BayesPy Documentation

Release 0.1

Jaakko Luttinen

June 09, 2014

ONE

INTRODUCTION

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

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

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 The version in PyPI is quite outdated.

1.1 Project information

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

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

- Documentation:

    - http://bayespy.org

    - PDF file

    - RST format in `doc` directory

- Repository: https://github.com/bayespy/bayespy.git

- Bug reports: https://github.com/bayespy/bayespy/issues

- Mailing list: bayespy@googlegroups.com

- IRC: #bayespy @ freenode

- Author: Jaakko Luttinen jaakko.luttinen@iki.fi

- Latest release:

- Build status:

- Unit test coverage:

1.2 Similar projects

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

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

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

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

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

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

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

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

USER GUIDE

2.1 Installation

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

2.1.1 Installing requirements

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

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

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

A proper installation of the dependencies for Python 3 can be a bit tricky and you may refer to http://www.scipy.org/install.html for more detailed instructions about the SciPy stack. If your system has an older version of any of the packages (NumPy, SciPy or matplotlib) or it does not provide the packages for Python 3, you may set up a virtual environment and install the latest versions there. To create and activate a new virtual environment, run

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

If you have relevant system libraries installed (C compiler, Python development files, BLAS/LAPACK etc.), you may be able to install the Python packages from PyPI. For instance, on Ubuntu (>= 12.10), you may install the required system libraries for each package as:

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

Then installation/upgrade from PyPI should work:

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

Note that Matplotlib requires a quite recent version of Distribute (>=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
```

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

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

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

```
make html
make latexpdf
```

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

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

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

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

```python
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 as if we didn't know their values. The model can be defined as follows:

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

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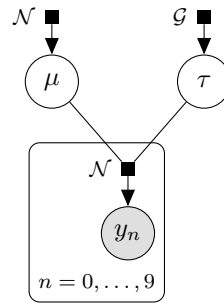

Figure 2.1: Directed factor graph of the example model.

constructed in BayesPy as follows:

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

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

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

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

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

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

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

```python
Q.update(repeat=20)
```

```
Iteration 1: loglike=-5.910731e+01 (0.010 seconds)
Iteration 2: loglike=-5.721245e+01 (0.000 seconds)
Iteration 3: loglike=-5.721009e+01 (0.000 seconds)
Iteration 4: loglike=-5.721007e+01 (0.010 seconds)
Converged.
```

Now the algorithm converged after four iterations, before the requested 20 iterations.

VB approximates the true posterior $p(\mu, \tau | \mathbf{y})$ with a distribution which factorizes with respect to the nodes: $q(\mu)q(\tau)$. The resulting approximate posterior distributions $q(\mu)$ and $q(\tau)$ can be examined, for instance, by plotting the marginal probability density functions:
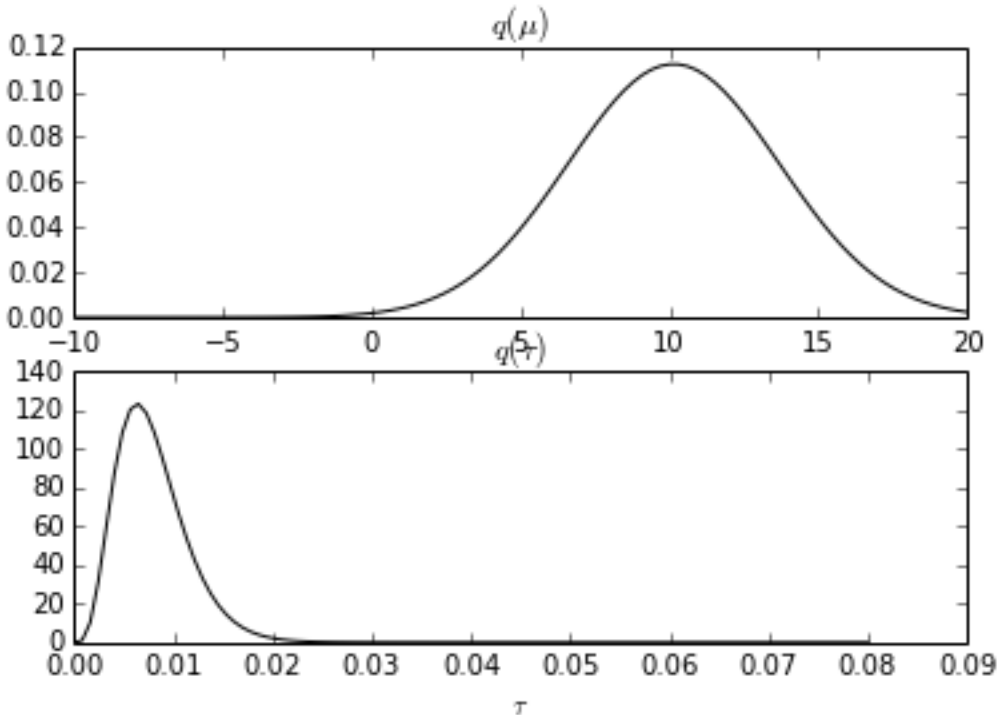
```python
import bayespy.plot as bpplt
# The following two two lines are just for enabling matplotlib plotting in notebooks
%matplotlib inline
```

```
bpplt.pyplot.plot([])
bpplt.pyplot.subplot(2, 1, 1)
bpplt.pdf(mu, np.linspace(-10, 20, num=100), color='k', name=r'\mu')
bpplt.pyplot.subplot(2, 1, 2)
bpplt.pdf(tau, np.linspace(1e-6, 0.08, num=100), color='k', name=r'\tau');
```


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

2.3 Constructing the model

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

2.3.1 Creating nodes

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

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

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

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

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

Conjugacy of the parents

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

Independence of the parents

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

```
from bayespy.nodes import Dot
Y = Dot(X, X)
```

```
ValueError: Parent nodes are not independent
```

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

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

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

2.3.2 Effects of the nodes on inference

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

1. the prior probability distribution for the variables,

2. the factorization of the posterior approximation,

3. the functional form of the posterior approximation for the variables.

Prior probability distribution

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

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

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

Posterior factorization

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

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

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

Functional form of the posterior

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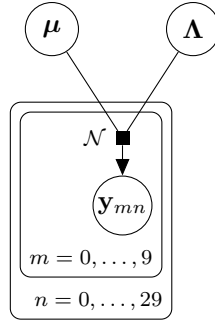
2.3.3 Using plate notation

Defining plates

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

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

This can also be visualized as a graphical model:

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

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

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

Sharing and broadcasting plates

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

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

which has the following graphical representation:
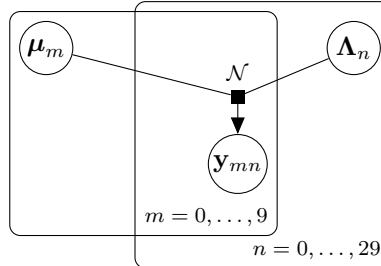
This can be constructed in BayesPy, for instance, as:

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

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

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

1. The two plates have equal size

2. The parent has size 1 (or no plate)

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

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

Plates in deterministic nodes

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

Plates and shape

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

```python
import numpy as np
mu = [ [0,0], [1,1], [2,2] ]
Lambda = [ [[1.0, 0.0],
            [0.0, 1.0]],
           [[1.0, 0.9],
            [0.9, 1.0]],
           [[1.0, -0.3],
            [-0.3, 1.0]] ]
X = Gaussian(mu, Lambda)
print("Shape of mu:", np.shape(mu))
print("Shape of Lambda:", np.shape(Lambda))
print("Plates of X:", X.plates)

Shape of mu: (3, 2)
Shape of Lambda: (3, 2, 2)
Plates of X: (3,)
```

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

Factorization of plates

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

Irregular plates

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

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

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

Plates of mu: (3,)
Plates of Lambda: (3,)
Plates of Z: (100,)
Plates of X: (100,)
```

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

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

```
from bayespy.nodes import GaussianARD, Gamma
tau = Gamma(1, 1, plates=(5,4,3))
X = GaussianARD(0, tau, shape=(4,3))
print("Plates of tau:", tau.plates)
print("Plates of X:", X.plates)

Plates of tau: (5, 4, 3)
Plates of X: (5,)
```

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

2.4 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

2.4.1 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 3×3 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.

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.

2.5 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.

EXAMPLES

3.1 Regression

3.1.1 Linear regression

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

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

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

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

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

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

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

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

Q.update(repeat=100)

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

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

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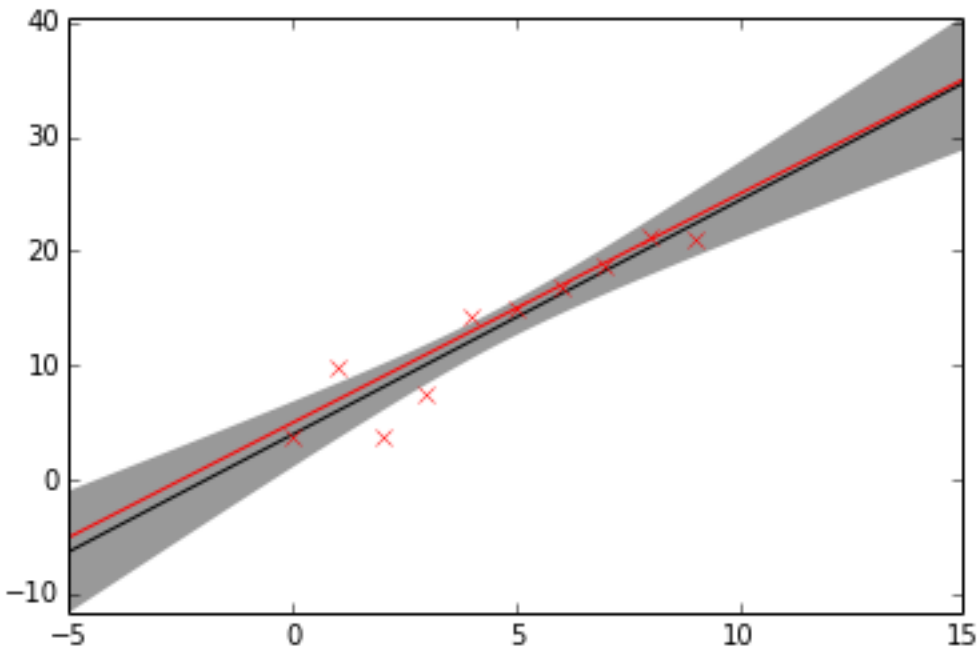


```python
bpplt.pdf(tau, np.linspace(1e-6,1,100), color='k')
bpplt.pyplot.axvline(s**(-2), color='r')
# Add labels
bpplt.pyplot.title(r'$q(\tau)$')
bpplt.pyplot.xlabel(r'$\tau$');
```
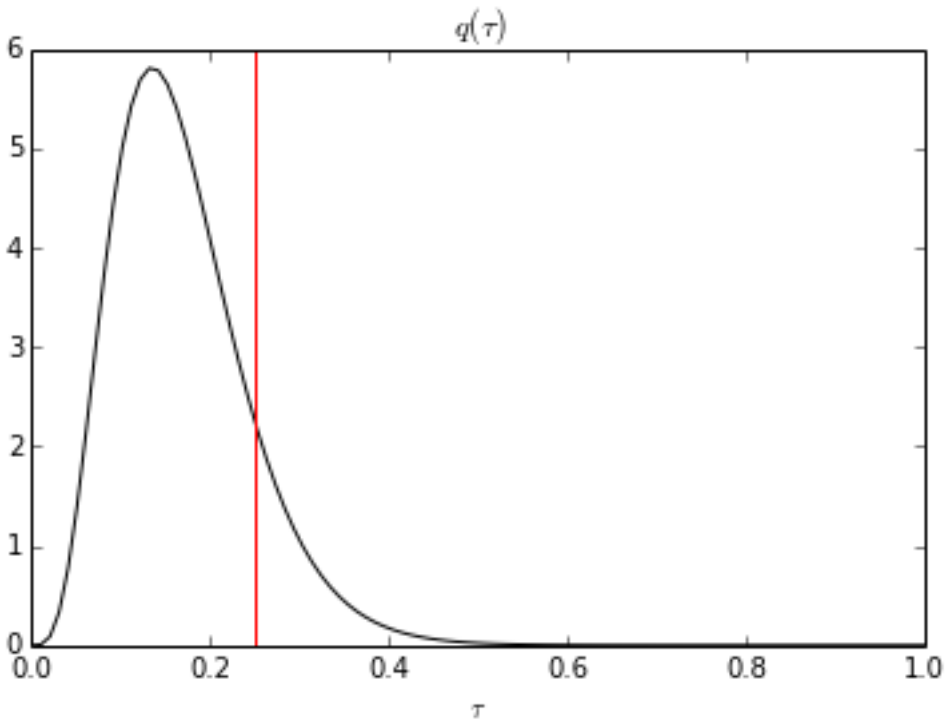
```
bpplt.contour(B, np.linspace(1,3,1000), np.linspace(1,9,1000), n=10, colors='k')
bpplt.plot(c, x=k, color='r', marker='x', linestyle='None', markersize=10, markeredgewidth=2)
# Add labels
bpplt.pyplot.title(r'$q(k,c)$')
bpplt.pyplot.xlabel(r'$k$')
bpplt.pyplot.ylabel(r'$c$');
```
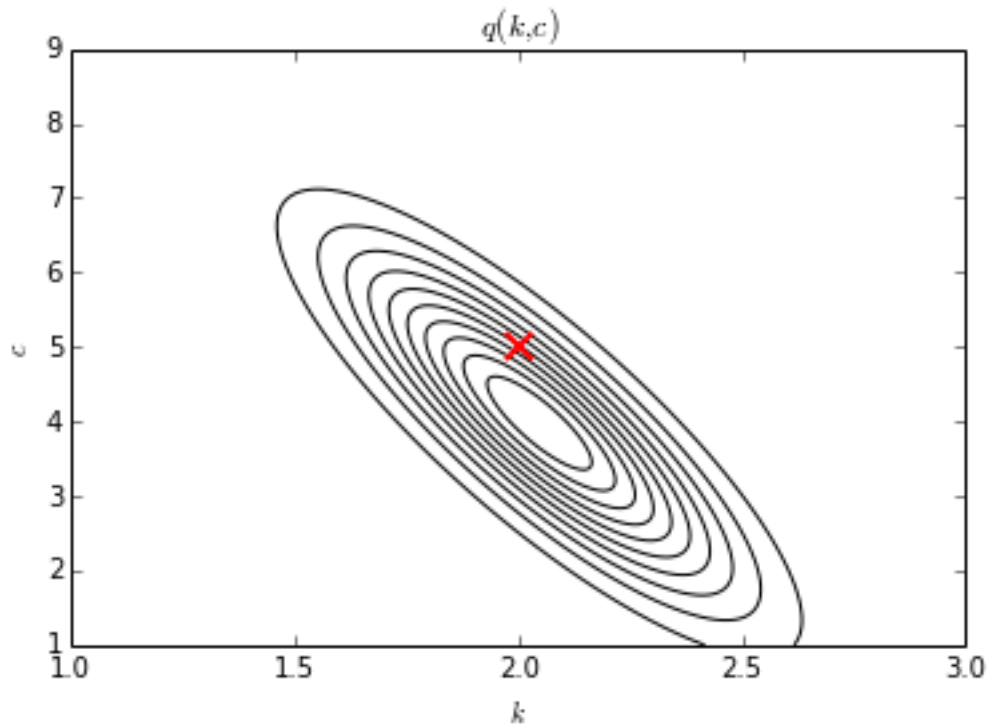
3.1.2 Improving accuracy

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


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


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


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


Q.update(repeat=10)


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


bpplt.pdf(B_tau.get_marginal_logpdf(gaussian=None, gamma=True),
          np.linspace(1e-6,1,100), color='k')
bpplt.pyplot.axvline(s**(-2), color='r')
# Add labels
bpplt.pyplot.title(r'$q(\tau)$')
bpplt.pyplot.xlabel(r'$\tau$');
```
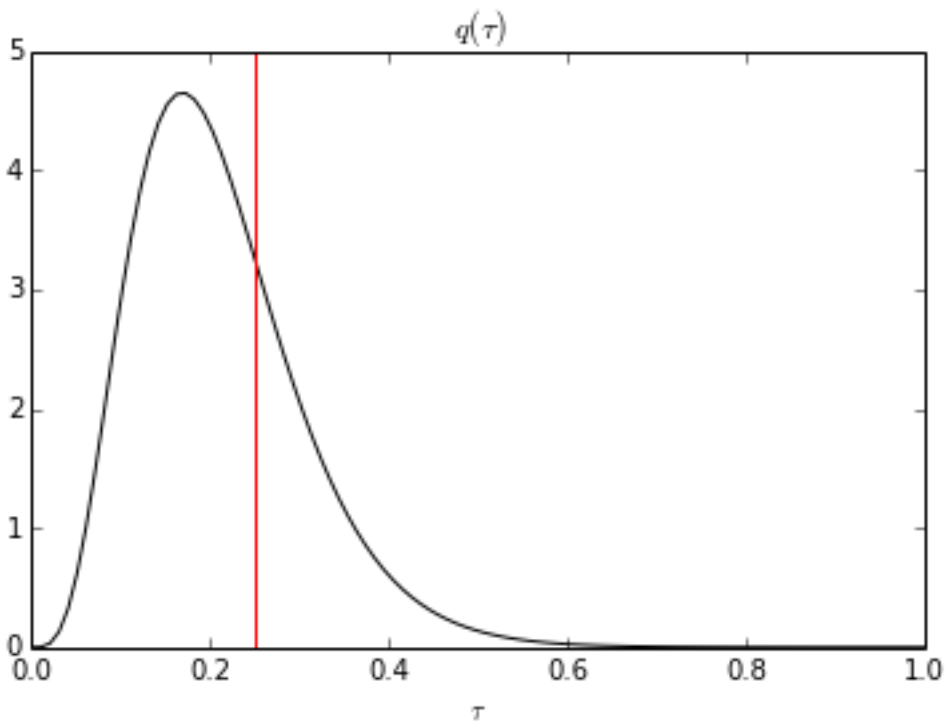
```
bpplt.contour(B_tau.get_marginal_logpdf(gaussian=[0,1], gamma=False),
              np.linspace(1,3,100), np.linspace(1,9,100),
              n=10, colors='k')
# Plot the true value
bpplt.plot(c, x=k, color='r', marker='x', linestyle='None', markersize=10, markeredgewidth=2)
# Add labels
bpplt.pyplot.title(r'$q(k,c)$')
bpplt.pyplot.xlabel(r'$k$')
bpplt.pyplot.ylabel(r'$c$');
```
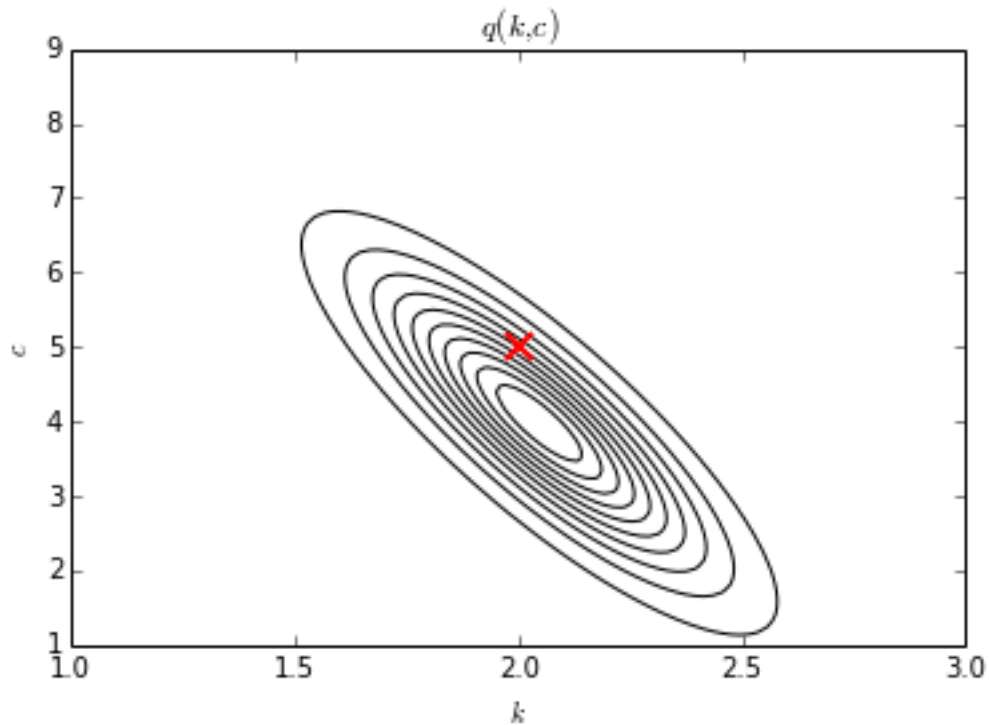
```
bpplt.contour(B_tau.get_marginal_logpdf(gaussian=[0], gamma=True),
              np.linspace(1,3,100), np.linspace(1e-6,1,100),
              n=10, colors='k')
bpplt.plot(s**(-2), x=k, color='r', marker='x', linestyle='None', markersize=10, markeredgewidth=2)
bpplt.pyplot.title(r'$q(k,\tau)$')
bpplt.pyplot.xlabel(r'$k$')
bpplt.pyplot.ylabel(r'$\tau$');
```
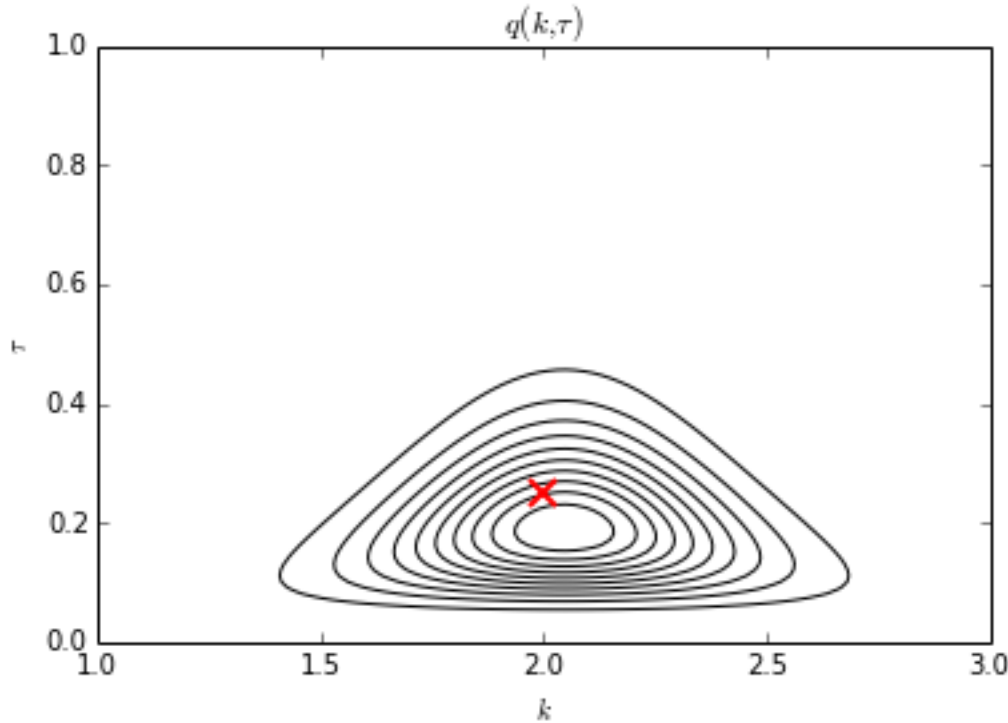
```
xh = np.linspace(-5, 15, 100)
Xh = np.vstack([xh, np.ones(len(xh))]).T
Fh_tau = SumMultiply('i,i', B_tau, Xh)
bpplt.plot(Fh_tau, x=xh, scale=2)
bpplt.plot(y, x=x, color='r', marker='x', linestyle='None')
bpplt.plot(k*xh+c, x=xh, color='r')
```

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

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


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


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

3.1.3 Multivariate regression

3.1.4 Non-linear regression

3.2 Gaussian mixture model

Do some stuff:

```python
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!

3.3 Bernoulli mixture model

blaa blaa blaa

```python
import numpy as np
D = 10
p0 = [0.1, 0.9, 0.1, 0.9, 0.1, 0.9, 0.1, 0.9, 0.1, 0.9]
p1 = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]
p2 = [1.0, 0.9, 0.8, 0.7, 0.6, 0.5, 0.4, 0.3, 0.2, 0.1]
p = np.array([p0, p1, p2])
from bayespy.utils import random
N = 100
z = random.categorical([1/3, 1/3, 1/3], size=N)
x = random.bernoulli(p[z])

from bayespy.nodes import Categorical, Dirichlet
K = 5
R = Dirichlet(K*[1e-3],
              name='R')
Z = Categorical(R,
                plates=(N,1),
                name='Z')

from bayespy.nodes import Mixture, Bernoulli, Beta
P = Beta([1e-1, 1e-1],
         plates=(D,K),
         name='P')
X = Mixture(Z, Bernoulli, P)

X.observe(x)

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

Q.update(repeat=10)

Iteration 1: loglike=nan (0.005 seconds)
Iteration 2: loglike=nan (0.003 seconds)
```

```
Iteration 3: loglike=nan (0.003 seconds)
Iteration 4: loglike=nan (0.003 seconds)
Iteration 5: loglike=nan (0.003 seconds)
Iteration 6: loglike=nan (0.003 seconds)
Iteration 7: loglike=nan (0.003 seconds)
Iteration 8: loglike=nan (0.003 seconds)
Iteration 9: loglike=nan (0.003 seconds)
Iteration 10: loglike=nan (0.003 seconds)
```

```
/home/jluttine/workspace/bayespy/bayespy/inference/vmp/nodes/dirichlet.py:91: RuntimeWarning: divide
  logp = np.log(p)
/home/jluttine/workspace/bayespy/bayespy/inference/vmp/nodes/expfamily.py:71: RuntimeWarning: invalid
  L = L + np.sum(phi_i * u_i, axis=axis_sum)
/home/jluttine/workspace/bayespy/bayespy/inference/vmp/nodes/expfamily.py:71: RuntimeWarning: invalid
  L = L + np.sum(phi_i * u_i, axis=axis_sum)
/home/jluttine/workspace/bayespy/bayespy/inference/vmp/nodes/mixture.py:229: UserWarning: The natural
  warnings.warn("The natural parameters of mixture distribution "
```

```python
import bayespy.plot.plotting as bpplt
bpplt.beta_hinton(P)
import matplotlib.pyplot as plt
plt.show()
```

```
/home/jluttine/workspace/bayespy/bayespy/plot/plotting.py:204: RuntimeWarning: invalid value encounte
  _w = np.abs(w)
```

3.4 Discrete hidden Markov model

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

3.4.1 Known parameters

This example follows the one presented in Wikipedia. Each day, the state of the weather is either 'rainy' or 'sunny'.
The weather follows a first-order discrete Markov process with the following initial state probability and state transition
probabilities:

```python
from bayespy.nodes import CategoricalMarkovChain
# Initial state probabilities
a0 = [0.6, 0.4] # p(rainy)=0.6, p(sunny)=0.4
# State transition probabilities
A = [[0.7, 0.3], # p(rainy->rainy)=0.7, p(rainy->sunny)=0.3
     [0.4, 0.6]] # p(sunny->rainy)=0.4, p(sunny->sunny)=0.6
# The length of the process
N = 1000
# Markov chain
Z = CategoricalMarkovChain(a0, A, states=N)
```

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

```python
from bayespy.nodes import Categorical, Mixture
# Emission probabilities
P = [[0.1, 0.4, 0.5],
     [0.6, 0.3, 0.1]]
```

```
# Observed process
Y = Mixture(Z, Categorical, P)
```

In order to test our method, we'll generate artificial data using this model:

```
# Draw realization of the weather process
weather = Z.random()
# Using this weather, draw realizations of the activities
activity = Mixture(weather, Categorical, P).random()
```

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

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

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

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

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

```
Q.update()
```

```
Iteration 1: loglike=-1.091583e+03 (0.090 seconds)
```

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

3.4.2 Unknown parameters

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

```
from bayespy.nodes import Dirichlet
# Initial state probabilities
a0 = Dirichlet([0.1, 0.1])
# State transition probabilities
A = Dirichlet([[0.1, 0.1],
               [0.1, 0.1]])
# Markov chain
Z = CategoricalMarkovChain(a0, A, states=N)
```

Second, the emission probabilities are also given quite non-informative priors:

```
# Emission probabilities
P = Dirichlet([[0.1, 0.1, 0.1],
               [0.1, 0.1, 0.1]])
# Observed process
Y = Mixture(Z, Categorical, P)
```

We use the same data as before:

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

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

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

If we ran the VB algorithm now, we would get a result where all both states would have identical emission probability distribution. This happens because of a non-random default initialization. `P` is initialized in such a way that both states have the same distribution, and `Z` is initialized in such a way that each state has equal probability. Thus, the VB algorithm won't separate them. In such cases, it is necessary to use a random initialization. In principle, it is possible to use random initialization for either variable and then update the other variable first. In the case of mixture distributions, it might be better to initialize the parameters (`P`) randomly and update the state assignments (`Z`) first.

```
P.initialize_from_random()
Q.update(Z, A, a0, P, repeat=20)
```

```
Iteration 1: loglike=-1.115941e+03 (0.090 seconds)
Iteration 2: loglike=-1.115671e+03 (0.090 seconds)
Iteration 3: loglike=-1.115603e+03 (0.100 seconds)
Iteration 4: loglike=-1.115574e+03 (0.090 seconds)
Iteration 5: loglike=-1.115555e+03 (0.090 seconds)
Iteration 6: loglike=-1.115538e+03 (0.100 seconds)
Iteration 7: loglike=-1.115521e+03 (0.090 seconds)
Iteration 8: loglike=-1.115504e+03 (0.090 seconds)
Iteration 9: loglike=-1.115487e+03 (0.090 seconds)
Iteration 10: loglike=-1.115469e+03 (0.090 seconds)
Iteration 11: loglike=-1.115451e+03 (0.100 seconds)
Iteration 12: loglike=-1.115433e+03 (0.090 seconds)
Iteration 13: loglike=-1.115413e+03 (0.090 seconds)
Iteration 14: loglike=-1.115394e+03 (0.090 seconds)
Iteration 15: loglike=-1.115374e+03 (0.090 seconds)
Iteration 16: loglike=-1.115354e+03 (0.100 seconds)
Iteration 17: loglike=-1.115333e+03 (0.090 seconds)
Iteration 18: loglike=-1.115312e+03 (0.090 seconds)
Iteration 19: loglike=-1.115290e+03 (0.090 seconds)
Iteration 20: loglike=-1.115268e+03 (0.090 seconds)
```

In order to update the variables in that order, one may explicitly give the nodes in that order to the `update` method. However, the default update order is the one used when constructing `Q`, which is the same in this case, thus we could have ignored listing the nodes to the `update` method.

Plot the estimated state transition probabilities:

```
# NOTE: These three lines are just to enable inline plotting in IPython Notebooks.
import matplotlib.pyplot as plt
%matplotlib inline
plt.plot([])
# Plot the state transition matrix
import bayespy.plot.plotting as bpplt
bpplt.dirichlet_hinton(A)
```
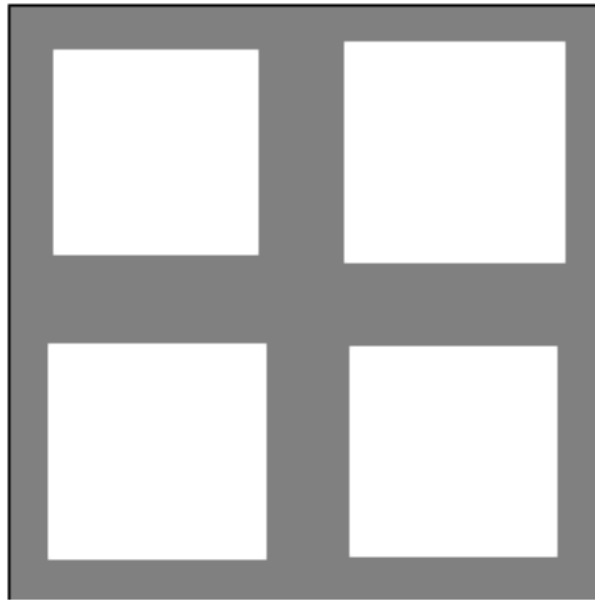
Plot the estimated emission probabilities:

```
bpplt.dirichlet_hinton(P)
```
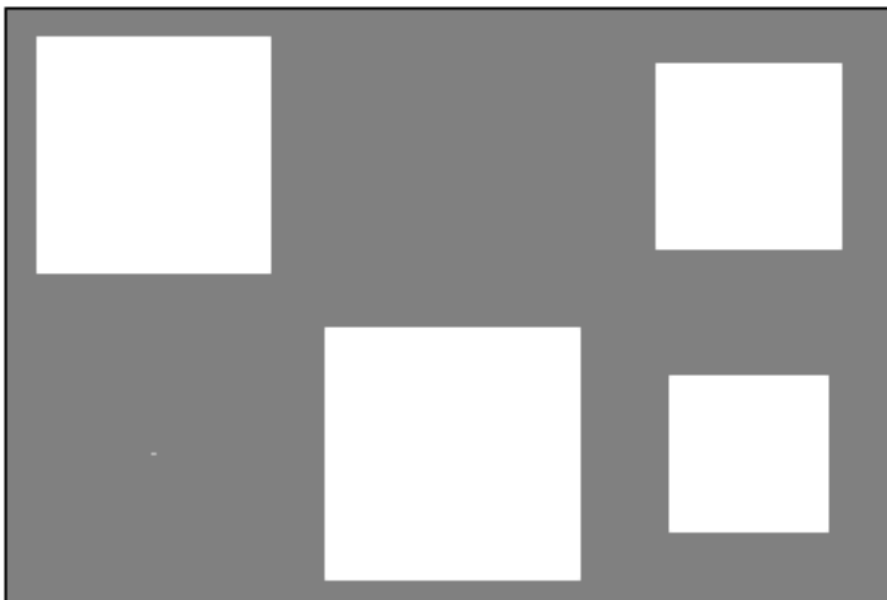

It is interesting that these estimated parameters are very different from the true parameters. This happens because of un-identifiability: different parameters lead to similar marginal distributions over the observed process.

3.5 Hidden Markov model

blaa blaa

3.6 Principal component analysis

Yeah.
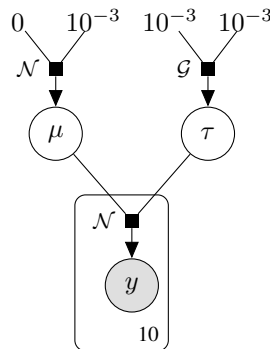
Figure 3.1: Directed factor graph of the example model.

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

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

3.7 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:

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

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

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 construced 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)
```

3.7.1 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 $\mathbf{C} \to \mathbf{C}\mathbf{R}^{-1}$ and $\mathbf{X} \to \mathbf{R}\mathbf{X}$. 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 $\mathbf{A} \to \mathbf{R}\mathbf{A}\mathbf{R}^{-1}$. For more details, refer to *Fast Variational Bayesian Linear State-Space Model (Luttinen, 2013).

In BayesPy, the transformations can be used as follows:

```python
# 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:

```python
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:

```python
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
```

3.8 Latent Dirichlet allocation

blaa blaa blaa..

DEVELOPER GUIDE

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

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

4.1 Variational message passing

The general update equation for factorized approximation:

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

where $\mathrm{pa}(\boldsymbol{\theta})$ and $\mathrm{ch}(\boldsymbol{\theta})$ are the set of parents and children of $\boldsymbol{\theta}$, respectively. 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

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

where $\boldsymbol{\Theta} = \{\boldsymbol{\theta}_j\}$ is the set of parents. 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

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

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

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

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

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

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

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

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

$$\tilde{\boldsymbol{\phi}}_{\boldsymbol{\theta}} = \boldsymbol{\phi}_{\boldsymbol{\theta}} \left(\{\mathbf{m}_{\mathbf{z} \to \boldsymbol{\theta}}\}_{\mathbf{z} \in \mathrm{pa}(\boldsymbol{\theta})} \right) + \sum_{\mathbf{x} \in \mathrm{ch}(\boldsymbol{\theta})} \mathbf{m}_{\mathbf{x} \to \boldsymbol{\theta}}.$$

4.2 Implementing nodes

USER API

5.1 Nodes

5.1.1 Stochastic nodes

Nodes for Gaussian variables:

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

bayespy.nodes.Gaussian

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

Node for Gaussian variables.

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

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

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

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

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

Mean vector

Lambda : Wishart-like node or array

Precision matrix

See also:

Wishart, GaussianARD, GaussianWishart, GaussianGammaARD, GaussianGammaISO

Methods

add_plate_axis(to_plate)	
delete()	Delete this node and the children
get_mask()	

Table 5.2 – continued from previous page

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

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

bayespy.nodes.Gaussian.rotate

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

bayespy.nodes.Gaussian.rotate_matrix

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

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

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

> **Parameters** **R1** : ndarray
>
> > A matrix from the left
>
> **R2** : ndarray
>
> > A matrix from the right

bayespy.nodes.Gaussian.save

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

   group can be the root

bayespy.nodes.Gaussian.set_plotter

Gaussian.**set_plotter**(*plotter*)

bayespy.nodes.Gaussian.show

Gaussian.**show**()

bayespy.nodes.Gaussian.unobserve

Gaussian.**unobserve**()

bayespy.nodes.Gaussian.update

Gaussian.**update**()

Attributes

dims	
plates	

bayespy.nodes.Gaussian.dims

Gaussian.**dims** = None

bayespy.nodes.Gaussian.plates

Gaussian.**plates** = None

bayespy.nodes.GaussianARD

class bayespy.nodes.**GaussianARD**(*mu*, *alpha*, *ndim=None*, *shape=None*, ***kwargs*)
Node for Gaussian variables with ARD prior.

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

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

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

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

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

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

> **alpha** : gamma-like node or array
>
>> Diagonal elements of the precision matrix

See also:

Gamma, Gaussian, GaussianGammaARD, GaussianGammaISO, GaussianWishart

Methods

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

bayespy.nodes.GaussianARD.add_plate_axis

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

bayespy.nodes.GaussianARD.delete

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

bayespy.nodes.GaussianARD.get_mask

GaussianARD.**get_mask**()

bayespy.nodes.GaussianARD.get_moments

GaussianARD.**get_moments**()

bayespy.nodes.GaussianARD.get_shape

GaussianARD.**get_shape**(*ind*)

bayespy.nodes.GaussianARD.has_plotter

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

bayespy.nodes.GaussianARD.initialize_from_mean_and_covariance

GaussianARD.**initialize_from_mean_and_covariance**(*mu*, *Cov*)

bayespy.nodes.GaussianARD.initialize_from_parameters

GaussianARD.**initialize_from_parameters**(**args*)

bayespy.nodes.GaussianARD.initialize_from_prior

GaussianARD.**initialize_from_prior**()

bayespy.nodes.GaussianARD.initialize_from_random

GaussianARD.**initialize_from_random**()

bayespy.nodes.GaussianARD.initialize_from_value

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

bayespy.nodes.GaussianARD.load

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

bayespy.nodes.GaussianARD.logpdf

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

bayespy.nodes.GaussianARD.lower_bound_contribution

GaussianARD.**lower_bound_contribution**(*gradient=False*)

bayespy.nodes.GaussianARD.lowerbound

GaussianARD.**lowerbound**()

bayespy.nodes.GaussianARD.move_plates

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

bayespy.nodes.GaussianARD.observe

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

bayespy.nodes.GaussianARD.pdf

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

bayespy.nodes.GaussianARD.plot

GaussianARD.**plot**(***kwargs*)
    Plot the node distribution using the plotter of the node

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

bayespy.nodes.GaussianARD.random

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

bayespy.nodes.GaussianARD.rotate

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

bayespy.nodes.GaussianARD.rotate_plates

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

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

---

bayespy.nodes.GaussianARD.save

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

> group can be the root

bayespy.nodes.GaussianARD.set_plotter

GaussianARD.**set_plotter**(*plotter*)

bayespy.nodes.GaussianARD.show

GaussianARD.**show**()

bayespy.nodes.GaussianARD.unobserve

GaussianARD.**unobserve**()

bayespy.nodes.GaussianARD.update

GaussianARD.**update**()

Attributes

dims	
plates	

bayespy.nodes.GaussianARD.dims

GaussianARD.**dims** = None

bayespy.nodes.GaussianARD.plates

GaussianARD.**plates** = None

Nodes for precision and scale variables:

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

bayespy.nodes.Gamma

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

Parameters **a** : scalar or array

Shape parameter

b : gamma-like node or scalar or array

Rate parameter

Methods

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

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

dims	tuple() -> empty tuple
plates	

bayespy.nodes.Gamma.dims

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

bayespy.nodes.Gamma.plates

$Gamma.$**plates** $=$ **None**

bayespy.nodes.Wishart

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

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

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

> **Parameters** **n** : scalar or array

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

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

>> $\mathbf{V}$, scale matrix.

Methods

add_plate_axis(to_plate)	
delete()	Delete this node and the children

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

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

bayespy.nodes.Wishart.save

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

     group can be the root

bayespy.nodes.Wishart.set_plotter

Wishart.**set_plotter**(*plotter*)

bayespy.nodes.Wishart.show

Wishart.**show**()

bayespy.nodes.Wishart.unobserve

Wishart.**unobserve**()

bayespy.nodes.Wishart.update

Wishart.**update**()

Attributes

dims	
plates	

bayespy.nodes.Wishart.dims

Wishart.**dims** = None

bayespy.nodes.Wishart.plates

Wishart.**plates** = None

bayespy.nodes.Exponential

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

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

> **Parameters** **l** : gamma-like node or scalar or array
>
>> Rate parameter

See also:

Gamma, Poisson

Notes

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

Methods

add_plate_axis(to_plate)	
as_diagonal_wishart()	
delete()	Delete this node and the children
get_mask()	
get_moments()	
get_shape(ind)	
has_plotter()	Return True if the node has a plotter
initialize_from_parameters(*args)	
initialize_from_prior()	
initialize_from_random()	
initialize_from_value(x, *args)	
load(group)	Load the state of the node from a HDF5 file.
logpdf(X[, mask])	Compute the log probability density function Q(X) of this node.

Continued on next page

Table 5.11 – continued from previous page

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

bayespy.nodes.Exponential.add_plate_axis

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

bayespy.nodes.Exponential.as_diagonal_wishart

Exponential.**as_diagonal_wishart**()

bayespy.nodes.Exponential.delete

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

bayespy.nodes.Exponential.get_mask

Exponential.**get_mask**()

bayespy.nodes.Exponential.get_moments

Exponential.**get_moments**()

bayespy.nodes.Exponential.get_shape

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

bayespy.nodes.Exponential.has_plotter

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

bayespy.nodes.Exponential.initialize_from_parameters

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

bayespy.nodes.Exponential.initialize_from_prior

Exponential.**initialize_from_prior**()

bayespy.nodes.Exponential.initialize_from_random

Exponential.**initialize_from_random**()

bayespy.nodes.Exponential.initialize_from_value

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

bayespy.nodes.Exponential.load

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

bayespy.nodes.Exponential.logpdf

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

bayespy.nodes.Exponential.lower_bound_contribution

Exponential.**lower_bound_contribution**(*gradient=False*)

bayespy.nodes.Exponential.lowerbound

Exponential.**lowerbound**()

bayespy.nodes.Exponential.move_plates

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

bayespy.nodes.Exponential.observe

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

bayespy.nodes.Exponential.pdf

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

bayespy.nodes.Exponential.plot

Exponential.**plot**(***kwargs*)

> Plot the node distribution using the plotter of the node

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

bayespy.nodes.Exponential.random

Exponential.**random**()

> Draw a random sample from the distribution.

bayespy.nodes.Exponential.save

Exponential.**save**(*group*)

> Save the state of the node into a HDF5 file.

> group can be the root

bayespy.nodes.Exponential.set_plotter

Exponential.**set_plotter**(*plotter*)

bayespy.nodes.Exponential.show

Exponential.**show**()

> Print the distribution using standard parameterization.

bayespy.nodes.Exponential.unobserve

Exponential.**unobserve**()

bayespy.nodes.Exponential.update

Exponential.**update**()

Attributes

dims	tuple() -> empty tuple
plates	

bayespy.nodes.Exponential.dims

Exponential.**dims** = ((), ())

bayespy.nodes.Exponential.plates

Exponential.**plates** = None

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

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

bayespy.nodes.GaussianGammaISO

class bayespy.nodes.**GaussianGammaISO**(*args*, ***kwargs*)

Node for Gaussian-gamma (isotropic) random variables.

The prior:

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

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

Currently, supports only vector variables.

Methods

`add_plate_axis`(to_plate)	
`delete`()	Delete this node and the children
`get_gaussian_mean_and_variance`()	Return the mean and variance of the distribution
`get_marginal_logpdf`([gaussian, gamma])	Get the (marginal) log pdf of a subset of the variables
`get_mask`()	
`get_moments`()	
`get_shape`(ind)	
`has_plotter`()	Return True if the node has a plotter
`initialize_from_parameters`(*args)	
`initialize_from_prior`()	
`initialize_from_random`()	
`initialize_from_value`(x, *args)	
`load`(group)	Load the state of the node from a HDF5 file.
`logpdf`(X[, mask])	Compute the log probability density function Q(X) of this node.
`lower_bound_contribution`([gradient])	
`lowerbound`()	
`move_plates`(from_plate, to_plate)	
`observe`(x, *args[, mask])	Fix moments, compute f and propagate mask.
`pdf`(X[, mask])	Compute the probability density function of this node.
`plot`(**kwargs)	Plot the node distribution using the plotter of the node
`plotmatrix`()	Creates a matrix of marginal plots.
`random`()	Draw a random sample from the distribution.
`save`(group)	Save the state of the node into a HDF5 file.
`set_plotter`(plotter)	

Continued on next page

5.1. Nodes

Table 5.14 – continued from previous page	
show()	Print the distribution using standard parameterization.
unobserve()	
update()	

bayespy.nodes.GaussianGammaISO.add_plate_axis

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

bayespy.nodes.GaussianGammaISO.delete

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

bayespy.nodes.GaussianGammaISO.get_gaussian_mean_and_variance

GaussianGammaISO.**get_gaussian_mean_and_variance**()
 Return the mean and variance of the distribution

bayespy.nodes.GaussianGammaISO.get_marginal_logpdf

GaussianGammaISO.**get_marginal_logpdf**(*gaussian=None*, *gamma=None*)
 Get the (marginal) log pdf of a subset of the variables

> **Parameters gaussian** : list or None
>
>> Indices of the Gaussian variables to keep or None
>
> **gamma** : bool or None
>
>> True if keep the gamma variable, otherwise False or None
>
> **Returns** function
>
>> A function which computes log-pdf

bayespy.nodes.GaussianGammaISO.get_mask

GaussianGammaISO.**get_mask**()

bayespy.nodes.GaussianGammaISO.get_moments

GaussianGammaISO.**get_moments**()

bayespy.nodes.GaussianGammaISO.get_shape

GaussianGammaISO.**get_shape**(*ind*)

bayespy.nodes.GaussianGammaISO.has_plotter

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

bayespy.nodes.GaussianGammaISO.initialize_from_parameters

GaussianGammaISO.**initialize_from_parameters**(**args*)

bayespy.nodes.GaussianGammaISO.initialize_from_prior

GaussianGammaISO.**initialize_from_prior**()

bayespy.nodes.GaussianGammaISO.initialize_from_random

GaussianGammaISO.**initialize_from_random**()

bayespy.nodes.GaussianGammaISO.initialize_from_value

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

bayespy.nodes.GaussianGammaISO.load

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

bayespy.nodes.GaussianGammaISO.logpdf

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

bayespy.nodes.GaussianGammaISO.lower_bound_contribution

GaussianGammaISO.**lower_bound_contribution**(*gradient=False*)

bayespy.nodes.GaussianGammaISO.lowerbound

GaussianGammaISO.**lowerbound**()

bayespy.nodes.GaussianGammaISO.move_plates

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

bayespy.nodes.GaussianGammaISO.observe

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

bayespy.nodes.GaussianGammaISO.pdf

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

bayespy.nodes.GaussianGammaISO.plot

GaussianGammaISO.**plot**(***kwargs*)
    Plot the node distribution using the plotter of the node

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

bayespy.nodes.GaussianGammaISO.plotmatrix

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

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

bayespy.nodes.GaussianGammaISO.random

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

bayespy.nodes.GaussianGammaISO.save

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

    group can be the root

bayespy.nodes.GaussianGammaISO.set_plotter

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

bayespy.nodes.GaussianGammaISO.show

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

bayespy.nodes.GaussianGammaISO.unobserve

`GaussianGammaISO.`**`unobserve`**`()`

bayespy.nodes.GaussianGammaISO.update

`GaussianGammaISO.`**`update`**`()`

Attributes

dims	
plates	

bayespy.nodes.GaussianGammaISO.dims

`GaussianGammaISO.`**`dims`** = **None**

bayespy.nodes.GaussianGammaISO.plates

`GaussianGammaISO.`**`plates`** = **None**

bayespy.nodes.GaussianGammaARD

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

   The prior:

$$p(x, \tau | \mu, \alpha, a, b) = p(x | \tau, \mu, \alpha) p(\tau | a, b)$$
$$p(x | \alpha, \mu, \alpha) = \mathcal{N}(x | \mu, \mathrm{diag}(\boldsymbol{\alpha \tau}))$$
$$p(\tau | a, b) = \mathcal{G}(\tau | a, b)$$

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

> **Warning:** Not yet implemented.

   See also:

   `Gaussian`, `GaussianARD`, `Gamma`, `GaussianGammaISO`, `GaussianWishart`

   Methods

add_plate_axis(to_plate)	
delete()	Delete this node and the children
get_mask()	
get_moments()	

Table 5.16 – continued from previous page

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

bayespy.nodes.GaussianGammaARD.add_plate_axis

GaussianGammaARD.**add_plate_axis**(*to_plate*)

bayespy.nodes.GaussianGammaARD.delete

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

bayespy.nodes.GaussianGammaARD.get_mask

GaussianGammaARD.**get_mask**()

bayespy.nodes.GaussianGammaARD.get_moments

GaussianGammaARD.**get_moments**()

bayespy.nodes.GaussianGammaARD.get_shape

GaussianGammaARD.**get_shape**(*ind*)

bayespy.nodes.GaussianGammaARD.has_plotter

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

bayespy.nodes.GaussianGammaARD.initialize_from_parameters

GaussianGammaARD.**initialize_from_parameters**(**args*)

bayespy.nodes.GaussianGammaARD.initialize_from_prior

GaussianGammaARD.**initialize_from_prior**()

bayespy.nodes.GaussianGammaARD.initialize_from_random

GaussianGammaARD.**initialize_from_random**()

bayespy.nodes.GaussianGammaARD.initialize_from_value

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

bayespy.nodes.GaussianGammaARD.load

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

bayespy.nodes.GaussianGammaARD.logpdf

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

bayespy.nodes.GaussianGammaARD.lower_bound_contribution

GaussianGammaARD.**lower_bound_contribution**(*gradient=False*)

bayespy.nodes.GaussianGammaARD.lowerbound

GaussianGammaARD.**lowerbound**()

bayespy.nodes.GaussianGammaARD.move_plates

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

bayespy.nodes.GaussianGammaARD.observe

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

bayespy.nodes.GaussianGammaARD.pdf

GaussianGammaARD.**pdf**(*X*, *mask=True*)

    Compute the probability density function of this node.

bayespy.nodes.GaussianGammaARD.plot

GaussianGammaARD.**plot**(***kwargs*)

    Plot the node distribution using the plotter of the node

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

bayespy.nodes.GaussianGammaARD.random

GaussianGammaARD.**random**()

    Draw a random sample from the distribution.

bayespy.nodes.GaussianGammaARD.save

GaussianGammaARD.**save**(*group*)

    Save the state of the node into a HDF5 file.

    group can be the root

bayespy.nodes.GaussianGammaARD.set_plotter

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

bayespy.nodes.GaussianGammaARD.unobserve

GaussianGammaARD.**unobserve**()

bayespy.nodes.GaussianGammaARD.update

GaussianGammaARD.**update**()

Attributes

dims	
plates	

bayespy.nodes.GaussianGammaARD.dims

GaussianGammaARD.**dims** = **None**

---

bayespy.nodes.GaussianGammaARD.plates

GaussianGammaARD.**plates** = None

bayespy.nodes.GaussianWishart

class bayespy.nodes.**GaussianWishart**(**args*, ***kwargs*)
Node for Gaussian-Wishart random variables.

The prior:

$$p(x, \Lambda | \mu, \alpha, V, n)$$
$$p(x | \Lambda, \mu, \alpha) = (N)(x | \mu, \alpha^{-1} Lambda^{-1})$$
$$p(\Lambda | V, n) = (W)(\Lambda | n, V)$$

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

Currently, supports only vector variables.

Methods

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

bayespy.nodes.GaussianWishart.add_plate_axis

GaussianWishart.**add_plate_axis**(*to_plate*)

bayespy.nodes.GaussianWishart.delete

GaussianWishart.**delete**()
>   Delete this node and the children

bayespy.nodes.GaussianWishart.get_mask

GaussianWishart.**get_mask**()

bayespy.nodes.GaussianWishart.get_moments

GaussianWishart.**get_moments**()

bayespy.nodes.GaussianWishart.get_shape

GaussianWishart.**get_shape**(*ind*)

bayespy.nodes.GaussianWishart.has_plotter

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

bayespy.nodes.GaussianWishart.initialize_from_parameters

GaussianWishart.**initialize_from_parameters**(**args*)

bayespy.nodes.GaussianWishart.initialize_from_prior

GaussianWishart.**initialize_from_prior**()

bayespy.nodes.GaussianWishart.initialize_from_random

GaussianWishart.**initialize_from_random**()

bayespy.nodes.GaussianWishart.initialize_from_value

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

bayespy.nodes.GaussianWishart.load

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

bayespy.nodes.GaussianWishart.logpdf

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

bayespy.nodes.GaussianWishart.lower_bound_contribution

GaussianWishart.**lower_bound_contribution**(*gradient=False*)

bayespy.nodes.GaussianWishart.lowerbound

GaussianWishart.**lowerbound**()

bayespy.nodes.GaussianWishart.move_plates

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

bayespy.nodes.GaussianWishart.observe

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

bayespy.nodes.GaussianWishart.pdf

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

bayespy.nodes.GaussianWishart.plot

GaussianWishart.**plot**(***kwargs*)
    Plot the node distribution using the plotter of the node

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

bayespy.nodes.GaussianWishart.random

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

bayespy.nodes.GaussianWishart.save

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

    group can be the root

bayespy.nodes.GaussianWishart.set_plotter

GaussianWishart.**set_plotter**(*plotter*)

bayespy.nodes.GaussianWishart.show

GaussianWishart.**show**()
> Print the distribution using standard parameterization.

bayespy.nodes.GaussianWishart.unobserve

GaussianWishart.**unobserve**()

bayespy.nodes.GaussianWishart.update

GaussianWishart.**update**()

Attributes

dims	
plates	

bayespy.nodes.GaussianWishart.dims

GaussianWishart.**dims** = None

bayespy.nodes.GaussianWishart.plates

GaussianWishart.**plates** = None

Nodes for discrete count variables:

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

bayespy.nodes.Bernoulli

class bayespy.nodes.**Bernoulli**(*p*, ***kwargs*)
> Node for Bernoulli random variables.

> The node models a binary random variable $z \in \{0, 1\}$ with prior probability $p \in [0, 1]$ for value one:

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

> > **Parameters** **p** : beta-like node

Probability of a successful trial

Examples

```python
from bayespy.nodes import Bernoulli, Beta
p = Beta([1e-3, 1e-3])
z = Bernoulli(p, plates=(10,))
z.observe([0, 1, 1, 1, 0, 1, 1, 1, 0, 1])
p.update()
import bayespy.plot as bpplt
import numpy as np
bpplt.pdf(p, np.linspace(0, 1, num=100))
```

Methods

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

bayespy.nodes.Bernoulli.add_plate_axis

Bernoulli.**add_plate_axis**(*to_plate*)

bayespy.nodes.Bernoulli.delete

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

bayespy.nodes.Bernoulli.get_mask

Bernoulli.**get_mask**()

bayespy.nodes.Bernoulli.get_moments

Bernoulli.**get_moments**()

bayespy.nodes.Bernoulli.get_shape

Bernoulli.**get_shape**(*ind*)

bayespy.nodes.Bernoulli.has_plotter

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

bayespy.nodes.Bernoulli.initialize_from_parameters

Bernoulli.**initialize_from_parameters**(**args*)

bayespy.nodes.Bernoulli.initialize_from_prior

Bernoulli.**initialize_from_prior**()

bayespy.nodes.Bernoulli.initialize_from_random

Bernoulli.**initialize_from_random**()

bayespy.nodes.Bernoulli.initialize_from_value

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

bayespy.nodes.Bernoulli.load

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

bayespy.nodes.Bernoulli.logpdf

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

bayespy.nodes.Bernoulli.lower_bound_contribution

Bernoulli.**lower_bound_contribution**(*gradient=False*)

bayespy.nodes.Bernoulli.lowerbound

Bernoulli.**lowerbound**()

bayespy.nodes.Bernoulli.move_plates

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

bayespy.nodes.Bernoulli.observe

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

bayespy.nodes.Bernoulli.pdf

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

bayespy.nodes.Bernoulli.plot

Bernoulli.**plot**(***kwargs*)
    Plot the node distribution using the plotter of the node

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

bayespy.nodes.Bernoulli.random

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

bayespy.nodes.Bernoulli.save

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

    group can be the root

bayespy.nodes.Bernoulli.set_plotter

Bernoulli.**set_plotter**(*plotter*)

bayespy.nodes.Bernoulli.show

`Bernoulli.`**`show`**`()`
    Print the distribution using standard parameterization.

bayespy.nodes.Bernoulli.unobserve

`Bernoulli.`**`unobserve`**`()`

bayespy.nodes.Bernoulli.update

`Bernoulli.`**`update`**`()`

Attributes

dims
plates

bayespy.nodes.Bernoulli.dims

`Bernoulli.`**`dims`** = **None**

bayespy.nodes.Bernoulli.plates

`Bernoulli.`**`plates`** = **None**

bayespy.nodes.Binomial

class `bayespy.nodes.`**`Binomial`**(*n, p, **kwargs*)
    Node for binomial random variables.

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

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

        Parameters **n** : scalar or array

            Number of trials

        p : beta-like node or scalar or array

            Probability of a success in a trial

    See also:

    `Bernoulli`, `Multinomial`, `Beta`

Examples

```python
from bayespy.nodes import Binomial, Beta
p = Beta([1e-3, 1e-3])
x = Binomial(10, p)
x.observe(7)
p.update()
import bayespy.plot as bpplt
import numpy as np
bpplt.pdf(p, np.linspace(0, 1, num=100))
```

Methods

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

bayespy.nodes.Binomial.add_plate_axis

Binomial.**add_plate_axis**(*to_plate*)

bayespy.nodes.Binomial.delete

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

bayespy.nodes.Binomial.get_mask

Binomial.**get_mask**()

bayespy.nodes.Binomial.get_moments

Binomial.**get_moments**()

bayespy.nodes.Binomial.get_shape

Binomial.**get_shape**(*ind*)

bayespy.nodes.Binomial.has_plotter

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

bayespy.nodes.Binomial.initialize_from_parameters

Binomial.**initialize_from_parameters**(**args*)

bayespy.nodes.Binomial.initialize_from_prior

Binomial.**initialize_from_prior**()

bayespy.nodes.Binomial.initialize_from_random

Binomial.**initialize_from_random**()

bayespy.nodes.Binomial.initialize_from_value

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

bayespy.nodes.Binomial.load

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

bayespy.nodes.Binomial.logpdf

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

bayespy.nodes.Binomial.lower_bound_contribution

Binomial.**lower_bound_contribution**(*gradient=False*)

bayespy.nodes.Binomial.lowerbound

`Binomial.lowerbound()`

bayespy.nodes.Binomial.move_plates

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

bayespy.nodes.Binomial.observe

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

bayespy.nodes.Binomial.pdf

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

bayespy.nodes.Binomial.plot

`Binomial.plot`(***kwargs*)
   Plot the node distribution using the plotter of the node

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

bayespy.nodes.Binomial.random

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

bayespy.nodes.Binomial.save

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

   group can be the root

bayespy.nodes.Binomial.set_plotter

`Binomial.set_plotter`(*plotter*)

bayespy.nodes.Binomial.show

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

5.1. Nodes   **73**

bayespy.nodes.Binomial.unobserve

`Binomial.`**`unobserve`**`()`

bayespy.nodes.Binomial.update

`Binomial.`**`update`**`()`

Attributes

dims	
plates	

bayespy.nodes.Binomial.dims

`Binomial.`**`dims`** = None

bayespy.nodes.Binomial.plates

`Binomial.`**`plates`** = None

bayespy.nodes.Categorical

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

Node for categorical random variables.

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

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

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

Probabilities for each category

See also:

`Bernoulli`, `Multinomial`, `Dirichlet`

Methods

add_plate_axis(to_plate)	
delete()	Delete this node and the children
get_mask()	
get_moments()	
get_shape(ind)	
has_plotter()	Return True if the node has a plotter
initialize_from_parameters(*args)	

<div align="center">Table 5.25 – continued from previous page</div>

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

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

dims	
plates	

bayespy.nodes.Categorical.dims

Categorical.**dims** = None

---

bayespy.nodes.Categorical.plates

Categorical.**plates** = None

bayespy.nodes.Multinomial

class bayespy.nodes.**Multinomial**(*n, p, **kwargs*)

Node for multinomial random variables.

Assume there are K categories and N trials each of which leads a success for exactly one of the categories. Given the probabilities $p_0, \ldots, p_{K-1}$ for the categories, multinomial distribution is gives the probability of any combination of numbers of successes for the categories.

The node models the number of successes $x_k \in \{0, \ldots, n\}$ in n trials with probability p_k for success in K categories.

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

Parameters **n** : scalar or array

N, number of trials

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

p, probabilities of successes for the categories

See also:

Dirichlet, Binomial, Categorical

Methods

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

Table 5.27 – continued from previous page

`unobserve()`	
`update()`	

bayespy.nodes.Multinomial.add_plate_axis

Multinomial.**add_plate_axis**(*to_plate*)

bayespy.nodes.Multinomial.delete

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

bayespy.nodes.Multinomial.get_mask

Multinomial.**get_mask**()

bayespy.nodes.Multinomial.get_moments

Multinomial.**get_moments**()

bayespy.nodes.Multinomial.get_shape

Multinomial.**get_shape**(*ind*)

bayespy.nodes.Multinomial.has_plotter

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

bayespy.nodes.Multinomial.initialize_from_parameters

Multinomial.**initialize_from_parameters**(**args*)

bayespy.nodes.Multinomial.initialize_from_prior

Multinomial.**initialize_from_prior**()

bayespy.nodes.Multinomial.initialize_from_random

Multinomial.**initialize_from_random**()

bayespy.nodes.Multinomial.initialize_from_value

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

bayespy.nodes.Multinomial.load

Multinomial.**load**(*group*)

    Load the state of the node from a HDF5 file.

bayespy.nodes.Multinomial.logpdf

Multinomial.**logpdf**(*X*, *mask=True*)

    Compute the log probability density function Q(X) of this node.

bayespy.nodes.Multinomial.lower_bound_contribution

Multinomial.**lower_bound_contribution**(*gradient=False*)

bayespy.nodes.Multinomial.lowerbound

Multinomial.**lowerbound**()

bayespy.nodes.Multinomial.move_plates

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

bayespy.nodes.Multinomial.observe

Multinomial.**observe**(*x*, **args*, *mask=True*)

    Fix moments, compute f and propagate mask.

bayespy.nodes.Multinomial.pdf

Multinomial.**pdf**(*X*, *mask=True*)

    Compute the probability density function of this node.

bayespy.nodes.Multinomial.plot

Multinomial.**plot**(***kwargs*)

    Plot the node distribution using the plotter of the node

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

bayespy.nodes.Multinomial.random

Multinomial.**random**()

    Draw a random sample from the distribution.

bayespy.nodes.Multinomial.save

Multinomial.**save**(*group*)

 Save the state of the node into a HDF5 file.

 group can be the root

bayespy.nodes.Multinomial.set_plotter

Multinomial.**set_plotter**(*plotter*)

bayespy.nodes.Multinomial.show

Multinomial.**show**()

 Print the distribution using standard parameterization.

bayespy.nodes.Multinomial.unobserve

Multinomial.**unobserve**()

bayespy.nodes.Multinomial.update

Multinomial.**update**()

Attributes

dims	
plates	

bayespy.nodes.Multinomial.dims

Multinomial.**dims** = None

bayespy.nodes.Multinomial.plates

Multinomial.**plates** = None

bayespy.nodes.Poisson

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

 Node for Poisson random variables.

 The node uses Poisson distribution:

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

 where λ is the rate parameter.

> **Parameters l** : gamma-like node or scalar or array
>
> > λ, rate parameter

See also:

Gamma, Exponential

Methods

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

bayespy.nodes.Poisson.add_plate_axis

Poisson.**add_plate_axis**(*to_plate*)

bayespy.nodes.Poisson.delete

Poisson.**delete**()
> Delete this node and the children

bayespy.nodes.Poisson.get_mask

Poisson.**get_mask**()

bayespy.nodes.Poisson.get_moments

Poisson.**get_moments**()

bayespy.nodes.Poisson.get_shape

Poisson.**get_shape**(*ind*)

bayespy.nodes.Poisson.has_plotter

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

bayespy.nodes.Poisson.initialize_from_parameters

Poisson.**initialize_from_parameters**(**args*)

bayespy.nodes.Poisson.initialize_from_prior

Poisson.**initialize_from_prior**()

bayespy.nodes.Poisson.initialize_from_random

Poisson.**initialize_from_random**()

bayespy.nodes.Poisson.initialize_from_value

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

bayespy.nodes.Poisson.load

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

bayespy.nodes.Poisson.logpdf

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

bayespy.nodes.Poisson.lower_bound_contribution

Poisson.**lower_bound_contribution**(*gradient=False*)

bayespy.nodes.Poisson.lowerbound

Poisson.**lowerbound**()

bayespy.nodes.Poisson.move_plates

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

bayespy.nodes.Poisson.observe

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

bayespy.nodes.Poisson.pdf

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

bayespy.nodes.Poisson.plot

Poisson.**plot**(***kwargs*)
  Plot the node distribution using the plotter of the node

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

bayespy.nodes.Poisson.random

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

bayespy.nodes.Poisson.save

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

  group can be the root

bayespy.nodes.Poisson.set_plotter

Poisson.**set_plotter**(*plotter*)

bayespy.nodes.Poisson.show

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

bayespy.nodes.Poisson.unobserve

Poisson.**unobserve**()

bayespy.nodes.Poisson.update

```
Poisson.update()
```

Attributes

dims	tuple() -> empty tuple
plates	

bayespy.nodes.Poisson.dims

```
Poisson.dims = ((),)
```

bayespy.nodes.Poisson.plates

```
Poisson.plates = None
```

Nodes for probabilities:

Beta(alpha, **kwargs)	Node for beta random variables.
Dirichlet(*args, **kwargs)	Node for Dirichlet random variables.

bayespy.nodes.Beta

class bayespy.nodes.**Beta**(*alpha*, ***kwargs*)

Node for beta random variables.

The node models a probability variable $p \in [0, 1]$ as

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

where a and b are prior counts for success and failure, respectively.

> **Parameters** **alpha** : (...,2)-shaped array
>
> > Two-element vector containing a and b

Examples

```python
from bayespy.nodes import Bernoulli, Beta
p = Beta([1e-3, 1e-3])
z = Bernoulli(p, plates=(10,))
z.observe([0, 1, 1, 1, 0, 1, 1, 1, 0, 1])
p.update()
import bayespy.plot as bpplt
import numpy as np
bpplt.pdf(p, np.linspace(0, 1, num=100))
```

Methods

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

bayespy.nodes.Beta.add_plate_axis

Beta.**add_plate_axis**(*to_plate*)

bayespy.nodes.Beta.delete

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

bayespy.nodes.Beta.get_mask

Beta.**get_mask**()

bayespy.nodes.Beta.get_moments

Beta.**get_moments**()

bayespy.nodes.Beta.get_shape

Beta.**get_shape**(*ind*)

bayespy.nodes.Beta.has_plotter

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

bayespy.nodes.Beta.initialize_from_parameters

Beta.**initialize_from_parameters**(*args*)

bayespy.nodes.Beta.initialize_from_prior

Beta.**initialize_from_prior**()

bayespy.nodes.Beta.initialize_from_random

Beta.**initialize_from_random**()

bayespy.nodes.Beta.initialize_from_value

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

bayespy.nodes.Beta.load

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

bayespy.nodes.Beta.logpdf

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

bayespy.nodes.Beta.lower_bound_contribution

Beta.**lower_bound_contribution**(*gradient=False*)

bayespy.nodes.Beta.lowerbound

Beta.**lowerbound**()

bayespy.nodes.Beta.move_plates

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

bayespy.nodes.Beta.observe

Beta.**observe**(*x*, **args*, *mask=True*)

    Fix moments, compute f and propagate mask.

bayespy.nodes.Beta.pdf

Beta.**pdf**(*X*, *mask=True*)

    Compute the probability density function of this node.

bayespy.nodes.Beta.plot

Beta.**plot**(***kwargs*)

    Plot the node distribution using the plotter of the node

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

bayespy.nodes.Beta.random

Beta.**random**()

    Draw a random sample from the distribution.

bayespy.nodes.Beta.save

Beta.**save**(*group*)

    Save the state of the node into a HDF5 file.

    group can be the root

bayespy.nodes.Beta.set_plotter

Beta.**set_plotter**(*plotter*)

bayespy.nodes.Beta.show

Beta.**show**()

    Print the distribution using standard parameterization.

bayespy.nodes.Beta.unobserve

Beta.**unobserve**()

bayespy.nodes.Beta.update

Beta.**update**()

Attributes

dims	
plates	

bayespy.nodes.Beta.dims

Beta.**dims** = None

bayespy.nodes.Beta.plates

Beta.**plates** = None

bayespy.nodes.Dirichlet

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

Node for Dirichlet random variables.

The node models a set of probabilities $\{\pi_0, \ldots, \pi_{K-1}\}$ which satisfy $\sum_{k=0}^{K-1} \pi_k = 1$ and $\pi_k \in [0,1] \; \forall k = 0, \ldots, K-1$.

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

where α_k are concentration parameters.

The posterior approximation has the same functional form but with different concentration parameters.

> **Parameters alpha** : (...,K)-shaped array
>
> > Prior counts α_k

See also:

Beta, Categorical, Multinomial, CategoricalMarkovChain

Methods

add_plate_axis(to_plate)	
delete()	Delete this node and the children
get_mask()	
get_moments()	
get_shape(ind)	
has_plotter()	Return True if the node has a plotter
initialize_from_parameters(*args)	
initialize_from_prior()	
initialize_from_random()	
initialize_from_value(x, *args)	
load(group)	Load the state of the node from a HDF5 file.
logpdf(X[, mask])	Compute the log probability density function Q(X) of this node.
lower_bound_contribution([gradient])	
lowerbound()	

Table 5.34 – continued from previous page

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

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**()

> Draw a random sample from the distribution.

bayespy.nodes.Dirichlet.save

Dirichlet.**save**(*group*)

> Save the state of the node into a HDF5 file.

> group can be the root

bayespy.nodes.Dirichlet.set_plotter

Dirichlet.**set_plotter**(*plotter*)

bayespy.nodes.Dirichlet.show

Dirichlet.**show**()

> Print the distribution using standard parameterization.

bayespy.nodes.Dirichlet.unobserve

Dirichlet.**unobserve**()

bayespy.nodes.Dirichlet.update

Dirichlet.**update**()

Attributes

dims	
plates	

bayespy.nodes.Dirichlet.dims

Dirichlet.**dims** = None

---

bayespy.nodes.Dirichlet.plates

`Dirichlet.`**`plates`** `= None`

Nodes for dynamic variables:

`CategoricalMarkovChain`(pi, A[, states])	Node for categorical Markov chain random variables.
`GaussianMarkovChain`(mu, Lambda, A, nu[, n])	Node for Gaussian Markov chain random variables.
`SwitchingGaussianMarkovChain`(mu, Lambda, B, ...)	Node for Gaussian Markov chain random variables with switching d
`VaryingGaussianMarkovChain`(mu, Lambda, B, S, nu)	Node for Gaussian Markov chain random variables with time-varyin

bayespy.nodes.CategoricalMarkovChain

class `bayespy.nodes.`**`CategoricalMarkovChain`**(*pi*, *A*, *states=None*, ***kwargs*)

Node for categorical Markov chain random variables.

The node models a Markov chain which has a discrete set of K possible states and the next state depends only on the previous state and the state transition probabilities. The graphical model is shown below:
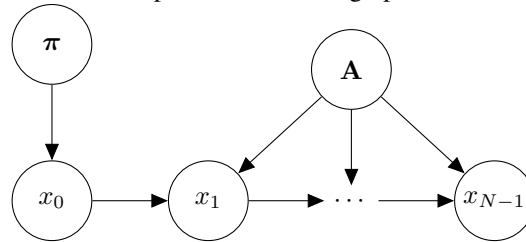
where π contains the probabilities for the initial state and $\mathbf{A}$ is the state transition probability matrix. It is possible to have $\mathbf{A}$ varying in time.

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

where

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

This node can be used to construct hidden Markov models by using `Mixture` for the emission distribution.

> **Parameters** **pi** : Dirichlet-like node or (...,K)-array
>
> > π, probabilities for the first state. K-dimensional Dirichlet.
>
> **A** : Dirichlet-like node or (K,K)-array or (...,1,K,K)-array or (...,N-1,K,K)-array
>
> > $\mathbf{A}$, probabilities for state transitions. K-dimensional Dirichlet with plates (K,) or (...,1,K) or (...,N-1,K).
>
> **states** : int, optional
>
> > N, the length of the chain.

See also:

`Categorical`, `Dirichlet`, `GaussianMarkovChain`, `Mixture`, `SwitchingGaussianMarkovChain`

Methods

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

bayespy.nodes.CategoricalMarkovChain.add_plate_axis

CategoricalMarkovChain.**add_plate_axis**(*to_plate*)

bayespy.nodes.CategoricalMarkovChain.delete

CategoricalMarkovChain.**delete**()
> Delete this node and the children

bayespy.nodes.CategoricalMarkovChain.get_mask

CategoricalMarkovChain.**get_mask**()

bayespy.nodes.CategoricalMarkovChain.get_moments

CategoricalMarkovChain.**get_moments**()

bayespy.nodes.CategoricalMarkovChain.get_shape

CategoricalMarkovChain.**get_shape**(*ind*)

bayespy.nodes.CategoricalMarkovChain.has_plotter

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

bayespy.nodes.CategoricalMarkovChain.initialize_from_parameters

CategoricalMarkovChain.**initialize_from_parameters**(*args*)

bayespy.nodes.CategoricalMarkovChain.initialize_from_prior

CategoricalMarkovChain.**initialize_from_prior**()

bayespy.nodes.CategoricalMarkovChain.initialize_from_random

CategoricalMarkovChain.**initialize_from_random**()

bayespy.nodes.CategoricalMarkovChain.initialize_from_value

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

bayespy.nodes.CategoricalMarkovChain.load

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

bayespy.nodes.CategoricalMarkovChain.logpdf

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

bayespy.nodes.CategoricalMarkovChain.lower_bound_contribution

CategoricalMarkovChain.**lower_bound_contribution**(*gradient=False*)

bayespy.nodes.CategoricalMarkovChain.lowerbound

CategoricalMarkovChain.**lowerbound**()

bayespy.nodes.CategoricalMarkovChain.move_plates

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

bayespy.nodes.CategoricalMarkovChain.observe

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

bayespy.nodes.CategoricalMarkovChain.pdf

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

bayespy.nodes.CategoricalMarkovChain.plot

CategoricalMarkovChain.**plot**(***kwargs*)
    Plot the node distribution using the plotter of the node

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

bayespy.nodes.CategoricalMarkovChain.random

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

bayespy.nodes.CategoricalMarkovChain.save

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

    group can be the root

bayespy.nodes.CategoricalMarkovChain.set_plotter

CategoricalMarkovChain.**set_plotter**(*plotter*)

bayespy.nodes.CategoricalMarkovChain.show

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

bayespy.nodes.CategoricalMarkovChain.unobserve

CategoricalMarkovChain.**unobserve**()

bayespy.nodes.CategoricalMarkovChain.update

CategoricalMarkovChain.**update**()

Attributes

dims
plates

bayespy.nodes.CategoricalMarkovChain.dims

CategoricalMarkovChain.**dims** = None

bayespy.nodes.CategoricalMarkovChain.plates

CategoricalMarkovChain.**plates** = None

bayespy.nodes.GaussianMarkovChain

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

Node for Gaussian Markov chain random variables.

In a simple case, the graphical model can be presented as:
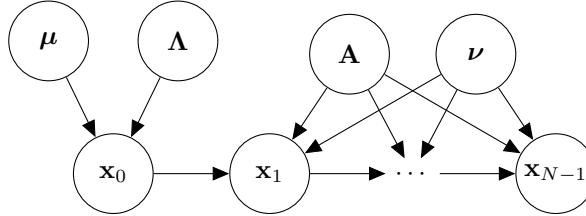
where $\boldsymbol{\mu}$ and $\boldsymbol{\Lambda}$ are the mean and the precision matrix of the initial state, $\mathbf{A}$ is the state dynamics matrix and $\boldsymbol{\nu}$ is the precision of the innovation noise. It is possible that $\mathbf{A}$ and/or $\boldsymbol{\nu}$ are different for each transition instead of being constant.

The probability distribution is

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

where

$$p(\mathbf{x}_0) = \mathcal{N}(\mathbf{x}_0 | \boldsymbol{\mu}, \boldsymbol{\Lambda})$$
$$p(\mathbf{x}_n | \mathbf{x}_{n-1}) = \mathcal{N}(\mathbf{x}_n | \mathbf{A}_{n-1} \mathbf{x}_{n-1}, \mathrm{diag}(\boldsymbol{\nu}_{n-1})).$$

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

$\boldsymbol{\mu}$, mean of x_0, D-dimensional with plates (...)

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

$\boldsymbol{\Lambda}$, precision matrix of x_0, $D \times D$ -dimensional with plates (...)

A : Gaussian-like node or (D,D)-array or (...,1,D,D)-array or (...,N-1,D,D)-array

$\mathbf{A}$, state dynamics matrix, D-dimensional with plates (D,) or (...,1,D) or (...,N-1,D)

nu : gamma-like node or (D,)-array or (...,1,D)-array or (...,N-1,D)-array

$\boldsymbol{\nu}$, diagonal elements of the precision of the innovation process, plates (D,) or (...,1,D) or (...,N-1,D)

n : int, optional

N, the length of the chain. Must be given if $\mathbf{A}$ and ν are constant over time.

See also:

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

Methods

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

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

bayespy.nodes.GaussianMarkovChain.rotate

GaussianMarkovChain.**rotate**(*R*, *inv=None*, *logdet=None*)

bayespy.nodes.GaussianMarkovChain.save

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

    group can be the root

bayespy.nodes.GaussianMarkovChain.set_plotter

GaussianMarkovChain.**set_plotter**(*plotter*)

bayespy.nodes.GaussianMarkovChain.show

GaussianMarkovChain.**show**()

bayespy.nodes.GaussianMarkovChain.unobserve

GaussianMarkovChain.**unobserve**()

bayespy.nodes.GaussianMarkovChain.update

GaussianMarkovChain.**update**()

Attributes

dims	
plates	

bayespy.nodes.GaussianMarkovChain.dims

GaussianMarkovChain.**dims** = None

bayespy.nodes.GaussianMarkovChain.plates

GaussianMarkovChain.**plates** = None

bayespy.nodes.SwitchingGaussianMarkovChain

class bayespy.nodes.**SwitchingGaussianMarkovChain**(*mu*, *Lambda*, *B*, *Z*, *nu*, *n=None*, ***kwargs*)

Node for Gaussian Markov chain random variables with switching dynamics.

The node models a sequence of Gaussian variables :math:'mathbf{x}_0,ldots,mathbf{x}_{N-1}$ with linear Markovian dynamics. The dynamics may change in time, which is obtained by having a set of matrices and at each time selecting one of them as the state dynamics matrix. The graphical model can be presented as:
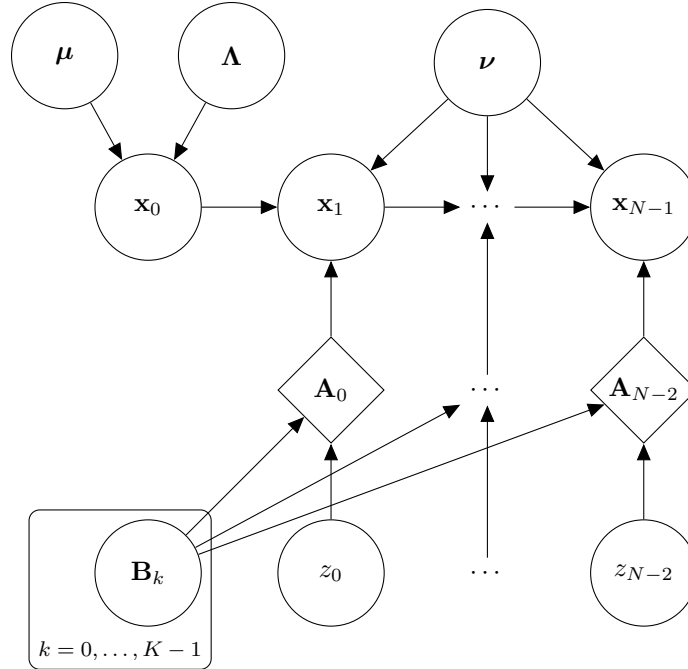
where $\boldsymbol{\mu}$ and $\boldsymbol{\Lambda}$ are the mean and the precision matrix of the initial state, $\boldsymbol{\nu}$ is the precision of the innovation noise, and $\mathbf{A}_n$ are the state dynamics matrix obtained by selecting one of the matrices $\{\mathbf{B}_k\}_{k=0}^{K-1}$ at each time. The selections are provided by $z_n \in \{0, \ldots, K-1\}$. The probability distribution is

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

where

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

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

$\boldsymbol{\mu}$, mean of x_0, D-dimensional with plates (...)

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

$\boldsymbol{\Lambda}$, precision matrix of x_0, $D \times D$ -dimensional with plates (...)

B : Gaussian-like node or (...,D,D,K)-array

$\{\mathbf{B}_k\}_{k=0}^{K-1}$, a set of state dynamics matrix, $D \times K$-dimensional with plates (...,D)

Z : categorical-like node or (...,N-1)-array

$\{z_0, \ldots, z_{N-2}\}$, time-dependent selection, K-categorical with plates (...,N-1)

nu : gamma-like node or (...,D)-array

$\boldsymbol{\nu}$, diagonal elements of the precision of the innovation process, plates (...,D)

n : int, optional

N, the length of the chain. Must be given if $\mathbf{Z}$ does not have plates over the time domain (which would not make sense).

See also:

Gaussian, GaussianARD, Wishart, Gamma, GaussianMarkovChain, VaryingGaussianMarkovChain, Categorical, CategoricalMarkovChain

Notes

Equivalent model block can be constructed with GaussianMarkovChain by explicitly using Gate to select the state dynamics matrix. However, that approach is not very efficient for large datasets because it does not utilize the structure of $\mathbf{A}_n$, thus it explicitly computes huge moment arrays.

Methods

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

bayespy.nodes.SwitchingGaussianMarkovChain.add_plate_axis

SwitchingGaussianMarkovChain.**add_plate_axis**(*to_plate*)

bayespy.nodes.SwitchingGaussianMarkovChain.delete

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

bayespy.nodes.SwitchingGaussianMarkovChain.get_mask

SwitchingGaussianMarkovChain.**get_mask**()

bayespy.nodes.SwitchingGaussianMarkovChain.get_moments

SwitchingGaussianMarkovChain.**get_moments**()

bayespy.nodes.SwitchingGaussianMarkovChain.get_shape

SwitchingGaussianMarkovChain.**get_shape**(*ind*)

bayespy.nodes.SwitchingGaussianMarkovChain.has_plotter

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

bayespy.nodes.SwitchingGaussianMarkovChain.initialize_from_parameters

SwitchingGaussianMarkovChain.**initialize_from_parameters**(**args*)

bayespy.nodes.SwitchingGaussianMarkovChain.initialize_from_prior

SwitchingGaussianMarkovChain.**initialize_from_prior**()

bayespy.nodes.SwitchingGaussianMarkovChain.initialize_from_random

SwitchingGaussianMarkovChain.**initialize_from_random**()

bayespy.nodes.SwitchingGaussianMarkovChain.initialize_from_value

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

bayespy.nodes.SwitchingGaussianMarkovChain.load

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

bayespy.nodes.SwitchingGaussianMarkovChain.logpdf

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

bayespy.nodes.SwitchingGaussianMarkovChain.lower_bound_contribution

SwitchingGaussianMarkovChain.**lower_bound_contribution**(*gradient=False*)

bayespy.nodes.SwitchingGaussianMarkovChain.lowerbound

SwitchingGaussianMarkovChain.**lowerbound**()

bayespy.nodes.SwitchingGaussianMarkovChain.move_plates

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

bayespy.nodes.SwitchingGaussianMarkovChain.observe

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

bayespy.nodes.SwitchingGaussianMarkovChain.pdf

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

bayespy.nodes.SwitchingGaussianMarkovChain.plot

SwitchingGaussianMarkovChain.**plot**(***kwargs*)
    Plot the node distribution using the plotter of the node

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

bayespy.nodes.SwitchingGaussianMarkovChain.random

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

bayespy.nodes.SwitchingGaussianMarkovChain.rotate

SwitchingGaussianMarkovChain.**rotate**(*R*, *inv=None*, *logdet=None*)

bayespy.nodes.SwitchingGaussianMarkovChain.save

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

    group can be the root

bayespy.nodes.SwitchingGaussianMarkovChain.set_plotter

SwitchingGaussianMarkovChain.**set_plotter**(*plotter*)

bayespy.nodes.SwitchingGaussianMarkovChain.show

SwitchingGaussianMarkovChain.**show**()

bayespy.nodes.SwitchingGaussianMarkovChain.unobserve

SwitchingGaussianMarkovChain.**unobserve**()

bayespy.nodes.SwitchingGaussianMarkovChain.update

SwitchingGaussianMarkovChain.**update**()

Attributes

dims	
plates	

bayespy.nodes.SwitchingGaussianMarkovChain.dims

SwitchingGaussianMarkovChain.**dims** = None

bayespy.nodes.SwitchingGaussianMarkovChain.plates

SwitchingGaussianMarkovChain.**plates** = None

bayespy.nodes.VaryingGaussianMarkovChain

class bayespy.nodes.**VaryingGaussianMarkovChain**(*mu*, *Lambda*, *B*, *S*, *nu*, *n=None*, ***kwargs*)
    Node for Gaussian Markov chain random variables with time-varying dynamics.

The node models a sequence of Gaussian variables :math:'mathbf{x}_0,ldots,mathbf{x}_{N-1}$ with linear Markovian dynamics. The time variability of the dynamics is obtained by modelling the state dynamics matrix as a linear combination of a set of matrices with time-varying linear combination weights [R1]. The graphical model can be presented as:
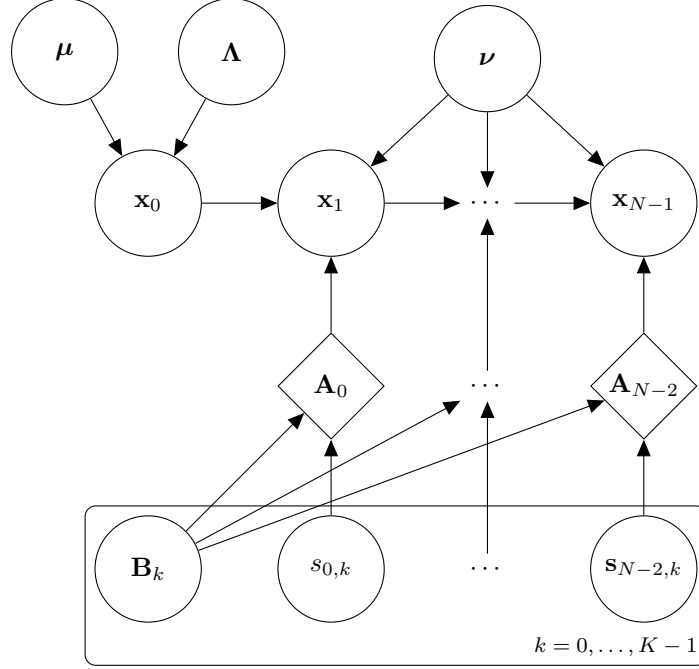
where $\boldsymbol{\mu}$ and $\boldsymbol{\Lambda}$ are the mean and the precision matrix of the initial state, $\boldsymbol{\nu}$ is the precision of the innovation noise, and $\mathbf{A}_n$ are the state dynamics matrix obtained by mixing matrices $\mathbf{B}_k$ with weights $s_{n,k}$.

The probability distribution is

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

where

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

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

> $\boldsymbol{\mu}$, mean of x_0, D-dimensional with plates (...)

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

> $\boldsymbol{\Lambda}$, precision matrix of x_0, $D \times D$ -dimensional with plates (...)

B : Gaussian-like node or (...,D,D,K)-array

> $\{\mathbf{B}_k\}_{k=0}^{K-1}$, a set of state dynamics matrix, $D \times K$-dimensional with plates (...,D)

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

> $\{\mathbf{s}_0, \ldots, \mathbf{s}_{N-2}\}$, time-varying weights of the linear combination, K-dimensional with plates (...,N-1)

nu : gamma-like node or (...,D)-array

> $\boldsymbol{\nu}$, diagonal elements of the precision of the innovation process, plates (...,D)

n : int, optional

N, the length of the chain. Must be given if $\mathbf{S}$ does not have plates over the time domain (which would not make sense).

See also:

Gaussian, GaussianARD, Wishart, Gamma, GaussianMarkovChain, SwitchingGaussianMarkovChain

Notes

Equivalent model block can be constructed with GaussianMarkovChain by explicitly using SumMultiply to compute the linear combination. However, that approach is not very efficient for large datasets because it does not utilize the structure of $\mathbf{A}_n$, thus it explicitly computes huge moment arrays.

References

[R1]

Methods

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

bayespy.nodes.VaryingGaussianMarkovChain.add_plate_axis

VaryingGaussianMarkovChain.**add_plate_axis**(*to_plate*)

bayespy.nodes.VaryingGaussianMarkovChain.delete

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

bayespy.nodes.VaryingGaussianMarkovChain.get_mask

VaryingGaussianMarkovChain.**get_mask**()

bayespy.nodes.VaryingGaussianMarkovChain.get_moments

VaryingGaussianMarkovChain.**get_moments**()

bayespy.nodes.VaryingGaussianMarkovChain.get_shape

VaryingGaussianMarkovChain.**get_shape**(*ind*)

bayespy.nodes.VaryingGaussianMarkovChain.has_plotter

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

bayespy.nodes.VaryingGaussianMarkovChain.initialize_from_parameters

VaryingGaussianMarkovChain.**initialize_from_parameters**(**args*)

bayespy.nodes.VaryingGaussianMarkovChain.initialize_from_prior

VaryingGaussianMarkovChain.**initialize_from_prior**()

bayespy.nodes.VaryingGaussianMarkovChain.initialize_from_random

VaryingGaussianMarkovChain.**initialize_from_random**()

bayespy.nodes.VaryingGaussianMarkovChain.initialize_from_value

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

bayespy.nodes.VaryingGaussianMarkovChain.load

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

bayespy.nodes.VaryingGaussianMarkovChain.logpdf

VaryingGaussianMarkovChain.**logpdf**(*X*, *mask=True*)

> Compute the log probability density function Q(X) of this node.

bayespy.nodes.VaryingGaussianMarkovChain.lower_bound_contribution

VaryingGaussianMarkovChain.**lower_bound_contribution**(*gradient=False*)

bayespy.nodes.VaryingGaussianMarkovChain.lowerbound

VaryingGaussianMarkovChain.**lowerbound**()

bayespy.nodes.VaryingGaussianMarkovChain.move_plates

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

bayespy.nodes.VaryingGaussianMarkovChain.observe

VaryingGaussianMarkovChain.**observe**(*x*, **args*, *mask=True*)

> Fix moments, compute f and propagate mask.

bayespy.nodes.VaryingGaussianMarkovChain.pdf

VaryingGaussianMarkovChain.**pdf**(*X*, *mask=True*)

> Compute the probability density function of this node.

bayespy.nodes.VaryingGaussianMarkovChain.plot

VaryingGaussianMarkovChain.**plot**(***kwargs*)

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

bayespy.nodes.VaryingGaussianMarkovChain.random

VaryingGaussianMarkovChain.**random**()

> Draw a random sample from the distribution.

bayespy.nodes.VaryingGaussianMarkovChain.rotate

VaryingGaussianMarkovChain.**rotate**(*R*, *inv=None*, *logdet=None*)

bayespy.nodes.VaryingGaussianMarkovChain.save

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

  group can be the root

bayespy.nodes.VaryingGaussianMarkovChain.set_plotter

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

bayespy.nodes.VaryingGaussianMarkovChain.show

VaryingGaussianMarkovChain.**show**()

bayespy.nodes.VaryingGaussianMarkovChain.unobserve

VaryingGaussianMarkovChain.**unobserve**()

bayespy.nodes.VaryingGaussianMarkovChain.update

VaryingGaussianMarkovChain.**update**()

Attributes

dims	
plates	

bayespy.nodes.VaryingGaussianMarkovChain.dims

VaryingGaussianMarkovChain.**dims** = None

bayespy.nodes.VaryingGaussianMarkovChain.plates

VaryingGaussianMarkovChain.**plates** = None

Other stochastic nodes:

Mixture(z, node_class, *params[, cluster_plate])	Node for exponential family mixture variables.

bayespy.nodes.Mixture

class bayespy.nodes.**Mixture**(*z*, *node_class*, **params*, *cluster_plate=-1*, ***kwargs*)
  Node for exponential family mixture variables.

  The node represents a random variable which is sampled from a mixture distribution. It is possible to mix any

exponential family distribution. The probability density function is

$$p(x|z = k, \boldsymbol{\theta}_0, \ldots, \boldsymbol{\theta}_{K-1}) = \phi(x|\boldsymbol{\theta}_k),$$

where ϕ is the probability density function of the mixed exponential family distribution and $\boldsymbol{\theta}_0, \ldots, \boldsymbol{\theta}_{K-1}$ are the parameters of each cluster. For instance, ϕ could be the Gaussian probability density function $\mathcal{N}$ and $\boldsymbol{\theta}_k = \{\boldsymbol{\mu}_k, \boldsymbol{\Lambda}_k\}$ where $\boldsymbol{\mu}_k$ and $\boldsymbol{\Lambda}_k$ are the mean vector and precision matrix for cluster k.

> **Parameters** **z** : categorical-like node or array
>
> > z, cluster assignment
>
> **node_class** : stochastic exponential family node class
>
> > Mixed distribution
>
> **params** : types specified by the mixed distribution
>
> > Parameters of the mixed distribution. If some parameters should vary between clusters, those parameters' plate axis *cluster_plate* should have a size which equals the number of clusters. For parameters with shared values, that plate axis should have length 1. At least one parameter should vary between clusters.
>
> **cluster_plate** : int, optional
>
> > Negative integer defining which plate axis is used for the clusters in the parameters. That plate axis is ignored from the parameters when considering the plates for this node. By default, mix over the last plate axis.

See also:

`Categorical`, `CategoricalMarkovChain`

Examples

A simple 2-dimensional Gaussian mixture model with three clusters for 100 samples can be constructed, for instance, as:

```python
from bayespy.nodes import (Dirichlet, Categorical, Mixture,
                           Gaussian, Wishart)
alpha = Dirichlet([1e-3, 1e-3, 1e-3])
Z = Categorical(alpha, plates=(100,))
mu = Gaussian(np.zeros(2), 1e-6*np.identity(2), plates=(3,))
Lambda = Wishart(2, 1e-6*np.identity(2), plates=(3,))
X = Mixture(Z, Gaussian, mu, Lambda)
```

Methods

`add_plate_axis`(to_plate)	
`delete`()	Delete this node and the children
`get_mask`()	
`get_moments`()	
`get_shape`(ind)	
`has_plotter`()	Return True if the node has a plotter
`initialize_from_parameters`(*args)	
`initialize_from_prior`()	
`initialize_from_random`()	

Table 5.46 – continued from previous p

initialize_from_value(x, *args)	
integrated_logpdf_from_parents(x, index)	Approximates the posterior predictive pdf int p(x|parents) q(parents) dparent
load(group)	Load the state of the node from a HDF5 file.
logpdf(X[, mask])	Compute the log probability density function Q(X) of this node.
lower_bound_contribution([gradient])	
lowerbound()	
move_plates(from_plate, to_plate)	
observe(x, *args[, mask])	Fix moments, compute f and propagate mask.
pdf(X[, mask])	Compute the probability density function of this node.
plot(**kwargs)	Plot the node distribution using the plotter of the node
random()	Draw a random sample from the distribution.
save(group)	Save the state of the node into a HDF5 file.
set_plotter(plotter)	
unobserve()	
update()	

bayespy.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|parents) q(parents) dparents in log-scale as int q(parents_i) exp(int q(**parents**_i) log p(x|parents) **dparents**_i) dparents_i.

bayespy.nodes.Mixture.load

Mixture.**load**(*group*)

Load the state of the node from a HDF5 file.

bayespy.nodes.Mixture.logpdf

Mixture.**logpdf**(*X*, *mask=True*)

Compute the log probability density function Q(X) of this node.

bayespy.nodes.Mixture.lower_bound_contribution

Mixture.**lower_bound_contribution**(*gradient=False*)

bayespy.nodes.Mixture.lowerbound

Mixture.**lowerbound**()

bayespy.nodes.Mixture.move_plates

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

bayespy.nodes.Mixture.observe

Mixture.**observe**(*x*, **args*, *mask=True*)

Fix moments, compute f and propagate mask.

bayespy.nodes.Mixture.pdf

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

bayespy.nodes.Mixture.plot

`Mixture.``plot`(***kwargs*)
    Plot the node distribution using the plotter of the node

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

bayespy.nodes.Mixture.random

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

bayespy.nodes.Mixture.save

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

    group can be the root

bayespy.nodes.Mixture.set_plotter

`Mixture.``set_plotter`(*plotter*)

bayespy.nodes.Mixture.unobserve

`Mixture.``unobserve`()

bayespy.nodes.Mixture.update

`Mixture.``update`()

Attributes

dims	
plates	

bayespy.nodes.Mixture.dims

`Mixture.``dims` = None

> **bayespy.nodes.Mixture.plates**
>
> Mixture.**plates** = None

5.1.2 Deterministic nodes

Dot(*args, **kwargs)	Node for computing inner product of several Gaussian vectors.
SumMultiply(*args[, iterator_axis])	Node for computing general products and sums of Gaussian nodes.
Gate(Z, X[, gated_plate, moments])	Deterministic gating of one node.

bayespy.nodes.Dot

bayespy.nodes.**Dot**(*args*, **kwargs*)

> Node for computing inner product of several Gaussian vectors.

> This is a simple wrapper of the much more general SumMultiply. For now, it is here for backward compatibility.

bayespy.nodes.SumMultiply

class bayespy.nodes.**SumMultiply**(*args*, *iterator_axis=None*, **kwargs*)

> Node for computing general products and sums of Gaussian nodes.

> The node is similar to *numpy.einsum*, which is a very general function for computing dot products, sums, products and other sums of products of arrays.

> For instance, the equivalent of

> np.einsum('abc,bd,ca->da', X, Y, Z)

> would be given as

> SumMultiply('abc,bd,ca->da', X, Y, Z)

> or

> SumMultiply(X, [0,1,2], Y, [1,3], Z, [2,0], [3,0])

> which is similar to the other syntax of numpy.einsum.

> This node operates similarly as numpy.einsum. However, you must use all the elements of each node, that is, an operation like np.einsum('ii->i',X) is not allowed. Thus, for each node, each axis must be given unique id. The id identifies which axes correspond to which axes between the different nodes. Also, Ellipsis ('...') is not yet supported for simplicity. It would also have some problems with constant inputs (because how to determine ndim), so let us just forget it for now.

> Each output axis must appear in the input mappings.

> The keys must refer to variable dimension axes only, not plate axes.

> The input nodes may be Gaussian-gamma (isotropic) nodes.

> The output message is Gaussian-gamma (isotropic) if any of the input nodes is Gaussian-gamma.

Notes

This operation can be extremely slow if not used wisely. For large and complex operations, it is sometimes more efficient to split the operation into multiple nodes. For instance, the example above could probably be computed faster by

```
XZ = SumMultiply(X, [0,1,2], Z, [2,0], [0,1])
F = SumMultiply(XZ, [0,1], Y, [1,2], [2,0])
```

because the third axis ('c') could be summed out already in the first operation. This same effect applies also to numpy.einsum in general.

Examples

Sum over the rows: 'ij->j'

Inner product of three vectors: 'i,i,i'

Matrix-vector product: 'ij,j->i'

Matrix-matrix product: 'ik,kj->ij'

Outer product: 'i,j->ij'

Vector-matrix-vector product: 'i,ij,j'

Methods

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

bayespy.nodes.SumMultiply.add_plate_axis

SumMultiply.**add_plate_axis**(*to_plate*)

bayespy.nodes.SumMultiply.delete

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

bayespy.nodes.SumMultiply.get_mask

SumMultiply.**get_mask**()

bayespy.nodes.SumMultiply.get_moments

SumMultiply.**get_moments**()

bayespy.nodes.SumMultiply.get_parameters

SumMultiply.**get_parameters**()

bayespy.nodes.SumMultiply.get_shape

SumMultiply.**get_shape**(*ind*)

bayespy.nodes.SumMultiply.has_plotter

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

bayespy.nodes.SumMultiply.lower_bound_contribution

SumMultiply.**lower_bound_contribution**(*gradient=False*)

bayespy.nodes.SumMultiply.move_plates

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

bayespy.nodes.SumMultiply.plot

SumMultiply.**plot**(***kwargs*)
    Plot the node distribution using the plotter of the node

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

bayespy.nodes.SumMultiply.set_plotter

SumMultiply.**set_plotter**(*plotter*)

Attributes

plotes

bayespy.nodes.SumMultiply.plates

SumMultiply.**plates** = None

bayespy.nodes.Gate

class bayespy.nodes.**Gate**(*Z*, *X*, *gated_plate=-1*, *moments=None*, ***kwargs*)

Deterministic gating of one node.

Gating is performed over one plate axis.

Note: You should not use gating for several variables which parents of a same node if the gates use the same gate assignments. In such case, the results will be wrong. The reason is a general one: A stochastic node may not be a parent of another node via several paths unless at most one path has no other stochastic nodes between them.

Methods

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

bayespy.nodes.Gate.add_plate_axis

Gate.**add_plate_axis**(*to_plate*)

bayespy.nodes.Gate.delete

Gate.**delete**()

Delete this node and the children

bayespy.nodes.Gate.get_mask

Gate.**get_mask**()

bayespy.nodes.Gate.get_moments

Gate.**get_moments**()

bayespy.nodes.Gate.get_shape

Gate.**get_shape**(*ind*)

bayespy.nodes.Gate.has_plotter

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

bayespy.nodes.Gate.lower_bound_contribution

Gate.**lower_bound_contribution**(*gradient=False*)

bayespy.nodes.Gate.move_plates

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

bayespy.nodes.Gate.plot

Gate.**plot**(***kwargs*)
> Plot the node distribution using the plotter of the node

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

bayespy.nodes.Gate.set_plotter

Gate.**set_plotter**(*plotter*)

Attributes

plbates

bayespy.nodes.Gate.plates

Gate.**plates** = None

5.1.3 Base nodes (for developers)

These nodes should be interesting only for developers.

node.Node(*parents, **kwargs)	Base class for all nodes.
stochastic.Stochastic(*args[, initialize, dims])	Base class for nodes that are stochastic.
deterministic.Deterministic(*args, **kwargs)	Base class for deterministic nodes.

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

add_plate_axis(to_plate)	
delete()	Delete this node and the children
get_mask()	
get_moments()	
get_shape(ind)	
has_plotter()	Return True if the node has a plotter
move_plates(from_plate, to_plate)	
plot(**kwargs)	Plot the node distribution using the plotter of the node
set_plotter(plotter)	

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**

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

[add_plate_axis](to_plate)	
[delete]()	Delete this node and the children
[get_mask]()	
[get_moments]()	
[get_shape](ind)	
[has_plotter]()	Return True if the node has a plotter
[load](group)	Load the state of the node from a HDF5 file.
[lowerbound]()	
[move_plates](from_plate, to_plate)	
[observe](x[, mask])	Fix moments, compute f and propagate mask.
[plot](**kwargs)	Plot the node distribution using the plotter of the node
[random]()	Draw a random sample from the distribution.
[save](group)	Save the state of the node into a HDF5 file.
[set_plotter](plotter)	
[unobserve]()	
[update]()	

bayespy.inference.vmp.nodes.stochastic.Stochastic.add_plate_axis

Stochastic.**add_plate_axis**(*to_plate*)

bayespy.inference.vmp.nodes.stochastic.Stochastic.delete

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

bayespy.inference.vmp.nodes.stochastic.Stochastic.get_mask

Stochastic.**get_mask**()

bayespy.inference.vmp.nodes.stochastic.Stochastic.get_moments

Stochastic.**get_moments**()

bayespy.inference.vmp.nodes.stochastic.Stochastic.get_shape

Stochastic.**get_shape**(*ind*)

bayespy.inference.vmp.nodes.stochastic.Stochastic.has_plotter

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

bayespy.inference.vmp.nodes.stochastic.Stochastic.load

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

bayespy.inference.vmp.nodes.stochastic.Stochastic.lowerbound

Stochastic.**lowerbound**()

bayespy.inference.vmp.nodes.stochastic.Stochastic.move_plates

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

bayespy.inference.vmp.nodes.stochastic.Stochastic.observe

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

bayespy.inference.vmp.nodes.stochastic.Stochastic.plot

Stochastic.**plot**(***kwargs*)
    Plot the node distribution using the plotter of the node

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

bayespy.inference.vmp.nodes.stochastic.Stochastic.random

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

bayespy.inference.vmp.nodes.stochastic.Stochastic.save

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

    group can be the root

bayespy.inference.vmp.nodes.stochastic.Stochastic.set_plotter

Stochastic.**set_plotter**(*plotter*)

bayespy.inference.vmp.nodes.stochastic.Stochastic.unobserve

Stochastic.**unobserve**()

bayespy.inference.vmp.nodes.stochastic.Stochastic.update

Stochastic.**update**()

Attributes

plates

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

Stochastic.**plates** = None

bayespy.inference.vmp.nodes.deterministic.Deterministic

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

Base class for deterministic nodes.

Sub-classes must implement: 1. For implementing the deterministic function:

_compute_moments(self, *u)

2. One of the following options: a) Simple methods:

_compute_message_to_parent(self, index, m, *u) not? _compute_mask_to_parent(self, index, mask)

(a) More control with: _compute_message_and_mask_to_parent(self, index, m, *u)

Sub-classes may need to re-implement: 1. If they manipulate plates:

_compute_mask_to_parent(index, mask) _plates_to_parent(self, index) _plates_from_parent(self, index)

Methods

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

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)	Node for presenting constant values.

bayespy.inference.vmp.nodes.constant.Constant

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

 Node for presenting constant values.

 The node wraps arrays into proper node type.

Methods

add_plate_axis(to_plate)	
delete()	Delete this node and the children
get_mask()	
get_moments()	
get_shape(ind)	
has_plotter()	Return True if the node has a plotter
move_plates(from_plate, to_plate)	
plot(**kwargs)	Plot the node distribution using the plotter of the node
set_plotter(plotter)	

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

5.2 Plotting

5.2.1 Functions

5.2.2 Plotters

DEVELOPER API

Moments, Distributions, utils.misc, utils.random etc ...?

INDICES AND TABLES

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

[R1] J. Luttinen, T. Raiko, A. Ilin, "Linear State-Space Model with Time-Varying Dynamics," submitted to ECML 2014.

I

M

V

W