BayesPy Documentation

Release 0.1

Jaakko Luttinen

May 12, 2014

CHAPTER

ONE

INTRODUCTION

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

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

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 The version in PyPI is quite outdated.

1.1 Project information

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

1.2 Similar projects

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

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

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

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

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

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

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

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

CHAPTER

TWO

USER GUIDE

2.1 Installation

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

2.1.1 Installing 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
```

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 pyzwq
```

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

```
sudo aptitude install pandoc
```

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

```
make notebooks
```

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

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

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

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

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

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

2.2 Quick start guide

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

2.2.1 Simple example

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

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

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

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

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

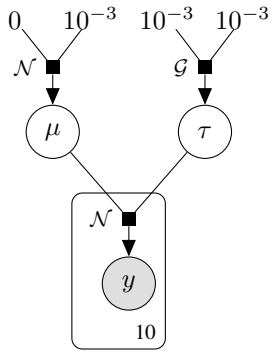


Figure 2.1: Directed factor graph of the example model.

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

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

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

Next we want to estimate the posterior distribution. In principle, we could use different inference engines (e.g., MCMC or EP) but currently only variational Bayesian (VB) engine is implemented. The engine is initialized by giving the nodes and the inference algorithm can be run as long as wanted (20 iterations in this case):

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

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

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

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

Todo

Add an example of visualizing the results.

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

2.2.2 Constructing the model

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

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

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

Name and plates

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

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

would be created as

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

It is also possible that the parents have plates. The validity of the plates between a child and a parent is checked by comparing the plates plate-wise from the trailing plates and working the way forward. A plate of the child is compatible with a plate of the parent if either of the following conditions is met:

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

Table below shows an example of compatible plates for a child and two parent nodes.

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

For instance, a model

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

could be created as

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

Multi-dimensional nodes

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

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

Deterministic and constant nodes

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

2.2.3 Providing the data

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

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

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

Missing values

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

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

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

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

2.2.4 Performing inference

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

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

Choosing the inference method

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

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

Initializing the inference

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

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

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

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

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

Running the inference algorithm

The approximation methods are based on iterative algorithms, which can be run using `update` method. By default, it takes one iteration step updating all nodes once. However, you can give as arguments the nodes you want to update and they are updated in the given order. It is possible to give same nodes several times, for instance:

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

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

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

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

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

2.2.5 Examining the results

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

Todo

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

2.3 Constructing the model

2.4 Performing inference

2.5 Examining results

CHAPTER

THREE

EXAMPLES

3.1 Gaussian mixture model

Do some stuff:

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

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

Nice!

3.2 Bernoulli mixture model

blaa blaa blaa

```
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 "

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)
```

3.3 Discrete hidden Markov model

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

3.3.1 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:

```
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:

```
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.

3.3.2 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.

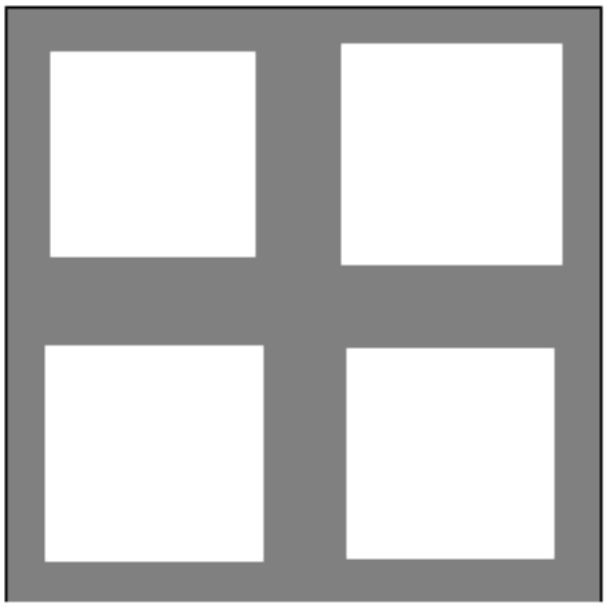
```
P.initialize_from_random()
Q.update(Z, A, a0, P, repeat=20)
```

```
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.

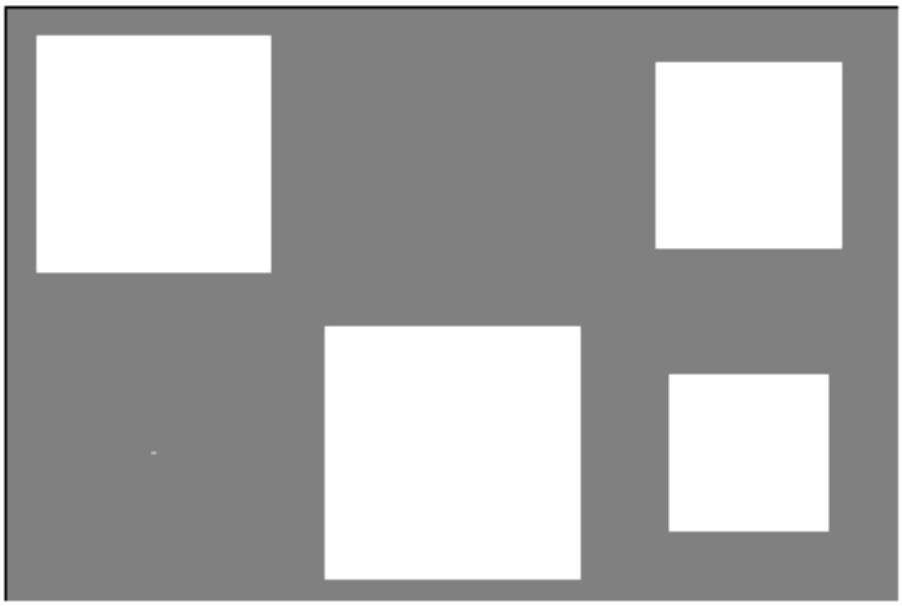
Plot the estimated state transition probabilities:

```
# 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.

3.4 Hidden Markov model

blaa blaa

3.5 Principal component analysis

Yeah.

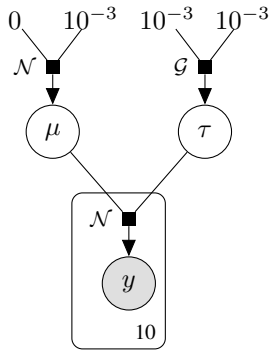


Figure 3.1: Directed factor graph of the example model.

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

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

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

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

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

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)
```

3.6.1 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:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Norm of function gradient:  5782.51
Gradient relative error = 5.61705e-05 and absolute error = 0.324807
Iteration 29: loglike=-7.283818e+03 (0.210 seconds)
Norm of numerical gradient: 9067.22
Norm of function gradient:  9067.21
Gradient relative error = 2.4973e-05 and absolute error = 0.226435
Iteration 30: loglike=-7.283121e+03 (0.200 seconds)
Norm of numerical gradient: 9594.54

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

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

3.7 Latent Dirichlet allocation

blaa blaa blaa..

CHAPTER

FOUR

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

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

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

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

4.1.1 Univariate normal distribution

4.1.2 Multivariate normal distribution

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

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

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

$$\begin{aligned}
\mathbf{u}(\mathbf{x}) &= \begin{bmatrix} \mathbf{x} \\ \mathbf{x}\mathbf{x}^{\mathrm{T}} \end{bmatrix} \\
\boldsymbol{\phi}(\boldsymbol{\mu}, \boldsymbol{\Lambda}) &= \begin{bmatrix} \boldsymbol{\Lambda}\boldsymbol{\mu} \\ -\frac{1}{2}\boldsymbol{\Lambda} \end{bmatrix} \\
\boldsymbol{\phi}_{\boldsymbol{\mu}}(\mathbf{x}, \boldsymbol{\Lambda}) &= \begin{bmatrix} \boldsymbol{\Lambda}\mathbf{x} \\ -\frac{1}{2}\boldsymbol{\Lambda} \end{bmatrix} \\
\boldsymbol{\phi}_{\boldsymbol{\Lambda}}(\mathbf{x}, \boldsymbol{\mu}) &= \begin{bmatrix} -\frac{1}{2}\mathbf{x}\mathbf{x}^{\mathrm{T}} + \frac{1}{2}\mathbf{x}\boldsymbol{\mu}^{\mathrm{T}} + \frac{1}{2}\boldsymbol{\mu}\mathbf{x}^{\mathrm{T}} - \frac{1}{2}\boldsymbol{\mu}\boldsymbol{\mu}^{\mathrm{T}} \\ \frac{1}{2} \end{bmatrix} \\
g(\boldsymbol{\mu}, \boldsymbol{\Lambda}) &= -\frac{1}{2}\operatorname{tr}(\boldsymbol{\mu}\boldsymbol{\mu}^{\mathrm{T}}\boldsymbol{\Lambda}) + \frac{1}{2}\log|\boldsymbol{\Lambda}| \\
g_{\boldsymbol{\phi}}(\boldsymbol{\phi}) &= \frac{1}{4}\boldsymbol{\phi}_1^{\mathrm{T}}\boldsymbol{\phi}_2^{-1}\boldsymbol{\phi}_1 + \frac{1}{2}\log|-2\boldsymbol{\phi}_2| \\
f(\mathbf{x}) &= -\frac{D}{2}\log(2\pi) \\
\overline{\mathbf{u}}(\boldsymbol{\phi}) &= \begin{bmatrix} -\frac{1}{2}\boldsymbol{\phi}_2^{-1}\boldsymbol{\phi}_1 \\ \frac{1}{4}\boldsymbol{\phi}_2^{-1}\boldsymbol{\phi}_1\boldsymbol{\phi}_1^{\mathrm{T}}\boldsymbol{\phi}_2^{-1} - \frac{1}{2}\boldsymbol{\phi}_2^{-1} \end{bmatrix}
\end{aligned}$$

4.1.3 Gamma distribution

4.1.4 Wishart distribution

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

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

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

$$
\begin{aligned}
\mathbf{u}(\boldsymbol{\Lambda}) &= \begin{bmatrix} \boldsymbol{\Lambda} \\ \log|\boldsymbol{\Lambda}| \end{bmatrix} \\
\boldsymbol{\phi}(n, \mathbf{V}) &= \begin{bmatrix} -\frac{1}{2}\mathbf{V} \\ \frac{1}{2}n \end{bmatrix} \\
\boldsymbol{\phi}_n(\boldsymbol{\Lambda}, \mathbf{V}) &= \begin{bmatrix} \frac{1}{2}\log|\boldsymbol{\Lambda}| + \frac{1}{2}\log|\mathbf{V}| + \frac{D}{2}\log 2 \\ -1 \end{bmatrix} \\
\boldsymbol{\phi}_{\mathbf{V}}(\boldsymbol{\Lambda}, n) &= \begin{bmatrix} -\frac{1}{2}\boldsymbol{\Lambda} \\ \frac{1}{2}n \end{bmatrix} \\
g(n, \mathbf{V}) &= \frac{n}{2}\log|\mathbf{V}| - \frac{nD}{2}\log 2 - \log\Gamma_D\left(\frac{n}{2}\right) \\
g_{\boldsymbol{\phi}}(\boldsymbol{\phi}) &= \boldsymbol{\phi}_2 \log|-\boldsymbol{\phi}_1| - \log\Gamma_D(\boldsymbol{\phi}_2) \\
f(\boldsymbol{\Lambda}) &= -\frac{D+1}{2}\log|\boldsymbol{\Lambda}| \\
\overline{\mathbf{u}}(\boldsymbol{\phi}) &= \begin{bmatrix} -\boldsymbol{\phi}_2\boldsymbol{\phi}_1^{-1} \\ -\log|-\boldsymbol{\phi}_1| + \psi_D(\boldsymbol{\phi}_2) \end{bmatrix}
\end{aligned}
$$

4.1.5 Normal-Gamma distribution

4.1.6 Gaussian-Wishart distribution

4.1.7 Gaussian-Gamma distribution

4.1.8 Mixture distribution

$$
\mathbf{x} \sim \mathrm{Mix}_{\mathcal{D}}\left(\lambda, \left\{\boldsymbol{\Theta}_1^{(n)}, \ldots, \boldsymbol{\Theta}_K^{(n)}\right\}_{n=1}^{N}\right)
$$

$$
\lambda \in \{1, \ldots, N\}, \quad \mathcal{D} \text{ is an exp.fam. distribution}, \quad \boldsymbol{\Theta}_k^{(n)} \text{ are parameters of } \mathcal{D}
$$

$$
\begin{aligned}
\log \mathrm{Mix}_{\mathcal{D}}\left(\mathbf{x} \middle| \lambda, \left\{\boldsymbol{\Theta}_1^{(n)}, \ldots, \boldsymbol{\Theta}_K^{(n)}\right\}_{n=1}^{N}\right) &= \sum_{n=1}^{N}[\lambda = n]\mathbf{u}_{\mathcal{D}}(\mathbf{x})^{\mathrm{T}}\boldsymbol{\phi}_{\mathcal{D}}\left(\boldsymbol{\Theta}_1^{(n)}, \ldots, \boldsymbol{\Theta}_K^{(n)}\right) \\
&\quad + \sum_{n=1}^{N}[\lambda = n]g_{\mathcal{D}}\left(\boldsymbol{\Theta}_1^{(n)}, \ldots, \boldsymbol{\Theta}_K^{(n)}\right) + f_{\mathcal{D}}(\mathbf{x})
\end{aligned}
$$

$$
\begin{aligned}
\mathbf{u}(\mathbf{x}) &= \mathbf{u}_{\mathcal{D}}(\mathbf{x}) \\
\boldsymbol{\phi}\left(\lambda, \left\{\boldsymbol{\Theta}_1^{(n)}, \ldots, \boldsymbol{\Theta}_K^{(n)}\right\}_{n=1}^{N}\right) &= \sum_{n=1}^{N} [\lambda = n] \boldsymbol{\phi}_{\mathcal{D}}\left(\boldsymbol{\Theta}_1^{(n)}, \ldots, \boldsymbol{\Theta}_K^{(n)}\right) \\
\boldsymbol{\phi}_{\lambda}\left(\mathbf{x}, \left\{\boldsymbol{\Theta}_1^{(n)}, \ldots, \boldsymbol{\Theta}_K^{(n)}\right\}_{n=1}^{N}\right) &= \begin{bmatrix} \mathbf{u}_{\mathcal{D}}(\mathbf{x})^{\mathrm{T}} \boldsymbol{\phi}_{\mathcal{D}}\left(\boldsymbol{\Theta}_1^{(1)}, \ldots, \boldsymbol{\Theta}_K^{(1)}\right) + g_{\mathcal{D}}\left(\boldsymbol{\Theta}_1^{(1)}, \ldots, \boldsymbol{\Theta}_K^{(1)}\right) \\ \vdots \\ \mathbf{u}_{\mathcal{D}}(\mathbf{x})^{\mathrm{T}} \boldsymbol{\phi}_{\mathcal{D}}\left(\boldsymbol{\Theta}_1^{(N)}, \ldots, \boldsymbol{\Theta}_K^{(N)}\right) + g_{\mathcal{D}}\left(\boldsymbol{\Theta}_1^{(N)}, \ldots, \boldsymbol{\Theta}_K^{(N)}\right) \end{bmatrix} \\
\boldsymbol{\phi}_{\boldsymbol{\Theta}_l^{(m)}}\left(\mathbf{x}, \lambda, \left\{\boldsymbol{\Theta}_1^{(n)}, \ldots, \boldsymbol{\Theta}_K^{(n)}\right\}_{n=1}^{N} \setminus \left\{\boldsymbol{\Theta}_l^{(m)}\right\}\right) &= [\lambda = m] \boldsymbol{\phi}_{\mathcal{D} \to \boldsymbol{\Theta}_l}\left(\mathbf{x}, \left\{\boldsymbol{\Theta}_k^{(m)}\right\}_{k \neq l}\right) \\
g\left(\lambda, \left\{\boldsymbol{\Theta}_1^{(n)}, \ldots, \boldsymbol{\Theta}_K^{(n)}\right\}_{n=1}^{N}\right) &= \sum_{n=1}^{N} [\lambda = n] g_{\mathcal{D}}\left(\boldsymbol{\Theta}_1^{(n)}, \ldots, \boldsymbol{\Theta}_K^{(n)}\right) \\
g(\boldsymbol{\phi}) &= g_{\mathcal{D}}(\boldsymbol{\phi}) \\
f(\mathbf{x}) &= f_{\mathcal{D}}(\mathbf{x}) \\
\overline{\mathbf{u}}(\boldsymbol{\phi}) &= \overline{\mathbf{u}}_{\mathcal{D}}(\boldsymbol{\phi})
\end{aligned}
$$

4.2 Implementing nodes

CHAPTER

FIVE

API

5.1 Stochastic nodes

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

Random variable nodes

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

5.1.1 bayespy.nodes.Gaussian

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

VMP node for Gaussian variable.

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

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

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

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

Plates!

Parent nodes? Child nodes?

See also:

`Wishart`

Notes

Message passing equations:

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

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

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

$$\begin{aligned}
\mathbf{u}(\mathbf{x}) &= \begin{bmatrix} \mathbf{x} \\ \mathbf{x}\mathbf{x}^{\mathrm{T}} \end{bmatrix} \\
\boldsymbol{\phi}(\boldsymbol{\mu}, \boldsymbol{\Lambda}) &= \begin{bmatrix} \boldsymbol{\Lambda}\boldsymbol{\mu} \\ -\frac{1}{2}\boldsymbol{\Lambda} \end{bmatrix} \\
\boldsymbol{\phi}_{\boldsymbol{\mu}}(\mathbf{x}, \boldsymbol{\Lambda}) &= \begin{bmatrix} \boldsymbol{\Lambda}\mathbf{x} \\ -\frac{1}{2}\boldsymbol{\Lambda} \end{bmatrix} \\
\boldsymbol{\phi}_{\boldsymbol{\Lambda}}(\mathbf{x}, \boldsymbol{\mu}) &= \begin{bmatrix} -\frac{1}{2}\mathbf{x}\mathbf{x}^{\mathrm{T}} + \frac{1}{2}\mathbf{x}\boldsymbol{\mu}^{\mathrm{T}} + \frac{1}{2}\boldsymbol{\mu}\mathbf{x}^{\mathrm{T}} - \frac{1}{2}\boldsymbol{\mu}\boldsymbol{\mu}^{\mathrm{T}} \\ \frac{1}{2} \end{bmatrix} \\
g(\boldsymbol{\mu}, \boldsymbol{\Lambda}) &= -\frac{1}{2}\operatorname{tr}(\boldsymbol{\mu}\boldsymbol{\mu}^{\mathrm{T}}\boldsymbol{\Lambda}) + \frac{1}{2}\log|\boldsymbol{\Lambda}| \\
g_{\boldsymbol{\phi}}(\boldsymbol{\phi}) &= \frac{1}{4}\boldsymbol{\phi}_1^{\mathrm{T}}\boldsymbol{\phi}_2^{-1}\boldsymbol{\phi}_1 + \frac{1}{2}\log|-2\boldsymbol{\phi}_2| \\
f(\mathbf{x}) &= -\frac{D}{2}\log(2\pi) \\
\overline{\mathbf{u}}(\boldsymbol{\phi}) &= \begin{bmatrix} -\frac{1}{2}\boldsymbol{\phi}_2^{-1}\boldsymbol{\phi}_1 \\ \frac{1}{4}\boldsymbol{\phi}_2^{-1}\boldsymbol{\phi}_1\boldsymbol{\phi}_1^{\mathrm{T}}\boldsymbol{\phi}_2^{-1} - \frac{1}{2}\boldsymbol{\phi}_2^{-1} \end{bmatrix}
\end{aligned}$$

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, 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**

5.1.2 bayespy.nodes.Gamma

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

Node for gamma random variables.

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

5.1.3 bayespy.nodes.Wishart

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

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

Continued on next page

Table 5.6 – continued from previous page

show()
unobserve()
update()

bayespy.nodes.Wishart.add_plate_axis

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

bayespy.nodes.Wishart.delete

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

bayespy.nodes.Wishart.get_mask

Wishart.**get_mask**()

bayespy.nodes.Wishart.get_moments

Wishart.**get_moments**()

bayespy.nodes.Wishart.get_shape

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

bayespy.nodes.Wishart.has_plotter

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

bayespy.nodes.Wishart.initialize_from_parameters

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

bayespy.nodes.Wishart.initialize_from_prior

Wishart.**initialize_from_prior**()

bayespy.nodes.Wishart.initialize_from_random

Wishart.**initialize_from_random**()

bayespy.nodes.Wishart.initialize_from_value

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

bayespy.nodes.Wishart.load

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

bayespy.nodes.Wishart.logpdf

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

bayespy.nodes.Wishart.lower_bound_contribution

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

bayespy.nodes.Wishart.lowerbound

Wishart.**lowerbound**()

bayespy.nodes.Wishart.move_plates

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

bayespy.nodes.Wishart.observe

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

bayespy.nodes.Wishart.pdf

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

bayespy.nodes.Wishart.plot

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

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

bayespy.nodes.Wishart.save

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

group can be the root

bayespy.nodes.Wishart.set_plotter

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

bayespy.nodes.Wishart.show

Wishart.**show**()

bayespy.nodes.Wishart.unobserve

Wishart.**unobserve**()

bayespy.nodes.Wishart.update

Wishart.**update**()

Attributes

dims
plates

bayespy.nodes.Wishart.dims

Wishart.**dims** = None

bayespy.nodes.Wishart.plates

Wishart.**plates** = None

5.1.4 bayespy.nodes.Dirichlet

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

Node for Dirichlet random variables.

Methods

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()	

Continued on next page

Table 5.8 – continued from previous page

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.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**

5.1.5 bayespy.nodes.GaussianMarkovChain

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

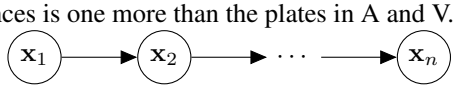
VMP node for Gaussian Markov chain.

Parents are: *mu* is the mean of x0 (Gaussian) *Lambda* is the precision of x0 (Wishart) *A* is the dynamic matrix (Gaussian) *v* is the diagonal precision of the innovation (Gamma) An additional dummy parent is created: 'N' is the number of time instances

Output is Gaussian variables.

Time dimension is over the last plate.

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



See also:

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

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()	
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**()

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**

Categorical(*args, **kwargs)	Node for categorical random variables.
Mixture(*args, **kwargs)	

5.1.6 bayespy.nodes.Categorical

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

5.1.7 bayespy.nodes.Mixture

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

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)	
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
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.save

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

group can be the root

bayespy.nodes.Mixture.set_plotter

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

bayespy.nodes.Mixture.unobserve

Mixture.**unobserve**()

bayespy.nodes.Mixture.update

Mixture.**update**()

Attributes

dims
plates

bayespy.nodes.Mixture.dims

Mixture.**dims** = None

bayespy.nodes.Mixture.plates

Mixture.**plates** = None

5.2 Deterministic nodes

Dot(*args, **kwargs)	Node for computing inner product of several Gaussian vectors.

5.2.1 bayespy.nodes.Dot

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

Node for computing inner product of several Gaussian vectors.

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

5.3 Base nodes

These nodes should be interesting only for developers.

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 nodes that are deterministic.

5.3.1 bayespy.inference.vmp.nodes.node.Node

class bayespy.inference.vmp.nodes.node.**Node**(**parents*, ***kwargs*)

Base class for all nodes.

mask dims plates parents children name

Sub-classes must implement: 1. For computing the message to children:

get_moments(self):

2.For computing the message to parents: _get_message_and_mask_to_parent(self, index)

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

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

Methods

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

5.3.2 bayespy.inference.vmp.nodes.stochastic.Stochastic

class bayespy.inference.vmp.nodes.stochastic.**Stochastic**(**args*, *initialize=True*, *dims=None*, ***kwargs*)

Base class for nodes that are stochastic.

u observed

Sub-classes must implement: _compute_message_to_parent(parent, index, u_self, *u_parents) _update_distribution_and_lowerbound(self, m, *u) lowerbound(self) _compute_dims initialize_from_prior()

If you want to be able to observe the variable: _compute_fixed_moments_and_f

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

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

Methods

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

Continued on next page

Table 5.21 – continued from previous page

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
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.save

Stochastic.**save**(*group*)

Save the state of the node into a HDF5 file.

group can be the root

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

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

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

Stochastic.**unobserve**()

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

Stochastic.**update**()

Attributes

plates

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

Stochastic.**plates** = **None**

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

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

Base class for nodes that are deterministic.

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

_compute_moments(self, *u)

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

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

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

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

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

Methods

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)

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

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

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**

CHAPTER

SIX

INDICES AND TABLES

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

A

C

D

G

H

I

L

M

N

O

P

R

S

U

W