> method), 59
<code>initialize_from_prior()</code> (<code>bayespy.nodes.Exponential</code> method), 76	<code>initialize_from_random()</code> (<code>bayespy.nodes.GaussianARD</code> method), 64
<code>initialize_from_prior()</code> (<code>bayespy.nodes.Gamma</code> method), 68	<code>initialize_from_random()</code> (<code>bayespy.nodes.GaussianGammaARD</code> method), 84
<code>initialize_from_prior()</code> (<code>bayespy.nodes.Gaussian</code> method), 59	<code>initialize_from_random()</code> (<code>bayespy.nodes.GaussianGammaISO</code> method), 80
<code>initialize_from_prior()</code> (<code>bayespy.nodes.GaussianARD</code> method), 63	<code>initialize_from_random()</code> (<code>bayespy.nodes.GaussianMarkovChain</code> method), 125
<code>initialize_from_prior()</code> (<code>bayespy.nodes.GaussianGammaARD</code> method), 84	<code>initialize_from_random()</code> (<code>bayespy.nodes.GaussianWishart</code> method), 88
<code>initialize_from_prior()</code> (<code>bayespy.nodes.GaussianGammaISO</code> method), 80	<code>initialize_from_random()</code> (<code>bayespy.nodes.Mixture</code> method), 139

[initialize_from_random\(\)](#) (bayespy.nodes.Multinomial method), 104
[initialize_from_random\(\)](#) (bayespy.nodes.Poisson method), 108
[initialize_from_random\(\)](#) (bayespy.nodes.SwitchingGaussianMarkovChain method), 130
[initialize_from_random\(\)](#) (bayespy.nodes.VaryingGaussianMarkovChain method), 135
[initialize_from_random\(\)](#) (bayespy.nodes.Wishart method), 72
[initialize_from_value\(\)](#) (bayespy.inference.vmp.nodes.expfamily.ExponentialFamily method), 169
[initialize_from_value\(\)](#) (bayespy.nodes.Bernoulli method), 92
[initialize_from_value\(\)](#) (bayespy.nodes.Beta method), 112
[initialize_from_value\(\)](#) (bayespy.nodes.Binomial method), 96
[initialize_from_value\(\)](#) (bayespy.nodes.Categorical method), 100
[initialize_from_value\(\)](#) (bayespy.nodes.CategoricalMarkovChain method), 120
[initialize_from_value\(\)](#) (bayespy.nodes.Dirichlet method), 116
[initialize_from_value\(\)](#) (bayespy.nodes.Exponential method), 76
[initialize_from_value\(\)](#) (bayespy.nodes.Gamma method), 68
[initialize_from_value\(\)](#) (bayespy.nodes.Gaussian method), 59
[initialize_from_value\(\)](#) (bayespy.nodes.GaussianARD method), 64
[initialize_from_value\(\)](#) (bayespy.nodes.GaussianGammaARD method), 84
[initialize_from_value\(\)](#) (bayespy.nodes.GaussianGammaISO method), 80
[initialize_from_value\(\)](#) (bayespy.nodes.GaussianMarkovChain method), 125
[initialize_from_value\(\)](#) (bayespy.nodes.GaussianWishart method), 88
[initialize_from_value\(\)](#) (bayespy.nodes.Mixture method), 140
[initialize_from_value\(\)](#) (bayespy.nodes.Multinomial method), 104
[initialize_from_value\(\)](#) (bayespy.nodes.Poisson method), 108
[initialize_from_value\(\)](#) (bayespy.nodes.SwitchingGaussianMarkovChain method), 130
[initialize_from_value\(\)](#) (bayespy.nodes.VaryingGaussianMarkovChain method), 135
[initialize_from_value\(\)](#) (bayespy.nodes.Wishart method), 72
[inner\(\)](#) (in module bayespy.utils.linalg), 240
[integrated_logpdf_from_parents\(\)](#) (bayespy.nodes.Mixture method), 140
[intervals\(\)](#) (in module bayespy.utils.random), 245
[inv\(\)](#) (in module bayespy.utils.linalg), 241
[invwishart_rand\(\)](#) (in module bayespy.utils.random), 245
[is_callable\(\)](#) (in module bayespy.utils.misc), 251
[is_numeric\(\)](#) (in module bayespy.utils.misc), 251
[is_shape_subset\(\)](#) (in module bayespy.utils.misc), 251
[is_string\(\)](#) (in module bayespy.utils.misc), 251
[isinteger\(\)](#) (in module bayespy.utils.misc), 251
K
[kaly.ExponentialFamily](#)
[kalman_filter\(\)](#) (in module bayespy.utils.misc), 251
L
[load\(\)](#) (bayespy.inference.VB method), 148
[load\(\)](#) (bayespy.inference.vmp.nodes.expfamily.ExponentialFamily method), 169
[load\(\)](#) (bayespy.inference.vmp.nodes.stochastic.Stochastic method), 166
[load\(\)](#) (bayespy.nodes.Bernoulli method), 92
[load\(\)](#) (bayespy.nodes.Beta method), 112
[load\(\)](#) (bayespy.nodes.Binomial method), 96
[load\(\)](#) (bayespy.nodes.Categorical method), 100
[load\(\)](#) (bayespy.nodes.CategoricalMarkovChain method), 121
[load\(\)](#) (bayespy.nodes.Dirichlet method), 116
[load\(\)](#) (bayespy.nodes.Exponential method), 76
[load\(\)](#) (bayespy.nodes.Gamma method), 68
[load\(\)](#) (bayespy.nodes.Gaussian method), 59
[load\(\)](#) (bayespy.nodes.GaussianARD method), 64
[load\(\)](#) (bayespy.nodes.GaussianGammaARD method), 84
[load\(\)](#) (bayespy.nodes.GaussianGammaISO method), 80
[load\(\)](#) (bayespy.nodes.GaussianMarkovChain method), 125
[load\(\)](#) (bayespy.nodes.GaussianWishart method), 88
[load\(\)](#) (bayespy.nodes.Mixture method), 140
[load\(\)](#) (bayespy.nodes.Multinomial method), 104
[load\(\)](#) (bayespy.nodes.Poisson method), 108
[load\(\)](#) (bayespy.nodes.SwitchingGaussianMarkovChain method), 130
[load\(\)](#) (bayespy.nodes.VaryingGaussianMarkovChain method), 135
[load\(\)](#) (bayespy.nodes.Wishart method), 72
[logdet\(\)](#) (bayespy.utils.misc.CholeskyDense method), 256
[logdet\(\)](#) (bayespy.utils.misc.CholeskySparse method), 257
[logdet_chol\(\)](#) (in module bayespy.utils.linalg), 241
[logdet_chol\(\)](#) (in module bayespy.utils.misc), 252
[logdet_cov\(\)](#) (in module bayespy.utils.linalg), 241
[logdet_tri\(\)](#) (in module bayespy.utils.linalg), 241

loglikelihood_lowerbound()	(bayespy.inference.VB method), 148
logpdf()	(bayespy.inference.vmp.nodes.expfamily.ExponentialFamily method), 185
lower_bound_contribution()	(bayespy.inference.vmp.nodes.gaussian.WrapToGaussianWishart method), 169
logpdf()	(bayespy.nodes.Bernoulli method), 92
logpdf()	(bayespy.nodes.Beta method), 112
logpdf()	(bayespy.nodes.Binomial method), 96
logpdf()	(bayespy.nodes.Categorical method), 100
logpdf()	(bayespy.nodes.CategoricalMarkovChain method), 121
logpdf()	(bayespy.nodes.Dirichlet method), 116
logpdf()	(bayespy.nodes.Exponential method), 76
logpdf()	(bayespy.nodes.Gamma method), 68
logpdf()	(bayespy.nodes.Gaussian method), 59
logpdf()	(bayespy.nodes.GaussianARD method), 64
logpdf()	(bayespy.nodes.GaussianGammaARD method), 84
logpdf()	(bayespy.nodes.GaussianGammaISO method), 81
logpdf()	(bayespy.nodes.GaussianMarkovChain method), 125
logpdf()	(bayespy.nodes.GaussianWishart method), 88
logpdf()	(bayespy.nodes.Mixture method), 140
logpdf()	(bayespy.nodes.Multinomial method), 104
logpdf()	(bayespy.nodes.Poisson method), 108
logpdf()	(bayespy.nodes.SwitchingGaussianMarkovChain method), 130
logpdf()	(bayespy.nodes.VaryingGaussianMarkovChain method), 135
logpdf()	(bayespy.nodes.Wishart method), 72
logsumexp()	(in module bayespy.utils.misc), 252
longMessage	(bayespy.utils.misc.TestCase attribute), 268
lower_bound_contribution()	(bayespy.inference.vmp.nodes.deterministic.Deterministic method), 172
lower_bound_contribution()	(bayespy.inference.vmp.nodes.expfamily.ExponentialFamily method), 169
lower_bound_contribution()	(bayespy.inference.vmp.nodes.gaussian.GaussianGammaARD method), 179
lower_bound_contribution()	(bayespy.inference.vmp.nodes.gaussian.GaussianGammaISO method), 178
lower_bound_contribution()	(bayespy.inference.vmp.nodes.gaussian.GaussianToGaussianGammaISO method), 176
lower_bound_contribution()	(bayespy.inference.vmp.nodes.gaussian.WrapToGaussianGammaARD method), 183
lower_bound_contribution()	(bayespy.inference.vmp.nodes.gaussian.WrapToGaussianGammaISO method), 181
lower_bound_contribution()	(bayespy.nodes.Bernoulli method), 92
lower_bound_contribution()	(bayespy.nodes.Beta method), 112
lower_bound_contribution()	(bayespy.nodes.Binomial method), 96
lower_bound_contribution()	(bayespy.nodes.Categorical method), 100
lower_bound_contribution()	(bayespy.nodes.CategoricalMarkovChain method), 121
lower_bound_contribution()	(bayespy.nodes.Dirichlet method), 116
lower_bound_contribution()	(bayespy.nodes.Exponential method), 76
lower_bound_contribution()	(bayespy.nodes.Gamma method), 68
lower_bound_contribution()	(bayespy.nodes.Gaussian method), 59
lower_bound_contribution()	(bayespy.nodes.GaussianARD method), 64
lower_bound_contribution()	(bayespy.nodes.GaussianGammaARD method), 85
lower_bound_contribution()	(bayespy.nodes.GaussianGammaISO method), 81
lower_bound_contribution()	(bayespy.nodes.GaussianMarkovChain method), 125
lower_bound_contribution()	(bayespy.nodes.GaussianWishart method), 88
lower_bound_contribution()	(bayespy.nodes.Mixture method), 140
lower_bound_contribution()	(bayespy.nodes.Multinomial method), 104
lower_bound_contribution()	(bayespy.nodes.Poisson method), 108
lower_bound_contribution()	(bayespy.nodes.SumMultiply method), 144
lower_bound_contribution()	(bayespy.nodes.SwitchingGaussianMarkovChain method), 130
lower_bound_contribution()	(bayespy.nodes.VaryingGaussianMarkovChain method), 135

- lower_bound_contribution() (bayespy.nodes.Wishart method), 72
- lowerbound() (bayespy.inference.vmp.nodes.exponential.ExponentialFamily method), 170
- lowerbound() (bayespy.inference.vmp.nodes.stochastic.Stochastic method), 166
- lowerbound() (bayespy.nodes.Bernoulli method), 92
- lowerbound() (bayespy.nodes.Beta method), 112
- lowerbound() (bayespy.nodes.Binomial method), 96
- lowerbound() (bayespy.nodes.Categorical method), 100
- lowerbound() (bayespy.nodes.CategoricalMarkovChain method), 121
- lowerbound() (bayespy.nodes.Dirichlet method), 116
- lowerbound() (bayespy.nodes.Exponential method), 76
- lowerbound() (bayespy.nodes.Gamma method), 68
- lowerbound() (bayespy.nodes.Gaussian method), 60
- lowerbound() (bayespy.nodes.GaussianARD method), 64
- lowerbound() (bayespy.nodes.GaussianGammaARD method), 85
- lowerbound() (bayespy.nodes.GaussianGammaISO method), 81
- lowerbound() (bayespy.nodes.GaussianMarkovChain method), 125
- lowerbound() (bayespy.nodes.GaussianWishart method), 88
- lowerbound() (bayespy.nodes.Mixture method), 140
- lowerbound() (bayespy.nodes.Multinomial method), 104
- lowerbound() (bayespy.nodes.Poisson method), 108
- lowerbound() (bayespy.nodes.SwitchingGaussianMarkovChain method), 130
- lowerbound() (bayespy.nodes.VaryingGaussianMarkovChain method), 135
- lowerbound() (bayespy.nodes.Wishart method), 72
- M**
- m_chol() (in module bayespy.utils.misc), 252
- m_chol_inv() (in module bayespy.utils.misc), 252
- m_chol_logdet() (in module bayespy.utils.misc), 252
- m_chol_solve() (in module bayespy.utils.misc), 252
- m_digamma() (in module bayespy.utils.misc), 252
- m_dot() (in module bayespy.utils.linalg), 241
- m_dot() (in module bayespy.utils.misc), 252
- m_outer() (in module bayespy.utils.misc), 252
- m_solve_triangular() (in module bayespy.utils.misc), 252
- make_equal_length() (in module bayespy.utils.misc), 253
- make_equal_ndim() (in module bayespy.utils.misc), 253
- mask() (in module bayespy.utils.random), 245
- maxDiff (bayespy.utils.misc.TestCase attribute), 268
- mean() (in module bayespy.utils.misc), 253
- minimize() (in module bayespy.utils.optimize), 246
- Mixture (class in bayespy.nodes), 137
- mmdot() (in module bayespy.utils.linalg), 241
- Moments (class in bayespy.inference.vmp.nodes.node), 186
- move_plates() (bayespy.inference.vmp.nodes.constant.Constant method), 174
- move_plates() (bayespy.inference.vmp.nodes.deterministic.Deterministic method), 172
- move_plates() (bayespy.inference.vmp.nodes.exponential.ExponentialFamily method), 170
- move_plates() (bayespy.inference.vmp.nodes.gaussian.GaussianGammaARD method), 179
- move_plates() (bayespy.inference.vmp.nodes.gaussian.GaussianGammaISO method), 178
- move_plates() (bayespy.inference.vmp.nodes.gaussian.GaussianToGaussian method), 176
- move_plates() (bayespy.inference.vmp.nodes.gaussian.WrapToGaussianGaussian method), 183
- move_plates() (bayespy.inference.vmp.nodes.gaussian.WrapToGaussianGaussian method), 181
- move_plates() (bayespy.inference.vmp.nodes.gaussian.WrapToGaussianWishart method), 185
- move_plates() (bayespy.inference.vmp.nodes.node.Node method), 164
- move_plates() (bayespy.inference.vmp.nodes.stochastic.Stochastic method), 166
- move_plates() (bayespy.nodes.Bernoulli method), 93
- move_plates() (bayespy.nodes.Beta method), 112
- move_plates() (bayespy.nodes.Binomial method), 97
- move_plates() (bayespy.nodes.Categorical method), 100
- move_plates() (bayespy.nodes.CategoricalMarkovChain method), 121
- move_plates() (bayespy.nodes.Dirichlet method), 116
- move_plates() (bayespy.nodes.Exponential method), 76
- move_plates() (bayespy.nodes.Gamma method), 68
- move_plates() (bayespy.nodes.Gate method), 146
- move_plates() (bayespy.nodes.Gaussian method), 60
- move_plates() (bayespy.nodes.GaussianARD method), 64
- move_plates() (bayespy.nodes.GaussianGammaARD method), 85
- move_plates() (bayespy.nodes.GaussianGammaISO method), 81
- move_plates() (bayespy.nodes.GaussianMarkovChain method), 125
- move_plates() (bayespy.nodes.GaussianWishart method), 88
- move_plates() (bayespy.nodes.Mixture method), 140
- move_plates() (bayespy.nodes.Multinomial method), 104
- move_plates() (bayespy.nodes.Poisson method), 108
- move_plates() (bayespy.nodes.SumMultiply method), 144
- move_plates() (bayespy.nodes.SwitchingGaussianMarkovChain method), 130
- move_plates() (bayespy.nodes.VaryingGaussianMarkovChain method), 136
- move_plates() (bayespy.nodes.Wishart method), 72
- moveaxis() (in module bayespy.utils.misc), 253

Multinomial (class in bayespy.nodes), 102
 MultinomialDistribution (class in bayespy.inference.vmp.nodes.multinomial), 236
 MultinomialMoments (class in bayespy.inference.vmp.nodes.multinomial), 199
 multiply_shapes() (in module bayespy.utils.misc), 253
 mvdot() (in module bayespy.utils.linalg), 241

N

nans() (in module bayespy.utils.misc), 253
 nested_iterator() (in module bayespy.utils.misc), 253
 Node (class in bayespy.inference.vmp.nodes.node), 163
 nodes() (bayespy.inference.vmp.transformations.RotateGaussian method), 150
 nodes() (bayespy.inference.vmp.transformations.RotateGaussianARD method), 151
 nodes() (bayespy.inference.vmp.transformations.RotateGaussianMarkovChain method), 153
 nodes() (bayespy.inference.vmp.transformations.RotateMultinomial method), 156
 nodes() (bayespy.inference.vmp.transformations.RotateSwitchingMarkovChain method), 154
 nodes() (bayespy.inference.vmp.transformations.RotateVaryingMarkovChain method), 155

O

observe() (bayespy.inference.vmp.nodes.expfamily.ExponentialFamily method), 170
 observe() (bayespy.inference.vmp.nodes.stochastic.Stochastic method), 166
 observe() (bayespy.nodes.Bernoulli method), 93
 observe() (bayespy.nodes.Beta method), 112
 observe() (bayespy.nodes.Binomial method), 97
 observe() (bayespy.nodes.Categorical method), 100
 observe() (bayespy.nodes.CategoricalMarkovChain method), 121
 observe() (bayespy.nodes.Dirichlet method), 116
 observe() (bayespy.nodes.Exponential method), 76
 observe() (bayespy.nodes.Gamma method), 69
 observe() (bayespy.nodes.Gaussian method), 60
 observe() (bayespy.nodes.GaussianARD method), 64
 observe() (bayespy.nodes.GaussianGammaARD method), 85
 observe() (bayespy.nodes.GaussianGammaISO method), 81
 observe() (bayespy.nodes.GaussianMarkovChain method), 126
 observe() (bayespy.nodes.GaussianWishart method), 89
 observe() (bayespy.nodes.Mixture method), 140
 observe() (bayespy.nodes.Multinomial method), 105
 observe() (bayespy.nodes.Poisson method), 108

observe() (bayespy.nodes.SwitchingGaussianMarkovChain method), 131
 observe() (bayespy.nodes.VaryingGaussianMarkovChain method), 136
 observe() (bayespy.nodes.Wishart method), 72
 orth() (in module bayespy.utils.random), 245
 outer() (in module bayespy.utils.linalg), 241

P

pdf() (bayespy.inference.vmp.nodes.expfamily.ExponentialFamily method), 170
 pdf() (bayespy.nodes.Bernoulli method), 93
 pdf() (bayespy.nodes.Beta method), 113
 pdf() (bayespy.nodes.Binomial method), 97
 pdf() (bayespy.nodes.Categorical method), 101
 pdf() (bayespy.nodes.CategoricalMarkovChain method), 121
 pdf() (bayespy.nodes.Dirichlet method), 117
 pdf() (bayespy.nodes.Exponential method), 77
 pdf() (bayespy.nodes.Gamma method), 69
 pdf() (bayespy.nodes.Gaussian method), 60
 pdf() (bayespy.nodes.GaussianARD method), 64
 pdf() (bayespy.nodes.GaussianGammaARD method), 85
 pdf() (bayespy.nodes.GaussianGammaISO method), 81
 pdf() (bayespy.nodes.GaussianMarkovChain method), 126
 pdf() (bayespy.nodes.GaussianWishart method), 89
 pdf() (bayespy.nodes.Mixture method), 140
 pdf() (bayespy.nodes.Multinomial method), 105
 pdf() (bayespy.nodes.Poisson method), 108
 pdf() (bayespy.nodes.SwitchingGaussianMarkovChain method), 131
 pdf() (bayespy.nodes.VaryingGaussianMarkovChain method), 136
 pdf() (bayespy.nodes.Wishart method), 72
 pdf() (in module bayespy.plot), 156
 PDFPlotter (class in bayespy.plot), 159
 plates (bayespy.inference.vmp.nodes.constant.Constant attribute), 175
 plates (bayespy.inference.vmp.nodes.deterministic.Deterministic attribute), 173
 plates (bayespy.inference.vmp.nodes.expfamily.ExponentialFamily attribute), 171
 plates (bayespy.inference.vmp.nodes.gaussian.GaussianGammaARDToGaussian attribute), 180
 plates (bayespy.inference.vmp.nodes.gaussian.GaussianGammaISOToGaussian attribute), 178
 plates (bayespy.inference.vmp.nodes.gaussian.GaussianToGaussianGamma attribute), 176
 plates (bayespy.inference.vmp.nodes.gaussian.WrapToGaussianGammaARD attribute), 183
 plates (bayespy.inference.vmp.nodes.gaussian.WrapToGaussianGammaISO attribute), 182

plates (bayespy.inference.vmp.nodes.gaussian.WrapToGaussianWishart attribute), 185

plates (bayespy.inference.vmp.nodes.node.Node attribute), 165

plates (bayespy.inference.vmp.nodes.stochastic.Stochastic attribute), 167

plates (bayespy.nodes.Bernoulli attribute), 94

plates (bayespy.nodes.Beta attribute), 114

plates (bayespy.nodes.Binomial attribute), 98

plates (bayespy.nodes.Categorical attribute), 102

plates (bayespy.nodes.CategoricalMarkovChain attribute), 122

plates (bayespy.nodes.Dirichlet attribute), 118

plates (bayespy.nodes.Exponential attribute), 78

plates (bayespy.nodes.Gamma attribute), 70

plates (bayespy.nodes.Gate attribute), 146

plates (bayespy.nodes.Gaussian attribute), 61

plates (bayespy.nodes.GaussianARD attribute), 66

plates (bayespy.nodes.GaussianGammaARD attribute), 86

plates (bayespy.nodes.GaussianGammaISO attribute), 82

plates (bayespy.nodes.GaussianMarkovChain attribute), 127

plates (bayespy.nodes.GaussianWishart attribute), 90

plates (bayespy.nodes.Mixture attribute), 141

plates (bayespy.nodes.Multinomial attribute), 106

plates (bayespy.nodes.Poisson attribute), 110

plates (bayespy.nodes.SumMultiply attribute), 144

plates (bayespy.nodes.SwitchingGaussianMarkovChain attribute), 132

plates (bayespy.nodes.VaryingGaussianMarkovChain attribute), 137

plates (bayespy.nodes.Wishart attribute), 74

plates_from_parent() (bayespy.inference.vmp.nodes.bernoulli.BernoulliDistribution method), 230

plates_from_parent() (bayespy.inference.vmp.nodes.beta.BetaDistribution method), 227

plates_from_parent() (bayespy.inference.vmp.nodes.binomial.BinomialDistribution method), 232

plates_from_parent() (bayespy.inference.vmp.nodes.categorical.CategoricalDistribution method), 233

plates_from_parent() (bayespy.inference.vmp.nodes.categorical_markov_chain.CategoricalMarkovChainDistribution method), 235

plates_from_parent() (bayespy.inference.vmp.nodes.dirichlet.DirichletDistribution method), 228

plates_from_parent() (bayespy.inference.vmp.nodes.expfamily.ExponentialFamilyDistribution method), 204

plates_from_parent() (bayespy.inference.vmp.nodes.gamma.GammaDistribution method), 222

plates_from_parent() (bayespy.inference.vmp.nodes.gaussian.GaussianARDDistribution method), 208

plates_from_parent() (bayespy.inference.vmp.nodes.gaussian.GaussianGammaARD method), 206

plates_to_parent() (bayespy.inference.vmp.nodes.bernoulli.BernoulliDistribution method), 230

plates_to_parent() (bayespy.inference.vmp.nodes.beta.BetaDistribution method), 227

plates_to_parent() (bayespy.inference.vmp.nodes.binomial.BinomialDistribution method), 232

plates_to_parent() (bayespy.inference.vmp.nodes.categorical.CategoricalDistribution method), 233

plates_to_parent() (bayespy.inference.vmp.nodes.categorical_markov_chain.CategoricalMarkovChainDistribution method), 235

plates_to_parent() (bayespy.inference.vmp.nodes.dirichlet.DirichletDistribution method), 228

plates_to_parent() (bayespy.inference.vmp.nodes.expfamily.ExponentialFamilyDistribution method), 204

plates_to_parent() (bayespy.inference.vmp.nodes.gamma.GammaDistribution method), 215

plates_to_parent() (bayespy.inference.vmp.nodes.gaussian.GaussianARDDistribution method), 215

plates_to_parent() (bayespy.inference.vmp.nodes.gaussian.GaussianGammaARD method), 215

plates_to_parent() (bayespy.inference.vmp.nodes.gaussian.GaussianGammaISO method), 215

plates_to_parent() (bayespy.inference.vmp.nodes.gaussian.GaussianMarkovChain method), 215

plates_to_parent() (bayespy.inference.vmp.nodes.gaussian.GaussianWishart method), 215

plates_to_parent() (bayespy.inference.vmp.nodes.multinomial.MultinomialDistribution method), 237

plates_to_parent() (bayespy.inference.vmp.nodes.poisson.PoissonDistribution method), 239

plates_to_parent() (bayespy.inference.vmp.nodes.stochastic.Distribution method), 202

plates_to_parent() (bayespy.inference.vmp.nodes.wishart.WishartDistribution method), 224

plates_to_parent() (bayespy.inference.vmp.nodes.poisson.PoissonDistribution method), 239

plates_to_parent() (bayespy.inference.vmp.nodes.stochastic.DistributionByNodes method), 202

plates_to_parent() (bayespy.inference.vmp.nodes.wishart.WishartDistribution method), 224

plot() (bayespy.inference.VB method), 148

plot() (bayespy.inference.vmp.nodes.constant.Constant method), 174

plot() (bayespy.inference.vmp.nodes.deterministic.Deterministic method), 173

plot() (bayespy.inference.vmp.nodes.expfamily.ExponentialFamily method), 170

plot() (bayespy.inference.vmp.nodes.gaussian.GaussianGammaARDToGaussianWishart method), 180

plot() (bayespy.inference.vmp.nodes.gaussian.GaussianGammaISOToGaussianGammaARD method), 178

plot() (bayespy.inference.vmp.nodes.gaussian.GaussianToGaussianGammaISO method), 176

plot() (bayespy.inference.vmp.nodes.gaussian.WrapToGaussianGammaARD method), 183

plot() (bayespy.inference.vmp.nodes.gaussian.WrapToGaussianGammaISO method), 181

plot() (bayespy.inference.vmp.nodes.gaussian.WrapToGaussianWishart method), 185

plot() (bayespy.inference.vmp.nodes.node.Node method), 164

plot() (bayespy.inference.vmp.nodes.stochastic.Stochastic method), 167

plot() (bayespy.nodes.Bernoulli method), 93

plot() (bayespy.nodes.Beta method), 113

plot() (bayespy.nodes.Binomial method), 97

plot() (bayespy.nodes.Categorical method), 101

plot() (bayespy.nodes.CategoricalMarkovChain method), 121

plot() (bayespy.nodes.Dirichlet method), 117

plot() (bayespy.nodes.Exponential method), 77

plot() (bayespy.nodes.Gamma method), 69

plot() (bayespy.nodes.Gate method), 146

plot() (bayespy.nodes.Gaussian method), 60

plot() (bayespy.nodes.GaussianARD method), 65

plot() (bayespy.nodes.GaussianGammaARD method), 85

plot() (bayespy.nodes.GaussianGammaISO method), 81

plot() (bayespy.nodes.GaussianMarkovChain method), 126

plot() (bayespy.nodes.GaussianWishart method), 89

plot() (bayespy.nodes.Mixture method), 141

plot() (bayespy.nodes.Multinomial method), 105

plot() (bayespy.nodes.Poisson method), 109

plot() (bayespy.nodes.SumMultiply method), 144

plot() (bayespy.nodes.SwitchingGaussianMarkovChain method), 131

plot() (bayespy.nodes.VaryingGaussianMarkovChain method), 136

plot() (bayespy.nodes.Wishart method), 73

plot() (in module bayespy.plot), 157

DistributionByNodes() (bayespy.inference.VB method), 148

Plot() (bayespy.nodes.GaussianGammaISO method), 81

Plotter (class in bayespy.plot), 157

Poisson (class in bayespy.nodes), 106

PoissonDistribution (class in bayespy.inference.vmp.nodes.poisson), 237

PoissonMoments (class in bayespy.inference.vmp.nodes.poisson), 200

Random() (bayespy.inference.vmp.nodes.bernoulli.BernoulliDistribution method), 230

random() (bayespy.inference.vmp.nodes.beta.BetaDistribution method), 227

random() (bayespy.inference.vmp.nodes.binomial.BinomialDistribution method), 232

random() (bayespy.inference.vmp.nodes.categorical.CategoricalDistribution method), 234

random() (bayespy.inference.vmp.nodes.categorical_markov_chain.CategoricalMarkovChain method), 235

random() (bayespy.inference.vmp.nodes.dirichlet.DirichletDistribution method), 228

random() (bayespy.inference.vmp.nodes.expfamily.ExponentialFamily method), 170

random() (bayespy.inference.vmp.nodes.expfamily.ExponentialFamilyDistribution method), 204

random() (bayespy.inference.vmp.nodes.gamma.GammaDistribution method), 223

random() (bayespy.inference.vmp.nodes.gaussian.GaussianARDDistribution method), 208

random() (bayespy.inference.vmp.nodes.gaussian.GaussianDistribution method), 206

random() (bayespy.inference.vmp.nodes.gaussian.GaussianGammaARDDistribution method), 211

random() (bayespy.inference.vmp.nodes.gaussian.GaussianGammaISODistribution method), 210

random() (bayespy.inference.vmp.nodes.gaussian.GaussianWishartDistribution method), 213

random() (bayespy.inference.vmp.nodes.gaussian_markov_chain.GaussianMarkovChain method), 215

random() (bayespy.inference.vmp.nodes.gaussian_markov_chain.SwitchingGaussianMarkovChain method), 219

random() (bayespy.inference.vmp.nodes.gaussian_markov_chain.VaryingGaussianMarkovChain method), 221

random() (bayespy.inference.vmp.nodes.multinomial.MultinomialDistribution method), 237

random() (bayespy.inference.vmp.nodes.poisson.PoissonDistribution method), 239

random() (bayespy.inference.vmp.nodes.stochastic.DistributionByNodes method), 202

[random\(\)](#) (bayespy.inference.vmp.nodes.stochastic.Stochastic method), 167
[random\(\)](#) (bayespy.inference.vmp.nodes.wishart.WishartDistribution method), 224
[random\(\)](#) (bayespy.nodes.Bernoulli method), 93
[random\(\)](#) (bayespy.nodes.Beta method), 113
[random\(\)](#) (bayespy.nodes.Binomial method), 97
[random\(\)](#) (bayespy.nodes.Categorical method), 101
[random\(\)](#) (bayespy.nodes.CategoricalMarkovChain method), 121
[random\(\)](#) (bayespy.nodes.Dirichlet method), 117
[random\(\)](#) (bayespy.nodes.Exponential method), 77
[random\(\)](#) (bayespy.nodes.Gamma method), 69
[random\(\)](#) (bayespy.nodes.Gaussian method), 60
[random\(\)](#) (bayespy.nodes.GaussianARD method), 65
[random\(\)](#) (bayespy.nodes.GaussianGammaARD method), 85
[random\(\)](#) (bayespy.nodes.GaussianGammaISO method), 81
[random\(\)](#) (bayespy.nodes.GaussianMarkovChain method), 126
[random\(\)](#) (bayespy.nodes.GaussianWishart method), 89
[random\(\)](#) (bayespy.nodes.Mixture method), 141
[random\(\)](#) (bayespy.nodes.Multinomial method), 105
[random\(\)](#) (bayespy.nodes.Poisson method), 109
[random\(\)](#) (bayespy.nodes.SwitchingGaussianMarkovChain method), 131
[random\(\)](#) (bayespy.nodes.VaryingGaussianMarkovChain method), 136
[random\(\)](#) (bayespy.nodes.Wishart method), 73
[remove_whitespace\(\)](#) (in module bayespy.utils.misc), 253
[repeat_to_shape\(\)](#) (in module bayespy.utils.misc), 253
[rmse\(\)](#) (in module bayespy.utils.misc), 253
[rotate\(\)](#) (bayespy.inference.vmp.transformations.RotateGaussian method), 150
[rotate\(\)](#) (bayespy.inference.vmp.transformations.RotateGaussianARD method), 152
[rotate\(\)](#) (bayespy.inference.vmp.transformations.RotateGaussianMarkovChain method), 153
[rotate\(\)](#) (bayespy.inference.vmp.transformations.RotateMultiple method), 156
[rotate\(\)](#) (bayespy.inference.vmp.transformations.RotateSwitchingMarkovChain method), 154
[rotate\(\)](#) (bayespy.inference.vmp.transformations.RotateVaryingMarkovChain method), 155
[rotate\(\)](#) (bayespy.inference.vmp.transformations.RotationOptimizer method), 149
[rotate\(\)](#) (bayespy.nodes.Gaussian method), 60
[rotate\(\)](#) (bayespy.nodes.GaussianARD method), 65
[rotate\(\)](#) (bayespy.nodes.GaussianMarkovChain method), 126
[rotate\(\)](#) (bayespy.nodes.SwitchingGaussianMarkovChain method), 131
[rotate\(\)](#) (bayespy.nodes.VaryingGaussianMarkovChain method), 136
[rotate_plates\(\)](#) (bayespy.nodes.GaussianARD method), 65
[RotateGaussian](#) (class in bayespy.inference.vmp.transformations), 150
[RotateGaussianARD](#) (class in bayespy.inference.vmp.transformations), 151
[RotateGaussianMarkovChain](#) (class in bayespy.inference.vmp.transformations), 152
[RotateMultiple](#) (class in bayespy.inference.vmp.transformations), 155
[RotateSwitchingMarkovChain](#) (class in bayespy.inference.vmp.transformations), 153
[RotateVaryingMarkovChain](#) (class in bayespy.inference.vmp.transformations), 154
[RotationOptimizer](#) (class in bayespy.inference.vmp.transformations), 149
[rts_smoother\(\)](#) (in module bayespy.utils.misc), 254
[run\(\)](#) (bayespy.utils.misc.TestCase method), 267

S

[save\(\)](#) (bayespy.inference.VB method), 148
[save\(\)](#) (bayespy.inference.vmp.nodes.expfamily.ExponentialFamily method), 170
[save\(\)](#) (bayespy.inference.vmp.nodes.stochastic.Stochastic method), 167
[save\(\)](#) (bayespy.nodes.Bernoulli method), 93
[save\(\)](#) (bayespy.nodes.Beta method), 113
[save\(\)](#) (bayespy.nodes.Binomial method), 97
[save\(\)](#) (bayespy.nodes.Categorical method), 101
[save\(\)](#) (bayespy.nodes.CategoricalMarkovChain method), 122
[save\(\)](#) (bayespy.nodes.Dirichlet method), 117
[save\(\)](#) (bayespy.nodes.Exponential method), 77
[save\(\)](#) (bayespy.nodes.Gamma method), 69
[save\(\)](#) (bayespy.nodes.Gaussian method), 61
[save\(\)](#) (bayespy.nodes.GaussianARD method), 65
[save\(\)](#) (bayespy.nodes.GaussianGammaARD method), 85
[save\(\)](#) (bayespy.nodes.GaussianGammaISO method), 82
[save\(\)](#) (bayespy.nodes.GaussianMarkovChain method), 126
[save\(\)](#) (bayespy.nodes.GaussianWishart method), 89
[save\(\)](#) (bayespy.nodes.Mixture method), 141
[save\(\)](#) (bayespy.nodes.Multinomial method), 105
[save\(\)](#) (bayespy.nodes.Poisson method), 109

save() (bayespy.nodes.SwitchingGaussianMarkovChain method), 131
 save() (bayespy.nodes.VaryingGaussianMarkovChain method), 136
 save() (bayespy.nodes.Wishart method), 73
 set_autosave() (bayespy.inference.VB method), 148
 set_plotter() (bayespy.inference.vmp.nodes.constant.Constant method), 175
 set_plotter() (bayespy.inference.vmp.nodes.deterministic.Deterministic method), 173
 set_plotter() (bayespy.inference.vmp.nodes.expfamily.ExponentialFamily method), 170
 set_plotter() (bayespy.inference.vmp.nodes.gaussian.GaussianGammaARDtoGaussianWishart method), 180
 set_plotter() (bayespy.inference.vmp.nodes.gaussian.GaussianGammaISOtoGaussianGammaARD method), 178
 set_plotter() (bayespy.inference.vmp.nodes.gaussian.GaussianISOtoGaussianISO method), 176
 set_plotter() (bayespy.inference.vmp.nodes.gaussian.WrapToGaussianGammaARD method), 183
 set_plotter() (bayespy.inference.vmp.nodes.gaussian.WrapToGaussianGammaISO method), 181
 set_plotter() (bayespy.inference.vmp.nodes.gaussian.WrapToGaussianWishart method), 185
 set_plotter() (bayespy.inference.vmp.nodes.node.Node method), 165
 set_plotter() (bayespy.inference.vmp.nodes.stochastic.Stochastic method), 167
 set_plotter() (bayespy.nodes.Bernoulli method), 93
 set_plotter() (bayespy.nodes.Beta method), 113
 set_plotter() (bayespy.nodes.Binomial method), 97
 set_plotter() (bayespy.nodes.Categorical method), 101
 set_plotter() (bayespy.nodes.CategoricalMarkovChain method), 122
 set_plotter() (bayespy.nodes.Dirichlet method), 117
 set_plotter() (bayespy.nodes.Exponential method), 77
 set_plotter() (bayespy.nodes.Gamma method), 69
 set_plotter() (bayespy.nodes.Gate method), 146
 set_plotter() (bayespy.nodes.Gaussian method), 61
 set_plotter() (bayespy.nodes.GaussianARD method), 65
 set_plotter() (bayespy.nodes.GaussianGammaARD method), 85
 set_plotter() (bayespy.nodes.GaussianGammaISO method), 82
 set_plotter() (bayespy.nodes.GaussianMarkovChain method), 126
 set_plotter() (bayespy.nodes.GaussianWishart method), 89
 set_plotter() (bayespy.nodes.Mixture method), 141
 set_plotter() (bayespy.nodes.Multinomial method), 105
 set_plotter() (bayespy.nodes.Poisson method), 109
 set_plotter() (bayespy.nodes.SumMultiply method), 144
 set_plotter() (bayespy.nodes.SwitchingGaussianMarkovChain method), 131
 set_plotter() (bayespy.nodes.VaryingGaussianMarkovChain method), 136
 set_plotter() (bayespy.nodes.Wishart method), 73
 setup() (bayespy.inference.vmp.transformations.RotateGaussian method), 150
 setup() (bayespy.inference.vmp.transformations.RotateGaussianARD method), 152
 setup() (bayespy.inference.vmp.transformations.RotateGaussianMarkovChain method), 153
 setup() (bayespy.inference.vmp.transformations.RotateMultiple method), 156
 setup() (bayespy.inference.vmp.transformations.RotateSwitchingMarkovChain method), 155
 setup() (bayespy.inference.vmp.transformations.RotateVaryingMarkovChain method), 154
 setUp() (bayespy.utils.misc.TestCase method), 267
 setUp() (bayespy.utils.misc.TestCase method), 267
 shortDescription() (bayespy.utils.misc.TestCase method), 267
 show() (bayespy.nodes.Bernoulli method), 93
 show() (bayespy.nodes.Beta method), 113
 show() (bayespy.nodes.Binomial method), 97
 show() (bayespy.nodes.Categorical method), 101
 show() (bayespy.nodes.CategoricalMarkovChain method), 122
 show() (bayespy.nodes.Dirichlet method), 117
 show() (bayespy.nodes.Exponential method), 77
 show() (bayespy.nodes.Gamma method), 69
 show() (bayespy.nodes.Gaussian method), 61
 show() (bayespy.nodes.GaussianARD method), 65
 show() (bayespy.nodes.GaussianGammaISO method), 82
 show() (bayespy.nodes.GaussianMarkovChain method), 126
 show() (bayespy.nodes.GaussianWishart method), 89
 show() (bayespy.nodes.Multinomial method), 105
 show() (bayespy.nodes.Poisson method), 109
 show() (bayespy.nodes.SwitchingGaussianMarkovChain method), 131
 show() (bayespy.nodes.VaryingGaussianMarkovChain method), 136
 show() (bayespy.nodes.Wishart method), 73
 skipTest() (bayespy.utils.misc.TestCase method), 268
 solve() (bayespy.utils.misc.CholeskyDense method), 256
 solve() (bayespy.utils.misc.CholeskySparse method), 257
 solve_triangular() (in module bayespy.utils.linalg), 242
 sphere() (in module bayespy.utils.random), 245
 squeeze() (in module bayespy.utils.misc), 254
 squeeze_to_dim() (in module bayespy.utils.misc), 254
 Stochastic (class in bayespy.inference.vmp.nodes.stochastic), 165
 subTest() (bayespy.utils.misc.TestCase method), 268
 sum_multiply() (in module bayespy.utils.misc), 254
 sum_product() (in module bayespy.utils.misc), 254
 sum_to_dim() (in module bayespy.utils.misc), 255

sum_to_shape() (in module bayespy.utils.misc), 255

SumMultiply (class in bayespy.nodes), 142

svd() (in module bayespy.utils.random), 245

SwitchingGaussianMarkovChain (class in bayespy.nodes), 127

SwitchingGaussianMarkovChainDistribution (class in bayespy.inference.vmp.nodes.gaussian_markov_chain), 215

symm() (in module bayespy.utils.misc), 255

T

T() (in module bayespy.utils.misc), 248

t_logpdf() (in module bayespy.utils.random), 246

tearDown() (bayespy.utils.misc.TestCase method), 268

tearDownClass() (bayespy.utils.misc.TestCase method), 268

tempfile() (in module bayespy.utils.misc), 255

TestCase (class in bayespy.utils.misc), 257

trace_solve_gradient() (bayespy.utils.misc.CholeskyDense method), 256

trace_solve_gradient() (bayespy.utils.misc.CholeskySparse method), 257

tracedot() (in module bayespy.utils.linalg), 242

trueos() (in module bayespy.utils.misc), 255

U

unique() (in module bayespy.utils.misc), 255

unobserve() (bayespy.inference.vmp.nodes.expfamily.ExponentialFamily method), 170

unobserve() (bayespy.inference.vmp.nodes.stochastic.Stochastic method), 167

unobserve() (bayespy.nodes.Bernoulli method), 93

unobserve() (bayespy.nodes.Beta method), 113

unobserve() (bayespy.nodes.Binomial method), 97

unobserve() (bayespy.nodes.Categorical method), 101

unobserve() (bayespy.nodes.CategoricalMarkovChain method), 122

unobserve() (bayespy.nodes.Dirichlet method), 117

unobserve() (bayespy.nodes.Exponential method), 77

unobserve() (bayespy.nodes.Gamma method), 69

unobserve() (bayespy.nodes.Gaussian method), 61

unobserve() (bayespy.nodes.GaussianARD method), 65

unobserve() (bayespy.nodes.GaussianGammaARD method), 86

unobserve() (bayespy.nodes.GaussianGammaISO method), 82

unobserve() (bayespy.nodes.GaussianMarkovChain method), 126

unobserve() (bayespy.nodes.GaussianWishart method), 89

unobserve() (bayespy.nodes.Mixture method), 141

unobserve() (bayespy.nodes.Multinomial method), 105

unobserve() (bayespy.nodes.Poisson method), 109

unobserve() (bayespy.nodes.SwitchingGaussianMarkovChain method), 131

unobserve() (bayespy.nodes.VaryingGaussianMarkovChain method), 137

unobserve() (bayespy.nodes.Wishart method), 73

update() (bayespy.inference.VB method), 148

update() (bayespy.inference.vmp.nodes.expfamily.ExponentialFamily method), 171

update() (bayespy.inference.vmp.nodes.stochastic.Stochastic method), 167

update() (bayespy.nodes.Bernoulli method), 94

update() (bayespy.nodes.Beta method), 113

update() (bayespy.nodes.Binomial method), 98

update() (bayespy.nodes.Categorical method), 101

update() (bayespy.nodes.CategoricalMarkovChain method), 122

update() (bayespy.nodes.Dirichlet method), 117

update() (bayespy.nodes.Exponential method), 77

update() (bayespy.nodes.Gamma method), 69

update() (bayespy.nodes.Gaussian method), 61

update() (bayespy.nodes.GaussianARD method), 65

update() (bayespy.nodes.GaussianGammaARD method), 86

update() (bayespy.nodes.GaussianGammaISO method), 82

update() (bayespy.nodes.GaussianMarkovChain method), 127

update() (bayespy.nodes.GaussianWishart method), 89

update() (bayespy.nodes.Mixture method), 141

update() (bayespy.nodes.Multinomial method), 105

update() (bayespy.nodes.Poisson method), 109

update() (bayespy.nodes.SwitchingGaussianMarkovChain method), 132

update() (bayespy.nodes.VaryingGaussianMarkovChain method), 137

update() (bayespy.nodes.Wishart method), 73

V

VaryingGaussianMarkovChain (class in bayespy.nodes), 132

VaryingGaussianMarkovChainDistribution (class in bayespy.inference.vmp.nodes.gaussian_markov_chain), 219

VB (class in bayespy.inference), 147

vb_optimize() (in module bayespy.utils.misc), 255

vb_optimize_nodes() (in module bayespy.utils.misc), 255

W

Wishart (class in bayespy.nodes), 70

wishart_rand() (in module bayespy.utils.random), 246

WishartDistribution (class in bayespy.inference.vmp.nodes.wishart), 223

WishartMoments (class in bayespy.inference.vmp.nodes.wishart), 193

WrapToGaussianGammaARD (class in
bayespy.inference.vmp.nodes.gaussian), [182](#)

WrapToGaussianGammaISO (class in
bayespy.inference.vmp.nodes.gaussian), [180](#)

WrapToGaussianWishart (class in
bayespy.inference.vmp.nodes.gaussian), [183](#)

write_to_hdf5() (in module bayespy.utils.misc), [255](#)

Z

zipper_merge() (in module bayespy.utils.misc), [256](#)