Probabilistic Network Library for MATLAB

User Guide and Reference Manual

Version	Version History	Date
-001	Original Issue	June, 2004

Contents

Chapter **Contents**

Chapter 1

Overview

Chapter 2

User Guide

Chapter 3

Reference Manual

Overview

1

This manual describes *Probabilistic Network Library for MATLAB* (*PNLM*), the general tool for working with graphical models. The library contains high-performance implementation of algorithms for working with Bayesian networks and Markov networks, such as belief propagation and Junction tree inference, maximum likelihood and expectation maximization. The library is aimed at a wide spectrum of graphical models applications including computer vision, pattern recognition, data mining, and decision theory. The PNLM core engine will be optimized and parallized to give maximum performance on Intel® Architectures.

About This Library

The library can be roughly divided into three parts:

— *graphical models* that implement graphical models (Bayesian and Markov networks), including dynamic graphical models (Dynamic Bayesian networks). This part includes an implementation of the graph structure along with the factors (tabular and Gaussian so far) to specify the factorized distribution;

— *inference engines* that contain Naive inference, Junction tree inference, and belief propagation;

— *learning engines* that implement maximum likelihood, expectation maximization, and score-based structure learning.

About This Software

The library is open source and free for use on license terms.

The library is available for MATLAB only.

About This Manual

The manual consists of two parts: User Guide and Reference Manual. The first part contains the overview of the implemented algorithms together with sample calls of PNLM functions to solve specific tasks. The second part gives a systematic description of the objects and their member functions.

Notational Conventions

In this manual, notational conventions include:

- Fonts used for distinction between the text and the code
- Naming conventions.

Font Conventions

The following font conventions are used:

This type style	Newly introduced important notions in User Guide; for example, *Markov Random Fields, chain graph*.
`This type style`	Mixed with the uppercase in class structure names as in `CGraph`; also used in function names, code examples, public destructor names, and call statements; for example, `virtual void CCPD`, `~CFactor()`.
`This type style`	Variables and parameter types in arguments discussion; for example, *`SerialNumber`*, *`data`*, *`dtTabular`*.

Naming Conventions

The PNLM software uses the following naming conventions for different items:

- All class names start with prefix C, for example, `CGraphicalModel`.
- All global functions start with prefix `pnl`, for example, `pnlDetermineDistributionType`.
- Every new part of a function name starts with an uppercase character, without underscore; for example, `GetDomainSize`.

User Guide

2

Graphical Models

A *probabilistic graphical model* (PGM) is a factorized joint probability distribution over a set of random variables which are called *model domain*. A *factor* is a function defined on a small subset of variables called *factor domain*. From the probabilistic viewpoint the factorized representation encodes independence relationships, while from the technical viewpoint it relaxes strict memory and computing power requirements for using PGMs, which allows exploitation of models with large domains.

Probabilistic graphical models have three components:

- variables (model domain)
- factorization type (structure)
- factors proper.

Variables of the model can be either discrete vectors, which take a finite number of values, or continuous vectors.

All commonly used factorization types have a corresponding graph representation. Nodes of a graph correspond to random variables. In this documentation we will further identify the notion of a random variable with the notion of a node in a graph. Edges of the graph reflect the factorization of the joint probability distribution.

PNLM implements some important classes of graphical models:

— *Markov Random Fields* (*MRF*s), also called *Markov Networks* (*MNet*s), that are characterized by undirected graphs. The domain of each factor is a number of nodes of the graph, which form a clique.

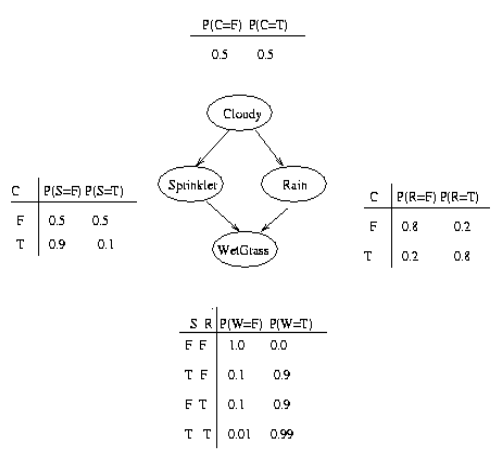
— *Bayesian Networks* (*BNets*) are represented by *directed acyclic graphs* (*DAG*), where each factor is associated with a child node and has the domain consisting of all parent nodes and the child node.

A factor in a Bayesian Network has the form of *conditional probability distribution* (*CPD*) for a child node, with its parent nodes provided. In this context a directed edge from node *A* to node *B* may be interpreted as a causal relationship, though the absence of the edge does not mean that nodes are statistically independent.

The third constituent of a graphical model is a factor. It may have different forms and functionality depending on the type of the model. MRF factors are arbitrary positive functions called *potentials*. BNet factors are CPDs – positive functions that sum to 1 over the child node regardless of parent node values.

A graphical model is created in PNLM by the routine shown in Example 2-1. The routine applies to Bayesian networks and with some changes - to Markov Networks. The model of the example is called “water-sprinkler”. The graph structure of the model and its parameters (CPDs) are all shown in Figure 2-1:

Figure 2-1 Water-sprinkler model



The nodes are numbered as follows:

Cloudy (C) = 0;

Sprinkler (S) = 1;

Rain (R) = 2;

Wet Grass (W) = 3

PNLM has special containers for storing scalar and vector data. A `CValue` object is created to store inhomogeneous scalar data used as evidence. `pnlVector` is a template that stores vector data. For the sake of brevity PNL defines several synonyms to specializations of `pnlVector`. For more details see Reference Manual.

Example 2-1 Creation of water sprinkler Bayesian network

```
function bnet = WaterSprinklerBNetCreation()

numOfNds = 4;

%create NodeType objects and specify node types for
%all nodes of the model;
nodeTypes{1} = CNodeType(1, 2);% node type - discrete and binary
nodeAssociation(1:numOfNds) = 0;
%means that all nodes are of the same node type,
%which is 0th one in the array of node types for the model

%tables of probability distribution
matrix1 = [ 0.5, 0.5 ];
matrix2 = [ 0.5, 0.5; 0.9, 0.1 ];
matrix3 = [ 0.8, 0.2; 0.2, 0.8 ];
matrix4(:,:,1) = [ 1.0, 0.1; 0.1, 0.01 ];
matrix4(:,:,2) = [ 0.0, 0.9; 0.9, 0.99 ];

matrices = {matrix1, matrix2, matrix3, matrix4};
%there are several ways to create bayesian network
if 1
   % neighbors can be of either one of three following types:
   % a parent, a child or just a neighbor - for undirected graphs.
   % if a neighbor of a node is it's parent, then neighbor type is ntParent
   % if it's a child, then ntChild and if it's a neighbor, then ntNeighbor
   nbrs = {
       [ 1, 2 ],
       [ 0, 3 ],
       [ 0, 3 ],
       [ 1, 2 ]
   };
   nbrsTypes = {
       { 'ntChild',  'ntChild'  },
```

Example 2-1 Creation of water sprinkler Bayesian network

```
        { 'ntParent', 'ntChild'  },
        { 'ntParent', 'ntChild'  },
        { 'ntParent', 'ntParent' }
    };

    %this is a creation of directed graph for the BNet model
    graph = CGraph( nbrs, nbrsTypes );


    %creation BNet
    bnet = CBNet( numOfNds, nodeTypes, nodeAssociation, graph );

    for i=1:numOfNds
        AllocFactorByDomainNumber(bnet, i-1);
        factor = GetFactor(bnet, i-1);
        AttachMatrix(factor, matrices{i}, 'matTable');
    end

else

    %create model domain
    MD = CModelDomainCreate( nodeTypes, nodeAssociation );

    %create adjacency matrix
    A = zeros(numOfNds,numOfNds);
    A(1,2) = 1;
    A(1,3) = 1;
    A(2,4) = 1;
    A(3,4) = 1;

    %create graph by adjacency matrix
    graph = CGraphCreateFromAdjMat(A);

    %create BNet by cgraph and model domain
    bnet = CBNetCreateByModelDomain( graph, MD );

    for i=1:numOfNds
        domain = [GetParents(graph, i-1); i-1];
        cpd = CTabularCPDCreate(MD,domain, matrices{i});
        AttachFactor(bnet, cpd);
    end

end
```

Dynamic Graphical Models

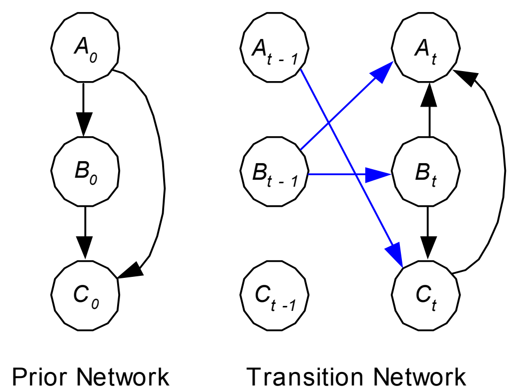
Dynamic Bayesian Network (DBN) represents a directed graphical model of stochastic processes that generalize *Hidden Markov models* (HMMs) and *Kalman Filter models* (KFMs) by representing the hidden and the observed state in terms of state variables, which can have complex interdependencies. DBN is defined by the following characteristics:

- prior, or initial, network
- transition network frequently named *two-slice temporal Bayesian Network* (2TBN).

Prior network determines distribution of probabilities for all variables at the initial moment of time. 2TBN represents a two-slice Bayesian network whose first layer nodes have no parameters associated with them and determine the system at the previous moment of time while each second layer node has conditional probabilities (Figure 2-2).

Nodes of the second slice can have parents both in that very same layer (corresponding to time t), and in the layer that represents the previous moment. Note, that the word "dynamic" does not mean that the network changes over time. It only means that a dynamic process is modelled.

Figure 2-2 Dynamic Bayesian Networks

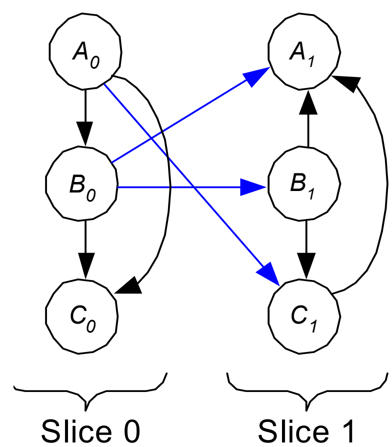


The semantics of the DBN can be defined by unrolling the 2TBN for T time slices. The resulting joint probability distribution is defined by the formula:

$P(x_{0:T-1}) = \prod_{t=0}^{T-1}\prod_{i=0}^{n-1}P(x_t^i|\pi(x_t^i))$, where $\pi(x_t^i)$ means parents of x_t^i, that is, i^{th} node in t^{th} time-slice, n is the number of nodes.

The Dynamic Bayesian network is stored in terms of PNLM in a similar way as the Bayesian network. Suppose, the prior network consists of n nodes. Then the network stored internally to represent the Dynamic Bayesian network will consist of $2n$ nodes. First n nodes are joined in one graph to represent the topology of the prior network. Nodes with numbers starting with n to $2n-1$ are joined in a graph that represents the i^{th} slice, where $i > 0$. The joint graph is formed by the combination of the two layers (prior and the i^{th} layers). Figure 2-3 shows a Bayesian network constructed by unrolling in two time-slices of a dynamic Bayesian network. Note, that it is always possible to restore the prior and the transition networks.

Figure 2-3 Unrolled Bayesian Networks



A DBN is created in terms of PNL by the following routine:

Example 2-2 Creation of DBN

```
function dbn = ArHMMCreation

% Make an HMM with autoregressive Gaussian observations (switching AR
model)
%   X1 -> X2
%   |      |
%   v      v
%   Y1 -> Y2


nnodesPerSlice = 2;
%create adjacency matrix
dag = zeros(2*nnodesPerSlice, 2*nnodesPerSlice);
dag(1, 2) = 1;
dag(1, 3) = 1;
dag(2, 4) = 1;
dag(3, 4) = 1;
```

Example 2-2 Creation of DBN

```
%create graph using adjacency matrix
graph = CGraphCreateFromAdjMat(dag);

%define types of nodes
nodeTypes = cell(1, 2);
nodeTypes{1} = CNodeType(1, 2);
nodeTypes{2} = CNodeType(0, 1);

nodeAss = [0 1 0 1];
MD = CModelDomainCreate(nodeTypes, nodeAss);
%create bnet corresponding to first two slices of dbn
bnet = CBNetCreateWithRandomMatrices(graph, MD);
%create dbn
dbn = CDBNCreate(bnet);
```

Inference Algorithms for Bayesian and Markov Networks

The inference problem in the context of a graphical model is equivalent to the estimation of *joint probability distribution*, also called *marginal distribution* or simply *marginal*, of one or several nodes without evidence or with a limited number of observed nodes:

$P(x_{q1},\ x_{q2},\ \dots\ x_{qk}|x_{e1},\ x_{e2},\dots x_{es})\ =\ P(X_q|X_e)$,

where e denotes the evidences or observed nodes, and q denotes the query nodes whose distribution is to be calculated.

This problem has several solutions. The most evident of them is the direct computation of joint probability distribution for all nodes of the graphical model followed by calculation of probability distribution for the query nodes using Bayes equation:

$$P(X_q|X_e) = \frac{P(X_q,\ X_e)}{P(X_e)} = \frac{\sum(P(x_1, x_2,\ \dots,\ x_N),\ i \notin\ q \cup e)}{\sum P(x_1, x_2,\ \dots,\ x_N),\ i \notin\ e}.$$

This joint probability distribution can be found through multiplication of all conditional probability distributions of a Bayesian network or of all joint probability distributions at the cliques of the Markov network. Before multiplication these conditional and unconditional distributions should be adjusted to the values of observed nodes of the network. The final step is to sum up the resulting values.

This description fully applies to the `CNaiveInfEngine`.

See Example 2-3 of call of such inference engine for the “water-sprinkler” model (Figure 2-1).

Example 2-3 Creation of inference engine for water srinkler BNet

```
pBNet = WaterSprinklerBNetCreation;

% create junction tree inference engine
jtree = CJtreeInfEngineCreate(pBNet);

%create evidence
%let node 1 took on value 0 and node 2 took on value 1
obsNds = [ 1, 3 ];
obsNdsVls = [ 1, 1 ];

e = CEvidenceCreate( pBNet, obsNds, obsNdsVls );

% add evidence to engine
EnterEvidence( jtree, e );

%Finally, we can compute p=P(node_2 |node_1 = 0, node_3 = 1) as follows.
query = [ 2 ];

MarginalNodes( jtree, query );
margPot = GetQueryJPD( jtree );
matrix = GetMatrix( margPot, 'matTable' );
barh(matrix);

disp('example of inference on Water Sprinkler BNet is completed');
```

However, the direct computation is too laborious, as the complexity of computations grows exponentially with the number of nodes in a network. This type of computation appears to be ineffective even for small models and is seldom used in practice.

To reduce the complexity of computations, you may use the distribution law. Since in certain areas of the network local distributions are independent of variables,you can apply the distribution law to calculate distributions for query nodes. So, for example, the probability distribution for the “water-sprinkler” problem at node 3, which has no observed variables, is expanded as follows:

$$P(x_3) = \sum_{x_0,\ x_1,\ x_2} P(x_0,\ x_1,\ x_2,\ x_3) = \sum_{x_0,\ x_1,\ x_2} P(x_0,\ x_1)\, P(x_0,\ x_2)\, P(x_1,\ x_2,\ x_3)$$

$$\sum_{x_0,\ x_1,\ x_2} P(x_0,\ x_1,\ x_2)\, P(x_1,\ x_2,\ x_3) = \sum_{x_0,\ x_1,\ x_2} P(x_0,\ x_1,\ x_2) \sum_{x_1,\ x_2} P(x_1,\ x_2,\ x_3)$$

A number of exact and approximate inference engines are based on the use of the distribution law.

Initially each component of the network, be it a single node or a group of nodes, is assigned a certain distribution function, which represents assumed node values of the network. In the course of iterative message passing between neighboring components of the network the distribution function is modified. So, two components can be neighbors in terms of one inference engine and non-neighbors in terms of another. The sequence of message passing, often called a protocol, can also be different: from one component to all the others and back (tree protocol, or serial protocol) or all-to-all simultaneous message passing (parallel protocol).

If a graph of a Bayesian or a Markov network is a tree, most obvious network components are nodes. Neighboring nodes in the graph are also neighbors in the network. In this case you use the exact *Pearl Inference* or *Belief Propagation*. If the graph contains undirected cycles, you can assume nodes as network components and get an approximate result. To get a more accurate result, you should increase the number of iterations and use, for example, the algorithm of Loopy Belief Propagation. In certain cases it does not converges or converges to a local minimum [MWJ], [H], yet it has proven to be exact on acyclic networks [P1]. A lot of research is being carried out at present on the adaptability of Belief Propagation to networks of various types ([WF2000], [WF2001]).

Inference engines of different types are created in the same manner:

```
InfEngine = CPearlInfEngineCreate( grm );
```

As the sample model contains an undirected cycle (through nodes 0, 1, 2, and 3) any inferred result is approximate.

To infer the exact result on an arbitrary network, nodes of the network are grouped into subsets, or clusters, which are set in accordance with the nodes of an auxiliary junction tree structure. Message passing in this case takes place between the nodes of this junction tree. This procedure is called *Junction Tree Inference*, which is exact [LS], [CDLS].

Particle-based Inference

Besides exact inference engines, for example, the Junction Tree Inference, there is an important class of *particle-based inference* methods. To approximate the joint distribution either of all or of a number of the network variables, the method generates a set of approximations, called *particles,* that represent a part of the probability mass. Particle-based approximate inference engine can calculate query potentials and estimate real states of query nodes. Commonly used particle-based method is `GibbsSampling`.

Inference Algorithms for DBNs

The inference problem in the context of a dynamic graphical model is equivalent to the marginal estimation of one or several nodes from a number of slices irrespective of whether the nodes are observed or hidden. It is implemented through the following computation $P(x(i,\ t)|y(:,\ t_1{:}t_2))$, where $x(i,\ t)$ represents the i^{th} hidden variable at time moment *t*, and *t* and $y(:,\ t_1{:}t_2)$ represent all the evidence between times t_1 and t_2. The algorithm often performs computation of joint probability distributions of variables over one or more time slices.

There are several types of inference problems for dynamic graphical models:

- filtering (on-line procedure)
- smoothing
- fixed-lag smoothing (on-line procedure)
- Viterbi decoding
- prediction.

Table 2-1 Types of Inference Problems for DBNs

Procedure	Goal
Filtering	$P(x(t)\|y(1{:}\ t))$ On-line procedure to estimate current model state.
Smoothing	$P(x(1{:}\ t)\|y(1{:}\ t))$ Off-line procedure to estimate the states of the past, given all evidence up to the current time *t*.
Fixed-Lag Smoothing	$P(x(t-dt)\|y(1{:}\ t))$ On-line procedure to estimate the state of some past moment (*t-dt*), given all evidence up to the current time *t*.

Table 2-1 Types of Inference Problems for DBNs (continued)

Procedure	Goal
Viterbi	$\max_{x(1:t)} P(x(1:t) \mid y(1:t))$ Off-line procedure to compute the most likely sequence of hidden states, given the data.
Prediction	$P(x(t+dt) \mid y(1:t))$ On-line procedure that extrapolates probability distribution for future time slices.

Note that filtering is equivalent to fixed-lag smoothing with zero lag.

Inference procedure can be implemented through various approaches, some of which are naïve as those that follow:

— combine all the latent nodes from a single layer into a single meganode and apply the forward-backward algorithm for HMM, if the nodes are discrete.

— unroll DBN and do inference, for example Junction Tree or Pearl Inference, for the BNet obtained as a result of unroll operation.

To compute statistics which are used to learn parameter values, you call inference (smoothing) for a BNet which is as long as the sequence of evidence. If sequences of evidences are of variable lengths, junction trees (for the Junction Tree Inference) should be constructed many times, which considearbly slows down the process, or precomputed and stored for all possible unrolled DBN, which requires a lot of memory. Hence, it is necessary to use a DBN with repeating structure. One of the algorithms that uses repeating structures of DBNs is Zweig's inference algorithm. The algorithm unrolls a DBN once to some T_{max} slices, creates a junction tree and splices out extra cliques from it, when $t < T_{max}$. But T_{max} should be preliminarily specified for the inference, and online inference can be performed for this maximum number of slices. PNL implements 1.5-slice Junction tree inference algorithm [Murphy02]. This approach involves the following steps:

1. Create a 1.5-slice DBN - one time slice of DBN plus interface nodes from the previous slice. Interface nodes are the nodes connected with the nodes from the next slice and they are always the same for all time slices.
2. Create a junction tree for the obtained network.

3. Link up all the junction trees via interfaces.

This algorithm can perform on-line inference with no preliminarily specified T_{max}. Inference procedure consists of two steps, which are the forward and the backward operation. They are the same as the steps in the classical inference algorithm for HMM. See Example 2-4.

Besides exact inferences described above there are different variants of approximate inferences. One of them is the Boyen-Koller inference (BK). BK inference is the approximate inference in which the belief state of the interface clique (clique consists of interface nodes and is used for message passing between slices in 1.5 Slice Junction tree inference) is represented as a product of marginals, even though the factors may be dependent. For details, see [BKUAI98] and [BKNIPS98], which discuss filtering and smoothing respectively. Note that the exact 1.5 Slice Junction tree inference is the special case of BK inference.

Example 2-4 Creation of inference engine for DBN

```
dbn = ArHMMCreation;

%define number of slices
nSlices = 4;
evidences = cell(1, nSlices);

%let node 1 is observed on the every slice
sliceObsNodes = [1];
MD = GetModelDomain(dbn);

%create random evidences
for i = 1 : nSlices
  val = pnlRand(0,1);
  evidences{i} = CEvidenceCreateByModelDomain(MD, sliceObsNodes,val );
end

%create 1.5 Slice Juncton tree inference
infEng = C1_5SliceJTreeInfEngine(dbn);

%defince procedure type (smoothing)
DefineProcedure(infEng,'ptSmoothing', nSlices);

%enter evidences into engine
EnterEvidence(infEng, evidences);

%start smoothing procedure
Smoothing(infEng);
```

Example 2-4 Creation of inference engine for DBN

```
%define query slice and query nodes
querySlice = floor(rand*nSlices);
if querySlice == 0
    query = [0];
else
    query = [0 2];
end

%JPD on query nodes
MarginalNodes(infEng, query, querySlice);
resPot = GetQueryJPD(infEng);

disp(GetDistributionType(resPot));
mat = GetMatrix(resPot, 'matTable');
disp(mat);
```

The dynamic bayesian network is created by a BNet with *2n* nodes, that is a DBN, unrolled in two slices. Nodes with numbers from *0* to *n-1* form a connected graph corresponding to the prior slice. The topology of the prior slice may differ from the topology of other slices with node numbers from n to *2n-1*.

Evidence of every slice with *n* nodes is formed from nodes with numbers from *0* to *n-1*.

To get inference results, the query for the prior slice (slice = *0*) should contain nodes with numbers from *0* to *n-1*. Probability distribution for other slices is acquired from the current *i-th* slice and the preceding slice *i-1*. In this query node numbers *n…2n-1* correspond to the nodes of the current slice, while node numbers *0…n-1* correspond to the nodes of the preceding slice.

Learning for Bayesian and Markov Networks

A graphical model can be defined by its structure and its set of parameters, which are *conditional probability distributions* for dynamic and static Bayesian networks and *potentials* for Markov network. Learning of a graphical model consists in the estimation of model factors so as to ensure the best explanation of information for the model.

Usually input data for learning is presented in a table, where columns correspond to variables of the model and each row represents a learning sample or observation. So, for example, Table 2-2 presents the input data for the sprinkler model (see Figure 2-1).

Table 2-2 Learning Data for Sprinkler Model

Node 1	Node 2	Node 3	Node 4
0	1	0	1
1	0	1	1
0	0	0	0
0	0	0	0
1	0	1	1
0	1	0	1

If a variable is hidden, its value will be missing from the data for learning. Samples in the table are assumed to be independent. The following four types of learning tasks are distinguished to correspond to different a priori information [Introd]:

Table 2-3 Types of Learning Tasks

Type of Task	Graphical Model Structure	Observability of Variables
Type 1	known	All variables are observed
Type 2	known	Some variables are not observed
Type 3	unknown	All variables are observed
Type 4	unknown	Some variables are not observed

NOTE

NOTE. *Only the first three types of learning tasks are considered below. Type four is not supported by the current version of PNLM.*

Type 1

This type of learning uses the ML algorithm which is based on *Maximum Likelihood Estimation* [JorBish]. The algorithm estimates parameters of the graphical model maximizing the value of the likelihood function $p(D|\theta)$, that is, the probability of observability of learning data D for given parameters θ.

Maximum Likelihood Estimation for Bayesian Network

Discrete Case. Consider the case when all variables of the network are discrete [JorBish]. For a given Bayesian network denote the total number of its nodes by U. For a certain node ν the set of all its parents may be denoted by π_ν and $\phi_\nu = \{\nu\} \cup \pi_\nu$. Let A be an arbitrary subset of nodes $A \subseteq U$. Then x_A stands for a tuple of values for the nodes from A. The count of observations, in which the nodes from the set A assume values specified by x_A tuple, may be denoted by $m(x_A)$. The logarithm of the previously described likelihood function is more convenient than the function itself. The logarithm may be found according to the formula:

$$l(\theta,\ D) = \log p(D|\theta) = \log\left(\prod_n p(x_{U,\ n}|\theta)\right) =$$

$$= \sum_{x_U} m(x_U) \log p(x_U|\theta) = \sum_\nu \sum_{x_{\phi_\nu}} m(x_{\phi_\nu}) \log \theta_\nu(x_{\phi_\nu})$$

The values maximizing this function are:

$$\hat{p}(x_\nu|x_{\pi_\nu}) = \hat{\theta}_\nu(x_{\phi_\nu}) = \frac{m(x_{\phi_\nu})}{m(x_{\pi_\nu})} .$$

These estimates are formed independently for each node in the graph.

Multivariate Gaussian Case. In PNLM the Multivariate Gaussian case is implemented only for Bayesian networks.

The vector $\hat{\vec{x}}^k$ may be formed as follows: $\hat{\vec{x}}^k = \langle \vec{y}_0^k, \vec{y}_1^k, \ldots, \vec{x}^k \rangle$, where $\vec{y}_i^k$ and $\vec{x}$ are the vectors of values of the i^{th} parent and its child in the k^{th} example of the table.

The current approach models the joint distribution over a node and its parents as the multivariate Gaussian distribution and finds its Ml estimation. The sufficient statistics after N examples are [Murphy98], [Jordan]:

$$\hat{\vec{\mu}} = \frac{1}{N}\sum_{i=1}^{N}\vec{x}_i, \quad \hat{\Sigma} = \frac{1}{N}\sum \vec{x}_i\vec{x}_i^T - \hat{\vec{\mu}}\hat{\vec{\mu}}^T$$

$\hat{\Sigma}$ and $\hat{\mu}$ can be broken up into blocks corresponding to parent nodes and the child:

$$\hat{\Sigma} = \begin{bmatrix} \hat{\Sigma}_{yy} & \hat{\Sigma}_{yx} \\ \hat{\Sigma}_{xy} & \hat{\Sigma}_{xx} \end{bmatrix}, \quad \hat{\mu} = \begin{bmatrix} \hat{\vec{\mu}}_y \\ \hat{\mu}_x \end{bmatrix}.$$

The result is the Gaussian distribution at the child node in a moment notation:

$$B = \hat{\Sigma}_{xy}\hat{\Sigma}_{yy}^{-1}, \quad \vec{\mu} = \hat{\mu}_x - B\hat{\mu}, \quad \Sigma = \hat{\Sigma}_{xx} - B\hat{\Sigma}_{yx},$$

where matrix B is broken into individual blocks, one for each parent.

Maximum Likelihood Estimation for Markov Network

Undirected models are more flexible than their directed counterparts. Assume that all network variables are discrete. In this case, the log likelihood is found as follows:

$$l(\theta, D) = \log p(D|\theta) = \sum_C \sum_{x_C} m(x_C)\log\psi_C(x_C) - N\log Z,$$

where $\psi_C(x_C)$ is the clique potential, N is the number of evidences, and Z is the normalization factor [JorBish], [Jirousek].

$$Z = \sum_{x_v}\prod_C \psi_C(x_C)$$

If potentials are defined on maximal cliques of the graph, the maximum likelihood estimates for decomposable graphs can be found through inspection:

for every clique	—	set the clique potential to the empirical marginal for that clique;
for every non-empty intersection between cliques	—	associate an empirical marginal with the intersection, and divide that empirical marginal by the potential of one of the two cliques that form the intersection.

If the graph is arbitrary, the *Iterative Proportional Fitting* (IPF) can be used [JorBish], [Jirousek]. If the graph is decomposable, this algorithm converges in a finite number of iterations, updating each potential once.

The IPF process runs as follows. Denote the potential of a clique C at i^{th} iteration by $\psi_C^i(x_C)$ and the joint probability distribution based on these parameter estimates by $p^i(x)$. In this notation the IPF can be written as follows:

$$\psi_C^{i+1}(x_C) = \psi_C^i(x_C)\frac{\tilde{m}(x_C)}{p^i(x_C)}, \text{ where } \tilde{m}(x_C) = \frac{m(x_C)}{N}.$$

The normalization factor Z remains constant through all iteration process, so IPF may be presented in terms of joint probabilities:

$$p^{i+1}(x_U) = p^i(x_U)\frac{\tilde{m}(x_C)}{p^i(x_C)}.$$

In PNLM the estimation of Markov network parameters is based on IPF.

Bayesian Update

Besides factor parameters with exact values (such as, mean and variance in Gaussian distribution), there are parameters in the form of unknown variables which have their own probability distributions with other parameters, termed *superparameters*.

Superparameters are variables too, thus finally there appears to be an infinite hierarchy of parameters. The current version of PNLM supports only a two-level hierarchy in discrete tabular distributions.

Let θ be a parameter of a probability distribution corresponding to some variable.

$P(\theta)$ is a prior distribution of the parameter θ. The task of Bayesian parameter learning is to update the given data *D*, that is to find the conditional distribution $P(\theta|D)$.

According to the Bayes formula

$$P(\theta|D)=\frac{P(D|\theta)*P(\theta)}{P(D)},$$

where $P(D)=\int P(\theta)P(D|\theta)$.

Based on the given parameter distribution function, the distribution function for the unknown variable x is $P(x)=\int P(\theta)P(x|D)d\theta$

The Dirichlet distribution with parameters $a_1, \ldots, a_n$ is a suitable prior distribution for a discrete multinomial distribution (where a variable can give *n* outcomes) with parameters $\theta_1,..,\theta_n$. Dirichlet parameters are interpreted in terms of pseudo counts, where a_i stands for an imaginary observed number of cases when the discrete variable has taken the ***i***-th value. When training data contains a small number of cases, positive pseudo counts allow to assign to its unobserved values a non-zero probability.

Let training data contain $N_1 + \ldots + N_n$ cases, and N_i be a number of cases when the ***i***-th value is observed.

On learning these cases, a posterior distribution of θ becomes a Dirichlet distribution with parameters $a_1+N_1,..,a_n+N_n$. The target distribution of x after integration through parameters is $P(x=i)=\frac{a_i+N_i}{\sum_{k=1}^{n}(a_k+N_k)}$.

This discussion applies to the case of an unconditional distribution where the considered node of the BNet does not have parents. However, you may easily extend it to cases when the node has parents. As there are counts N_{ij} and pseudo counts a_{ij}, that correspond to the case when *x* = *j*, and parents of *x* are in configuration *i*, the target distribution of x becomes $P(x=i)|parents(x)=i)=\frac{a_{ij}+N_{ij}}{\sum_{k=1}^{n}(a_{ik}+N_{ik})}$.

Type 2

This type of inference uses the *Expectation Maximization* (EM) algorithm [Dempster], [Jordan]. The algorithm first assumes the initial state of parameters θ^0 and then starts the iterative process alternately repeating two steps: E-step and M-step.

Consider the process at the i^{th} iteration:

E-step — For each example of the table the probability distribution of the unobserved variable is found from the values of observed variables and the current values of model parameters θ^{i-1}. The expectations of unobserved variables are calculated for each example in the table.

M-step — To maximize the value of the likelihood function, a new value of θ^i is found.

E-step is repeated with the new parameter values.

This process converges to a local maximum.

In PNLM the EM learning engine is implemented for:

- Bayesian networks with discrete or multivariate Gaussian distribution
- Markov networks with discrete distribution.

The following example considers learning of parameters for the water-sprinkler Bayesian network (see Figure 2-1). If all the nodes are observed, learning Type 1 is used. In this case the E-step, which creates an inference engine and performs the inference procedure, does not take place. If some nodes are hidden, learning Type 2 is used. In this case the E-step takes place, creating an instance of inference engine, which is a junction tree engine by default.

Example 2-5 Creation of learning engine for water-sprinkler BNet

```
bnet = WaterSprinklerBNetCreation
nnodes = GetNumberOfNodes(pBNet);

%generate random samles
nSamples = 100;
samples = GenerateSamples(bnet, nSamples);
for i = 1:nSamples
    evidence = samples{i};
```

Example 2-5 Creation of learning engine for water-sprinkler BNet

```
        %make arbitrary node unobserved
        node = round(rand*(nnodes-1));
        MakeNodeHidden(evidence, node);
    end

    %create learning engine
    eng = CEMLearningEngineCreate(bnet);

    %set observations
    SetData(eng, samples);

    %start learning procedure
    Learn(eng);

    disp('example of learnng is completed');
```

After learning parameters of the Bayesian network assume new values which maximize the likelihood function. The new values correspond to the array of learning data. The table with updated data may be used in further training in the following two ways:

Option 1. Ignore the data of the previous learning. Use the `SetData` function to implement the variant.

Example 2-6 Entering New Data

```
% entering new data and clear accumulated information.
% here evNew is the newly created array of evidences
SetData( eng, evNew )
%call learning
Learn(eng);
```

The parameters of the Bayesian network assume new values that correspond to the learning data.

Option 2. Use data from the previous learning. Use the `ApprendData` function to implement the variant.

Example 2-7 Using data from previous learning

```
AppendData( eng, evNew )
Learn( eng );
```

Type 3

The current version of PNLM carries out structure learning for static BNets and does not support other models. The learning engine calls Maximal Likelihood parameter learning. In this version of PNLM learning is carried out under the condition that the input data is complete, that is, when all nodes of training cases are observed. The algorithm supports graphical models with tabular distributions.

Structure Comparison Metric

One of the solutions to the learning task in this case is the computation of joint probability $p(D, S^h)$ for the learning data D and the model structure S^h:

$$\log p(D, S^h) = \log p(D|S^h) + \log p(S^h).$$

In the case of a Bayesian network with discrete variables, the first item in the above formula is found by applying *Bayesian Information Criterion* (BIC) [Jordan]:

$$\log p(D|S^h) \approx \log p(D|\theta^*, S^h) - \frac{d}{2}\log N,$$

where θ stands for the network parameters, N is the number of observations, and d is the number of network parameters. This criterion is a good approximation of the ML criterion discussed above. In BIC the first item shows the degree of consistency of the network parameters with the modelled data, and the addend reflects the descriptive complexity of the network. Vector θ^* can be found by the formula:

$$\theta^* = \underset{\theta}{\operatorname{argmax}} \log (p(D|\theta, S^h)p(\theta|S^h)).$$

The problem of selecting the most suitable Bayesian network from all network configurations is NP hard. The algorithm implemented in PNLM iterates through all graph topologies that contain no directed cycles. The total number of such topologies is $2^{\frac{N(N-1)}{2}}$, where N is the number of nodes, and $N!$ is the number of node permutations. The total number of Bayesian networks with the topology is $N! \cdot 2^{\frac{N(N-1)}{2}}$

The following example considers structure learning for a Bayesian network using `CBICLearningEngine`. This class is used for learning networks with discrete parents only.

Example 2-8 Structure learning for Bayesian network using PNLM

```
%generate samples from Water Sprinkler network
testBNet = WaterSprinklerBNetCreation;
nsamples = 200;
evidences = GenerateSamples(testBNet, nsamples);

%bayesian network reconstruction

%create an empty graph with the same number of nodes
nnodes = GetNumberOfNodes(testBNet);
mat = zeros(nnodes);
graph = CGraphCreateFromAdjMat(mat);

%create BNet with the same model domain
MD = GetModelDomain(testBNet);
bnet = CBNetCreateWithRandomMatrices(graph, MD);

%create learning engine
eng = CBicLearningEngineCreate(bnet);
%set input data
SetData(eng, evidences);
%startlearning
Learn(eng);

%get result of learning
resBNet = GetGraphicalModel(eng);
resGraph = GetGraph(resBNet);
resAdjMat = CreateAdjacencyMatrix(resGraph);

testGraph = GetGraph(testBNet);
testAdjMat = CreateAdjacencyMatrix(testGraph);

disp('make a comparison')

disp('Adjacency matrix of the test bnet');
disp(testAdjMat);

disp('Adjacency matrix of the result bnet');
disp(resAdjMat);
```

Learning for DBNs

Parameter estimation algorithms for DBNs correspond to the *Expectation Maximization* (EM) algorithms used for learning BNets. Note that the parameters of a model must be tied across time-slices. Thus, sequences of unbounded length may be modelled and the initial state of the dynamic system may be learned independently of the transition matrix. The expected sufficient statistics should be pooled for all the nodes that share the same parameters.

Example 2-9 Learning for DBN

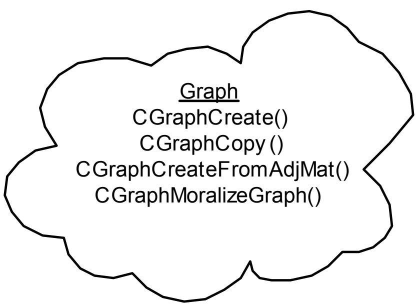
```
%get dynamic model
dbn = ArHMMCreation;

%generate evidences
nSeries = 30;
maxnumSlices = 10;
nts = floor(rand(1,nSeries)*maxnumSlices) +1;
samples = GenerateSamples(dbn, nts);

%create learning procedure
learnEng = CEmLearningEngineDBNCreate(dbn);

%enter evidences
SetData(learnEng, samples);

%start learning procedure
Learn(learnEng);

%results of learning
cpd=GetFactor(dbn, 1);

matMean0 = GetMatrix(cpd, 'matMean', -1, [0]);
matCov0 = GetMatrix(cpd, 'matCovariance', -1, [0]);

matMean1 = GetMatrix(cpd, 'matMean', -1, [1]);
matCov1 = GetMatrix(cpd, 'matCovariance', -1, [1]);

cpd=GetFactor(dbn, 3);

matMean0 = GetMatrix(cpd, 'matMean', -1, [0]);
matCov0 = GetMatrix(cpd, 'matCovariance', -1, [0]);
matWeights0 = GetMatrix(cpd, 'matWeights', 0, [0]);

matMean1 = GetMatrix(cpd, 'matMean', -1, [1])
matCov1 = GetMatrix(cpd, 'matCovaribvance', -1, [1]);
matWeights1 = GetMatrix(cpd, 'matWeights', 0, [1]);
```

Reference Manual

3

Graph

Class CGraph



Class `CGraph` represents the graph structure of a model and performs basic graph algorithms.

CGraphCreate

Creates class object.

```
graph = CGraphCreate( nbrsList, nbrsTypesList );
```

Arguments

graph	Class object.
nbrsList	Cell array of vectors with numbers of node neighbors.
nbrsTypesList	Cell array of cell arrays of neighbor types.

Discussion

This function creates a `CGraph` object using a list of node neighbors.

CGraphCreateFromAdjMat

Creates class object.

```
graph = CGraphCreateFromAdjMat( adjMat );
```

Arguments

graph	Class object.
adjMat	Adjacency matrix.

Discussion

This function creates a `CGraph` object using an adjacency matrix.

CGraphCopy

Creates replica of class object.

```
newGraph = CGraphCopy( graph );
```

Arguments

graph	Class object to be copied.

newGraph	New class object.

Discussion

This function creates a replica of a `CGraph` object.

CGraphMoralizeGraph

Creates class object by moralizing.

```
newGraph = CGraphMoralizeGraph( graph );
```

Arguments

graph	Class object to be moralized.
newGraph	New class object.

Discussion

This function creates a new `CGraph` object by moralizing the source graph.

GetTopologicalOrder

Returns numbers of nodes according to their topological order.

```
order = GetTopologicalOrder( graph );
```

Arguments

graph	Class object.
order	Numbers of nodes.

Discussion

This function returns numbers of nodes according to the order of their topological sorting.

AddEdge

Adds edge to existing graph.

```
AddEdge( startNode, endNode, directed );
```

Arguments

startNode	Starting node of the edge.
endNode	Ending node of the edge.
directed	Edge orientation. The argument shows if the edge is directed. Equals to 1 if the edge is directed, equals to 0 otherwise.

ChangeEdgeDirection

Changes direction of graph.

```
ChangeEdgeDirection( startNode, endNode );
```

Arguments

startNode	Starting node of the edge.
endNode	Ending node of the edge.

Discussion

This function changes the direction of a graph by changing the direction of its edge nodes.

GetNeighbors

Gets all neighbors of given node with orientation vector.

```
[nbrsOut, nbrsTypesOut] = GetNeighbors( nodeNum );
```

Arguments

graph	Class object.
nodeNum	Number of the node whose neighbors are to be found.
nbrsOut	Vector of node neighbors.
nbrsTypesOut	Cell array of vectors with types of node neighbors from *nbrsOut*.

GetNumberOfNeighbors

Returns number of neighbors of given node.

```
n = GetNumberOfNeighbors( graph,  nodeNum );
```

Arguments

graph	Class object.
nodeNum	Number of the node for which the number of neighbors is to be found.
n	Number of neighbours.

GetNumberOfNodes

Returns number of all nodes in graph.

```
n = GetNumberOfNodes( graph );
```

Arguments

graph	Class object.
n	Total number of nodes.

GetNumberOfEdges

Returns number of edges in graph.

```
n = GetNumberOfEdges( graph );
```

Arguments

graph	Class object.
n	Number of edges in the graph.

Discussion

This function returns the number of edges in a graph.

IsCompleteSubgraph

Checks subset of given nodes for completeness.

```
x = IsCompleteSubgraph( graph, subGraph );
```

Arguments

graph	Class object.
subGraph	Subset of nodes.
x	Flag of completeness.

Discussion

This function checks whether a subset of nodes is complete. The function returns 1 if the subset is complete, returns 0 otherwise.

IsChangeAllowed

Returns status flag for graph.

```
x = IsChangeAllowed( graph );
```

Arguments

graph	Class object.
x	Flag of status.

Discussion

This function returns checks if a graph may be changes. The function returns 1 if the change is allowed, returns 0 otherwise.

IsExistingEdge

Returns information on edge existence.

```
x = IsExistingEdge( graph, startNode, endNode );
```

Arguments

graph	Class object.
startNode	Starting node of the edge.
endNode	Ending node of the edge.
x	Flag of egde presence.

Discussion

This function checks if the graph has an edge. The function returns 1 if the edge exists in the graph, returns 0 otherwise.

RemoveEdge

Removes edge from graph.

```
RemoveEdge( graph, startNode, endNode );
```

Arguments

graph	Class object.
startNode	Starting node of the edge.
endNode	Ending node of the edge.

SetNeighbors

Sets neighbors for given node.

```
SetNeighbors( graph, nodeNum, nbrs, nbrsTypes );
```

Arguments

graph	Class object.

nodeNum	Number of the node whose neighbors should be set.
nbrs	Vector of node neighbors.
nbrsTypes	Cell array of node neighbor types.

ProhibitChange

Prohibits any change of `CGraph` *object.*

```
ProhibitChange( graph );
```

Arguments

graph	Class object.

FormCliqueFromSubgraph

Forms clique by connecting all nodes of subgraph.

```
FormCliqueFromSubgraph(  graph, subGraph );
```

Arguments

graph	Class object.
subGraph	Vector of nodes to form a clique.

Discussion

This function connects all nodes of a subgraph with each other so that they form a clique.

GetNumberOfParents

Returns number of parents of node.

```
n = GetNumberOfParents( graph, nodeNum );
```

Arguments

graph	Class object.
nodeNum	Node number.
n	Number of parents.

GetNumberOfChildren

Returns number of children of node.

```
GetNumberOfChildren( nodeNum );
```

Arguments

graph	Class object.
nodeNum	Node number.

IsDirected

Checks if graph is directed.

```
x = IsDirected( graph );
```

Arguments

graph	Class object.

x	Flag of the graph type.

Discussion

This function returns 1, if the graph is directed; returns 0, if the graph has at least one undirected edge.

IsUndirected

Checks if graph is undirected.

```
x = IsUndirected( graph );
```

Arguments

graph	Class object.
x	Flag of the graph type.

Discussion

This function returns 1, if the graph is undirected; returns 0, if the graph has at least one directed edge.

GetAdjacencyMatrix

Returns adjacency matrix.

```
adjMat = GetAdjacencyMatrix( graph );
```

Arguments

graph	Class object.
adjMat	Adjacency matrix.

Discussion

This function returns the adjacency matrix, which corresponds to the graph described by the related `CGraph` object. The adjacency matrix is not stored in the graph. It is formed only when you call the function.

ClearGraph

Clears graph.

```
ClearGraph( graph );
```

Arguments

graph	Class object.

Discussion

This function clears a graph by deleting lists of its node neighbors and sets the number of nodes to zero.

IsIdentical

Checks if two graphs are identical.

```
x = IsIdentical( graph, graphComp );
```

Arguments

graph	Source class object.
graphComp	Graph for comparison.
x	Flag.

Discussion

This function checks if two graphs are identical. Returns 1 if they are identical, returns 0 otherwise.

IsNotIdentical

Checks if two graphs are not identical.

```
x = IsNotIdentical( graph, graphComp );
```

Arguments

graph	Source class object.
graphComp	Graph for comparison.
x	Flag.

Discussion

This function checks if two graphs are not identical. Returns 1 the graphs are not identical, returns 0 otherwise.

GetParents

Returns vector of parents.

```
parents = GetParents( graph, nodeNum );
```

Arguments

graph	Class object.
nodeNum	Number of the node whose parents are to be found.
parents	Vector of numbers of parent nodes.

Discussion

This function returns parents of a node.

GetChildren

Returns vector of children.

```
children = GetChildren( graph, nodeNum );
```

Arguments

graph	Class object.
nodeNum	Number of the node whose children numbers are to be found.
children	Vector of numbers of children nodes.

Discussion

This function returns children of a node.

IsDAG

Checks if graph is DAG

```
x = IsDAG( graph );
```

Arguments

graph	Class object.
x	Flag of object type.

Discussion

This function checks if a graph is a directed acyclic object. The function returns 1, if the graph is a DAG, returns 0 otherwise.

IsTopologicallySorted

Checks if graph is topologically sorted.

```
x = IsTopologicallySorted( graph );
```

Arguments

graph	Class object.
x	Flag of topological sorting.

Discussion

This function returns 1 if the graph is topologically sorted, returns 0 otherwise.

NumberOfConnectivityComponents

Returns number of graph connectivity components.

```
x = NumberOfConnectivityComponents( graph );
```

Arguments

graph	Class object.

Discussion

This function returns the number of connectivity components of the graph. If the graph has more than one connectivity component, the inference engine throws an exception. In case of several connectivity components each of them should be treated as a separate graphical model.

GetConnectivityComponents

Returns connectivity components.

```
decomposition = GetConnectivityComponents( graph );
```

Arguments

graph	Class object.
decomposition	Cell array of connectivity components, represented as vectors.

SetTo

Assigns new value to `CGraph` *object.*

```
rGraph = SetTo( graph );
```

Arguments

graph	Class object.
rGraph	Graph to become identical to the source graph.

Discussion

This function creates a replica of the source graph by assigning to it a new value.

GetAncestry

Finds nodes that lie outside given subgraph but have ancestors inside.

```
closure = GetAncestry( graph, subGraph );
```

Arguments

graph	Class object.
subGraph	Vector of indices of the input subgraph.
closure	Vector of nodes.

Discussion

This function returns indices of nodes that do not lie but have ancestors in the given subgraph.

GetAncestralClosure

Finds nodes that either lie inside or have ancestors in given subgraph.

```
closure = GetAncestralClosure(graph, subGraph );
```

Arguments

graph	Class object.
subGraph	Vector of indices of the input subgraph.
closure	Vector of nodes.

Discussion

This function returns indices of nodes that either lie or have ancestors in the given subgraph.

GetAncestralClosureMask

Finds nodes that either lie inside or have ancestors in given subgraph.

```
closureMask = GetAncestralClosure( graph, subGraph );
```

Arguments

`graph`	Class object.
`subGraph`	Vector of indices of the input subgraph.
`closureMask`	Vector-mask.

Discussion

This function finds nodes that either lie inside or have ancestors in the given subgraph and fills the boolean mask accordingly. The `i`-th element of the `closureMask` is set to true only if the `i`-th node belongs to the ancestral closure.

GetSubgrConnectComponents

Finds plain connectivity components of induced subgraph.

```
decomp = GetSubgrConnectComponents( graph, subGraph );
```

Arguments

`graph`	Class object.

subGraph	Vector of indices of the input subgraph.
decomp	Cell of vectors with indices of decomposition nodes.

Discussion

This function finds plain connectivity components of the induced subgraph and fills in the decomposition output argument accordingly.

GetDConnectionList

Finds nodes `d`-connected to given node.

```
dseparationList = GetDConnectionList( graph, node, separator );
```

Arguments

graph	Class object.
node	Given node.
separator	Vector of nodes that constitute the separator.
dseparationList	Vector of node indices.

Discussion

This function finds nodes `d`-connected to the given node by the given separator. The definition of the `d`-connection runs as in [CDLS]:

node A is `d`-connected to node B if there is a non-blocked trail between them. A trail is blocked if it contains either a node from the separator with the trail edges meeting not head-to-head, or a node that has descendants in the separator with the trail edges meeting head-to-head.

GetDConnectionTable

Finds `d`-connection lists for all nodes of graph.

```
dseparationTable = GetDConnectionTable( graph, separator );
```

Arguments

graph	Class object.
separator	Separator for `d`-separation.
dseparationTable	Cell array of vectors of node indices.

Discussion

Finds `d`-connection lists for all nodes of the graph. The definition for the `d`-connection runs as in [CDLS].

NOTE. *Using* `GetDConnectionTable` *rather than* `GetDConnectionList` *for finding multiple* `d`*-separation lists ensures a much faster result.*

GetReachableSubgraph

Finds nodes reachable from given subgraph if certain pairs of edges are banned.

```
closure = GetReachableSubgraph( graph, subGraph, ban );
```

Arguments

graph	Class object.

subGraph	Given subgraph.
ban	Three-dimensional boolean mask.
closure	Vector of node indices.

Discussion

For every node *i* *ban[i]* should be a conventional two-dimesional boolean array.

The entry *ban[I][J][K]* is true if and only if the pair of edges <*j,i*>, <*i, k*> is banned for acceptable trails.

Function fills in the output vector closure with indices of nodes accessible from the subgraph with an acceptable trail.

GetReachableSubgraphByNode

Finds nodes reachable from given subgraph if certain pairs of edges are banned.

```
closure = GetReachableSubgraphByNode( graph, node, ban );
```

Arguments

graph	Class object.
node	Given node.
ban	Three-dimensional boolean mask.
closure	Vector of node indices.

Discussion

For every node *i* *ban[i]* should be a conventional two-dimesional boolean array.

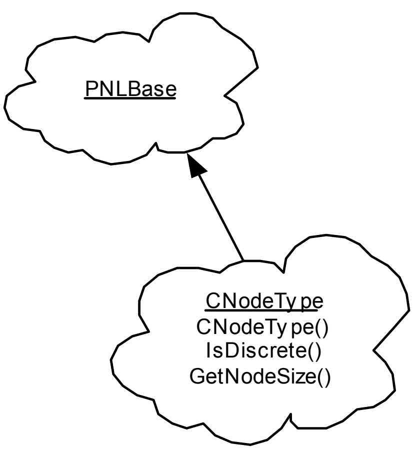
The entry *ban[I][J][K]* is true if and only if the pair of edges <*j,i*>, <*i, k*> is banned for acceptable trails.

Function fills in the output vector closure with indices of nodes accessible from the subgraph with an acceptable trail.

NOTE. *Using this method is not recommended. This method was designed for internal use only in older versions of the* `d`*-connection related methods.*

Node Types

Class CNodeType



Class `CNodeType` provides node types for the model. By default model nodes are binary and discrete.

CNodeType

Creates class object.

```
nt = CNodeType( type, size );
```

Arguments

type	Type of variable. For a discrete variable equals to 1 , for continuous variables equals to 1.
size	For a discrete variable - number of possible values.
	For a continuous variable - dimentionality.
nt	Class object.

IsDiscrete

Returns information on node discreteness.

```
x = IsDiscrete(nt);
```

Arguments

nt	Class object.
x	Flag of discreteness.

Discussion

This function returns 1 if the node is discrete, returns 0 otherwise.

GetNodeSize

Returns node size.

```
x = GetNodeSize( nt );
```

Arguments

nt	Class object.
x	Node size.

Discussion

This function returns the number of possible values if the node is discrete. If the node is continuous, the function returns its dimentionality.

SetType

Sets node type.|

```
SetType( nt, isDiscrete, ndSize, );
```

Arguments

nt	Class object.
isDiscrete	Type of node value. Equals to 1 if the node is discrete, equals to 0 if the node is continuous.
ndSize	Size of the new node.

Discussion

This function determines the type of a given node.

IsIdentical

Compares operands.

```
IsIdentical( nt, ntIn );
```

Arguments

nt	Source class object.
ntIn	Class object for comparison.

Discussion

This function compares two operands. Returns 1 if the operands are equal, returns 0 otherwise.

IsNotIdentical

Compares two operands.

```
IsNotIdentical( nt, ntIn );
```

Arguments

nt	Source class object.
ntIn	Class object for comparison.

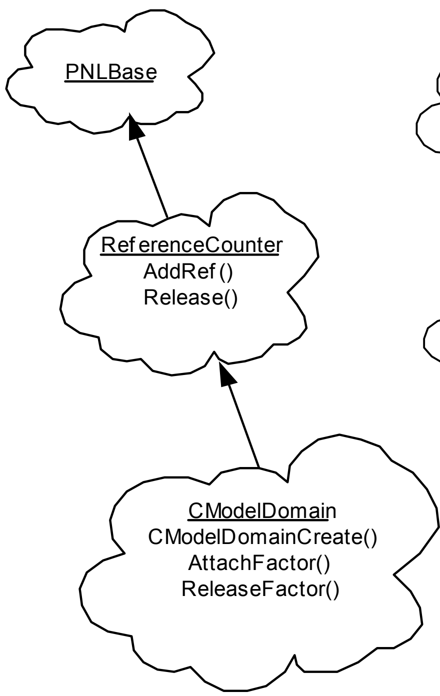
Discussion

This function compares two operands. The function returns 1 if the operands are not equal, returns 0 otherwise.

Model Domain

Model domain is a set of nodes that define a graphical model. A number of graphical models can have one model domain. This object stores information on the types of variables or nodes. For example, you can create a new graphical model using the description of model variables from the model domain.

Class CModelDomain



This class stores information on the types of variables that are used to create a graphical model and on the types of nodes that become observed during inference after evidence is entered. This class also stores temporary `CFactor` objects that are not attached to graphical models.

CModelDomainCreate

Creates class object.

```
md = CModelDomainCreate( variableTypes, variableAssociation );
md = CModelDomainCreate( variableTypes, variableAssociation, creatorOfMD );
```

Arguments

md	Model domain.
variableTypes	Cell array of node types.
variableAssociation	Vector of association of a variable with its type.
creatorOfMD	Graphical model that creates a model domain.

Discussion

This function creates a model domain with different variable types, where the association vector of every variable points at the variable type.

CModelDomainCreateIfAllTheSame

Creates class object.

```
md = CModelDomainCreateIfAllTheSame( nVariables );
md = CModelDomainCreateIfAllTheSame( numVariables, commonVariableType );
md = CModelDomainCreateIfAllTheSame( numVariables, commonVariableType ,
    createrOfMD );
```

Arguments

md	Model domain.
creatorOfMD	Graphical model that creates a model domain.
numVariables	Number of variables in the model domain.

CommonVariableType	Variable type.

Discussion

Three function versions are available.The first version creates a model domain using binary and discrete variables.The second version creates a model domain using any variables of the same type.The third version reates a model domain using a graphical model.

AttachFactor

Attaches factor to model domain.

```
n = AttachFactor(md, factor );
```

Arguments

md	Model domain.
factor	Factor to be attached.

Discussion

This function returns the number that a factor has in the model domain.

ReleaseFactor

Releases factor from model domain.

```
ReleaseFactor( md,factor );
```

Arguments

md	Model domain.
factor	Factor to be released.

IsAFactorOwner

Checks if model domain keeps query factor.

```
isOwner = IsAFactorOwner(md, factor );
```

Argument

md	Model domain.
factor	Factor for checking.

Discussion

The function returns 1 if the model domain is the owner of the query factor, returns 0 otherwise.

GetVariableType

Returns variable type to query variable.

```
vt = GetVariableType( md, varNumber );
```

Arguments

md	Model domain.
varNumber	Number of a variable.

GetVariabeTypes

Returns variable types for query variables.

```
varTypes = GetVariableTypes( md, vars );
```

Arguments

md	Model domain.
vars	Vector of variable numbers.
varTypes	Cell array of variable types.

Discussion

This function returns a cell array of variable types.

GetObsGauVarType

Returns variable types for observed Gaussian variables.

```
vt = GetObsGauVarType( md );
```

Arguments

md	Model domain.

GetObsTabVarType

Returns variable types for observed Tabular variables.

```
vt = GetObsTabVarType( md );
```

Arguments

md	Model domain.

Discussion

This function returns variable types for observed Tabular variables.

GetNumberOfVariableTypes

Returns number of variable types.

```
nvt = GetNumberOfVariableTypes( md );
```

Arguments

md	Model domain.
nvt	Number of variable types.

Discussion

This function returns the number of variable types.

GetVariableTypes

Returns all variable types.

```
varTypes = void CModelDomain::GetVariableTypes( md );
```

Arguments

md	Model domain.
varTypes	Vector of variable types.

Discussion

This function returns the cell array of variable types.

GetNumberVariables

Returns number of variables of model domain.

```
nv = GetNumberVariables( md );
```

Agruments

md	Model domain.

Discussion

This function returns the number of variables that belong to the model domain.

GetVariableAssociations

Returns association to variable types.

```
varAs = GetVariableAssociations( md );
```

Arguments

md	Model domain.
varAs	Vector of associations of variables with their variable types.

Discussion

The function returns to variables their variable types.

GetVariableAssociation

Returns variable association.

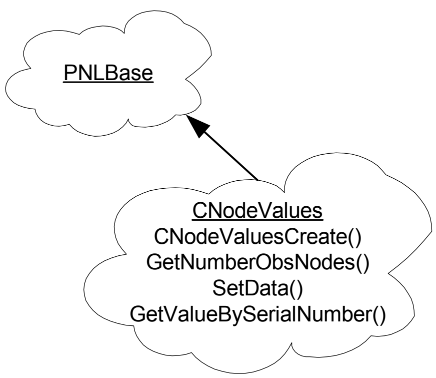
```
varAs = GetVariableAssociation( md, variable );
```

Arguments

md	Model domain.
variable	Number of a variable in the model domain.

Evidences

Class CNodeValues



Class `CNodeValues` is intended for storing values of variables. Values of discrete nodes are represented by integers. Values of continuous nodes are represented by *n* floats, where *n* is *`NodeSize`* of a corresponding node type.

A node can be observed either *potentially* or *actually*. When a node is observed *potentially*, its value is stored at a `CNodeValues` object and it can be made observed *actually*. To make a node *actually* observed, change its observability flag to *`true`*.

To facilitate the inference proceudre, you can create evidence for all nodes in advance and set observability flags of some nodes to `false`. If you need new observed nodes, toggle observability states of hidden nodes, make them observed, and perform inference with them.

Class `CNodeValues` is a superclass for `Model Domain`. It stores information on observed values of certain variables with no indication of varaible numbers in the graphical model. `Model Domain` stores information on the assosiation of variables with the graphical model nodes.

CNodeValuesCreate

Creates class object.

```
nv = CNodeValuesCreate( obsNdsTypes, obsValues );
```

Arguments

nv	Class object.
obsNdsTypes	Node types of observed nodes.
obsValues	Values of observed nodes.

GetValueBySerialNumber

Returns values of nodes.

```
value = GetValueBySerialNumber( nv,SerialNumber );
```

Arguments

nv	Class object.
SerialNumber	Serial number of an observed variable.

Discussion

This function returns the vector of observed node values.

GetNumberObsNodes

Returns number of observed nodes.

```
nNodes = GetNumberObsNodes(nv);
```

Arguments

nv	Class object.
nNodes	Number of nodes.

Discussion

This function returns the number of both potentially and actually observed nodes.

GetObsNodesFlags

Returns observability flags.

```
flags = GetObsNodesFlags( nv );
```

Arguments

nv	Class object.
flags	Flags that show if a variable is observed.

Discussion

This function returns the vector of flags that show if a variable is observed. The function returns 1 if a variable is observed, returns 0 otherwize.

GetRawData

Returns vector of values.

```
values = GetRawData( nv );
```

Arguments

nv	Class object.
values	Vector of values.

Discussion

This function returns the vector of variable values.

SetData

Replaces old values with new values.

```
SetData( nv, data );
```

Arguments

nv	Class object.
data	Vector of new values.

MakeNodeHiddenBySerialNum

Changes state of observability flag.

```
MakeNodeHiddenBySerialNum( nv,serialNum );
```

Arguments

nv	Class object.
serialNum	Number of the node, whose state is to be changed.

Discussion

This function changes the state of an observability flag from observed to hidden.

MakeNodeObservedBySerialNum

Changes observability flag from hidden to observed.

```
MakeNodeObservednBySerialNum(nv,serialNum );
```

Arguments

nv	Class object.
serialNum	Number of the hidden node, whose state is to be changed.

ToggleNodeStateBySerialNumber

Toggles observability type.

```
ToggleNodeStateBySerialNumber( nv, nodeNums );
```

Arguments

nv	Class object.
nodeNums	Serial numbers of the observed variables whose states are to be changed.

Discussion

This function changes the state of variables from potentially observed to actually observed and vice versa.

IsObserved

Checks if variable is observed.

```
flag = IsObserved( nv, nodeNum );
```

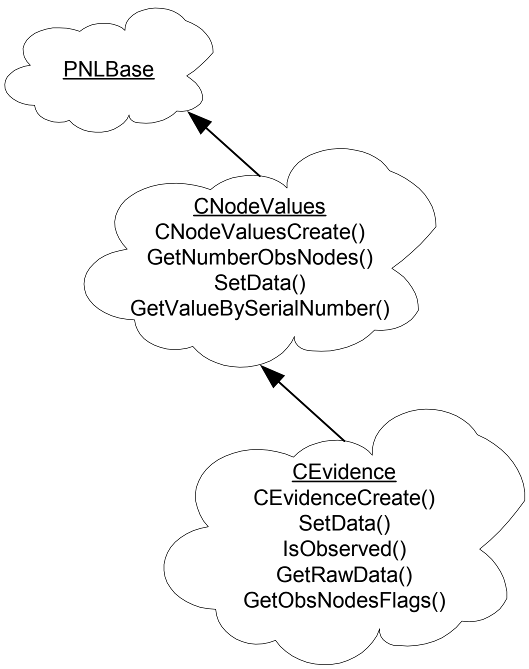
Arguments

nv	Class object.
nodeNum	Serial number of the variable to be checked.
flag	Observability flag.

Discussion

This function checks the state of a variable. The function returns 1 is the variable is observed, returns 0 otherwise.

Class CEvidence



Class `CEvidence` stores information on observed variables of a graphical model. It is a subclass of `Class CNodeValues`, which stores information on types and actual values of variables. Class `CEvidence` stores the array of numbers of observed variables that lie in the graphical model and the functions that are used to retrieve information on observed nodes of the model. These functions establish correspondence between numbers of the model nodes and their serial numbers in a `CNodeValues` object and call functions of a corresponding class.

CEvidenceCreate

Creates class object.

```
CNodeValuesCreate( grModel,obsNodes, obsValues);
```

Arguments

grModel	Graphical model.
obsNodes	Vector of observed nodes in the graphical model or of observed variables it the model domain.
obsValues	Cell array of observed values listed as in `obsNodes`.

CEvidenceCreateByModelDomain

Creates class object using model domain.

```
ev = CEvidenceCreateByModelDomain( md,obsNodes, obsValues );
```

Arguments

md	Model domain.
obsNodes	Vector of observed nodes in the graphical model or of observed variables it the model domain.
obsValues	Array of observed values listed as in `obsNodes`.

CEvidenceCreateByNodeValues

Creates class object using node values.

```
CEvidenceCreateByNodeValues( md,obsNodes, obsValues );
```

Arguments

md	Model domain.
nObsNds	Number of observed nodes or variables.
obsValues	Array of observed values listed as in `obsNodes`.

ToggleNodeState

Toggles node state.

```
ToggleNodeState( ev, nodeNums );
```

Arguments

ev	Class object.
nodeNums	Numbers of the observed nodes whose states are to be changed.

Discussion

This function makes potentially observed nodes actually observed and vice versa.

GetValues

Returns values of nodes.

```
GetValues( nv,  nodeNums );
```

Arguments

nv	Class object.
nodeNums	Serial numbers of observed nodes.

Discussion

The function returns values of observed nodes of the model.

GetAllObsNodes

Returns vector of numbers of observed nodes.

```
obsNodes = GetAllObsNodes(ev);
```

Arguments

ev	Class object.

Discussion

This function returns the vector of numbers of observed nodes.

IsNodeObserved

Checks status of node.

```
flag = IsNodeObserved( ev, nodeNum );
```

Arguments

ev	Class object.
nodeNum	Number of the node for checking.

Discussion

This function returns 1, if the node is observed, retrurns 0 otherwise.

MakeNodeObserved

Changes stete of observation flag .

```
MakeNodeObserved( ev, nodeNum );
```

Arguments

ev	Class object.
nodeNum	Number of the node whose state is to be changed.

Discussion

This function makes a hidden node observed, and throws an exception, if the source node is already observed.

MakeNodeHidden

Changes state of observation flag.

```
flag = MakeNodeHidden( ev, nodeNum );
```

Arguments

ev	Class object.
nodeNum	Number of the node whose state is to be changed.

Discussion

This function makes the observed node hidden and throws an exception, if it is already hidden.

GetObsNodesWithValues

Returns vectors of actually observed nodes and of their values.

```
[pObsNds, ObsValues, nodeTypes] = GetObsNodesWithValues( ev );
```

Arguments

ev	Class object.
ObsNds	Vector of numbers of observed nodes.
ObsValues	Cell array of vectors with raw data of actually observed values. The values are listed as in *ObsNds*.
nodeTypes	Vector with node types for observed nodes. The values are listed as in *ObsNds*.

Discussion

The function returns two vectors: the vector of observed nodes and the vector of their values.

GetModelDomain

Returns model domain.

```
md = GetModelDomain( ev );
```

Arguments

ev	Class object.
md	Model domain.

CEvidenceSaveForStaticModel

Saves evidences to file for statical model.

```
flag = CEvidenceSaveForStaticModel( fName, evVec);
```

Arguments

fName	File name.
evVec	Cell array of evidences.

Discussion

This function saves into a file evidence for a static graphical model. The function returns 1 if the evidence is saved, returns 0 otherwise.

CEvidenceSaveForDBN

Saves evidences to file for DBN.

```
flag = CEvidenceSaveForDBN( fName,  evVec );
```

Arguments

fName	File name.
evVec	Cell array of cell arrays of evidences.

Discussion

This function saves into a file evidence for a dynamic graphical model. The function returns 1 if the evidence is saved, returns 0 otherwise.

CEvidenceLoadForStaticModel

Loads evidence from file.

```
evVec = CEvidenceLoadForStaticModel( fName,md );
```

Arguments

fName	File name.
evVec	Cell array of loaded evidences.
md	Model domain.

Discussion

This function retrieves data from the file and creates evidences for a static graphical model.

CEvidenceLoadForDBN

Loads evidence to file.

```
evVec = CEvidenceLoadForDBN( fname, md );
```

Arguments

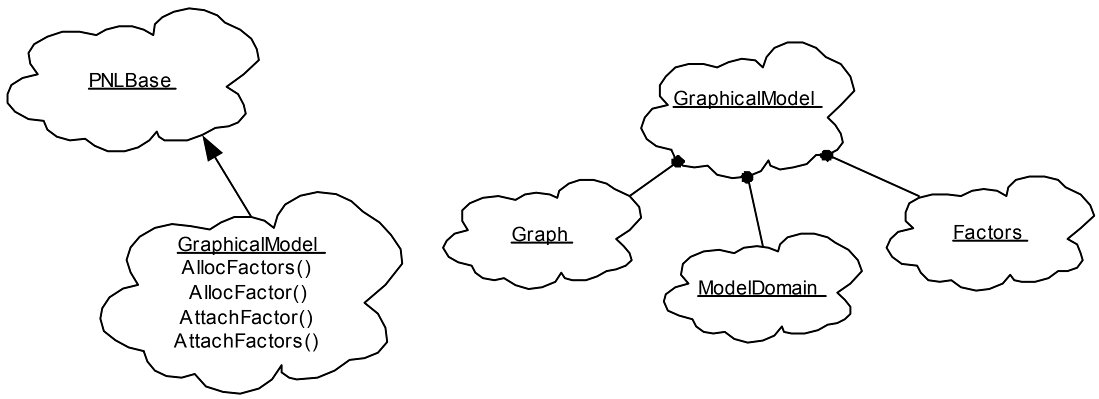
fname	File name.
evVec	Cell array of cell arrays of evidences.
md	Model domain.

Discussion

This function retrieves data from the file and creates evidences for a DBN.

Graphical Models

Class CGraphicalModel



Class `CGraphicalModel` represents a graphical model, which consists of a graph and a number of factors assigned to the graph nodes. Class `CGraphicalModel` is a superclass for [`Class CStaticGraphicalModel`] and [`Class CDynamicGraphicalModel`].

NOTE. *No instances of this class can be created, as the class is abstract.*

AllocFactor

Allocates factor to domain.

```
AllocFactor( grm, number );
```

Arguments

grm	Class object.
number	Vecor of nodes that form the domain.

Discussion

This function allocates a factor to a domain.

AllocFactorByDomainNumber

Allocates factor for domain by domain number.

```
AllocFactorByDomainNumber( grm, domain );
```

Arguments

grm	Class object.
domain	Number of the domain.

Discussion

This function allocates a factor on the domain nodes.

AllocFactors

Allocates space to all factors of model.

```
AllocFactors(grm);
```

Arguments

grm	Class object.

Discussion

This function allocates space to all the factors of the model.

AttachFactor

Attaches factor to model.

```
AttachFactor( grm, pFactor );
```

Arguments

grm	Class object.
factor	Factor to be attached to the model.

Discussion

This function attaches a factor to the model if the factor has an existing domain in terms of the graphical model.

AttachFactors

Attaches new factors and returns old factors.

```
factorsOld = AttachFactors( grm, factors);
```

Arguments

grm	Class object.
factors	`CFactors` object that corresponds to new factors.
factorsOld	Factors to be substituted for by new factors.

Discussion

This function attaches a set of factors stored in a `CFactors` object and returns the set of old factors for destruction.

GetGraph

Returns class object.

```
graph = GetGraph( grm );
```

Arguments

grm	Source class object.
graph	Class object attached to the model.

Discussion

This function returns a class object attached to the model.

GetModelType

Returns type of model.

```
mt = GetModelType( grm );
```

Arguments

grm	Class object.
mt	Model type.

GetNodeType

Returns node type.

```
nt = GetNodeType( grm, nodeNum);
```

Arguments

grm	Class object.
nodeNum	Number of the node whose node type is to be returned.
nt	Node type.

Discussion

This function returns a `CNodeType` object for the specified node number.

GetNodeTypes

Provides access to all node types of model.

```
GetNodeTypes( grm, nodeTypes );
```

Arguments

grm	Class object.
nodeTypes	Cell array of all `CNodeType` objects attached to the model.

GetNumberOfNodes

Returns number of nodes of model.

```
GetNumberOfNodes( grm );
```

Arguments

grm	Class object.

Discussion

This function returns the number of nodes per slice for both static and dynamic graphical models.

GetNumberOfNodeTypes

Returns number of node types of model.

```
nNodeTypes = GetNumberOfNodeTypes( grm );
```

Arguments

grm	Class object.
nNodeTypes	Number of node types.

GetNumberOfFactors

Returns number of factors attached to model.

```
nFactors = GetNumberOfFactors( grm );
```

Arguments

grm	Class object.
nFactors	Number of factors attached.

GetFactor

Returns class object by domain number.

```
factor = GetFactor( grm, domainNumber );
```

Arguments

grm	Class object.
domainNumber	Number of domain for which a factor is to be found.

Discussion

This function returns a class object using a specified domain number.

GetFactorsIntoVector

Returns all factors attached to subset of nodes.

```
factors = GetFactorsIntoVector( grm, nodes );
```

Arguments

grm	Graphical model.
nodes	Vector of nodes for which attached factors are to be found.
factors	Cell array of factors.

Discussion

This function returns all factors attached to nodes. Several factors may be attached to one subset of nodes if the latter is common for a number of domains.

GetModelDomain

Returns model domain.

```
md = GetModelDomain( grm );
```

Arguments

grm	Class object.
md	Model domain.

IsValid

Checks validity of graphical model.

```
[flag, descr]=IsValid( grm );
```

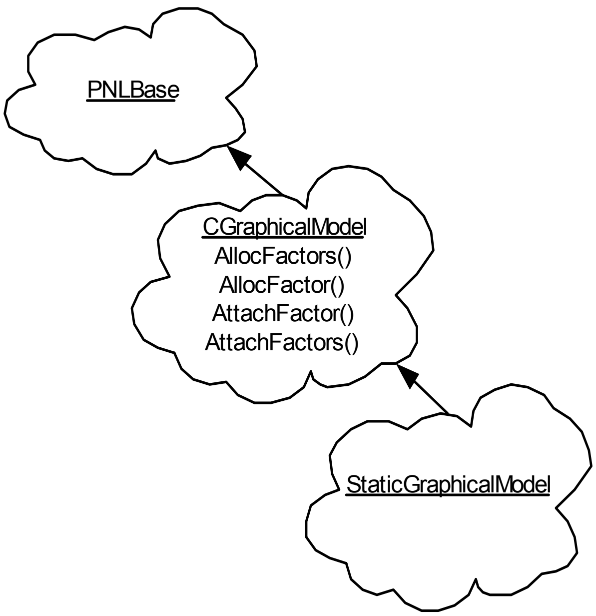
Arguments

grm	Class object.
flag	Flag of validity.
descr	Error message.

Discussion

This function checks the validity of a graphical model. Returns 1 if the model is valid, returns 0 otherwise.

Class CStaticGraphicalModel



`CStaticGraphicalModel` is a superclass for two classes: `Class CBNet` and `Class CMNet`.

NOTE. *No instances of this class can be created, as the class is abstract.*

IsValidAsBaseForDynamicModel

Checks validity of model for creation of dynamic model.

```
[flag, descr] = IsValidAsBaseForDynamicModel( grm );
```

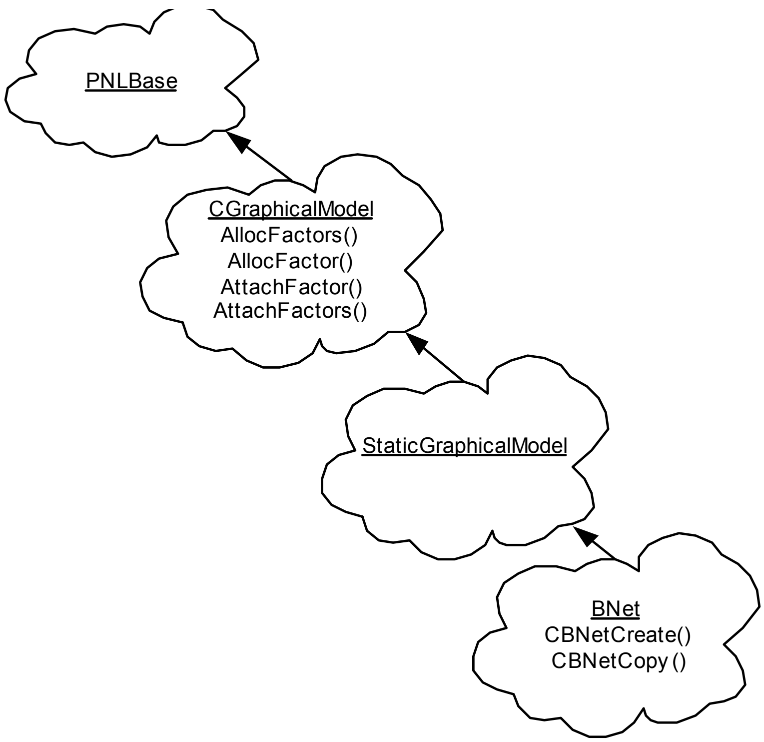
Arguments

grm	Class object.
descr	Error message.
flag	Flag of validity.

Discussion

This function checks if the model is valid for the creation of a dynamic graphical model.

Class CBNet



CBNetCreate

Creates class object.

```
BNet = CBNetCreate( numberOfNodes, nodeTypes, nodesAssociation, graph );
```

Arguments

numberOfNodes	Number of nodes.
nodeTypes	Cell array of node types.
nodesAssociation	Vector for nodes association with node types.
graph	Graph structure of the model.

CBNetCreateByModelDomain

Creates class object.

```
CBNetCreateByModelDomain( graph, md );
```

Arguments

graph	Graph structure of the model.
md	Model domain.

CBNetCreateWithRandomMatrices

Creates `CBNet` object with random matrices.

```
CBNetCreateWithRandomMatrices( graph, md );
```

Arguments

graph	Graph structure.
md	Model domain.

Discussion

This function creates a `BNet` object with dense random matrices.

CBNetCopy

Creates new object by copying.

```
CBNetCopy( BNet );
```

Arguments

BNet	Class object to be copied.

Discussion

This function creates a new `CBNet` object by copying the source object.

ConvertToSparse

Converts object with dense matrices into object with sparse matrices.

```
BNetSparce = ConvertToSparse( BNet );
```

Arguments

BNet	Class object.

Discussion

This function converts a `CBNet` object with dense matrices into a `CBNet` object with sparse matrices.

ConvertToDense

Converts object with sparse matrices into object with dense matrices.

```
BNetDense = ConvertToDense( BNet );
```

Arguments

`BNet`	Class object.

Discussion

This function converts a `BNet` object with sparse matrices into a `BNet` object with dense matrices.

CreateTabularCPD

Creates tabular CPD.

```
CreateTabularCPD( BNet, childNodeNumber, matrixData );
```

Arguments

`BNet`	Class object.
`childNodeNumber`	Factor number.
`matrixData`	Matrix with data.

Discussion

This function creates a tabular CPD using given data.

FindMixtureNodes

Finds numbers of mixture nodes.

```
mixNds = FindMixtureNodes( BNet );
```

Arguments

BNet	Class object.
mixNds	Vector of mixture nodes.

Discussion

This function finds numbers of mixture nodes of a mixture Gaussian distribution.

GenerateSamples

Generates random evidences for `BNet` *given evidence.*

```
evidences = GenerateSamples( BNet, nSamples, );
evidences = GenerateSamples( BNet, nSamples, ev );
```

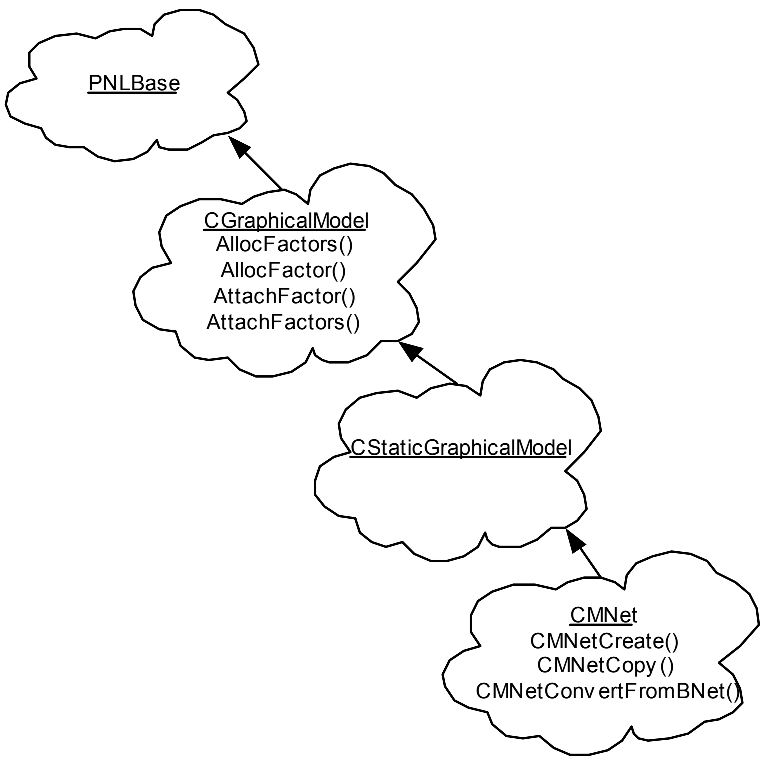
Arguments

BNet	Class object.
evidences	Cell array of evidences to be created.
nSamples	Number of samples.
ev	Given evidence.

Discussion

This function generates samples from a static graphical model.

Class CMNet



CMNetCreate

Creates class object.

```
MNet = CMNetCreate( numberOfNodes, nodeTypes, nodesAssociation, cliques );
```

Arguments

MNet	Class object.
numberOfNodes	Number of nodes in the model.
nodeTypes	Vector of node types.
nodesAssociation	Vector of assosiation of nodes with their types.
cliques	Cliques.

CMNetCreateByModelDomain

Creates class object by model domain.

```
MNet = CMNetCreateByModelDomain( cliques, MD );
```

Arguments

MNet	Class object.
cliques	Cliques.
MD	Model domain.

CMNetCreateWithRandomMatrices

Creates object with random matrices.

```
MNet = CMNetCreateWithRandomMatrices(cliques,MD);
```

Arguments

MNet	Class object.
cliques	Cliques.
MD	Model domain.

Discussion

This function creates a class object with dense random matrices. Covariance matrices of the Gaussian distribution are matrix units.

GetClique

Returns clique nodes.

```
clq = GetClique( MNet, clqNum );
```

Arguments

MNet	Markov network.
clqNum	Number of a clique.
clq	Vector of clique nodes.

CMNetConvertFromBNet

Creates class object by converting input `BNet`*.*

```
MNet = CMNetConvertFromBNet( BNet );
```

Arguments

MNet	Class object.
BNet	Bayesian network.

Discussion

This function converts the input `BNet` object into a `MNet` object.

CMNetConvertFromBNetUsingEv

Creates object by converting input `BNet` *using given evidence.*

```
MNet = CMNetConvertFromBNetUsingEv( BNet, ev );
```

Arguments

MNet	Class object.
BNet	Bayesian network.
ev	Evidence.

Discussion

This function converts the input `BNet` object into a `MNet` object using evidence.

CMNetCopy

Creates object by copying input `MNet`.

```
MNetNew = CMNetCopy( MNet );
```

Arguments

MNet	Markov network to be copied.
MNetNew	New object of the class.

Discussion

This function creates a new object of the class by copying the input `MNet`.

CreateTabularPotential

Allocates factor and creates matrix.

```
CreateTabularPotential( MNet, domain, data );
```

Arguments

MNet	Class object.
domain	Array of nodes.
data	Matrix with data.

Discussion

This function allocates a factor and creates a new matrix with the given data.

ComputeLogLik

Computes logarithm of likelihood.

```
LogLik = ComputeLogLik( MNet, ev );
```

Arguments

MNet	Class object.
ev	Evidence.
LogLik	Logarithm of likelihood.

Discussion

This function computes the logarithm of likelihood.

GetClqsNumsForNode

Specifies numbers of cliques with node.

```
clqs = GetClqsNumsForNode( MNet, node );
```

Arguments

MNet	Class object.
node	Node number.
clqs	Vector of cliques that contain *node*.

Discussion

This function specifies numbers of cliques that contain a given node.

GetNumberOfCliques

Returns number of cliques of model.

```
nClq = GetNumberOfCliques( MNet );
```

Arguments

nClq	Number of cliques.
MNet	Class object.

GenerateSamples

Generates random evidence from `MNet`.

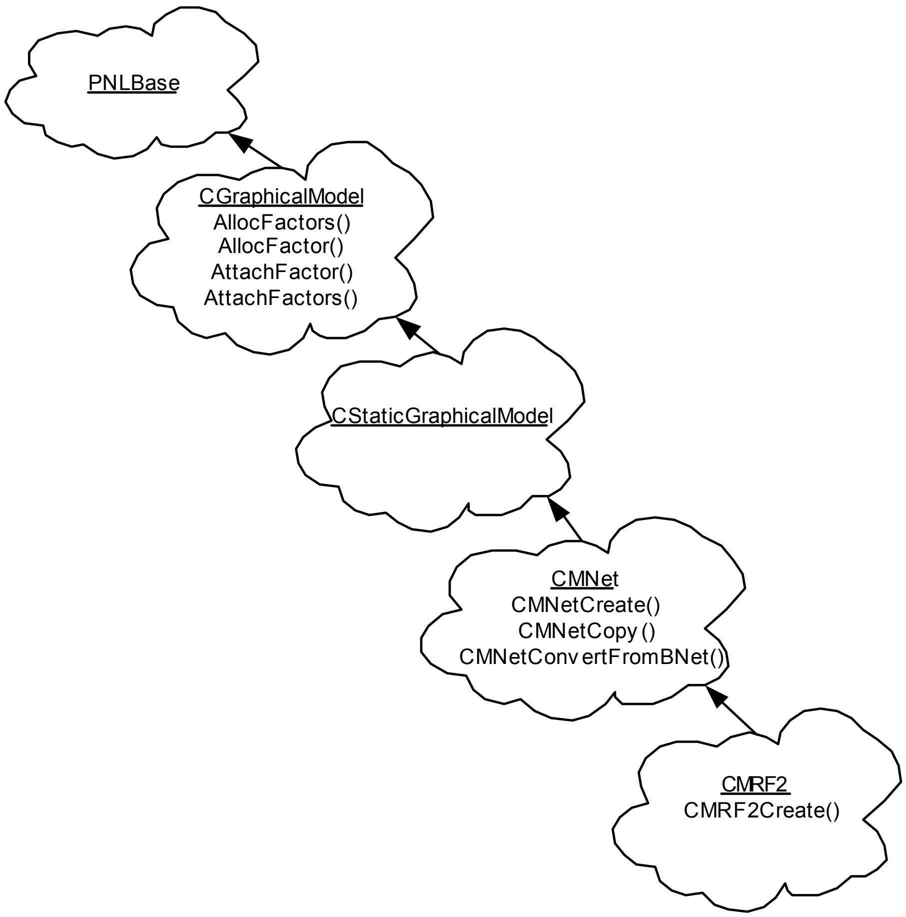
```
evidences = GenerateSamples( MNet, nSamples );
```

```
evidences = GenerateSamples( MNet, nSamples, ev );
```

Arguments

MNet	Class object.
evidences	Cell array of evidences to be created.
nSamples	Number of samples.
ev	Given evidence.

Class CMRF2



`Class CMNet` is a superclass for class `CMRF2`, which represents a pairwise Markov network. Class `CMRF2` implements `CMNet` virtual functions, with objects whose cliques contain two nodes.

CMRF2Create

Creates class object.

```
MRF2 = CMRF2Create( numberOfNodes, nodeTypes,nodeAssociation, cliques );
```

Arguments

MRF2	Class object.
cliques	Cell array of vectors with clique nodes.
numberOfNodes	Number of nodes.
nodeTypes	Cell array of node types.
nodeAssociation	Vector of node association.

CMRF2CreateByModelDomain

Creates class object using model domain.

```
MRF2 = CMRF2CreateByModelDoamin( cliques, MD );
```

Arguments

MRF2	Class object.
cliques	Cliques.
MD	Model domain.

CMRF2CreateWithRandomMatrices

Creates class object using random matrices.

```
MRF2 = CMRF2CreateWithRandomMatrices( cliques,MD );
```

Arguments

MRF2	Class object.
cliques	Cliques.
MD	Model domain.

Class CFactorGraph

A factor graph is the graphical representation of a factorized distribution. All factors of the distribution are represented by factor-nodes, which are connected to variable-nodes of the factor domain. A factor graph is the resulting graph of the distribution.

Class `CFactorGraph` is a graphical model that consists of a set of factors. The set of factors has its probability distribution. All the factors are potentials.

CFactorGraphCreate

Creates class object.

```
fGraph = CFactorGraphCreate( MD, numFactors );
```

Arguments

MD	Model domain.
numFactors	Number of factors in the factor graph.

Discussion

This function creates a factor graph with several allocated factors.

CFactorGraphCreateByFactors

Creates class object.

```
fGraph = CFactorGraphCreateByFactors( MD, factors );
```

Arguments

MD	Model domain.
factors	`CFactors` object whose factors describe a factor graph object.

Discussion

This function creates a factor graph from all factors of the model domain.

CFactorGraphCopy

Creates replica of input object.

```
fGraph = CFactorGraphCopy( FG );
```

Arguments

FG	Input class object.
fGraph	New class object.

Discussion

This function creates a new `CFactorGraph` object by copying the input object.

Shrink

Creates factor graph by shrinking all potentials of given factor.

```
fGraphNew = Shrink( fGraph,evidence );
```

Arguments

fGraph	Class object.
evidence	Given evidence.
fGraphNew	New object of the class.

Discussion

This function creates a factor graph with given evidence by shrinking all the potentials of the given factor.

GetNumFactorsAllocated

Returns numbers of allocated factors.

```
nbrsFactors = GetNumFactorsAllocated();
```

Arguments

nbrsFactors	Vector of numbers of allocated factors.

CFactorGraphConvertFromBNet

Creates class object by converting `BNet` *object.*

```
fGraph = CFactorGraphConvertFromBNet( BNet );
```

Arguments

BNet	Object to be converted.

Discussion

This function creates a `CFactorGraph` object by converting the given `BNet` object.

CFactorGraphConvertFromMNet

Creates class object by converting `MNet` *object.*

```
fGraph = CFactorGraphConvertFromMNet( MNet );
```

Arguments

MNet	Object to be copied.

Discussion

This function creates a `CFactorGraph` object by converting a `MNet` object.

IsValid

Checks validity of function.

```
[flag, description ] = IsValid( MNet);
```

Arguments

description	Error message.

Discussion

This function checks if the object is valid.

GetNbrFactors

Returns factors neighboring to given node.

```
nbrsFactors = GetNbrFactors( fGgaph, node  );
```

Arguments

fGgaph	Factor graph.
node	Node.
nbrsFactors	Vector of numbers of the factors that neighbor with the node factors.

Discussion

This function returns factors neighboring to the given node. A factor is called neighboring to the node if the latter lies in the factor domain.

GetNumNbrFactors

Returns number of factors neighboring with given node.

```
nf = GetNumNbrFactors( fGraph, node );
```

Arguments

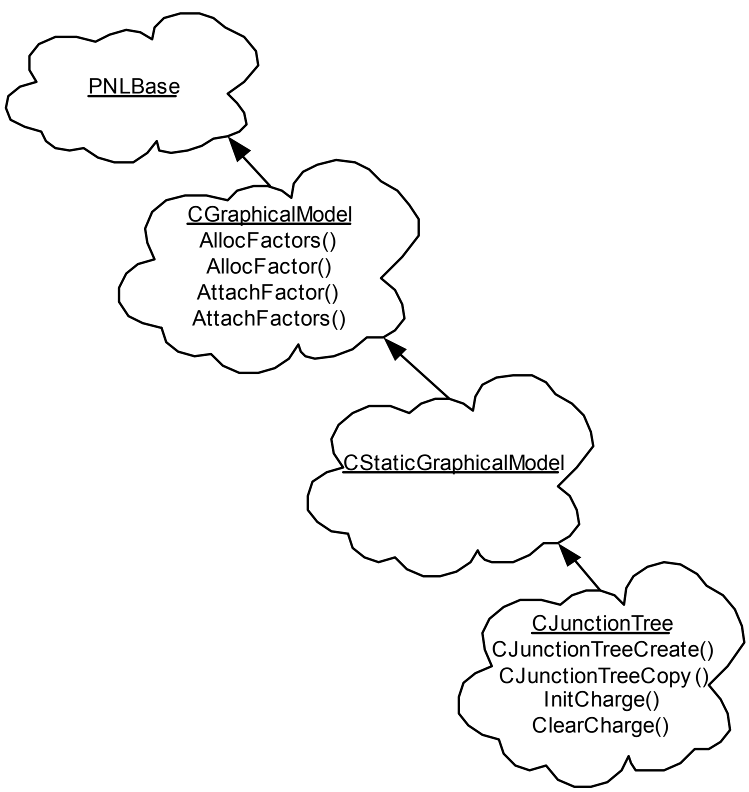
fGraph	Factor graph.

`node`	Number of the node.
`nf`	Number of neighboring factors.

Discussion

This function returns the number of factors neighboring with a given node.

Class CJunctionTree



This class represents the structure of a Junction tree. It is used in the Junction Tree Inference Engine for internal local computations. A class object is created on the creation of `JTreeInfEngine`.

CJunctionTreeCreate

Creates Junction tree.

```
JTree = CJunctionTreeCreate( grModel );
JTree = CJunctionTreeCreate( grModel, subGrToConnect );
```

Arguments

grModel	Graphical model from which the tree is to be constructed.
subGrToConnect	Subgraphs you want to appear in the tree.
JTree	Class object.

CJunctionTreeCopy

Creates replica of input Junction tree.

```
JTreeNew = CJunctionTreeCopy( JTree );
```

Arguments

JTree	Class object to be copied.

Discussion

This function creates a class object through copying the input Junction tree.

GetNodePotential

Returns potential defined of Junction tree clique.

```
pot = GetNodePotential( JTree, nodeNum );
```

Arguments

JTree	Class object.
nodeNum	Number of the clique in the Junction tree.

Discussion

This function returns the potential of a junction tree clique.

GetSeparatorPotential

Returns potential defined for separator between two cliques.

```
pot = GetSeparatorPotential( JTree, firstClqNum, secondClqNum );
```

Arguments

JTree	Class object.
firstClqNum	Number of the first clique.
secondClqNum	Number of the second clique.

Discussion

This function returns the potential of the separator between two cliques.

InitCharge

Initializes charge for Junction tree.

```
InitCharge(JTree, grModel, evidence );
InitCharge(JTree, grModel, evidence, sumOnMixtureNode );
```

Arguments

JTree	Class object.
grModel	Graphical model.
evidence	Evidence.
sumOnMixtureNode	Flag showing if the distribution for the mixture node is to be computed during inference.

Discussion

This function initializes charge for a Junction tree. The Junction tree charge comprises both potentials for cliques and potentials for separators.

ClearCharge

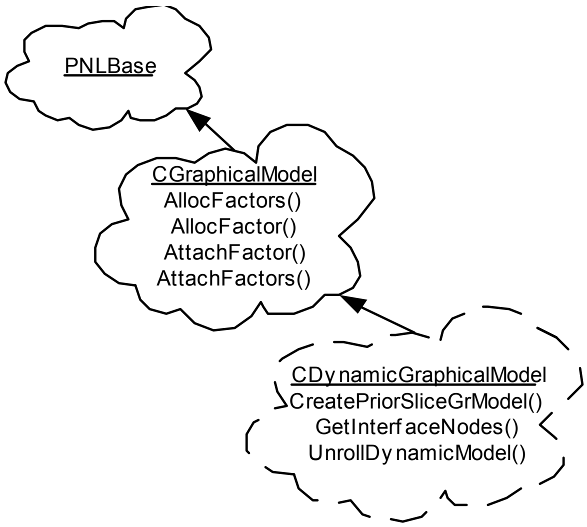
Clears charge.

```
ClearCharge( JTree );
```

Arguments:

JTree	Class object.

Class CDynamicGraphicalModel



`CDynamicGraphicalModel` is a superclass for all classes that work with dynamic graphical models.

CreatePriorSliceGrModel

Creates static graphical model.

```
grModel = CreatePriorSliceGrModel( dModel );
```

Arguments

dModel	Class object.
grModel	Static graphical model.

Discussion

This function creates a static graphical model corresponding to the prior slice of the dynamic graphical model.

UnrollDynamicModel

Creates static graphical model by unrolling dynamic graphical model.

```
grModel = UnrollDynamicModel( dModel, numOfSlices );
```

Arguments

dModel	Class object.
numOfSlices	Number of slices.
grModel	Static graphical model.

Discussion

This function unrolls a dynamic graphical model as a number of slices and thus constructs a static graphical model.

GetInterfaceNodes

Returns numbers of interface nodes.

```
interfaceNds = GetInterfaceNodes( dModel );
```

Arguments

dModel	Class object.
interfaceNds	Array of interface nodes.

Discussion

This function returns numbers of interface nodes.

GetStaticModel

Returns static graphical model.

```
grModel = GetStaticModel(dModel);
```

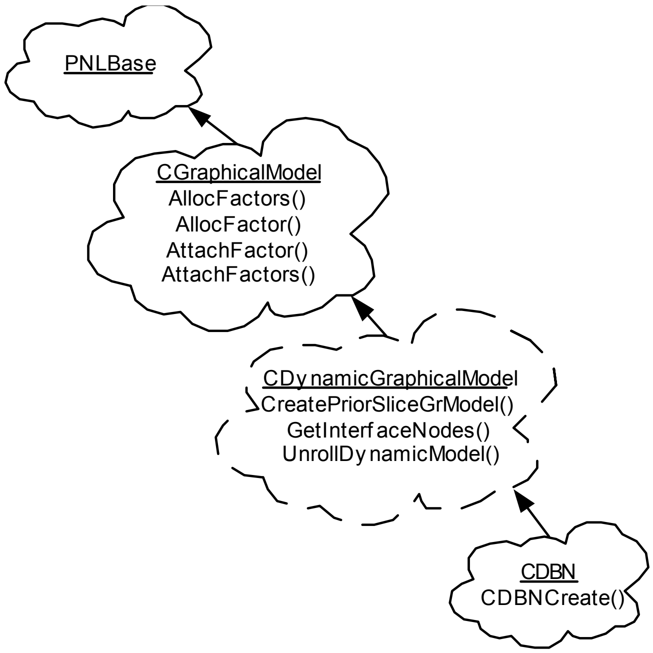
Arguments

dModel	Class object.
grModel	Static graphical model.

Discussion

This member function returns the static graphical model which was used for the creation of a dynamic graphical model.

Class CDBN



Class `CDBN` is a subclass of `Class CDynamicGraphicalModel`. It is intended for the implementation of virtual functions of the parent class.

CDBNCreate

Creates class object.

```
DBN = CDBNCreate( grModel );
```

Arguments

grModel	BNet that represents a DBN unrolled for first two time-slices.

GenerateSamples

Generates samples from DBN.

```
evidences = GenerateSamples( DBN, nSlices );
```

Arguments

DBN	Class object.
nSlices	Vector of the number of slices for which evidence is generated.
evidences	Cell array of cell arrays of generated evidence.

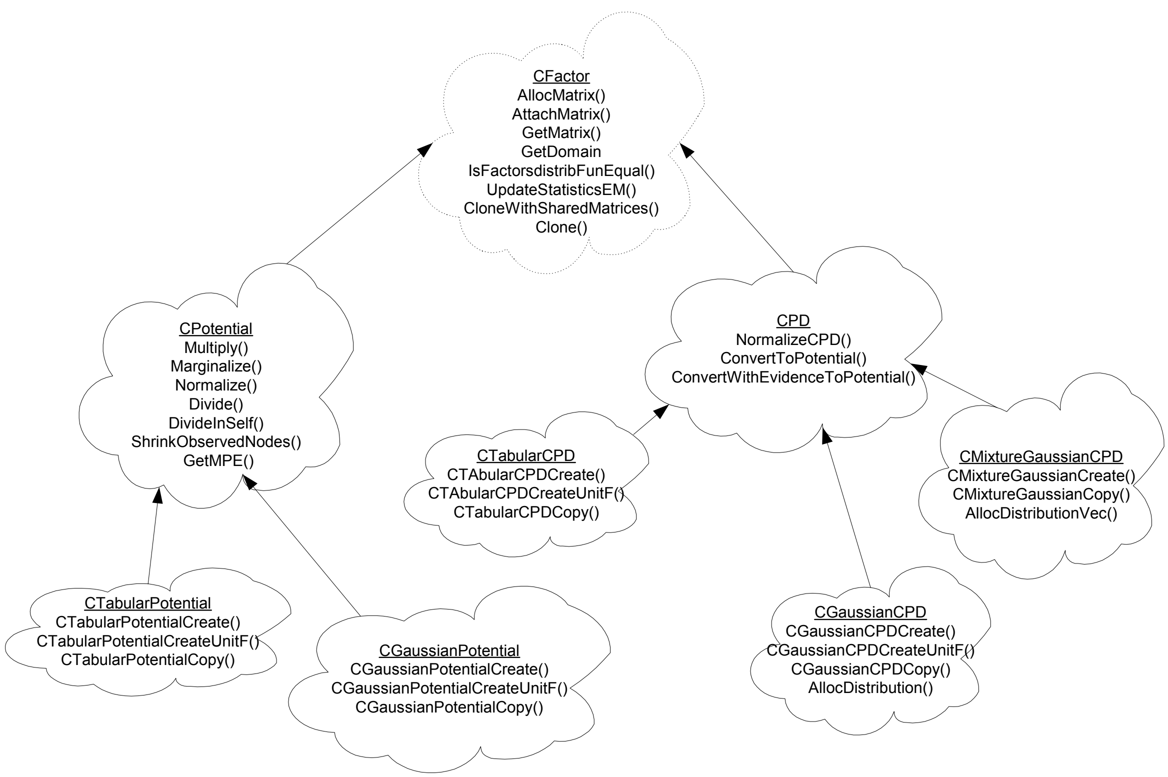
Discussion

This function generates samples from a dynamic graphical model.

Factors

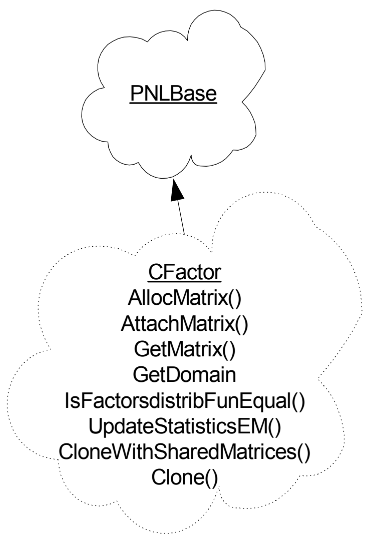
Class `CFactors` and its subclasses `CPotential`, `CCPD`, `CTabularPotential`, `CGaussianPotencial`, `CTabularCPD` and `CGaussianCPD` store the factor domain - factors of a grapical model related to one or a number of nodes. `CPotential`, `CCPD`, and `CTabularFactor` subclasses are abstract. For the hierarchy of the classes see Figure 3-1.

Figure 3-1 Class `CFactors` and its subclasses



This class stores joint probability distribution for a `CFactor` object, and conditional distribution for a `CCPD` object. Both types of distribution are implemented with `Class CMultiDMatrix` objects. The number of objects for discrete and continuous distributions may be different. Functions, specific of a distribution type, are stored in the internal class `CData`.

Class CFactor



AttachMatrix

Attaches matrix to factor.

```
AttachMatrix( factor, matrix, mType );
AttachMatrix( factor, matrix, mType, matrixNum );
AttachMatrix( factor, matrix, mType, matrixNum, discrParentValuesIndices);
```

Arguments

matrix	`CMultiDMatrix` object.
mType	Matrix type.
MatrixNum	Number of the matrix, if several matrices of a given type are associated with the factor. The argument is omitted, if only one matrix is involved.
discrParentValuesIndices	Vector of values of discrete parents.

Discussion

This function enters data into a matrix and associates the matrix with the factor.

GetFactorType

Returns factor type.

```
factorType = GetFactorType(factor);
```

Arguments

factor	Class object.
factorType	Type of factor.

Discussion

This function returns the type of an input factor. It may be either *`ptFactor`* or *`ptCPD`*.

GetDistributionType

Returns distribution type.

```
distrType = GetDistributionType( factor );
```

Arguments

factor	Class object.
distrType	Type of distribution.

Discussion

This function returns the type of distribution. It may be `dtTabular`, `dtGaussian` or `dtCondGaussian`.

GetDomain

Returns factor domain.

```
domain = GetDomain( factor, domainSize, domain );
```

Arguments

factor	Class object.
domainSize	Vector of the domain size.
domain	Vector of numbers that specify serial numbers of the graphical model nodes associated with the factor domain.

GetDomainSize

Returns size of factor domain.

```
domainSize = GetDomainSize(factor);
```

Arguments

factor	Class object.
domainSize	Vector of the domain size.

Discussion

This function returns the number of nodes associated with the factor.

GetMatrix

Returns matrix attached to factor.

```
matrix = GetMatrix( factor, mType );
matrix = GetMatrix( factor, mType, matrixNum );
matrix = GetMatrix( factor, mType, matrixNum, iscrParentValuesIndices );
```

Arguments

factor	Class object.
mType	Type of matrix.
matrixNum	Number of matrix. The agrument is used when there are several matrices of a given type. If there is only one matrix of a given type, the argument is omitted.
discrParentValuesIndices	Vector of values of discrete parents.

Discussion

This function returns the matrix by matrix type if the matrix of this type has been attached to the factor. Matrix may be of the following types: `matTable`, `matMean`, `matCov`, `matWeights`, `math`, and `matK`.

IsValid

Checks factor validity.

```
[ flag, discription ] =IsValid( factor );
```

Arguments

factor	Class object.
discription	Error message.

Discussion

This function checks martix validity.The function returns 1 if the matrices are allocated, returns 0 otherwise.

IsFactorsDistribFunEqual

Compares distributions.

```
flag = IsFactorsDistribFunEqual( factor,factorToComp, eps ) const;
flag = IsFactorsDistribFunEqual( factor,factorToComp, eps, withCoeff );
```

Arguments

factor	Class object.
factorToComp	Factor for comparison.
eps	Accuracy of comparison.
withCoeff	Flag of the comparison type. To compare normalizing constants for Gaussian and Conditional Gaussian distribution, set the flag to 0.

Discussion

This function compares distributions. The function returns 1, if the factor distributions are of the same size, type and have the same floating point matrices to represent them.

TieDistribFun

Assignes input factor distribution to object.

```
TieDistribFun( factor, factorToTie );
```

Arguments

factor	Class object.
factorToTie	Factor distribution to be assigned.

Discussion

This function assignes a distribution to an object only if both factors are of the same type. If the factors are of different types, the function throws an exception.

IsDistributionSpecific

Checks whether distribution is specific.

```
flag = IsDistributionSpecific( factor );
```

Arguments

factor	Class object.

Discussion

This function checks whether a distribution is specific or not and returns:

- 0 - the distribution is full (Tabular, Gaussian or Conditional Gaussian, non-delta, non-uniform, non-mixed, invalid. If the distribution is invalid, call `IsValid` function to check the status).
- 1 - the distribution is uniform (has no attached matrices). A special flag shows that the distribution is uniform.

- 2 - the distribution is a delta function (has only a mean matrix) To check if the distribution is valid, call `IsValid` function.
- 3 - the distribution is mixed (product of the multiplication of a distribution by a delta function in some dimensions).

GenerateSample

Generates sample from factor.

```
evidence = GenerateSample( factor, maximize );
```

Arguments

factor	Class object.
evidences	Generated evidence.
maximize	Flag of maximization.

Discussion

This function generates a sample from the factor using the current evidence data.

CFactorCopyWithNewDomain

Creates class object.

```
factorNew = CFactorCopyWithNewDomain( factor, domain, modelDomain );
factorNew = CFactorCopyWithNewDomain( factor, domain, modelDomain,obsIndices);
```

Arguments

factor	Class object.
domain	Node numbers in the domain.
modelDomain	New model domain.

obsIndices	Vector of indices of the observed nodes.

Discussion

This function creates a class object through copying the source class object and changing its domain.

Clone

Creates replica of object.

```
factorNew = Clone( factor );
```

Arguments

factor	Source class object.
factorNew	New class object.

CloneWithSharedMartices

Creates replica of factor.

```
factorNew = CloneWithSharedMatrices( factor );
```

Arguments

factor	Source class object.
factorNew	New class object.

Discussion

This function creates a class object so that the new factor shares its matrices with the source factor.

CreateAllNecessaryMatrices

Creates matrices necessary to make factor valid.

```
CreateAllNecessaryMatrices( factor, typeOfMatrices );
```

Arguments

factor	Class object.
typeOfMatrices	Flag of the generation type. For random matrices the flag equals to 1.

Discussion

This function creates matrices that are needed to make a factor valid. Covariance matrix for the Gaussian distribution is unitary.

ChangeOwnerToGraphicalModel

Releases model domain from factor.

```
ChangeOwnerToGraphicalModel();
```

IsOwnedByModelDomain

Checks if factor is owner of model domain.

```
flag = IsOwnedByModelDomain();
```

GetModelDomain

Returns model domain.

```
md = GetModelDomain( factor );
```

Arguments

factor	Class object.
md	Model domain.

ConvertToSparse

Converts factor distribution with dense matrices into distribution function with sparce matrices.

```
ConvertToSparse( factor );
```

Arguments

factor	Class object.

Discussion

This function converts a factor distribution function with dense matrices into a factor distribution function with sparse matrices.

ConvertToDense

Converts factor distribution with sparce matrices into distribution with dense matrices.

```
ConvertToDense(factor);
```

Arguments

factor	Class object.

Discussion

This function converts a factor distribution function with sparse matrices into a factor distribution function with dense matrices.

IsSparse

Checks if distribution matrices are sparse.

```
flag = IsSparse( factor );
```

Arguments

factor	Class object.
flag	Flag of status.

IsDense

Checks if distribution matrices are dense.

```
flag = IsDense(factor);
```

Arguments

factor	Class object.
flag	Flag of status.

GetObsPositions

Returns observed positions of domain.

```
obsPos = GetObsPositions( factor  );
```

Arguments

factor	Class object.
obsPos	Vector of observed positions in the domain.

MakeUnitFunction

Transforms distribution function into unit function distribution.

```
MakeUnitFunction( factor );
```

Arguments

factor	Class object.

ConvertStatisticToPot

Creates potential using statistical data of distribution function.

```
potential = ConvertStatisticToPot( factor, numOfSamples );
```

Arguments

factor	Class object.

numOfSamples	Number of samples.
potential	Potential.

UpdateStatisticsEM

Collects statistical data.

```
UpdateStatisticsEM( factor, infData );
UpdateStatisticsEM( factor, infData, evidence );
```

Arguments

factor	Class object.
infData	Inference result.
evidence	`CEvidence` object.

Discussion

This function updates statistical data.

UpdateStatisticsML

Collects statistical data.

```
StatisticalDataML( factor, evidences );
```

Arguments

factor	Class object.
evidences	Cell array of evidences.

SetStatistics

Sets statistical data.

```
SetStatistics( factor, mat,  matrixType );
SetStatistics( factor, mat,  matrixType, parentsComb  );
```

Arguments

factor	Class object.
mat	Matrix with statistical data.
matrixType	Type of matrix.
parentsComb	Combination of discrete parents.

Discussion

This function sets statistical data for learning procedures.

ProcessingStatisticalData

Updates factor distribution function after collecting statistical data.

```
ProcessingStatisticalData( factor, numEvidences );
```

Arguments

factor	Class object.
numEvidences	Number of evidences.

Discussion

This function performs factor estimation and updates a factor distribution function with newly acquired statistical data.

GetLogLik

Calculates likelihood of input evidence.

```
logLik = GetLogLik( factor, ev );
```

Arguments

factor	Class object.
ev	Evidence.
logLik	Logarithm of likelihood.

Discussion

This function returns the logarithm of likelihood of the input data.

AreThereAnyObsPositions

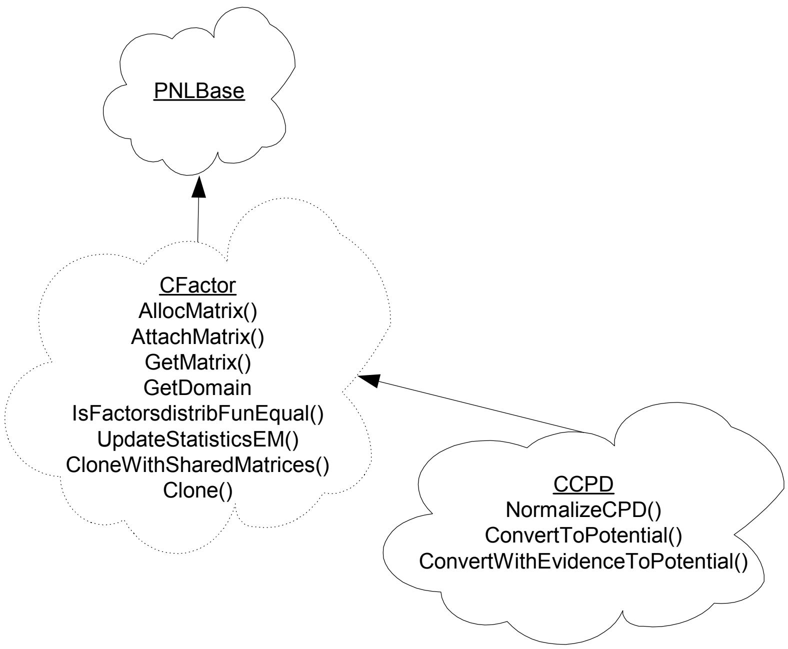
Checks if factor has observed nodes.

```
flag = AreThereAnyObsPositions( factor );
```

Arguments

factor	Class object.
flag	Flag of observed positions.

Class CCPD



ConvertToPotential

Creates `CPotential` *object by converting class object.*

```
potential = ConvertToPotential( CPD );
```

Arguments

CPD	Class object.
potential	Potential.

Discussion

This function converts a `CCPD` object into a `CPotential` object and returns `CPotential` object.

ConvertWithEvidenceToPotential

Converts `CPD` *to* `CPotential` *using evidence.*

```
pot = ConvertWithEvidenceToPotential( CPD, ev );
pot = ConvertWithEvidenceToPotential( CPD, ev, flagSumOnMixtureNode );
```

Arguments

CPD	Class object.
ev	Given evidence.
flagSumOnMixtureNode	Flag of mixture node addition.

Discussion

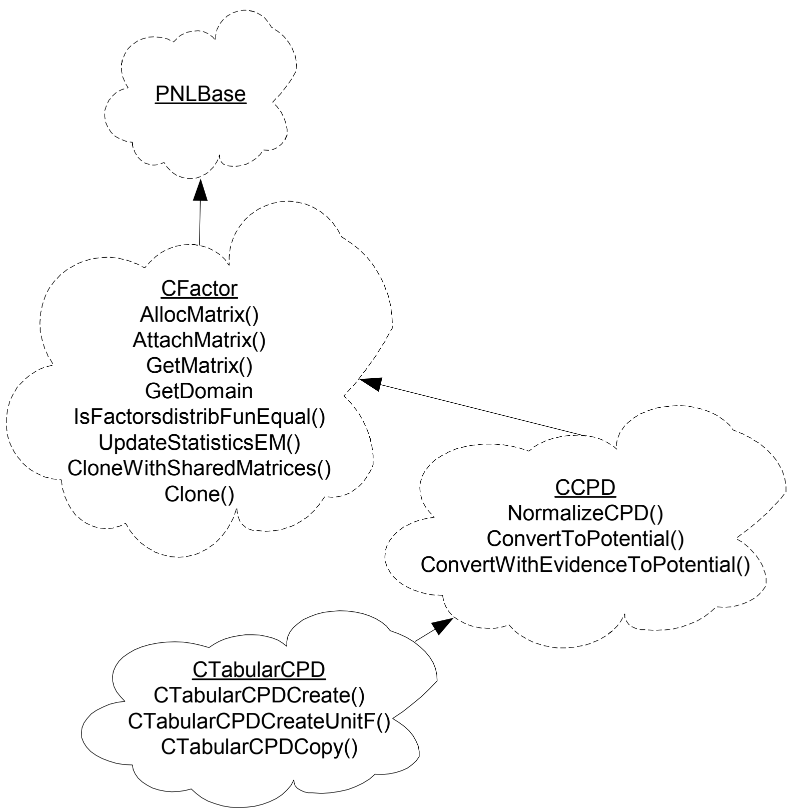
This function converts a `CCPD` object to a `CPotential` object using evidence. This function can change the distribution type of a `CPD`, unlike the combination of `ConvertToPotential` and `ShrinkObserved`.

NormalizeCPD

Normalizes `CPD`.

```
NormalizeCPD( CPD );
```

Class CTabularCPD



CTabularCPDCreate

Returns class object.

```
tCPD = CTabularCPDCreate( MD, domain );
tCPD = CTabularCPDCreate( MD, domain, matrix );
```

Arguments

domain	Array of numbers of domain nodes.
matrix	Matrix with data.
MD	Model domain.

CTabularCPDCopy

Creates a replica of input object.

```
tCPDNew = CTabularCPDCopy( tCPD );
```

Arguments

tCPD	Source class object.
tCPDNew	New class object.

CreateUnitF

Creates CPD as unit function.

```
tCPD = CTabularCPDCreateUnitF( domain, MD );
```

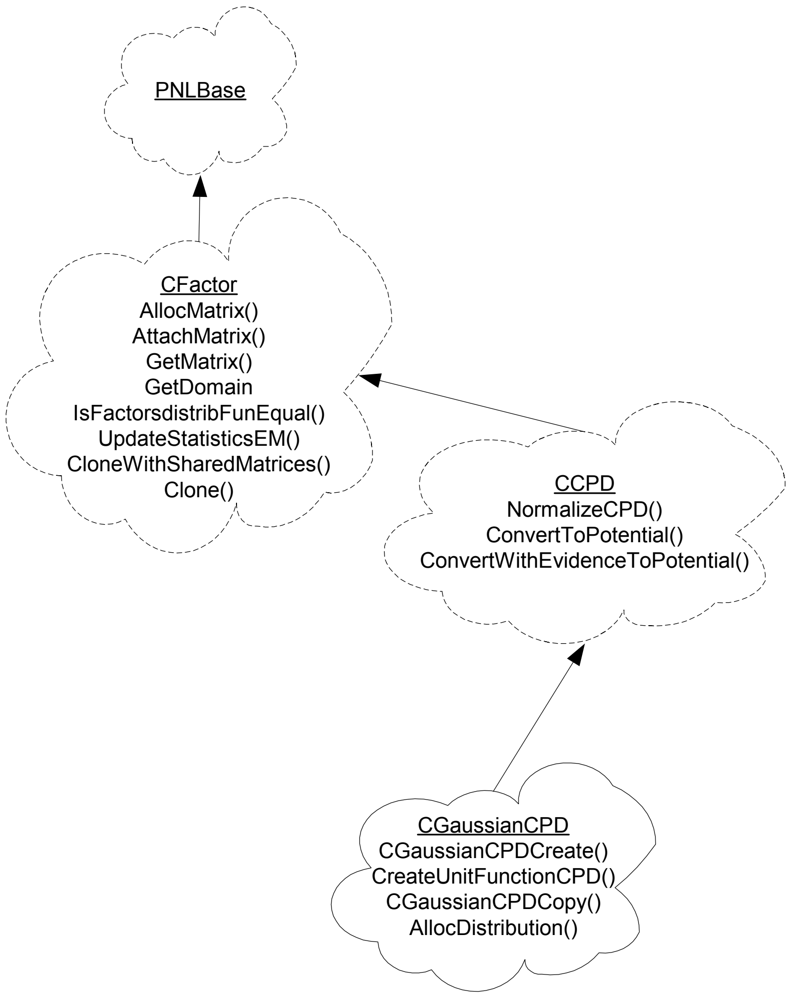
Arguments

domain	Vector of numbers of domain nodes.
MD	Model domain.
tCPD	Tabular `CPD`.

Discussion

This function creates a `CPD` that becomes a unit potential when converted to a `CPotential` object.

Class CGaussianCPD



CGaussianCPDCreate

Creates class object.

```
gCPD = CGaussianCPDCreate( domain, MD );
```

Arguments

domain	Vector of numbers of domain nodes.
MD	Model domain.

CGaussianCPDCreateUnitF

Creates CPD as unit function.

```
gCPD = CGaussianCPDCreateUnitF( domain, MD );
```

Arguments

domain	Vector of domain nodes.
MD	Model domain.

Discussion

This function creates a `CPD` that becomes a unit potential when converted to a `CPotential` object.

CGaussianCPDCopy

Creates replica of input object.

```
gCPDNew = CGaussianCPDCopy( gCPD );
```

Arguments

gCPD	Source class object.
gCPDNew	New class object.

AllocDistribution

Allocates Gaussian distribution on Gaussian child node.

```
AllocDistribution( gCPD, mean, cov, normCoeff,  weights, parentCombination );
```

Arguments

gCPD	Class object.
mean	Mean matrix.
cov	Covariance matrix. This argument is entered rowwise.
normCoeff	Value of normalization constant.
weights	Cell array of weight matrices.
parentCombination	Vector of values of discrete parents.

Discussion

This function allocates a Gaussian distribution to a class object for a given combination of discrete parents.

SetCoefficient

Sets normalization constant to Gaussian CPD object.

```
SetCoefficient( gCPD, coeff );
```

```
SetCoefficient( gCPD, coeff,  parentCombination );
```

Arguments

gCPD	Class object.
coeff	Float value of normalization constant.
parentCombination	Vector of values of discrete parents.

GetCoefficient

Obtains value of normalization constant.

```
coeff = GetCoefficient( gCPD );
coeff = GetCoefficient( gCPD, parentCombination );
```

Arguments

gCPD	Class object.
coeff	Float value of normalization constant.
parentCombination	Vector of values of discrete parents.

Class CMixtureGaussianCPD

CMixtureGaussianCPDCreate

Returns class object.

```
mgCPD = CMixtureGaussianCPDCreate( domain, MD, sumCoeff );
```

Arguments

domain	Vector of numbers of domain nodes.

MD	Model domain.
sumCoeff	Mixture coefficient.

CMixtureGaussianCPDCopy

Creates replica of class object.

```
mgCPDNew = CMixtureGaussianCPDCopy( mgCPD );
```

Arguments

mgCPD	Source class object.
mgCPDNew	New class object.

AllocDistributionVec

Allocates mixture Gaussian distribution.

```
AllocDistributionVec(mgCPD, mean, cov, normCoeff, weights, parentCombination);
```

Arguments

mgCPD	Class object.
mean	Mean matrix.
cov	Covariance matrix. The argument is entered rowwise.
normCoeff	Value of normalization constant.
weights	Cell array of weight matrices.
parentCombination	Vector of values of discrete parents.

Discussion

This function allocates a mixture Gaussian distribution to the given discrete parent combination.

SetCoefficientVec

Sets normalization constant to mixture Gaussian `CPD`.

```
SetCoefficientVec( mgCPD, coeff, parentCombination );
```

Arguments

mgCPD	Class object.
coeff	Value of the normalization constant.
parentCombination	Vector of values of discrete parents.

GetCoefficientVec

Obtains value of normalization constant.

```
coeff = GetCoefficientVec( mgCPD, parentCombination );
```

Arguments

mgCPD	Class object.
parentCombination	Vector of values of discrete parents.
coeff	Value of the normalization constant.

GetProbabilities

Returns vector of probabilities of mixture node.

```
probabilities = GetProbabilities( mgCPD );
```

Arguments

mgCPD	Class object.
probabilities	Vector of probabilities.

Discussion

This function returns the vector of probabilities of a mixture node.

Class CPotential

Class `CPotential` implements basic operations with factors. To perform an operation with a `Class CCPD` object, first use `ConvertToPotential` function to generate a potential and then call the functions you need.

You can perform the following operations with a `CPotential` object:

— multiplication;

— division;

— normalization;

— marginalization;

— shrinking if factor nodes are observed;

— expansion to the initial size.

Multiply

Multiplies two potentials and returns resulting potential.

```
resPot = Multiply( pot, pot1 );
```

Arguments

`pot`	Multiplied class object.
`pot1`	Multiplier.
`resPot`	Resulting potential.

MultiplyInSelf

Multiplies two potentials and saves the result in source object.

```
MultiplyInSelf( pot, smallPotential );
```

Arguments

`pot`	Muliplied class object.
`pot1`	Multiplier.
`smallPotential`	Right-side multiplier.

Discussion

This function multipies the source potential by a multiplier and saves the resulting potential in the source object.

DivideInSelf

Divides class object and saves result in it.

`resPot = DivideInSelf( pot, smallPotential );`

Arguments

pot	Class object.
smallPotential	Divisor.
resPot	Resulting potential.

Discussion

This function performs division of the input object by another object of the class and saves the result in the input object without creating a new class object.

GetNormalized

Creates new normalized potential.

`resPot = GetNormalized( pot );`

Arguments

pot	Class object.
resPot	Resulting potential.

Discussion

This function creates a normalized class object with the domain of the source object.

Normalize

Normalizes class object.

```
Normalize(pot);
```

Arguments

pot	Class object.

Marginalize

Marginalizes object.

```
resPot = Marginalize( pot, smallDom );
resPot = Marginalize( pot, smallDom, maximize  );
```

Arguments

pot	Class object.
smallDom	Vector of numbers of nodes that form the domain of the marginalized object.
maximize	Flag of the marginalization type. For discrete variables: • 0 stands for simple addition (default); • 1 stands for finding maximum value. For continuous variables: • both are integration operations.

Discussion

This function creates a new object either through addition or through integration of the source object with nodes that do not belong to the domain of the new object. The domain of the new object should be a subset of the source object domain.

ShrinkObservedNodes

Creates class object with different observed nodes.

```
resPot = ShrinkObservedNodes( pot, Ev );
```

Arguments

pot	Class object.
Ev	Given evidence.

Discussion

This function creates a new class object whose domain is a replica of the source object domain while the observed node values are different from their counterparts in the source object. The joint probability distribution of the new factor changes in accordance with the values of its observed nodes.

ExpandObservedNodes

Expands dimensions corresponding to observed nodes.

```
resPot = ExpandObservedNodes( pot, ev, updateInCanonical ) ;
```

Arguments

pot	Class object.
ev	Given evidence.
updateInCanonical	Flag of distribution, set to 1 by default.

Discussion

This function returns delta functions that served as multipliers for the input Gaussian distribution.

Divide

Divides factor.

```
resPot = Divide( pot, otherPot );
```

Arguments

pot	Class object.
otherPot	Divisor potential.

Discussion

This function creates a new object by division of the input factor.

MarginalizeInPlace

Marginalizes input object.

```
resPot = MarginalizeInPlace( pot, oldPot );
resPot = MarginalizeInPlace( pot, oldPot, corrPositions, maximize );
```

Arguments

pot	Class object.
oldPot	Initial potential.
corrPositions	Vector of positions for marginalisation.
maximize	Flag of marginalisation with maximization.

Discussion

This function marginalizes the input object and stores the result in it.

GetMPE

Returns maximum probability explanation.

```
ev = GetMPE(pot);
```

Arguments

pot	Class object.
ev	Evidence.

Class CTabularPotential

CTabularPotentialCreate

Returns class object.

```
pot = CTabularPotentialCreate( MD, domain, data, obsIndices );
```

Arguments

pot	Class object.

domain	Array of numbers of domain nodes.
data	Matrix with data.
MD	Model domain.
obsIndices	Vector of ndices of observed nodes of the domain.

CTabularPotentialCopy

Creates new tabular potential as copy of input object.

```
tPotNew = CTabularPotentialCopy( tPot );
```

Arguments

tPot	Class object.

CTabularPotentialCreateUnitF

Creates potential in form of unit function.

```
tPot = CTabularPotentialCreateUnitF( domain, MD );
tPot = CTabularPotentialCreateUnitF( domain, MD, asDense );
tPot = CTabularPotentialCreateUnitF( domain, MD, asDense, obsIndices);
```

sArguments

domain	Array of numbers of domain nodes.
MD	Model domain.
asDense	Flag of matrix type.
obsIndices	Numbers of observed positions.
nNodes	Number of nodes in the domain.

Class CGaussianPotential

CGaussianPotentialCreate

Creates class object.

```
gPot = CGaussianPotentialCreate(MD, domain, inMoment, matMean, matCov,
    normCoeff, obsIndices );
```

Arguments

domain	Numbers of domain nodes.
inMoment	Flag of the desired form of a Gaussian potential: 1 - moment form; 0 - canonical form. This flag defines the interpretation of the next three arguments.
if *inMoment* = 1,	
matMean	Matrix mean .
matCov	Matrix covariance.
normCoeff	Value of the normalization constant in the moment form.
if *inMoment* = 0,	
matMean	Matrix H .
matCov	Matrix K.
normCoeff	Value of the normalization constant in the canonical form.
obsIndices	Vector of indices of observed nodes of the domain.
MD	Model domain.

CGaussianPotentialCopy

Creates replica of input object.

```
gPotNew = CGaussianPotentialCopy( gPot );
```

Arguments

gPot	Class object.

CGaussianPotentialCreateDeltaF

Creates Delta function as `CGaussianPotential`.

```
gPot = CGaussianPotentialCreateDeltaF(domain, MD, matMean );
gPot = CGaussianPotentialCreateDeltaF(domain, MD, matMean, isInMoment );
gPot = CGaussianPotentialCreateDeltaF(domain, MD, matMean, isInMoment,
    obsIndices );
```

Arguments

domain	Numbers of domain nodes.
matMean	Float values of the mean matrix.
isInMoment	Flag of the form of the resulting potential: 1 - moment form (mean, covariance matrices, normalization constant) 0 - canonical form (canonical matrices g, H, K).
MD	Model domain.
obsIndices	Vector of indices of observed nodes in the domain.

CGaussianPotentialCreateUnitF

Creates class object in form of unit function distribution.

```
gPot = CGaussianPotentialCreateUnitF(domain, MD);
gPot = CGaussianPotentialCreateUnitF(domain, MD, isInCanonical);
gPot = CGaussianPotentialCreateUnitF(domain, MD,isInCanonical, obsIndices);
```

Arguments

domain	Numbers of domain nodes.
isInCanonical	Flag of the form of the unit function: 1 - canonical form; 0 - moment form.
obsIndices	Indices of observed domain nodes.
MD	Model domain.

Discussion

This function creates a class object in the form of a unit function distribution.

SetCoefficient

Sets normalization constant to class object.

```
SetCoefficient( gPot, coeff, isForCanonical );
```

Arguments

coeff	Float value of the normalization constant.
isForCanonical	Flag of the distribution type to set the coefficient:

1 - canonical form

0 - moment form.

GetCoefficient

Gets value of normalization constant.

```
coeff = GetCoefficient( gpot, isforCanonical );
```

Arguments

isforCanonical Flag of distribution type to get the coefficient:

1 - canonical form;

0 - moment form.

Class CFactors

Class `CFactors` stores `Inference Engines` objects and provides them for a graphical model. It allows to create class objects independently of the model and easily attach them to the model when needed.

CFactorsCreate

Creates `CFactors` *class object.*

```
factors = CFactorsCreate( NumOfFactors );
```

Arguments

numOfFactors Maximal number of factors in the factor array. The argument equals to the total number of nodes and cliques for `BNet` and for `MNet` objects respectivelly.

GetNumberOfFactors

Returns current number of factors in factor array.

```
nf = GetNumberOfFactors( factors );
```

Arguments

factors	Class object.
nf	Number of factors.

GetFactor

Returns factor.

```
factor = GetFactor( factors, factorNum );
```

Arguments

factors	Class object.
factorNum	Factor index in the array of factors.

Discussion

This function returns a factor with the index equal to *factorNum*.

AddFactor

Adds new factor to graphical model.

```
num = AddFactor( factors, factor );
```

Arguments

`factors`	Class object.
`factor`	Factor to be set in the factor array.

Discussion

This function adds a factor to the graphical model and returns the index of the factor in the factors array.

ShrinkObsNdsForAllFactors

Shrinks all factors stored in `CFactors` *class using input evidence.*

```
ShrinkObsNdsForAllFactors( factors, evidence );
```

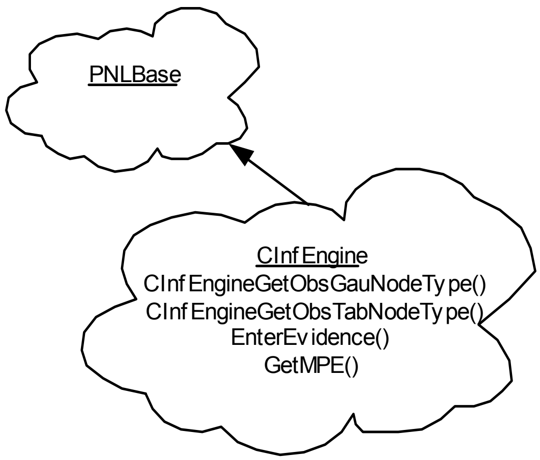
Arguments

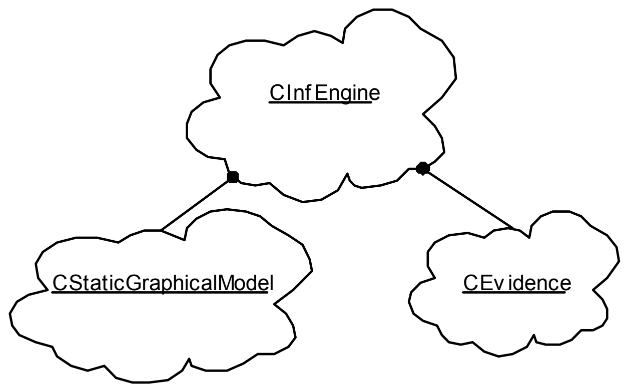
`factors`	Class object.
`evidence`	Evidence.

Discussion

This function shrinks all the factors stored in `CFactors` class using the input evidence.

Inference Engines





Class CInfEngine

`CInfEngine` is the superclass of other classes that implement inference in graphical models. The class stores all functions of its subclasses.

pnlDetermineDistributionType

Returns type of distribution.

```
distrtype = pnlDetermineDistributionType( numOfAllNds, numOfObsNds,
    obsNdsIndices, allNdsTypes );
```

Arguments

numOfAllNds	Total number of nodes.
numOfObsNds	Number of observed nodes.
obsNdsIndices	Vector of indices of observed nodes.
allNdsTypes	Cell array of node types.

Discussion

This function can return the following types of distribution:

— `dtTabular`, if all hidden nodes are discrete;

— `dtGaussian`, if all hidden nodes are continuous;

— `dtCondGaussian`, if some of the nodes are discrete and some are continuous.

— `dtScalar`, if all nodes are observed.

pnlDetermineDistribTypeByMD

Returns type of distribution.

```
distrtype = pnlDetermineDistribTypeByMD(MD, nQueryNodes, query, ev );
```

Arguments

MD	Model domain.
nQueryNodes	Number of query nodes.

query	Query nodes.
ev	Evidence.

Discussion

This function can return the following types of distribution:

- — *dtTabular*, if all hidden, nodes are discrete;
- — *dtGaussian*, if all hidden nodes are continuous;
- — *dtCondGaussian*, if some of the nodes are discrete and some are continuous.
- — *dtScalar,* if all nodes are observed.

EnterEvidence

Starts inference in graphical model.

```
EnterEvidence( infEng, evidence );
EnterEvidence( infEng, evidence, maximize );
EnterEvidence( infEng, evidence, maximize, sumOnMixtureNode );
```

Arguments

infEng	Class object.
evidence	`CEvidence` object with observed nodes and their values.
maximize	Flag of inference with MPE.
sumOnMixtureNode	Flag of addition on a mixture node.

Discussion

This function starts inference in a graphical model. The inference procedure may start either with this function or with the function `MarginalNodes`, depending on the type of inference.

MarginalNodes

Calculates joint probability distribution for given nodes.

```
MarginalNodes( infEng, query);
MarginalNodes( infEng, query, nodeExpandJPD );
```

Arguments

infEng	Class object.
query	Vector of nodes whose joint probability distribution is to be calculated.
nodeExpandJPD	Flag of JPD expansion.

Discussion

This function calculates joint probability distribution for a number of given nodes. The function creates Maximum Probability Explanation (MPE) for the distribution as a `Model Domain` object. You may obtain the MPE and the factor using coordinate functions.

GetQueryJPD

Returns `CPotential` *object.*

```
pot = GetQueryJPD( infEng );
```

Arguments

infEng	Class object.

Discussion

This function returns the joint probability distribution which is calculated by the function MarginalNodes.

GetMPE

Returns MPE.

```
evidence = GetMPE( infEng );
```

Arguments

infEng	Class object.

Discussion

This function returns the Maximum Probability Explanation which is calculated by the function MarginalNodes.

GetModel

Returns graphical model processed by inference engine.

```
grModel = GetModel( infEng );
```

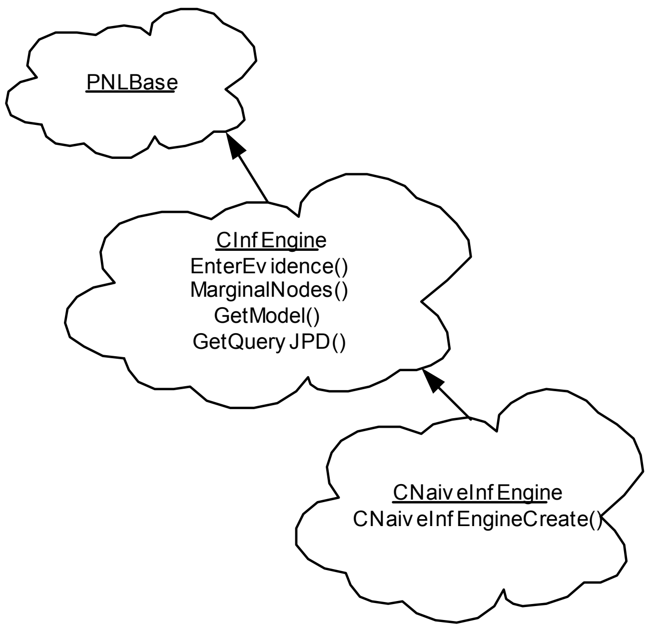
Arguments

infEng	Class object.
grModel	Graphical model.

Discussion

This function returns the graphical model which is processed by the infrence engine. The function is used in learning algorithms.

Class CNaiveInfEngine



CNaiveInfEngineCreate

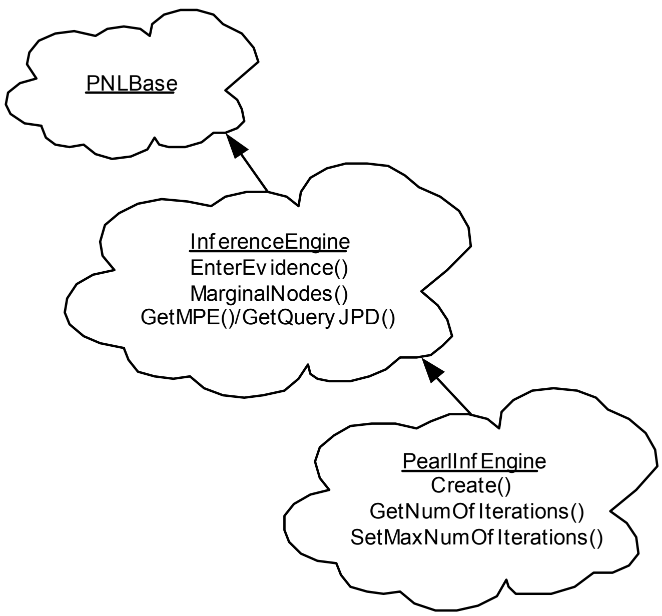
Creates class object.

```
infEng = CNaiveInfEngineCreate( grModel );
```

Arguments

grModel	Static graphical model. It can either be an `MRF` (`MRF2`) or a `BNet` object.

Class CPearlInfEngine



CPearlInfEngineCreate

Creates class object.

```
infEng = CPearlInfEngineCreate( grModel );
```

Arguments

grModel	Static graphical model. It can be either an `MRF2` or a `BNet` oblect. `BNet` cannot have directed cycles.

Discussion

This function creates a class object from the input graphical model. If the graph of the input model (`Class CBNet` or `Class CMRF2`) has undirected loops, the inference result is approximate. If all nodes of the graph are undirected, the inference result is exact.

CPearlInfEngineIsModelValid

Checks if model is valid for Pearl inference.

```
flag = CPearlInfEngineIsModelValid( grModel );
```

Arguments

grModel	Graphical model.

SetMaxNumberOfIterations

Sets maximum number of iterations for parallel protocol.

```
SetMaxNumberOfIterations( infEng, maxNumOfIters );
```

Arguments

infEng	Class object.
maxNumOfIters	Maximum number of iterations.

Discussion

This function sets the maximum number of iterations for the parallel protocol.

GetNumberOfProvideIterations

Returns number of inference iterations.

```
nIter = GetNumberOfProvideIterations( infEng );
```

Arguments

infEng	Class object.
nIter	Number of iterations.

Discussion

This function returns the number of iterations that were performed during inference.

SetTolerance

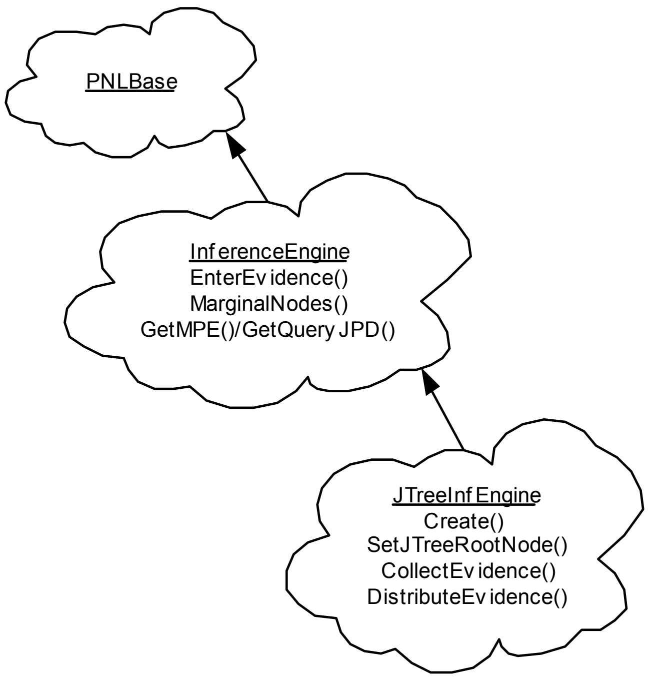
Sets tolerance for convergency check.

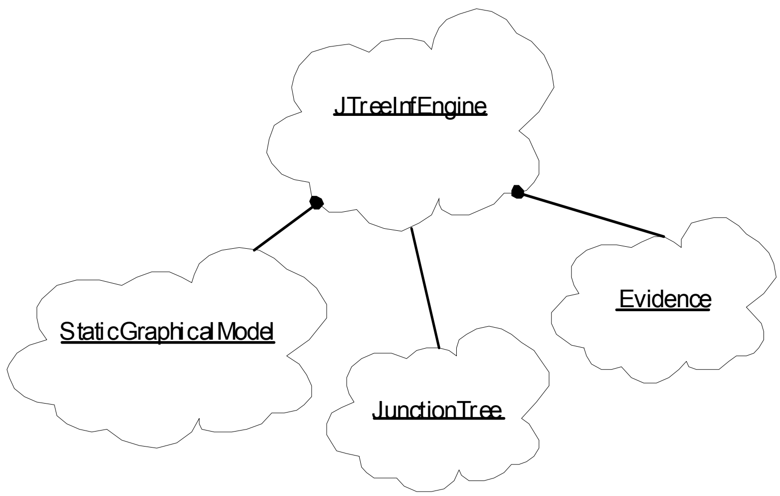
```
SetTolerance( infEng, tolerance );
```

Arguments

infEng	Class object.
tolerance	Precision value.

Class CJtreeInfEngine





CJTreeInfEngineCreate

Creates class object.

```
infEng = CJtreeInfEngineCreate( grModel );
infEng = CJtreeInfEngineCreate( grModel, SubGrToConnect );
```

Arguments

infEng	Class object.
grModel	Static graphical model. It can be either an `MRF2` or a `BNet` object. A `BNet` object cannot have directed cycles.
SubGrToConnect	Cell array of cell arrays of nodes to be connected.

CJtreeInfEngineCreateFromJTree

Creates class object.

```
infEng = CJtreeInfEngineCreateFromJTree( grModel, jTree );
```

Arguments

grModel	Static graphical model. It can be either an `MRF2` or a `BNet` object. A `BNet` object cannot have directed cycles.
jTree	Junction tree.

CJTreeInfEngineCopy

Creates replica of `CJTreeInfEngine` *object.*

```
infEngNew = CJtreeInfEngineCopy( infEng );
```

Arguments

infEng	Source class object.
infEngNew	New class object.

GetEvidence

Returns given evidence.

```
ev = GetEvidence( infEng );
```

GetJTreeRootNode

Returns number of root node.

```
nodeNum = GetJTreeRootNode( infEng );
```

Arguments

infEng	Class object.
nodeNum	Number of the root node.

Discussion

This function returns the number of the root node of the Junctin tree.

GetClqNumsContainingSubset

Returns numbers of Junction tree cliques with common subset of nodes.

```
clqsContSubset = GetClqNumsContainingSubset( infEng, subset );
```

Arguments

infEng	Class object.
subset	Subset of nodes.
clqsContSubset	Vector of numbers of cliques with a common subset.

Discussion

This function returns numbers of the Junction tree cliques that have a common subset of nodes.

GetNodesConnectedByUser

Returns set of connected nodes.

```
nds = GetNodesConnectedByUser( infEng, nodeSetNum );
```

Arguments

infEng	Class object.
nodeSetNum	Number of the set of nodes.
nnds	Vector of connected nodes.

Discussion

This function returns the set of nodes that were connected when the Junction tree was created.

SetJTreeRootNode

Sets root of Junction tree.

```
SetJTreeRootNode( infEng, nodeNum );
```

Arguments

infEng	Class object.
nodeNum	Number of a node to become the root node.

Discussion

This function turns a given node of the Junction tree into its root node.

GetLogLik

Returns logarithm of likelihood.

```
logLik = GetLogLik( infEng );
```

Arguments

infEng	Class object.

Discussion

This function divides the potential of a Junction tree node by the distribution function.

CollectEvidence

Collects evidence.

```
CollectEvidence();
```

DistributeEvidence

Distributes evidence.

```
DistributeEvidence( infEng );
```

Arguments

infEng	Class object.

ShrinkObserved

Initializes Junction tree using given evidence.

```
ShrinkObserved( evidence, maximize, sumOnMixtureNode, bRebuildJTreee );
```

Arguments

evidence	Evidence.
maximize	Flag of maximization.
sumOnMixtureNode	Flag of summation on the mixture node.
bRebuildJTree	Flag of the Junction tree rebuilding.

Discussion

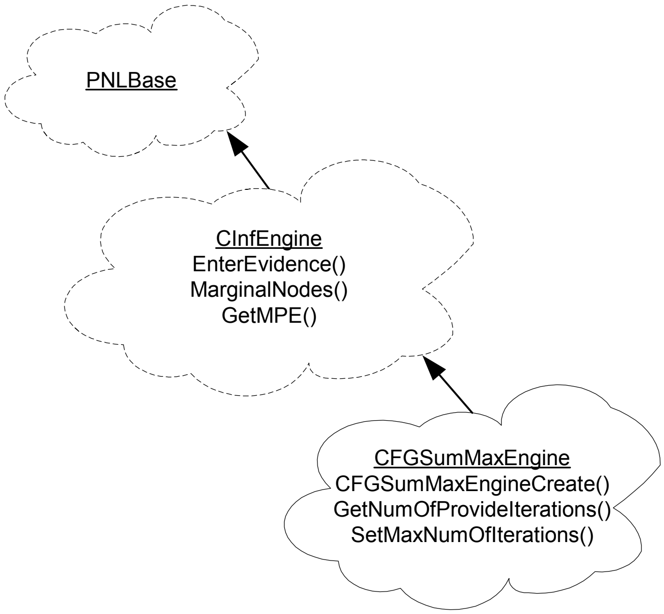
This function initializes a Junction tree using given evidence.

GetQueryMPE

Returns most probable distribution.

```
pot = QueryMPE( infEng );
```

Class CFGSumMaxInfEngine



The class implements belief propagation on a factor graph model.

CFGSumMaxInfEngineCreate

Creates object of class.

```
infEng = CFGSUMMaxInfEngineCreate( grModel );
```

Arguments

grModel	Factor graph.

Discussion

This function creates a class object. Inference is implemented for `FactorGraph` models only.

SetMaxNumberOfIterations

Sets maximum number of iterations for inference.

```
SetMaxNumberOfIterations( infEng, number );
```

Arguments

infEng	Class object.
number	Maximum number of iterations.

Discussion

This function sets the maximum number of iterations for the inference procedure.

GetNumberOfProvideIterations

Returns number of iterations provided during inference.

```
nIter = GetNumberOfProvideIterations( infEng );
```

SetTolerance

Sets value of tolerance used in convergence check.

```
SetTolerance( infEng, tolerance );
```

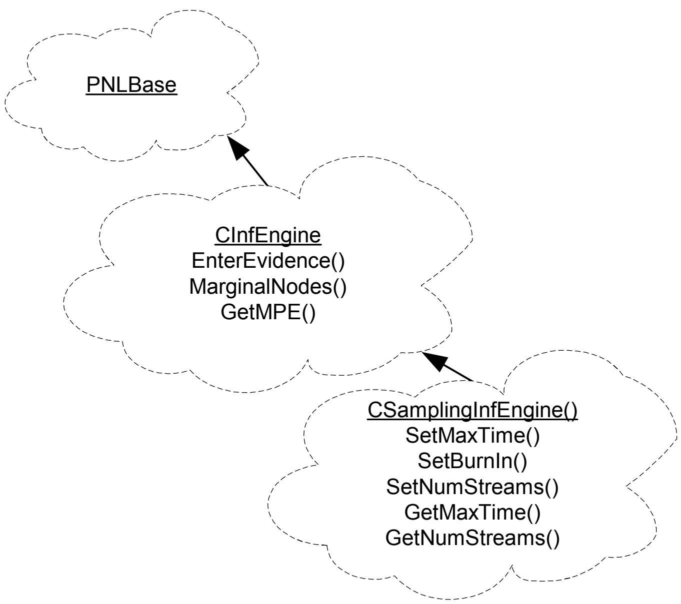
Arguments

tolerance	Tolerance value.

Discussion

This function sets the value of tolerance which to be used in convergence checking.

Class CSamplingInfEngine



Class `CSamplingInfEngine` is a superclass for classes `CGibbsSamplingInfEngine` and `CGibbsWithAnnealingInfEngine` that implement inference in static graphical models using stochastic simulates technique known as Markov chain Monte Carlo. This technique generates samples from the required posterior distribution. Inference constructs Markov chain with stationary distribution $P\langle h/ v\rangle$.

Let $h^{\langle t\rangle}$ be a model state at a certain time t (by the state of a model we mean values of its hidden variables). According to the following formula the changed value of a variable at time $t+1$ is: $\langle x_k / \{x_i^{\langle t-1\rangle} = a_i, i \neq k\}\rangle = \frac{1}{a}P\langle x_k, \{x_k^{\langle t-1\rangle} = a_i, i \neq k$

SetMaxTime

Sets maximum number of sampling iterations.

```
SetMaxTime( infEng, time );
```

Arguments

infEng	Class object.
time	Maximum number of iterations.

Discussion

This function sets the maximum number of sampling iterations.

SetBurnIn

Sets number of iterations before statistics collection.

```
SetBurnIn( infEng, time );
```

Arguments

infEng	Class object.
time	Number of iterations.

Discussion

This function sets the number of iterations to be performed before the statistical data is collected.

SetNumStreams

Sets number of streams for sampling.

```
SetNumStreams( infEng, nStreams );
```

Arguments

infEng	Class object.
nStreams	Number of streams.

Discussion

This function sets the number of independent streams for sampling.

GetMaxTime

Returns maximum number of sampling iterations.

```
tMax=GetMaxTime( infEng);
```

Arguments

infEng	Class object.
tMax	Maximum number of sampling iterations.

Discussion

This function returns the maximum number of sampling iterations.

GetBurnIn

Returns number of iterations before statistics collection.

```
n = GetBurnIn( infEng );
```

Arguments

infEng	Class object.
n	Number of iterations.

Discussion

This function returns the number of iterations before the statistical data is collected.

GetNumStreams

Returns number of sampling streams.

```
n = GetNumStreams( infEng );
```

Arguments

infEng	Class object.
n	Number of streams.

Continue

Continues sampling and updates statistics.

```
Continue( infEng, dT);
```

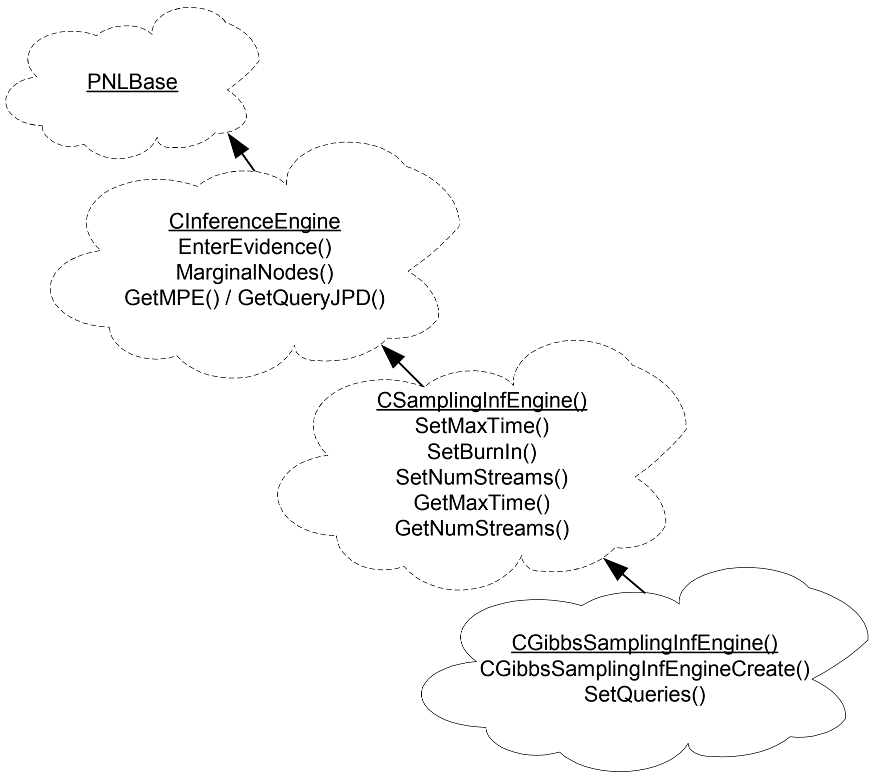
Arguments

infEng	Class object.
dT	Number of additional samples.

Discussion

This function continues sampling procedure and the update of statistical data.

Class CGibbsSamplingInfEngine



CGibbsSamplingInfEngineCreate

Creates class object.

`infEng = CGibbsSamplingInfEngineCreate( grModel );`

Arguments

grModel	Graphical model.

Discussion

This function creates either a `MRF` (`MRF2`) object or a `BNet` object.

SetQueries

Sets possible queries.

```
SetQueryes( infEng, queries );
```

Arguments

queries	Cell array of vectors with query nodes.

Discussion

This function sets possible queries. This function is compulsory before calling `EnterEvidence`.

UseDSeparation

Conditions d-separation use in sampling for `BNet`.

```
UseDSeparation( infEng, isUsing );
```

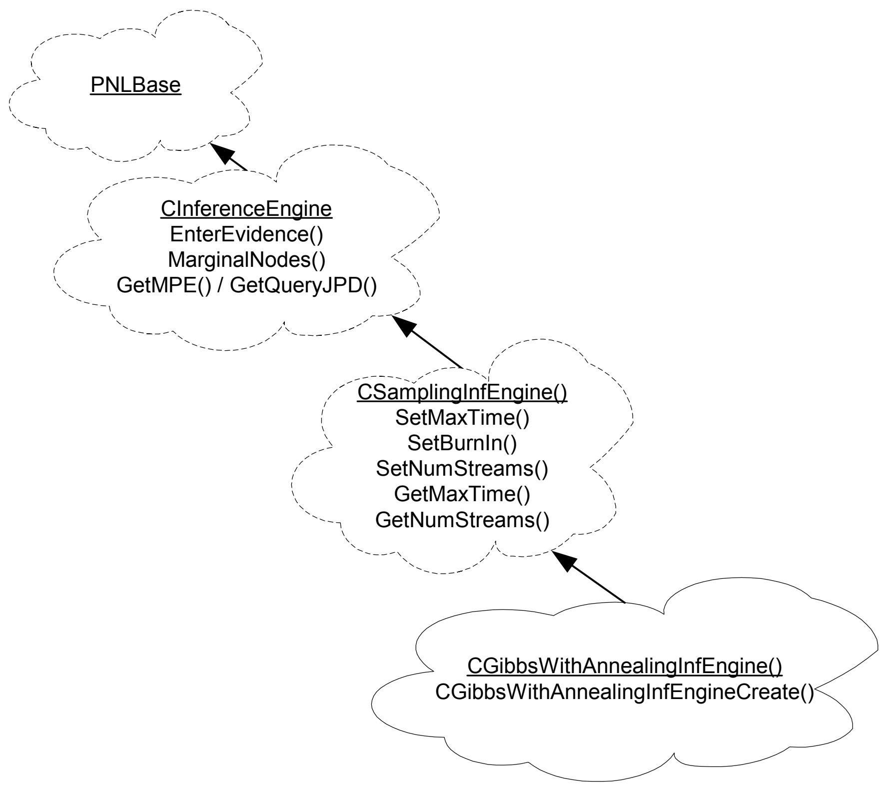
Arguments

infEng	Class object.
isUsing	Flag of *d*-separation.

Discussion

This function conditions the use of *d*-separatrion in sampling for `BNet`.

Class CGibbsWithAnnealingInfEngine



`CGibbsWithAnnealingInfEngine` implements Gibbs Sampler with annealing schedule $T(S)$

$$T(S) = \frac{C}{\log(1 + S)} \quad (1),$$

where $T(S)$ is the temperature which depends on the sampling iteration and C is a parameter.

This inference obtains maximum probability explanation for nodes.

For more detailed information see [Stuart Geman and Donald Geman. Stochastic Relaxation, Gibbs Distribution, and the Bayesian Restoration of Images].

CGibbsWithAnnealingInfEngCreate

Creates class object.

```
CGibbsWithAnnealingInfEngCreate( grModel );
```

Arguments

grModel	Graphical model.

SetAnnealingCoefficientC

Changes default coefficient C of annealing schedule.

```
SetAnnealingCoefficientC( infEng, val );
```

Arguments

val	Value of the coefficient C.

Discussion

This function sets a new value for the coefficient C of the annealing schedule.

SetAnnealingCoefficientS

Changes default coefficient S of annealing schedule.

```
SetAnnealingCoefficientS( infEng, val );
```

Arguments

val	Value of the coefficient S.

Discussion

This function sets a new value for the coefficient S of the annealing schedule.

GetCurrentTemp

Returns value of current temperature.

```
t = GetCurrentTemp( infEng );
```

Arguments

infEng	Class object.
t	Temperature value.

UseAdaptation

Initiates adaptation during inference.

```
UseAdaptation( infEng,isUsed );
```

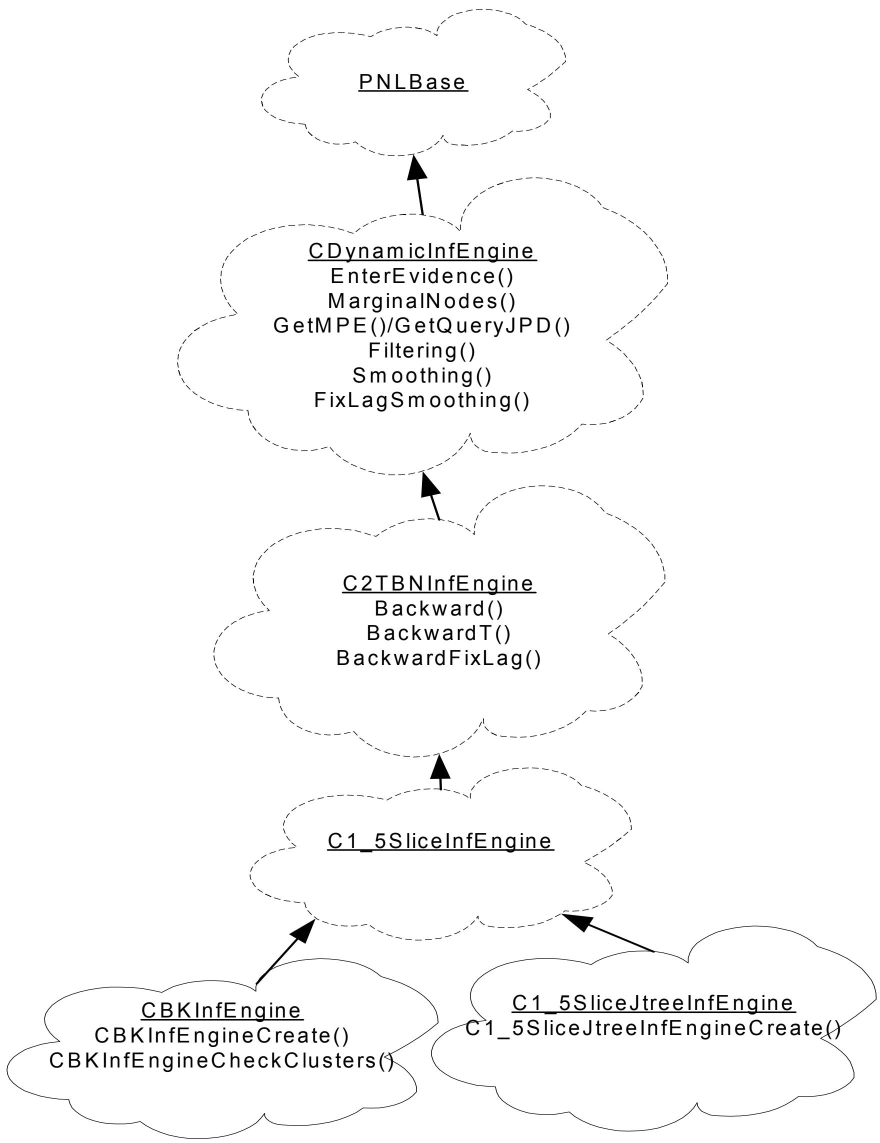
Arguments

infEng	Class object.
isUsed	Flag of adaptation.

Class CDynamicInfEngine

Class CDynamicInfEngine is a superclass for all classes that implement inference in dynamic graphical models. Figure 3-2 shows the hierarchy of CDynamicInfEngine class.

Figure 3-2 Structure of `CDynamicInfEngine` Class



DefineProcedure

Defines type of inference procedure.

```
DefineProcedure( infEng, procedure, lag );
```

Arguments

infEng	Class object.
procedure	Type of inference: `itFiltering`, `itSmoothing`, `itFixLagSmoothing` or `itViterbi`.
lag	Integer value. Corresponds to the value of lag in the fixed-lag smoothing inference and to the number of time slices in the smoothing inference. In the filtering inference the argument equals to 0.

Discussion

This function defines one of the following types of inference procedure: filtering, smoothing, fixed-lag smoothing and Viterby decoding.

EnterEvidence

Enters evidence to engine.

```
EnterEvidence( infEng, evidences);
```

Arguments

infEng	Class object.
evidences	Cell array of evidences.

MarginalNodes

Marginalizes joint probability distribution to query nodes.

```
MarginalNodes( query, timeSlice );
MarginalNodes( query, timeSlice, notExpandJPD );
```

Arguments

query	Vector of nodes for the query.
timeSlice	Query slice number. Equals to 0 in the filtering and the fixed-lag smoothing inference procedures because they are on-line procedures.
notExpandJPD	Flag of expansion.

Discussion

This function marginalizes the joint probability distribution to the set of nodes in a slice given as the `query` input argument, at the time `timeSlice`. In the filtering inference the argument `timeSlice` should equal to the current time and in the fixed-lag smoothing it should equal to time-lag.

Let the model have `N` nodes per slice. Then nodes in the query with numbers from `0` to `N - 1` belong to the slice `timeSlice - 1` and nodes with numbers from `N` to `2N - 1` belong to the slice `timeSlice`. For prior time-slices, such as, for example, `timeSlice = 0`, nodes in the query have numbers from `0` to `N - 1`.

GetQueryJPD

Returns joint probability distribution of query nodes.

```
pot = GetQueryJPD( InfEng );
```

Arguments

infEng	Class object.
pot	Joint probability distribution of query nodes.

GetMPE

Returns maximum probablility explanation of query nodes.

```
ev = GetMPE( infEng );
```

Arguments

infEng	Class object.
ev	Maximum probability explanation of query nodes.

Filtering

Performs filtering procedure.

```
Filtering( infEng, timeSlice );
```

Arguments

infEng	Class object.
timeSlice	Number of the current time slice.

Smoothing

Performs smoothing procedure.

```
Smoothing( infEng );
```

Arguments

infEng	Class object.

FixLagSmoothing

Performs fixed-lag smoothing procedure.

```
FixLagSmoothing( infEng, timeSlice );
```

Arguments

infEng	Class object.
timeSlice	Number of the current time slice.

Discussion

This function performs fixed-lag smoothing with a lag defined in `DefineProcedure`.

FindMPE

Finds maximum probability explanation.

```
FindMPE( infEng );
```

Arguments

infEng	Class object.

Discussion

This function performs the procedure of Viterbi decoding.

GetDynamicModel

Returns source dynamic model.

```
grModel = GetDynamicModel( infEng );
```

Arguments

infEng	Class object.
grModel	Graphical model.

GetProcedureType

Returns type of inference procedure.

```
prType = GetProcedureType( infEng );
```

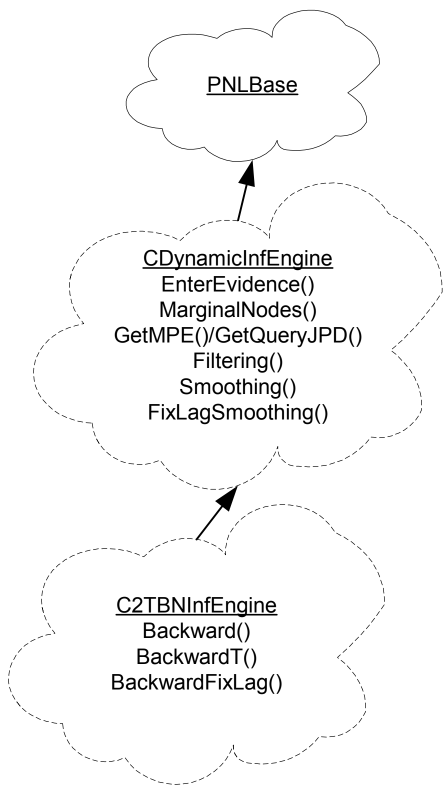
Arguments

infEng	Class object.
prType	Type of inference.

Discussion

This function returns the type of implemented inference: `ptFiltering`, `ptSmoothing`, `ptFixLagSmoothing`, `ptViterbi`.

Class C2TBNInfEngine



`C2TBNInfEngine` is a superclass for all dynamic inference engine classes, that use forward-backward operations between slices. With such structure of classes an inference procedure (filtering, smoothing, and the others) can be implemented with the combination of functions `ForwardFirst`, `Forward`, `BackwardT`, `Backward` of its subclasses.

ForwardFirst

Performs forward operation for prior slice.

```
ForwardFirst( infEng, evidence );
```

Arguments

infEng	Class object.
evidence	Evidence for the prior slice.

Forward

Performs forward operation.

```
Forward( infEng, evidence );
```

Arguments

infEng	Class object.
evidence	Evidence for any but the prior slice.

BackwardT

Performs first backward operation after last forward operation.

```
BackwardT( infEng );
```

Arguments

infEng	Class object.

Backward

Performs backward operation.

```
Backward( infEng );
```

Arguments

infEng	Class object.

BackwardFixLag

Performs sequence of backward operations.

```
BackwardFixLag( infEng );
```

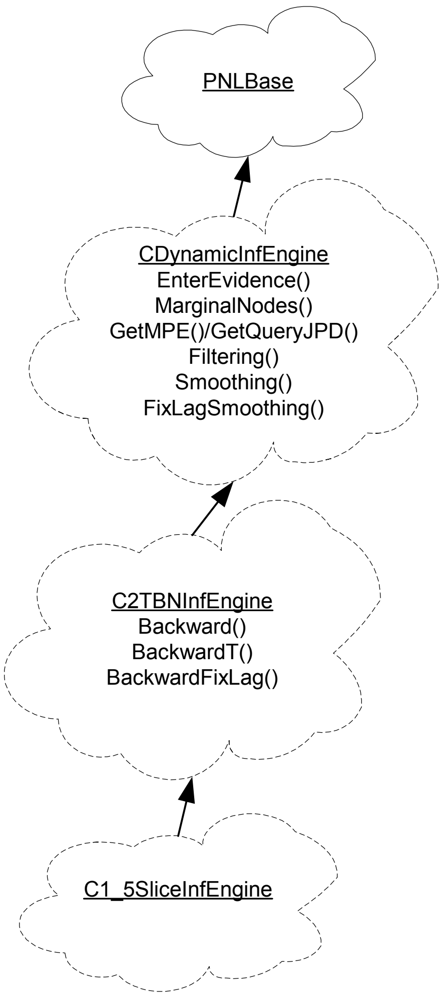
Arguments

infEng	Class object.

Discussion

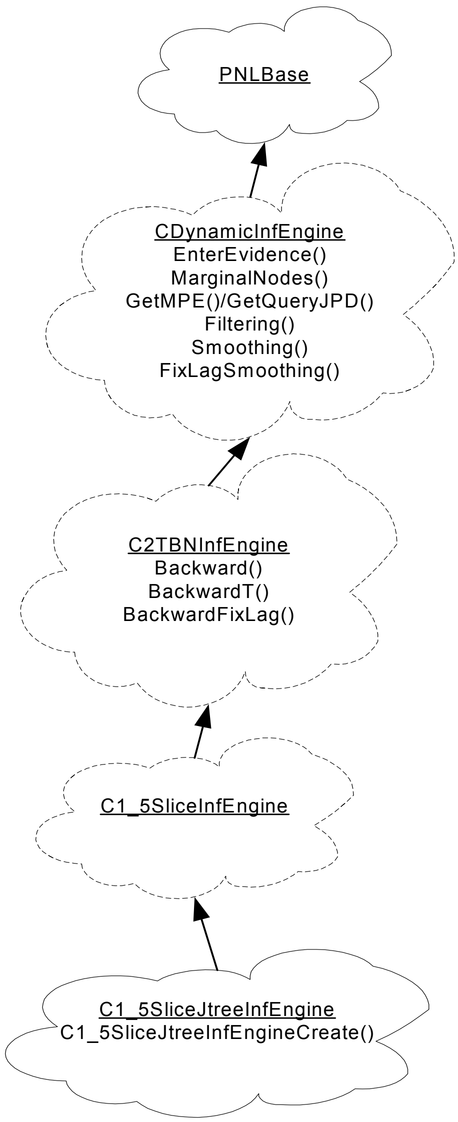
This function performs a number of backward operations in the fixed-lag smoothing procedure, restoring data for intermediate steps. The number of operations equals to the value of lag.

Class C1_ 5SliceInfEngine



`C1_ 5SliceInfEngine` is a superclass for all inference procedures that perform forward-backward operations between 1.5 slices.

Class C1_5SliceJTreeInfEngine



C1_5SliceJTreeEngineCreate

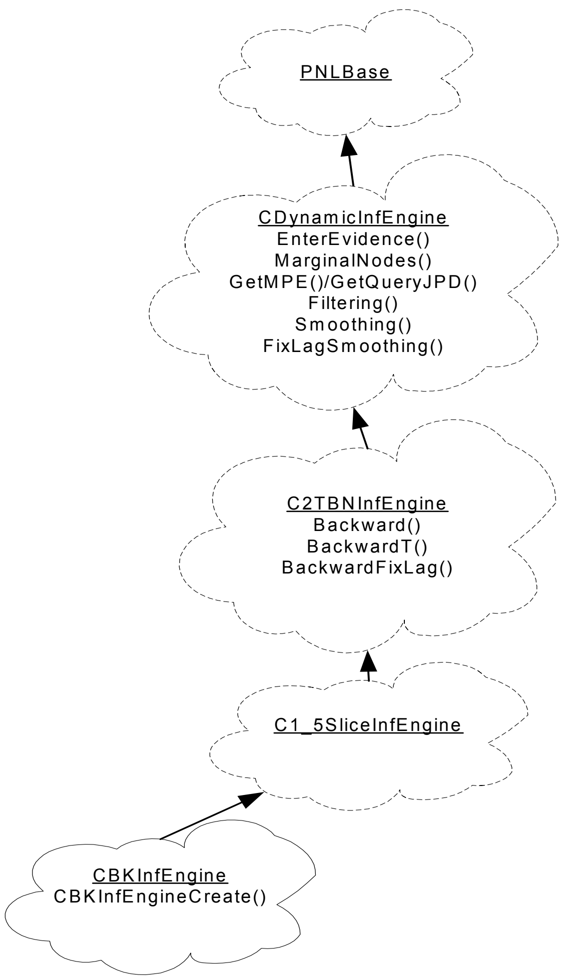
Creates class object.

```
infEng = C1_5SliceJtreeInfEngineCreate( grModel );
```

Arguments

grModel	Dynamic graphical model.

Class CBKInfEngine



CBKInfEngineCreate

Creates class object.

```
infEng = CBKInfEngineCreate( grModel, isFF );
```

Arguments

grModel	Dynamic graphical mode.
isFF	Flag of factorization. Equals to 1 for the fully factorixed inference whose every interface node belongs to a separate cluster. Equals to 0 for the exact inference whose interface nodes lie in one clique.

CBKInfEngineCreate

Creates class object.

```
infEng = CBKInfEngineCreate( grModel, clusters );
```

Arguments

grModel	Dynamic graphical model.
clusters	Cell array of vectors with nodes of a cluster.

CBKInfEngCheckClusters

Checks validity of clusters.

```
CheckClustersValidity( clusters, interfNds );
```

Arguments

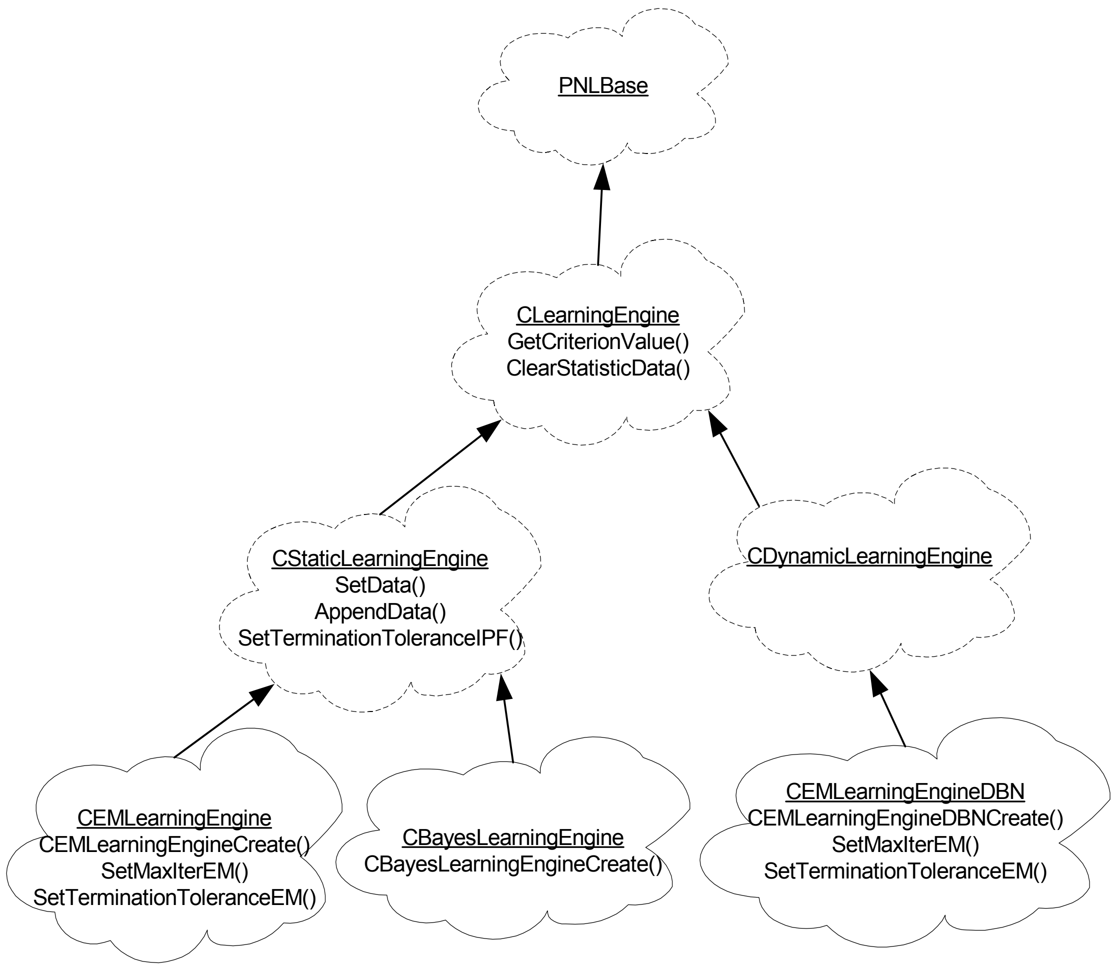
clusters	Cell array of vectors with inference nodes. Each node belongs only to one cluster.
interfNds	Vector of inference nodes.

Discussion

This function checks if a cluster is valid. A vector of the BK inference can be composed of vectors which contain numbers of interface model nodes. If the nodes belong to one class they are sure to lie in one clique of the Junction tree. If the nodes belong to different clusters the inference is fully factorized. If the nodes belong to the same cluster the inference is exact.

Learning Engines

Figure 3-3 Structure of learning engines



Class CLearningEngine

Learn

Performs learning.

```
Learn( learnEng );
```

Arguments:

learnEng	Class object.

Discussion

This function trains a graphical model using given data. In parameter learning the function upgrades factors using given evidence. In structure learning the function creates a new graphical model.

GetCriterionValue

Returns array of criterion values used in learning.

```
value = GetScore( learnEng );
```

Arguments

learnEng	Class object.
value	Vector of values.

Discussion

This function returns numeric values of the criterion that is to be maximized during model learning.

ClearStatisticData

Clears statistical data.

```
ClearStatisticData( learnEng );
```

Arguments

learnEng	Class object.

Class CStaticLearningEngine

SetData

Sets statistical data for learning.

```
SetData( learnEng, evidences );
```

Arguments

learnEng	Class object.
evidences	Cell array of `CEvidence` objects.

Discussion

This function sets statistical data for learning. When you call this function, the prior statistical data is deleted.

AppendData

Acquires evidence information.

```
AppendData( learnEng, evidences );
```

Arguments

learnEng	Class object.
evidences	Cell array of `CEvidence` objects.

Discussion

This function adds a set of evidences to previously collected data.

GetStaticModel

Returns graphical model.

```
grModel = GetStaticModel( learnEng );
```

Arguments

learnEng	Class object.

Class CEMLearningEngine

Class `CEMLearningEngine` is used in the learning of Bayesian networks with discrete or multivariate Gaussian nodes as well as in the learning of Markov networks with discrete nodes. The learning is based on *Expectation Maximization* (EM) algorithm.

CEMLeaningEngineCreate

Creates class object.

```
learnEng = CEMLearningEngineCreate( grModel );
learnEng = CEMLearningEngineCreate( grModel, infEng );
```

Arguments

grModel	Graphical model.
infEng	Inference engine used in the lerning procedure. By default the Junction tree inference engine is used.

SetMaxIterEM

Sets maximum iteration depth for Expectation Maximization.

```
SetMaxIterEM( learnEng, numOfIter );
```

Arguments

learnEng	Class object.
numOfIter	Maximum iteration depth.

Discussion

This function sets the maximum number of iterations to be performed in the course of learning.

SetTerminationToleranceEM

Sets termination tolerance.

```
SetTerminationToleranceEM( learnEng, precision );
```

Arguments

learnEng	Class object.
precision	Precision.

Discussion

This function sets the termination condition for EM. EM is over, if the difference between the logarithm of likelihood at a current step and at the previous step does not exceed a certain set value.

Class CBayesLearningEngine

Class `CBayesLearningEngine` is used for training `BNet` objects, where parameters of a `CPD` are not fixed and have their own probability distributions (see User Guide). The current version of Matlab PNL supports parameters distributions only for a `CTabular CPD`. Both prior and posterior parameters distributions of a tabular `CPD` are Dirichlet. Dirichlet table distribution is stored in `Tabular_CPD` in the form of a matrix of the same size as `CPT`. Initial values are specified either by `AllocMatrix` or by `AttachMatrix` functions of the corresponding factor. Dirichlet priors have the form of pseudo counts which stand for an imaginary observed number of cases and assume any non-negative values.

In the course of learning parameters are updated. Updated prior parameters may be used as priors in future learning. In the current version of PNL Bayesian parameter learning is supported only if the input data is complete, that is, if all the `BNet` nodes of training samples are observed.

CBayesLearningEngineCreate

Creates class object.

```
learnEng = CBayesLearningEngineCreate( grModel );
```

Arguments

grModel	Graphical model.

Discussion

This function creates a class object. It applies only to `BNet` graphical models only.

Class CBICLearningEngine

Class `CBICLearningEngine` is used for learning a Bayesian network with discrete nodes when model structure is unknown and all variables are observed. The learning is based on Bayesian Information Criterion (BIC). The result of learning is a new Bayesian network.

CBICLearningEngineCreate

Creates class object.

```
learnEng = CBICLearningEngineCreate( grModel );
```

Arguments

grModel	Graphical model.

GetGraphicalModel

Returns created graphical model.

```
grModel = GetGraphicalModel( learnEng );
```

Arguments

learnEng	Class object.
grModel	Graphical model.

Discussion

This function returns a topologically sorted graphical model which was created as a result of structure learning.

GetOrder

Returns array of values corresponding to node numbers.

```
reordering = GetOrder( learnEng );
```

Arguments

learnEng	Class object.
reordering	Vector of values that correspond to node numbers.

Discussion

This function returns the array of values that correspond to node numbers of the source graphical model.

Class CDynamicLearningEngine

`CDynamicLearningEngine` is a superclass for all classes that implement learning for dynamic graphical models. The class contains functions that belong to its child classes.

SetData

Sets statistical data for learning.

```
SetData( learnEng, evidences );
```

Arguments

learnEng	Class object.
evidences	Cell array of cell arrays of evidences.

Discussion

This function sets statistical data for learning. The data has the form of an array where the number of rows equals to the number of series and the number of elements in a row - to the number of time slices for each series.

GetDynamicModel

Returns model of learning engine.

```
grModel = GetDynamicModel( learnEng );
```

Arguments

learnEng	Class object.
grModel	Graphical model.

Class CEMLearningEngineDBN

Class `CEMLearningEngineDBN` is used in the learning of Dynamic Bayesian Networks. The learning is based on *Expectation Maximization* (EM) algorithm.

CEMLearningEngineDBNCreate

Creates class object.

```
CEMLearningEngineDBNCreate( DBN );
CEMLearningEngineDBNCreate( DBN, infEng );
```

Arguments

DBN	Graphical model to be trained.
infEng	Inference engine. By default `C1_5SliceJtreeInfEngine` is used.

SetTerminationToleranceEM

Sets termination tolerance.

```
SetTerminationToleranceEM( learnEng, precision );
```

Arguments

learnEng	Class object.
precision	Float value of precision. To determine the breakpoint of the learning procedure, you compare the value of precision with the difference between logarithms of likelihoods of two neighboring steps.

Discussion

This function sets the termination condition for EM. EM procedure stops when the difference between the logarithm of likelihood value at a current step and at its prior step does not exceed the value of precision.

SetMaxIterEM

Sets maximum iteration depth for EM.

```
SetMaxIterEM( learnEng, nIter );
```

Arguments

learnEng	Class object.
nIter	Maximum iteration depth.

Discussion

This function sets the maximal number of iterations to be performed in the learning process.

SetTerminationToleranceEM

Sets termination tolerance.

```
SetTerminationToleranceEM( learnEng, precision );
```

Arguments

learnEng	Class object.
precision	Precision.

Discussion

This function sets the termination condition for EM. EM stops when the difference between the logarithm of likelihood at a current step and at its preceding step does not exceed the value of precision.

Random Number Generation

pnlSeed

Reinitializes random number generator.

```
pnlSeed(s);
```

Arguments

s	Integer that reinitializes the initial random number generator.

Discussion

This function reinitializes the initial random number generator. As RNG is initialised automatically on loading the library, you call this function only if there is some need for it. For example, you may call it to perform non-repeatable experiments in different calls of application.

pnlRand

Generates random numbers uniformly distributed over specified numerical interval.

```
pnlRand(left, right);
```

Arguments

left	Left boundary of the interval.
right	Right boundary of the interval.

Discussion

This function generates random numbers uniformly over a specified numerical interval.

pnlRandNormal

Generates normally distributed random numbers.

```
pnlSeed(mean, cov);
```

Arguments

mean	Mean of the normal distribution.
cov	Covariance matrix of the normal distribution.

Discussion

This function generates a vector from a multivariate normal distribution.

Index

Bibliography

[BKUAI98] X. Boyen and D. Koller. *Tractable Inference for Complex Stochastic Processes*, UAI 98

[BKNIPS98] X. Boyen and D. Koller. *Approximate learning of dynamic models*, NIPS 98

[CDLS] R.G. Cowell, A. P. Dawid, S. L. Lauritzen, and D. J. Spiegelhalter. *Probabilistic Networks and Expert Systems*. Springer-Verlag New York, 1999.

[Dempster] A. Dempster, N. Laird, D. Rubin. *Maximum Likelihood From Incomplete Data via the EM Algorithm*. Journal of the Royal Statistical Society, B 39: 1-38, 1977.

[H] G. B. Horn. *Iterative Decoding and Pseudocodewords*. Ph.D. thesis, Department of Electrical Engineering, California Institute of Technology, Pasadena, CA, May 1999.

[Introd] A Brief Introduction to Graphical Models and Bayesian Networks. http://www.ai.mit.edu/~murphyk/Bayes/bnintro.html

[Jirousek] R. Jirousek and S. Preucil. *On the Effective Implementation of the Iterative Proportional Fitting Procedure*. Computational Statistics Quarterly, 4: 269-282, 1990.

[JorBish] M. Jordan, C. Bishop. *An Introduction to Graphical Models*.

[Jordan] M.I. Jordan, editor. *Learning in Graphical Models*. MIT Press, 1999.

[LS] S. L. Lauritzen and D. J. Spiegelhalter. *Local Computations With Probabilities on Graphical Structures and Their Applications to Expert Systems.* J. Roy. Stat. Soc. B, 50, 157-224, 1988.

[Murphy98] K. Murphy. *Inference and Learning in Hybrid Bayesian Networks*. EECS, University of California, 1998.

[Murphy02] K. P. Murphy. *Dynamic Bayesian Networks: Representation, Inference and Learning*, PhD thesis. UC Berkeley, Computer Science Division, July 2002.

[MWJ] K. P. Murphy, Y. Weiss, and M. I. Jordan. *Loopy Belief Propagation for Approximate Inference: An Empirical Study*. Uncertainty in Artificial Intelligence, to appear.

[P1] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, 1988.

[WF2000] Y. Weiss, and W. T. Freeman. *Correctness of Belief Propagation in Gaussian Graphical Models of Arbitrary Topology*. In S. Solla, T. K. Lean, and K.R. Muller, editors, *Advances in Neural Information Processing Systems*, 12, 2000.

[WF2001] Y. Weiss, and W. T. Freeman. *On the Optimality of Solutions of the Max-Product Belief Propagation Algorithm in Arbitrary Graphs*. IEEE Transactions on Information Theory, 47:2, 723-735, 2001.