

---

# BayesPy Documentation

*Release 0.1*

**Jaakko Luttinen**

May 06, 2014



## CONTENTS

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Project information . . . . .	1
1.2	Similar projects . . . . .	1
<b>2</b>	<b>User guide</b>	<b>3</b>
2.1	Installation . . . . .	3
2.2	Quick start guide . . . . .	5
2.3	Nodes . . . . .	10
<b>3</b>	<b>Examples</b>	<b>11</b>
3.1	Example: Mixture of Gaussians . . . . .	11
3.2	Example: Principal component analysis . . . . .	11
3.3	Linear state-space model . . . . .	11
<b>4</b>	<b>Developer guide</b>	<b>17</b>
4.1	Variational message passing . . . . .	17
<b>5</b>	<b>API</b>	<b>21</b>
5.1	Stochastic nodes . . . . .	21
5.2	Deterministic nodes . . . . .	46
5.3	Base nodes . . . . .	46
<b>6</b>	<b>Indices and tables</b>	<b>55</b>
	<b>Index</b>	<b>57</b>



## INTRODUCTION

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

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

### 1.1 Project information

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](mailto:bayespy@googlegroups.com)
- IRC: `#bayespy` @ [freenode](https://freenode.net)
- Author: Jaakko Luttinen [jaakko.luttinen@iki.fi](mailto: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.probp.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.

## 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 ( $\geq 1.2$ )
- 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 ( $\geq 12.10$ ), 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
```

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
```

The documentation can be compiled to HTML and PDF by running the following commands in the `doc` directory:

```
make html
make latexpdf
```

## 2.1.4 Converting notebooks

The documentation uses IPython notebooks for the examples. This is a convenient format for sharing Python examples with comments. The notebooks can be converted, for instance, to documentation files or Python scripts. BayesPy repository contains those notebook files (`.ipynb`) and their conversions to RST format for the documentation. If you want to convert the notebooks into RST files, Python scripts or some other format yourself, follow these instructions. First, the following packages are required:



```
pip install ipython pyzmq
```

You need quite a recent IPython. You may also need to install Pandoc. In Ubuntu, this can be done as:

```
sudo aptitude install pandoc
```

Now, the notebooks can be converted to RST for the documentation by running the following command in the `doc` directory:

```
make notebooks
```

Or you can convert the notebooks to RST or Python (or something else) for your own use:

```
ipython nbconvert --to rst doc/source/_notebooks/*.ipynb
ipython nbconvert --to python doc/source/_notebooks/*.ipynb
```

The Python scripts can be used to run the examples as such. There are also more formats available in case you want the examples in HTML, LaTeX, or some other format.

You can also open the notebooks interactively in a web browser by going to the notebooks directory and running the IPython notebook:

```
cd doc/source/_notebooks
ipython notebook
```

This should run a simple server and open a web browser.

## 2.2 Quick start guide

- Construct the model (Bayesian network)
- Put the data in
- Run inference
- Examine posterior results

### 2.2.1 Simple example

Let's begin with a simple example which shows the basic steps. In this case, we do not use any real data but generate some toy data. The dataset consists of ten samples from the Gaussian distribution with mean 5 and standard deviation 10:

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

Now, given this data we would like to estimate the mean and the standard deviation. We can construct a simple model shown below as a directed factor graph. Alternatively, the model can also be defined using explicit mathematical notation:

$$p(\mathbf{y}|\mu, \tau) = \prod_{n=1}^{10} \mathcal{N}(y_n|\mu, \tau)$$
$$p(\mu) = \mathcal{N}(\mu|0, 10^{-3})$$
$$p(\tau) = \mathcal{G}(\tau|10^{-3}, 10^{-3})$$

Note that we parameterize the normal distribution using the mean and the precision (i.e., the inverse of the variance). The model can be constructed in BayesPy as follows:

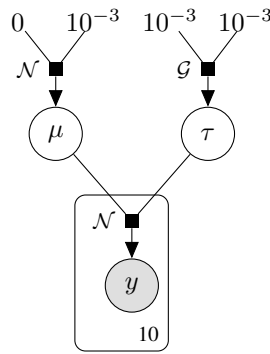


Figure 2.1: Directed factor graph of the example model.

```
import bayespy as bp
mu = bp.nodes.Normal(0, 1e-3)
tau = bp.nodes.Gamma(1e-3, 1e-3)
y = bp.nodes.Normal(mu, tau, plates=(10,))
```

This is quite self-explanatory given the model definitions above. Now, we use the generated data:

```
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 the nodes and the inference algorithm can be run as long as wanted (20 iterations in this case):

```
from bayespy.inference import VB
Q = VB(mu, tau, y)
Q.update(repeat=20)
```

In VB, the true posterior  $p(\mu, \tau | \mathbf{y})$  is approximated with a factorized distribution  $q(\mu)q(\tau)$ . The resulting approximate posterior distributions  $q(\mu)$  and  $q(\tau)$  can be examined as:

```
mu.show()
tau.show()
```

```
import bayespy.plotting as plt
```

---

## Todo

Add an example of visualizing the results.

---

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.

## 2.2.2 Constructing the model

In BayesPy, the model is constructed by creating nodes which form a network. Roughly speaking, a node corresponds to a random variable from a specific probability distribution. In the example, `mu` was `Normal` node corresponding to  $\mu$  from the normal distribution. However, a node can also correspond to a set of random variables or nodes can be deterministic not corresponding to any random variable.

When you create a node, you give its parents as parameters. The role and the number of the parents depend on the node. For instance, `Normal` node takes two parents (mean and precision) and `Gamma` node takes two parents (scale and rate).

**Warning:** Currently, it is important that the parent has the correct node type, because the model construction and VB inference engine are not yet separated. For instance, the parents mean and precision of `Normal` node must be `Normal` and `Gamma` nodes (or other nodes that have similar output), respectively. Thus, currently one can build only conjugate-exponential family models.

## Name and plates

In general, the nodes take some optional parameters: `name` and `plates`. The parameter `name` is used to give a name for the variable. The parameter `plates` is used to define plates, that is, a repetitive collection of nodes that are independent given the parents. For instance, the following set of i.i.d. random variables

$$y_{mn} \sim \mathcal{N}(\mu, \tau), \quad m = 1, \dots, 10, \quad n = 1, \dots, 30$$

would be created as

```
y = bp.nodes.Normal(mu, tau, plates=(10, 30))
```

It is also possible that the parents have plates. The validity of the plates between a child and a parent is checked by comparing the plates plate-wise from the trailing plates 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 and two parent nodes.

node	plates					
parent1		9	1	5	1	10
parent2			15	5	1	1
child	5	9	15	5	1	10

For instance, a model

$$\begin{aligned} \mu_m &\sim \mathcal{N}(0, 10^{-3}), \\ \tau_n &\sim \mathcal{G}(10^{-3}, 10^{-3}), \\ y_{mn} &\sim \mathcal{N}(\mu_m, \tau_n), \quad m = 1, \dots, 10, \quad n = 1, \dots, 30 \end{aligned}$$

could be created as

```
mu = bp.nodes.Normal(0, 1e-3, plates=(10, 1))
tau = bp.nodes.Gamma(1e-3, 1e-3, plates=(30,))
y = bp.nodes.Normal(mu, tau, plates=(10, 30))
```

## Multi-dimensional nodes

Sometimes a random variable is multi-dimensional. For instance, a multivariate normal distribution is a probability distribution for vectors. Quite often, the dimensionality can be deduced implicitly from the parents, thus the user may not need to provide it explicitly. However, it is important to know that the values are stored in a NumPy array where the plates are the leading axes and the dimensions are the trailing axes. This becomes relevant, for instance, when providing the data for an observed multi-dimensional node. To make a clear distinction between scalar and multi-dimensional distributions, there is often a multi-dimensional counterpart of a scalar node. For instance, the normal distribution for scalars is provided by the node `Normal`, but the node for the multivariate normal distribution is `Gaussian`. Below is a more complete table of correspondence.

Scalar	Multi-dimensional
Normal	Gaussian
Gamma	Wishart
Bernoulli	Categorical
Binomial	Multinomial
Beta	Dirichlet

## Deterministic and constant nodes

In addition to the random variable nodes, there are two special types of nodes: constant and deterministic. Neither one has any probability distribution associated with them. A constant node has no parents and the value of the node is fixed. Constant nodes are created implicitly as parent nodes when you give numeric values as parents when creating a node, thus, the user is not required to create any constant nodes explicitly. A deterministic, on the other hand, defines some function. It transforms the parents to produce a new variable which is a non-random function of the parents. For instance, `Dot` node computes the dot product of its parents.

### 2.2.3 Providing the data

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

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

It is important that the shape of the `data` array matches the shape of the node `y`, which is the combination of the plates and the dimensionality. For instance, if `y` is `Wishart` node for  $3 \times 3$  matrices with plates  $(5, 1, 10)$ , the actual shape of `y` would be  $(5, 1, 10, 3, 3)$ . The data array must have this shape exactly, that is, no broadcasting rules are applied.

### Missing values

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

```
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. To be more precise, the mask is applied to the plates, *not* to the data array directly. Unlike for the data itself, standard NumPy broadcasting rules are applied for the mask with respect to the plates. So, if the variable has plates  $(5, 1, 10)$ , the mask could have a shape  $(1,)$ ,  $(10,)$ ,  $(5, 1, 1)$  or  $(5, 1, 10)$ .

From implementational point of view, the inference algorithms ignore the missing plates automatically if they are not needed. Thus, the missing values are integrated out giving more accurate approximations and the computations may also be faster.

### 2.2.4 Performing inference

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

- Choosing and constructing the inference engine
- Initializing the engine
- Running the inference algorithm

## 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 nodes of the model.

```
from bayespy.inference import VB
Q = VB(node1, node2, node3, node4)
```

## Initializing the inference

The inference engines give some initialization to the nodes by default. However, the inference algorithms can be sensitive to the initialization, thus it is sometimes necessary to have full control over the initialization. There may be different initialization methods, but for VB you can, for instance, initialize in one of the following ways:

- `initialize_from_prior`: Use only parent nodes to update the node.
- `initialize_from_parameters`: Use the given parameter values for the distribution.

A random initialization for VB has to be performed manually, because it is not obvious what is actually wanted. For instance, one way to achieve it is to first update from the parents, then to draw a random sample from that distribution and to set the values of the parameters based on that. For `Normal` node, one could draw the mean parameter randomly and choose the precision parameter arbitrarily:

```
x = bp.nodes.Normal(mu, tau, plates=(10,))
x.initialize_from_prior()
x.initialize_from_parameters(x.random(), 1)
```

In this case, the precision was set to one. The default initialization method is `initialization_from_prior`, which is performed when the node is created. If the initialization uses the values of the parents, they should be initialized before the children.

## 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. However, you can give as arguments the nodes you want to update and they are updated in the given order. It is possible to give same nodes several times, for instance:

```
Q.update(node1, node3, node1, node4)
```

This would update `node3` and `node4` once, and `node1` twice. In order to update several times, one can use the optional argument `repeat`.

```
Q.update(node3, node4, repeat=5)
Q.update(node1, node2, node3, node4, repeat=10)
```

This first updates `node3` and `node4` five times and then all the nodes ten times. This might be useful, for instance, if updating some nodes is expensive and should be done rarely or if updating some nodes in the beginning would cause the algorithm to converge to a bad solution.

**Warning:** Ideally, one constructs the model and then chooses the inference method to be used - possibly trying several different methods. However, the model construction is not yet separated from the model construction, that is, the constructed model network is also the variational message passing network for VB inference.

## 2.2.5 Examining the results

After the results have been obtained, it is important to be able to examine the results easily. `show` method prints the approximate posterior distribution of the node. Also, `get_moments` can be used to obtain the sufficient statistics of the node.

---

### Todo

In order to examine the results more carefully, `get_parameters` method should return the parameter values of the approximate posterior distribution. The user may use these values for arbitrarily complex further analysis.

---

## 2.3 Nodes

### 2.3.1 Stochastic nodes

Normal

Gaussian

Gamma

Wishart

Bernoulli

Categorical

Binomial

Multinomial

Beta

Dirichlet

Mixture

Constant

### 2.3.2 Deterministic nodes

Dot product (of Gaussian variables)

**EXAMPLES**

### 3.1 Example: Mixture of Gaussians

Do some stuff:

```
from bayespy.nodes import Dirichlet
alpha = Dirichlet([1e-3, 1e-3, 1e-3])
print(alpha._message_to_child())

[array([-666.66994695, -666.66994695, -666.66994695])]
```

Nice!

### 3.2 Example: Principal component analysis

Yeah.

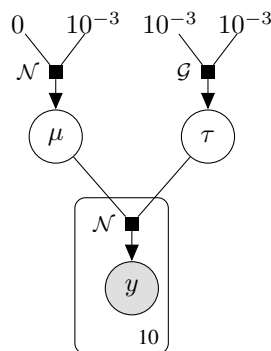


Figure 3.1: Directed factor graph of the example model.

```
from bayespy.nodes import GaussianARD
GaussianARD(0, 1)

<bayespy.inference.vmp.nodes.gaussian.GaussianARD at 0x7fa3343bce90>
```

### 3.3 Linear state-space model

This example is also available as an IPython notebook or a Python script.

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.

First, let us generate some toy data:

```
import numpy as np

M = 30
N = 400

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]])
c = np.random.randn(M, 4)
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]) + np.random.randn(4)
    f[:, n+1] = np.dot(c, x[n+1])
    y[:, n+1] = f[:, n+1] + 3*np.random.randn(M)
```

The linear state-space model can be constructed as follows:

```
from bayespy.inference.vmp.nodes.gaussian_markov_chain import GaussianMarkovChain
from bayespy.inference.vmp.nodes.gaussian import GaussianARD
from bayespy.inference.vmp.nodes.gamma import Gamma
from bayespy.inference.vmp.nodes.dot import SumMultiply

D = 10

# Dynamics matrix with ARD
alpha = Gamma(1e-5,
              1e-5,
              plates=(D,),
              name='alpha')
A = GaussianARD(0,
               alpha,
               shape=(D,),
               plates=(D,),
               name='A')

# Latent states with dynamics
X = GaussianMarkovChain(np.zeros(D),          # mean of x0
                       1e-3*np.identity(D),  # prec of x0
                       A,                      # dynamics
                       np.ones(D),            # innovation
```



```

        n=N,                                # time instances
        name='X',
        initialize=False)
X.initialize_from_value(np.zeros((N,D))) # just some empty values, X is
                                         # updated first anyway

# Mixing matrix from latent space to observation space using ARD
gamma = Gamma(1e-5,
              1e-5,
              plates=(D, ),
              name='gamma')
C = GaussianARD(0,
               gamma,
               shape=(D, ),
               plates=(M, 1),
               name='C')
# Initialize nodes (must use some randomness for C, and update X before C)
C.initialize_from_random()

# Observation noise
tau = Gamma(1e-5,
            1e-5,
            name='tau')

# Observations
F = SumMultiply('i,i',
               C,
               X,
               name='F')
Y = GaussianARD(F,
               tau,
               name='Y')

```

An inference machine using variational Bayesian inference with variational message passing is then constructed as

```

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

```

Observe the data partially (80% is marked missing):

```

from bayespy.utils import random

# Add missing values randomly (keep only 20%)
mask = random.mask(M, N, p=0.2)
Y.observe(y, mask=mask)

```

Then inference (100 iterations) can be run simply as

```

Q.update(repeat=10)

Iteration 1: loglike=-3.118644e+04 (0.210 seconds)
Iteration 2: loglike=-1.129540e+04 (0.210 seconds)
Iteration 3: loglike=-9.139376e+03 (0.210 seconds)
Iteration 4: loglike=-8.704676e+03 (0.220 seconds)
Iteration 5: loglike=-8.531889e+03 (0.200 seconds)
Iteration 6: loglike=-8.386198e+03 (0.210 seconds)
Iteration 7: loglike=-8.255826e+03 (0.210 seconds)
Iteration 8: loglike=-8.176274e+03 (0.210 seconds)
Iteration 9: loglike=-8.139579e+03 (0.210 seconds)

```

Iteration 10: loglike=-8.117779e+03 (0.210 seconds)

### 3.3.1 Speeding up with parameter expansion

VB inference can converge extremely slowly if the variables are strongly coupled. Because VMP updates one variable at a time, it may lead to slow zigzagging. This can be solved by using parameter expansion which reduces the coupling. In state-space models, the states  $\mathbf{x}_n$  and the loadings  $\mathbf{C}$  are coupled through a dot product  $\mathbf{C}\mathbf{x}_n$ , which is unaltered if the latent space is rotated arbitrarily:

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

Thus, one intuitive transformation would be  $\mathbf{C} \rightarrow \mathbf{C}\mathbf{R}^{-1}$  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 \*Fast Variational Bayesian Linear State-Space Model (Lutinen, 2013).

In BayesPy, the transformations can be used as follows:

```
# Import the parameter expansion module
from bayespy.inference.vmp import transformations

# Rotator of the state dynamics matrix
rotA = transformations.RotateGaussianARD(Q['A'], Q['alpha'])
# Rotator of the states (includes rotation of the state dynamics matrix)
rotX = transformations.RotateGaussianMarkovChain(Q['X'], rotA)
# Rotator of the loading matrix
rotC = transformations.RotateGaussianARD(Q['C'], Q['gamma'])
# Rotation optimizer
R = transformations.RotationOptimizer(rotX, rotC, D)
```

Note that it is crucial to select the correct rotation class which corresponds to the particular model block exactly. The rotation can be performed after each full VB update:

```
for ind in range(10):
    Q.update()
    R.rotate()
```

```
Iteration 11: loglike=-8.100983e+03 (0.210 seconds)
Iteration 12: loglike=-7.622913e+03 (0.210 seconds)
Iteration 13: loglike=-7.452057e+03 (0.200 seconds)
Iteration 14: loglike=-7.385975e+03 (0.200 seconds)
Iteration 15: loglike=-7.351449e+03 (0.210 seconds)
Iteration 16: loglike=-7.331026e+03 (0.210 seconds)
Iteration 17: loglike=-7.317997e+03 (0.200 seconds)
Iteration 18: loglike=-7.309212e+03 (0.200 seconds)
Iteration 19: loglike=-7.303074e+03 (0.210 seconds)
Iteration 20: loglike=-7.298661e+03 (0.210 seconds)
```

If you want to implement your own rotations or check the existing ones, you may use debugging utilities:

```
for ind in range(10):
    Q.update()
    R.rotate(check_bound=True,
             check_gradient=True)
```

```

Iteration 21: loglike=-7.295401e+03 (0.210 seconds)
Norm of numerical gradient: 3905.05
Norm of function gradient: 3905.05
Gradient relative error = 6.39002e-05 and absolute error = 0.249533
Iteration 22: loglike=-7.292861e+03 (0.210 seconds)
Norm of numerical gradient: 6245.37

/home/jluttine/workspace/bayespy/bayespy/inference/vmp/transformations.py:142: UserWarning: Rotation
warnings.warn("Rotation gradient has relative error %g" % err)
/home/jluttine/workspace/bayespy/bayespy/inference/vmp/transformations.py:142: UserWarning: Rotation
warnings.warn("Rotation gradient has relative error %g" % err)

Norm of function gradient: 6245.43
Gradient relative error = 7.56396e-05 and absolute error = 0.472397
Iteration 23: loglike=-7.290841e+03 (0.210 seconds)
Norm of numerical gradient: 3984.43
Norm of function gradient: 3984.43
Gradient relative error = 6.78117e-05 and absolute error = 0.270191
Iteration 24: loglike=-7.289243e+03 (0.210 seconds)
Norm of numerical gradient: 13053.7

/home/jluttine/workspace/bayespy/bayespy/inference/vmp/transformations.py:142: UserWarning: Rotation
warnings.warn("Rotation gradient has relative error %g" % err)
/home/jluttine/workspace/bayespy/bayespy/inference/vmp/transformations.py:142: UserWarning: Rotation
warnings.warn("Rotation gradient has relative error %g" % err)

Norm of function gradient: 13053.8
Gradient relative error = 2.65118e-05 and absolute error = 0.346078
Iteration 25: loglike=-7.287794e+03 (0.200 seconds)
Norm of numerical gradient: 4144.61
Norm of function gradient: 4144.59
Gradient relative error = 7.02612e-05 and absolute error = 0.291205
Iteration 26: loglike=-7.286531e+03 (0.210 seconds)
Norm of numerical gradient: 5821.72

/home/jluttine/workspace/bayespy/bayespy/inference/vmp/transformations.py:142: UserWarning: Rotation
warnings.warn("Rotation gradient has relative error %g" % err)
/home/jluttine/workspace/bayespy/bayespy/inference/vmp/transformations.py:142: UserWarning: Rotation
warnings.warn("Rotation gradient has relative error %g" % err)

Norm of function gradient: 5821.73
Gradient relative error = 4.57892e-05 and absolute error = 0.266572
Iteration 27: loglike=-7.285469e+03 (0.210 seconds)
Norm of numerical gradient: 15766.4
Norm of function gradient: 15766.4
Gradient relative error = 3.5184e-05 and absolute error = 0.554724
Iteration 28: loglike=-7.284584e+03 (0.200 seconds)
Norm of numerical gradient: 5782.51

/home/jluttine/workspace/bayespy/bayespy/inference/vmp/transformations.py:142: UserWarning: Rotation
warnings.warn("Rotation gradient has relative error %g" % err)
/home/jluttine/workspace/bayespy/bayespy/inference/vmp/transformations.py:142: UserWarning: Rotation
warnings.warn("Rotation gradient has relative error %g" % err)

Norm of function gradient: 5782.51
Gradient relative error = 5.61705e-05 and absolute error = 0.324807
Iteration 29: loglike=-7.283818e+03 (0.210 seconds)
Norm of numerical gradient: 9067.22

```

```
Norm of function gradient: 9067.21
Gradient relative error = 2.4973e-05 and absolute error = 0.226435
Iteration 30: loglike=-7.283121e+03 (0.200 seconds)
Norm of numerical gradient: 9594.54
```

```
/home/jluttine/workspace/bayespy/bayespy/inference/vmp/transformations.py:142: UserWarning: Rotation
warnings.warn("Rotation gradient has relative error %g" % err)
/home/jluttine/workspace/bayespy/bayespy/inference/vmp/transformations.py:142: UserWarning: Rotation
warnings.warn("Rotation gradient has relative error %g" % err)
```

```
Norm of function gradient: 9594.62
Gradient relative error = 5.43175e-05 and absolute error = 0.521151
```

## DEVELOPER GUIDE

How to document: [https://github.com/numpy/numpy/blob/master/doc/HOWTO\\_DOCUMENT.rst.txt](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](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

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

### 4.1.1 Univariate normal distribution

### 4.1.2 Multivariate normal distribution

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

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

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

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

### 4.1.3 Gamma distribution

### 4.1.4 Wishart distribution

$$\boldsymbol{\Lambda} \sim \mathcal{W}(n, \mathbf{V}),$$

$$n > D - 1, \quad \boldsymbol{\Lambda}, \mathbf{V} \in \mathbb{R}^{D \times D}, \quad \boldsymbol{\Lambda}, \mathbf{V} \text{ symmetric positive definite}$$

$$\log \mathcal{W}(\boldsymbol{\Lambda} | n, \mathbf{V}) = -\frac{1}{2} \text{tr}(\boldsymbol{\Lambda} \mathbf{V}) + \frac{n}{2} \log |\boldsymbol{\Lambda}| + \frac{n}{2} \log |\mathbf{V}| - \frac{D+1}{2} \log |\boldsymbol{\Lambda}| - \frac{nD}{2} \log 2 - \log \Gamma_D \left( \frac{n}{2} \right)$$

$$\begin{aligned}
 \mathbf{u}(\Lambda) &= \begin{bmatrix} \Lambda \\ \log |\Lambda| \end{bmatrix} \\
 \phi(n, \mathbf{V}) &= \begin{bmatrix} -\frac{1}{2} \mathbf{V} \\ \frac{1}{2} n \end{bmatrix} \\
 \phi_n(\Lambda, \mathbf{V}) &= \begin{bmatrix} \frac{1}{2} \log |\Lambda| + \frac{1}{2} \log |\mathbf{V}| + \frac{D}{2} \log 2 \\ -1 \end{bmatrix} \\
 \phi_{\mathbf{V}}(\Lambda, n) &= \begin{bmatrix} -\frac{1}{2} \Lambda \\ \frac{1}{2} n \end{bmatrix} \\
 g(n, \mathbf{V}) &= \frac{n}{2} \log |\mathbf{V}| - \frac{nD}{2} \log 2 - \log \Gamma_D \left( \frac{n}{2} \right) \\
 g\phi(\phi) &= \phi_2 \log |-\phi_1| - \log \Gamma_D(\phi_2) \\
 f(\Lambda) &= -\frac{D+1}{2} \log |\Lambda| \\
 \bar{\mathbf{u}}(\phi) &= \begin{bmatrix} -\phi_2 \phi_1^{-1} \\ -\log |-\phi_1| + \psi_D(\phi_2) \end{bmatrix}
 \end{aligned}$$

#### 4.1.5 Normal-Gamma distribution

#### 4.1.6 Gaussian-Wishart distribution

#### 4.1.7 Gaussian-Gamma distribution

#### 4.1.8 Mixture distribution

$$\mathbf{x} \sim \text{Mix}_{\mathcal{D}} \left( \lambda, \left\{ \Theta_1^{(n)}, \dots, \Theta_K^{(n)} \right\}_{n=1}^N \right)$$

$\lambda \in \{1, \dots, N\}$ ,  $\mathcal{D}$  is an exp.fam. distribution,  $\Theta_k^{(n)}$  are parameters of  $\mathcal{D}$

$$\begin{aligned}
 \log \text{Mix}_{\mathcal{D}} \left( \mathbf{x} \middle| \lambda, \left\{ \Theta_1^{(n)}, \dots, \Theta_K^{(n)} \right\}_{n=1}^N \right) &= \sum_{n=1}^N [\lambda = n] \mathbf{u}_{\mathcal{D}}(\mathbf{x})^T \phi_{\mathcal{D}} \left( \Theta_1^{(n)}, \dots, \Theta_K^{(n)} \right) \\
 &\quad + \sum_{n=1}^N [\lambda = n] g_{\mathcal{D}} \left( \Theta_1^{(n)}, \dots, \Theta_K^{(n)} \right) + f_{\mathcal{D}}(\mathbf{x})
 \end{aligned}$$

$$\begin{aligned}
 \mathbf{u}(\mathbf{x}) &= \mathbf{u}_{\mathcal{D}}(\mathbf{x}) \\
 \phi \left( \lambda, \left\{ \boldsymbol{\Theta}_1^{(n)}, \dots, \boldsymbol{\Theta}_K^{(n)} \right\}_{n=1}^N \right) &= \sum_{n=1}^N [\lambda = n] \phi_{\mathcal{D}} \left( \boldsymbol{\Theta}_1^{(n)}, \dots, \boldsymbol{\Theta}_K^{(n)} \right) \\
 \phi_{\lambda} \left( \mathbf{x}, \left\{ \boldsymbol{\Theta}_1^{(n)}, \dots, \boldsymbol{\Theta}_K^{(n)} \right\}_{n=1}^N \right) &= \begin{bmatrix} \mathbf{u}_{\mathcal{D}}(\mathbf{x})^T \phi_{\mathcal{D}} \left( \boldsymbol{\Theta}_1^{(1)}, \dots, \boldsymbol{\Theta}_K^{(1)} \right) + g_{\mathcal{D}} \left( \boldsymbol{\Theta}_1^{(1)}, \dots, \boldsymbol{\Theta}_K^{(1)} \right) \\ \vdots \\ \mathbf{u}_{\mathcal{D}}(\mathbf{x})^T \phi_{\mathcal{D}} \left( \boldsymbol{\Theta}_1^{(N)}, \dots, \boldsymbol{\Theta}_K^{(N)} \right) + g_{\mathcal{D}} \left( \boldsymbol{\Theta}_1^{(N)}, \dots, \boldsymbol{\Theta}_K^{(N)} \right) \end{bmatrix} \\
 \phi_{\boldsymbol{\Theta}_i^{(m)}} \left( \mathbf{x}, \lambda, \left\{ \boldsymbol{\Theta}_1^{(n)}, \dots, \boldsymbol{\Theta}_K^{(n)} \right\}_{n=1}^N \setminus \left\{ \boldsymbol{\Theta}_i^{(m)} \right\} \right) &= [\lambda = m] \phi_{\mathcal{D} \rightarrow \boldsymbol{\Theta}_i} \left( \mathbf{x}, \left\{ \boldsymbol{\Theta}_k^{(m)} \right\}_{k \neq i} \right) \\
 g \left( \lambda, \left\{ \boldsymbol{\Theta}_1^{(n)}, \dots, \boldsymbol{\Theta}_K^{(n)} \right\}_{n=1}^N \right) &= \sum_{n=1}^N [\lambda = n] g_{\mathcal{D}} \left( \boldsymbol{\Theta}_1^{(n)}, \dots, \boldsymbol{\Theta}_K^{(n)} \right) \\
 g(\phi) &= g_{\mathcal{D}}(\phi) \\
 f(\mathbf{x}) &= f_{\mathcal{D}}(\mathbf{x}) \\
 \bar{\mathbf{u}}(\phi) &= \bar{\mathbf{u}}_{\mathcal{D}}(\phi)
 \end{aligned}$$



## 5.1 Stochastic nodes

For testing, I can tell that `Gaussian` is used for modelling Gaussian variables.

Random variable nodes

---

Normal	
<code>Gaussian(*args, **kwargs)</code>	VMP node for Gaussian variable.
<code>Gamma(a, b, **kwargs)</code>	Node for gamma random variables.
<code>Wishart(*args, **kwargs)</code>	
<code>Dirichlet(*args, **kwargs)</code>	Node for Dirichlet random variables.
<code>GaussianMarkovChain(*args, **kwargs)</code>	VMP node for Gaussian Markov chain.

---

### 5.1.1 `bayespy.nodes.Gaussian`

`class bayespy.nodes.Gaussian(*args, **kwargs)`  
VMP node for Gaussian variable.

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).

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

Plates!

Parent nodes? Child nodes?

**See also:**

`Wishart`

#### Notes

Message passing equations:

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

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

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

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

## Methods

---

<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>	
<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>	
<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.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(*args)`

**bayespy.nodes.Gaussian.initialize\_from\_prior**

`Gaussian.initialize_from_prior()`

**bayespy.nodes.Gaussian.initialize\_from\_random**

`Gaussian.initialize_from_random()`

**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()`

### **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  $R1 \cdot X \cdot R2'$ , which is identical to  $\text{kron}(R1, R2) \cdot x$  (??)

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

## 5.1.2 bayespy.nodes.Gamma

**class** bayespy.nodes.**Gamma** (*a, b, \*\*kwargs*)  
Node for gamma random variables.

### Methods

---

<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>	
<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.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()`

**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
```

**5.1.3 bayespy.nodes.Wishart**

```
class bayespy.nodes.Wishart(*args, **kwargs)
```

**Methods**

<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>	
<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>save(group)</code>	Save the state of the node into a HDF5 file.
<code>set_plotter(plotter)</code>	

Continued on next page

Table 5.6 – continued from previous page

---

```
show()
unobserve()
update()
```

---

### **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()
```

### **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.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`

## **5.1.4 bayespy.nodes.Dirichlet**

**class** `bayespy.nodes.Dirichlet(*args, **kwargs)`

Node for Dirichlet random variables.

### **Methods**

---

<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>	

Continued on next page

Table 5.8 – 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>	
<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.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()`

### **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()`

**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()`

**bayespy.nodes.Dirichlet.unobserve**

`Dirichlet.unobserve()`

**bayespy.nodes.Dirichlet.update**

`Dirichlet.update()`

**Attributes**

---

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

---

**bayespy.nodes.Dirichlet.dims**

`Dirichlet.dims = None`

**bayespy.nodes.Dirichlet.plates**

`Dirichlet.plates = None`

### 5.1.5 bayespy.nodes.GaussianMarkovChain

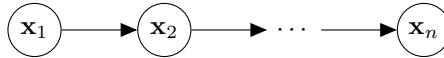
**class** bayespy.nodes.**GaussianMarkovChain** (\*args, \*\*kwargs)  
 VMP node for Gaussian Markov chain.

Parents are: *mu* is the mean of  $x_0$  (Gaussian) *Lambda* is the precision of  $x_0$  (Wishart) *A* is the dynamic matrix (Gaussian) *v* is the diagonal precision of the innovation (Gamma) An additional dummy parent is created: 'N' is the number of time instances

Output is Gaussian variables.

Time dimension is over the last plate.

Hmm.. The number of time instances is one more than the plates in A and V. Input N -> Output N+1.



**See also:**

bayespy.inference.vmp.nodes.gaussian.Gaussian, bayespy.inference.vmp.nodes.wishart.Wishart

#### Methods

---

<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>	
<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>	
<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.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()`

**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()`

### **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

<p><code>dims</code></p> <p><code>plates</code></p>
---

### bayespy.nodes.GaussianMarkovChain.dims

GaussianMarkovChain.**dims** = None

### bayespy.nodes.GaussianMarkovChain.plates

GaussianMarkovChain.**plates** = None

<p><code>Categorical(*args, **kwargs)</code> Node for categorical random variables.</p> <p><code>Mixture(*args, **kwargs)</code></p>
--

## 5.1.6 bayespy.nodes.Categorical

**class** bayespy.nodes.**Categorical**(\*args, \*\*kwargs)  
Node for categorical random variables.

### Methods

<p><code>add_plate_axis(to_plate)</code></p> <p><code>delete()</code></p> <p><code>get_mask()</code></p> <p><code>get_moments()</code></p> <p><code>get_shape(ind)</code></p> <p><code>has_plotter()</code></p> <p><code>initialize_from_parameters(*args)</code></p> <p><code>initialize_from_prior()</code></p> <p><code>initialize_from_random()</code></p> <p><code>initialize_from_value(x, *args)</code></p> <p><code>load(group)</code></p>	<p>Delete this node and the children</p> <p>Return True if the node has a plotter</p> <p>Load the state of the node from a HDF5 file.</p>
--	---

Continued on next page

Table 5.13 – 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.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()`

**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()`

### **Attributes**

---

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

---

### **bayespy.nodes.Categorical.dims**

`Categorical.dims = None`

### **bayespy.nodes.Categorical.plates**

`Categorical.plates = None`

## **5.1.7 bayespy.nodes.Mixture**

`class bayespy.nodes.Mixture(*args, **kwargs)`

## Methods

---

<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>	
<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>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.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()`

### **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  $\int p(x|\text{parents}) q(\text{parents}) d\text{parents}$  in log-scale as  $\int q(\text{parents}_i) \exp(\int q(\text{parents}_i) \log p(x|\text{parents}) d\text{parents}_i) d\text{parents}_i$ .

### **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.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**

---

`dims`  
`plates`

---

### bayespy.nodes.Mixture.dims

Mixture.dims = None

### bayespy.nodes.Mixture.plates

Mixture.plates = None

## 5.2 Deterministic nodes

---

<code>Dot(*args, **kwargs)</code>	Node for computing inner product of several Gaussian vectors.
-----------------------------------	---

---

### 5.2.1 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.

## 5.3 Base nodes

These nodes should be interesting only for developers.

---

<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>deterministic.Deterministic(*args, **kwargs)</code>	Base class for nodes that are deterministic.

---

### 5.3.1 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)`

#### Methods

Continued on next page
------------------------

Table 5.19 – continued from previous page

<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>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.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`

## 5.3.2 `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)`

#### Methods

---

<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>load(group)</code>	Load the state of the node from a HDF5 file.
<code>lowerbound()</code>	
<code>move_plates(from_plate, to_plate)</code>	
<code>observe(x[, mask])</code>	Fix moments, compute f and propagate mask.
<code>plot(**kwargs)</code>	Plot the node distribution using the plotter of the node
<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.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.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`

## **5.3.3 bayespy.inference.vmp.nodes.deterministic.Deterministic**

**class** `bayespy.inference.vmp.nodes.deterministic.Deterministic (*args, **kwargs)`

Base class for nodes that are deterministic.

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)
```

## Methods

---

<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.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`

---

`constant.Constant`(*moments, x, \*\*kwargs*)

---

## **5.3.4 bayespy.inference.vmp.nodes.constant.Constant**

**class** `bayespy.inference.vmp.nodes.constant.Constant` (*moments, x, \*\*kwargs*)

#### **Methods**

---

<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>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.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`

## INDICES AND TABLES

- *genindex*
- *modindex*
- *search*



## A

add\_plate\_axis() (bayespy.inference.vmp.nodes.constant.Constant method), 53  
 add\_plate\_axis() (bayespy.inference.vmp.nodes.deterministic.Deterministic method), 51  
 add\_plate\_axis() (bayespy.inference.vmp.nodes.node.Node method), 47  
 add\_plate\_axis() (bayespy.inference.vmp.nodes.stochastic.Stochastic method), 49  
 add\_plate\_axis() (bayespy.nodes.Categorical method), 40  
 add\_plate\_axis() (bayespy.nodes.Dirichlet method), 33  
 add\_plate\_axis() (bayespy.nodes.Gamma method), 26  
 add\_plate\_axis() (bayespy.nodes.Gaussian method), 23  
 add\_plate\_axis() (bayespy.nodes.GaussianMarkovChain method), 36  
 add\_plate\_axis() (bayespy.nodes.Mixture method), 43  
 add\_plate\_axis() (bayespy.nodes.Wishart method), 30  
 as\_diagonal\_wishart() (bayespy.nodes.Gamma method), 26

## C

Categorical (class in bayespy.nodes), 39  
 Constant (class in bayespy.inference.vmp.nodes.constant), 52

## D

delete() (bayespy.inference.vmp.nodes.constant.Constant method), 53  
 delete() (bayespy.inference.vmp.nodes.deterministic.Deterministic method), 51  
 delete() (bayespy.inference.vmp.nodes.node.Node method), 47  
 delete() (bayespy.inference.vmp.nodes.stochastic.Stochastic method), 49  
 delete() (bayespy.nodes.Categorical method), 40  
 delete() (bayespy.nodes.Dirichlet method), 33  
 delete() (bayespy.nodes.Gamma method), 26  
 delete() (bayespy.nodes.Gaussian method), 23  
 delete() (bayespy.nodes.GaussianMarkovChain method), 37  
 delete() (bayespy.nodes.Mixture method), 43  
 delete() (bayespy.nodes.Wishart method), 30

Deterministic (class in bayespy.inference.vmp.nodes.deterministic), 50  
 dims (bayespy.nodes.Categorical attribute), 42  
 dims (bayespy.nodes.Dirichlet attribute), 35  
 dims (bayespy.nodes.Gamma attribute), 29  
 dims (bayespy.nodes.Gaussian attribute), 25  
 dims (bayespy.nodes.GaussianMarkovChain attribute), 39  
 dims (bayespy.nodes.Mixture attribute), 46  
 dims (bayespy.nodes.Wishart attribute), 32  
 Dirichlet (class in bayespy.nodes), 32  
 Dot() (in module bayespy.nodes), 46

## G

Gamma (class in bayespy.nodes), 26  
 Gaussian (class in bayespy.nodes), 21  
 GaussianMarkovChain (class in bayespy.nodes), 36  
 get\_mask() (bayespy.inference.vmp.nodes.constant.Constant method), 53  
 get\_mask() (bayespy.inference.vmp.nodes.deterministic.Deterministic method), 51  
 get\_mask() (bayespy.inference.vmp.nodes.node.Node method), 47  
 get\_mask() (bayespy.inference.vmp.nodes.stochastic.Stochastic method), 49  
 get\_mask() (bayespy.nodes.Categorical method), 40  
 get\_mask() (bayespy.nodes.Dirichlet method), 33  
 get\_mask() (bayespy.nodes.Gamma method), 27  
 get\_mask() (bayespy.nodes.Gaussian method), 23  
 get\_mask() (bayespy.nodes.GaussianMarkovChain method), 37  
 get\_mask() (bayespy.nodes.Mixture method), 43  
 get\_mask() (bayespy.nodes.Wishart method), 30  
 get\_moments() (bayespy.inference.vmp.nodes.constant.Constant method), 53  
 get\_moments() (bayespy.inference.vmp.nodes.deterministic.Deterministic method), 51  
 get\_moments() (bayespy.inference.vmp.nodes.node.Node method), 47  
 get\_moments() (bayespy.inference.vmp.nodes.stochastic.Stochastic method), 49  
 get\_moments() (bayespy.nodes.Categorical method), 40

[get\\_moments\(\) \(bayespy.nodes.Dirichlet method\), 33](#)  
[get\\_moments\(\) \(bayespy.nodes.Gamma method\), 27](#)  
[get\\_moments\(\) \(bayespy.nodes.Gaussian method\), 23](#)  
[get\\_moments\(\) \(bayespy.nodes.GaussianMarkovChain method\), 37](#)  
[get\\_moments\(\) \(bayespy.nodes.Mixture method\), 43](#)  
[get\\_moments\(\) \(bayespy.nodes.Wishart method\), 30](#)  
[get\\_shape\(\) \(bayespy.inference.vmp.nodes.constant.Constant method\), 53](#)  
[get\\_shape\(\) \(bayespy.inference.vmp.nodes.deterministic.Deterministic method\), 51](#)  
[get\\_shape\(\) \(bayespy.inference.vmp.nodes.node.Node method\), 47](#)  
[get\\_shape\(\) \(bayespy.inference.vmp.nodes.stochastic.Stochastic method\), 49](#)  
[get\\_shape\(\) \(bayespy.nodes.Categorical method\), 40](#)  
[get\\_shape\(\) \(bayespy.nodes.Dirichlet method\), 33](#)  
[get\\_shape\(\) \(bayespy.nodes.Gamma method\), 27](#)  
[get\\_shape\(\) \(bayespy.nodes.Gaussian method\), 23](#)  
[get\\_shape\(\) \(bayespy.nodes.GaussianMarkovChain method\), 37](#)  
[get\\_shape\(\) \(bayespy.nodes.Mixture method\), 43](#)  
[get\\_shape\(\) \(bayespy.nodes.Wishart method\), 30](#)

## H

[has\\_plotter\(\) \(bayespy.inference.vmp.nodes.constant.Constant method\), 53](#)  
[has\\_plotter\(\) \(bayespy.inference.vmp.nodes.deterministic.Deterministic method\), 51](#)  
[has\\_plotter\(\) \(bayespy.inference.vmp.nodes.node.Node method\), 47](#)  
[has\\_plotter\(\) \(bayespy.inference.vmp.nodes.stochastic.Stochastic method\), 49](#)  
[has\\_plotter\(\) \(bayespy.nodes.Categorical method\), 40](#)  
[has\\_plotter\(\) \(bayespy.nodes.Dirichlet method\), 33](#)  
[has\\_plotter\(\) \(bayespy.nodes.Gamma method\), 27](#)  
[has\\_plotter\(\) \(bayespy.nodes.Gaussian method\), 23](#)  
[has\\_plotter\(\) \(bayespy.nodes.GaussianMarkovChain method\), 37](#)  
[has\\_plotter\(\) \(bayespy.nodes.Mixture method\), 44](#)  
[has\\_plotter\(\) \(bayespy.nodes.Wishart method\), 30](#)

## I

[initialize\\_from\\_parameters\(\) \(bayespy.nodes.Categorical method\), 40](#)  
[initialize\\_from\\_parameters\(\) \(bayespy.nodes.Dirichlet method\), 33](#)  
[initialize\\_from\\_parameters\(\) \(bayespy.nodes.Gamma method\), 27](#)  
[initialize\\_from\\_parameters\(\) \(bayespy.nodes.Gaussian method\), 23](#)  
[initialize\\_from\\_parameters\(\) \(bayespy.nodes.GaussianMarkovChain method\), 37](#)  
[initialize\\_from\\_parameters\(\) \(bayespy.nodes.Mixture method\), 44](#)  
[initialize\\_from\\_parameters\(\) \(bayespy.nodes.Wishart method\), 30](#)  
[initialize\\_from\\_prior\(\) \(bayespy.nodes.Categorical method\), 40](#)  
[initialize\\_from\\_prior\(\) \(bayespy.nodes.Dirichlet method\), 34](#)  
[initialize\\_from\\_prior\(\) \(bayespy.nodes.Gamma method\), 27](#)  
[initialize\\_from\\_prior\(\) \(bayespy.nodes.Gaussian method\), 23](#)  
[initialize\\_from\\_prior\(\) \(bayespy.nodes.GaussianMarkovChain method\), 37](#)  
[initialize\\_from\\_prior\(\) \(bayespy.nodes.Mixture method\), 44](#)  
[initialize\\_from\\_prior\(\) \(bayespy.nodes.Wishart method\), 30](#)  
[initialize\\_from\\_random\(\) \(bayespy.nodes.Categorical method\), 41](#)  
[initialize\\_from\\_random\(\) \(bayespy.nodes.Dirichlet method\), 34](#)  
[initialize\\_from\\_random\(\) \(bayespy.nodes.Gamma method\), 27](#)  
[initialize\\_from\\_random\(\) \(bayespy.nodes.Gaussian method\), 23](#)  
[initialize\\_from\\_random\(\) \(bayespy.nodes.GaussianMarkovChain method\), 37](#)  
[initialize\\_from\\_random\(\) \(bayespy.nodes.Mixture method\), 44](#)  
[initialize\\_from\\_random\(\) \(bayespy.nodes.Wishart method\), 30](#)  
[initialize\\_from\\_value\(\) \(bayespy.nodes.Categorical method\), 41](#)  
[initialize\\_from\\_value\(\) \(bayespy.nodes.Dirichlet method\), 34](#)  
[initialize\\_from\\_value\(\) \(bayespy.nodes.Gamma method\), 27](#)  
[initialize\\_from\\_value\(\) \(bayespy.nodes.Gaussian method\), 23](#)  
[initialize\\_from\\_value\(\) \(bayespy.nodes.GaussianMarkovChain method\), 37](#)  
[initialize\\_from\\_value\(\) \(bayespy.nodes.Mixture method\), 44](#)  
[initialize\\_from\\_value\(\) \(bayespy.nodes.Wishart method\), 30](#)  
[integrated\\_logpdf\\_from\\_parents\(\) \(bayespy.nodes.Mixture method\), 44](#)

## L

[load\(\) \(bayespy.inference.vmp.nodes.stochastic.Stochastic method\), 49](#)  
[load\(\) \(bayespy.nodes.Categorical method\), 41](#)

load() (bayespy.nodes.Dirichlet method), 34  
 load() (bayespy.nodes.Gamma method), 27  
 load() (bayespy.nodes.Gaussian method), 23  
 load() (bayespy.nodes.GaussianMarkovChain method), 37  
 load() (bayespy.nodes.Mixture method), 44  
 load() (bayespy.nodes.Wishart method), 31  
 logpdf() (bayespy.nodes.Categorical method), 41  
 logpdf() (bayespy.nodes.Dirichlet method), 34  
 logpdf() (bayespy.nodes.Gamma method), 27  
 logpdf() (bayespy.nodes.Gaussian method), 24  
 logpdf() (bayespy.nodes.GaussianMarkovChain method), 37  
 logpdf() (bayespy.nodes.Mixture method), 44  
 logpdf() (bayespy.nodes.Wishart method), 31  
 lower\_bound\_contribution()  
     (bayespy.inference.vmp.nodes.deterministic.Deterministic method), 52  
 lower\_bound\_contribution() (bayespy.nodes.Categorical method), 41  
 lower\_bound\_contribution() (bayespy.nodes.Dirichlet method), 34  
 lower\_bound\_contribution() (bayespy.nodes.Gamma method), 27  
 lower\_bound\_contribution() (bayespy.nodes.Gaussian method), 24  
 lower\_bound\_contribution()  
     (bayespy.nodes.GaussianMarkovChain method), 38  
 lower\_bound\_contribution() (bayespy.nodes.Mixture method), 44  
 lower\_bound\_contribution() (bayespy.nodes.Wishart method), 31  
 lowerbound() (bayespy.inference.vmp.nodes.stochastic.Stochastic method), 49  
 lowerbound() (bayespy.nodes.Categorical method), 41  
 lowerbound() (bayespy.nodes.Dirichlet method), 34  
 lowerbound() (bayespy.nodes.Gamma method), 28  
 lowerbound() (bayespy.nodes.Gaussian method), 24  
 lowerbound() (bayespy.nodes.GaussianMarkovChain method), 38  
 lowerbound() (bayespy.nodes.Mixture method), 44  
 lowerbound() (bayespy.nodes.Wishart method), 31

## M

Mixture (class in bayespy.nodes), 42  
 move\_plates() (bayespy.inference.vmp.nodes.constant.Constant method), 53  
 move\_plates() (bayespy.inference.vmp.nodes.deterministic.Deterministic method), 52  
 move\_plates() (bayespy.inference.vmp.nodes.node.Node method), 47  
 move\_plates() (bayespy.inference.vmp.nodes.stochastic.Stochastic method), 49

move\_plates() (bayespy.nodes.Categorical method), 41  
 move\_plates() (bayespy.nodes.Dirichlet method), 34  
 move\_plates() (bayespy.nodes.Gamma method), 28  
 move\_plates() (bayespy.nodes.Gaussian method), 24  
 move\_plates() (bayespy.nodes.GaussianMarkovChain method), 38  
 move\_plates() (bayespy.nodes.Mixture method), 45  
 move\_plates() (bayespy.nodes.Wishart method), 31

## N

Node (class in bayespy.inference.vmp.nodes.node), 46

## O

observe() (bayespy.inference.vmp.nodes.stochastic.Stochastic method), 49  
 observe() (bayespy.nodes.Categorical method), 41  
 observe() (bayespy.nodes.Dirichlet method), 34  
 observe() (bayespy.nodes.Gamma method), 28  
 observe() (bayespy.nodes.Gaussian method), 24  
 observe() (bayespy.nodes.GaussianMarkovChain method), 38  
 observe() (bayespy.nodes.Mixture method), 45  
 observe() (bayespy.nodes.Wishart method), 31

## P

pdf() (bayespy.nodes.Categorical method), 41  
 pdf() (bayespy.nodes.Dirichlet method), 34  
 pdf() (bayespy.nodes.Gamma method), 28  
 pdf() (bayespy.nodes.Gaussian method), 24  
 pdf() (bayespy.nodes.GaussianMarkovChain method), 38  
 pdf() (bayespy.nodes.Mixture method), 45  
 pdf() (bayespy.nodes.Wishart method), 31  
 plates (bayespy.inference.vmp.nodes.constant.Constant attribute), 54  
 plates (bayespy.inference.vmp.nodes.deterministic.Deterministic attribute), 52  
 plates (bayespy.inference.vmp.nodes.node.Node attribute), 48  
 plates (bayespy.inference.vmp.nodes.stochastic.Stochastic attribute), 50  
 plates (bayespy.nodes.Categorical attribute), 42  
 plates (bayespy.nodes.Dirichlet attribute), 35  
 plates (bayespy.nodes.Gamma attribute), 29  
 plates (bayespy.nodes.Gaussian attribute), 26  
 plates (bayespy.nodes.GaussianMarkovChain attribute), 39  
 plates (bayespy.nodes.Mixture attribute), 46  
 plates (bayespy.nodes.Wishart attribute), 32  
 plot() (bayespy.inference.vmp.nodes.constant.Constant method), 53  
 plot() (bayespy.inference.vmp.nodes.deterministic.Deterministic method), 52  
 plot() (bayespy.inference.vmp.nodes.node.Node method), 47

plot() (bayespy.inference.vmp.nodes.stochastic.Stochastic method), 50  
 plot() (bayespy.nodes.Categorical method), 41  
 plot() (bayespy.nodes.Dirichlet method), 35  
 plot() (bayespy.nodes.Gamma method), 28  
 plot() (bayespy.nodes.Gaussian method), 24  
 plot() (bayespy.nodes.GaussianMarkovChain method), 38  
 plot() (bayespy.nodes.Mixture method), 45  
 plot() (bayespy.nodes.Wishart method), 31

## R

random() (bayespy.nodes.Categorical method), 42  
 random() (bayespy.nodes.Dirichlet method), 35  
 random() (bayespy.nodes.Gamma method), 28  
 random() (bayespy.nodes.Gaussian method), 24  
 random() (bayespy.nodes.GaussianMarkovChain method), 38  
 rotate() (bayespy.nodes.Gaussian method), 24  
 rotate() (bayespy.nodes.GaussianMarkovChain method), 38  
 rotate\_matrix() (bayespy.nodes.Gaussian method), 25

## S

save() (bayespy.inference.vmp.nodes.stochastic.Stochastic method), 50  
 save() (bayespy.nodes.Categorical method), 42  
 save() (bayespy.nodes.Dirichlet method), 35  
 save() (bayespy.nodes.Gamma method), 28  
 save() (bayespy.nodes.Gaussian method), 25  
 save() (bayespy.nodes.GaussianMarkovChain method), 38  
 save() (bayespy.nodes.Mixture method), 45  
 save() (bayespy.nodes.Wishart method), 31  
 set\_plotter() (bayespy.inference.vmp.nodes.constant.Constant method), 53  
 set\_plotter() (bayespy.inference.vmp.nodes.deterministic.Deterministic method), 52  
 set\_plotter() (bayespy.inference.vmp.nodes.node.Node method), 48  
 set\_plotter() (bayespy.inference.vmp.nodes.stochastic.Stochastic method), 50  
 set\_plotter() (bayespy.nodes.Categorical method), 42  
 set\_plotter() (bayespy.nodes.Dirichlet method), 35  
 set\_plotter() (bayespy.nodes.Gamma method), 28  
 set\_plotter() (bayespy.nodes.Gaussian method), 25  
 set\_plotter() (bayespy.nodes.GaussianMarkovChain method), 38  
 set\_plotter() (bayespy.nodes.Mixture method), 45  
 set\_plotter() (bayespy.nodes.Wishart method), 32  
 show() (bayespy.nodes.Categorical method), 42  
 show() (bayespy.nodes.Dirichlet method), 35  
 show() (bayespy.nodes.Gamma method), 28  
 show() (bayespy.nodes.Gaussian method), 25

show() (bayespy.nodes.GaussianMarkovChain method), 39  
 show() (bayespy.nodes.Wishart method), 32  
 Stochastic (class in bayespy.inference.vmp.nodes.stochastic), 48

## U

unobserve() (bayespy.inference.vmp.nodes.stochastic.Stochastic method), 50  
 unobserve() (bayespy.nodes.Categorical method), 42  
 unobserve() (bayespy.nodes.Dirichlet method), 35  
 unobserve() (bayespy.nodes.Gamma method), 29  
 unobserve() (bayespy.nodes.Gaussian method), 25  
 unobserve() (bayespy.nodes.GaussianMarkovChain method), 39  
 unobserve() (bayespy.nodes.Mixture method), 45  
 unobserve() (bayespy.nodes.Wishart method), 32  
 update() (bayespy.inference.vmp.nodes.stochastic.Stochastic method), 50  
 update() (bayespy.nodes.Categorical method), 42  
 update() (bayespy.nodes.Dirichlet method), 35  
 update() (bayespy.nodes.Gamma method), 29  
 update() (bayespy.nodes.Gaussian method), 25  
 update() (bayespy.nodes.GaussianMarkovChain method), 39  
 update() (bayespy.nodes.Mixture method), 45  
 update() (bayespy.nodes.Wishart method), 32

## W

Wishart (class in bayespy.nodes), 29