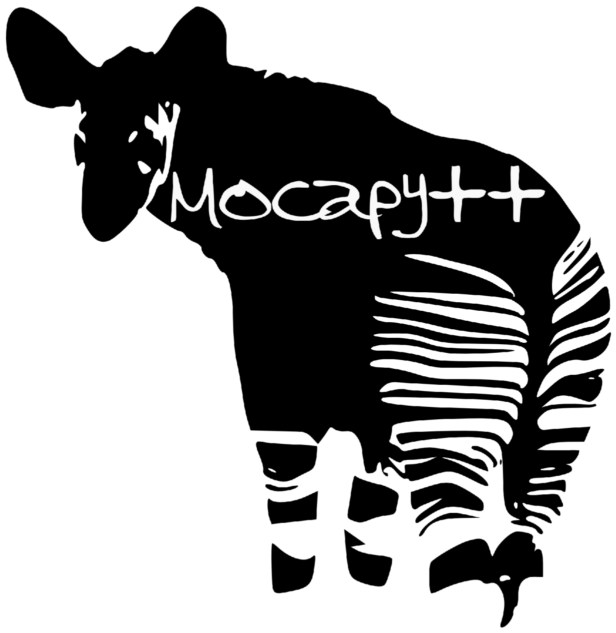

Version 1.05
January 2010

Martin Paluszewski, Jes Frellsen and Thomas Hamelryck
Bioinformatics Centre
Department of Biology
University of Copenhagen
Ole Maaloes Vej 5
2200 Copenhagen N
Denmark

E-mail: palu@binf.ku.dk, frellsen@binf.ku.dk, thamelry@binf.ku.dk

Contents

Chapter 1

What is Mocapy++?

1.1 Introduction

Mocapy++ is a freely available toolkit for parameter learning and inference in *dynamic Bayesian networks* (DBNs) [11], using *Markov chain Monte Carlo* (MCMC) methods [13]. We routinely use Mocapy++ for datasets that contain several hundred thousands of observations. Mocapy++ is a general purpose toolkit, but it contains quite some functionality that is specific to the field of bioinformatics, and in particular structural bioinformatics.

The name 'Mocapy' stands for *Markov Chain **Mo**nte **Ca**rlo and **Py**thon*. These were the key ingredients in the original implementation of Mocapy by T. Hamelryck dating back to 2004. The architecture of the DBN can be fully specified by the user, and various forms of discrete and real valued data are supported. Recently, Mocapy was for example used for constructing probabilistic models of the 3D structure of proteins [15, 5] and RNA [10]. Today, Mocapy has been re-implemented in C++, but the name is kept for historical reasons.

1.2 Notation

We'll use capital letters to refer to nodes, and small letters for node values. For example:

- $P(X)$: the probability distribution over all values that the node X can adopt.
- $P(X = x)$: the probability that node X adopts value x
- $P(x)$: shorthand for the above

The parents of a node X are denoted as Pa(X). The number of values that a node can adopt is denoted as Size(X). The children of X are Ch(X). In a dynamic Bayesian network, node n at position t is $X_{n,t}$.

1.3 Dynamic Bayesian networks

So what is a Bayesian network? A Bayesian network (BN) [11, 16] is a directed acyclic graph (DAG) in which the nodes represent random variables and the edges encode the conditional independencies between the variables[1]. A BN specifies the joint probability

[1]Formally, the absence of an edge between two nodes A, B guarantees that there is a set of nodes $\mathbf{Z}$ that renders these two nodes conditionally independent:

$$P(A, B \mid \mathbf{Z}) = P(A \mid \mathbf{Z})P(B \mid \mathbf{Z})$$

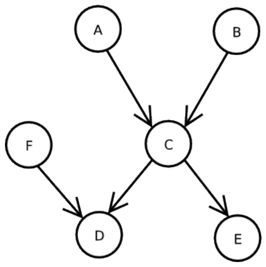

Figure 1.1: A very simple BN with 6 nodes and 5 edges.

distribution over its variables. The graph structure is a helpful tool for designing the probabilistic model and also makes it possible to develop efficient algorithms for learning and inference. The joint probability distribution of all the variables in a BN is given by:

$$P(X_1, \ldots, X_n) = \prod_{i=0}^{n-1} P(X_i \mid \mathrm{Pa}(X_i))$$

where the product runs over all n nodes $X_0, \ldots, X_{n-1}$, and $\mathrm{Pa}(X_i)$ denotes the parents of node X_i.

For example, consider the simple BN shown in Fig. 1.1. Nodes A, B are *parent nodes* of node C, and node C is a *child node* of nodes A, B. The probability of observing a certain set of values a, b, c, d, e, f for nodes A, B, C, D, E, F is given by:

$$P(a, b, c, d, e, f) = P(c \mid a, b)P(d \mid c, f)P(e \mid c)P(a)P(b)P(f)$$

A *dynamic Bayesian network* (DBN) is a BN that represents sequential data (for a good overview, see [11, 22]). The term 'dynamic' refers to the fact that a DBN is often used to model time sequences. In that respect, *sequential Bayesian network* would actually be a better name, since DBNs are also used to model sequences in which time does not play a role (for example, amino acid or DNA sequences).

The well-known *hidden Markov model* (HMM, [25, 9]) can be considered as the simplest possible DBN. Variants of the classic HMM such as the *factorial HMM* [12] can be treated as a DBN as well. In general, the DBN provides a rich framework in which many HMM-like and other probabilistic models can be treated in a uniform way. A DBN toolkit such as Mocapy++ allows the user to concentrate on the model itself, without having to implement customized inference and learning algorithms.

Another advantage of using the DBN paradigm is that it can lead to a drastic reduction in the number of parameters as compared to an HMM representation. Consider a sequence of observations. Each position in the sequence (called a sequence *slice*) is characterized by a set of random variables. Each slice in the sequence can be represented by an ordinary BN, which is said to be duplicated along the sequence positions. The sequential dependencies are then represented by edges between the consecutive DBNs. Hence, a DBN is defined by two components:

- A set of nodes that represent all random variables for a given position (called a *slice*) in the sequence, and the edges between those nodes (the *intra edges*). This set of nodes and edges forms the BN that will be duplicated along the length of the sequence.
- A set of edges that connect nodes in two consecutive slices in the sequence (the *inter edges*).

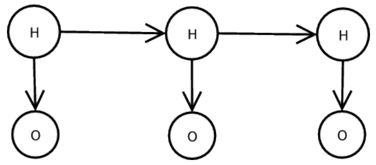

Figure 1.2: Three slices of a simple HMM in the DBN representation. Hidden nodes are labelled H, observed nodes O.

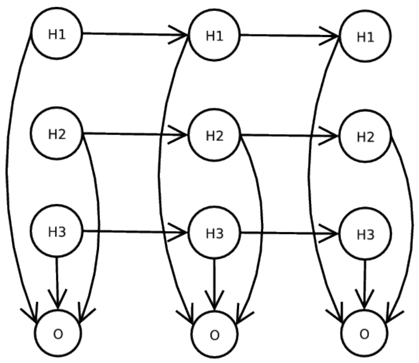

Figure 1.3: Three slices of a Factorial HMM. Hidden nodes are labelled $H1, H2, H3$, observed nodes are labelled O.

A DBN's structure can be specified conveniently by specifying all the nodes and edges that belong to two consecutive slices. This set of nodes and edges is then duplicated as necessary to model a given sequence. Some examples will probably make this more clear.

Fig. 1.2 shows the DBN representation of a HMM. In the figure, three slices are shown, corresponding to a sequence of length 3. The BN describing each slice consists simply of the hidden node H, the observed node O, and an edge that connects these two nodes. Two consecutive slices are connected by an edge between the two consecutive hidden nodes H. Note that we leave out the sequence postion index of the nodes for clarity.

Note that this diagram should not be confused with the typical diagrams that are used to represent HMMs. In a BN diagram, the nodes are random variables and the edges encode the conditional independencies. In a HMM diagram, the nodes are states and the edges represent possible transitions between states. In a DBN diagram of an ordinary HMM, an HMM state corresponds to a value of the hidden random variable, and a transition between states corresponds to a non-zero entry in the conditional probability table of the corresponding DBN.

Fig. 1.3 represents a Factorial HMM [12]. In this case, there are three hidden nodes (H_1 to H_3), which are the parents of the observed node O. The value of the observed node O depends on the values of the three hidden H nodes. The BN corresponding to each slice consists of nodes H_1, H_2, H_3 and O, and the connections between the H nodes and the O node. The edges that connect the consecutive BNs are the (H_1, H_1), (H_2, H_2) and (H_3, H_3) edges.

A DBN usually models a sequence by considering a set of observed nodes whose values depend on a set of hidden nodes (that is, nodes whose values cannot be observed). The process of assigning probabilities to the possible values of the hidden nodes given the values of the observed nodes is called *inference*. For HMMs, inference is typically done using the *forward-backward algorithm*.

Once the architecture of a DBN is defined, its parameters are typically optimized in a *learning step*, using a set of training sequences. This is typically done using the *expectation maximization* (EM) algorithm, which is called the Baum-Welch algorithm in the HMM world. The EM algorithm produces a *maximum likelihood* (ML) point estimate of the parameters. An alternative is a full Bayesian procedure, which produces a posterior probability distribution of the parameters, as opposed to a point estimate.

This is of course a very rudimentary, informal and incomplete introduction to HMMs, BNs and DBNs. Luckily, there are many excellent introductory books and articles available (for example [25, 11, 16, 3, 9, 21, 22]).

1.4 Inference in DBNs

Inference in DBNs can be done in many different ways. Deterministic algorithms such as *junction tree inference* are based on deterministic message-passing algorithms, while non-deterministic algorithms use various sampling methods to perform inference.

Mocapy++ uses a MCMC technique called Gibbs sampling to perform inference [23, 13, 6], i.e. to approximate the probability distribution over the values of the hidden nodes. Sampling methods such as Gibbs sampling are attractive for the following reasons:

- They are comparatively easy to parallelize efficiently on a cluster computer with a minimum of performance loss due to sending messages between the cluster nodes.
- They allow complicated network architectures and large datasets, especially when implemented on cluster computers.
- They are flexible: it is quite easy to try out various sampling variants (for example blocaked Gibbs sampling), and to use non-standard node types (i.e. apart from the standard discrete and Gaussian nodes).

Tests with Mocapy++ show that the combination of Gibbs sampling and DBNs is quite powerful. How does Gibbs sampling work? G cycles (or sweeps) of Gibbs sampling are performed like this:

1. The values of the hidden nodes are initialised somehow, for example by sampling according to $P(H_{n,t} \mid \text{Pa}(H_{n,t}))$, where $H_{n,t}$ is the hidden node n at sequence position t, and $\text{Pa}(H_{n,t})$ are the values of the parents of node n at position t.
2. Pick a random hidden node $H_{n,t}$ (i.e. hidden node n at sequence position t).
3. Sample a value $h_{n,t}$ for $H_{n,t}$ based on the values of the nodes in its so-called *Markov Blanket,* i.e. the values of the child and parent nodes of node $H_{n,t}$:

   $$h_{n,t} \sim P(H_{n,t} \mid \text{Pa}(H_{n,t})) \prod_{C \in \text{Ch}(H_{n,t})} P(C \mid \text{Pa}(C))$$

   The product runs over all children C of $H_{n,t}$.
4. Go to step 2, until the sequence has been sampled G times.

Let us illustrate the concept of the Markov Blanket with an example. Suppose node C has two parents A, B and two children D, E (see Fig. 1.1). In addition, node D has F as its single parent. All nodes except C are observed (with node values a, b, d, e, f). Then, a node value c for node C will be sampled from the following distribution:

$$c \sim P(C \mid a, b)P(e \mid C)P(d \mid C, f)$$

The above algorithm describes a *random sweep Gibbs sampler*, i.e. the nodes and sequence positions are traversed in a random way during one sampling cycle. Other traversals are also possible in principle. In a DBN, parents and children might be present in three different slices (for example, A might be in slice $t-1$, B, D, F in slice t and E in slice $t+1$). This makes the bookkeeping in a DBN slightly more complicated than in a BN.

Typically, the first cycles of Gibbs sampling (called the *burn in period*) are discarded, because the values of the sampled nodes need some time to reach a stationary distribution. There are unfortunately no simple rules that tell you how long the burn in period needs to be. Testing convergence on an artificial dataset for which the expected parameter values are known is an accepted way to evaluate convergence. This artificial dataset can for example be generated by generating a set of sequences through sampling the DBN or by using a standard data set.

Each time a hidden node is sampled, the sampled node value is stored. When the sampling is finished, the sampled node values can be used to approximate the probability distribution of the node values of the hidden nodes.

1.5 Parameter learning in DBNs

1.5.1 Expectation Maximization

Parameter learning of a DBN with hidden nodes is often done using the *expectation maximization* (EM) method (for a good introduction to learning in BNs, see [11, 16, 22], a very clear explanation of the EM algorithm can be found in [9]). In the E-step, the values of the hidden nodes are inferred using the current DBN parameter setting. In the subsequent M-step, the inferred values of the hidden nodes are used to update the DBN's parameters. The E- and M-step cycles are repeated until convergence (or until the user's patience is exhausted). The EM method produces a maximum likelihood point estimate of the parameters.

Let's take a look at this in a bit more detail (the following discussion is largely based on [11]). In learning using ML estimation, we want to maximize the log likelihood (LogLik) $\mathcal{L}(\theta)$ in function of the DBN's parameters θ:

$$\mathcal{L}(\theta) = \log \sum_{\mathbf{h}} P(\mathbf{h}, \mathbf{o}|\theta)$$

where the sum runs over all possible sets of hidden node values, and $\mathbf{o} = o_1, \ldots, o_n$ is the set of observed node values. In practice, this computation is typically intractable using a brute force approach. However, the problem can be simplified by making use of a lower bound $\mathcal{F}(\theta)$ of $\mathcal{L}(\theta)$. First, we introduce an arbitrary probability distribution $q(\mathbf{h})$ over $\mathbf{h}$:

$$\mathcal{L}(\theta) = \log \sum_{\mathbf{h}} q(\mathbf{h}) \frac{P(\mathbf{h}, \mathbf{o}|\theta)}{q(\mathbf{h})}$$

Because the log function is a concave function, we can apply Jensen's inequality[2] to obtain a lower bound $\mathcal{F}$ of $\mathcal{L}$. Hence:

$$\mathcal{L}(\theta) \geq \mathcal{F}(q, \theta) = \sum_{\mathbf{h}} q(\mathbf{h}) \log \frac{P(\mathbf{h}, \mathbf{o}|\theta)}{q(\mathbf{h})}$$

$$\mathcal{F}(q, \theta) = \sum_{\mathbf{h}} q(\mathbf{h}) \log P(\mathbf{h}, \mathbf{o}|\theta) - \sum_{\mathbf{h}} q(\mathbf{h}) \log q(\mathbf{h})$$

[2]For any concave function f the following is true: $E[\mathrm{f}(x)] \leq \mathrm{f}(E[x])$, where $E[\cdot]$ is the expectation according to $P(x)$. To apply the inequality in the above case, note that the summation in the $\mathcal{L}(\theta)$ expression is simply the expectation of $\frac{P(\mathbf{h},\mathbf{o}|\theta)}{q(\mathbf{h})}$ with respect to $q(\mathbf{h})$.

This expression is similar to the free energy expression in statistical mechanics, where the first term is an energy term[3] and the second term is an entropy term. In the EM procedure, $\mathcal{F}(q,\theta)$ is first optimized with respect to q keeping θ fixed at its current value θ^{curr}, and then with respect to θ using q obtained from the E-step (q^{next}).

E-step:

$$q^{\text{next}} = \text{argmax}(q)\left[\mathcal{F}(q,\theta^{\text{curr}})\right]$$

M-step:

$$\theta^{\text{next}} = \text{argmax}(\theta)\left[\mathcal{F}(q^{\text{next}},\theta)\right]$$

The maximum in the E-step arises for the following value of q:

$$q(\mathbf{h}) = P(\mathbf{h} \mid \mathbf{o}, \theta^{\text{curr}})$$

In Mocapy++, this distribution is approximated using Gibbs sampling.

The EM cycle is repeated until convergence or until the maximum number of cycles is reached. In the next section, we will explain how Mocapy++ does this in practice using MCMC EM.

1.5.2 Monte Carlo EM for DBNs

Often, inference of the hidden nodes in a DBN cannot be done (efficiently) using deterministic methods. A solution is to use a stochastic inference method, i.e. the hidden node values are inferred by sampling. Several versions of the EM algorithm exist that use some form of stochastic inference for the E-step. Mocapy++ uses Gibbs sampling to infer the hidden node values in the E-step, and supports two different types of EM.

In *Monte Carlo EM (MC-EM)* a large number of samples is used in the E-step [1]. In *stochastic EM (S-EM)* only one sample is generated for each hidden node [8, 1], and one essentially makes use of a 'completed' dataset where the hidden node values are filled in.

Mocapy++ supports both forms of EM. Tests have pointed out that S-EM is both fast and efficient if you have a lot of data (for example, we routinely use several of 100.000s slices in combination with S-EM). If data is sparse, it probably makes more sense to use MC-EM, unless you want to locate several maxima of the LogLik function by using S-EM [13].

We briefly sketch the background of the EM method with a stochastic E-step. Recall the distribution $q(\mathbf{h})$ of subsection 1.5.1:

$$q(\mathbf{h}) = P(\mathbf{h} \mid \mathbf{o}, \theta^{\text{curr}})$$

In Mocapy++, $q(\mathbf{h})$ is approximated by sampling a set of hidden node values from $P(\mathbf{h}|\mathbf{o},\theta^{\text{curr}})$ by Gibbs sampling. Each hidden node value is sampled conditional upon all the other node values by Markov blanket sampling. At the end of this E-step, the sampled hidden node values and the values of the observed nodes can be treated as a completed dataset in which all node values are observed (S-EM), or the samples can be used to approximate $q(\mathbf{h})$ (MC-EM). In the M-step, the completed dataset (S-EM) or the samples (MC-EM) are then used to update the parameters of the model.

How is this all performed in practice? One typically starts with a DBN initiated with random values for the parameters and the hidden nodes, or starting values that somehow 'make sense'. In the E-step (which is the inference step), the values of the hidden nodes are inferred using the current DBN parameter values, i.e. a set of samples is generated by Gibbs sampling. Each time a node is sampled, the *expected sufficient*

[3]The energy of a configuration $(\mathbf{h},\mathbf{o})$ is in this case defined as $E(\mathbf{h},\mathbf{o}) = \log P(\mathbf{h},\mathbf{o}|\theta)$.

statistics (ESS) for a hidden node are updated. Essentially, the ESS are set of values that describe the samples in such a way that the parameters of a node's probability distribution can be calculated from them. The ESS of all samples are then used in the M-step to update the parameters.

For example, in the case of discrete nodes (with the categorical distribution), the ESS are simply the counts of the parent/child value combinations observed in the generated samples [11]. The categorical distribution of the discrete node is characterized by a *conditional probability density* (CPD). In the case of the discrete node, the CPD is a matrix that holds the probabilities of each combination of node and parent values. For example, a discrete node of size 2 with two parents of sizes 3 and 4 respectively will have a $3 \times 4 \times 2$ matrix as its CPD. The sum of the matrix values (that is, of the probabilities of all the parent and node value combinations) needs to be 1.

During the Gibbs sampling cycle, each time a discrete node is sampled, the values of the node and of its parents are evaluated and the relevant count in the CPD matrix is incremented. In the M-step, parent/child counts are then used to update the parameters of the discrete node, which in this case just means normalizing each row of the CPD matrix so it sums to one.

Chapter 2

Installation, license and other practicalities

2.1 Implementation

Mocapy++ is implemented as a static program library in C++. The library is highly modular and new nodes (that is, probability distributions) can be added easily. Mocapy++'s design aims to be clear, but also efficient. For object serialization and special math functions the Boost C++ library is used and for linear algebra the LAPACK library [2] is used. Mocapy++ uses CMake [26] to locate packages and configure the build system. All objects are serializable, meaning that Mocapy++ can be suspended and later resumed at any state during training or sampling.

2.2 Obtaining Mocapy++

Mocapy++ and its documentation can be downloaded from SourceForge (`https://sourceforge.net/projects/mocapy/`) as a gzipped tar file containing the source code. Simply unzip and untar it in a suitable directory. The SVN repository of Mocapy++ is also located on SourceForge. It is possible for everyone to get the very latest version from this repository, but some features might, of course, be untested.

2.3 Installing Mocapy++

Installing Mocapy++ is in most cases simple. From the root of the Mocapy++ directory (the directory containing `src` and `examples` directories) type:

```
cmake .       (Remember the dot)
make
```

CMake will check if dependent packages are installed on the system and construct the necessary makefiles. The make command compiles and links the Mocapy++ library and the example programs.

The required programs/libraries are:

- GCC (version 4.1.2 or later. Should work on other C++ compilers, but this is currently untested)
- CMake (version 2.6 or later)

- Boost (version 1.36 or later)
- LAPACK (version 3.2 or later)
- GNU Fortrain (version 4.3 or later)

If CMake complains about variables that are set to NOTFOUND, it means that it cannot find one or more required packages. This happens when packages are not installed at their default location. In that case, you can help CMake by pointing to the correct packages yourself. For example, if CMake cannot find LAPACK because it is installed in a subdirectory of /usr/lib/ you can do something like:

```
cmake -DLAPACK_LIBRARY:FILEPATH="/usr/lib/atlas/liblapack.so.3" . (again remember t
```

2.4 Using the Mocapy++ Library

Source code using Mocapy++ must always include the *mocapy.h* header like this:

```
#include "mocapy.h"
```

When compiling your Mocapy++ dependent source code, the required libraries (LAPACK, Boost and Mocapy++) must also be linked with your program. The easiest way to do this, is to place your source code in the examples directory and add it to the list of programs in the CMakeLists.txt file (the PROGS variable). Then CMake will automatically generate the correct makefile for your program. Chapter 4 describes how to use Mocapy++ in more details.

2.5 License

Mocapy++ comes with the *GNU General Public License* (see section 9 for more information). Informally speaking (please refer to the license file itself for the legally binding license), that all modified or altered versions of this Mocapy++ release will be free software as well.

2.6 Other DBN packages

Is Mocapy++ the best available tool for you? That depends entirely on what you want to do, how much data you have, the available computer power, etc. As far as we know, Mocapy++ is the only publically available toolkit that can handle directional statistics, such as the bivariate von Mises and Kent probability distributions. Before you decide to use Mocapy++, you might want to take a look at the following packages for example:

- Kevin Murphy's Bayes Net Toolbox for Matlab (BNT, `http://www.cs.ubc.ca/~murphyk/Software/BNT/bnt.html`): this widely used toolkit supports HMMs and general DBNs. Unlike Mocapy++, it mainly focuses on deterministic methods like junction tree inference.
- The Graphical Models Toolkit (GMTK, `http://ssli.ee.washington.edu/~bilmes/gmtk/`): Specially focused on DBNs, but only binaries are available.

Chapter 3

Background and Features

3.1 Performance

Mocapy++ is designed to deal with large datasets. we routinely use datasets with 250.000 slices (corresponding to 1600 protein sequences), in combination with a HMM with an observed Kent node (ie. each slice consists of a discrete value and a 3D real vector). It takes less than 10h to do 250 S-EM steps (including 50 burn-in steps). Convergence was reached after about 50 EM steps (including 50 burn-in steps, i.e. after about 3h 30min).

3.2 DBN architectures

The architecture of the DBN, i.e. the number and type of the nodes and the connections between the nodes, can be specified by the user. In principle, there are no limits with respect to the number of nodes, edges between nodes or data size, provided that you have enough memory and patience. Gaussian, Kent, von Mises, Poisson and multinomial nodes need to have exactly one discrete parent node, and they need to be in the same slice as their parent. Partly observed nodes (i.e. some values are missing in the sequence for a particular node) are supported.

3.3 Node types

Currently, Mocapy++ has 7 different node types, each based on a specific probability distribution: categorical (which we will call *discrete node* here), uni- and multivariate Gaussian, von Mises (univariate on the circle, and bivariate on the torus), Poisson, multinomial and Kent (on the sphere). In the future, more node types might be added. Adding a new node simply means implementing new distribution and ESS classes as described in Chapter 6.

3.3.1 Gaussian, multinomial and discrete nodes

Categorical (for discrete, categorical data), multinomial (for vectors of counts) and Gaussian distributions are standard, so they'll not be discussed at length here. The technicalities of parameter estimation for Gaussian nodes can be found in [22]. Mocapy++ supports Gaussian distributions with unconstrained, spherical and diagonal covariance matrices.

3.3.2 Kent node

The Kent distribution [17, 19, 24, 18, 4, 15], also known as the 5-parameter Fisher-Bingham distribution, is a directional distribution, in this case on the 2D sphere[1]. It is the 2D member of a larger class of N-dimensional distributions called the Fisher-Bingham class of distributions. For this class of distributions, parameter estimation becomes difficult if the dimension becomes larger than 2 (which is the dimension of the Kent distribution). The density function of the Kent distribution is:

$$K_{\kappa,\beta,\gamma_1,\gamma_2,\gamma_3}(\mathbf{x}) = C(\kappa,\beta)\exp\left\{\kappa\mathbf{x}\cdot\gamma_1 + \beta\left[(\mathbf{x}\cdot\gamma_2)^2 - (\mathbf{x}\cdot\gamma_3)^2\right]\right\}$$

where x is a 3D unit vector that specifies a point on the 2D sphere.

The various parameters can be interpreted as follows:

- κ: a concentration parameter. The higher κ, the more concentrated the density becomes.
- β: determines how elliptical the equal probability contours of the distribution will be. The larger β (with condition $\kappa > 2\beta$), the more elliptical. If $\beta = 0$, the Kent distribution becomes the von Mises-Fisher distribution on the 2D sphere.
- γ_1: the mean vector. The three ε vectors are unit vectors on the 2D sphere.
- γ_2: the main axis of the elliptical equal probability contours.
- γ_3: the secondary axis of the elliptical equal probability contours.

The normalizing factor $C(\kappa,\beta)$ is given by:

$$C(\kappa,\beta) = \frac{\sqrt{(\kappa-2\beta)(\kappa+2\beta)}}{2\pi e^{\kappa}}$$

The advantage of the Kent distribution over the von Mises-Fisher distribution on the 2D sphere is that the equal probability contours of the density are not restricted to be circular: they can be elliptical as well. The Kent distribution is equivalent to a Gaussian distribution with unrestricted covariance. Hence, for 2D directional data the Kent distribution is richer than the corresponding von Mises-Fisher distribution, i.e. it might be more suited if the data contains non-circular clusters.

3.3.3 Bivariate von Mises node

Another directional distribution is the bivariate von Mises distribution on the torus. This distribution can be used to model bivariate angular data. We used this distribution to capture the properties of the Ramachandran plot [5].

The density function of the bivariate von Mises (cosine variant) distribution is:

$$f(\phi,\psi) = c(\kappa_1,\kappa_2,\kappa_3)\exp\left(\kappa_1\cos(\phi-\mu) + \kappa_2\cos(\psi-\nu) - \kappa_3\cos(\phi-\mu-\psi+\nu)\right)$$

where ϕ, ψ are angles in $[0, 2\pi[$. Such an angle pair defines a point on the torus.

The distribution has 5 parameters:

- μ and ν are the means for ϕ and ψ respectively.
- κ_1 and κ_2 are the concentration of ϕ and ψ respectively.
- κ_3 is related to their correlation.

This distribution is described in greater detail in Mardia et al. [20] and Boomsma et al. [5]

[1]The 2D sphere is the surface of the 3D ball.

3.4 Inference

Inference in our case means estimating the values of the hidden nodes, given a set of values for the observed nodes. Mocapy++ provides two approaches to inference: approximative methods based on Gibbs sampling and deterministic methods based on treating the DBN as a HMM.

Calculating the probability distribution over the values of the hidden nodes (i.e. the smoothing distribution) and calculating the most probably combination of the hidden node values (i.e. the Viterbi path) can be done in a deterministic way by treating the DBN as a HMM, and applying classic HMM algorithms.

Turning the DBN into a HMM can lead to intractable calculation when you have a lot of hidden nodes that can adopt a large number of values. If you have 10 discrete hidden nodes that each can adopt 4 different values, this would lead to a HMM with one hidden node that can adopt $4^{10} = 1048576$ states. In this case, one can approximate the smoothing distribution and the Viterbi path using approximative methods based on Gibbs sampling.

3.4.1 Deterministic inference

3.4.1.1 Smoothing

The task of calculating the probabilty distribution $P(H_{n,t} \mid \mathbf{o})$ for any hidden node H with index n and sequence position t is generally called *smoothing*. This task can be performed by treating the DBN as a HMM and applying the forward-backward algorithm [3, 9]. Scaling of the probabilities is done as described in [3], appendix D.

3.4.1.2 Viterbi path

The Viterbi path is the sequence of hidden node values that best explains the values of the observed nodes (i.e. that leads to the highest likelihood value). In other words it means calculating the following:

$$\mathbf{h}^v = \arg\max_{\mathbf{h}} \left[P(\mathbf{h} \mid \mathbf{o}) \right]$$

where $\mathbf{o} = o_1, \ldots, o_n$ are the values of the observed nodes and $\mathbf{h}^v = h_1^v, \ldots, h_m^v$ is the Viterbi path. The Viterbi path plays a crucial role in many Bioinformatics applications [3, 9].

The Viterbi path can be calculated by treating the DBN as a HMM and applying a dynamic programming algorithm [3, 9]. Mocapy++'s Viterbi path calculation uses log probabilities to avoid underflows.

3.4.2 Stochastic inference

3.4.2.1 Smoothing

For large DBNs, the forward-backward algorithm can become intractable. In that case one can approximate the posterior distribution by generating a large set of samples (by Gibbs sampling) and examining the values of the hidden nodes. The quality of the approximation will of course depend on the complexity of your model and the number of samples used.

3.4.2.2 Viterbi path

As is the case for the forward-backward algorithm, the deterministic calculation of the Viterbi path can become intractable for large DBNs. Again, one can use an approximative algorithm based on Gibbs sampling.

In principle, one could generate a large set of samples and simply pick the one with the highest LogLik. In practice, the number of possible hidden node combinations is vast and this approach becomes infeasible. A good solution to this problem is to generate a set of samples (say 100) and to construct a sequence with a high LogLik by tying together fragments from the samples.

The algorithm to approximate a Viterbi path using a set of sampled sequences uses a Dynamic Programming approach to combine the best hidden node values from a set of sampled sequences (see Algorithm 1). The algorithm is conceptually similar to a Viterbi path approximation algorithm [14] which uses a set of samples generated by *particle filtering*, an online stochastic inference method. The latter algorithm was applied to an HMM with continuous hidden and observed states.

The algorithm can be used in an iterative manner. In the first run, an initial Viterbi path approximation is calculated from a set of samples. In the next steps, new samples are generated but this time the Viterbi path approximation that was calculated in the previous step is added to the set of samples. In this way, one gradually improves the approximation. The procedure can be stopped when the LogLik seems to have converged. This procedure can of course get stuck in a local maximum.

3.5 Parameter learning using MC-EM and S-EM

Parameter learning is done using MC-EM or S-EM, making use of Gibbs sampling for the E-step (the inference step). When new sampling methods are added, these will also become available for the MC-EM procedure.

3.6 Priors

In Mocapy++, so far only the discrete node supports the use of a prior. First, let's consider how an entry in a CPD of a discrete node H is calculated. A CPD is a matrix filled with values of the type $P(H = n \mid \text{Pa}(H) = \mathbf{p})$, which corresponds to the chance of observing value n for node H when H's parent nodes $\text{Pa}(H)$ adopt values $\mathbf{p} = p_1, \ldots, p_s$ (where s is the number of parents). If all sequence node values are known, this is simply calculated as follows:

$$P(H = h \mid \text{Pa}(H) = \mathbf{p}) = \frac{\text{counts}(H = h, \text{Pa}(H) = \mathbf{p})}{\sum_{i=0}^{\text{Size}(H)-1} \text{counts}(H = i, \text{Pa}(H) = \mathbf{p})}$$

where $\text{counts}(H = h, \text{Pa}(H) = \mathbf{p})$ is the number of times the node value h is observed for node H together with parent values $\mathbf{p}$ in the sequences. The sum runs over all possible node values for H. For discrete nodes, a simple pseudo-count prior is supported (which corresponds to a Dirichlet prior). The CPD entries are calculated as follows:

$$P(H = h \mid \text{Pa}(H) = \mathbf{p}) = \frac{\text{counts}(H = h, \text{Pa}(H) = \mathbf{p}) + \mathbf{c}_h}{\sum_{i=0}^{\text{Size}(H)-1} [\text{counts}(H = i, \text{Pa}(H) = \mathbf{p}) + \mathbf{c}_i]}$$

where $\mathbf{c}$ is the pseudo count vector (with dimension equal to $\text{Size}(H)$).

Algorithm 1 Mocapy++'s Viterbi path approximation algorithm. $samples[m, l]$ corresponds to slice l in sample m. $P(samples[m, l]|samples[n, l-1])$ is the likelihood of all the node values in slice l, where the values for the nodes are taken from position l of sample m, and the nodes' parent values in the previous slice are taken from position $l-1$ of sample n. Sequence indices run from 0 to $L-1$. If a previous approximation of the Viterbi path is available, it can be added to the initial list of samples. The procedure will then come up with an improved variant of it or produce the same path.

```
$S$=number of samples
$L$=sequence length
$V$=number of scalars in a sequence slice
$samples$=sequence ($S \times L \times V$ matrix)
$score$=matrix($S \times L$)
$path$=matrix($S \times L$)
# Start of dynamic programming
# 1. Initialization
for $m = 0$ to $S-1$:
    $score[m, 0]$=log P($samples[m, 0]$])
# 2. Recursion
for $l = 0$ to $L-1$:
    for $m = 0$ to $S-1$:
        $p$=matrix($S$)
        for $n = 0$ to $S-1$:
            $p[n]$=log P($samples[m, l]$|$samples[n, l-1]$)
        $index$=argmax($p$)
        $score[m, l]$=$score[m, l-1]$+$p[index]$
        $path[m, l]$=$index$
# 3. Termination
$index$=argmax($score[L-1]$)
$viterbi$=matrix($L \times V$)
$viterbi[L-1]$=$samples[index, L-1]$
# 4. Backtracking
for $l = L-2$ to $0$:
    $index$=$path[index, l]$
    $viterbi[l]$=$samples[index, l]$
report $viterbi$
```

3.7 Structure learning is absent!

Structure learning is absent in Mocapy++. We also have no plans to implement it, since we do not need it, at least not at the moment. Anybody who wants to add this functionality to Mocapy++ is welcome to contact us.

3.8 Python Interface

A Python interface to Mocapy++ can be made through SWIG (www.swig.org) and an example of a SWIG interface file is included in the package (pythoninterface.i). A Python application using the example interface is able to create a DBN, load parameters and sample sequences. Additional interface methods can be added to pythoninterface.i as needed. Read examples/pythoninterface.txt for information on how to compile a Python interface.

Chapter 4

Using Mocapy++

Here we describe the details of constructing the nodes, creating the DBN and using the DBN in various ways. Another way of learning to use Mocapy++ is to read the example programs in the `examples` directory. The basic idea behind the example programs is to set up a DBN with user specified parameters and generate samples from this network. Then, another DBN with initially random parameters use the samples and the EM algorithm to learn the parameters of the network generating the samples.

4.1 Preliminaries

Programs using the Mocapy++ library must be linked with not only Mocapy++, but also LAPACK and Boost. The easiest way to do this is to let CMake build the makefiles automatically. This can be done by copying your main source file to the examples directory and add the file name (without extension) to the PROGS line in the CMakeLists.txt file. Future calls of `make` will now also compile your program.

When using Mocapy++, the first step is to import the headers and preferably use the `mocapy` namespace:

```
#include "mocapy.h"
using namespace mocapy;
```

Obviously, sampling depends to a great extend on the generation of random numbers. By using a specific seed for the random number generating that Mocapy++ is using, one can make the process reproducible, if that is desirable. Mocapy++ uses the random number generator in the `Boost` library. This random number generator can be seeded to reproduce a Mocapy++ run exactly:

```
mocapy_seed(a);
```

4.2 Representation of the sequence data

Mocapy++ stores data in vectors of *multidimensional arrays* (MDArray). Each sequence corresponds to one MDArray. Another array, the `mismask` array, indicates whether the value of a certain node at a certain position in the sequence is observed or not. MOCAPY_OBSERVED in the mismask array indicates the node value is observed, while MOCAPY_HIDDEN indicates the node value is not observed. Finally, it is also possible to flag some nodes as simply absent at some points in the sequence using MOCAPY_MISSING.

This is most easily explained with a simple example. Suppose the DBN has two nodes, a 2-dim Gaussian child node and a discrete (unobserved) parent node. Suppose the sequence is 100 slices long. The following code creates suitable `data` and `mismask` arrays:

```
MDArray<double> data = MDArray<double>(vec(100, 3));
MDArray<eMISMASK> mismask = MDArray<eMISMASK>(vec(100,2));
mismask.set_wildcard(vec(ALL,0), MOCAPY_HIDDEN);
```

Columns 1 and 2 in the data array will hold the real values that are associated with the 2-dim Gaussian node, while column zero holds the integer value of the discrete node. Of course, it is your responsibility to fill the data array with meaningful values somehow. Note that the order of the nodes in the DBN definition determines the order of the values in the data array. Parents need to come first, followed by their children.

There are helper functions to fill in the values of the data and mismask arrays from files:

```
MDArray<double> data = data_loader("data.dat");
MDArray<eMISMASK> mismask = toMismask(
    data_loader("mismask.dat") );
```

The format of the data files is:

- A line contains values in a slice separated by space (or another character specified as argument to the data loader)
- A block of lines represents a sequence.
- An empty line separates two sequences.
- Lines starting with # are ignored

It is the users responsibility that the input files have the correct format since little format checking is done. An example of a file is:

```
# This line is ignored
0 5.4 0 1
0 3.1 6 1
0 4.0 1 8

0 8.0 7 5
0 3.5 4 0
0 5.7 5 5
```

Reading in the lines above results in two sequences of length three having four values in a slice. The example file `hmm_simple.cpp` shows examples of how data and mismask arrays can be loaded from files.

4.3 Creating the DBN

4.3.1 Creating the nodes

The first step in the definition of a DBN is the creation of the appropriate nodes. For example, the following code creates a discrete node with node size 4 (i.e. it can adopt values 0,1,2 or 3):

```
Node* dnode = NodeFactory::new_discrete_node(4);
```

The node size is the number of different values the node can adopt, starting from 0. Since no initial values are specified for the *conditional probability density* (CPD), they are initiated to random values (while avoiding transition probabilities that are 0). The CPD is a matrix that holds the probabilities of each combination of node and parent values. For example, a discrete node of size 2 with two parents of sizes 3 and 4 respectively will have a $3 \times 4 \times 2$ matrix as its CPD. The sum of the matrix values (i.e. of the probabilities of all the parent and node value combinations) needs to be 1.

Of course, one can initiate a discrete parent with specific CPD values. For the above described node a possible initialisation is:

```
CPD cpd(vec(3,4,2));
cpd.normalize();
Node* dnode = NodeFactory::new_discrete_node(2, "dnode_cpd",
    false, cpd);
```

The second argument of the factory command above is the name of the node. The third argument specifies how the parameters should be initialized (`false`: random, `true`: uniform). The fourth argument is the user specified CPD.

Similarly, Gaussian, Kent, multinomial, Von-Mises and Poisson nodes are created as follows:

```
Node* gnode = NodeFactory::new_gaussian_node(n);
Node* mnode = NodeFactory::new_multinomial_node(n);
Node* knode = NodeFactory::new_kent_node();
Node* vm1D = NodeFactory::new_vonmises_node();
Node* vm2D = NodeFactory::new_vonmises2d_node();
Node* pnode = NodeFactory::new_poisson_node();
```

Note that the dimensionality of the Kent, VM and Poisson nodes is fixed and can therefore not be specified as argument.

Initialization with specific values for the Gaussian node is done like this:

```
MDArray<double> means(vec(2,2));
means.set_values( vec(0.0, 0.0, 10.0, 10.0) );
MDArray<double> covs(vec(2,2));
covs.set_values( vec(1.0, 0.0, 2.0, 0.5) );
Node* gnode = NodeFactory::new_gaussian_node(2, "gnode",
    false, means, covs, FULL);
```

For the Gaussian node, there are three possible choices for the covariance matrix: full, spherical or diagonal. This is specified by the 6th parameter, which should be `'FULL'`, `'SPHE'` or `'DIAG'`.

When the Kent node is created with random parameters, the maximum values for the κ and β parameters can be specified. The default values are 5000 and 100, respectively. Keep in mind that $\kappa > 2\beta$. Note that the dimension does not need to be specified for the Kent node because the Kent distribution is a distribution on the 3D sphere (i.e. dim $= 3$). For example:

```
vector<double> kappas, betas;
vector<MDArray<double> > es;
Node* knode = NodeFactory::new_kent_node("knode", kappas,
    betas, es, 40.0, 4.0);
```

Specific values for κ, β and the $\gamma_1, \gamma_2, \gamma_3$ vectors can also be given of course.

Sometimes it is useful to fix a node's parameters during learning. This can be done by setting the `node.fixed` attribute to `true`. Setting the attribute to `false` will allow the parameters to be updated again in the following EM steps.

4.3.2 Defining the DBN architecture

After creating the necessary node objects, the next step is to specify the DBN architecture.

- Define the nodes in two consecutive slices. These nodes are arguments to the `set_slices` method.
- Specify the connections between the nodes, both inside the slices and between the two slices. This is done using the `add_intra` and `add_inter` methods of the `DBN` object.
- Finally, initialize the DBN by calling the `construct` method.

The following example is typical for a HMM:

```
// We assume that nodes h0, o0, h1 and o1 are created
DBN dbn;
vector<Node*> start_nodes = vec(h0, o0);
vector<Node*> end_nodes = vec(h1, o1);
dbn.set_slices(start_nodes, end_nodes);
dbn.add_intra("h0", "o0");
dbn.add_inter("h0", "h1");
dbn.construct();
```

The arguments to the `add_intra` and `add_inter` methods are either indices that refer to the position of the nodes in the lists of start and end nodes or their names. Note that the nodes in these two lists need to be in topological order! Parents should come first, followed by their children later in the list.

In the above HMM example, the observed nodes are *untied*, i.e. the observed node that models the observations at $l = 0$ is different from the observed node that models those at $l > 0$. It is possible to use *tied* nodes as well, by using the same observed node `ot` for all positions l:

```
DBN dbn;
vector<Node*> start_nodes = vec(h0, ot);
vector<Node*> end_nodes = vec(h1, ot);
dbn.set_slices(start_nodes, end_nodes);
dbn.add_intra("h0", "ot");
dbn.add_inter("h0", "h1");
```

Note that node `ot` is connected to both `h0` and `h1`. It is not possible to tie `h0` and `h1`, because they have different conditional probability distributions (`h0` has no parents, while `h1` has one discrete hidden parent).

4.4 Sampling a sequence from a DBN

Once a DBN is created, it is easy to sample a sequence using the `sample_sequence` function:

```
pair<Sequence, double> seq_ll = dbn.sample_sequence(100);
```

The method returns a sequence (here of length 100) and the LogLik. This method is of course ideal to create a test for the learning procedure.

Like all data (see next section), the returned sequence is a vector of MDArray. It's dimensions are `(output_size, sequence_length)`. The output size is the number of numbers that fill in the values of all nodes. For example, the output size for a 2D Gaussian node and a discrete node is $2 + 1 = 3$.

4.5 LogLik of a fully observed sequence

When all nodes are observed, calculation of the LogLik is of course trivial. In this case, the calculation can be done using a DBN object:

```
double ll=dbn.calc_ll(seq);
```

The returned LogLik is the LogLik divided by the number of sequence slices (i.e. $\mathcal{L}^{\mathcal{M}}$).

4.6 Calculation of the posterior distribution

4.6.1 Approximative method

Approximative calculation of the posterior distribution is done using an `InfEngineMCMC` object. This object is created using four arguments:

- A `DBN` object.
- A sampler object. This object takes care of sampling the hidden nodes in the DBN. Currently, this needs to be a `GibbsRandom` object.
- A sequence object, i.e. an `MDArray` containing the data.
- A mismask object, which indicates which nodes are hidden where in the sequence.

The `InfEngine` objects is created as follows:

```
GibbsRandom sampler = GibbsRandom(dbn);
InfEngineMCMC inf = InfEngineMCMC(dbn, &sampler, seq,
    mismask);
```

Sampling in Mocapy++ is simply done by generating a set of sequences where the hidden nodes have been sampled using Gibbs sampling. The probability of each hidden node value given the observed node values can then be calculated easily from the set of generated sequences. The likelihood of a sequence can be calculated in a similar way from the likelihoods of the generated sequences.

4.7 Calculating the Viterbi path

4.7.1 Approximative

Mocapy++ can approximate the Viterbi path using a set of sampled sequences. This is done by applying a dynamic programming algorithm that strings together pieces of the sampled sequences so as to find the sequence with the highest likelihood.

The path is calculated by the `InfEngineMCMC` class. First, one creates a Viterbi path generator object:

```
GibbsRandom mcmc = GibbsRandom(dbn);
InfEngineMCMC inf = InfEngineMCMC(dbn, &mcmc, seq, mismask);
inf.initialize_viterbi_generator(50, 50, true);
```

Burn-in and random initialisation are done once initially as explained in the previous section. The first time the `next` method of the Viterbi path generator is called, it generates 50 sequence samples, and uses these to approximate the Viterbi path. Subsequent calls of `the 'next' method` again sample 50 sequences. But this time, the previously generated Viterbi path is added to this set before a new Viterbi path approximation is calculated. In other words, after the initial `next` call, subsequent calls try to improve the initially generated Viterbi path approximation:

```
for (uint i=0; i<1000; i++) {
    Sample s = inf.viterbi_next();
    cout << "LL=" << s.ll/s.slice_count;
}
```

The sequence of `next` calls is typically stopped when the LogLik has converged. The more sequences are sampled and the higher the number of `next` calls, the more precise the Viterbi path will be. Keep in mind that it is in principle possible to get stuck in a local LogLik maximum, so you might want to retry the procedure with different seed values (using `mocapy_seed`) and select the best Viterbi path. The complexity of the procedure to calculate the Viterbi path is O^2, so the method can become slow if you use many sequences and `next` calls.

4.8 Parameter learning

4.8.1 S-EM and MC-EM

Parameter learning is done using the `EMEngine` class. The E- and M-step are done by the `do_E_step` and `do_M_step`, respectively. The average conditional LogLik per slice is returned by the `get_loglik` method.

The number of burn-in steps and the number of mcmc steps of the sampling are specified by the first and second arguments of the `do_E_step` method. Arguments to `EMEngine` are the DBN object whose parameters will be updated and the object that will actually perform the sampling steps. Currently, Mocapy++ only implements Gibbs sampling in the `GibbsRandom` class, but more sampling method will be added. In addition, it is also possible to attach weights to the sequences using the `weight_list` argument: this should be a list of weights (from 0.0 to 1.0) for each sequence.

The following code shows the MC-EM procedure (100 EM steps). Each E-step initializes the hidden node values to random values, performs 10 burn-in steps and generates 10 sequence samples for each sequence. The M-step uses the latter 10 samples to update the parameters.

```
GibbsRandom mcmc = GibbsRandom(model_dbn);
EMEngine em = EMEngine(model_dbn, seq_list, mismask_list);
for (uint i=0; i<100; i++) {
    em.do_E_step(50, 50, true);
    double ll = em.get_loglik();
    em.do_M_step();
    cout << "LL=" << ll << endl;
}
```

The next code snippet shows an S-EM procedure. At the first iteration, the hidden node values are initialised to random values, and 50 burn-in steps are done. After that, the hidden node values are re-used between E-steps. Each E-step generates a single sample for each sequence, which is then used in the M-step.

```
GibbsRandom mcmc = GibbsRandom(model_dbn);
EMEngine em = EMEngine(model_dbn, seq_list, mismask_list);
for (uint i=0; i<100; i++) {
    if (i==0) {
        em.do_E_step(1, 50, true);
    }
    else {
        em.do_E_step(1, 0, false);
    }
    double ll = em.get_loglik();
    em.do_M_step();
    cout << "LL=" << ll << endl;
}
```

Other approaches are possible. After a certain amount of iterations of MC-EM, when the parameters are getting closer to their optimal values, one can skip the burn-in steps and the random initialisation of the hidden nodes. It is of course also possible to turn off the random initialisation while still having a burn in period, which can give the values of the hidden nodes the chance to adjust to the new parameters.

4.8.2 Using Priors

Mocapy++ supports a pseudo count prior for the discrete nodes (`PseudoCountPrior` class). Creating `PseudoCountPrior` objects is done as follows (for a `DiscreteNode` with `node_size=2` that has one parent with `node_size=3`):

```
MDArray<double> pcounts(vec(3,2));
pcounts.set(10,10,10);
PseudoCountPrior prior(pcounts);
```

The prior object is passed to the density object of the `DiscreteNode` after creation;

```
Node* n = NodeFactory::new_discrete_node(O_SIZE, "n");
((DiscreteNode*)n)->get_densities()->set_prior(&prior);
```

4.9 DBN Persistence

After an expensive learning step, one probably wants to save the trained DBN for future use. It's quite easy to save a trained DBN by making use of Boost's built-in persistency features.

Saving a trained DBN object can be done as follows:

```
dbn.save("dbn.pickle")
```

Loading the DBN again is done with the `load` function:

```
dbn.load("dbn.pickle")
```

You can for example save a copy of the `DBN` object after every EM-step, and re-use a saved `DBN` object when something goes wrong during learning (i.e. a power failure).

Saving and loading DBNs is done using the functionality of the `Boost` library. You can use this mechanism to save and retrieve any Mocapy++ object if needed.

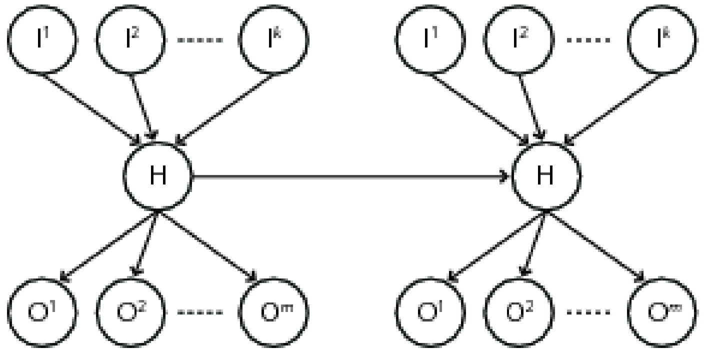

Figure 4.1: Two slices of an input-output HMM with multiple inputs and outputs. For a given slice, j, the DBN has a set of input variables $\mathbf{I}_j = \{I_j^1, \dots, I_j^k\}$, a hidden state H_j and a set of output variables $\mathbf{O}_j = \{O_j^1, \dots, O_j^m\}$. **I** and **H** are all discrete variables, while each of the variables in **O** can be either discrete or continuous.

4.10 Inference in hidden Markov models and mixture models

Mocapy++ provides exact methods for doing inference in hidden Markov models (HMMs) and mixture models (MMs). In particular, Mocapy++ implements algorithms for sampling and calculating probabilities of partially observed data given the models.

4.10.1 Inference in hidden Markov models

Mocapy++ implements exact methods for doing inference in the class of hidden Markov models that can be described by the DBN depicted in Figure 4.1. We will denote this type of models input-output HMMs (IOHMMs) with multiple inputs and outputs, though the term IOHMM often is used for a broader class of models [21]. The IOHMMs can have an arbitrary number of input and output variables; in fact the input variables are optional while a least one output variable is required. The DBN in Figure 4.1 corresponds to the factorization

$$P(\mathbf{I}, \mathbf{H}, \mathbf{O}) = \prod_j P(\mathbf{I}_j) P(H_j | H_{j-1}, \mathbf{I}_j) P(\mathbf{O}_j | H_j) \ ,$$

where $\mathbf{I}_j = \{I_j^1, \dots, I_j^k\}$ is the set of input variables and $\mathbf{O}_j = \{O_j^1, \dots, O_j^m\}$ is the set of output variables in slice j.

In the following sections we will assume that the output variables are divided into two mutually exclusive sets: a set of observed variables $\mathbf{O}^{\text{obs}}$ and a set of unobserved variables $\mathbf{O}^{\text{unobs}}$. An output node can be in the observed set in one slice while the corresponding node in the following slice can be in the unobserved set; as an example you could have $O_j^1 \in \mathbf{O}^{\text{obs}}$ and $O_{j+1}^1 \in \mathbf{O}^{\text{unobs}}$. In practice the division is declared using a mismask array and flagging the observed variables as `MOCAPY_OBSERVED` and the unobserved variables as `MOCAPY_HIDDEN`.

4.10.1.1 Calculating probability of output variables

The `LikelihoodInfEngineHMM` class is used to calculate the probability of the observed subset, $\mathbf{O}^{\text{obs}}$, conditioned on the input variables

$$P(\mathbf{O}^{\text{obs}} | \mathbf{I}) = \sum_{\mathbf{H}} \prod_j P(H_j | \mathbf{I}_j, H_{j-1}) P(\mathbf{O}_j^{\text{obs}} | H_i) \ .$$

In the implementation, the sum over hidden states is calculated using a slightly modified version of the classical forward algorithm [8].

The `LikelihoodInfEngineHMM` class has the constructor:

```
LikelihoodInfEngineHMM(DBN* newdbn, uint newhiddennodeindex,
    bool checkdbn=true)
```

The argument `new_dbn` is a pointer to the DBN object and `new_hidden_node_index` specifies the index of the hidden node within a slice of the model. If the argument `check_dbn` is set to true then the constructor checks if the model complies with the structure from figure 4.1.

After an `LikelihoodInfEngineHMM` object has been constructed, the `calc_ll` method can be used for calculating the probability of a sequence, `seq`, based on a given mismask, `mismask`. The method has the prototype:

```
double LikelihoodInfEngineHMM::calcll(Sequence & seq,
    MDArray<eMISMASK> & mismask, int start=0,
    int end=-1, bool multiplybyinput=false)
```

The arguments `start` and `end` can be used to calculate the probability of a subsequence of the slices in the dataset, that is $P(\mathbf{O}^{\text{obs}}_{j=\texttt{start}...\texttt{end}-1}|\mathbf{I}, H_{\texttt{start-1}}, H_{\texttt{end}})$. The default value of -1 for `end` means the end of the sequence. If `multiply_by_input` is set to true the input variables are included in the probability, in other words the method calculates $P(\mathbf{O}^{\text{obs}}, \mathbf{I})$. Note that the method assumes that all input variables are always observed – event if they are flagged as otherwise in the mismask.

4.10.1.2 Sampling

The `SampleInfEngineHMM` class is used to sample the sequence of hidden states, $\mathbf{H}$, and the output variables flagged as unobserved, $\mathbf{O}^{\text{unobs}}$, from the conditional distribution $P(\mathbf{H}, \mathbf{O}^{\text{unobs}}|\mathbf{O}^{\text{obs}}, \mathbf{I})$. This is done using a slightly modified version of the forward-backtrack algorithm [7].

The class constructor has the prototype:

```
SampleInfEngineHMM(DBN* newdbn, Sequence & newseq,
    MDArray<eMISMASK> & newmismask, uint newhiddennodeindex,
    bool checkdbn=true, double newweight=1.0);
```

The first argument `new_dbn` is a pointer to the DBN, `new_seq` is the initial sequence used for sampling and `new_mismask` is the mismask defining which output variables are observed and which are unobserved. Again `new_hidden_node_index` specifies the index of the hidden node within a slice of the model and if the argument `check_dbn` is set to true then constructor checks if the model complies with the structure from Figure 4.1.

After an `SampleInfEngineHMM` object has been constructed, the `sample_next` method can be used for sampling $\mathbf{H}$ and $\mathbf{O}^{\text{unobs}}$:

```
Sequence & SampleInfEngineHMM::samplenext()
```

The method takes no arguments and returns the new sample. The probability of this sample can be calculated using the method:

```
double SampleInfEngineHMM::calcll(MDArray<eMISMASK> & mismask,
    int start=0, int end=-1, bool multiplybyinput=false)
```

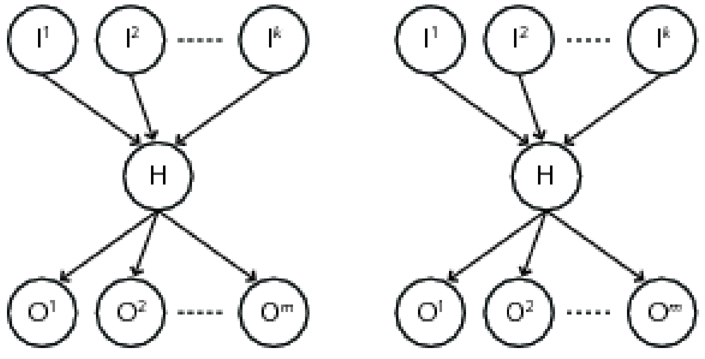

Figure 4.2: Two slices of an input-output MM (IOMM) with multiple inputs and outputs. For a given slice, j, the the DBN has a set of input variables $\mathbf{I}_j = \{I_j^1, \ldots, I_j^k\}$, a hidden state H_j and a set of output variables $\mathbf{O}_j = \{O_j^1, \ldots, O_j^m\}$. **I** and **H** are all discrete variables, while each of the variables in **O** can be either discrete or continuous.

The arguments has the same meaning as for the `cacl_ll` method for the `LikelihoodInfEngineHMM` class.

The method `set_start_end` can be used to only sample a subsequence of the slices; the method has the prototype:

```
SampleInfEngineHMM::setstartend(int start=0, int end=-1);
```

The default values of `start` and `end` means that the whole sequence is sampled in each call to the method `sample_next`. By calling this method and changing these values, the subsequence calls of `sample_next` will only sample the subsequence $\mathbf{H}_{j=\texttt{start}\ldots\texttt{end}-1}$ and $\mathbf{O}^{\text{unobs}}_{j=\texttt{start}\ldots\texttt{end}-1}$ from the conditional distribution

$$P(\mathbf{H}_{j=\texttt{start}\ldots\texttt{end}-1}, \mathbf{O}_{j=\texttt{start}\ldots\texttt{end}-1} | \mathbf{O}^{\text{obs}}, \mathbf{I}, H_{\texttt{start}-1}, H_{\texttt{end}}) .$$

Note that objects of both the `SampleInfEngineHMM` and `LikelihoodInfEngineHMM` class assumes that the DBN given to the constructor stays unchanged within the lifetime of the object. If the distributions in the DBN are changed the methods will give incorrect results.

4.10.2 Inference in mixture models

Mocapy++ also implements exact methods for doing inference in the class of mixture models (MM) that can described by the DBN in Figure 4.2. We call this type of models for input-output MMs (IOMMs) with multiple inputs and outputs. As for the IOHMMs, the IOMMs can have an arbitrary number of input and output variables. The DBN in Figure 4.2 corresponds to the factorization

$$P(\mathbf{I}, \mathbf{H}, \mathbf{O}) = \prod_j P(\mathbf{I}_j) P(H_j|\mathbf{I}_j) P(\mathbf{O}_j|H_j) ,$$

where $\mathbf{I}_j = \{I_j^1, \ldots, I_j^k\}$ is the set of input variables and $\mathbf{O}_j = \{O_j^1, \ldots, O_j^m\}$ is the set of output variables in slice j. As for the IOHMMs, we will assume that the output variables are divided into two mutually exclusive sets: a set of observed variables $\mathbf{O}^{\text{obs}}$ and a set of unobserved variables $\mathbf{O}^{\text{unobs}}$.

Mocapy++ provides two classes for doing inference in IOMMs: the `LikelihoodInfEngineMM` class and the `SampleInfEngineMM` class. These classes have the same interface as the corresponding classes for IOHMMs. The classes only differ in how to calculate the distribution over the hidden states. For IOHMM this distribution, $P(\mathbf{H}|\mathbf{O}_j^{\mathrm{obs}}, \mathbf{I})$, can be calculated directly due the absence of dependencies between the hidden states.

Chapter 5

Examples

5.1 Learning

5.1.1 HMM with discrete output node

This example shows how a simple HMM with discrete output is implemented in Mocapy++. The program first creates a DBN object with random parameters and samples a set of sequences from it for use as artificial training data. Then, a second DBN object with random parameters is trained using these generated sequences. During learning, the LogLik values are printed.

```
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include "mocapy.h"
using namespace mocapy;

int main(void) {
    mocapy_seed(574);
    // Number of trainining sequences
    int N = 100;
    // Sequence lengths
    int T = 100;
    // Gibbs sampling parameters
    int MCMC_BURN_IN = 10;
    // HMM hidden and observed node sizes
    uint H_SIZE=2;
    uint O_SIZE=2;
    bool init_random=false;
    CPD th0_cpd;
    th0_cpd.set_shape(2); th0_cpd.set_values(vec(0.1, 0.9));
    CPD th1_cpd;
    th1_cpd.set_shape(2,2); th1_cpd.set_values( vec(0.95, 0.05, 0.1, 0.9));
    CPD to0_cpd;
    to0_cpd.set_shape(2,2); to0_cpd.set_values( vec(0.1, 0.9, 0.8, 0.2));
    // The target DBN (This DBN generates the data)
    Node* th0 = NodeFactory::new_discrete_node(H_SIZE, "th0", init_random, th0_cpd)
    Node* th1 = NodeFactory::new_discrete_node(H_SIZE, "th1", init_random, th1_cpd)
    Node* to0 = NodeFactory::new_discrete_node(O_SIZE, "to0", init_random, to0_cpd)
    DBN tdbn;
```

```
tdbn.set_slices(vec(th0, to0), vec(th1, to0));
tdbn.add_intra("th0", "to0");
tdbn.add_inter("th0", "th1");
tdbn.construct();
// The model DBN (this DBN will be trained)
// For mh0, get the CPD from th0 and fix parameters
Node* mh0 = NodeFactory::new_discrete_node(H_SIZE, "mh0", init_random, CPD(), t
Node* mh1 = NodeFactory::new_discrete_node(H_SIZE, "mh1", init_random);
Node* mo0 = NodeFactory::new_discrete_node(O_SIZE, "mo0", init_random);
DBN mdbn;
mdbn.set_slices(vec(mh0, mo0), vec(mh1, mo0));
mdbn.add_intra("mh0", "mo0");
mdbn.add_inter("mh0", "mh1");
mdbn.construct();
cout << "*** TARGET ***" << endl;
cout << "th0: " << *th0 << endl;
cout << "th1: " << *th1 << endl;
cout << "to0: " << *to0 << endl;
cout << "*** MODEL ***" << endl;
cout << "mh0: " << *mh0 << endl;
cout << "mh1: " << *mh1 << endl;
cout << "mo0: " << *mo0 << endl;
vector<Sequence> seq_list;
vector< MDArray<eMISMASK> > mismask_list;
cout << "Generating data" << endl;
MDArray<eMISMASK> mismask;
mismask.repeat(T, vec(MOCAPY_HIDDEN, MOCAPY_OBSERVED));
// Generate the data
double sum_LL(0);
for (int i=0; i<N; i++) {
     pair<Sequence, double> seq_ll = tdbn.sample_sequence(T);
     sum_LL += seq_ll.second;
     seq_list.push_back(seq_ll.first);
     mismask_list.push_back(mismask);
 }
 cout << "Average LL: " << sum_LL/N << endl;
 GibbsRandom mcmc = GibbsRandom(&mdbn);
 EMEngine em = EMEngine(&mdbn, &mcmc, &seq_list, &mismask_list);
 cout << "Starting EM loop" << endl;
 double bestLL=-1000;
 uint it_no_improvement(0);
 uint i(0);
 // Start EM loop
 while (it_no_improvement<100) {
     em.do_E_step(1, MCMC_BURN_IN, true);
     double ll = em.get_loglik();
     cout << "LL= " << ll;
     if (ll > bestLL) {
         cout << " * saving model *" << endl;
         mdbn.save("discrete_hmm.dbn");
         bestLL = ll;
         it_no_improvement=0;
     }
     else { it_no_improvement++; cout << endl; }
```

```
            i++;
            em.do_M_step();
        }
        cout << "DONE" << endl;
        mdbn.load("discrete_hmm.dbn");
        cout << "*** TARGET ***" << endl;
        cout << "th0: " << *th0 << endl;
        cout << "th1: " << *th1 << endl;
        cout << "to0: " << *to0 << endl;
        cout << "*** MODEL ***" << endl;
        cout << "mh0: " << *mh0 << endl;
        cout << "mh1: " << *mh1 << endl;
        cout << "mo0: " << *mo0 << endl;
        delete th0;
        delete th1;
        delete to0;
        delete mh0;
        delete mh1;
        delete mo0;
        return EXIT_SUCCESS;
    }
```

5.1.2 HMM with Gaussian output node

Changing the above HMM into an HMM with real valued Gaussian observations is trivial. Below only the relevant difference with the previous HMM code is shown.

```
    MDArray<double> means( vec(2,2) );
    means.set_values( vec(0.0,0.0,10.0,10.0) );
    Node* to0 = NodeFactory::new_gaussian_node(O_SIZE, "to0",
        false, means);
    Node* mo0 = NodeFactory::new_gaussian_node(O_SIZE, "mo0",
        true);
```

5.1.3 Factorial HMM with discrete output node

The following code fragment illustrates how to implement the Factorial HMM that is shown in figure 1.3:

```
    // HMM hidden and observed node sizes
    uint H_SIZE=5;
    uint O_SIZE=3;
    Node* th1 = NodeFactory::new_discrete_node(H_SIZE, "th1");
    Node* th2 = NodeFactory::new_discrete_node(H_SIZE, "th2");
    Node* th3 = NodeFactory::new_discrete_node(H_SIZE, "th3");
    Node* th01 = NodeFactory::new_discrete_node(H_SIZE, "th01");
    Node* th02 = NodeFactory::new_discrete_node(H_SIZE, "th02");
    Node* th03 = NodeFactory::new_discrete_node(H_SIZE, "th03");
    Node* to = NodeFactory::new_discrete_node(O_SIZE, "to");
    DBN tdbn;
    tdbn.set_slices(vec(th01, th02, th03, to),
        vec(th1, th2, th3, to));
    tdbn.add_intra("th1", "to");
    tdbn.add_intra("th2", "to");
    tdbn.add_intra("th3", "to");
    tdbn.add_inter("th1", "th01");
    tdbn.add_inter("th2", "th02");
    tdbn.add_inter("th3", "th03");
    tdbn.construct();
```

Chapter 6

Mocapy++'s design

Mocapy++'s design is fairly transparent. Figure 6.1 shows a UML-like diagram of its architecture, featuring inheritance and interactions among the classes. Some points to note are:

- All node classes are instantiated from the `ChildNode` template class which is derived from the `Node` class.
- The type parameters of the `ChildNode` template are the `ESS` class and the `Densities` class.
- The `ESS` class contains the *Expected Sufficient Statistics* (ESS) for the given node. Basically it implements how a sample (from the Gibbs sampler) should be added and stored, such that estimation of the distributions parameters can be done. An `ESS` class must be implemented for each node type.
- The `Densities` class implements the estimation and sampling procedures of the given distribution.
- EM learning is done by the `EMEngine` class.
- Deterministic inference (smoothing and Viterbi) is done by the `InfEngineHMM` class. Sampling and stochastic Viterbi is done by the `InfEngineMCMC` class.
- Currently there is one class that does the actual sampling, i.e. the `GibbsSampler` class. It is easy to implement a new sampler class by using the `MCMC` class as a base class.
- The `EMEngine, InfEngineHMM`, `InfEngineMCMC` and `GibbsSampler` classes all interact directly with the `DBN` and `Node` classes to perform their operations. What goes on in the `Node` classes itself is however completely shielded by the uniform `Node` class interface.
- The `DiscretePriors` module contains the prior for discrete node.

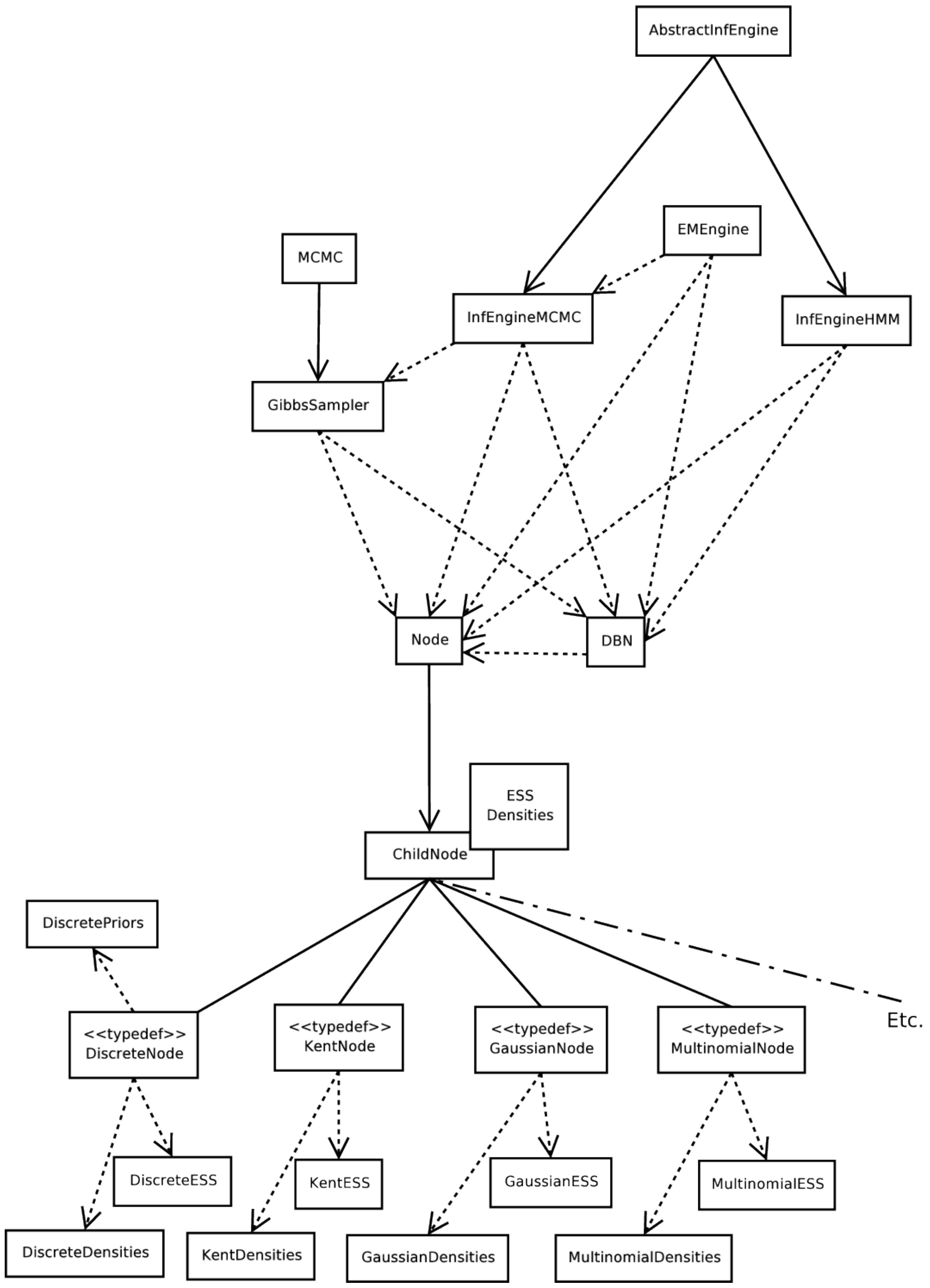

Figure 6.1: Mocapy++'s overall design. Full arrows indicate inheritance, dotted arrows indicate a dependency (i.e. 'a uses b'). Dotted boxes indicate modules, ordinary boxes indicate classes.

Chapter 7

Extending Mocapy

We've taken great care to document Mocapy++'s code, and have tried to keep it's architecture (see previous chapter) as clear as possible without compromising performance too much. Hence, it should be possible to understand how Mocapy++ works, and subsequently to extend the toolkit.

7.1 More Node types

Adding new node types is easy in Mocapy++. To add a new node X, the following four files must be created:
{X}ess.{h,cpp} {X}densities.{h,cpp}

The {X}ESS class inherits from the abstract ESSBase class and you must override the `construct` and `add_ptv` methods. In the `construct` method you must define the appropriate shape of the ESS, and in `add_ptv` you must add a sample point to the ESS. You can look at the existing *ess.cpp files for examples/inspiration.

The {X}Densities class inherits from the abstract DensitiesBase class. You should write your own constructor with the appropriate parameters for your node.

You must override the following methods:

- construct(parent_sizes): Initialize your parameters (mean, covariance, CPD, etc.).
- estimate(ess): Estimate the parameters of the node based on the ESS (having the structure you chose in {X}ess.cpp)
- sample(ptv): Return a sample based on indicated parent values. Note that *ptv* stand for "**p**arent and **t**his node's **v**alues": the values of the parents nodes and the value of the node to which the method belongs.
- get_lik(ptv, log): Return likelihood, $P(\text{child} \mid \text{parent})$
- get_parameters(): Return the distribution's parameters
- get_string(): Return string representation of parameters

7.2 More sampling methods

It's easy to add new sampling algorithms. Sampling is done by a subclass of the `MCMC` base class in the `MCMC` module. Currently, only random-sweep Gibbs sampling is implemented in the `GibbsRandom` class, which can also be found in the `MCMC` module. This can serve as a good example of how to proceed.

Chapter 8

Acknowledgements

Mocapy++ is part of a protein structure prediction project at the *Bioinformatics Centre, University of Copenhagen, Denmark.*

Funding came from the *Lundbeckfond* (February 2004-January 2005, `www.lundbeckfonden.dk`), a *Marie Curie intra-European fellowship* within the *6th European community framework programme* (February 2005-January 2006) and a grant from the *Danish research council* (*Fund for technology and production sciences*; project *"Data driven protein structure prediction"*).

We thank Anders Krogh, *Bioinformatics Center, University of Copenhagen*, for his support of this project. John T. Kent, *Department of Statistics, University of Leeds, UK*, kindly contributed a method to sample efficiently from the Kent distribution. Kanti Mardia and Charles C. Taylor, *University of Leeds, UK*, developed the bivariate von Mises distribution.

In addition, we thank Ole Winther, *Digital Signal Processing, Informatics and Mathematical Modelling, Technical University of Denmark*, Niels Hansen, *Bioinformatics Center, University of Copenhagen*, Søren Feodor Nielsen, *Department of Statistics and Operations research, Institute for Mathematical Sciences, University of Copenhagen* and Jesper Ferkinghoff-Borg, *Niels Bohr Institute, Denmark* for interesting discussions and suggestions.

Finally, the authors thank all people at *Bioinformatics Centre, University of Copenhagen, Denmark* who have helped implementing Mocapy++: Christian Andreetta, Wouter Boomsma, Mikael Borg, Jes Frellsen, Tim Harder and Kasper Stovgaard. All shortcomings in this project are of course due to the authors.

Chapter 9

GNU general public license

Bibliography

[1] Andrieu, C., de Freitas, N., Doucet, A., and Jordan, M. I. An introduction to MCMC for machine learning. *Machine Learning 50* (2003), 5–43.

[2] Angerson, E., Bai, Z., Dongarra, J., Greenbaum, A., Mckenney, A., Du Croz, J., Hammarling, S., Demmel, J., Bischof, C., Bischof, C., Sorensen, D., and A10. Lapack: A portable linear algebra library for high-performance computers. In *Supercomputing '90. Proceedings of* (1990), pp. 2–11.

[3] Baldi, P., and Brunak, S. *Bioinformatics : the machine learning approach.* MIT Press, 1998.

[4] Boomsma, W., Kent, J., Mardia, K., Taylor, C., and Hamelryck, T. Graphical models and directional statistics capture protein structure. In *Interdisciplinary Statistics and Bioinformatics* (2006), S. Barber, P. Baxter, K. Mardia, and R. Walls, Eds., vol. 25, Leeds University Press, pp. 91–94.

[5] Boomsma, W., Mardia, K. V., Taylor, C. C., Ferkinghoff-Borg, J., Krogh, A., and Hamelryck, T. A generative, probabilistic model of local protein structure. *Proceedings of the National Academy of Sciences 105*, 26 (July 2008), 8932–8937.

[6] Brooks, S. Markov chain Monte Carlo method and its applications. *The Statistician 47* (1998), 69–100.

[7] Cawley, S. L., and Pachter, L. HMM sampling and applications to gene finding and alternative splicing. *Bioinformatics 19 Suppl 2* (October 2003).

[8] Diebolt, J., and Ip, E. *Markov chain Monte Carlo in practice.* Chapman & Hall/CRC, 1996, ch. 15: Stochastic EM: method and application, pp. 259–273.

[9] Durbin, R., Eddy, S. R., Krogh, A., and Mitchison, G. *Biological sequence analysis: probabilistic models of proteins and nucleic acids.* Cambridge Univ. Press, 2000.

[10] Frellsen, J., Moltke, I., Thiim, M., and Hamelryck, T. A probabilistic model of RNA conformational space. *PLoS Computational Biology (to appear)* (2009).

[11] Ghahramani, Z. Learning dynamic Bayesian networks. *Lecture Notes in Computer Science 1387* (1998), 168–197.

[12] Ghahramani, Z., and Jordan, M. I. Factorial hidden markov models. *Machine Learning 29* (1997), 245–273.

[13] Gilks, W., Richardson, S., and Spiegelhalter, D., Eds. *Markov chain Monte-Carlo in practice.* Chapman & Hall/CRC, London, 1996.

[14] Godsill, S., Doucet, A., and West, M. Maximum a posteriori sequence estimation using Monte Carlo particle filters. *Ann. Inst. Stat. Math. 53* (2001), 82–96.

[15] HAMELRYCK, T., KENT, J., AND KROGH, A. Sampling realistic protein conformations using local structural bias. *PLoS Comput. Biol. 2*, 9 (2006), e131.

[16] JORDAN, M., Ed. *Learning in graphical models.* MIT Press, 1998.

[17] KENT, J. The Fisher-Bingham distribution on the sphere. *J. Royal Stat. Soc. 44* (1982), 71–80.

[18] KENT, J., AND HAMELRYCK, T. Using the Fisher-Bingham distribution in stochastic models for protein structure. In *Quantitative Biology, Shape Analysis, and Wavelets* (2005), S. Barber, P. Baxter, K. Mardia, and R. Walls, Eds., vol. 24, Leeds University Press, pp. 57–60.

[19] LEONG, P., AND CARLILE, S. Methods for spherical data analysis and visualization. *J. Neurosci. Met. 80* (1998), 191–200.

[20] MARDIA, KANTI, V., TAYLOR, CHARLES, C., SUBRAMANIAM, AND GANESH, K. Protein bioinformatics and mixtures of bivariate von mises distributions for angular data. *Biometrics 63*, 2 (June 2007), 505–512.

[21] MURPHY, K. An introduction to graphical models. Tech. rep., UC Berkeley, Computer Science Division, 2001.

[22] MURPHY, K. *Dynamic Bayesian Networks: Representation, Inference and Learning.* PhD thesis, UC Berkeley, Computer Science Division, 2002.

[23] NEAL, R. Probablistic inference using markov chain monte carlo methods. Tech. rep., Dept. of Computer Science, University of Toronto, September 1993.

[24] PEEL, D., WHITEN, W., AND MCLACHLAN, G. Fitting Mixtures of Kent Distributions to Aid in Joint Set Identification. *J. Am. Stat. Ass. 96* (2001), 56–63.

[25] RABINER, L. A Tutorial in Hidden Markov Models and Selected Applications in Speech Recognition. *Proc. of the IEEE 77* (1989), 257–286.

[26] WOJTCZYK, M., AND KNOLL, A. A cross platform development workflow for c/c++ applications. In *Software Engineering Advances, 2008. ICSEA '08. The Third International Conference on* (2008), pp. 224–229.