intel®

Probabilistic Network Library

User Guide and
Reference Manual

Version	Version History	Date
-001	Original Issue	July, 2002
-002	Changed the title: the former title "Probabilistic Graphical Models Toolkit"; edited and amended User Guide; all prefixes in function names changed from CPGM- and EPGM- to C- and E- respectively; changed names of 8 classes; made numerous changes in desription of functions and in examples; added 50% new functions and operators.	February, 2003
-003	Major functionality updates	December, 2003
-004	Functionality updates	March, 2004

Contents

Contents

intel®

intel.

Bibliography

Index

Overview

This manual describes *Probabilistic Network Library* (*PNL*), 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 PNL core engine will be optimized and paralllized 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.

intel.

PNL is a library implemented in C++, exporting C++ API, and some preliminary version of MATLAB API. The library is currently available for Windows. Linux port is also planned.

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 PNL 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 PNL 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

Graphical Models

A *probabilistic graphical model* (*PGM*) is a factorized joint probability distribution over a set of random variables termed a *model domain*. Each *factor* is a function defined on some small subset of variables and such subsets are called *factor domains*. 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. The nodes of the 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.

PNL 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 set of nodes in the graph, which form a clique.

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

A factor in a BNet has a meaning of *conditional probability distribution* (*CPD*) of the child node, given the parent nodes. In this context a directed edge from node *A* to node *B* may be interpreted as a causal relationship though absence of the edge does not mean the nodes are statistically independent.

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

A graphical model in terms of PNL is created by the following routine, which is shown here for a Bayesian network model only, but also true for Markov Networks, if implemented with minor changes. The model represented is called "water-sprinkler". The graph structure of the model and the parameters (CPDs in this case) are all shown in Figure 2-1:

Figure 2-1   Water-sprinkler model
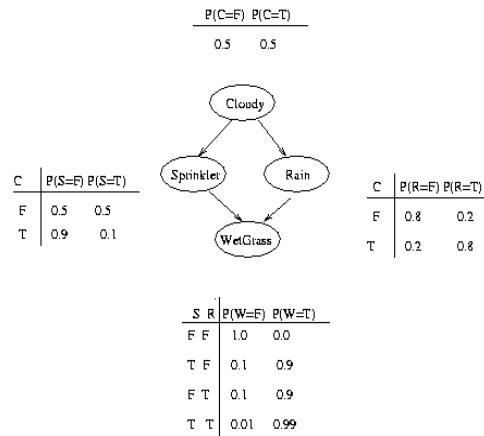
The nodes are numbered as follows:

Cloudy (C) = 0;

Sprinkler (S) = 1;

Rain (R) = 2;

Wet Grass (W) = 3

PNL has special containers for storing scalar and vector data. Value object is designed to store inhomogeneous scalar data used for evidence. `pnlVector` is a template intended for storing vector data. For the sake of brevity PNL defines several synonyms to specializations of `pnlVector`. For more details please see Reference Manual.

.

Example 2-1   Creation of water sprinkler Bayesian network

```
const int numOfNds = 4;

 // 1 STEP:
```

intel₂

2-3

Example 2-1   Creation of water sprinkler Bayesian network

```
// need to specify the graph structure of the model;
// there are two way to do it

CGraph *pGraph;
if(1)
{
    // Graph creation using adjacency matrix

    int numAdjMatDims  = 2;

    int ranges[] = { numOfNds, numOfNds };

    intVector matrixData( numOfNds*numOfNds, 0 );

    CDenseMatrix<int>* adjMat = CDenseMatrix<int>::Create( numAdjMatDims,
        ranges, &matrixData.front() );

    int indices[] = { 0, 1 };

    adjMat->SetElementByIndexes( 1, indices );

    indices[1] = 2;
    adjMat->SetElementByIndexes( 1, indices );

    indices[0] = 1;
    indices[1] = 3;
    adjMat->SetElementByIndexes( 1, indices );

    indices[0] = 2;
    adjMat->SetElementByIndexes( 1, indices );

    // this is a creation of directed graph for the BNet model based on
//adjacency matrix
    pGraph = CGraph::Create(adjMat);
}
else
{
    // Graph creation using neighbors list

    int numOfNbrs[numOfNds] = { 2, 2, 2, 2 };
    int nbrs0[] = { 1, 2 };
    int nbrs1[] = { 0, 3 };
    int nbrs2[] = { 0, 3 };
    int nbrs3[] = { 1, 2 };

    // number of neighbors for every node
    int *nbrs[] = { nbrs0, nbrs1, nbrs2, nbrs3 };

    // neighbors can be of either one of the three following types:
```

Example 2-1   Creation of water sprinkler Bayesian network

```
      // a parent, a child (for directed arcs) or just a neighbor (for
//undirected graphs).
      // Accordingly, the types are ntParent, ntChild or ntNeighbor.

      ENeighborType nbrsTypes0[] = { ntChild, ntChild };
      ENeighborType nbrsTypes1[] = { ntParent, ntChild };
      ENeighborType nbrsTypes2[] = { ntParent, ntChild };
      ENeighborType nbrsTypes3[] = { ntParent, ntParent };

      ENeighborType *nbrsTypes[] = { nbrsTypes0, nbrsTypes1,
          nbrsTypes2, nbrsTypes3 };

      // this is creation of a directed graph for the BNet model using
neighbors list
      pGraph = CGraph::Create( numOfNds, numOfNbrs, nbrs, nbrsTypes );
    }

    // 2 STEP:
    // Creation NodeType objects and specify node types for all nodes of the
model.

    nodeTypeVector  nodeTypes;

    // number of node types is 1, because all nodes are of the same type
    // all four are discrete and binary
    CNodeType nt(1,2);
    nodeTypes.push_back(nt);

    intVector nodeAssociation;
    // reflects association between node numbers and node types
    // nodeAssociation[k] is a number of node type object in the
    // node types array for the k-th node
    nodeAssociation.assign(numOfNds, 0);

    // 2 STEP:
    // Creation base for BNet using Graph, types of nodes and nodes association

    CBNet* pBNet = CBNet::Create( numOfNds, nodeTypes, nodeAssociation, pGraph
);

    // 3 STEP:
    // Allocation space for all factors of the model
    pBNet->AllocFactors();

    // 4 STEP:
    // Creation factors and attach their to model

    //create raw data tables for CPDs
```

Example 2-1   Creation of water sprinkler Bayesian network

```
float table0[] = { 0.5f, 0.5f };
float table1[] = { 0.5f, 0.5f, 0.9f, 0.1f };
float table2[] = { 0.8f, 0.2f, 0.2f, 0.8f };
float table3[] = { 1.0f, 0.0f, 0.1f, 0.9f, 0.1f, 0.9f, 0.01f, 0.99f };

float* table[] = { table0, table1, table2, table3 };

int i;
for( i = 0; i < numOfNds; ++i )
{
    pBNet->AllocFactor(i);

    CFactor* pFactor = pBNet->GetFactor(i);

    pFactor->AllocMatrix( table[i], matTable );
```

```
}
```

NOTE. *If the graph structure is not known by the time the model is created, user can create an empty graph specifying all the arguments of the* CGraph::Create *member function equal to* NULL. *Then user can first call the member function of the* CGraph *class named* AddNodes() *to specify the number of nodes in the graph and either add edges one by one by the call of the member function* AddEdge(), *or set all the neighbors for each node by the call of the member function* SetNeighbors(). *See the declarations of all these functions below.*

- CGraph::AddNodes(int *newNumOfNds*);
- CGraph::SetNeighbors(int *nodeNum*, int *numOfNbrs*, const int* *nbrs*, const ENeighborType **nbrsTypes*);
- CGraph::AddEdge(int *startNode*, int *endNode*, int *directed*);
- CGraph::RemoveEdge(int *startNode*, int *endNode*);

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 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 in which nodes from the first layer have no parameters associated with them and determine the system at the previous moment of time while each node from the second layer has conditional probabilities ().

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

Prior Network   Transition Network
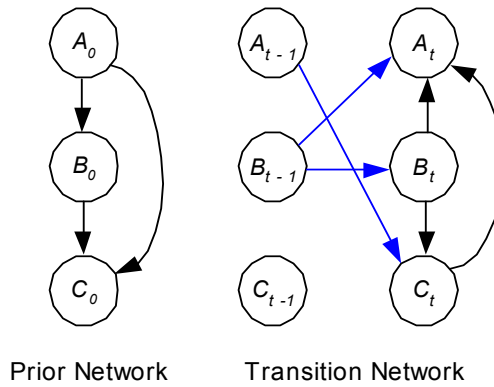
The semantics of DBN can be defined by unrolling the 2TBN for *T* time slices. The resulting joint probability distribution is then defined by expression

$$P(x_{0:T-1}) = \prod_{t=0}^{T-1} \prod_{i=0}^{n-1} P(x_t^i \mid \pi(x_t^i)), \text{ where } \pi(x_t^i) \text{ means parents of } x_t^i, \text{ that is, } i^{th}$$

node in t^{th} time-slice, n is the number of nodes.

The Dynamic Bayesian network storage in terms of PNL is implemented similarly to the Bayesian network storage. Suppose, the prior network consists of *n* nodes. The network, which is stored internally to represent the Dynamic Bayesian network, will then consist of 2*n* nodes, where the first *n* nodes are joint in one Graph to represent the topology of the prior network and nodes with numbers starting with *n* to 2*n-1* are joint in a Graph, which represents the ith slice, where $i > 0$. The joint Graph is represented

by those two layers (prior and i^{th}) and is obtained after joining them together.
Figure 2-3 shows a Bayesian network constructed by unrolling for 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
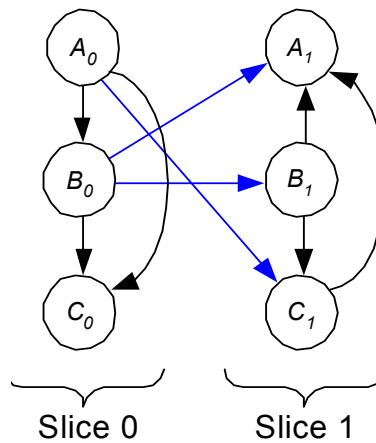
The following routine creates a DBN in terms of PNL:

Example 2-2

```
Creation of DBN model
X0 ->X1
|      |
v      v
Y0 ->Y1

all nodes are discrete and binary
*/

    //Create static model
    const int nnodes = 4;//Number of nodes

    // 1) First need to specify the graph structure of the model;
    int numOfNeigh[] = {2, 2, 2, 2};
    int neigh0[] = {1, 2};
    int neigh1[] = {0, 3};
```

Example 2-2

```
int neigh2[] = {0, 3};
int neigh3[] = {1, 2};

ENeighborType orient0[] = { ntChild, ntChild };
ENeighborType orient1[] = { ntParent, ntChild };
ENeighborType orient2[] = { ntParent, ntChild };
ENeighborType orient3[] = { ntParent, ntParent };

int *neigh[] = { neigh0, neigh1, neigh2, neigh3 };
ENeighborType *orient[] = { orient0, orient1, orient2, orient3 };


CGraph* pGraph = CGraph::Create( nnodes, numOfNeigh, neigh, orient);

// 2) Creation of the Model Domain.

nodeTypeVector variableTypes;
const int numNt = 1;//number of Node types (all nodes are discrete)
variableTypes.resize(numNt);
variableTypes[0].SetType(1, 2);

intVector variableAssociation;
variableAssociation.assign(nnodes, 0);

CModelDomain *pMD;
pMD = CModelDomain::Create( variableTypes, variableAssociation );


// 3) Creation static BNet with random matrices
CBNet *pBNet = CBNet::CreateWithRandomMatrices( pGraph, pMD );


// 4) Creation DBN
CDBN *pDBN = CDBN::Create( pDBN );
```

Inference Algorithms for Bayesian and Markov Networks

An 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 when certain nodes of the graphical model are observed:

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

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

intel₀

This problem has several possible solutions. The most evident of them is 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:

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

By definition, this joint probability distribution can be found by multiplying all conditional probability distributions of a Bayesian network or all joint probability distributions at the cliques of a Markov network. Before multiplication is started, these conditional and unconditional distributions should be adjusted according to the values at the observed nodes of the network. The final step is to sum up the resulting values.

This description fully applies to the `NaiveInfEngine`.

See [Example 2-3](#) of call of such inference engine for the "water-sprinkler" model ([Figure 2-1](#)).

Example 2-3   Inference engine creation water sprinkler BNet

```
//create Water - Sprinkler BNet
    CBNet* pWSBnet =
pnlExCreateWaterSprinklerBNet();//CreateWaterSprinklerBNet();

    //get content of Graph
    pWSBnet->GetGraph()->Dump();

    //create simple evidence for node 0 from BNet
    CEvidence* pEvidForWS;
  //make one node observed
    int nObsNds = 1;
    //the observed node is 0
    int obsNds[] = { 0 };
    //node 0 takes its second value (from two possible values {0, 1})
    valueVector obsVals;
    obsVals.resize(1);
    obsVals[0].SetInt(1);
    pEvidForWS = CEvidence::Create( pWSBnet, nObsNds, obsNds, obsVals
);
```

Example 2-3   Inference engine creation water sprinkler BNet

```
//create Naive inference for BNet
CNaiveInfEngine* pNaiveInf = CNaiveInfEngine::Create( pWSBnet );

//enter evidence created before
pNaiveInf->EnterEvidence( pEvidForWS );

//get a marginal for query set of nodes
int numQueryNds = 2;
int queryNds[] = { 1, 3 };

pNaiveInf->MarginalNodes( queryNds, numQueryNds );
const CPotential* pMarg = pNaiveInf->GetQueryJPD();


 intVector obsNds;
pConstValueVector obsVls;
pEvidForWS->GetObsNodesWithValues(&obsNds, &obsVls);

int i;
for( i = 0; i < obsNds.size(); i++ )
{
 std::cout<<" observed value for node "<<obsNds[i];
 std::cout<<" is "<<obsVls[i]->GetInt()<<std::endl;
}

int nnodes;
const int* domain;
pMarg->GetDomain( &nnodes, &domain );
std::cout<<" inference results: \n";

std::cout<<" probability distribution for nodes [ ";

for( i = 0; i < nnodes; i++ )
{
 std::cout<<domain[i]<<" ";
}

std::cout<<"]"<<std::endl;

CMatrix<float>* pMat = pMarg->GetMatrix(matTable);

// graphical model hase been created using dense matrix
// so, the marginal is also dense
EMatrixClass type = pMat->GetMatrixClass();
if( ! ( type == mcDense || type == mcNumericDense || type ==
mc2DNumericDense ) )
{
 assert(0);
}
```

Example 2-3   Inference engine creation water sprinkler BNet

```
    int nEl;
    const float* data;
    static_cast<CNumericDenseMatrix<float>*>(pMat)->GetRawData(&nEl,
&data);
    for( i = 0; i < nEl; i++ )
    {
      std::cout<<" "<<data[i];
    }
    std::cout<<std::endl;


    delete pEvidForWS;
    delete pNaiveInf;
    delete pWSBnet;
```

This direct computation, however, is too laborious, as the complexity grows exponentially with respect to the number of network nodes, and becomes ineffective even for small models. For this reason it is very seldom used in practice.

The idea, which helps to reduce the complexity of computation, is the distribution law. Since local distributions in certain parts of the network are independent of the variables in other parts, the distribution law can be applied to calculate distributions for the query nodes. For instance, for the "water-sprinkler" problem the probability distribution at node 3 with no observed variables can be expanded in the following way:

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

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

This is the idea underlying some exact and approximate inference engines.

Initially each component of the network, which may be either a single node or a group of nodes combined together, is assigned a certain distribution function, representing the assumed node values of the network. Then these functions are modified in the course of iterative message passing between the neighboring components of the network. Generally, two components can be neighbors in terms of one inference engine and

intel®

non-neighbors in terms of another. Order 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 the graph of Bayesian or Markov network is a tree, the most obvious network components are the nodes. Following this approach, neighboring nodes in the graph are also neighbors in the network. This is called *Pearl Inference* or *Belief Propagation*, which is exact. If the graph contains undirected cycles, then assuming nodes as network components helps to get an approximate result; to improve the approximation, the number of iterations should be increased. This algorithm is often referred to as Loopy Belief Propagation. However, in certain cases it may not converge or converge to a local minimum [MWJ], [H], yet it was 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:

```
CInferenceEngine* pInfEngine = CPearlInfEngine::Create( pGraphModel );
```

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

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

Particle-based Inference

Besides exact inference engines, for example, 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 samples a set of approximations. They represent a part of the probability mas and are called *particles*. Particle-based approximate inference engine can calculate the query potential and estimate real states of query nodes. Commonly used particle-based methods are `LWSamping`, `GibbsSampling` and `ParticleFiltering`.

A particle-based inference engine is easily implemented and works with a wide range of models, among them non-linear, non-Gaussian CPDs. It converts heuristics to provably correct algorithms by using them as proposal distributions and under the

restriction of the number of particles, it provides the exact answer. For the dynamic adaptation among multi-cue observations the engine provides an intuitive fusion mechanism.

Inference Algorithms for DBNs

An inference problem in the context of a dynamic graphical model is equivalent to the marginal estimation of one or several nodes from some slices irrespective of whether certain nodes of certain slices are observed or hidden, that is, to compute $P(x(i, t)|y(:, t_1:t_2))$, where $x(i, t)$ represents the ith 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. Computing joint distributions of variables over one or more time slices is also often needed.

Several types of inference problems are distinguished 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	$\max\limits_{x(1:\,t)} P(x(1:\,t)|y(1:\,t))$ Off-line procedure to compute the most likely sequence of hidden states, given the data.
Prediction	$P(x(t+dt)|y(1:\,t))$ 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 necessary to learn the parameter values, inference, which is smoothing in this case, should be called for a BNet that is as long as the observations sequence. If the sequences of evidences are of variable lengths, junction trees (for Junction Tree Inference) should be constructed every time, which extremely slows down the process, or precompute and store junction trees 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 idea of the algorithm is to unroll a DBN once to some $T_{\max}$ slices, to create a junction tree and splice 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.

intel®

2. Create a junction tree for the obtained network.

3. Link up all the junction trees via interfaces.

This algorithm is able to do 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 Boyen-Koller inference (BK). BK inference is the approximate inference in which the belief state of the interface clique (clique consists of interface nodes 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 not be independent. For details, see [BKUAI98] and [BKNIPS98]. The first paper discusses filtering and theory while the NIPS98 paper discusses smoothing. Note that the exact 1.5 Slice Junction tree inference is the special case of BK inference.

Example 2-4   Creation of inference algorithm for DBN

```
CBNet *pBNetForArHMM = pnlExCreateRndArHMM();
   CDBN *pArHMM = CDBN::Create( pBNetForArHMM );

   //Create an inference engine
   C1_5SliceJtreeInfEngine* pInfEng;
   pInfEng = C1_5SliceJtreeInfEngine::Create(pArHMM);

   //Number of time slices for unrolling
   int nTimeSlices = 5;
   const CPotential* pQueryJPD;

   //Crate evidence for every slice
   CEvidence** pEvidences;
   pEvidences = new CEvidence*[nTimeSlices];

   //Let node 1 is always observed
   const int obsNodesNums[] = { 1 };
   valueVector obsNodesVals(1);

   int i;
   for( i = 0; i < nTimeSlices; i++ )
   {
       // Generate random value
       // all nodes in the model are discrete
       obsNodesVals[0].SetInt(rand()%2);
```

int_{el} ®

Example 2-4   Creation of inference algorithm for DBN

```
    pEvidences[i] = CEvidence::Create( pArHMM, 1, obsNodesNums,
        obsNodesVals );
}

// Create smoothing procedure
pInfEng->DefineProcedure(ptSmoothing, nTimeSlices);
// Enter created evidences
pInfEng->EnterEvidence(pEvidences, nTimeSlices);
// Start smoothing process
pInfEng->Smoothing();

// Choose query set of nodes for every slice
int queryPrior[] = { 0 };
int queryPriorSize = 1;
int query[] = { 0, 2 };
int querySize = 2;

// inference results gaining and representation
std::cout << " Results of smoothing " << std::endl;

int slice = 0;
pInfEng->MarginalNodes( queryPrior, queryPriorSize, slice );
pQueryJPD = pInfEng->GetQueryJPD();

std::cout<<"Query slice"<<slice<<std::endl;

int nnodes;
const int* domain;
pQueryJPD->GetDomain( &nnodes, &domain );

std::cout<<" domain :";

for( i = 0; i < nnodes; i++ )
{
    std::cout<<domain[i]<<" ";
}

std::cout<<std::endl;
CMatrix<float>* pMat = pQueryJPD->GetMatrix(matTable);

// graphical model hase been created using dense matrix
std::cout<<" probability distribution \n";

int nEl;
const float* data;
static_cast<CNumericDenseMatrix<float>*>(pMat)->GetRawData(&nEl,
&data);

for( i = 0; i < nEl; i++ )
```

Example 2-4   Creation of inference algorithm for DBN

```
{
    std::cout<<" "<<data[i];
}

std::cout << std::endl;

for( slice = 1; slice < nTimeSlices; slice++ )
{
    pInfEng->MarginalNodes( query, querySize, slice );
    pQueryJPD = pInfEng->GetQueryJPD();

    std::cout<<"Query slice"<<slice<<std::endl;
    // Representation information using Dump()
    pQueryJPD->Dump();
}

slice = 0;

//Create filtering procedure
pInfEng->DefineProcedure( ptFiltering );
pInfEng->EnterEvidence( &(pEvidences[slice]), 1 );
pInfEng->Filtering( slice );

pInfEng->MarginalNodes( queryPrior, queryPriorSize );
pQueryJPD = pInfEng->GetQueryJPD();

std::cout<<" Results of filtering " << std::endl;
std::cout<<" Query slice "<<slice<<std::endl;
pQueryJPD->Dump();


for( slice = 1; slice < nTimeSlices; slice++ )
{
    pInfEng->EnterEvidence( &(pEvidences[slice]), 1 );
    pInfEng->Filtering( slice );

    pInfEng->MarginalNodes( query, querySize );
    pQueryJPD = pInfEng->GetQueryJPD();

    std::cout<<" Query slice "<<slice<<std::endl;
    pQueryJPD->Dump();
}

//Create fixed-lag smoothing (online)
int lag = 2;
pInfEng->DefineProcedure( ptFixLagSmoothing, lag );

for (slice = 0; slice < lag + 1; slice++)
{
```

Example 2-4   Creation of inference algorithm for DBN

```
      pInfEng->EnterEvidence( &(pEvidences[slice]), 1 );
}


pInfEng->FixLagSmoothing( slice );

pInfEng->MarginalNodes( queryPrior, queryPriorSize );
pQueryJPD = pInfEng->GetQueryJPD();

std::cout<<" Results of fixed-lag smoothing " << std::endl;
std::cout<<" Query slice "<<slice<<std::endl;
pQueryJPD->Dump();

std::cout << std::endl;

for( ; slice < nTimeSlices; slice++ )
{
      pInfEng->EnterEvidence( &(pEvidences[slice]), 1 );
    pInfEng->FixLagSmoothing( slice );

    pInfEng->MarginalNodes( query, querySize );
    pQueryJPD = pInfEng->GetQueryJPD();

    std::cout<<" Query slice "<<slice<<std::endl;
    pQueryJPD->Dump();
}

delete pInfEng;

for( slice = 0; slice < nTimeSlices; slice++)
{
    delete pEvidences[slice];
}
delete pArHMM;
```

Dynamic Bayesian networks are created by BNet with *2n* nodes, that is a DBN, unrolled in the form of 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 of nodes with numbers from *0* to *n-1*.

intel®

2-20

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 nodes of the preceding slice.

Learning for Bayesian and Markov Networks

A graphical model can be defined by its structure and the set of parameters that are *conditional probability distributions* for Dynamic and Static Bayesian networks and *potentials* for Markov network. A learning task of a graphical model is a task of estimating model factors, so they give the best explanation of information available for the model.

Most often, the input data for the learning task is presented in a table, where columns correspond to variables of the model and each row represents learning sample, or observation. For example, Table 2-2 shows cases 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

Certain variables could be unobserved, which means that their values are missing in the learning data. The 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

intel®

Table 2-3   Types of Learning Tasks (continued)

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

> **NOTE.** *Only the first three types of learning tasks are considered below and relative examples are given. Type 4 is not supported by the current version of PNL.*

Type 1

The algorithm used in this learning task is called ML algorithm and is based on *Maximum Likelihood Estimation* [JorBish].

The method finds estimate for parameters of the graphical model maximizing the value of likelihood function $p(D|\theta)$, that is, probability of observing 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 full set 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 $m(x_A)$. Logarithm of the previously described likelihood function is more convenient to use instead of the function itself, in which case the logarithm can be found as follows:

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

![intel]

$$= \sum_{x_U} m(x_U)\log p(x_U|\theta) = \sum_{v} \sum_{x_{\phi_v}} m(x_{\phi_v})\log\theta_v(x_{\phi_v})$$

The values that maximize this function are as follows:

$$\hat{p}(x_v|x_{\pi_v}) = \hat{\theta}_v(x_{\phi_v}) = \frac{m(x_{\phi_v})}{m(x_{\pi_v})} \quad .$$

These estimates are formed independently at each of the nodes in the graph.

Multivariate Gaussian Case. In PNL the Multivariate Gaussian case is implemented only for a Bayesian network.

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

The proposed approach is to model the joint distribution over a node and its parents as multivariate Gaussian distribution and then find its Ml estimation. The sufficient statistics after N examples are [Murphy98], [Jordan]:

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

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

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

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

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

where matrix B is broken up 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 the potentials are defined on maximal cliques in the graph, maximum likelihood estimates for decomposable graphs can be found by inspection in the following way:

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]. When 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 ith 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:

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

In PNL 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 PNL supports only a two-level hierarchy in discrete tabular distributions.

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

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

According to the Bayes formula

$$\theta | D \rangle = \frac{P\langle D| \theta \rangle * P\langle \theta \rangle}{P\langle D \rangle},$$

where $\langle D \rangle = \int P\langle \theta \rangle P\langle D| \theta \rangle$.

Based on the given parameter distribution function, the distribution function for the unknown variable x is $\langle x \rangle = \int P\langle \theta \rangle P\langle x| D \rangle a$

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.

intel₍ᵣ₎

Let training data contain $N_1 + ... + N_n$ 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 $P(x = i) = \dfrac{a_i + N_i}{\sum_{n} \langle a_k + N_k \rangle}$.

This discussion applies to the case of an unconditional distribution where the considered node of 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 $P(x = i) | parents \langle x \rangle = i) = \dfrac{a_{ij} + N_{ij}}{\sum_{n} \langle a_{ik} + N_{ik} \rangle}$.

Type 2

This problem can be solved iteratively by using *Expectation Maximization* (EM) algorithm [Dempster], [Jordan]. The idea of the algorithm is as follows: first, assume the initial state of parameters θ^0 and then start the iterative process alternately repeating two steps: E-step and M-step.

Consider the process at 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   The new value of θ^i is found that maximizes the value of the likelihood function. E-step is repeated with the new parameter values.

This process converges to a local maximum.

In *PNL* 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 Bayesian network defined in the sprinkler example (see Figure 2-1). If all the nodes are observed, it is Type 1 learning. In this case E-step, which creates an inference engine and performs the inference procedure, does not need to be called. A learning task is of Type 2, if some nodes are hidden. Then the procedure creates an instance of inference engine, which is a junction tree engine by default, and uses it in the E-step.

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

```
 CBNet* pWSBnet = pnlExCreateWaterSprinklerBNet();

//create WS BNet with different matrices
 std::cout<<"Learning procedure \n ";
 CGraph *pGraph = CGraph::Copy( pWSBnet->GetGraph() );
 CModelDomain *pMD = pWSBnet->GetModelDomain();

 CBNet* pWSLearnBNet = CBNet::CreateWithRandomMatrices( pGraph, pMD );

 //loading data from file
 const char * fname = "..\\c_pgmtk\\examples\\Data\\casesForWS";

 pEvidencesVector evVec;

 if( ! pnlLoadEvidences(fname,  &evVec, pMD) )
 {
     printf("can't open file with cases");
     exit(1);
 }
 int numOfSamples = evVec.size();
 std::cout<<"Number of cases for learning = "<<numOfSamples<<std::endl;

 //create learning engine
 CEMLearningEngine *pLearn = CEMLearningEngine::Create( pWSLearnBNet );

 //set data for learning
 pLearn->SetData( numOfSamples, &evVec.front() );
 pLearn->Learn();

 //get information from learned model
 int nFactors = pWSLearnBNet->GetNumberOfFactors();
 const CFactor *pCPD;
 const CNumericDenseMatrix<float> *pMatForCPD;
 int numOfEl;
 const float *dataCPD;
```

intel

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

```
int f;
for( f = 0; f < nFactors; f++ )
{
    std::cout<<std::endl<<" probability distribution for node
        "<<f<<std::endl;
    pCPD = pWSLearnBNet->GetFactor(f);
    //all matrices are dense
    pMatForCPD = static_cast<CNumericDenseMatrix<float> *>
        (pCPD->GetMatrix(matTable));
    pMatForCPD->GetRawData( &numOfEl, &dataCPD );

    int j;
    for( j = 0; j < numOfEl; j++ )
    {
        std::cout<<" "<<dataCPD[j];
    }
}
std::cout<<std::endl;

int ev;
for( ev = 0; ev < evVec.size(); ev++ )
{
    delete evVec[ev];
}
delete pLearn;
delete pWSBnet;
delete pWSLearnBNet;
```

After this procedure, the parameters of the Bayesian network assume new values that maximize the likelihood function. The new values correspond to the array of learning data in the best way.

A new table with cases may be used in further training in the following two ways:

Option 1. Ignore data from the previous learning. Use <u>SetData</u> member function to implement the variant.

Example 2-6   Entering New Data

```
// entering new data and clear accumulated information.
// here pEvNew is the pointer to a newly created array of Evidences
pLearn->SetData( newNumOfCases, pEvNew)
// call learning
pLearn->Learn();
```

intel®

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

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

Example 2-7   Using data from previous learning

```
pLearn->AppendData( newNumOfCases, pEvNew )
pLearn->Learn();
```

Type 3

The current version of *PNL* carries out structure learning for static and dynamic `BNets` and does not support other models. The learning engine calls Maximal Likelihood parameter learning. In this version of PNL 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, Gaussian and conditional Gaussian 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 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]:

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

where θ stands for 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 network parameters with the modelled data, and the addend reflects the descriptive complexity of the network. Vector θ^* can be found from the following condition:

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

Structure Search Method

The structure learning algorithm for *PNL* Bayesian networks employs the hill-climbing search. It scores all neighbors of the current structure (DAG) and moves to the neighbor with the highest score. The problem of selecting the best Bayesian network from all the network configurations is NP hard. Among heuristic iterative algorithms that solve this problem there are Greedy Hillclimbing, Best-first, and algorithms based on Monte Carlo methods. One of the algorithms implemented in *PNL* 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 the number of node permutations is $N!$. The total number of Bayesian

networks with the topology is $N! \cdot 2^{\frac{N(N-1)}{2}}$

The hill-climbing search is local: on reaching a local maximum the learning procedure stops. To implement search it requires an initial structure. You may choose a number of different initial structures (starting points) and run the hill-climbing search for each of them. You can set the search procedure to run several times for all itructures but for the first that you specify. The system will randomly generate other legal starting points for learning and return the best structure.

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

Example 2-8   Structure Learning for Bayesian Network Using PNL

```
// create an empty graph with number of nodes numOfNds
int numOfNds = 4;
CGraph *pGraph = CGraph::Create( numOfNds, NULL, NULL, NULL );
// here the user has to set all other variable values necessary to
//create a Bayesian network with an empty graph (the number of node
// types, actual node types, etc.)
const int numOfNdTypes = 1;
// number of node types is 1, because all nodes are of the same type
// all four are discrete and binary

CNodeType *nodeTypes = new CNodeType [numOfNdTypes];
int *nodeAssociation = new int [numOfNds];

nodeTypes[0].SetType(1, 2);// node type - discrete and binary
```

Example 2-8   **Structure Learning for Bayesian Network Using PNL**   (continued)

```
int i;
for( i = 0; i < numOfNds; ++i )
{
        nodeAssociation[i] = 0;
}

pBNet = CBNet::Create( numOfNds, numOfNdTypes, nodeTypes, nodeAssociation,
pGraph );
pBNet->AllocFactors();
for( i = 0; i < numOfNds; i++ )
 {
   pBNet->AllocFactor(i);
   pBNet->GetFactor(i)->CreateAllNecessaryMatrices();
}
// create learning engine
CBICLearningEngine *pLearn = CBICLearningEngine::Create( pBNet );

// set input data
pLearn->SetData(numOfCases, pEv);

//start learning
pLearn->Learn();

// the output Bayesian network
const CBNet *pFinalBNet = NULL;

pFinalBNet= static_cast<const CBNet *>(pLearn->GetGraphicalModel());

// the output graphical model is sorted topologically
// get the relation to initial node numeration
cost int *reordering = pLearn->GetOrder();
```

Learning for DBNs

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

intel

See [Example 2-9](#) of calling a parameter learning engine for DBN.

Example 2-9   Calling of parameter learning engine for DBN

```
CBNet *pBNetForArHMM = pnlExCreateRndArHMM();
CDBN *pArHMM = CDBN::Create( pBNetForArHMM );
      //Create learning procedure for DBN

    pEvidencesVecVector evidencesOut;

    const int nTimeSeries = 500;
    intVector nSlices(nTimeSeries);
    //define number of slices in the every time series
    pnlRand(nTimeSeries, &nSlices.front(), 3, 20);
    // Generate evidences in a random way
    pArHMM->GenerateSamples( &evidencesOut, nSlices);

    // Create DBN for learning
    CDBN *pDBN = CDBN::Create(pnlExCreateRndArHMM());

    // Create learning engine
    CEMLearningEngineDBN *pLearn = CEMLearningEngineDBN::Create( pDBN );

    // Set data for learning
    pLearn->SetData( evidencesOut );

    // Start learning
    try
    {
        pLearn->Learn();
    }
    catch(CAlgorithmicException except)
    {
        std::cout << except.GetMessage() << std::endl;
    }
    //Ïîëó÷åíå ðåçóëüòàòîâ îáó÷åíå, îòîáðàæåíå
    int nFactors = pDBN->GetNumberOfFactors();
    const CTabularDistribFun* pDistribFun;
    const CFactor* pCPD;
    int i;
    for( i = 0; i < nFactors; i++ )
    {
        pCPD = pArHMM->GetFactor(i);

        int nnodes;
        const int* domain;
        pCPD->GetDomain( &nnodes, &domain );

        std::cout<<" node "<<domain[nnodes -1]<<" hase the parents ";
        int node;
```

Example 2-9   Calling of parameter learning engine for DBN

```
    for( node = 0; node < nnodes-1; node++ )
    {
        std::cout<<domain[node]<<" ";
    }

    std::cout<<" Conditional probability distribution for node
"<<i<<std::endl;

    std::cout<<" initial model"<<std::endl;
     pDistribFun = static_cast<const
CTabularDistribFun*>(pCPD->GetDistribFun());
    pDistribFun->Dump();


    std::cout<<" model after learning"<<std::endl;
    pCPD = pDBN->GetFactor(i);
    pDistribFun = static_cast<const
CTabularDistribFun*>(pCPD->GetDistribFun());
    pDistribFun->Dump();

    }

    for( i = 0; i < evidencesOut.size(); i++ )
    {
        int j;
        for( j = 0; j < evidencesOut[i].size(); j++ )
        {
            delete evidencesOut[i][j];
        }
    }
    delete pDBN;
    delete pArHMM;
```

Log Subsystem

Attachment to Output and Dumping

When a `Log` object is created it is automatically attached to the output. An embedded type is dumped through `Log` interface. All strings dumped through `Log` have the prefix of the first constructor argument. The second and the third constructor arguments control filtering.

When a `Log` object is created it is automatically attached to the multiplexor and when the object is destroyed it is automatically detached from it.

Example 2-10   Code example

```
Class Point {
double x, y, z;
public:
    Point(double x_, double y_, double z_): x(x_), y(y_), z(z_) {}
    void dump() const;
};

void Point::dump() const
{
    Log out("point: ", eLOG_INFO, eLOGSRV_PNL);

    out << "x = " << x << '\n';
    out << "y = " << y << '\n';
    out << "z = " << z << '\n';
}

Point pt(1.0, 2.0, 3.0);
pt.dump();
```

Devices which ensure outputting with level `eLOG_INFO` and service `eLOGSRV_PNL` dump the following strings:

point: x = 1.0

point: y = 2.0

point: z = 3.0

Creating a Driver

When a driver is created it is automatically attached to the multiplexor. When a driver is destroyed it is automatically detached from the multiplexor.

In the current version of PNL only one driver writes to `std::ostream`: `LogDrvStream`.

Example 2-11   Code example

```
{
LogDrvStream tempStream("c:\\pnl_add.log", eLOG_RESULT| eLOG_SYSERR|
eLOG_PROGERR,
```

intel

2-34

Example 2-11   Code example

```
eLOGSRV_ALL);
Log out("demo out: ", eLOG_RESULT|eLOG_DEBUG, eLOGSRV_PNL);

out << "test string\n";
…
}
```

The driver `tempStream` is created before the implementation of the function. The driver receives the logging information that corresponds to the given *level* and *service*. Thus, for example, *out* dumps to `tempStream`.

Description of Log Subsystem Classes

Log	Implements optimization. Has no virtual functions and does not presuppose derivation.
LogMultiplexor	Implements commutation. There is only one `LogMultiplexor` and it does not have any derived classes. Generally you do not create or destroy this class.
LogDriver	Basic purely virtual class.

- **LogDrvStream**. This driver outputs to `std::ostream`. It may be created by `std::ostream`. In this case you need to delete `std::ostream` after destroing `LogDrvStream`. It may also be created by the file name. In this case the `std::ostream` is deleted by `LogDrvStream`.

- **LogDrvSystem**. This driver resembles `LogDrvStream` but is configured through `ConfigureSystem`, not through `Configure`, so that its configuration cannot be changed by `LogMultiplexor`.

Filtering Control

Filtering is controled through a pair of parameters: *level* and *service*.

- *level* identifies a type of information outputted through `Log` such as, for example, a system error message `eLOG_SYSERR` or a message with debugging information `eLOG_DEBUG`.

- *service* identifies a part of the system from which the information comes.

Table 2-4 graphically represents level and service as they specify a subset of rectangles.

Table 2-4   Parameter graphic representation

	eLOGSRV_LOG	eLOGSRV_EXCEPTION_HANDLING	eLOGSRV_POTENTIAL
eLOG_RESULT			
eLOG_SYSERR	2		2
eLOG_PROGERR		1	
eLOG_WARNING	2		2
eLOG_NOTICE	2		2
eLOG_INFO		3	3
eLOG_DEBUG		3	3

- `[eLOG_PROGERR, eLOGSRV_EXCEPTION_HANDLING]`
- `[eLOG_SYSERR|eLOG_WARNING|eLOG_NOTICE, eLOGSRV_LOG|eLOGSRV_POTENTIAL]`
- `[eLOG_INFO|eLOG_DEBUG, eLOGSRV_EXCEPTION_HANDLING|eLOGSRV_POTENTIAL]`

Reference Manual

Graph

Class CGraph
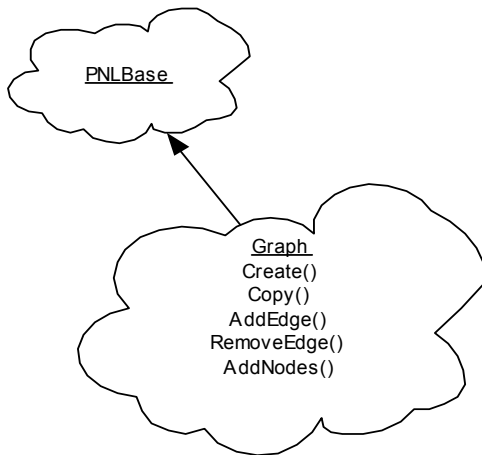
Class `CGraph` represents the graph structure of the model and carries out some basic graph algorithms.

Public Member Functions

Create

Creates object of class.

```
static CGraph* CGraph::Create( int numOfNds,  const int *numOfNbrs, const int
    *const *nbrsList, const EneighborType *const *nbrsTypes );
```

```
static CGraph* CGraph::Create( int numOfNds, const int *const *adjMat );
```

```
static CGraph* CGraph::Create( const intVecVector& nbrsList,const
    neighborTypeVecVector& nbrsTypesList );
```

```
static CGraph* CGraph::Create(const CMatrix<int>* pAdjMat);
```

Arguments

numOfNds	Number of nodes of the graph.
numOfNbrs	Array of integer, each ith integer is the number of neighbors of the ith nodes.
nbrsList	List of neighbors for a node.
nbrsTypes	2D list of neighbor types. Each element shows the type of the corresponding neighbor from *nbrsList*.
nbrsTypesList	2D list of neighbor types.
adjMat	2D array of integers, represents an adjacent matrix.
pAdjMat	2D integer matrix, represents an adjacent matrix.

Discussion

Call of this function creates an instance of class `CGraph`. Call of the class destructor deletes the instance. A node may have a neighbor of the following type: *ntParent*, *ntChild*, *ntNeighbor*.

Copy

Creates class object by copying.

```
static CGraph* CGraph::Copy( const CGraph *pGraph );
```

Arguments

pGraph   Pointer to a `CGraph` object to be copied.

Discussion

Call of this function creates a new object of class `CGraph` by copying the input object
and returns a pointer to it. Call of the class destructor deletes the instance.

GetTopologicalOrder

*Returns numbers of nodes according to their
topological order.*

```
void CGraph:: GetTopologicalOrder( intVector *order ) const;
```

Arguments

order   Returned parameter. Numbers of nodes according
                                 to their topological order.

Discussion

This function returns numbers of nodes according to the order of their topological
sorting. The function assumes that the graph is a DAG object.

MoralizeGraph

Creates class object by moralizing.

```
static CGraph* CGraph::MoralizeGraph( const CGraph *pGraph );
```

Arguments

pGraph	Pointer to a CGraph object to be moralized.

Discussion

Call of this function creates a new object of class CGraph by moralizing the input object and returns a pointer to it. Call of the class destructor deletes the instance.

AddEdge

Adds edge to existing graph.

```
void CGraph::AddEdge( int startNode, int endNode, int directed );
```

Arguments

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

ChangeEdgeDirection

Changes direction of existing graph.

```
void CGraph::ChangeEdgeDirection( int startNode, int endNode );
```

Arguments

startNode	Starting node of the edge for which direction should be changed.
endNode	Ending node of the edge for which direction should be changed.

GetNeighbors

Gets all neighbors for given node with orientation vector.

```
int CGraph::GetNeighbors( int nodeNum, int *numOfNbrs, const int **nbrs, const
    ENeighborType **nbrsTypes ) const;
void CGraph::GetNeighbors( int nodeNum, intVector* nbrsOut,
    neighborTypeVector *nbrsTypesOut ) const;
```

Arguments

nodeNum	Number of the node for which neighbors should be found.
numOfNbrs	Returned value, pointer to the variable that takes the value equal to the number of neighbors for the node.
nbrs	Returned value, pointer to the array of the neighbors of the node.
nbrsOut	Returned value. Array of the neighbors of the node.
nbrsTypes	Returned value, each element of which shows the type of the corresponding neighbor from *nbrs*.

---

nbrsTypesOut	Returned value, each element of which shows the type of the corresponding nieghbor from *nbrsOut.*

GetNumberOfNeighbors

Returns number of neighbors for given node.

```
inline int CGraph::GetNumberOfNeighbors( int nodeNum ) const;
```

Arguments

nodeNum	Number of the node for which the number of the neighbors should be found.

GetNumberOfNodes

Returns number of all nodes in graph.

```
inline int CGraph::GetNumberOfNodes() const;
```

GetNumberOfEdges

Returns number of all edges in graph.

```
inline int CGraph::GetNumberOfEdges() const;
```

intel.

IsCompleteSubgraph

Checks subset of given nodes for completeness.

```
int CGraph::IsCompleteSubgraph( int numOfNdsInSubgraph, const int *subgraph )
    const;
int CGraph::IsCompleteSubgraph( const intVector& subGraphIn ) const;
```

Arguments

numOfNdsInSubgraph	Number of nodes in the subset.
subgraph	Subset of nodes.

Discussion

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

IsChangeAllowed

Returns status flag for graph.

```
inline int CGraph::IsChangeAllowed() const;
```

Discussion

This function returns 1 if the change of the graph is allowed and 0 otherwise.

IsExistingEdge

Returns information on edge existence.

```
int CGraph::IsExistingEdge( int startNode, int endNode ) const;
```

Arguments

startNode   Starting node of the edge.

endNode   Ending node of the edge.

Discussion

This function returns 1 for the edge if it exists in the graph and 0 otherwise.

RemoveEdge

Removes edge from graph.

```
int CGraph::RemoveEdge( int startNode, int endNode );
```

Arguments

startNode   Starting node of the edge.

endNode   Ending node of the edge.

SetNeighbors

Sets neighbors for given node.

```
void CGraph::SetNeighbors( int nodeNum, int numOfNbrs,  const int *nbrs, const
    ENeighborType *nbrsTypes );
```

```
void CGraph::SetNeighbors( int nodeNum, const intVector& nbrs,const
   neighborTypeVector& nbrsTypes );
```

Arguments

nodeNum	Number of the node for which neighbors should be set.
numOfNbrs	Number of the neighbors for the node.
nbrs	Array of the neighbors for the node.
nbrsTypes	1D array of types of the neighbors of the node numbered *nodeNum*.

ProhibitChange

Prohibits any change of CGraph *object.*

```
inline void CGraph::ProhibitChange() const;
```

FormCliqueFromSubgraph

Forms a clique by connecting all the nodes of the subgraph between each other.

```
void CGraph::FormCliqueFromSubgraph( int numOfNds, const int *subGraph );
void CGraph::FormCliqueFromSubgraph( const intVector& subGraph );
```

Arguments

numOfNds	Number of nodes in subgraph of nodes.
subGraph	Subgraph, all the nodes of which should be connected to each other to form a clique.

GetNumberOfParents

Returns number of parents of the node.

```
inline int CGraph::GetNumberOfParents( int nodeNum ) const;
```

Arguments

nodeNum   Number of the node, for which number of parents is queried.

GetNumberOfChildren

Returns number of children of the node.

```
inline int CGraph::GetNumberOfChildren( int nodeNum ) const;
```

Arguments

nodeNum   Number of the node, for which number of children is queried.

IsDirected

Checks if the graph is directed.

```
inline int CGraph::IsDirected() const;
```

Discussion

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

IsUndirected

Checks if the graph is undirected.

```
inline int CGraph::IsUndirected() const;
```

Discussion

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

GetAdjacencyMatrix

Returns adjacency matrix.

```
inline void CGraph::GetAdjacencyMatrix( CMatrix<int>** pAdjMatOut ) const;
```

Arguments

pAdjMatOut   Returned parameter. Pointer to the adjacency martix for the graph.

Discussion

This function returns an adjacency matrix, which corresponds to the graph described by the related `CGraph` object. Note, that the adjacency matrix is not stored inside the graph and should be formed only when the user calls this member function.

ClearGraph

Clears graph.

```
inline void CGraph::ClearGraph();
```

Discussion

This function clears the graph by deleting lists of neighbors for all nodes and then setting the number of nodes equal to zero.

operator ==

Checks if two graphs are identical.

```
inline bool CGraph::operator ==( const CGraph& rGraph ) const;
```

Arguments

rGraph   Reference to the graph, which is the right-hand-side operand of the comparison operation.

Discussion

This function checks if two graphs are identical, and returns 'true', if they are, and 'false' otherwise.

operator ! =

Checks if two graphs are not identical.

```
inline bool CGraph::operator !=(const CGraph& rGraph) const;
```

Arguments

rGraph   Reference to the graph, which is the right-hand-side operand of the comparison operation.

Discussion

This function checks if two graphs are not identical, and returns 'true', if they are not, and 'false' otherwise.

> **NOTE.** *For the two above functions* `operator ==` *and* `operator !=` *the graphs are identical, if they have the same number of nodes and nodes have the same number of neighbors, which are numbered the same and are of the same type.*

GetParents

Returns vector of parents for node.

```
inline void CGraph::GetParents( int nodeNum, intVector *parents ) const;
```

Arguments

nodeNum   Number of the node, for which parents are inquired.

parents   Actually returned parameter. A pointer to a 1D vector of integers that are numbers of nodes, which appear to be parents of the node numbered *nodeNum*.

GetChildren

Returns vector of children for node.

```
inline void CGraph::GetChildren( int nodeNum, intVector *children ) const;
```

Arguments

nodeNum	Number of the node, for which children are inquired.
children	Actually returned parameter. A pointer to a 1D vector of integers that are numbers of nodes, which appear to be children of the node numbered *nodeNum*.

IsDAG

Checks if graph is directed acyclic graph.

```
int CGraph::IsDAG() const;
```

Discussion

This member function returns 1, if the graph is a directed acyclic graph (DAG), and 0 otherwise.

IsTopologicallySorted

Checks if graph is topologically sorted.

```
int CGraph::IsTopologicallySorted() const;
```

Discussion

This member function returns 1, if the graph is topologically sorted. Number of a parent is always less than numbers of its children.

NumberOfConnectivityComponents

Returns number of graph connectivity components.

```
int CGraph::NumberOfConnectivityComponents() const;
```

Discussion

This function returns a number of connectivity components of the graph it has been called for. Note that if graph has more than one connectivity components, the inference engine throws an exception. This means that in the case of several connectivity components all of them should be treated as separate graphical models.

GetConnectivityComponents

Returns connectivity components.

```
void CGraph::GetConnectivityComponents( intVecVector *decompositionOut )
    const;
```

Arguments

decompositionOut   Returned parameter. Array of connectivity components.

Discussion

This function returns connectivity components.

operator =

Assigns new value to graph object.

```
CGraph& CGraph::operator =( const CGraph& rGraph );
```

Arguments

rGraph   Reference to a graph, which is the right-hand-side operand
                            of the assignment operation.

Discussion

This function assigns one existing graph to the other, so that the result is an identical
copy of the input graph.

Dump

Dumps graph.

```
void CGraph::Dump() const;
```

Discussion

This function dumps the graph, that is, all the neighbors and neighbors types for all
nodes, to the standard output.

GetAncestry

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

```
void CGraph::GetAncestry( intVector const &subGraph, intVector *closure )
    const;
```

Arguments

subGraph   Vector of indices of the input subgraph.

closure   Output vector.

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.*

```
void CGraph::GetAncestralClosure( intVector const &subGraph,
 intVector *closure ) const;
```

Arguments

subGraph   Vector of indices of the input subgraph.

closure   Output vector.

Discussion

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

GetAncestralClosure

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

```
void Cgraph::GetAncestralClosure( intVector const &subGraph,
boolVector *closureMask ) const;
```

Arguments

subGraph   Vector of indices of the input subgraph.

closureMask   Output boolean 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.

GetSubgraphConnectivityComponents

*Finds plain connectivity components of induced
subgraph.*

```
void CGraph::GetSubgraphConnectivityComponents( intVector const &subGraph,
    intVecVector *decomposition ) const;
```

Arguments

subGraph   Vector of indices of the input subgraph.

decomposition   Output vector of vectors with indices of the 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.

```
void CGraph::GetDConnectionList( int node, intVector const &separator,
    intVector *dseparationList ) const;
```

Arguments

node	Given node.
separator	Separator for `d`-separation.
dseparationList	Output vector of indices of nodes.

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.

```
void CGraph::GetDConnectionTable( intVector const &separator, intVecVector
    *dseparationTable ) const;
```

Arguments

separator   Separator for *d*-separation.

dseparationTable   Output vector of vectors of indices of nodes.

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.

```
void CGraph::GetReachableSubgraph( intVector const &subgraph, bool *ban[],
    intVector *closure ) const;
void CGraph::GetReachableSubgraph( int node, bool *ban[], intVector *closure )
    const;
```

Arguments

subgraph   Given subgraph.

ban   Three-dimensional boolean mask.

closure   Output vector of indices of nodes.

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 a 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.*

Class CDAG

PNLBase

Graph
Create()
Copy()
AddEdge()
RemoveEdge()
AddNodes()

DAG
Create()
Clone()
CreateAncestorMatrix()
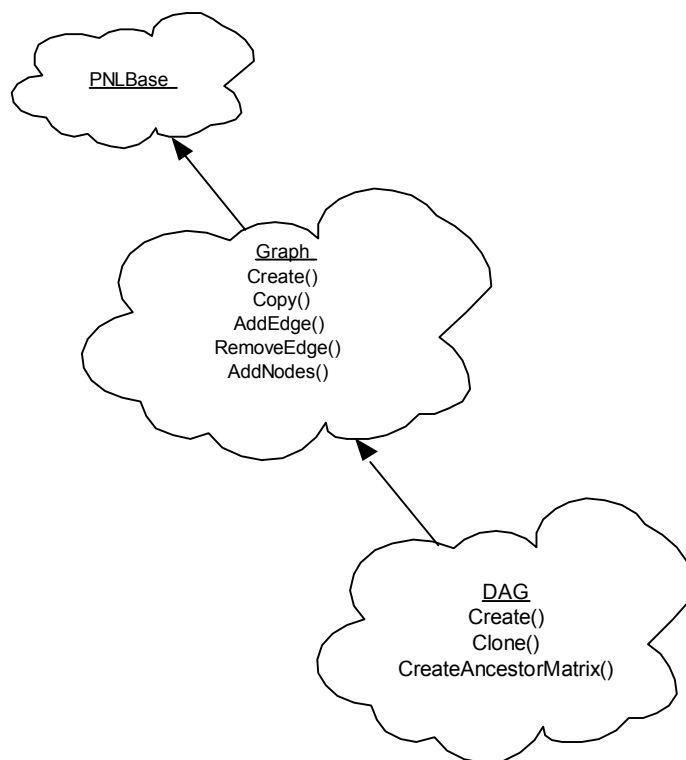
Class `CDAG` represents the structure of a `DAG`, with its ancestor matrix being a variable of the class.

Public Member Functions

Create

Creates class object.

```
static CDAG* CDAG::Create( int numOfNds, const int *numOfNbrsIn,
    const int *const *nbrsListIn,  const ENeighborType *const *nbrsTypesIn );
static CDAG* CDAG::Create( int numOfNds, const int *const *adjMatIn );
static CDAG* CDAG::Create(const CMatrix<int>* pAdjMat);
static CDAG* CDAG::Create(const CGraph& pGraph);
```

Arguments

numOfNds	Number of nodes of the DAG.
numOfNbrsIn	Array of integer, each ith integer is the number of neighbors of the ith nodes.
nbrsListIn	List of neighbors for a node.
nbrsTypesIn	2D list of neighbor types. Each element shows the type of the corresponding neighbor from *nbrsListIn*.
adjMatIn	2D array of integers, represents an adjacent matrix.
pAdjMat	2D integer matrix, represents an adjacent matrix.
pGraph	Reference to a CGraph instance.

Discussion

This function creates a CDAG object.

Change

Changes DAG *object.*

```
CDAG* CDAG::Change(int iStartNode, int iEndNode, EDAGChangeType iChangeType);
```

Arguments

iStartNode	ID of the node from which the directed edge comes.
iEndNode	ID of the node at which the directed edge points.
iChangeType	One of three operations: addition, deletion or reversing of the direction of the edge.

Discussion

This function allows to add, to delete or to reverse an edge of a DAG object. The function returns *NULL* if the change did not occur, otherwise returns the new CDAG object.

ClearContent

Deletes ancestor matrix of DAG.

```
void CDAG::ClearContent();
```

Discussion

This function deletes the ancestor matrix of the DAG object.

Clone

Creates replica of DAG.

```
CDAG* CDAG::Clone();
```

Discussion

This function copies the DAG object and thus creates its replica.

CreateAncestorMatrix

Creates ancestor matrix for DAG

```
void CDAG::CreateAncestorMatrix();
```

Discussion

This function creates the ancestor matrix for a DAG object.

CreateMinimalSpanningTree

Creates minimal spanning tree for given weight matrices.

```
CDAG* CDAG::CreateMinimalSpanningTree(const CMatrix<float>* pWeightMatrix,
    const CMatrix<float>* pWeightMatrix2);
```

Arguments

pWeightMatrix	N*N 2D weight matrix, the primary weight. Element *[i,j]* connects node *i* to node *j*.
pWeightMatrix2	N*N 2D weight matrix, the secondary weight.

Discussion

This function creates a `CDAG` instance for the minimal spanning tree.

DoMove

Changes `DAG` *object.*

```
bool CDAG::DoMove( int iStartNode, int iEndNode, EDAGChangeType iChangeType );
```

Arguments

iStartNode   ID of the node from which the directed edge comes.

iEndNode   ID of the node at which the directed edge points.

iChangeType   One of three operations: addition, deletion or reversing of the direction of the edge.

Discussion

This function changes the DAG object by addition, deletion or reversing of an object edge. This function returns 'true' when the change of the `DAG` object was permitted and the object was changed, returns 'false' otherwise.

GetAllEdges

Gets all edges of `DAG` *object.*

```
void CDAG::GetAllEdges( EDGEVECTOR* pvOutput );
```

Arguments

pvOutput   `EDGE` vector which is used to store all the edges of the `DAG`.

Discussion

This function gets all the edges of the DAG. *EDGE* is a data structure which contains two integers: node ID from which the edge comes and node ID at which the edge points.

GetAllNeighbours

Generates all valid neighbour DAGs for object.

```
void CDAG::GetAllNeighbours( POINTVECTOR &vNeighbours, EDGEOPVECTOR &vOPs,
 bool (*IsValid)(CDAG* pDAG) );
```

Arguments

vNeighbors	Vector of void* which stores all valid neighbors of the DAG object.
vOPs	*EDGEOP* vector which stores all valid changes to the DAG object.
IsValid(CDAG pDAG)*	External function called to check if a DAG is valid.

Discussion

This function generates all neighbor DAGs of the given class object that satisfy the conditions of the `IsValid` function.

EDGEOP is a data structure which contains three fields: the start node ID, the end node ID and an edge change type: add, delete or reverse.

If *vNeighbours* and *vOPs* have the corresponding relationship, that is if the `DoMove`*(vOPs[i])* is carried out , we can get *vNeighbours[i]*.

This function generates all neighbor DAGs of the given DAG and later eliminates all the neighbors that do not satisfy the condition of the `IsValid()` function, that is all the neighbors for which the `IsValid()` function returns 'false'. If the `IsValid()` is not applied the function generates all legal neighbor DAGs of the object.

intel

GetAllValidMove

Generates all valid moves for DAG.

```
void CDAG::GetAllValidMove(EDGEOPVECTOR *pvOutput, intVector*
    pvAncesstorVector, intVector* pvDescendantsVector, intVector*
    pvNotParents, intVector* pvNotChild) );
```

Arguments

pvOutput	EDGEOP vector. Stores all valid changes to the DAG.
pvAncesstorVector	Integer vector.
pvDescendantsVector	Integer vector.
pvNotParent	Integer vector.
pvNotChild	Integer vector.

Discussion

This function generates all valid moves for the DAG. This function is carried out under the condition that the *pvDescendantsVector* nodes of a generated DAG are not ancestors of *pvAncesstorVector* nodes and that *pvNotParent* nodes are not ancestors of *pvNotChild* nodes.

GetEdgeDirection

Gets direction of edge.

```
int CDAG::GetEdgeDirect(int startNode, int endNode);
```

Arguments

startNode	Node ID from which the edge comes.
EndNode	Node ID at which the edge points.

Discussion

This function learns the direction of the edge. The function returns

1 if the edge is directed from *startNode* to *EndNode*

-1 if the edge is directed from *EndNode* to *startNode*

0 if there is no edge between the two nodes.

GetMaxFanIn

Computes MaxFanIn for DAG *and returns it to object.*

```
int CDAG::GetMaxFanIn();
```

GetSubDAG

Gets part of DAG

```
CDAG* CDAG::GetSubDAG( intVector &vSubNodesSet );
```

Arguments

vSubNodesSet   Integer vector which stores the node IDs that are to be extracted from the DAG.

IsEquivalent

Compares two DAG*s.*

```
bool CDAG::IsEquivalent( CDAG *pDAG );
```

Arguments

pDAG1   DAG instance compared to the given DAG.

Discussion

This function compares two objects of the class. The function returns 'true' if the DAGs are identical, returns 'false' otherwise.

IsValidMove

*Checks if move generates non-*DAG *object.*

```
bool CDAG::IsValidMove(int iStartNode, int iEndNode, EDAGChangeType
    iChangeType);
```

Arguments

iStartNode   Start node ID of the move.

iEndNode   End node ID of the move.

iChangType   One of the following operations: *add, delete, reverse.*

Discussion

This function checks if the move can generate a non-DAG object. The function returns 'true' if the move generates a DAG, returns 'false' if the move generates a non-DAG.

MarkovBlanket

Computes Markov Blanket for node.

```
void CDAG::MarkovBlanket(int nNodeNumber, intVector *pvOutPut);
```

Arguments

nNodeNumber	Node ID for which Markov Blanket is to be computed.
pvOutPut	Pointer to the integer vector which stores the computed Markov Blanket.

Discussion

This function computes the Markov Blanket for a node.

RandomCreateADAG

Creates random DAG

```
CDAG* CDAG::RandomCreateADAG(int iNodeNumber, const intVector &vAncesstor,
    const intVector &vDescendants, intVector* pvNotParent, intVector*
    pvNotChild);
```

Arguments

iNodeNumber	Number of nodes in generated DAG.
vAncesstor, *vDescendants*	Integer vectors.Make sure that among *vDescendants* nodes of the generated DAG there are no ancestors of *vAncestor* nodes.
pvNotParent, *pvNotChild*	Integer vectors. Make sure that among *pvNotParent* nodes of the generated DAG there are no ancestors of *pvNotChild* nodes.

SetSubDAG

Replaces part of DAG *by input sub-*DAG

```
bool CDAG::SetSubDag(intVector &vSubNodeSet, CDAG *pSubDAG);
```

Arguments

vSubNodeSet   Integer vector. Stores node IDs that are to be modified.

pSubDAG   `DAG` that is to replace a part of the given `DAG`.

Discussion

This function replaces a part of the given `DAG` by a sub-`DAG`. The function returns 'true' if the replacement has been successful, otherwise returns 'false'.

SymmetricDifference

Compares structures of two `DAG`*s.*

```
int CDAG::SymmetricDifference(const CDAG* pDAG) const;
```

Arguments

pDAG   Pointer to a `DAG` to be compared with the given `DAG`.

Discussion

This function compares structures of two class objects.

TopologicalCreateDAG

Creates replica `DAG`.

```
CDAG* CDAG::TopologicalCreateDAG( intVector& vNodesMap );
```

Arguments

vNodesMap   Integer vector. Preserves the map of the node ID of the given `DAG` for new node IDs. As a result, the ith node of the newly created `DAG` corresponds to the node *vNodesMap[i]* of the given `DAG`.

Discussion

This function creates a replica of the given `DAG`. Node ID of the new `DAG` is provided by the input integer vector.

TopologicalSort

Sorts nodes of `DAG` topologically.

```
bool CDAG::TopologicalSort( intVector* pvOutput );
```
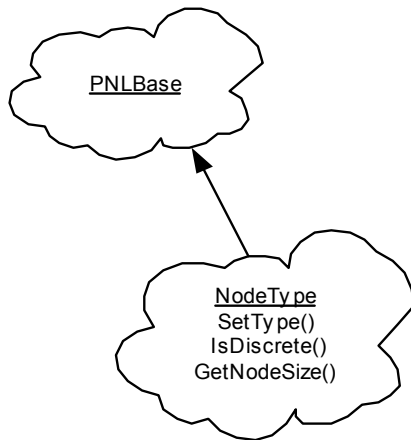
Arguments

pvOutput   Integer vector. Preserves the map of the node ID of the given `DAG` for new node IDs. As a result the ith node of the newly created `DAG` corresponds to the node *pvOutput[i]* of the given `DAG`.

Discussion

This function classifies a `DAG` object according to its topological order.

Node Types

Class CNodeType

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

Public Member Functions

IsDiscrete

Returns information on node discreteness.

```
inline int CNodeType::IsDiscrete() const;
```

Discussion

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

GetNodeSize

Returns node size.

```
inline int CNodeType::GetNodeSize() const;
```

SetType

Sets node type.|

```
inline void CNodeType::SetType( bool isDiscrete, int ndSize );
```

Arguments

isDiscrete   Type of node value. Equals to `true` if the node is discrete, equals to `false` if the node is continuous.

ndSize   New node size.

Discussion

This function sets the type of the given node.

operator==

Compares operands.

```
inline bool operator==( const CNodeType &ntIn ) const;
```

Arguments

ntIn   `CNodeType` object.

intel.

Discussion

This function compares two operands. Returns 'true' if the operands are equal, returns 'false' otherwise.

operator!=

Compares two operands.

```
inline bool operator!=( const CNodeType &ntIn ) const;
```

Arguments

*ntIn*   CNodeType object.

Discussion

This function compares two operands. The function returns 'true' if the operands are not equal; returns 'false' otherwise.

Model Domain

Model domain is a set of nodes that define a graphical model. Multiple graphical models can have one model domain. This object keeps all information about 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
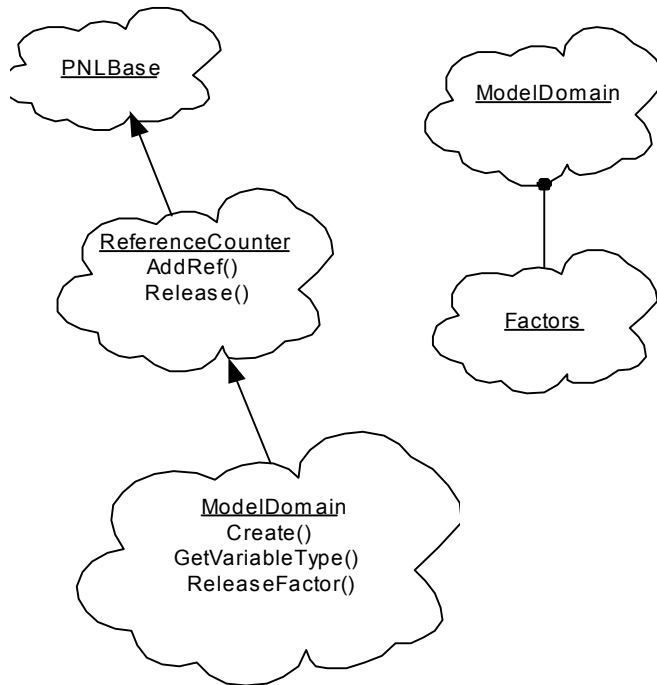
This class contains information on the variable types of all nodes to be used for creating a graphical model, as well as on the node types for nodes to become observed on entering evidence during inference. A graphical model keeps the pointer to the CModelDomain on which it was created and learns from it the information on node types.

This class also stores temporary CFactor objects that are not attached to any graphical model. Such objects appear during inference and learning procedures. When the CModelDomain is destroyed they are deleated form it automatically.

This class is derived from the CReferenceCounter class. It keeps pointers to CGraphicalModels that are based on the CModelDomain. The class cannot be deleted until any Graphical Model referres to it.

Create

Creates class object.

```
static CModelDomain* CModelDomain::Create( const nodeTypeVector&
    variableTypes, const intVector& variableAssociation, CGraphicalModel*
    pCreatorOfMD = NULL );
static CModelDomain* CModelDomain::Create(int numVariables, const CNodeType&
    commonVariableType = CNodeType(1,2), CGraphicalModel* pCreatorOfMD =
    NULL);
```

Arguments

variableTypes	Vector of different node types.
variableAssociation	Vector of variable association with the variable types.
pCreatorOfMD	Pointer to the graphical model which creates a model domain.
numVariables	Number of variables in the model domain.
CommonVariableType	Variable type.

Discussion

Two function versions are available.The first creates a model domain with different variable types, the association for every node pointing at a node type.The second creates a model domain with all variables of the same type.

AttachFactor

Attaches factor to model domain.

```
int CModelDomain::AttachFactor( const CFactor *pFactor );
```

Arguments

pFactor   Pointer to the factor to be attached.

Discussion

Returns the number of this factor in array of pointers to the factors on the model domain.

ReleaseFactor

Releases attached factors from model domain.

```
void CModelDomain::ReleaseFactor( const CFactor *pFactor );
```

Arguments

pFactor   Pointer to the factor which is to be released from the model domain.

IsAFactorOwner

Checks if model domain keeps pointer to query factor.

```
bool CModelDomain::IsAFactorOwner( const CFactor *pFactor );
```

Argument

pFactor   Pointer to the factor which is to be released from the model domain.

Discussion

The function returns the value of 1 if the model domain keeps a pointer to the query factor, that is, if the model domain is the owner of the query factor. Otherwise, returns 0.

intel

GetVariableType

Returns pointer to node type of query variable.

```
const CNodeType* CModelDomain::GetVariableType( int varNumber ) const;
```

Arguments

Frumpier   Number of variables.

Get Variable Types

Returns variable types for query variables.

```
void CModelDomain::GetVariableTypes( intVector& vars, pConstNodeTypeVector*
    varTypes ) const;
```

Arguments

vars   Vector of number of variables.

*varTypes*Returned parameter. Types of variables.

Discussion

This function fills in the input vector *varTypes* by const pointers to variable types.

GetObsGauVarType

*Returns pointer to observed Gaussian variable
type.*

```
inline const CNodeType* CModelDomain::GetObsGauVarType() const;
```

intel®

3-40

Discussion

This function returns a pointer to the observed Gaussian variable type.

GetObsTabVarType

Returns pointer to observed Tabular variable
type.

```
inline const CNodeType* CModelDomain::GetObsTabVarType() const;
```

Discussion

This function returns the pointer to the observed Tabular variable type.

GetNumberOfVariableTypes

Returns number of different variable types.

```
inline int CModelDomain::GetNumberOfVariableTypes() const;
```

Discussion

This function returns a number of different variable types.

GetVariableTypes

Returns all variable types.

```
void CModelDomain::GetVariableTypes( pConstNodeTypeVector* varTypes )const;
void CModelDomain::GetVariableTypes( nodeTypeVector* varTypes )const;
```

Arguments

varTypes   Returned parameter. Vector of node types.

Discussion

The function is available in two versions.

The first returns the vector of pointers to const node types.

The second creates objects and places them into the returnable vector.

GetNumberVariables

Returns number of variables of model domain.

```
inline int CModelDomain::GetNumberVariables() const;
```

Discussion

This function returns number of variables of the model domain.

GetVariableAssociations

Returns association to variable types.

```
void CModelDomain::GetVariableAssociations( intVector* variableAssociation )
    const;
inline const int* CModelDomain::GetVariableAssociations()const;
```

Arguments

variableAssociation   Returned parameter. Vector of associations of variables to
                          variable types.

Discussion

The function is available in two versions. The first adopts and fills the vector, the second returns the pointer to the association.

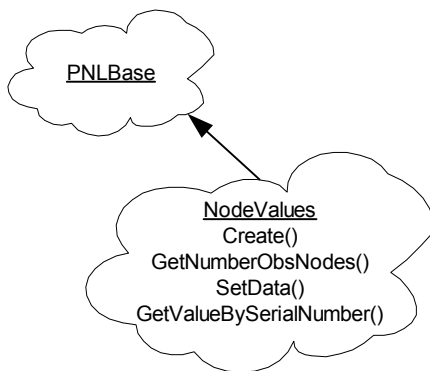# GetVariableAssociation

Returns variable association.

```
inline int CModelDomain::GetVariableAssociation(int variable)const;
```

Arguments

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, while values of continuous nodes – 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 *potentially* observed, it can be observed and its observed value is at `CNodeValues` object, but the node is not actually observed. To make it *actually* observed, the corresponding observability flag needs to be changed to *true* value.

Creation of evidences for the model is simplified in this way. The user can create evidence for all nodes, set observability flag for some of them to *false*, and start inference with such evidence. To start inference with other observed nodes, the user can just toggle node observability states for some of them.

Class `CNodeValues` is basic for Model Domain and contains information about observed values of certain variables without indicating the numbers of these variables in the graphical model. Model Domain contains information on correspondence between the variables and the graphical model nodes.

Public destructor `~CNodeValues()` deletes an object of the class.

Create

Creates class object.

```
static CNodeValues* CNodeValues::Create( int nObsNds, const CNodeType* const*
    obsNdsTypes, const valueVector& obsValues );
static CNodeValues*  CNodeValues::Create( const pConstNodeTypeVector&
    obsNdsTypes, const valueVector& obsValues );
```

Arguments

nObsNds	Number of the observed nodes.
obsNdsTypes	Node types of observed nodes.
obsValues	Values of the observed nodes.

Discussion

This function returns objects of class `CNodeValues`. All nodes of any created object are actually observed.

GetValueBySerialNumber

Returns pointer to value.

```
inline Value const* CNodeValues::GetValueBySerialNumber( int SerialNumber )
    const;
inline Value* CNodeValues::GetValueBySerialNumber( int SerialNumber );
```

Arguments

SerialNumber   Serial number of the observed variable.

Discussion

Both the functions return pointer to the value of the observed node. The `const` function solely provides observation of the observed node value. The `non-const` function enables user to change the value of the observed node in accordance with the returned pointer.

GetNumberObsNodes

Returns total number of observed nodes.

```
int CNodeValues::GetNumberObsNodes() const;
```

Discussion

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

GetObsNodesFlags

Returns pointer to array of observability flags.

```
const int * CNodeValues::GetObsNodesFlags() const;
```

Discussion

This function returns constant pointer to the array of observability flags for nodes of `CNodeValues` class objects.

GetRawData

Returns array of values.

```
void CNodeValues::GetRawData( valueVector* values ) const;
```

Arguments

values   Array of values.

Discussion

This function returns array of values.

GetOffset

Returns offsets in array of observed nodes.

```
const int * CNodeValues::GetOffset() const;
```

Discussion

This function returns the pointer to the array of offsets in the array of the values of observed variables. When the value of ith node needs to be found, the array of raw data should be addressed with the offset corresponding to the ith value in the array of the offsets.

SetData

Replaces old values with new values.

```
void CNodeValues::SetData( const valueVector& data );
```

Arguments

data   Array of new values.

GetNodeTypes

Returns array of pointers to node types.

```
const CNodeType *const* CNodeValues::GetNodeTypes() const;
```

Discussion

This function returns pointer to the array of pointers to CNodeType class objects corresponding to the node types of observed variables.

MakeNodeHiddenBySerialNum

*Changes observation flag from actually observed
to hidden.*

```
inline void CNodeValues::MakeNodeHiddenBySerialNum( int serialNum );
```

Arguments

serialNum   Number of an observed node, the state of which is to be
                               changed.

MakeNodeObservedBySerialNum

*Changes observation flag from hidden to actually
observed.*

```
inline void CNodeValues::MakeNodeObservednBySerialNum( int serialNum );
```

Arguments

serialNum   Number of a hidden node, the state of which is to be
                               changed.

ToggleNodeStateBySerialNumber

Toggles observability type.

```
void CNodeValues::ToggleNodeStateBySerialNumber(int numOfNds, int *nodeNums );
void CNodeValues::ToggleNodeStateBySerialNumber( const intVector& nodeNums );
```
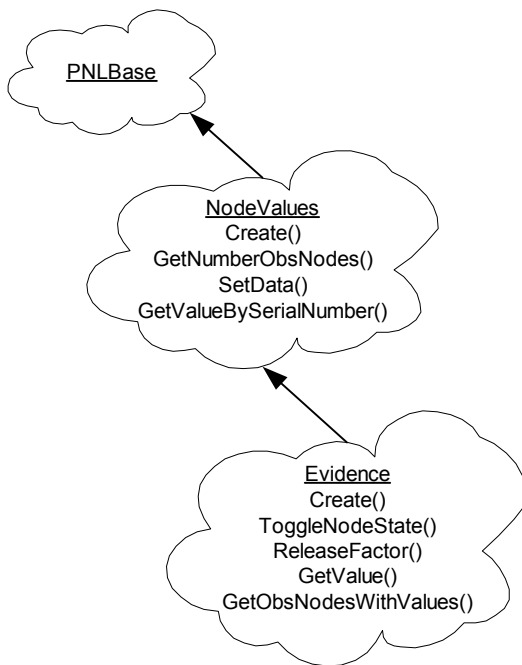
Arguments

numOfNds	Number of the nodes that require change of the state.
nodeNums	Serial numbers of the observed variables that require change of state.

Discussion

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

Class CEvidence

Class CEvidence contains information on observed variables of graphical model: what nodes are observed and what values are taken. Class CEvidence is based on Class CNodeValues, which contains information on types and actual values of certain variables. Class CEvidence proper stores the array of numbers of observed variables in

a graphical model, along with methods of getting information about an observed node or a set of observed nodes in a model. All the methods consist in establishing correspondence between numbers of the model nodes and their serial numbers within a `CNodeValues` object, as well as in calling the corresponding member functions of that class.

Public Member Functions

Create

Returns class object.

```
static CEvidence* CEvidence::Create( const CModelDomain* pMD, const intVector&
    obsNodes, const valueVector& obsValues );
static CEvidence* CEvidence::Create( const CNodeValues* values, const
    intVector& obsNodes, const CModelDomain* pMD, int takeIntoObservationFlags
    = 0 );
static CEvidence* CEvidence::Create( const CGraphicalModel* pGrModel, const
    intVector& obsNodes, const valueVector& obsValues);
static CEvidence* CEvidence::Create( const CModelDomain *pMD, int nObsNodes,
    const int *obsNodes, const valueVector& obsValues );
static CEvidence* CEvidence::Create( const CGraphicalModel *pGrModel,
    int nObsNodes, const int *obsNodes, const valueVector& obsValues );
static CEvidence* CEvidence::Create( const CNodeValues *values, int nObsNodes,
    const int *obsNodes, const CModelDomain* pMD, int takeIntoObservationFlags
    = 0 );
```

Arguments

pMD	Pointer to the model domain.
pGrModel	Pointer to a graphical model.
nObsNds	Number of the observed nodes or variables.
obsNodes	Array of observed nodes in the graphical model or observed variables it the model domain.

obsValues	Array of the observed values listed in the same order as in the array *obsNodes*.
takeIntoObservationFlags	Flag makes all nodes or variables actually observed if *takeIntoObservationFlag* = 0 or puts the same flags as in *pNodeValues* object if *takeIntoObservationFlag* = 1.

Discussion

This function creates a CEvidence object.

ToggleNodeState

Toggles observability type.

```
void CEvidence::ToggleNodeState( int numOfNds, int *nodeNums );
void CEvidence::ToggleNodeState( const intVector& nodeNums );
```

Arguments

numOfNds	Number of the nodes that require change of the state.
nodeNums	Numbers of the observed nodes that require change of the state.

Discussion

This function changes the state of the nodes from potentially observable to actually observable and vice versa.

GetValue

Returns pointer to value of observed node.

```
const Value* CEvidence::GetValue( int nodeNum ) const;
Value* CEvidence::GetValue( int nodeNum );
```

Arguments

nodeNum   Serial number of the observed node in the model.

Discussion

Both the functions return pointer to the value of the observed node by its serial number in the graphical model. The `const` function solely provides observation of the observed node value. The `non-const` function enables you to change the value of the observed node in accordance with the returned pointer.

GetAllObsNodes

Returns pointer to array of numbers of observed nodes.

```
const int * CEvidence::GetAllObsNodes() const;
```

Discussion

This function returns pointer to the array of numbers of observed nodes. To get the number of observed nodes, use the function GetNumberObsNodes of basic `CEvidence` class.

IsNodeObserved

Returns current observability status of the node.

```
int CEvidence::IsNodeObserved( int nodeNum ) const;
```

Arguments

nodeNum   Number of the node in the graphical model.

Discussion

This function returns 1, if node is actually observed, and 0 otherwise.

MakeNodeObserved

Changes observation flag for node to actually
observed.

```
void CEvidence::MakeNodeObserved( int nodeNum );
```

Arguments

nodeNum   Number of the node in the graphical model.

Discussion

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

MakeNodeHidden

Changes observation flag for node to hidden.

```
void CEvidence::MakeNodeHidden( int nodeNum );
```

Arguments

nodeNum   Number of the node in the graphical model.

Discussion

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

GetObsNodesWithValues

Returns pointer to vector of actually observed
nodes and their values.

```
void CEvidence::GetObsNodesWithValues( intVector* pObsNds, pConstValueVector*
    pObsValues,pConstNodeTypeVector* pNodeTypes = NULL ) const;
```

Arguments

pObsNds   Pointer to the vector that contains numbers of actually observed nodes from the graphical model.

pObsValues   Pointer to the vector of pointers to raw data of actually observed values. The order of these values corresponds to the order at *pObsNds*.

pNodeTypes   Pointer to the vector that contains pointers to the node types of observed nodes. The order of these values corresponds to the order at *pObsNds*.

Discussion

The function takes pointers to vectors and fills these vectors with requested information.

CModelDomain

Returns model domain.

```
inline const CModelDomain* CEvidence::GetModelDomain() const;
```

Dump

Dumps evidence content.

```
void CEvidence::Dump() const;
```

Save

Saves evidences to file.

```
static bool CEvidence::Save(const char *fname,  pConstEvidenceVector& evVec);
```

Arguments

fname   File name.

evVec   Array of evidences.

Discussion

This function saves evidences created for the static graphical model into a file. This function returns 'true' if the evidence is saved, returns 'false' otherwise.

Save

Saves evidences to file.

```
static bool CEvidence::Save(const char *fname,  pConstEvidenceVecVector&
    evVec);
```

Arguments

fname   File name.

evVec   Array of evidences.

Discussion

This function saves evidences created for the dynamic graphical model into a file. This function returns 'true' if the evidence is saved, returns 'false' otherwise.

Load

Loads evidences from file.

```
static bool CEvidence::Load( const char *fname,  pEvidencesVector* evVec,
    const CModelDomain *pMD );
```

Arguments

fname   File name.

evVec   Empty vector which is to store evidences.

pMD   Pointer to the model domain.

Discussion

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

Load

Loads evidences to file.

```
static bool CEvidence::Load(const char *fname,  pEvidencesVecVector* evVec,
const CModelDomain *pMD);
```

Arguments

fname	File name.
evVec	Array of evidences.
pMD	Pointer to the model domain.

Discussion

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

Graphical Models

Class CGraphicalModel
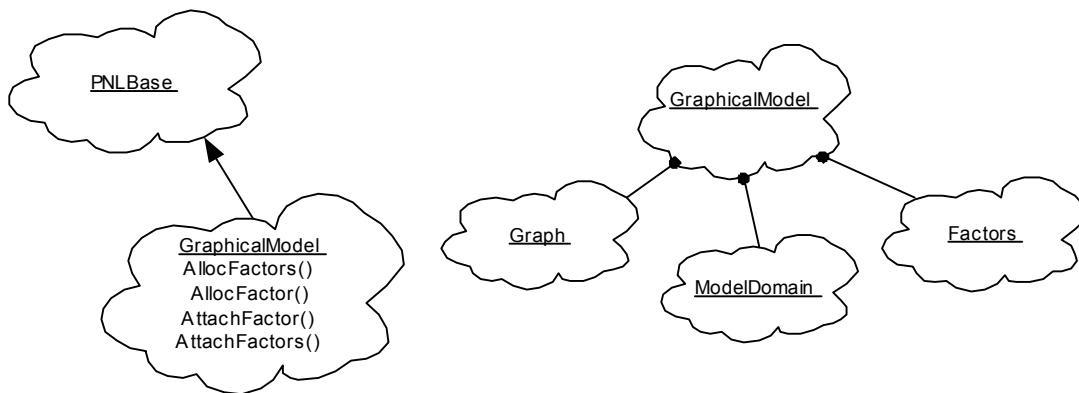
Class `CGraphicalModel` represents a graphical model, which consists of the graph and of factors that are set for the graph nodes. Class `CGraphicalModel` is a parent to two classes: Class CStaticGraphicalModel and Class CDynamicGraphicalModel.

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

Public Member Functions

AllocFactor

Allocates factor for domain.

```
int CGraphicalModel::AllocFactor( int number );
int CGraphicalModel::AllocFactor( int numOfNdsInDomain, int *domain );
int CGraphicalModel::AllocFactor( const intVector& domain );
```

Arguments

number	Index of the domain for which the factor should be allocated.
numOfNdsInDomain	Number of nodes in the domain for which the factor should be allocated.
domain	Domain for which the factor is allocated.

Discussion

This function allocates a factor for a domain. The domain is specified either by its index or by the nodes from this domain. `AllocFactor` is a virtual function implemented separately for Class CMNet, Class CBNet, and Class CMRF2.

AllocFactors

Allocates space for all factors of model.

```
int CGraphicalModel::AllocFactors();
```

Discussion

This function allocates space for all the factors of the model, that is, creates an object of the `Class CFactors` and attaches it to the model. `AllocFactors` is a virtual function implemented separately for `Class CMNet`, `Class CBNet`, and `Class CMRF2`.

AttachFactor

Attaches factor to model.

```
int CGraphicalModel::AttachFactor( CFactor *pFactor );
```

Arguments

pFactor   Pointer to the 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. `AttachFactor` is a virtual function implemented separately for `Class CMNet`, `Class CBNet`, and `Class CMRF2`.

AttachFactors

Attaches set of new factors and returns set of old
factors for destruction.

```
CFactors* CGraphicalModel::AttachFactors( CFactors *pFactors);
```

Arguments

pFactors   New factors to be attached to the model instead of the old
factors.

Discussion

This function attaches a set of factors stored in the `Class CFactors` object and
returns the set of old factors for the user to destroy them.

GetGraph

Returns non-constant pointer to class object.

```
inline CGraph* CGraphicalModel::GetGraph() const;
```

Discussion

This function returns a non-constant pointer to the `CGraph` class object, which is
attached to the model.

GetModelType

Returns type of model.

```
inline CModelTypes CGraphicalModel::GetModelType() const;
```

GetNodeAssociations

*Returns constant pointer to array of node
associations for all nodes of model.*

```
inline const int* CGraphicalModel::GetNodeAssociations() const;
```

GetNodeType

*Returns constant pointer to class object for
specified node number.*

```
inline const CNodeType* CGraphicalModel::GetNodeType(int nodeNum) const;
```

Arguments

nodeNum   Number of the node for which the node type is inquired.

Discussion

This function returns a constant pointer to the   `Class CNodeType` object for the
specified node number.

GetNodeTypes

Provides access to all node types of model.

```
inline void CGraphicalModel::GetNodeTypes( nodeTypeVector* nodeTypes ) const;
inline void CGraphicalModel::GetNodeTypes( pConstNodeTypeVector* nodeTypes )
    const;
```

Arguments

nodeTypes   Returned parameter. Array of all `CNodeType` objects
                               attached to the model.

GetNumberOfNodes

Returns number of nodes for model.

```
inline int CGraphicalModel::GetNumberOfNodes() const;
```

Discussion

This function returns the whole number of nodes for the static graphical model and
returns the number of the nodes per slice for the dynamic graphical model.

GetNumberOfNodeTypes

Returns number of node types for model.

```
inline int CGraphicalModel::GetNumberOfNodeTypes() const;
```

GetNumberOfFactors

Returns number of factors attached to model.

```
inline int CGraphicalModel::GetNumberOfFactors() const;
```

GetFactor

Returns non-constant pointer to class object for
specified domain number.

```
CFactor* CGraphicalModel::GetFactor( int domainNumber );
```

Arguments

domainNumber   Number of domain for which the factor needs to be found.

Discussion

This function returns a non-constant pointer to the   Class CFactor object for the specified domain number.

GetFactors

Returns all factors attached to specified subset of
nodes.

```
int CGraphicalModel::GetFactors( int nNodes, const int* nodes,  int *nFactors,
    CFactor ***factors );
virtual void CGraphicalModel::GetFactors( int nNodes, const int* nodes,int
    *nFactors, CFactor ***factors ) const = 0;
virtual int CGraphicalModel::GetFactors( int nNodes, const int* nodes,
    pFactorVector *factors) const = 0;
virtual int CGraphicalModel::GetFactors( const intVector& nodes, pFactorVector
    *factors ) const;
```

Arguments

nodes   Subset of nodes for which all the attached factors need to be found.

---

nNodes	Number of nodes in a subset for which the attached factors need to be found.
nFactors	Returned parameter. Pointer to the variable that specifies the number of factors attached to the subset of nodes.
factors	Factors attached to the subset of nodes specified in the input.

Discussion

This function enables the user to receive all the factors attached to the specified subset of nodes. Several factors may be attached to the same subset if the subset is a common part of several domains.

GetModelDomain

Returns model domain.

```
inline CModelDomain* CGraphicalModel::GetModelDomain() const;
```

IsValid

Checks validity of graphical model.

```
virtual bool CGraphicalModel::IsValid( std::string* descriptionOut = NULL )
    const = 0;
```

Arguments

descriptionOut   Error message.

Discussion

This function checks the validity of the function. The function returns 'true' if the model is valid, returns 'false' otherwise.

Class CStaticGraphicalModel
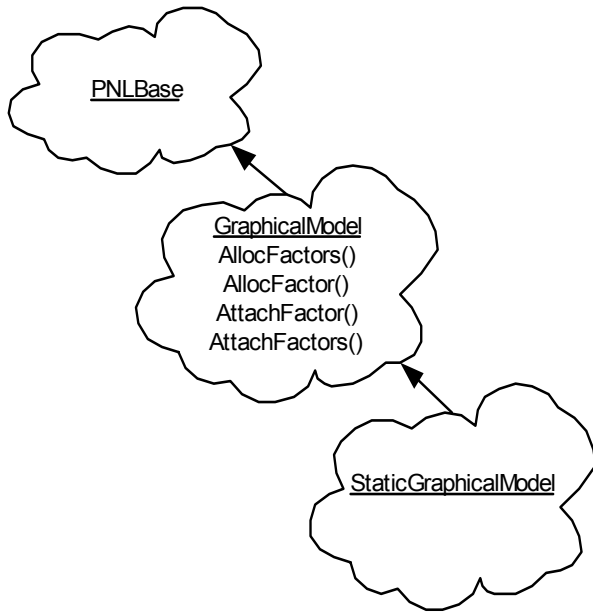
Class `CStaticGraphicalModel` is a parent to two subclasses: `Class CBNet` and `Class CMNet`.

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

IsValid

Checks validity of model for creation of dynamic model.

```
bool CStaticGraphicalModel::IsValidAsBaseForDynamicModel(std::string*
    descriptionOut = NULL) const;
```
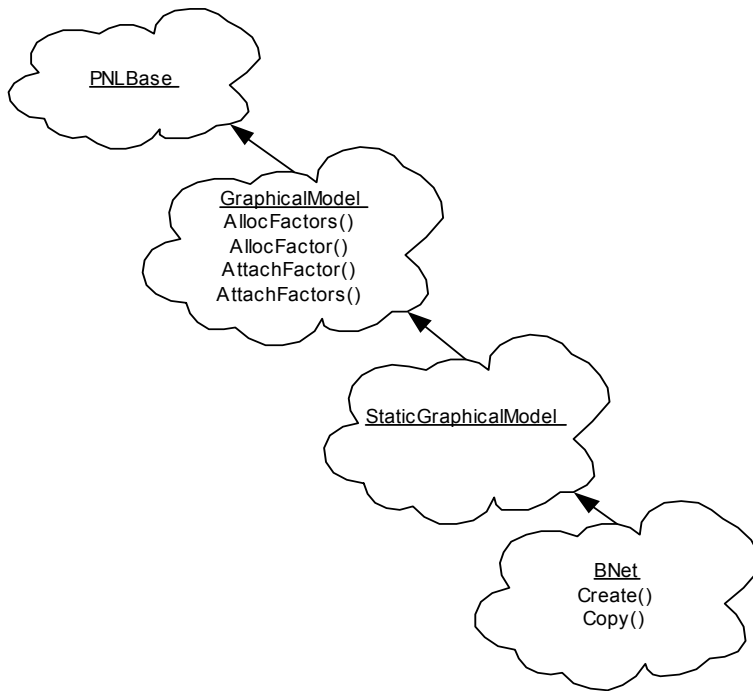
Arguments

descriptionOut   Error message.

Discussion

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

Class CBNet

Public Member Functions

Create

Creates class object.

```
static CBNet* CBNet::Create( int numberOfNodes, int numberOfNodeTypes,const
    CNodeType *nodeTypes,const int *nodesAssociation, CGraph *pGraph );
```

```
static CBNet* CBNet::Create( int numberOfNodes,const nodeTypeVector&
    nodeTypes,                const intVector& nodesAssociation, CGraph *pGraph
    );
static CBNet* CBNet::Create( CGraph *pGraph, CModelDomain* pMD );
```

Arguments

numberOfNodeTypes	Number of all node types.
nodeTypes	Array of node types.
nodesAssociation	Array for nodes association with node types.
pGraph	Graph structure of the model.
numberOfNodes	Number of nodes.
pMD	Model domain.

Discussion

This function creates a class object.

Copy

Creates new object by copying.

```
static CBNet* CBNet::Copy(const CBNet* pBNet);
```

Arguments

pBNet	Pointer to a CBNet object to be copied.

Discussion

This function creates a new CBNet object by copying the input object and returns a pointer to it. Call of the class destructor deletes the instance.

CreateWithRandomMatrices

Creates BNet *object with random matrices.*

```
static CBNet* CBNet::CreateWithRandomMatrices( CGraph* pGraph, CModelDomain*
    pMD );
```

Arguments

pGraph	Graph structure.
pMD	Model domain.

Discussion

This function creates a *BNet* object with dense random matrices. Coverings matrices of a Gaussian distribution are matrix units.

ConvertToSparse

Converts object with dense matrices into object
with sparse matrices.

```
CBNet* CBNet::ConvertToSparse() const;
```

Discussion

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

ConvertToDense

*Converts object with sparse matrices into object
with dense matrices.*

```
CBNet* CBNet::ConvertToDense() const;
```

Discussion

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

CreateTabularCPD

Creates valid tabular CPD.

```
void CBNet::CreateTabularCPD( int childNodeNumber, const floatVector&
    matrixData );
```

Arguments

childNodeNumber	Factor number.
matrixData	Array of matrix data.

Discussion

This function creates a tabular CPD using the given data.

FindMixtureNodes

Finds numbers of mixture nodes.

```
void CBNet::FindMixtureNodes( intVector* mixureNds );
```

Arguments

mixureNds   Input parameter. Empty vector of mixture nodes.

Discussion

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

GenerateSamples

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

```
virtual void CBNet::GenerateSamples( pEvidencesVector* evidences, int
    nSamples,const CEvidence* pEv = NULL ) const;
```

Arguments

evidences   Input-output parameter. An empty vector of evidences to be created.

nSamples   Input parameter. Number of samples.

pEv   Given evidence.

Discussion

This function generates samples from the static graphical model.

IsValid

Checks model validity.

```
bool CBNet::IsValid(std::string* descriptionOut = NULL) const;
```

Arguments

descriptionOut   Output argument with an error message.

intel

Discussion

This function checks the validity of the model. Returns 'true' if the model is valid, returns 'false' otherwise.

Class CMNet
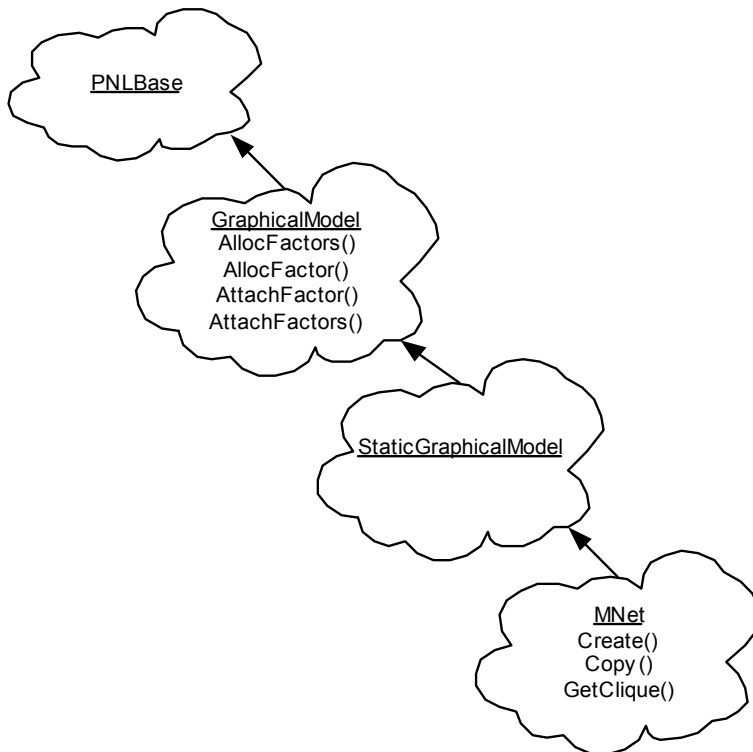
Public Member Functions

Create

Creates object of subclass.

```
static CMNet* CMNet:: Create( int numberOfCliques, const int *cliqueSizes,
    const int **cliques, CModelDomain* pMD );
static CMNet* CMNet::Create( const intVecVector& cliques, CModelDomain* pMD
static CMNet* CMNet::Create( int numberOfNodes, int numberOfNodeTypes,
    const CNodeType *nodeTypes, const int *nodesAssociation, int
    numberOfCliques, const int *cliqueSizes, const int **cliques );
static CMNet* CMNet::Create(int numberOfNodes,const nodeTypeVector& nodeTypes,
    const intVector& nodesAssociation, const intVecVector& cliques );
```

Arguments

numberOfCliques	Number of cliques in the model.
cliqueSizes	Array of clique sizes.
cliques	Cliques.
pMD	Model domain.
numberOfNodes	Number of nodes in the model.
numberOfNodeTypes	Number of node types in the model.
nodeTypes	Array of node types.
nodesAssociation	Array of nodes association with node types.

CreateWithRandomMatrices

Creates object with random matrices.

```
static CMNet* CMNet::CreateWithRandomMatrices( int numberOfCliques,const int
    *cliqueSizes,const int **cliques, CModelDomain* pMD);
staticCMNet*CMNet::CreateWithRandomMatrices(constintVecVector&cliques,
    CModelDomain* pMD);
```

Arguments

numberOfCliques	Number of cliques.
cliqueSizes	Array of clique sizes.
cliques	Cliques.
pMD	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.

```
inline void CMNet::GetClique( int clqNum, int *clqSize, const int **clq )const;
inline void CMNet::GetClique( int clqNum, intVector *clq) const;
```

Arguments

clqNum	Number of the clique.
clqSize	Returned parameter. Number of clique nodes.
clq	Returned parameter. Array of clique nodes.

intel

Discussion

This function returns clique nodes.

ConvertFromBNet

Creates class object by converting input `BNet`.

```
static CMNet* CMNet::ConvertFromBNet(const CBNet *pBNet);
```

Arguments

pBNet   Bayesian network.

Discussion

This function creates a class object by converting the input `BNet`.

ConvertFromBNetUsingEvidence

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

```
static CMNet* CMNet::ConvertFromBNetUsingEvidence(const CBNet *pBNet,
    const CEvidence *pEvidence );
```

Arguments

pBNet   Bayesian network.

pEvidence   Evidence.

Discussion

This function creates a class object by converting the input `BNet`.

Copy

Creates object by copying input MNet.

```
static CMNet* CMNet::Copy(const CMNet *pMNet);
```

Arguments

pMNet   Markov network.

Discussion

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

CreateTabularPotential

Allocates factor and creates matrix.

```
void CMNet::CreateTabularPotential( const intVector& domain,
        const floatVector& data );
```

Arguments

domain   Array of nodes.
data   Given data.

Discussion

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

ComputeLogLik

Computes logarithm of likelihood.

```
virtual float CMNet::ComputeLogLik( const CEvidence *pEv ) const;
```

Arguments

pEv   Evidence.

Discussion

This function computes the logarithm of likelihood.

GetClqsNumsForNode

Gets cliques containing node.

```
inline void CMNet::GetClqsNumsForNode( int node, intVector *clqs ) const;
```

Arguments

node   Node number.

clqs   Cliques containing the *node.*

Discussion

This function gets numbers of cliques that contain the given node.

GetNumberOfCliques

Returns number of cliques of model.

```
inline int CMNet::GetNumberOfCliques() const;
```

GenerateSamples

Generates random evidence.

```
virtual void CMNet::GenerateSamples( pEvidencesVector* evidences, int
    nSamples, const CEvidence *pEvIn = NULL ) const;
```

Arguments

evidences	Input-output parameter. An empty vector of evidences that will be created by this method.
nSamples	Input parameter. Number of samples.
pEvIn	Input parameter. Pointer to evidence.

Discussion

This function generates random evidences from `MNet`.

Class CMRF2
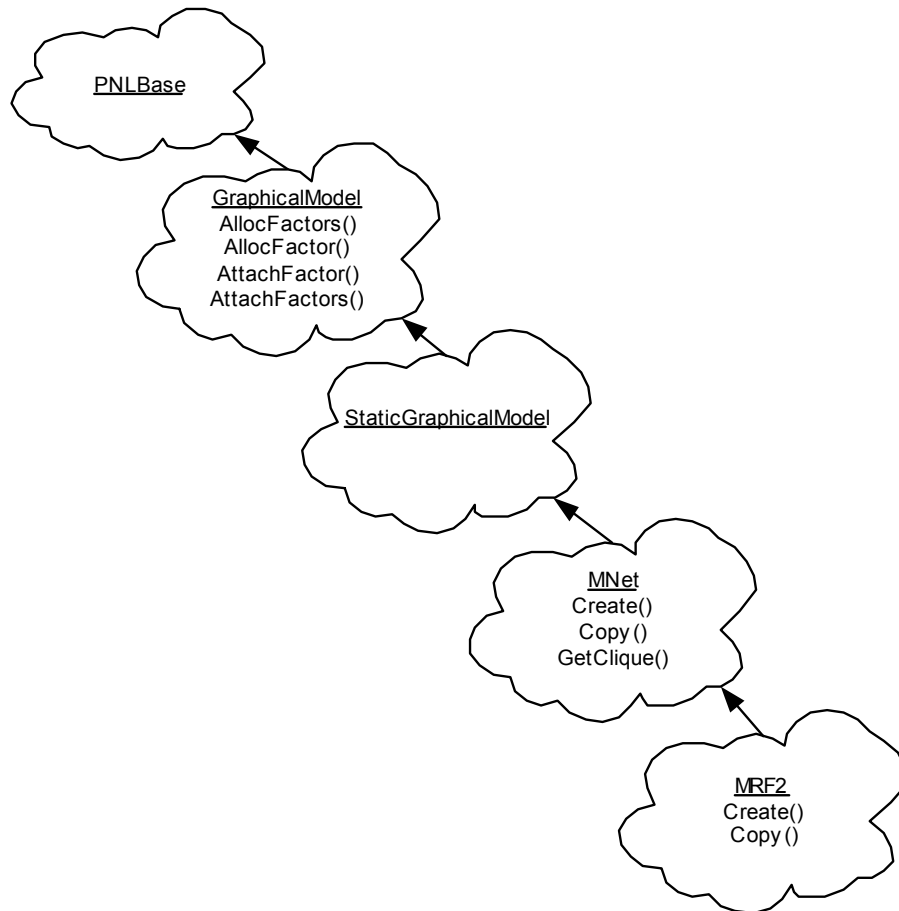
Class CMNet is a superclass for subclass CMRF2 that represents a pairwise Markov network. This subclass implements CMNet virtual functions, so that the implementation takes into account that all the cliques consist of only two nodes.

Public Member Functions

Create

Creates class object.

```
static CMRF2* CMRF2::Create( int numberOfCliques,  const int *cliqueSizesIn,
    const int **cliquesIn, CModelDomain* pMD );
```

```
static CMRF2* CMRF2::Create( const intVecVector& clqsIn, CModelDomain* pMD );
```

```
static CMRF2* CMRF2::Create( int numberOfNodes, int numberOfNodeTypes,
    const CNodeType *nodeTypesIn,const int *nodeAssociationIn, int
    numberOfCliques,const int *cliqueSizesIn, const int **cliquesIn );
```

```
static CMRF2* CMRF2::Create( int numberOfNodes, const nodeTypeVector&
    nodeTypesIn,const intVector& nodeAssociationIn, const intVecVector&
    cliquesIn );
```

Arguments

numberOfCliques	Number of cliques.
cliqueSizesIn	Returned parameter. Sizes of cliques.
cliquesIn	Cliques.
pMD	Model domain.
numberOfNodes	Number of nodes.
numberOfNodeTypes	Number of node types.
nodeTypesIn	Node types.
nodeAssociationIn	Association of nodes.

CreateWithRandomMatrices

Creates object with random matrices.

```
static CMRF2* CMRF2::CreateWithRandomMatrices( int numberOfCliques,const int
    *cliqueSizesIn,const int **cliquesIn,CModelDomain* pMD);
```

```
static CMRF2* CMRF2::CreateWithRandomMatrices( const intVecVector& clquesIn,
ModelDomain* pMD);
```

Arguments

numberOfCliques	Number of cliques.
cliqueSizesIn	Clique size.
cliquesIn	Cliques.
pMD	Model domain.

Class CFactorGraph
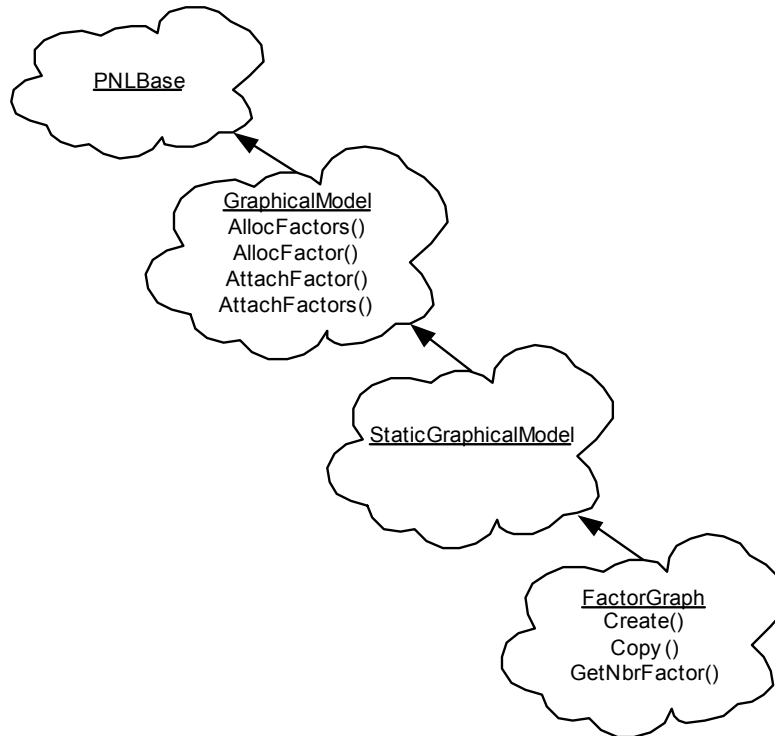
Factor graph is a graphical representation of the factorized distribution. All factors of the distribution are represented by factor-nodes, which are connected to variable-nodes lying in the factor domain. The resulting graph of the distribution is called the factor graph.

The `CFactorGraph` class is a graphical model which consists of a set of factors. The set of factors constitutes a probability distribution. All the factors are potentials.

Public Member Functions

Create

Creates class object.

```
static CFactorGraph* CFactorGraph::Create( CModelDomain* pMD, const CFactors*
    pFactors );
```

Arguments

pMD   Pointer to the model domain.

pFactors   Pointer to the CFactors object which contains all factors
                         describing the factor graph object.

Discussion

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

Create

Creates class object.

```
static CFactorGraph* CFactorGraph::Create(CModelDomain* pMD, int numFactors );
```

Arguments

pMD   Pointer to the model domain.

numFactors   Number of factors in the factor graph.

Discussion

This function creates a factor graph of several allocated factors.

Copy

Creates replica of input object.

```
static CFactorGraph* CFactorGraph::Copy( const CFactorGraph* pFG );
```

Arguments

pFG   Pointer to the `CFactorGraph` object.

Discussion

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

Shrink

Creates factor graph with given evidence by
shrinking all potentials of given factor.

```
CFactorGraph* CFactorGraph::Shrink( const CEvidence* pEvidence )const;
```

Arguments

pEvidence   Pointer to the `Evidence` object.

Discussion

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

GetNumFactorsAllocated

Returns numbers of allocated factors.

```
inline int CFactorGraph::GetNumFactorsAllocated() const;
```

ConvertFromBNet

Creates class object by converting BNet object.

```
static CFactorGraph* CFactorGraph::ConvertFromBNet( const CBNet* pBNet );
```

Arguments

pBNet   Pointer to a CBNet object to be converted.

Discussion

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

ConvertFromMNet

Creates class object by converting MNet object.

```
static CFactorGraph* CFactorGraph::ConvertFromMNet( const CMNet* pMNet );
```

Arguments

pMNet   Pointer to a CMNet object to be copied

Discussion

This function creates a CFactorGraph object by converting MNet object.

IsValid

Checks validity of function.

```
bool CFactorGraph::IsValid( std::string* descriptionOut = NULL ) const;
```

Arguments

descriptionOut   Error message.

GetNbrFactors

Returns numbers of factors.

```
inline void CFactorGraph::GetNbrFactors( int node, intVector* nbrsFactorsOut )
    const;
```

Arguments

node   Node.

nbrsFactors   Numbers of factors.

Discussion

This function returns numbers of factors neighboring to the node.

GetNbrFactors

Returns factors neighboring to given node.

```
inline void CFactorGraph::GetNbrFactors( int node, intVector* nbrsFactors )
    const;
```

Arguments

node   Number of the node.

nbrsFactors   Returned parameter.Vector of numbers of factors neighboring to 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 to given
node.*

```
inline int CFactorGraph::GetNumNbrFactors( int node ) const;
```

Arguments

node   Number of the node.

Discussion

This function returns the number of factors neighboring to the 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 Junction tree instance is created on the creation of the `JtreeInfEngine`.

Public Member Functions

Create

Creates Junction tree.

```
static CJunctionTree* CJunctionTree::Create( const CStaticGraphicalModel*
    pGrModel,const intVecVector& subGrToConnect = intVecVector() );
static CJunctionTree* CJunctionTree::Create( const CStaticGraphicalModel
    *pGrModel, int numOfSubGrToConnect = 0,const int *subGrToConnectSizes =
    NULL, const int **subGrToConnect = NULL );
```

Arguments

pGrModel	Graphical model from which the tree is to be constructed.
numOfSubGrToConnect	Number of subgraphs to be connected to the tree.
subGrToConnectSizes	Sizes of subgraphs to be connected to the tree.
subGrToConnect	Subgraphs the user wants to appear in the tree.

Copy

Creates replica of input Junction tree.

```
const CJunctionTree* pJTree
```

Arguments

pJTree	Junction tree to be copied.

Discussion

This function copies the input Junction tree.

GetNodeContent

Returns clique of Junction tree.

```
inline void CJunctionTree::GetNodeContent( int nodeNumber, int *nodeContentSz,
    const int **content ) const;
```

Arguments

nodeNumber	Number of the clique.
nodeContentSz	Returned parameter. Size of the clique.
content	Returned parameter.Pointer to the clique.

Discussion

This function returns the Junction tree clique with the number of *nodeNumber*.

GetNodesConnectedByUser

Returns set of connected nodes.

```
inline void CJunctionTree::GetNodesConnectedByUser( int nodeSetNum, int
    *numOfNds, const int **nds ) const;
```

Arguments

nodeSetNum	Number of the set of nodes.
numOfNds	Return parameter. Size of the set of nodes.
nds	Return parameter. Pointer to the set of nodes.

Discussion

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

GetFactorAssignmentToClique

Returns arrays of indices that show assignment of
input model factors to Junction tree cliques.

```
inline void CJunctionTree::GetFactorAssignmentToClique( int *numberOfFactors,
    const int **factorAssign ) const;
```

Arguments

numberOfFactors	Return parameter. Number of factors of the input model.
factorAssign	Return parameter. Pointer to an array of indices.

Discussion

This function returns arrays of indices that show to what cliques of the Junction tree input model factors are assigned.

GetSeparatorDomain

Returns domain of separator between two cliques
of Junction tree.

```
inline void CJunctionTree::GetSeparatorDomain( int firstClqNum, int
    secondClqNum, int *domSize, const int **domain ) const;
```

Arguments

firstClqNum	Number of the first clique.
secondClqNum	Number of the second clique.
domSize	Return parameter.Size of the domain on the separator.
domain	Return parameter.Pointer to the separator.

Discussion

This function returns the domain of the separator which is located between two cliques of the Junction tree.

GetNodePotential

Returns pointer to potential defined for Junction tree clique.

```
inline CPotential* CJunctionTree::GetNodePotential( int nodeNum );
```

Arguments

nodeNum   Number of the clique of the Junction tree.

Discussion

This function returns the pointer to the potential that is defined for a clique of the Junction tree.

GetSeparatorPotential

Returns pointer to potential defined for separator between two cliques.

```
inline CPotential* CJunctionTree::GetSeparatorPotential( int firstClqNum, int
    secondClqNum );
```

Arguments

firstClqNum   Number of the first clique.

secondClqNum   Number of the second clique.

Discussion

This function returns the pointer to the potential that is defined for the separator between two cliques.

GetClqNumsContainingSubset

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

```
inline void CJunctionTree::GetClqNumsContainingSubset( int numOfNdsInSubset,
    const int *subset, int *numOfClqs, const int **clqsContSubset ) const;
```

Arguments

numOfNdsInSubset	Size of the subset.
subset	Pointer to the subset of nodes.
numOfClqs	Return parameter. Number of cliques with a common subset.
clqsContSubset	Pointer to the array of numbers of cliques with a common subset.

Discussion

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

InitCharge

Initializes charge for Junction tree.

```
void CJunctionTree::InitCharge( const CStaticGraphicalModel *pGrModel, const
    CEvidence *pEvidence, int sumOnMixtureNode = 1 );
```

intel

Arguments

pGrModel	Pointer to the input graphical model.
pEvidence	Pointer to the evidence to be taken into account.
sumOnMixtureNode	Shows if the distribution for the mixture node is to be computed during inference.

Discussion

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

ClearCharge

Clears charge.

```
void CJunctionTree::ClearCharge();
```

operator=

Performs initialization.

```
CJunctionTree& CJunctionTree::operator=( const CJunctionTree &JTree );
```

Arguments

JTree	RHS of the assignment operator.

Discussion

This function performs initialisation by copying potentials of one tree to another tree under the condition that the structures of Junction trees are identical.

GetNumberOfNodes

Returns numbers of Junction tree nodes.

```
inline int CJunctionTree::GetNumberOfNodes() const;
```

Discussion

This function returns numbers of Junction tree nodes. The node numbers correspond to the numbers of the Junction tree cliques.

DumpNodeContents

Dumps Junction tree cliques to outstream.

```
inline void CJunctionTree::DumpNodeContents() const;
```

Discussion

This function dumps Junction tree cliques out to the outstream. Standard output of the outstream is set by default but may be redirected.

Class CDynamicGraphicalModel
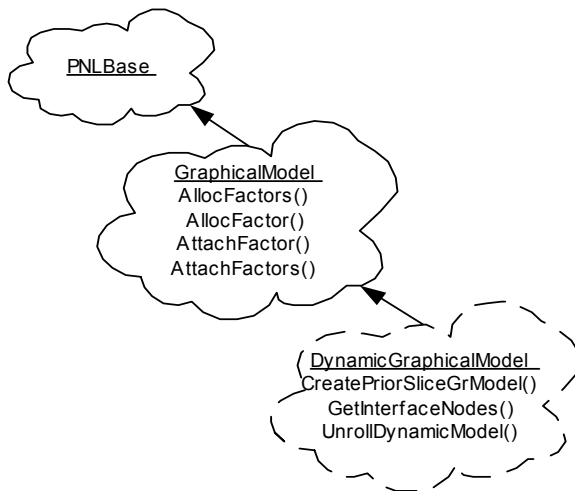
Class `CDynamicGraphicalModel` is a superclass for all classes that implement dynamic graphical models.

Public Member Functions

CreatePriorSliceGrModel

Creates static graphical model.

```
virtual CStaticGraphicalModel*
    CDynamicGraphicalModel::CreatePriorSliceGrModel()const = 0;
```

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 of dynamic graphical model.

```
CStaticGraphicalModel* CDynamicGraphicalModel::UnrollDynamicModel( int
    numOfSlices );
```

Arguments

numOfSlices   Number of slices.

Discussion

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

GetInterfaceNodes

Returns numbers of interface nodes.

```
inline void CDynamicGraphicalModel::GetInterfaceNodes( intVector*
    interfaceNds) const;
inline void CDynamicGraphicalModel::GetInterfaceNodes( int *numOfNds, const
    int **interfaceNds ) const;
```

Arguments

numOfNds   Returned parameter. Number of interface nodes.

interfaceNds   Returned parameter. Array of interface nodes.

Discussion

This function returns numbers of interface nodes.

GetStaticModel

Returns a pointer to static graphical model.

```
inline CStaticGraphicalModel* CDynamicGraphicalModel::GetStaticModel() const;
```

Discussion

This member function returns a pointer to the static graphical model used for creating a dynamic graphical model.

Class CDBN
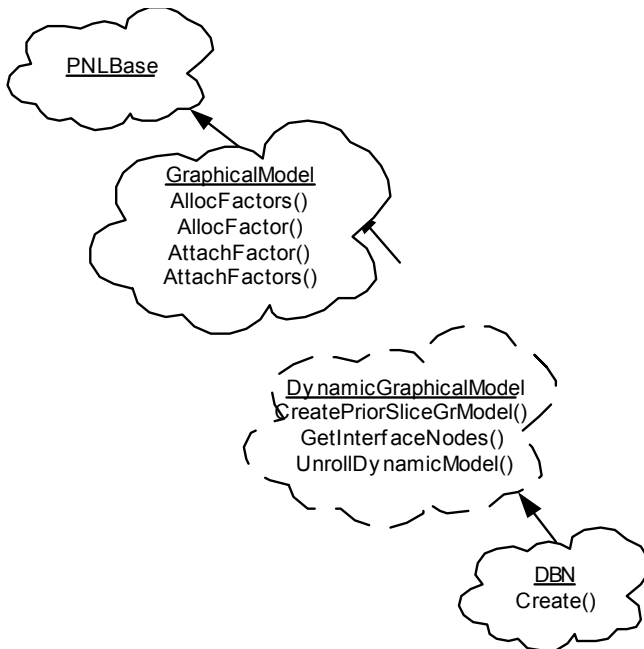
CDBN is a subclass of `Class CDynamicGraphicalModel` and implements virtual functions of the parent class.

Create

Creates class object.

```
static CDBN* CDBN::Create( CStaticGraphicalModel *pGrModel );
```

Arguments

pGrModel   Pointer to `BNet`, which is to represent a DBN unrolled for
                               first two time-slices.

Discussion

Thi function creates a `CDBN` object.

GenerateSamples

Generates samples from DBN.

```
void CDBN::GenerateSamples( pEvidencesVecVector* evidences,const intVector&
    nSlices) const;
```

Arguments

evidences   Output parameter. Generated evidence.

nSlices   Input parameter. Number of slices for which evidence is generated.

Discussion

Generates samples from the dynamic graphical model.

Distribution Functions

Class CDistribFun

---
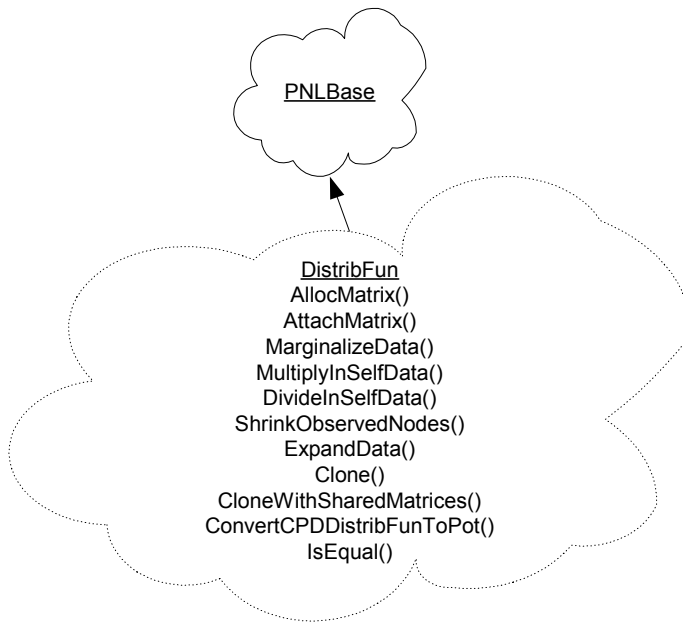
operator=

Assigns data to object from input distribution function.

```
virtual CDistribFun& CDistribFun::operator = ( const CDistribFun& rDistrFun );
```

Arguments

rDistrFun   Reference to the `CDistribFun` object.

Discussion

This function assigns data from the input factor to the object for which it is called only when both of them are of the same size and type.

GetNodeTypesVector

Returns node types to nodes for which it was created.

```
inline const pConstNodeTypeVector CDistribFun::*GetNodeTypesVector() const;
```

Discussion

This function returns node types to the nodes for which it was created.

SetVariableType

Sets position for node type in distribution.

```
inline void CDistribFun::SetVariableType( int position, const CNodeType*
    varType );
```

Arguments

position	Order number.
varType	Pointer to node types.

Discussion

This function sets a certain position for the given node type in the distribution.

IsValid

Checks validity of distribution function.

```
virtual bool CDistribFun::IsValid( std::string* discription = NULL ) const = 0;
```

Arguments

discription   Error message*.*

Discussion

This function checks if the distribution function is valid.

AllocMatrix

Creates matrix and allocates it to factor.

```
virtual void CDistribFun::AllocMatrix( const float *data, EMatrixType mType,
    int numberOfWeightMatrix = -1,  const int *parentIndices = NULL );
```

Arguments

data	Array that corresponds to a certain part of the distribution.
mType	Type of matrix allocated to the factor.
numberOfWeightMatrix	Number of the matrix called if several matrices of a given type are associated with the factor.This argument is omitted if only one matrix is involved.
parentIndices	Array of values of discrete parents.

Discussion

This function creates a new matrix and allocates it to the distribution function.

AttachMatrix

Enters data in matrix and associates matrix with distribution function.

```
virtual void CDistribFun::AttachMatrix( CMatrix<float> *pMatrix,  EMatrixType
    mType,  int numberOfWeightMatrix = -1,  const int *parentIndices = NULL ) =
    0;
```

Arguments

pMatrix	Pointer to the `CMultiDMatrix` object.
mTypes	Matrix type.
numberOfWeightMatrix	Number of the matrix called if several matrices of a given type are associated with the factor. The argument is optional. It is omitted when one matrix is involved.
parentIndices	Array of values of discrete parents.

Discussion

This function enters data into the matrix and associates the matrix with the distribution function.

GetMatrix

Returns pointer to matrix attached to factor.

```
virtual CMatrix<float> CDistribFun::*GetMatrix( EMatrixType mType,  int
    numWeightMat = -1, const int *parentIndices = NULL ) const = 0;
```

Arguments

mType	Type of the matrix called.

`numWeightMat`	Number of the matrix called among several matrices of the given type. The argument is omitted when only one matrix is involved.
`paretnIndices`	Array of values of discrete parents.

Discussion

This function returns the pointer to a matrix attached to the factor. The function identifies one of the following matrix types: `matTable, matMean, matCov, matWeights, math,` and `matK.`

GetNumberOfNodes

Returns size of distribution function domain.

```
inline int CDistribFun::GetNumberOfNodes() const;
```

Discussion

This function returns the size of the domain corresponding to the distribution function.

IsDistributionSpecific

Checks whether distribution is specific.

```
virtual inline int CDistribFun::IsDistributionSpecific() const = 0;
```

Discussion

This function checks whether the distribution is specific and returns:

• 0 - if the distribution is full. It may be Tabular, Gaussian, Conditional Gaussian, non-delta, non-uniform, non-mixed.

To check the validity of the function call `IsValid` function.

• 1 - if the distribution is uniform. A distribution is uniform if it has no attached matrices. A uniform distribution has the flag indicating its type.

• 2 - if the distribution is delta function. A delta function has only one mean matrix.

 To check if the function is valid call `IsValid` function.

• 3 - if the distribution is mixed. A distribution is mixed if it is created by multiplication of a certain distribution by delta function in some distribution dimensions. A mixed distribution is an intermediate distribution representation.

GetStatisticalMatrix

Returns statistical matrix.

```
virtual CMatrix<float>* CDistribFun::GetStatisticalMatrix
    (EStatisticalMatrixType mType,int *parentIndices = NULL ) const = 0;
```

Arguments

`mType`	Matrix type.
`parentIndices`	Array of values of discrete parents.

Discussion

This function returns the pointer to the matrix attached to the factor. It identifies one of the following matrix types: *stMatTable, stMatMu, stMatSigma, stMatCoeff.*

MarginalizeData

Marginalizes object.

```
virtual void CDistribFun::MarginalizeData( const CDistribFun *pOldData,  const
    int *DimsOfKeep, int NumDimsOfKeep, int maximize ) = 0;
```

Arguments

pOldData	Pointer to the distribution function.
DimsOfKeep	Pointer to an array of numbers of the dimensions that should constitute the domain of the returned marginalized object.
NumDimsOfKeep	Size of the returned object domain.
maximize	Flag of the marginalization type: - for discrete variables: • 0 stands for simple summation; • 1 stands for finding maximum value. - for continuous variables: • both are integration operations.

Discussion

This function converts the source object to a new `CDistributionFunction` object that is generated either by adding or by integrating the source object referring to the nodes that do not lie in the returned object domain. The returned object domain should be a subset of the source domain.

MultiplyInSelfData

*Multiplies and puts result into distribution
function.*

```
virtual void CDistribFun::MultiplyInSelfData( const int *pBigDomain, const int
    *pSmallDomain, const CDistribFun *pOtherData ) = 0;
```

Arguments

pBigDomain	Dimensions numbers of the domain of the larger distribution function for which the function is called.
pSmallDomain	Dimensions numbers of the domain of the smaller distribution function.
pOtherData	Reference to the multiplier of `CDistribFun` type.

Discussion

This function changes the object, for which it is called, by changing its data matrices.

DivideInSelfData

Divides object.

```
virtual void CDistribFun::DivideInSelfData( const int *pBigDomain, const int
    *pSmallDomain, const CDistribFun *pOtherData );
```

Arguments

pBigDomain	Domain of the divident.
pSmallDomain	Domain of the divisor.
pOtherData	Pointer to the denominator of `CDistribFun` type.

Discussion

This function changes the object, for which it is called, by changing its data matrices.

ShrinkObservedNodes

Creates new distribution function by shrinking dimensions corresponding to observed nodes.

```
virtual void CDistribFun::ShrinkObservedNodes( const CDistribFun* pOldData,
    const int *pDimsOfObserved, const Value* const* pObsValues, int numObsDim,
    const CNodeType* pObsTabNT, const CNodeType* pObsGauNT );
```

Arguments

pOldData	Pointer to the source data.

pDimsOfObserved	Array of observed dimensions.
pObsValues	Array of observed values.
numObsDim	Number of observed dimensions.
pObsTabNT	Pointer to the observed Tabular node type.
pObsGauNT	Pointer to the observed Gaussian node type.

Discussion

This function creates a new distribution function with the same number of nodes as in the domain of the source distribution function but with modified observed nodes. Due to the change of values of the observed nodes the joint probability distribution changes too.

ExpandData

Expands dimensions corresponding to observation.

```
virtual void CDistribFun::ExpandData( const int* pDimsToExpand, int
    numDimsToExpand, const Value* const* valuesArray, const CNodeType* const
    *allFullNodeTypes, int UpdateCanonical = 1 );
```

Arguments

pDimsToExpand	Array of dimensions to expand.
numDimsToExpand	Number of dimensions to expand.
valuesArray	Array of observed values.
allFullNoodeTypes	Array of pointers of node types of variables in the distribution.
UpdateCanonical	Flag used for Gaussian distributions: 1 – updates the canonical form of the distribution, 0 – does not update the canonical form of the distribution.

Discussion

This function expands the probability distribution by filling empty spaces with zeros.

ClearStatisticalData

Sets to zero all elements of matrices used in learning process.

```
virtual void CDistribFun::ClearStatisticalData() = 0;
```

Discussion

This function sets to zero all elements of the matrices which are used in the learning process.

UpdateStatisticsEM

Updates statistical data.

```
virtual void CDistribFun::StatisticalDataEM( const CDistribFun* infData, const
    CEvidence *pEvidence = NULL, float weightingCoeff = 1.0f, const int* domain
    = NULL ) = 0;
```

Arguments

InfData	Pointer to the distribution function inference result.
pEvidence	Pointer to an *Evidence* object.
WeightingCoeff	Weighting coefficient.
domain	Domain node numbers.

Discussion

This function estimates factors and updates statistical data.

UpdateStatisticsML

Gathers statistical data.

```
virtual void CDistribFun::StatisticalDataML( const CEvidence* const*
    pEvidences, int EvidenceNumber,const int *domain, float weightingCoeff =
    1.0f ) = 0;
```

Arguments

pEvidences	Array of evidences.
EvidenceNumber	Number of evidences.
domain	Numbers of domain nodes.
WeightingCoeff	Weighting coefficient.

Discussion

This function estimates factors and updates statistical data.

SetStatistics

Sets statistical data.

```
virtual void CDistribFun::SetStatistics( const CMatrix<float>* pMat,
    EStatisticalMatrix matrixType, const int* parentsComb = NULL ) = 0;
```

Arguments

pMat	Input parameter. Matrix with statistical data.
matrixType	Type of matrix.
parentsComb	Combination of discrete parents.

Discussion

This function sets statistical data for learning.

GetNormalized

Normalizes distribution function.

```
virtual CDistribFun* CDistribFun::GetNormalized() const = 0;
```

Discussion

This function creates a new distribution function by normalizing the given distribution function.

ProcessingStatisticalData

Updates distribution function after gathering statistical data.

```
virtual float CDistribFun::ProcessingStatisticalData( float numEvidences ) =
    0;
```

Arguments

numEvidences   Number of evidences.

Discussion

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

Clone

Creates replica of distribution function.

```
virtual CDistribFun* CDistribFun::CloneDistribFun() const = 0;
```

CloneWithSharedMartices

Creates replica of distribution function.

```
virtual CDistribFun* CDistribFun::CloneWithSharedMatrices() const = 0;
```

Discussion

This function creates a replica of the distribution function so that the newly created distribution function shares its matrices with the source distribution function.

GetMultipliedDelta

Returns delta distributions that are multiplied by given distribution.

```
virtual int CDistribFun::GetMultipliedDelta( const int **positions, const
    float **values, const int **offsets ) const = 0;
```

Arguments

positions	Returned parameter. Array of positions multiplied by delta distribution.
values	Returned parameter. Mean values of delta distributions.
offsets	Returned parameter. Array of offsets to next mean values.

Discussion

This function returns delta distributions that are multiplied by the distribution. The returned integer value is the number of positions multiplied by delta distributions.

ConvertCPDDistribFunToPot

Converts distribution function for CPD into distribution function for potential.

```
virtual CDistribFun CDistribFun::*ConvertCPDDistribFunToPot()const = 0;
```

Discussion

Converts the distribution function which was created for use in a Conditional Probability Distribution (CPD) into the distribution function for use in a potential.

CPD_to_pi

Computes pi message for Pearl inference.

```
virtual CDistribFun CDistribFun::*CPD_to_pi( CDistribFun *const*
    allPiMessages, int *multParentIndices, int numMultNodes, int
    posOfExceptParent, int maximizeFlag = 0 )const = 0;
```

Arguments

allPiMessages	Array of all pi messages received by the current distribution.
multParentIndices	Indices of parent nodes.
numMultNodes	Number of parent nodes.
posOfExceptParent	Position of the parent which is not to be multiplied.
maximizeFlag	Flag of maximization in multiplication process.

Discussion

This function computes pi messages for Pearl inference.

CPD_to_lambda

Computes lambda message for Pearl inference.

```
virtual CDistribFun CDistribFun::*CPD_to_lambda( const CDistribFun *lambda,
    CDistribFun *const* allPiMessages, int *multParentIndices, int
    numMultNodes, int posOfExceptNode, int maximizeFlag = 0 );
```

Arguments

lambda	Received Lambda message.
allPiMessages	Array of all pi messages received by the current distribution.
multParentIndices	Indices of parent nodes.
numMultNodes	Number of parent nodes.
posOfExceptNode	Position of the parent that is not to be multiplied.
maximizeFlag	Flag of maximisation in multiplication process.

Normalize

Normalizes distribution function.

```
virtual CDistribFun* CDistribFun::Normalize();
```

Discussion

This function normalizes distribution function.

GetDistributionType

Returns distribution type.

```
EDistributionType CDistribFun::GetDistributionType() const;
```

Discussion

This function returns one of the following distribution types: *dtTabular*, *dtGaussian* or *dtCondGaussian*.

IsEqual

Compares distributions.

```
virtual int CDistribFun::IsEqual( const CDistribFun *dataToCompare, float
    epsilon, int withCoeff = 1) const = 0;
```

Arguments

dataToCompare	Pointer to the distribution function for comparison.
epsilon	Float value of accuracy to compare.
WithCoeff	Flag of the type of comparison. Normalizing constants for Gaussian and Conditional Gaussian distribution are compared if it is equal to 0.

Discussion

This function returns 1 if the compared distributions are of the same size, type and have the same floating point matrices, returns 0 otherwise.

GetMPE

Returns maximum probability explanation.

```
virtual CNodeValues* CDistribFun::GetMPE() = 0;
```

Discussion

This function returns maximum probability explanation of the distribution function.

ConvertToSparse

Converts distribution with dense matrices into distribution with sparse matrices.

```
virtual CDistribFun* CDistribFun::ConvertToSparse() const = 0;
```

Discussion

This function converts the distribution with dense matrices into the distribution with sparse matrices.

ConvertToDense

Converts distribution with sparce matrices into distribution with dense matrices.

```
virtual CDistribFun* CDistribFun::ConvertToDense() const = 0;
```

Discussion

This function converts the distribution with sparse matrices into the distribution with dense matrices.

IsSparse

Checks if distribution matrices are sparce.

```
virtual int CDistribFun::IsSparse() const = 0;
```

Discussion

This function checks if the matrices of the distribution are sparce.

IsDense

Checks if distribution matrices are dense.

```
virtual int CDistribFun::IsDense() const = 0;
```

Discussion

This function checks if the matrices of the distribution are dense.

ResetNodeTypes

Replaces node types of model domain by node
types of distribution function.

```
inline void CDistribFun::ResetNodeTypes( pConstNodeTypeVector &nodeTypes );
```

Arguments

nodeTypes   Array of `CNodeTypes` objects.

Discussion

This function replaces node types of the model domain by identical node types from another model domain.

CreateDefaultMatrices

Allocates default matrices to distribution function.

```
virtual void CDistribFun::CreateDefaultMatrices( int isRandom = 1 );
```

Arguments

isRandom   Type of matrix data.

Discussion

This function creates default matrices and allocates them to the distribution function.

GetMatricesValidityFlag

Checks validity of matrices.

```
inline bool CDistribFun::GetMatricesValidityFlag() const;
```

Dump

Dumps content of object.

```
virtual void CDistribFun::Dump() const = 0;
```

Class CTabularDistribFun
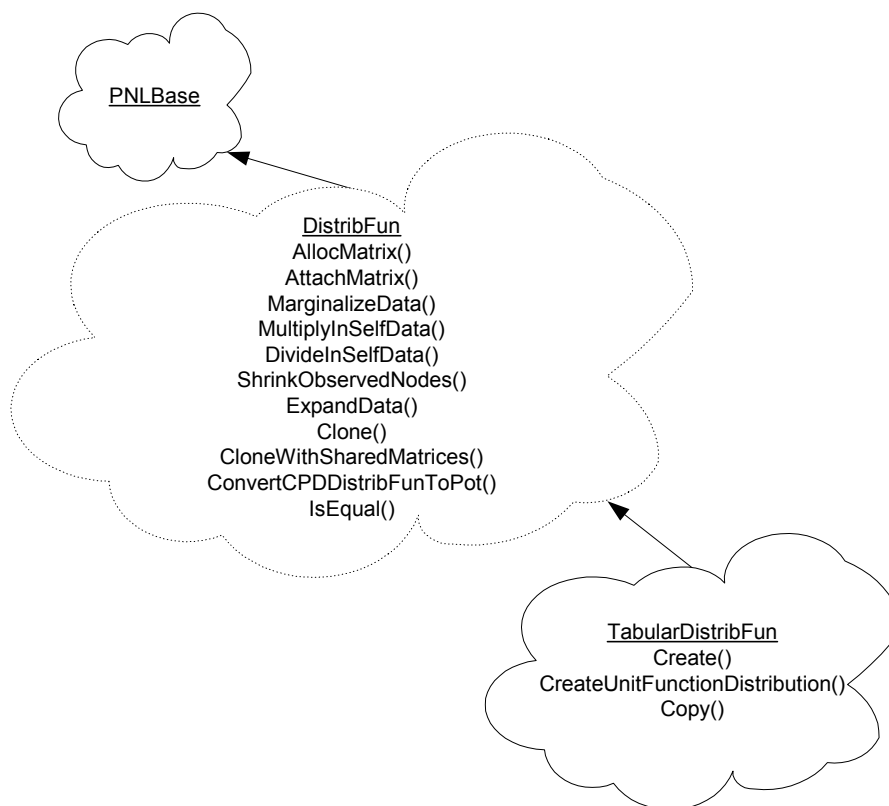
PNLBase

DistribFun
AllocMatrix()
AttachMatrix()
MarginalizeData()
MultiplyInSelfData()
DivideInSelfData()
ShrinkObservedNodes()
ExpandData()
Clone()
CloneWithSharedMatrices()
ConvertCPDDistribFunToPot()
IsEqual()

TabularDistribFun
Create()
CreateUnitFunctionDistribution()
Copy()

Create

Creates class object.

```
static CTabularDistribFun* CTabularDistribFun::Create( int NodeNumber, const
    CNodeType *const* NodeTypes, const float *data, int allocMatrices = 0, int
    asDense = -1  );
```

Arguments

NodeNumber	Number of nodes in the domain.
NodeTypes	Pointer to the array of CNodeTypes of the nodes in the domain.
data	Pointer to the array of float values of probabilities.
allocMatrices	Flag of matrix allocation.
asDense	Type of the martix created.

Discussion

This function creates a class object.

CreateUnitFunctionDistribution

Creates unit-function distribution.

```
static CTabularDistribFun* CTabularDistribFun::CreateUnitFunctionDistribution
    ( int NumberOfNodes, const CNodeType *const*nodeTypes, int asDense = 1 );
```

Arguments

NumberOfNodes	Number of nodes in the domain.
nodeTypes	Pointer to the array of *CNodeTypes* of the domain nodes.
asDense	Distribution function with dense matrices.

Discussion

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

Copy

Creates class object by copying input object.

```
static CTabularDistribFun* CTabularDistribFun::Copy(const CTabularDistribFun*
    pInpDistr );
```

BayesUpdateFactor

Updates statistical data.

```
void CTabularDistribFun::BayesUpdateFactor( const CEvidence* const*
    pEvidences, int nEv, const int* domain );
```

Arguments

pEvidences	Array of evidences.
nEv	Number of evidences.
domain	Array of nodes.

Discussion

This function updates statistical data by using priors.

PriorToCPD

Converts pseudo counts to probability.|

```
void CTabularDistribFun::PriorToCPD();
```

CPDToPi

Computes pi message for Pearl inference.

```
void CTabularDistribFun::CPDToPi( CDistribFun *const* allPiMessages, int
    *multParentIndices, int numMultNodes, int posOfExceptParent, int
    maximizeFlag = 0 ) const;
```

Arguments

allPiMessages	Array of all pi messages received by the current distribution.
multParentIndices	Indices of parent nodes.
numMultNodes	Number of parent nodes.
posOfExceptParent	Position of the parent which is not to be multiplied.
maximizeFlag	Flag of maximization in multiplication process.

Discussion

This function computes pi messages for Pearl inference.

CPDToLambda

Computes lambda message for Pearl inference.

```
void CTabularDistribFun::CPDToLambda( const CDistribFun *lambda, CDistribFun
    *const* allPiMessages, int *multParentIndices, int numMultNodes, int
    posOfExceptNode, int maximizeFlag = 0 ) const;
```

Arguments

lambda	Received Lambda message.
allPiMessages	Array of all pi messages received by the current distribution.
multParentIndices	Indices of parent nodes.
numMultNodes	Number of parent nodes.
posOfExceptNode	Position of the parent that is not to be multiplied.
maximizeFlag	Flag of maximisation in multiplication process.

Discussion

This function computes lambda message for Pearl inference.

IsMatrixNormalizedForCPD

Checks if martix is normalized for CPD.

```
bool CTabularDistribFun::IsMatrixNormalizedForCPD( float eps = 1e-5f ) const;
```

Arguments

eps	Accuracy.

Discussion

This function checks if the matirx is normalized for CPD.

Marginalize

Marginalizes object.

```
virtual void CTabularDistribFun::MarginalizeData( const CDistribFun *pOldData,
    const int *DimsOfKeep, int NumDimsOfKeep, int maximize ) = 0;
```

Arguments

pOldData	Pointer to the distribution function.
DimsOfKeep	Pointer to an array of numbers of the dimensions that should constitute the domain of the returned marginalized object.
NumDimsOfKeep	Size of the returned object domain.
maximize	Flag of the marginalization type. For discrete variables: • 0 stands for simple summation • 1 stands for finding maximum value. For continuous variables: • both are integration operations.

Discussion

This function marginalizes the source object. The object is marginalized in itself and thus should have an appropriate size.

Class CGaussianDistribFun

PNLBase

DistribFun
AllocMatrix()
AttachMatrix()
MarginalizeData()
MultiplyInSelfData()
DivideInSelfData()
ShrinkObservedNodes()
ExpandData()
Clone()
CloneWithSharedMatrices()
ConvertCPDDistribFunToPot()
IsEqual()

GaussianDistribFun
CreateInMomentForm()
CreateInCanonicalForm()
CreateDeltaDistribution()
CreateUnitFunctionDistribution
Copy
UpdateMomentForm
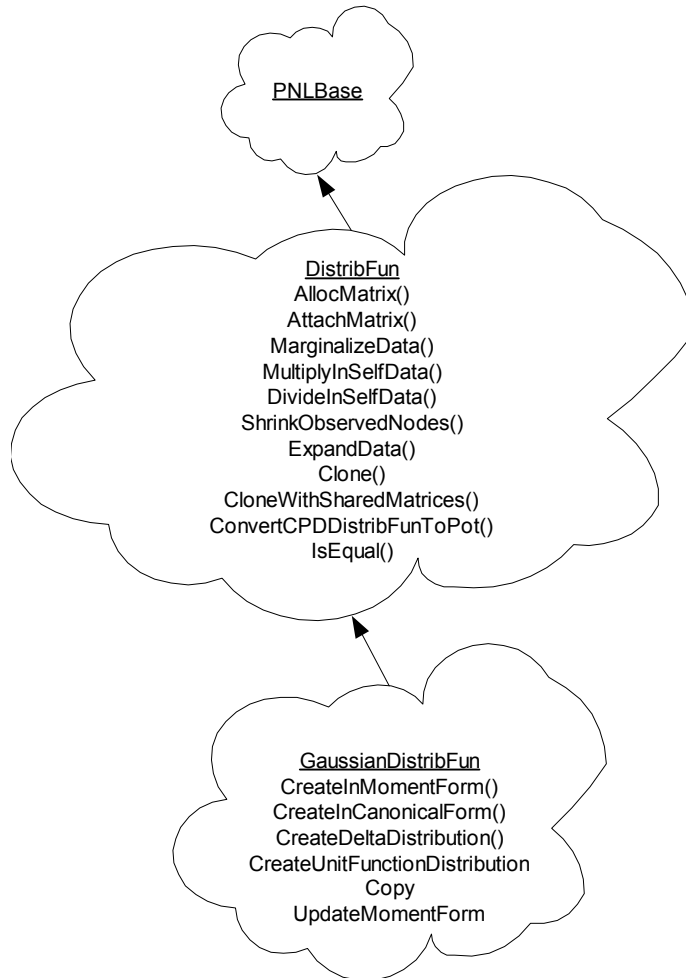
CreateInMomentForm

Creates class object in moment form.

```
static CGaussianDistribFun* CGaussianDistribFun::CreateInMomentForm( int
    isPot, int NumberOfNodes, const CNodeType *const* NodeTypes, const float
    *dataMean, const float *dataCov, const float **dataWeight = NULL );
```

Arguments

isPot	Flag of the desired Gaussian distribution use:
	1 - distribution is created for use in potential
	0 - distribution is created for use in CPD.
NumberOfNodes	Number of nodes in the domain.
NodeTypes	Pointer to the array of *CNodeTypes* of nodes in the domain.
dataMean	Pointer to the array of float values of data for matrix mean.
dataCov	Pointer to the array of float values of data for covariance matrix.
dataWeight	Pointers to weights of distributions.

Discussion

This function creates `CGaussianDistribFun` class object in the moment form.

CreateInCanonicalForm

Returns class object in canonical form.

```
static CGaussianDistribFun* CGaussianDistribFun::CreateInCanonicalForm( int
    numberOfNodes, const CNodeType *const* nodeTypes, const float *dataH, const
    float *dataK, float g = 0.0f );
```

Arguments

numberOfNodes	Number of nodes in domain.
nodeTypes	Pointer to the array of *CNodeTypes* of nodes in domain.
dataH	Pointer to the array of float values of data for matrix H.
dataK	Pointer to the array of float values of data for matrix K.

intel

g   Float value of normalisation constant in the canonical form.

Discussion

This function creates a `CGaussianDistribFun` object in the canonical form.

CreateDeltaDistribution

Creates class object of special form.

```
static CGaussianDistribFun* CGaussianDistribFun::CreateDeltaDistribution( int
    numberOfNodes, const CNodeType *const* nodeTypes, const float *dataMean,
    int isMoment = 1 );
```

Arguments

numberOfNodes	Number of nodes in the domain.
nodeTypes	Pointer to the array of `CNodeTypes` of nodes in the domain.
dataMean	Pointer to the array of float values of data for matrix mean.
isMoment	Flag of the desired form of Gaussian distribution: 1 - moment form, 0 - canonical form.

Discussion

This function creates a class object of the special form.

CreateUnitFunctionDistribution

Creates class object of special form.

```
static CGaussianDistribFun*
    CGaussianDistribFun::CreateUnitFunctionDistribution ( int numberOfNodes,
    const CNodeType *const*nodeTypes, int isPotential = 1, int isCanonical =
    1);
```

Arguments

numberOfNodes	Number of nodes in the domain.
nodeTypes	Pointer to the array of *CNodeTypes* of nodes in the domain.
isPotential	Flag of the desired use of the Gaussian distribution: 1 – distribution is created for use in potential, 0 - distribution is created for use in CPD
isCanonical	Flag of the desired form of Gaussian distribution: 0 - moment form, 1 - canonical form.

Discussion

This function creates a class object of the special form.

Copy

Creates replica of given class object.

```
static CGaussianDistribFun* CGaussianDistribFun::Copy( const
    CGaussianDistribFun* pInpDistr );
```

Arguments

pInpDistr	Pointer to the CGaussianDistribFun object to be copied.

Discussion

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

CheckMomentFormValidity

Checks if function is valid in moment form.

```
int CGaussianDistribFun::CheckMomentFormValidity();
```

CheckCanonicalFormValidity

Checks if function is valid in canonical form.

```
int CGaussianDistribFun::CheckCanonicalFormValidity();
```

GetCanonicalFormFlag

Checks if distribution function is canonical.

```
inline int CGaussianDistribFun::GetCanonicalFormFlag() const;
```

Discussion

This function checks if the function is canonical. The function returns 1 if the distribution function is canonical, returns 0 otherwise.

GetMomentFormFlag

Checks if distribution function is in moment form.

```
Inline int CGaussianDistribFun::GetMomentFormFlag() const;
```

Discussion

This function returns 1 if the distribution function is in moment form, returns 0 otherwise.

SetCoefficient

Sets normalization coefficient.

```
void CGaussianDistribFun::SetCoefficient( float coeff, int isCanonical = 1);
```

Arguments

coeff   Coefficient value.

isCanonical   Flag of distribution form.

Discussion

This function sets the normalization coefficient for the Gaussian distribution.

GetFactorFlag

Checks if function is for use in potential.

```
inline int CGaussianDistribFun::GetFactorFlag() const;
```

UpdateMomentForm

Updates moment form of distribution function.

```
void CGaussianDistribFun::UpdateMomentForm();
```

UpdateCanonicalForm

Updates canonical form of distribution function.

```
void CGaussianDistribFun::UpdateCanonicalForm();
```

1ComputeProbability

Computes probability of data setting.

```
double CGaussianDistribFun::CoputeProbability( const
    C2DNumericDenseMatrix<float>* pMatVariable, int asLog = 1, int
    numObsParents = 0, const int* obsParentsIndices = NULL,
    C2DNumericDenseMatrix<float>* const* pObsParentsMats = NULL ) const;
```

Arguments

pMatVariable	Pointer to the matrix with the given data.
asLog	Flag of taking the logarithm.
numObsParents	Number of observed parents.
obsParentsIndices	Indices.
pObsParentsMats	Matrices with the given data.

Discussion

This function computes likelihood of the given data.

Class CCondGaussianDistribFun

PNLBase

DistribFun
AllocMatrix()
AttachMatrix()
MarginalizeData()
MultiplyInSelfData()
DivideInSelfData()
ShrinkObservedNodes()
ExpandData()
Clone()
CloneWithSharedMatrices()
ConvertCPDDistribFunToPot()
IsEqual()

CondGaussianDistribFun
Create()
Copy()
SetDistribFun()
AllocDistribFun()
EnterDiscreteEvidence()
EnterFullContinuousEvidence()
GetDiscreteParentsIndices()
GetMatrixWithDistribution()

Create

Returns `CCondGaussianDistribFun` *class object*
in moment form.

```
static CCondGaussianDistribFun* CCondGaussianDistribFun::Create( int isPot,
    int nNodes, const CNodeType *const* nodeTypes, int asDenseMatrix = 1,
    CGaussianDistribFun* const pDefaultDistr = NULL );
```

Arguments

isPot	Flag of the desired use of the Gaussian distribution: 1 - distribution is created for use in potential, 0 - distribution is created for use in CPD.
nNodes	Number of nodes in the domain.
nodeTypes	Pointer to the array of *CNodeTypes* of nodes in the domain.
asDenseMatrix	Flag of the desired form of matrices of Gaussian distribution: 1 - dense 0 - sparse.
pDefaultDistr	Pointer to the Gaussian distribution, if it is used for all combinations of parents with the same matrices.

Copy

Creates class object by copying input object.

```
static CCondGaussianDistribFun* CCondGaussianDistribFun::Copy( const
    CCondGaussianDistribFun* pInputDistr );
```

Arguments

pInpDistr	Pointer to `CCondGaussianDistribFun` object.

Discussion

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

EnterDiscreteEvidence

Enters discrete evidence and creates new distribution function.

```
CCondGaussianDistribFun* CCondGaussianDistribFun::EnterDiscreteEvidence( int
    nDiscrObsNodes, const int* discrObsNodes, const int *discrValues, const
    CNodeType* pObsTabNodeType ) const;
```

Arguments

nDiscrObsNodes	Number of discrete observed nodes.
discrObsNodes	Positions of discrete observed nodes.
discrValues	Array of discrete values.
pObsTabNodeType	Pointer to the observed Tabular node type.

Discussion

This function creates a new distribution function entering discrete evidence.

EnterFullContinuousEvidence

Enters continuous evidence and creates new tabular distribution.

```
CTabularDistribFun* CCondGaussianDistribFun::EnterFullContinuousEvidence( int
    nContObsParents, const int* contObsParentsIndices, const
    C2DNumericDenseMatrix<float>* obsChildValue, C2DNumericDenseMatrix<float>*
    const* obsValues, const CNodeType* pObsGauNodeType ) const;
```

intel.

Arguments

nContObsParents	Number of observed continuous parents.
contObsParentsIndices	Positions of observed continuous parents.
obsChildValue	Value of the observed child node.
obsValues	Array of values of observed parents.
pObsGauNodeType	Pointer to the observed Gaussian node type.

Discussion

Creates a new tabular distribution by means of entering continuous evidence.

GetDiscreteParentsIndices

Returns discrete parent indices.

```
inline void CCondGaussianDistribFun::GetDiscreteParentsIndices( intVector*
    const discrParents ) const;
```

Arguments

discrParents   Output parameter.Node numbers of discrete parents in the domain.

Discussion

This function returns indices of discrete parents.

GetContinuousParentsIndices

Returns continuous parent indices.

```
inline void CCondGaussianDistribFun::GetContinuousParentsIndices( intVector*
    const contParents ) const;
```

Arguments

contParents   Output parameter.Node numbers of continuous parents in the domain.

Discussion

This function returns indices of continuous parents.

GetMatrixWithDistribution

Returns matrix with Gaussian distribution function.

```
inline CMatrix<CGaussianDistribFun*>*
    CCondGaussianDistribFun::GetMatrixWithDistribution();
```

Discussion

This function returns the coefficient for the input parent combination of the Gaussian distribution.

SetCoefficient

Sets normalization coefficient.

```
void CCondGaussianDistribFun::SetCoefficient( float coeff, int
    isCanonical,const int* pParentCombination );
```

Arguments

coeff   Coefficient to be set.

isCanonical   Flag of the distribution form.

pParentCombination   Pointer to the combination of discrete parents.

Discussion

This function sets normalization coefficient.

GetCoefficient

Gets value of normalization coefficient.

```
float CCondGaussianDistribFun::GetCoefficient( int isCanonical, const int*
    pParentCombination );
```

Arguments

isCanonical	Flag of distribution form.
pParentCombination	Pointer to combination of parents.

Discussion

This function gets the value of the normalization coefficient.

GetMatrixNumEvidences

Returns matrix with number of evidences of discrete parents.

```
inline CDenseMatrix<float>* CCondGaussianDistribFun::GetMatrixNumEvidences();
```

Dump

Dumps content of object.

```
void CCondGaussianDistribFun::Dump();
```

Class ScalarDistribFun

PNLBase

DistribFun
AllocMatrix()
AttachMatrix()
MarginalizeData()
MultiplyInSelfData()
DivideInSelfData()
ShrinkObservedNodes()
ExpandData()
Clone()
CloneWithSharedMatrices()
ConvertCPDDistribFunToPot()
IsEqual()

ScalarDistribFun
Create()
Copy()
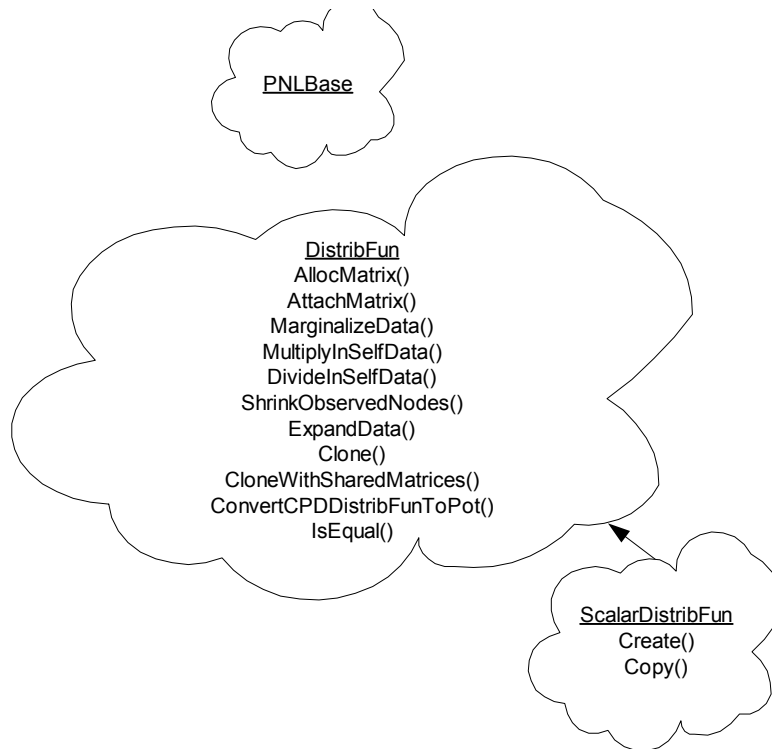
Create

Creates class object.

```
static CScalarDistribFun* CScalarDistribFun::Create( int NodeNumber, const
    CNodeType *const* NodeTypes, int asDense = 1 );
```

Arguments

NodeNumber   Number of nodes.

NodeTypes	Array of node types.
asDense	Flag.

Discussion

This function creates a class object.

Copy

Creates replica of input object.

```
static CScalarDistribFun* CScalarDistribFun::Copy( const CScalarDistribFun
    *pInpDistr );
```

Arguments

pInpDistr   Pointer to the input distribution.

Discussion

This function creates a replica of the input object.

Dump

Dumps content of object.

```
void CScalarDistribFun::Dump();
```

Class CTreeDistribFun
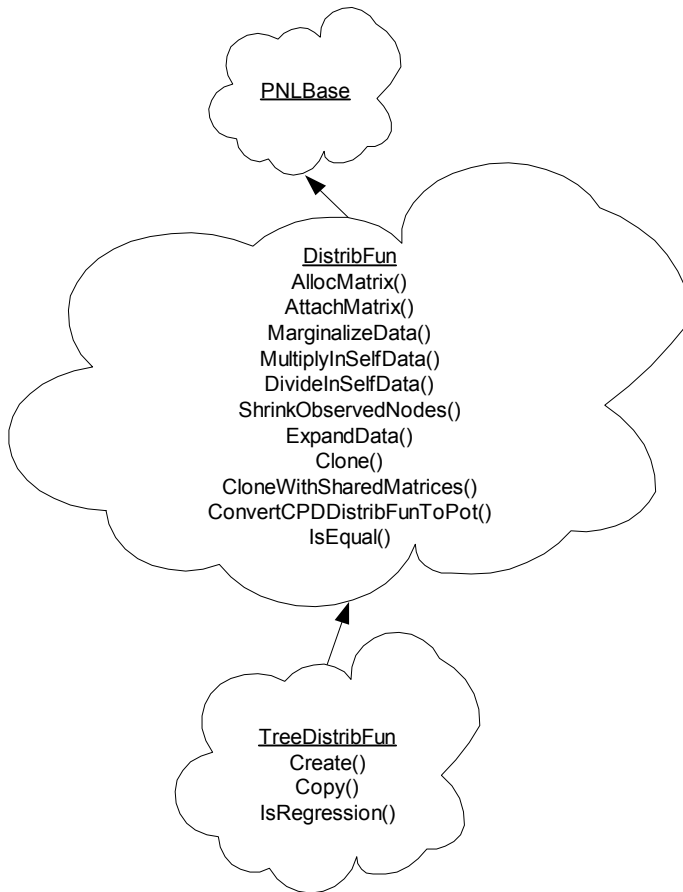
Create

Creates class object.

```
static CTreeDistribFun* TreeDistribFun::Create( int nodeNumber, const
    CNodeType *const* nodeTypes, const SCARTParams* params = 0);
```

Arguments

nodeNumber	Number of nodes in the domain.
nodeTypes	Pointer to the array of *CNodeTypes* nodes in the domain.
params	Parameters for underlying CART. The value NULL sets the parameter to default values.

Discussion

This function creates a class object.

SCARTParams contains the following structure members.

is_cross_val indicates whether to apply cross validation in calculation of the best pruning step for the tree. The *false* value designates test sample estimation. The default value is *true*.

cross_val_folds is the number of cross validation folds equal to the number of additional subtrees built during cross validation. The default value is 5.

learn_sample_par is the number of samples for learning in the test sample estimation. This parameter is set to 0.6 by default.

priors is optional priors in cases when the child node is discrete. This parameter is used to weight classes. The default value is NULL.

Copy

Creates class object by copying input object.

```
static CTreeDistribFun * TreeDistribFun::Copy(  const CTreeDistribFun *
    pInputDistr );
```

Arguments

pInputDistr	Pointer to the input distribution.

Discussion

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

Factors

Class `CFactors` and its child subclasses `CPotential`, `CCPD`, `CTabularPotential`, `CGaussianPotencial`, `CTabularCPD`, and `CGaussianCPD` store graphical model factors related to one or several nodes, that is, a factor domain. The `CPotential`, `CCPD`, and `CTabularFactor` subclasses are abstract. See Figure 3-1 for their hierarchy.

Figure 3-1   Class `CFactors` and child subclasses
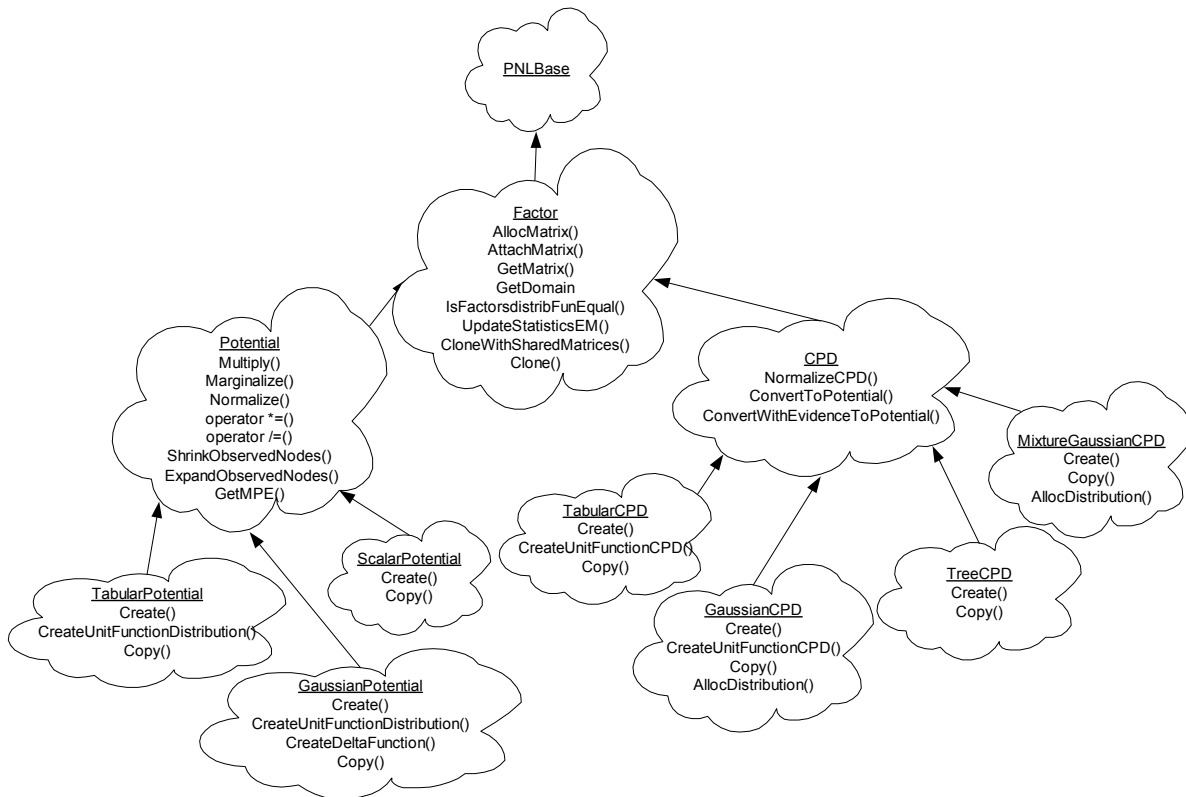
The factor stores joint probability distribution for a `CFactor` object, and conditional distribution for a `CCPD` object. Both types of distribution are implemented with `Class CMultiDMatrix` objects, the number of which depends on whether the distribution is discrete or continuous. All distribution type specific member functions are localized in the internal class `CData`.

Class CFactor
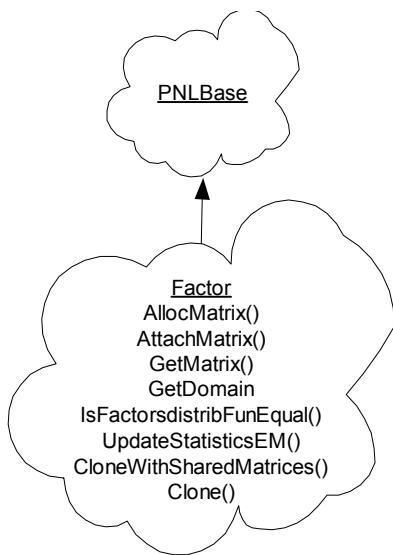
AllocMatrix

Creates matrix and allocates it to factor.

```
void CFactor::AllocMatrix( float *data, EMatrixType mType, int MatrixNum = -1,
    const int *discrParentValuesIndices = NULL );
```

Arguments

data	Array that corresponds to a specific part of distribution.
mType	Type of the matrix that is allocated for the factor.
MatrixNum	Number of a matrix, if several matrices of given type are associated with the factor. Optional argument that may be omitted, if only one matrix is involved.
discrParentValuesIndices	Array of values of discrete parents.

Discussion

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

AttachMatrix

Attaches matrix to factor.

```
void CFactor::AttachMatrix( CMultiDMatrix *matrix, EMatrixType mType, int
    MatrixNum = -1, const int *discrParentValuesIndices = NULL );
```

Arguments

matrix	Pointer to `CMultiDMatrix` object.
mType	Matrix type.
MatrixNum	Number of a matrix, if several matrices of given type are associated with the factor. Optional argument that may be omitted, if only one matrix is involved.
discrParentValuesIndices	Array of values of discrete parents.

Discussion

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

GetFactorType

Returns factor type.

```
inline EFactorType CFactor::GetFactorType() const;
```

Discussion

Factor type may be either *ptFactor* or *ptCPD*.

GetDistributionType

Returns distribution type.

```
inline EDistributionType CFactor::GetDistributionType() const;
```

Discussion

Factor type may be either *dtTabular*, *dtGaussian* or *dtCondGaussian*.

GetDomain

Returns pointer to factor domain and domain size.

```
void CFactor::GetDomain( int *DomainSize, const int **domain) const;
void CFactor::GetDomain( intVector* domain ) const;
```

Arguments

DomainSize   Returned parameter, pointer to the integer that specifies domain size.

domain	Returned parameter. Array of numbers that specify serial numbers of the graphical model nodes associated with the factor domain.

GetDomainSize

Returns size of factor domain.

```
inline int CFactor::GetDomainSize() const;
```

Discussion

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

GetMatrix

Returns pointer to matrix attached to factor.

```
CMatrix<float>* CFactor::GetMatrix( EMatrixType mType, int matrixNum = -1,
    const int *discrParentValuesIndices = NULL ) const;
```

Arguments

mType	Type of the matrix called.
matrixNum	Number of a matrix called among several matrices of a given type. Optional argument that may be omitted, if only one matrix is involved.
discrParentValuesIndices	Array of values of discrete parents.

Discussion

This function returns the pointer to 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*.

intel

operator =

Assigns data from input factor to the object.

```
CFactor& CFactor::operator =( const CFactor& rInputFactor );
```

Arguments

rInputFactor   Reference to the `CFactor` object.

Discussion

This function assigns data from input factor to the object for which it is called, only if both of them are of the same size and type.

IsValid

Checks factor validity.

```
virtual bool CFactor::IsValid( std::string* discription = NULL ) const;
```

Arguments

discription   Error message.

Discussion

This function checks martix validity.The function returns 'true' if matrices are allocated, returns 'false' otherwise.

IsFactorsDistribFunEqual

Compares distributions.

```
int CFactor::IsFactorsDistribFunEqual( const CFactor *pFactor, float eps, int
    withCoeff = 1 ) const;
```

Arguments

pFactor	Pointer to the factor to compare.
eps	Float value of accuracy to compare.
withCoeff	Flag of type of comparison: if it is equal to 0, normalizing constants for Gaussian and Conditional Gaussian distribution should be compared.

Discussion

This function returns 1, if distributions on factors are of the same type and size, with the same floating point matrices representing distributions.

TieDistribFun

Sets input factor distribution for object.

```
void CFactor::TieDistribFun( CFactor *pFactor );
```

Arguments

pFactor	Pointer to the CFactor object.

Discussion

This function sets the distribution for an object only if both factors are of the same form, and throws an exception otherwise.

IsDistributionSpecific

Checks whether distribution is specific.

```
int CFactor::IsDistributionSpecific() const;
```

Discussion

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

- 0 - if distribution is full (Tabular or Gaussian or Conditional Gaussian, non-delta, non-uniform, non-mixed; may be invalid, call <u>IsValid</u> function to check the status).
- 1 - the distribution is uniform: it has no matrices attached to it. It has the flag, which shows that the distribution is uniform.
- 2 - the distribution is delta function: it has only mean matrix (to check if the form is valid or not, use <u>IsValid</u> function).
- 3 - mixed distribution, that is, product of some distribution multiplied by delta function in some of distribution dimensions. It is an intermediate distribution presentation.

GenerateSample

Draws random sample from factor using
information from current evidence.

```
void CFactor::GenerateSample(CEvidence* evidences,int maximize = 0)const = 0;
```

Arguments

evidences	Input-Output parameter.Pointer to the current evidence.
maximize	Flag of maximization.

Discussion

Generates a sample from the factor.

CopyWithNewDomain

Copies the input object and creates new factor
with new domain and model domain.

```
static CFactor* CFactor::CopyWithNewDomain( const CFactor *factor, intVector
    &domain, CModelDomain *pModelDomain, const intVector& obsIndices =
    intVector() );
```

Arguments

factor	Pointer to the `CFactor` object.
domain	Node numbers in the domain.
pModelDomain	Pointer to the new model domain.
obsIndices	Indices of the observed nodes.

Discussion

Copies the factor and changes the domain for which it was created.

Clone

Creates replica of object.

```
virtual CFactor* CFactor::Clone() const = 0;
```

CloneWithSharedMartices

Creates replica of factor.

```
virtual CFactor* CFactor::CloneWithSharedMatrices() const = 0;
```

Discussion

This function creates a replica of the factor so that the newly created factor shares its matrices with the source factor.

CreateAllNecessaryMatrices

Creates matrices necessary to make factor valid.

```
virtual void CFactor::CreateAllNecessaryMatrices(int typeOfMatrices = 1);
```

Arguments

typeOfMatrices Flag of the type of matrix generation.
If it equals to 1 all matrices are random.

Discussion

This function creates all matrices which are necessary to make a factor valid. Covariance matrix for the Gaussian distribution is the matrix unit.

GetNumInHeap

Returns factor number in factor heap.

```
int CFactor::GetNumInHeap() const;
```

Discussion

This function is applied for `ModelDomain`.

ChangeOwnerToGraphicalModel

Releases model domain from factor.

```
void CFactor::ChangeOwnerToGraphicalModel() const;
```

IsOwnedByModelDomain

Checks if factor is owner of model domain.

```
bool CFactor::IsOwnedByModelDomain() const;
```

GetModelDomain

Returns pointer to model domain.

```
inline CModelDomain* CFactor::GetModelDomain() const;
```

GetArgType

Returns pointers to node types of domain nodes.

```
const pConstNodeTypeVector CFactor::*GetArgType() const;
```

ConvertToSparse

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

```
void CFactor::ConvertToSparse();
```

Discussion

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

ConvertToDense

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

```
void CFactor::ConvertToDense();
```

Discussion

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

IsSparse

Checks if distribution matrices are sparse.

```
virtual int CFactor::IsSparse() const = 0;
```

Discussion

This function checks if the matrices of the factor distribution are sparse.

IsDense

Checks if distribution matrices are dense.

```
virtual int CFactor::IsDense() const = 0;
```

Discussion

This function checks if the matrices of the distribution are dense.

GetObsPositions

Returns observed positions of domain.

```
inline void CFactor::GetObsPositions( intVector* obsPosOut ) const;
```

Arguments

obsPosOut   Observed positions of the domain.

GetDistribFun

Returns pointer to distribution function.

```
inline CDistribFun* CFactor::GetDistribFun() const;
```

SetDistribFun

Sets distribution function into factor.

```
void CFactor::SetDistribFun( const CDistribFun* data );
```

Arguments

data   Pointer to CDistribFun object.

Discussion

This function releases the old distribution function and creates a new factor
distribution by copying the input data.

MakeUnitFunction

*Transfers distribution function into unit function
distribution.*

```
void CFactor::MakeUnitFunction();
```

ConvertStatisticToPot

*Creates potential on the basis of statistical data
of distribution function.*

```
virtual CPotential* CFactor::ConvertStatisticToPot( int numOfSamples )
    const = 0;
```

Arguments

numOfSamples   Number of samples.

UpdateStatisticsEM

Updates statistical data.

```
virtual void CFactor::StatisticalDataEM( const CPotential* infData, const
    CEvidence *pEvidence = NULL ) = 0;
```

Arguments

infData	Inference result.
pEvidence	Pointer to an `Evidence` object.

Discussion

This function estimates factors and updates statistical data.

UpdateStatisticsML

Gathers statistical data.

```
virtual void CFactor::StatisticalDataML( const CEvidence* const* pEvidences,
    int evidenceNumber );
virtual void CFactor::StatisticalDataML( const pConstEvidenceVector&
    pEvidences );
```

Arguments

pEvidences	Array of evidences.
EvidenceNumber	Number of evidences.

Discussion

This function estimates factors and updates statistical data.

SetStatistics

Sets statistical data.

```
virtual void CFactor::SetStatistics( const CMatrix<float>* pMat,
    EStatisticalMatrix matrixType, const int* parentsComb = NULL ) = 0;
```

Arguments

pMat	Input parameter. Matrix with statistical data.
matrixType	Type of matrix.
parentsComb	Combination of discrete parents.

Discussion

This function sets statistical data for learning.

ProcessingStatisticalData

Updates factor distribution function after gathering statistical data.

```
virtual float CFactor::ProcessingStatisticalData( float numEvidences ) = 0;
```

Arguments

numEvidences	Number of evidences.

Discussion

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

GetLogLik

Returns likelihood of input data.

```
virtual float CFactor::GetLogLik( const CEvidence* pEv, const CPotential*
  pShrInfRes = NULL ) const = 0;
```

Arguments

`pEv`	Evidence.
`pShrInfRes`	Inference result. This parameter is needed if the domain contains unobserved nodes.

Discussion

This function returns the logarithm of likelihood.

AreThereAnyObsPositions

Checks if factor has observed nodes.

```
inline int CFactor::AreThereAnyObsPositions() const;
```

Class CCPD

```
               ╭─────────────╮
               │   PNLBase   │
               ╰──────┬──────╯
                      ▲
                      │
        ╭─────────────────────────────╮
        │            Factor            │
        │         AllocMatrix()        │
        │        AttachMatrix()        │
        │          GetMatrix()         │
        │          GetDomain          │
        │   IsFactorsdistribFunEqual() │
        │      UpdateStatisticsEM()    │
        │    CloneWithSharedMatrices() │
        │           Clone()            │◀──────╮
        ╰─────────────────────────────╯       │
                              ╭───────────────────────────────╮
                              │             CPD               │
                              │        NormalizeCPD()         │
                              │      ConvertToPotential()     │
                              │  ConvertWithEvidenceToPotential() │
                              ╰───────────────────────────────╯
```
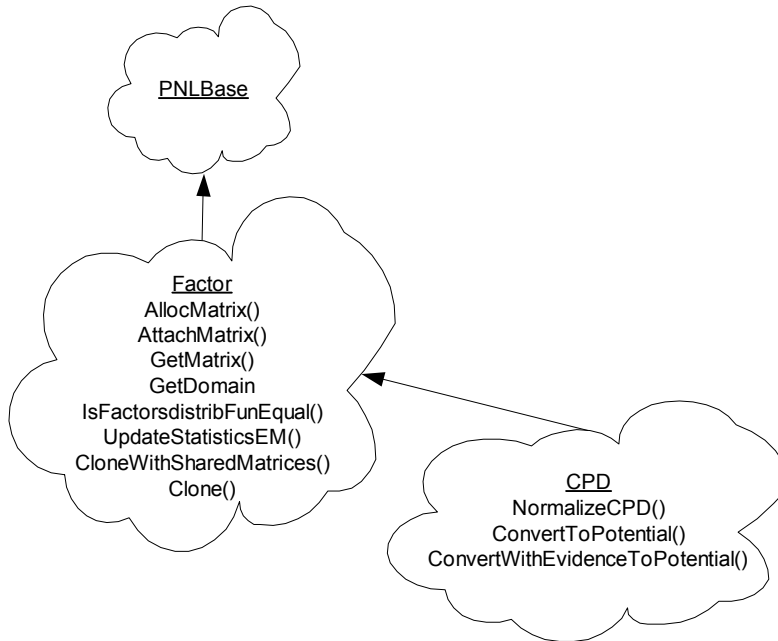
ConvertToPotential

*Converts class object to potential and returns
pointer to that potential.*

```
CPotential * CCPD::ConvertToPotential();
```

Discussion

This function converts a `CCPD` object to a `CPotential` object and returns a new
`CPotential` object.

ConvertWithEvidenceToPotential

Converts CPD to potential using evidence.

```
CPotential* CCPD::ConvertWithEvidenceToPotential( const CEvidence* pEv, int
    flagSumOnMixtureNode = 1) const;
```

Arguments

pEv   Evidence.

flagSumOnMixtureNode Flag of mixture node summation.

Discussion

This function converts CPD to Potential using evidence. This function can change
distribution type of CPD, unlike combination of ConvertToPotential and
ShrinkObservedNodes.

NormalizeCPD

Normalizes CPD.

```
virtual void CCPD::NormalizeCPD() = 0;
```

Class CTabularCPD
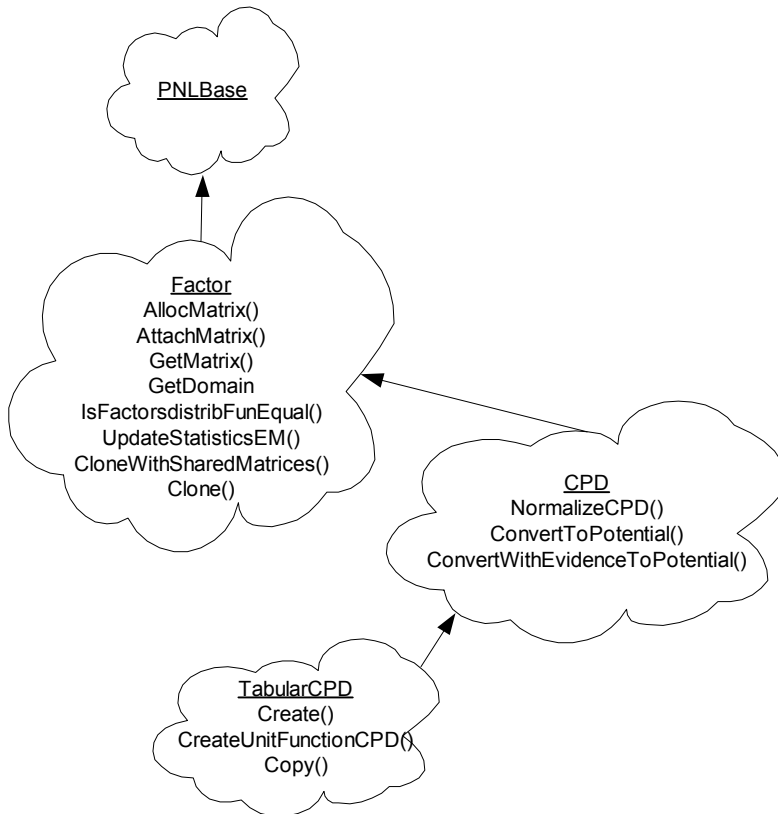
Create

Returns class object.

```
static CTabularCPD* CTabularCPD::Create( const intVector& domain,
    CModelDomain* pMD, const floatVector& data = floatVector() );

static CTabularCPD* CTabularCPD::Create( const int* domain, int nNodes,
    CModelDomain* pMD, const float* data = NULL);
```

Arguments

domain	Array of numbers of domain nodes.
nNodes	Number of nodes in domain.
data	Array of data.
pMD	Pointer to the model domain.

Copy

Creates new Tabular CPD as copy of input object.

```
static CTabularCPD* CTabularCPD::Copy( const CTabularCPD* pTabCPD );
```

Arguments

pTabCPD	Pointer to the CTabularCPD object.

CreateUnitFunctionCPD

Creates CPD that becomes Unit function after conversion to potential.

```
static CTabularCPD* CTabularCPD::CreateUnitFunctionCPD( const intVector&
    domain, CModelDomain* pMD );
static CTabularCPD* CTabularCPD::CreateUnitFunctionCPD( const int* domain,
    CModelDomain* pMD );
```

Arguments

domain	Array of numbers of domain nodes.
pMD	Pointer to the modail domain.

int͏el

Discussion

This function creates CPD as a unit function by converting the function to the potential.

Class CGaussianCPD
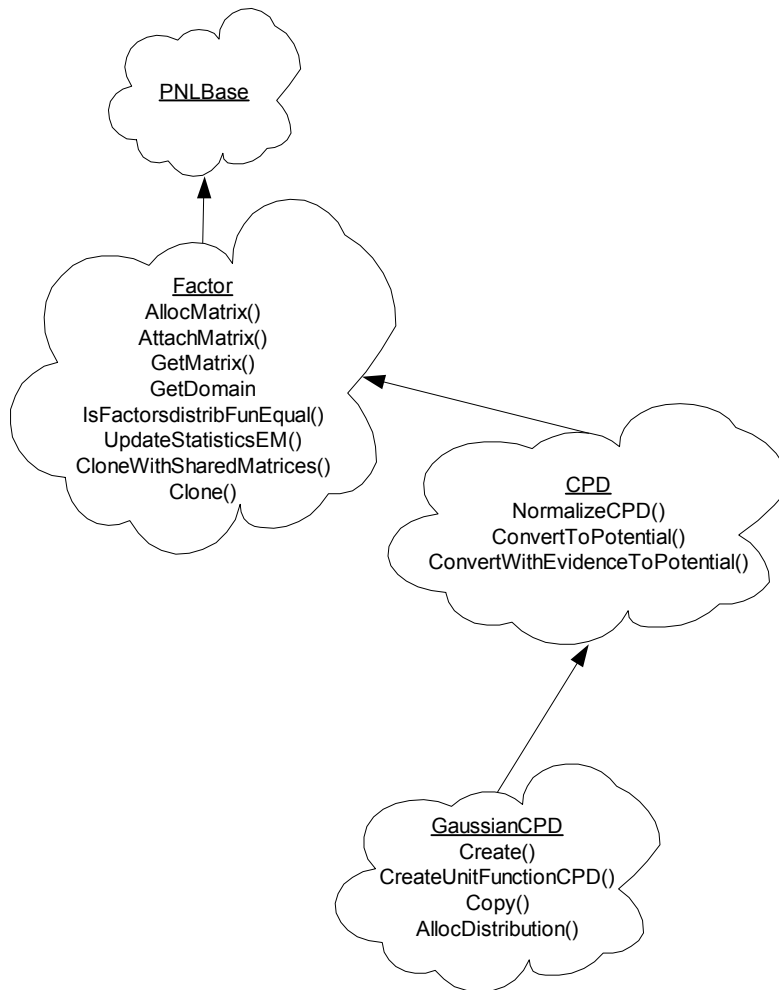
Create

Returns class object.

```
static CGaussianCPD* CGaussianCPD::Create( const intVector& domain,
    CModelDomain* pMD );
static CGaussianCPD* CGaussianCPD::Create( const int* domain, int nNodes,
    CModelDomain* pMD );
```

Arguments

domain	Array of numbers of domain nodes.
nNodes	Number of nodes in domain.
pMD	Pointer to the modail domain.

CreateUnitFunctionCPD

Creates CPD that becomes Unit function after
conversion to potential.

```
static CGaussianCPD* CGaussianCPD::CreateUnitFunctionCPD( const intVector&
    domain, CModelDomain* pMD );
static CGaussianCPD* CGaussianCPD::CreateUnitFunctionCPD( const int* domain,
    CModelDomain* pMD );
```

Arguments

domain	Array of numbers of domain nodes.
pMD	Pointer to the modail domain.

Discussion

This function creates CPD as a unit function by converting the function to the potential.

Copy

Creates new Gaussian CPD by copying input
CPD.

```
static CGaussianCPD* CGaussianCPD::Copy( const CGaussianCPD* pGaussCPD );
```

Arguments

pGaussCPD	Pointer to CGaussianCPD object.

AllocDistribution

Allocates Gaussian distribution on Gaussian
child node.

```
void CGaussianCPD::AllocDistribution( const float* pMean, const float* pCov,
    float normCoeff, float* const* pWeights, const int* parentCombination );
void CGaussianCPD::AllocDistribution( const floatVector& meanIn, const
    floatVector& covIn,
float normCoeff,const floatVecVector& weightsIn,const intVector&
    parentCombination = intVector() );
```

Arguments

pMean	Data for the mean matrix.
pCov	Data for the covariance matrix which is inputted rowwise.
normCoeff	Float value of normalization constant.
pWeights	Data for weight matrices.
parentCombination	Array of values of discrete parents.

intel.

Discussion

This function allocates a Gaussian distribution on a Gaussian child node with Gaussian parents and the given discrete parent combination.

SetCoefficient

Sets normalization constant to Gaussian CPD object.

```
void CGaussianCPD::SetCoefficient( float coeff, const int* parentCombination =
    NULL );
```

Arguments

`coeff`	Float value of normalization constant.
`parentCombination`	Pointer to the array of values of discrete parents.

GetCoefficient

Gets value of normalization constant.

```
float CGaussianCPD::GetCoefficient( const int* parentCombination = NULL  );
```

Arguments

`parentCombination`	Pointer to the array of values of discrete parents.

Class CMixtureGaussianCPD

Create

Returns class object.

```
static CMixtureGaussianCPD* CMixtureGaussianCPD::Create( const intVector&
    domain, CModelDomain* pMD, const floatVector& sumCoeff );
```

```
static CMixtureGaussianCPD* CMixtureGaussianCPD::Create( const int* domain,
    int nNodes, CModelDomain* pMD, const float* sumCoeff );
```

Arguments

domain	Array of numbers of domain nodes.
nNodes	Number of nodes in the domain.
pMD	Pointer to the modail domain.
sumCoeff	Mixture coefficient.

Copy

*Creates new mixture Gaussian CPD by coying
input CPD.*

```
static CMixtureGaussianCPD* CMixtureGaussianCPD::Copy( const CGaussianCPD*
    pGaussCPD );
```

Arguments

pGaussCPD	Pointer to CMixtureGaussianCPD object.

AllocDistributionVec

Allocates mixture Gaussian distribution.

```
void CMixtureGaussianCPD::AllocDistributionVec( const floatVector& meanIn,
    const floatVector& covIn,float normCoeff,const floatVecVector&
    weightsIn,const intVector& parentCombination );
```

Arguments

pMean	Data for the mean matrix.
pCov	Data for the covariance matrix which is inputted rowwise.
normCoeff	Float value of normalization constant.
pWeights	Data for weight matrices.
parentCombination	Array of values of discrete parents.

Discussion

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

SetCoefficient

Sets normalization constant to mixture Gaussian CPD.

```
void CMixtureGaussianCPD::SetCoefficient( float coeff, const int*
    parentCombination );
```

Arguments

coeff	Float value of the normalization constant.
parentCombination	Pointer to the array of values of discrete parents.

GetCoefficient

Gets value of normalization constant.

```
float CMixtureGaussianCPD::GetCoefficient( const int* parentCombination );
```

Arguments

parentCombination   Pointer to the array of values of discrete parents.

SetCoefficientVec

*Sets normalization constant to mixture Gaussian
CPD.*

```
void CMixtureGaussianCPD::SetCoefficientVec( float coeff, const intVector&
    parentCombination );
```

Arguments

coeff   Float value of the normalization constant.

parentCombination   Pointer to the array of values of discrete parents.

GetCoefficientVec

Gets value of normalization constant.

```
float CMixtureGaussianCPD::GetCoefficientVec( const intVector&
    parentCombination );
```

Arguments

parentCombination   Pointer to the array of values of discrete parents.

GetProbabilities

Returns vector of probabilities of mixture node.

```
inline void CMixtureGaussianCPD::GetProbabilities( loatVector* probabilities )
    const;
```

Arguments

probabilities   Returned parameter.Vector of probabilities.

Discussion

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

Class CTreeCPD

Create

Returns class object.

```
static CTreeCPD* CTreeCPD::Create( const intVector& domain, CModelDomain* pMD,
    const floatVector& data = floatVector() );
static CTreeCPD* CTreeCPD::Create( const int* domain, int nNodes,
    CModelDomain* pMD, const float* data = NULL);
```

Arguments

domain	Array of numbers of domain nodes.
nNodes	Number of nodes in domain.
data	Array of data.
pMD	Pointer to the modail domain.

Copy

Creates new Tree CPD as copy of input object.

```
static CTreeCPD* CTreeCPD::Copy( const CTreeCPD* pCPD );
```

Arguments

pCPD   Pointer to the CTreeCPD object.

Class CPotential

Class CPotential implements basic operations with factors. To perform an operation with a Class CCPD object, first ConvertToPotential function should be used and then appropriate member functions should be called for the generated potential.

Operations that can be performed with a CPotential object are as follows:

— multiplication and division of CPotential objects
— normalization
— marginalization
— factor shrinking in cases when factor nodes are observed
— factor expansion to the original size.

Multiply

Multiplies two factors and returns product.

```
CPotential* CPotential::Multiply( const CPotential *pOtherFactor ) const;
```

Arguments

pOtherFactor   Pointer to the multiplier factor.

Discussion

This function returns pointer to a new `CPotential` object, which is the product of two `CPotential` objects.

operator *=

Provides multiplication and puts result to potential.

```
CPotential& CPotential::operator *=( const CPotential &rSmallPotential );
```

Arguments

rSmallPotential   Reference to the right-hand-side multiplier of `CPotential` type.

Discussion

This function changes the object, for which it was called, by changing its data matrices.

operator /=

Provides division and puts result to potential.

```
CPotential& CPotential::operator /=( const CPotential &rSmallPotential );
```

Arguments

rSmallPotential   Reference to the denominator of `CPotential` type.

Discussion

This function changes the object, for which it was called, by changing its data matrices.

GetNormalized

Creates new normalized potential.

```
CPotential* CPotential::GetNormalized() const;
```

Discussion

This function returns pointer to a new normalized CPotential object with domain that is the same as domain of the object, for which this function is called.

Normalize

Normalizes potential for which it was called.

```
void CPotential::Normalize();
```

Marginalize

Marginalizes object.

```
CPotential * CPotential:: Marginalize( const int *pSmallDom, int domSize, int
    maximize = 0 ) const ;
CPotential * CPotential:: Marginalize( const intVector& pSmallDom, int
    maximize = 0 ) const ;
```

Arguments

pSmallDom	Array of numbers of nodes that should constitute a domain of the returned marginalized object.
domSize	Size of the returned object domain.
maximize	Flag of the marginalization type.

For discrete variables:

- 0 stands for simple summation;
- 1 stands for finding maximum value.

For continuous variables:

- both are integration operations.

Discussion

This function returns pointer to a new `CPotential` object that is generated from the source object by either adding or by integrating of the source object referring to the nodes that are not part of the returned object domain. The returned object domain should be a subset of the source domain.

ShrinkObservedNodes

Creates new factor without dimensions corresponding to observed nodes.

```
CPotential * CPotential::ShrinkObservedNodes( const CEvidence* pEv ) const;
```

Arguments

pEv	Pointer to the given evidence.

Discussion

This function creates a new factor with the same domain as the domain of the source factor but with modified observed nodes. Joint probability distribution also changes reflecting the modified values of the observed nodes.

ExpandObservedNodes

Expands dimensions corresponding to observed nodes.

```
CPotential * CPotential::ExpandObservedNodes ( const CEvidence* pEv. int
    updateInCanonical = 1 ) const;
```

Arguments

pEv	Pointer to the given evidence.
updateInCanonical	Flag of distribution form.

Discussion

This function expands the probability distribution by filling empty spaces with zeros.

GetMultiplyedDelta

Returns delta functions by which Gaussian distribution was multiplied.

```
int CPotential::GetMultiplyedDelta( const int** positions, const float**
    values, const int** offsets ) const;
int CPotential::GetMultiplyedDelta( intVector* positions, floatVector* values,
    intVector* offsets ) const;
```

Arguments

positions	Array of distribution positions which are multiplied by Delta distribution.
values	Array of values.
offsets	Returned parameter. Offsets of values.

Discussion

This function returns delta functions by which Gaussian distribution, for which it was called, was multiplied. Multiplication by delta distributions is an internal part of the library implementation of inference. Inference engines should return result in non-specified form, that is, without multiplied Delta distribution. The function is planned to be deleted from the final release of the library.

Divide

Divides factor by another and returns result.

```
CPotential * CPotential::Divide( const CPotential *pOtherFactor ) const;
```

Arguments

pOtherFactor   Pointer to the divisor factor.

Discussion

This function divides the factor for which it is called by the argument and returns the result, that is, pointer to a newly generated factor.

Dump

Dumps object content.

```
void CPotential::Dump()const;
```

Discussion

This function dumps the `CPotential` content, that is, domain, factor type, and distribution matrix to `std::cout`.

MarginalizeInPlace

Marginalizes input object.

```
void CPotential::MarginalizeInPlace( const CPotential* pOldPot, const int*
    corrPositions = NULL,  int maximize = 0);
```

Arguments

pOldPot	Pointer to the old potential.
corrPositions	Positions for marginalisation.
maximize	Flag of marginalisation with maximization.

Discussion

This function marginalizes an input object and stores the result in the given object.

GetMPE

Returns maximum probability explanation.

```
CEvidence* CPotential::GetMPE() const;
```

Class CTabularPotential

Create

Returns class object.

```
static CTabularPotential* CTabularPotential::Create( const intVector& domain,
    CModelDomain* pMD, const float* data, const intVector& obsIndices =
    intVector() );
```

```
static CTabularPotential* CTabularPotential::Create( const int* domain, int
    nNodes, CModelDomain* pMD, const float* data = NULL, const intVector&
    obsIndices = intVector() );
```

Arguments

domain	Array of numbers of domain nodes.
nNodes	Number of nodes in domain.
data	Array of data.
pMD	Pointer to the modail domain.
obsIndices	Indices of observed nodes of the domain.

Copy

Creates new tabular potential as copy of input object.

```
static CTabularPotential* CTabularPotential::Copy(const CTabularCPD* pTabCPD);
```

Arguments

pTabCPD	Pointer to the CTabularCPD object.

CreateUnitFunctionDistribution

Creates potential in form of unit function.

```
static CTabularPotential* CTabularPotential::CreateUnitFunctionCPD( const
    intVector& domain, CModelDomain* pMD, int asDense = 1, const intVector&
    obsIndices = intVector() );
static CTabularPotential* CTabularPotential::CreateUnitFunctionCPD( const int*
    domain, int nNodes, CModelDomain* pMD, int asDense = 1, const intVector&
    obsIndices = intVector() );
```

Arguments

domain	Array of numbers of domain nodes.
pMD	Pointer to the model domain.
asDense	Flag of matrix type.
obsIndices	Numbers of observed positions.
nNodes	Number of nodes in the domain.

Discussion

This function creates a potential as a unit function.

Class CGaussianPotential

Create

Creates class object.

```
static CGaussianPotential*  CGaussianPotential::Create( const intVector&
    domain,CModelDomain* pMD, int inMoment = -1, const floatVector& Vec =
    floatVector(),const floatVector&  Mat = floatVector(),float normCoeff =
    0.0f, const intVector& obsIndices = intVector() );
```

```
static CGaussianPotential* CGaussianPotential::Create( const int *domain, int
    nNodes, CModelDomain* pMD, int inMoment = -1, float const* pVec = NULL,
    float const* pMat = NULL, float normCoeff = 0.0f, const intVector&
    obsIndices = intVector());
```

Arguments

domain	Numbers of domain nodes.
nNodes	Number of nodes in the domain.
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,

pVec	Array for Mean value matrix data.
pMat	array for Covariance matrix data.
normCoeff	Value of normalization constant in the moment form.

if *inMoment* = 0,

pVec	Array for *H* matrix data.
pMat	array for *K* matrix data.
normCoeff	Value of normalization constant in the canonical form.
obsIndices	Indices of observed nodes of the domain.
pMd	Pointer to the model domain.

Copy

Creates new class object as copy of input object.

```
static CGaussianPotential* CGaussianPotential::Copy( const CGaussianPotential
    *pGauPot );
```

Arguments

pGauPot	Pointer to the `CGaussianPotential` object.

CreateDeltaFunction

Creates Delta function as `CGaussianPotential`.

```
static CGaussianPotential* CGaussianPotential::CreateDeltaFunction( const int
    *domain, int nNodes, CModelDomain* pMD, const float *mean, int isInMoment =
    1, const intVector& obsIndices = intVector() );

static CGaussianPotential* CGaussianPotential::CreateDeltaFunction( const
    intVector& domain, CModelDomain* pMD, const floatVector& mean, int
    isInMoment = 1, const intVector& obsIndices = intVector() );
```

Arguments

domain	Numbers of domain nodes.
nNodes	Number of nodes in the domain.
mean	Pointer to float values of the mean matrix.
isInMoment	Flag of the form of the resulting Potential:
	1 - moment form (mean, covariance matrices, and normalization constant)
	0 - canonical form (canonical matrices g, H, K).
pMD	Pointer to the model domain.
obsIndices	Indices of observed nodes of the domain.

CreateUnitFunctionDistribution

Creates CGaussianPotential *object to represent unit function distribution.*

```
static CGaussianPotential* CGaussianPotential::CreateUnitFunctionDistribution
    ( const int *domain, int nNodes,CModelDomain* pMD,int isInCanonical = 1,
    const intVector& obsIndices = intVector());

static CGaussianPotential* CGaussianPotential::CreateUnitFunctionDistribution
    ( const CNodeType*const *nodeTypes, const int *domain, int numOfNds, int
    isInCanonical = 1, const intVector& obsIndices = intVector() );
```

Arguments

domain	Numbers of domain nodes.
nNodes	Number of nodes in the domain.
isInCanonical	Flag of the desired form of Unit function:
	1 - canonical form
	0 - moment form.
obsIndices	Indices of the observed domain nodes.
pMd	Model domain.

Discussion

Creates a class object in the form of a unit function distribution.

SetCoefficient

Sets normalization constant to CGaussianPotential *object.*

```
void CGaussianPotential::SetCoefficient( float coeff, int isForCanonical );
```

intel

Arguments

coeff	Float value of normalization constant.
isForCanonical	Flag of distribution type to set the coefficient:
	1 - canonical form
	0 - moment form.

GetCoefficient

Gets value of normalization constant.

```
float CGaussianPotential::GetCoefficient( int isforCanonical );
```

Arguments

isforCanonical	Flag of distribution type to get the coefficient:
	1 - canonical form
	0 - moment form.

Class CFactors

Class CFactors represents a complete set of factors for a graphical model. The class is intended for storing an array of pointers to Class CFactor objects. It is separated from the model to enable user to create factors separately from the model and attach all of them to the existing model in a single move.

Create

Creates CFactors *class object.*

```
friend CFactors* CFactors::Create( int NumOfFactors );
```

Arguments

numOfFactors   Maximal number of factors in the factor array; equal to the number of nodes for *BNet* and to the number of cliques for all the Markov models.

Public Member Functions

GetNumberOfFactors

Returns current number of factors in factor array.

```
inline int CFactors::GetNumberOfFactors() const;
```

GetFactor

Returns pointer to factor.

```
inline CFactor* CFactors::GetFactor( int factorNum ) const;
```

Arguments

factorNum   Factor index in the array of factors.

Discussion

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

AddFactor

Adds new factor to graphical model.

```
inline int CFactors::AddFactor( CFactor *factor );
```

Arguments

factor   Pointer to the factor to be set in the factor array.

Discussion

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

ShrinkObsNdsForAllFactors

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

```
void CFactors::ShrinkObsNdsForAllFactors( const CEvidence *pEvidence );
```

Arguments

pEvidence   Evidence.

Discussion

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

Class CMatrix
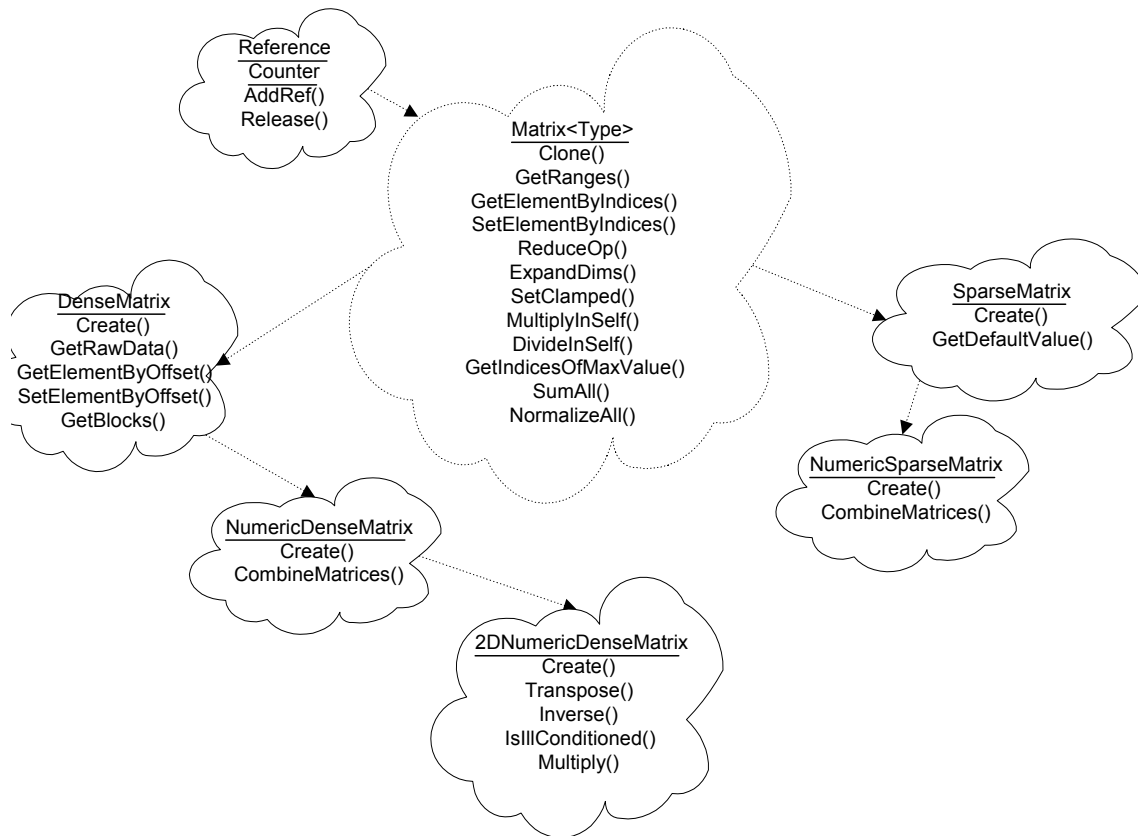
Template class `CMatrix<Type>` is an abstract class that declares basic operations with a multidimensional matrix of any type. The operations are used with various types of probability distributions. This class has two main derived subclasses: `CDenseMatrix` and `CSparseMatrix`.

Class `CMatrixIterator` is convenient for operations with matrix elements.

CreateEmptyMatrix

Creates empty multidimensional matrix with data
of default value.

```
virtual CMatrix<Type>* CMatrix::CreateEmptyMatrix( int dim, const int *range,
    int Clamp, Type defaultVal = Type(0) )const = 0;
```

Arguments

dim	Number of matrix dimensions.
range	Array of lengths of matrix dimensions.
Clamp	Status flag: • 1 means no change is allowed • 0 means change is allowed.
defaultVal	Default value to be set for the matrix.

Discussion

This function creates a multidimensional matrix with the default value data and returns the pointer to the matrix.

SetDataFromOtherMatrix

Assigns data from input matrix to object.

```
virtual void  CMatrix::SetDataFromOtherMatrix( const CMatrix<Type>* matInput )
    = 0;
```

Arguments

matInput	Pointer to the CMatrix object to be copied.

Discussion

This function assigns data from the input matrix to the matrix, for which it is called, only if both the matrices are of the same size.

Clone

Returns pointer to new matrix.

```
virtual CMatrix<Type>*  CMatrix::Clone() const = 0;
```

GetNumberDims

Returns number of matrix dimensions.

```
virtual inline int  CMatrix::GetNumberDims() const = 0;
```

GetRanges

*Returns pointer to array of dimensions and
number of matrix dimensions.*

```
virtual inline void  CMatrix::GetRanges( int *numOfDimsOut, const int
    **rangesOut ) const = 0;
```

Arguments

numOfDimsOut　　Returned parameter. Number of matrix dimensions.

rangesOut　　Returned parameter. Pointer to the array of the matrix dimensions.

Discussion

This function returns the number of matrix dimensions and the pointer to the array of the dimensions.

GetMatrixClass

Returns matrix class name.

```
virtual EMatrixClass  CMatrix::GetMatrixClass() const;
```

Discussion

This function returns the class name of the martix. A matrix may belong to one of the following classes: `mcBase, mcSparse, mcDense, mcNumericDense, mcNumericSparse, mc2DNumericDense, mc2DNumericSparse`.

ConvertToDense

Creates new matrix by conversion of given matrix
to dense matrix.

```
virtual CDenseMatrix<Type>*  CMatrix::ConvertToDense() const = 0;
```

Discussion

This function creates a new matrix by conversion of the given matrix to a dense matrix.The new matrix contains the same data as the matrix corresponding to the function but has another another form. When called for a dense matrix the function creates its replica.

ConvertToSparse

Creates new matrix by conversion of given matrix
to sparse matrix.

```
virtual CSparseMatrix<Type>*  CMatrix::ConvertToSparse() const = 0;
```

Discussion

This function creates a new matrix by conversion of the given matrix into a sparce matrix. The new matrix contains the same data as the matrix corresponding to the function but has another form. When called for a sparce matrix the function creates its replica.

GetElementByIndices

Returns value of matrix element by
multidimensional indices.

```
virtual inline Type CMatrix::GetElementByIndices( const int *multidimindices )
    const = 0;
```

Arguments

multidimindices   Pointer to the array of multidimensional indices of the addressed matrix element. The array length is equal to the number of matrix dimensions.

Discussion

This function returns the value of a multidimensional matrix element by multidimensional indices.

SetElementByIndices

Sets new value for matrix element by
multidimensional indices.

```
virtual inline void  CMatrix::SetElementByIndices( Type value, const int
    *multidimindices) = 0;
```

Arguments

value   Value to be set.

multidimindices   Pointer to the array of multidimensional indices of the
                     addressed matrix element. The array length is equal to the
                     number of matrix dimensions.

Discussion

This function sets a new value for a matrix element by multidimentional matrices.

ReduceOp

Collapses dimensions of multidimensional
matrix.

```
virtual CMatrix<Type>*  CMatrix::ReduceOp( const int *pDimsOfInterest, int
    numDimsOfInterest, int action = 2, const int *p0observedValues = NULL,
    CMatrix< Type > *output = NULL, EAccumType accumType = PNL_ACCUM_TYPE_STORE
    ) const = 0;
```

Arguments

pDimsOfInterest   Pointer to the array of dimensions to be preserved in the new
                     matrix.

NumDimsOfKeep   Number of dimensions to be preserved.

---

action	Type of matrix reduction: *action* $= 0$ - sum of all the dimensions except *pDimsOfInterest* *action* $= 1$ - selection of the maximum of the dimensions *action* $= 2$ - selection of the given value of a node.
pObservedValues	Pointer to an array of pointers to the values.
output	Pointer to the matrix to which the result of `ReduceOp` is to be passed.
accumType	Value of `EAccumType` class.

Discussion

This function collapses dimensions of a multidimensional matrix.

This function is used by `Marginalize` and `ShrinkObservedNodes` functions. In `Marginalize` function the action flag is either 0 or 1 and no pointers to the values are required. In `ShrinkObservedNodes` function the action flag is equal to 2 and the pointer *pObsValues* is required.

`EAccumType` provides the following values: `PNL_ACCUM_TYPE_STORE`, `PNL_ACCUM_TYPE_ACCUMULATE`, `PNL_ACCUM_TYPE_SUM`, `PNL_ACCUM_TYPE_ADD`, `PNL_ACCUM_TYPE_MAX`, `PNL_ACCUM_TYPE_MUL`.

The default value of `PNL_ACCUM_TYPE_STORE` is valid in all cases, which means that the previous content of the output matrix, if any, is ignored. If the output matrix was not explicitly given, `PNL_ACCUM_TYPE_STORE` is the only allowed value for *accumType*. If output matrix was given (`output != NULL`) the list of allowed values for *accumType* depends on the *action.*

If *action* = 0, *accumType* assumes one of the following values:
`PNL_ACCUM_TYPE_STORE`, `PNL_ACCUM_TYPE_ACCUMULATE`, `PNL_ACCUM_TYPE_SUM`, `PNL_ACCUM_TYPE_ADD`. `PNL_ACCUM_TYPE_STORE` is the default value. The other values stand for a pair-wise summation of the previous content of the output matrix and the content of the marginalized objects.
If *action* = 1 *accumType* assumes one of the following values:
`PNL_ACCUM_TYPE_STORE`, `PNL_ACCUM_TYPE_ACCUMULATE`, `PNL_ACCUM_TYPE_MAX`. `PNL_ACCUM_TYPE_STORE` is the default value. The other values stand for a pair-wise maximum of previous content of output matrix and the content of the marginalized

objects.

If action $= 2$ `accumType` assumes one of the following values:
`PNL_ACCUM_TYPE_STORE`, `PNL_ACCUM_TYPE_SUM`, `PNL_ACCUM_TYPE_ADD`,
`PNL_ACCUM_TYPE_MAX`, `PNL_ACCUM_TYPE_MUL`. `PNL_ACCUM_TYPE_STORE` is the
default value while `PNL_ACCUM_TYPE_SUM` and `PNL_ACCUM_TYPE_ADD` stand for a
pair-wise summation of previous content of output matrix and the content of the
shrinkage objects. `PNL_ACCUM_TYPE_MAX` stands for a pair-wise maximum of the
previous content of output matrix and the content of shrinked objects.
`PNL_ACCUM_TYPE_MUL` stands for a pair-wise multiplication of the previous content of
the output matrix and the content of shrinked objects.

ExpandDims

Creates new matrix by expanding specified
dimensions.

```
virtual CMatrix<Type> CMatrix::*ExpandDims(const int *dimsToExtend,
    const int *keepPosOfDims,const int *sizesOfExpandDims,int numDimsToExpand)
    const = 0;
```

Arguments

dimsToExtend	Pointer to an array of dimensions that should be expanded in the new matrix.
keepPosOfDims	Number of dimensions to be expanded.
sizesOfExpandDims	Pointer to an array of variables in the expanded dimensions that preserve their original values.
numDimsToExtend	Pointer to the array of new sizes of the expanded matrix.

Discussion

This function creates a new matrix by expanding specified dimensions. It expands
dimensions specified in *dims* only when their size before the expansion is equal to 1.
Dimensions are expanded to the size specified in *sizesOfExpandDims* by adding zeros

at the matrix points in `dims`. Variables at the matrix points specified in `keepPosOfDims` do not change their values.This function is used by the `ExpandObservedNodes` function.

ClearData

Sets all matrix elements to zero.

```
virtual inline void  CMatrix::ClearData() = 0;
```

Discussion

This function sets all matrix elements to zero.This function is used in learning process.

SetUnitData

Sets all elements of numeric matrix to ones.

```
virtual inline void  CMatrix::SetUnitData() = 0;
```

Discussion

This function sets all elements of the numeric matrix to ones. This function is used to get a matrix that does not change another matrix during multiplication.

SetClamp

Forbids matrix change in learning process.

```
inline int  CMatrix::SetClamp( int Clamp );
```

Arguments

Clamp   Status flag:
- 1 means no change is allowed
- 0 means change is allowed
- 0 means previous flag value is retained.

Discussion

This function sets a flag which prevents matrix change. On setting a new value the function returns the value of the flag.

GetClampValue

Returns clamp value.

```
inline int  CMatrix::GetClampValue() const;
```

Discussion

This function returns 1 if no change of the matrix is allowed in the learning process, and returns 0 if change is allowed.

MultiplyInSelf

Performs multiplication and puts result in matrix.

```
virtual void  CMatrix::MultiplyInSelf( const CMatrix<Type>* matToMult, int
    numDimsToMult, const int* indicesToMultInSelf, int isUnifrom = 0, const
    Type uniVal = Type(0) );
```

Arguments

matToMult   Pointer to the matrix to be multiplied.

numDimsToMult	Number of positions for multiplication. Should be the same as the number of dimensions in `matToMult`.
indicesToMultInSelf	Numbers of dimensions of the small matrix in the big matrix.
isUniform	Flag about matrix data. Returns 1 when the matrix does not contain any data and its values are equal to *uniVal,* returns 0 otherwise.
uniVal	Input parameter.Value.

Discussion

This function performs multiplication for numeric matrices only.
The multiplication is performed element by element under the conditions that the
`matToMult` matrix dimensions form a subset of dimensions of the matrix for which the
function was called and do not differ from the function dimensions in size.
To perform multiplication in an empty matrix use *isUniform* flag.

DivideInSelf

Performs division and puts the result in the matrix.

```
virtual void  CMatrix::DivideInSelf( const CMatrix<Type>* matToDiv, int
    numDimsToDiv, const int* indicesToDivInSelf );
```

Arguments

matToDiv	Pointer to the matrix for division.
numDimsToDiv	Number of positions for division. Should be the same as the number of dimensions in *matToMult*.
indicesToDivInSelf	Positions of dimensions of the small matrix in the big matrix.

Discussion

This function performs division for numeric matrices only. The division is performed element by element under the condition that the dimensions of the *matToDiv* matrix form a subset of dimensions of the matrix for which the function is called and do not differ from the function dimensions in size.

GetIndicesOfMaxValue

Returns vector of indices of max value.

```
virtual void  CMatrix::GetIndicesOfMaxValue( intVector* indicesOut ) const;
```

Arguments

indicesOut   Returned vector of indices of max value.

Discussion

This function returns the vector of indices of the max value for numeric matrices only.

NormalizeAll

Creates new matrix by normalizing matrix elements.

```
virtual CMatrix<Type>*  CMatrix::NormalizeAll() const;
```

Discussion

This function returns the pointer to a new matrix with the sum of all elements equal to 1.

Normalize

Normalizes matrix elements.

```
virtual void  CMatrix::Normalize();
```

Discussion

This function changes values of matrix elements dividing them by their sum.

SumAll

Returns sum of all matrix elements.

```
virtual Type  CMatrix::SumAll(int byAbsValue) const;
```

Arguments

ByAbsValue   Summation flag:
                  1 stands for the summation in absolute values;
                  0 stands for simple summation.

InitIterator

Initializes matrix iterator.

```
virtual CMatrixIterator<Type>*  CMatrix::InitIterator() const = 0;
```

Next

Moves iterator to next value.

```
virtual void  CMatrix::Next( CMatrixIterator<Type>* current ) const = 0;
```

Arguments

current   Pointer to the matrix iterator.

Value

Returns pointer to value pointed out by iterator.

```
virtual const Type*  CMatrix::Value( CMatrixIterator<Type>* current ) const =
    0;
```

Arguments

current   Pointer to the matrix iterator.

IsValueHere

Returns information on existence of next value in
matrix.

```
virtual int  CMatrix::IsValueHere( CMatrixIterator<Type>* current ) const = 0;
```

Arguments

current   Pointer to the matrix iterator.

Discussion

This function returns information on the existence of a next value in the matrix.

Index

Returns indices of value of multidimensional matrix pointed out by iterator.

```
virtual void CMatrix::Index( CMatrixIterator<Type>* current,intVector* index )
    const = 0;
```

Arguments

current   Pointer to the matrix iterator.

index   Output vector containing indices of the value pointed out by the iterator.

Discussion

This function returns indices of the value of the multidimensional matrix which is pounted out by the iterator.

Class CDenseMatrix

This class implements virtual functions declared in `CMatrix` class and adds some functionality relevant to its dense entity.

Create

Creates multidimensional dense matrix.

```
static CDenseMatrix<Type> *CDenseMatrix<Type>::Create( int dim, const int
    *range, const Type *data,int Clamp = 0 );
```

---

intel

Arguments

dim	Number of matrix dimensions.
range	Array of lengths of matrix dimensions.
data	Data array for the matrix. Array length is the product of matrix dimension lengths.
Clamp	Status flag: • 1 means no change is allowed • 0 means change is allowed.

Discussion

This function creates a multidimensional dense matrix and returns the pointer to it.

Copy

Creates input matrix replica.

```
static CDenseMatrix<Type> *CDenseMatrix<Type>::Copy( CDenseMatrix<Type>*const
    inputMat );
```

Arguments

pInputMat	Pointer to CDenseMatrix<Type> object to be copied.

GetRawData

Returns matrix data and data length.

```
void *CDenseMatrix<Type>::GetRawData( int *dataLength, const Type **data )
    const;
```

Arguments

dataLength   Returned parameter. Data length.

data   Returned parameter. Pointer to the data array.

Discussion

This function returns the data of the matrix and the data length.

GetRawDataLength

Returns size of raw data array.

```
inline int *CDenseMatrix<Type>::GetRawDataLength() const;
```

SetData

Sets new data for matrix.

```
void *CDenseMatrix<Type>::SetData( const Type* NewData );
```

Arguments

NewData   Pointer to the new data array.

Discussion

This function sets new data for the matrix. Size of new data array should be equal to the raw data length, which is the product of all matrix dimensions.

GetVector

Returns pointer to vector with matrix data.

```
const pnlVector<Type>* GetVector() const;
```

ConvertMultiDimIndex

*Returns offset in data array to element of dense
matrix by multidimensional indices.*

```
inline int *CDenseMatrix<Type>::ConvertMultiDimIndex( const int*
    multidimindexes ) const;
```

Arguments

multidimindices   Pointer to the array of multidimensional indices of the
                         addressed matrix element. The array length is equal to the
                         number of matrix dimensions.

GetElementByOffset

*Returns value of multidimensional matrix element
by offset in data array.*

```
inline Type *CDenseMatrix<Type>::GetElementByOffset(int linearindex) const;
```

Arguments

linearindex   Offset in data array to element of dense matrix.

SetElementByOffset

Sets new value for matrix element by offset in data array.

```
inline void CDenseMatrix<Type>::SetElementByOffset( Type value, int offset );
```

Arguments

value	Value to be set.
offset	Offset in data array to the element of the dense matrix

Class CSparseMatrix

The class implements virtual functions declared in `CMatrix` class and adds some functionality relevant to its sparse entity. This class is based on `CvSparseMat` implemented in OpenCv library and uses the core of this library. It also keeps the default value of a sparse matrix. By default this value is equal to 0.

Create

Creates multidimensional sparse matrix.

```
static CSparseMatrix<Type>* CSparseMatrix<Type>::Create( int dim, const int
    *range, const Type defaultValue, int Clamp = 0 );
```

Arguments

dim	Number of the matrix dimensions.
range	Array of lengths of the matrix dimensions.
defaultValue	Default value of the sparse matrix. The default value is assumed by all the values that are not specified by `SetElementByIndices`() function.

Clamp	Status flag: • 1 means no change is allowed • 0 means change is allowed.

Discussion

This function creates a sparse multidimensional matrix and returns the pointer to it.

Copy

Creates replica of input matrix.

```
static CSparseMatrix<Type>* CSparseMatrix<Type>::Copy( CSparseMatrix<Type>*
    const pInputMat );
```

Arguments

pInputMat	Pointer to the CSparseMatrix<Type> object to be copied.

GetDefaultValue

Returns default value of matrix.

```
inline const Type CSparseMatrix<Type>::GetDefaultValue() const;
```

IsExistingElement

Returns information on element existence at sparse matrix.

```
inline bool CSparseMatrix<Type>::IsExistingElement(const int *multidimindices)
    const;
```

Arguments

multidimindices   Pointer to the array of multidimensional indices of the addressed matrix element. The array length is equal to the number of matrix dimensions.

Discussion

The function returns 'true' if the element is kept in a sparse matrix, returns 'false' otherwise.

Class CNumericDenseMatrix

Create

Creates numeric multidimensional dense matrix.

```
static CNumericDenseMatrix<Type>* CNumericDenseMatrix<Type>::Create( int dim,
    const int *range, const Type *data, int Clamp = 0 );
```

Arguments

dim   Number of matrix dimensions.

range   Array of lengths of matrix dimensions.

data   Data array for the matrix. Array length is the product of matrix dimension lengths.

Clamp   Status flag:
                   • 1 means no change is allowed
                   • 0 means change is allowed.

Discussion

This function creates a multidimensional matrix of numeric data which is instantiated for float and double only and returns the pointer to it.

Class CNumericSparseMatrix

Create

Creates multidimensional sparse matrix.

```
static CNumericSparseMatrix<Type>* CNumericSparseMatrix<Type>::Create( int
    dim, const int *range, int Clamp = 0 );
```

Arguments

dim	Number of matrix dimensions.
range	Array of lengths of matrix dimensions.
Clamp	Status flag: • 1 means no change is allowed • 0 means change is allowed.

Discussion

This function creates a sparse multidimensional matrix of numeric data which is instantiated for `float` and `double` and returns the pointer to it. The default value used by sparse matrices is set to 0.

Class C2DNumericDenseMatrix

Create

Creates plain numeric dense matrix.

```
static C2DNumericDenseMatrix<Type> *C2DNumericDenseMatrix<Type>::Create( const
    int* lineSizes, const Type *data, int Clamp = 0 );
```

Arguments

lineSizes	Array of two integers, the first one is the number of rows of 2D matrix and the second one is the number of columns in the matrix.
data	Array of float matrix data in rows.
Clamp	Flag of clamping: 1 means that no change is permitted 0 means that change is permitted.

Discussion

This function creates a plain two-dimensional numeric dense matrix and returns the pointer to it.

Copy

Creates plain matrix by copying input matrix.

```
static C2DNumericDenseMatrix<Type>* C2DnumericDenseMatrix<Type>::Copy( const
    iC2DNumericDenseMatrix* pInpMat );
```

Arguments

pInpMat	Pointer to the input C2DNumericDenseMatrix matrix.

Discussion

This function creates a plain matrix by copying the input martix.

CreateIdentityMatrix

Creates matrix unit.

```
static C2DNumericDenseMatrix<Type>*
    C2DnumericDenseMatrix<Type>::CreateIdentityMatrix( int lineSize );
```

Arguments

lineSize   Number of rows and columns of the matrix.

Discussion

This function creates a matrix unit. All elements of this matrix are equal to 0 except the elements of the leading diagonal which are equal to unity.

IsSymmetric

Checks square matrix for symmetry.

```
int C2DnumericDenseMatrix<Type>::IsSymmetric( Type epsilon ) const;
```

Arguments

epsilon   Precision of matrix cells comparison.

Discussion

This function checks whether the square matrix is symmetrical. This function is called for square matrices only.

Trace

Returns trace of matrix.

```
Type C2DnumericDenseMatrix<Type>::Trace() const;
```

IsIllConditioned

Checks if matrix is ill-conditioned.

```
int C2DnumericDenseMatrix<Type>::IsIllConditioned( Type conditionRatio )
    const;
```

Arguments

conditionRatio   Limit on ratio between the largest and the smallest singular
                         values.

Discussion

This function checks the condition of the matrix comparing the ratio between the
largest and the smallest singular values. If the ratio is bigger than the input ratio the
member function returns 1, returns 0 otherwise.

Determinant

Returns matrix determinant.

```
Type C2DnumericDenseMatrix<Type>::Determinant() const;
```

Discussion

This function returns the determinant of a plain square matrix.

Inverse

Returns pointer to inverse matrix.

```
C2DNumericDenseMatrix<Type>* C2DnumericDenseMatrix<Type>::nverse() const;
```

Discussion

This function returns the pointer to the matrix inverse to the given flat square matrix.The function throws exception if the matrix is ill-conditioned.

Transpose

Transposes matrix.

```
C2DNumericDenseMatrix<Type>* C2DnumericDenseMatrix<Type>::Transpose()const;
```

Discussion

This function returns pointer to the transposed matrix.

GetLinearBlocks

Breaks linear matrix into blocks.

```
void C2DnumericDenseMatrix<Type>::GetLinearBlocks( const int *X, int
    xSize,const int *blockSizes, int numBlocks, C2DNumericDenseMatrix<Type>
    **matXOut, C2DNumericDenseMatrix<Type> **matYOut ) const;
```

Arguments

X	Pointer to the array of numbers of blocks that should enter matrix X.
xSize	Number of blocks that should enter matrix X.

blockSizes	Array of block sizes.
numBlocks	Number of blocks.
matXOut	Returned parameter. Pointer to the matrix of blocks that correspond to their numbers in matrix *X*.
matYOut	Returned parameter. Pointer to the matrix of blocks that do not belong to matrix *X*.

Discussion

This function breaks the matrix with dimension sizes *1* and *N* into the matrix with sizes *1* and *K*, *1* and *M*, where *K+M = N*. The matrix is divided into two parts which are related to two matrices. The resulting matrices are formed by assembling all the parts one-by-one.

X	X	Y	X	Y	X	->	X	X	X	X	&	Y	Y
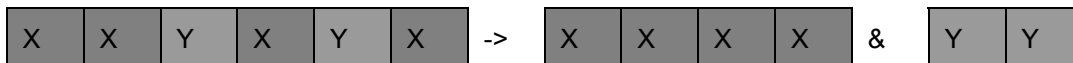
GetBlocks

Breaks matrix into blocks.

```
void C2DnumericDenseMatrix<Type>::GetBlocks( const int *X, int xSize,const int
    *blockSizes, int numBlocks, C2DNumericDenseMatrix<Type> **matXOut,
    C2DNumericDenseMatrix<Type> **matYOut, C2DNumericDenseMatrix<Type>
    **matXYOut, C2DNumericDenseMatrix<Type> **matYXOut )const;
```

Arguments

X	Pointer to the array of numbers of blocks that should enter matrix X.
xSize	Number of blocks that should enter matrix X.
blockSizes	Array of block sizes.
numBlocks	Number of blocks.
matXOut	Returned parameter. Pointer to the matrix of blocks that correspond to their numbers in matrix X.

matYOut	Returned parameter. Pointer to the matrix of blocks that do not belong to matrix X.
matXYOut	Returned parameter. Pointer to the matrix of blocks XY.
matYXOut	Returned parameter. Pointer to the matrix of blocks YX.

Discussion

This function breaks a flat square matrix into blocks, creates new matrices of the resulting blocks and returns the newly created matrices as arguments *matX*, *matY*, *matXY*, and *matYX*. Breaking of a matrix into a bigger number of blocks is carried out similarly. Returned matrices are created by placing blocks in accordance with their initial locations.

Figure 3-2   Breaking matrix in two blocks

	X	Y
X	X	XY
Y	YX	Y

Figure 3-3   Breaking matrix into several blocks

	Y	X	Y	X	Y
Y	Y	Y X	Y	YX	Y
X	XY	X	XY	X	XY
Y	Y	Y X	Y	YX	Y
X	XY	X	XY	X	XY
Y	Y	Y X	Y	YX	Y

pnlMultiply

Creates new matrix out of two source matrices.

```
friend PNL_API C2DNumericDenseMatrix<Type>*
    C2DnumericDenseMatrix<Type>::pnlMultiply( const
    C2DNumericDenseMatrix<Type>* pxMatrix1,const C2DNumericDenseMatrix<Type>*
    pxMatrix2, int maximize );
```

intel.

Arguments

pxMatrix1	Pointer to the first matrix.
pxMatrix2	Pointer to the second matrix.
maximize	Flag index: 1 stands for the maximum value 0 stands for a pair-wise product of the values.

Discussion

This function returns the pointer to a new matrix which is created out of two source matrices multiplied row-by-column. The source matrices should have at least one dimension of the same length.

Reference Counter

Class CReferenceCounter

Class `CReferenceCounter` helps to attach an object to other objects. A `CReferenceCounter` object cannot be created separately because it contains neither a public constructor nor a friendly function to call a constructor. A `CReferenceCounter` class object is created with a `CMatrix` object or a `CModelDomain` object. Class `CReferenceCounter` is designed to store pointers to related objects. A `CReferenceCounter` object may be deleted only if it is not associated with any other object.

There are two versions of the class: debug and release. The debug version stores pointers to referenced objects and finds them while calling Release(). The release version stores only the number of references, and either increases or decreases it.

AddRef

Increases number of references.

```
inline void CReferenceCounter:: AddRef( void* pObject );
```

intel

Arguments

pObject　　　　　Pointer to the object referenced to the CReferenceCounter object.

Release

Decreases number of references.

```
inline void CReferenceCounter::Release( void* pObject );
```

Arguments

pObject　　　　　Pointer to the object referenced to the CReferenceCounter object.

GetNumOfReferences

Returns number of referenced objects.

```
inline int CReferenceCounter::GetNumberOfReferences() const;
```

Inference Engines
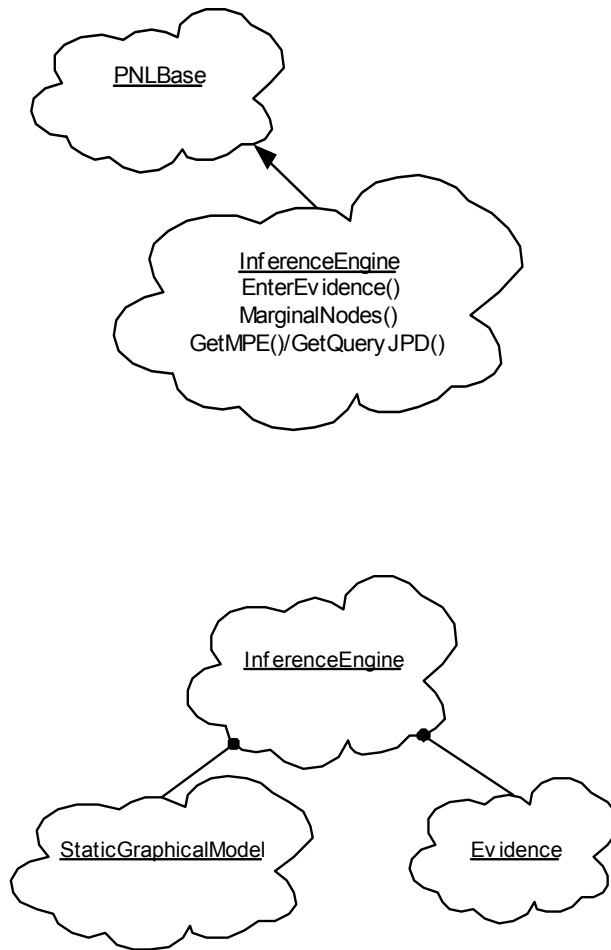
Class CInfEngine

Class `CInfEngine` is basic for all classes that implement inference in graphical
models. The class contains functions that belong to its child classes.

pnlDetermineDistributionType

Returns type of distribution.

```
EDistributionType CInfEngine::pnlDetermineDistributionType( int numOfAllNds,
    int numOfObsNds, int *pObsNdsIndices, const CNodeType **pAllNdsTypes );
```

Arguments

numOfAllNds	Number of all nodes.
NumOfObsNds	Number of the observed nodes.
pObsNdsIndices	Pointer to the array of indices of the observed nodes.
pAllNdsTypes	Pointer to the array of pointers to all node types.

Discussion

This function returns the following node distribution types:

— *dtTabular*, if all non-observed, or 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.

pnlDetermineDistributionType

Returns type of distribution.

```
EDistributionType CInfEngine::pnlDetermineDistributionType(const CModelDomain*
    pMD, int nQueryNodes, const int* query, const CEvidence* pEv );
```

Arguments

pMD	Model domain.
nQueryNodes	Number of query nodes.

`query`	Query nodes.
`pEv`	Evidence.

Discussion

This function returns the following node distribution types:

— `dtTabular`, if all non-observed, or 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.

```
void CInfEngine::EnterEvidence( CEvidence *evidence, int maximize = 0, int
    sumOnMixtureNode = 1 ) = 0;
```

Arguments

`evidence`	Pointer to a `CEvidence` object that contains the observed nodes and their values.
`maximize`	Optional parameter used in Pearl Inference for selection of message transmission type.
`sumOnMixtureNode`	Flag of summation on the mixture node.

Discussion

This function starts inference in the graphical model. Principal operations are performed either on call of this function or on call of the function MarginalNodes, depending on what inference type is implemented.

MarginalNodes

Calculates joint probability distribution.

```
void CInfEngine::MarginalNodes( const int *query, int querySize, int
    nodeExpandJPD = 1 ) = 0;
void CInfEngine::MarginalNodes( const intVector& query, int nodeExpandJPD =
    1);
```

Arguments

query	Array of numbers of the nodes addressed for the joint probability distribution.
querySize	Number of nodes addressed for joint probability distribution.
nodeExpandJPD	Flag of the expand operation.

Discussion

This function calculates joint probability distribution for nodes addressed. In process of calculations Most Probability Explanation (MPE) is also created as a <u>Model Domain</u> object for the calculated probability distribution. To receive MPE and the factor, special functions should be used.

GetQueryJPD

Returns const *pointer to joint probability distribution.*

```
const CFactor* CInfEngine::GetQueryJPD() const;
```

Discussion

This function returns to joint probability distribution calculated by using the function MarginalNodes. The returned pointer is `const`, as the memory is released when the CInfEngine object is deleted or when MarginalNodes is called again.

GetMPE

Returns MPE.

```
const CEvidence* CInfEngine::GetMPE() const;
```

Discussion

This function returns to most probability explanation calculated by using the function MarginalNodes. The returned pointer is `const`, as the memory is released when the CInfEngine object is deleted or when MarginalNodes is called again.

GetModel

Returns pointer to graphical model processed by inference engine.

```
inline CStaticGraphicalModel * CInfEngine::GetModel() const;
```

Discussion

This function returns the pointer to the graphical model which is processed by the infrence engine. This function is necessary for operation of learning algorithms.

GetObsGauNodeType

Returns type of observed Gaussian node.

```
inline static const CNodeType* CInfEngine::GetObsGauNodeType();
```

GetObsTabNodeType

Returns type of observed Tabular node.

```
inline static const CNodeType* CInfEngine::GetObsTabNodeType();
```

Class CNaiveInfEngine
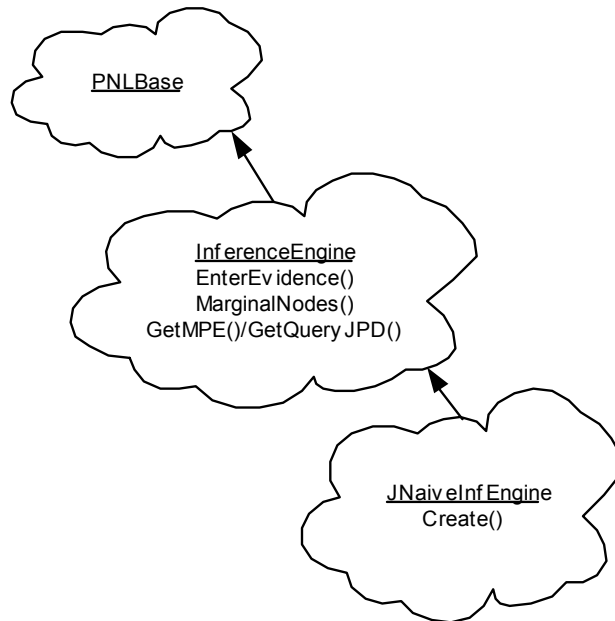
Create

Creates class object.

```
static CNaiveInfEngine* CNaiveInfEngine::Create( CStaticGraphicalModel*
    pGrModel );
```

Arguments

pGrModel   Pointer to a model, for which inference algorithm is to be
                                    carried out. Note, that it can either be an MRF (MRF2) or a
                                    BNet.

Class CPearlInfEngine

```
PNLBase
```

```
InferenceEngine
EnterEvidence()
MarginalNodes()
GetMPE()/GetQueryJPD()
```

```
PearlInfEngine
Create()
GetNumOfIterations()
SetMaxNumOfIterations()
```
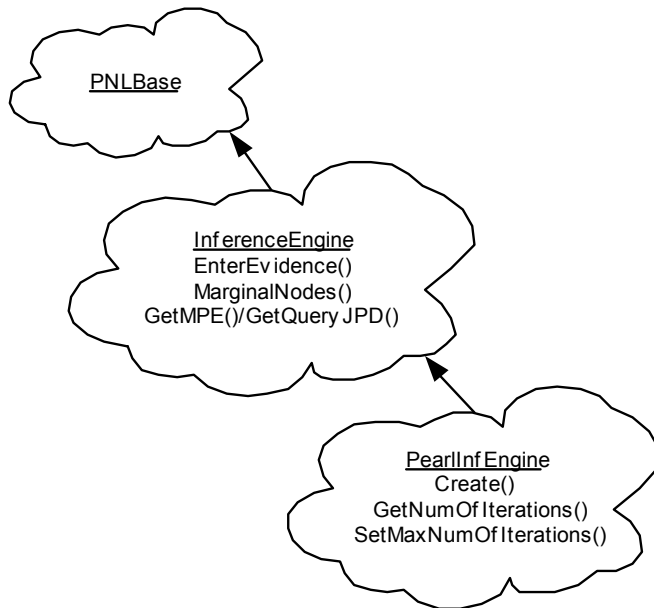
Create

Creates class object.

```
static CPearlInfEngine* CPearlInfEngine::Create( CStaticGraphicalModel*
    pGrModel );
```

Arguments

pGrModel   Pointer to a model, for which inference algorithm is to be
                       carried out. Note, that it can either be an MRF2 or a BNet. The
                       input BNet cannot have directed cycles in it.

Discussion

This function creates a Pearl Inference algorithm object from an input graphical model and returns a pointer to it. Note, that the result of inference is exact, if the graph of the input model (Class CBNet or Class CMRF2) does not have undirected loops in it. If it does, then the result is approximate.

IsInputModelValid

Checks validity of model for use in Pearl inference.

```
static bool CStaticGraphicalModel::IsInputModelValid( const
    CCsaticGraphicalModel *spGrModel );
```

Arguments

pGrModel   Pointer to the graphical model.

SetMaxNumberOfIterations

Sets maximum number of iterations for parallel protocol.

```
inline void CPearlInfEngine::SetMaxNumberOfIterations(int maxNumOfIters);
```

Arguments

maxNumOfIters Maximum number of iterations.

Discussion

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

GetNumberOfProvideIterations

Returns number of iterations performed.

```
inline int CPearlInfEngine::GetNumberOfProvideIterations() const;
```

SetTolerance

Sets tolerance for convergency check.

```
inline  void CPearlInfEngine::SetTolerance( float tolerance );
```

Arguments

tolerance   Input parameter. Precision.

Class CSpecPearlInference
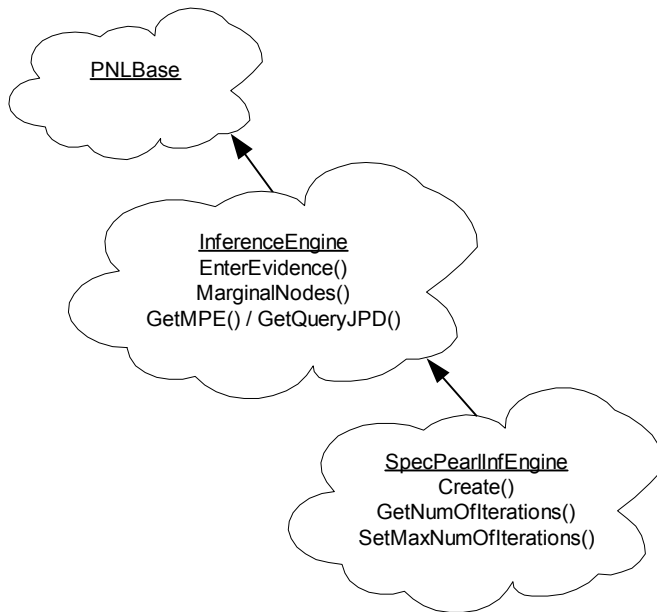
This class is a realisation of Pearl Inference that does not allocate memory during the inference procedure.

Create

Creates class object.

```
static CSpecPearlInfEngine* CSpecPearlInfEngine::Create(CStaticGraphicalModel*
    pGrModel );
```

Arguments

pGrModel   Pointer to a model, for which inference algorithm is to be carried out. Note, that it can either be an MRF2 or a BNet. The input BNet cannot have directed cycles in it.

Discussion

This function creates a Pearl Inference algorithm object from an input graphical model and returns a pointer to it. Note, that the result of inference is exact, if the graph of the input model (`Class CBNet` or `Class CMRF2`) does not have undirected loops in it. If it does, then the result is approximate.

IsInputModelValid

Checks validity of model for use in Pearl inference.

```
static bool  CSpecPearlInfEngine::IsInputModelValid( const
    CStaticGraphicalModel* pGrModel);
```

Arguments

pGrModel   Pointer to the graphical model.

SetMaxNumberOfIterations

Sets maximum number of iterations for parallel protocol.

```
inline void  CSpecPearlInfEngine::SetMaxNumberOfIterations(int maxNumOfIters)
```

Arguments

maxNumOfIters Maximum number of iterations.

Discussion

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

intel

GetNumberOfProvideIterations

Returns number of iterations performed.

```
inline int CSpecPearlInfEngine::GetNumberOfProvideIterations() const;
```

SetTolerance

Sets tolerance for convergency check.

```
inline  void CSpecPearlInfEngine::SetTolerance( float tolerance );
```

Arguments

tolerance   Input parameter. Precision.

Class CJtreeInfEngine
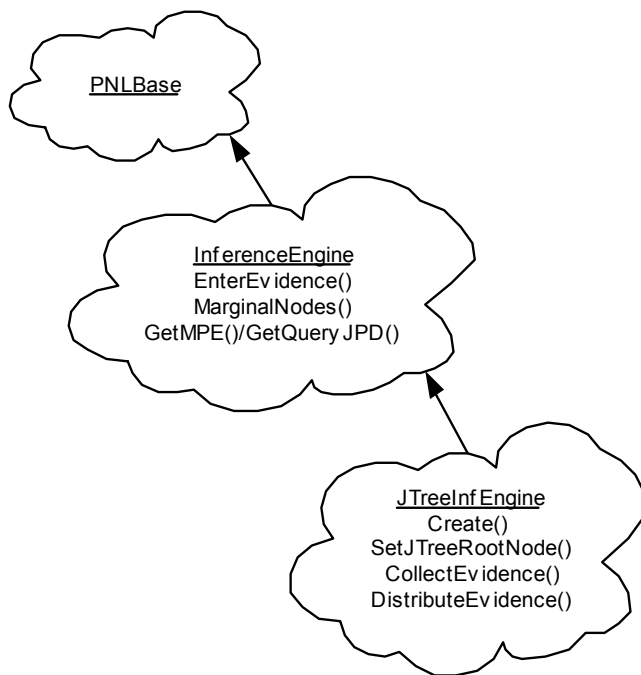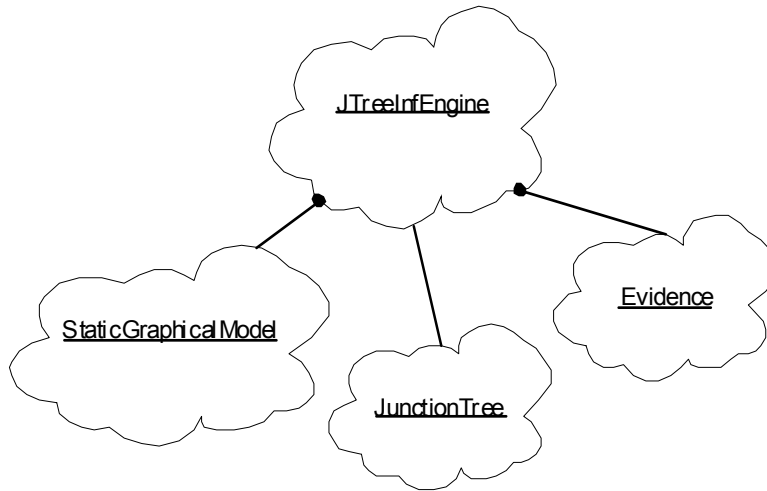
Create

Creates class object.

```
static CJtreeInfEngine* CJtreeInfEngine::Create( const CStaticGraphicalModel*
    pGrModel, int numOfSubGrToConnect, const int *SubGrToConnectSizes, int
    **SubGrToConnect );

static CJtreeInfEngine* CJtreeInfEngine::Create( const CStaticGraphicalModel
    *pGrModel, const intVecVector& SubGrToConnect = intVecVector() );

staticCJtreeInfEngine*CJtreeInfEngine::Create(constCStaticGraphicalModel
    *pGrModel, CJunctionTree *pJTree );
```

Arguments

pGrModel   Pointer to a model, for which inference algorithm is to be
                           carried out. Note, that it can either be an `MRF2` or a `BNet`. The
                           input `BNet` cannot have directed cycles in it.

numOfSubGrToConnect	Number of subgraphs, which the user wants to appear connected.
SubGrToConnect	Nodes to be connected.
SubGrToConnectSizes	Sizes of the subgraphs, which the user wants to appear connected.
pJTree	Pointer to the Junction tree.

Copy

Creates replica of CJTreeInfEngine *object.*

```
static CJtreeInfEngine* CJtreeInfEngine::Copy( const CJtreeInfEngine
    *pJTreeInfEng );
```

Arguments

pJTreeInfEng   Pointer to the Junction tree inference engine.

GetEvidence

Returns pointer to the evidence which was provided.

```
inline const CEvidence* CJtreeInfEngine::GetEvidence() const;
```

GetJTreeRootNode

Returns number of root node.

```
inline int CJtreeInfEngine::GetJTreeRootNode() const;
```

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.

```
inline void CJtreeInfEngine::GetClqNumsContainingSubset( int numOfNdsInSubset,
    const int *subset, int *numOfClqs, const int **clqsContSubset ) const;
inline void CJtreeInfEngine::GetClqNumsContainingSubset( const intVector&
    subset, intVector* clqsContSubset ) const;
```

Arguments

numOfNdsInSubset	Size of the subset.
subset	Subset of nodes.
numOfClqs	Returned parameter. Number of cliques with the common subset.
clqsContSubset	Returned parameter. Array of numbers of cliques with the common subset.

Discussion

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

GetNodesConnectedByUser

Returns set of connected nodes.

```
inline void CJtreeInfEngine::GetNodesConnectedByUser( int nodeSetNum, int
    *numOfNds, const int **nds ) const;
```

Arguments

nodeSetNum	Number of the set of nodes.
numOfNds	Return parameter. Size of the set of nodes.
nds	Return parameter. Pointer to the set of nodes.

Discussion

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

SetJTreeRootNode

Sets root of Junction tree.

```
inline virtual void CJtreeInfEngine::SetJTreeRootNode( int nodeNum ) const;
```

Arguments

nodeNum   Node number.

Discussion

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

GetLogLik

Returns logarithm of likelihood.

```
virtual float CJtreeInfEngine::GetLogLik() const;
```

MultJTreeNodePotByDistribFun

*Multiplies potential of Junctin tree node by
distribution function.*

```
virtual void CJtreeInfEngine::MultJTreeNodePotByDistribFun( int clqPotNum,
    const int *domainIn, const CDistribFun *pDistrFunIn );
```

Arguments

clqPotNum	Number of a clique of the potential.
domain	Numbers of domain nodes.
pDistrFun	Pointer to the distribution function.

Discussion

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

DivideJTreeNodePotByDistribFun

*Multiplies potential of Junctin tree node by
distribution function.*

```
virtual void    CJtreeInfEngine::DivideJTreeNodePotByDistribFun(  int
    clqPotNum, const int *domainIn, const CDistribFun *pDistrFunIn );
```

Arguments

clqPotNum	Number of a clique of the potential.
domain	Numbers of domain nodes.
pDistrFun	Pointer to the distribution function.

Discussion

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

CollectEvidence

Collects evidence.

```
virtual void CJtreeInfEngine::CollectEvidence();
```

DistributeEvidence

Distributes evidence.

```
virtual void CJtreeInfEngine::DistributeEvidence();
```

ShrinkObserved

Initializes Junction tree using given evidence.

```
virtual void CJtreeInfEngine::ShrinkObserved( const CEvidence *pEvidence, int
    maximize = 0, int sumOnMixtureNode = 1, bool bRebuildJTree = true );
```

Arguments

pEvidence	Pointer to 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.

```
const *CPotential CJtreeInfEngine::QueryMPE() const;
```

Class CExInfEngine

The class is a template derived from `CInfEngine`.

Create

Creates class object.

```
static CExInfEngine< INF_ENGINE, MODEL, FLAV, FALLBACK_ENGINE1,
    FALLBACK_ENGINE2 > *CExInfEngine::Create( CStaticGraphicalModel const
    *model );
```

Template arguments

`INF_ENGINE`	Type of underlying inference engine
`MODEL`	Type of graphical model to work with
`FLAV`	Flavor of limitations to work around
`FALLBACK_ENGINE1`	Fallback inference engine to use for submodels of size 1

FALLBACK_ENGINE2   Fallback inference engine to use for submodels of size 2

Arguments

model   Graphical model.

Discussion

Creates an object of the class.

Default values are provided for all template arguments beyond INF_ENGINE.

FLAV is the flavor of limitations to work with. It can be any combination of flags *PNL_EX INFENGINEFLAVOUR_DISCONNECTED, PNL_EXINFENGINEFLAVOUR_UNSORTED.*

Flag *PNL_EXINFENGINEFLAVOUR_DISCONNECTED* means that you work with the limitation of the underlying engine that does not support disconnected models. Flag *PNL_EXINFENGINEFLAVOUR_UNSORTED* means that you work with the limitation of the underlying engine that does not support topologically unsorted models. *FLAV* has the default value of *PNL_EXINFENGINEFLAVOUR_DISCONNECTED | PNL_EXINFENGINEFLAVOUR_UNSORTED.* *MODEL* has the default value of CBNet. FALLBACK_ENGINE1 has the default value of CNaiveInferenceEngine. FALLBACK_ENGINE2 has the default value of INF_ENGINE itself.

Class CFGSumMaxInfEngine
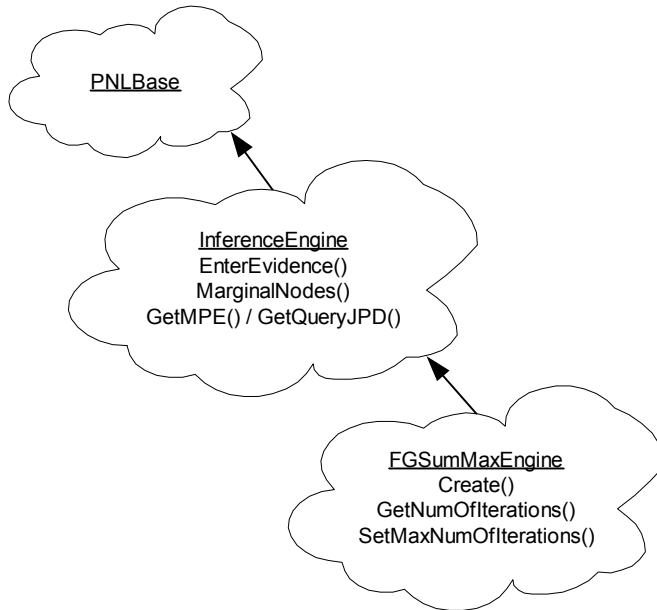
The class implements belief propagation on a factor graph model.

Create

Creates object of class.

```
static CFGSumMaxInfEngine* CJtreeInfEngine::Create( const
    CStaticGraphicalModel *pGrModel );
```

Arguments

PGrModel   Pointer to a model, for which inference is to be carried out.

Discussion

This function creates an object of the class. Inference is carried out for `FactorGraph` models only.

SetMaxNumberOfIterations

Sets maximum number of iterations for inference.

```
void CJtreeInfEngine::SetMaxNumberOfIterations( int number );
```

Arguments

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.

```
inline int CJtreeInfEngine::GetNumberOfProvideIterations() const;
```

SetTolerance

Sets value of tolerance used for check-up convergence.

```
void CJtreeInfEngine::SetTolerance( float tolerance);
```

Arguments

tolerance   Tolerance value.

Discussion

This function sets the value of tolerance which is used in the process of convergence checking.

Class CSamplingInfEngine
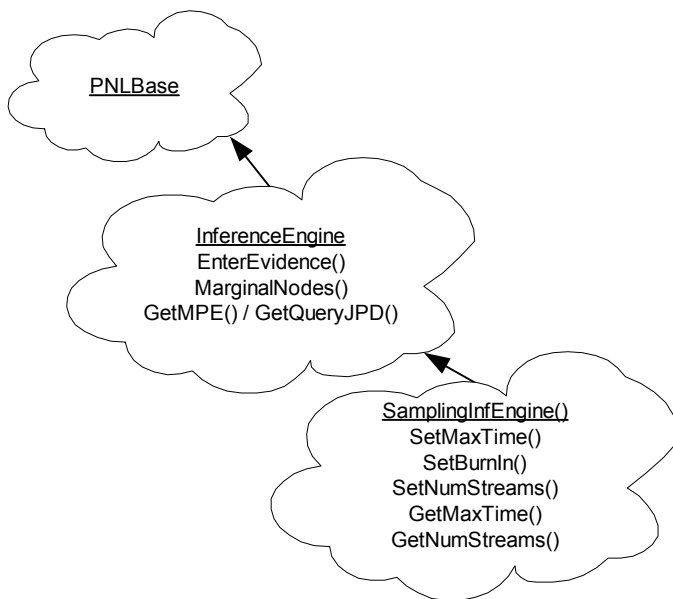
Class `CSamplingInfEngine` is a superclass for classes `CGibbsSamplingInfEngine` and `CGibbsWithAnnealingInfEngine` that implement inference in static graphical models using stochastic simulates technique known as Markov chain Monte Carlo. This technique generates samples from the required posterior 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: $\langle x_k / \{x_i^{\langle t-1 \rangle} = a_i, i \neq k \} = \frac{1}{a} P \langle x_k, \{x_k^{\langle t-1 \rangle} = a_i, i \neq k$

SetMaxTime

Sets maximum number of sampling iterations.

```
void CSamplingInfEngine::SetMaxTime( int time );
```

Arguments

time   Maximum number of iterations.

Discussion

This function sets the maximum number of sampling iterations.

SetBurnIn

Sets number of iterations before statistics collection.

```
void CSamplingInfEngine::SetBurnIn(int time);
```

Arguments

time   Number of iterations.

Discussion

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

SetNumStreams

Sets number of streams for sampling.

```
void CSamplingInfEngine::SetNumStreams( int nStreams );
```

Arguments

nStreams   Number of streams.

Discussion

This function sets the number of independent streams for sampling.

GetMaxTime

Returns maximum number of sampling iterations.

```
int CSamplingInfEngine::GetMaxTime();
```

Discussion

This function returns the maximum number of sampling iterations.

GetBurnIn

Returns number of iterations before statistics collection.

```
int CSamplingInfEngine::GetBurnIn();
```

Discussion

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

GetNumStreams

```
int CSamplingInfEngine::GetNumStreams();
```

Discussion

This function returns the number of sampling streams.

Continue

Continues sampling.

```
void CSamplingInfEngine::Continue( int dT);
```

Arguments

dT Number of additional samples.

Discussion

This function continues sampling procedure and statistics update.

Class CGibbsSamplingInfEngine
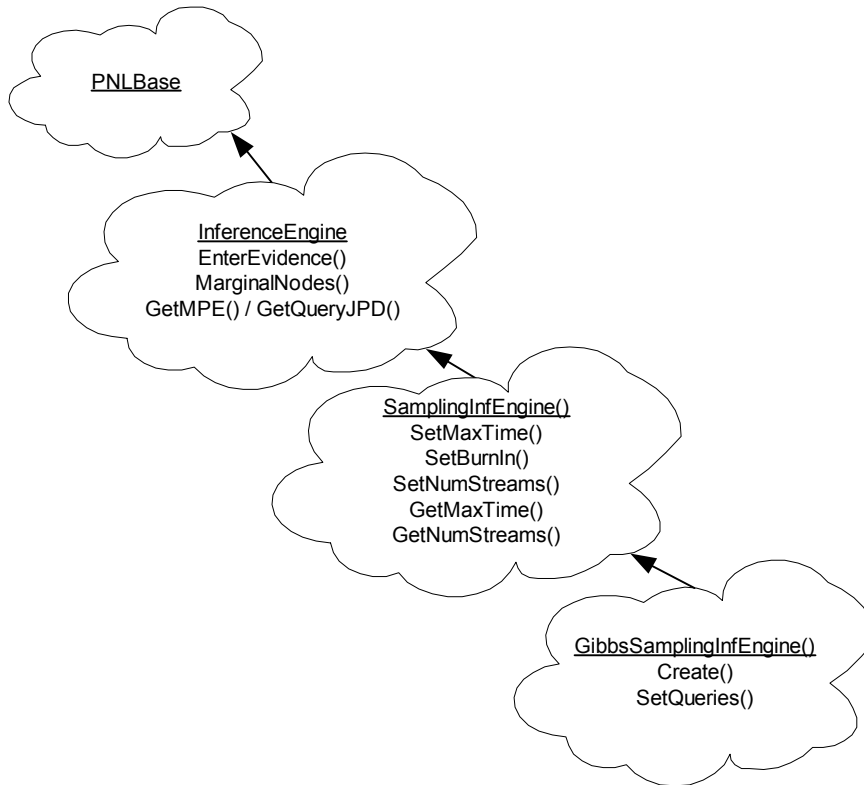
Create

Creates class object.

```
static CGibbsSamplingInfEngine* CGibbsSamplingInfEngine::Create( const
    CStaticGraphicalModel *pGrlModel );
```

Arguments

pGrModel   Pointer to a model, for which inference algorithm is to be carried out.

Discussion

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

SetQueries

Sets possible queries.

```
void CGibbsSamplingInfEngine::SetQueryes( intVecVector &queries );
```

Arguments

Queries   Vector of possible queries.

Discussion

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

UseDSeparation

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

```
void CGibbsSamplingInfEngine::UseDSeparation( bool isUsing );
```

Arguments

isUsing   Flag of *d*-separation.

Discussion

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

Class CGibbsWithAnnealingInfEngine
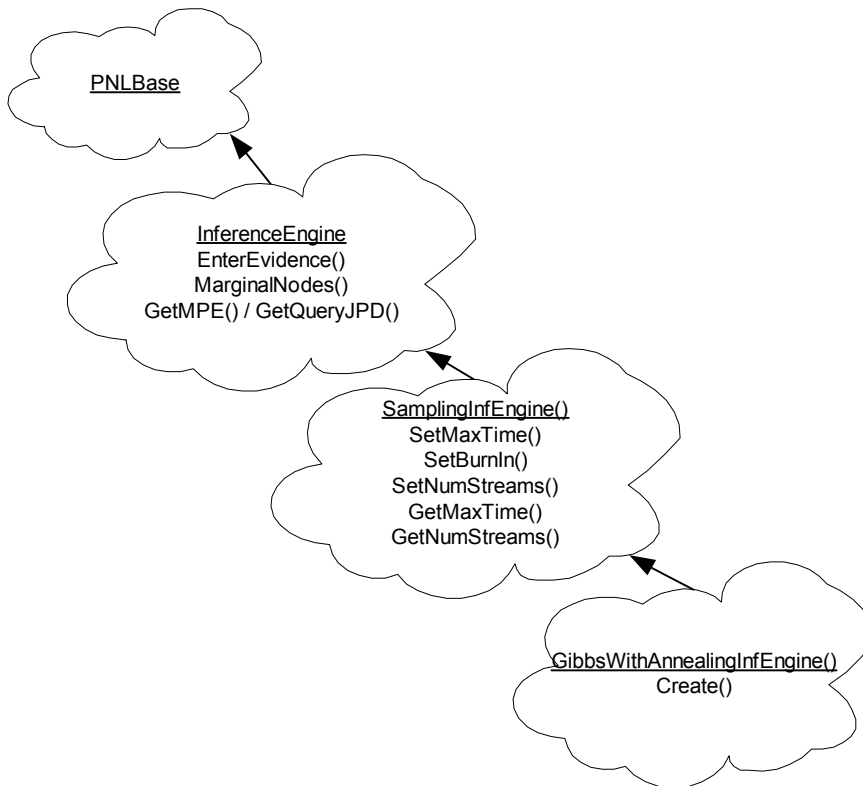
`CGibbsWithAnnealingInfEngine` implements Gibbs Sampler with annealing schedule $T\langle S\rangle$

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

where $T\langle S\rangle$ is the temperature which depends on the sampling iteration and C is a parameter.

This inference finds 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].

Create

Returns class object.

```
static CGibbsWithAnnealingInfEngine* CGibbsWithAnnealingInfEngine::Create(
    const CStaticGraphicalModel *pGrlModel );
```

Arguments

pGrModel   Pointer to a model, for which inference algorithm is to be carried out.

Discussion

This function creates a class object.

SetAnnealingCoefficientC

Changes default coefficient C of annealing schedule.

```
void CGibbsWithAnnealingInfEngine::SetAnnealingCoefficientC( float val );
```

Arguments

val   Value of the coefficient.

Discussion

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

SetAnnealingCoefficientS

Changes default coefficient S of annealing schedule.

```
void CGibbsWithAnnealingInfEngine::SetAnnealingCoefficientS( float val );
```

Arguments

val   Value of the coefficient.

Discussion

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

GetCurrentTemp

Returns value of current temperature.

```
float CGibbsWithAnnealingInfEngine::GetCurrentTemp() const;
```

UseAdaptation

Uses adaptation during inference.

```
void CGibbsWithAnnealingInfEngine::UseAdaptation( bool isUse );
```

Arguments

isUse   Flag of use of the adaptation.

Class CLWSamplingInfEngine
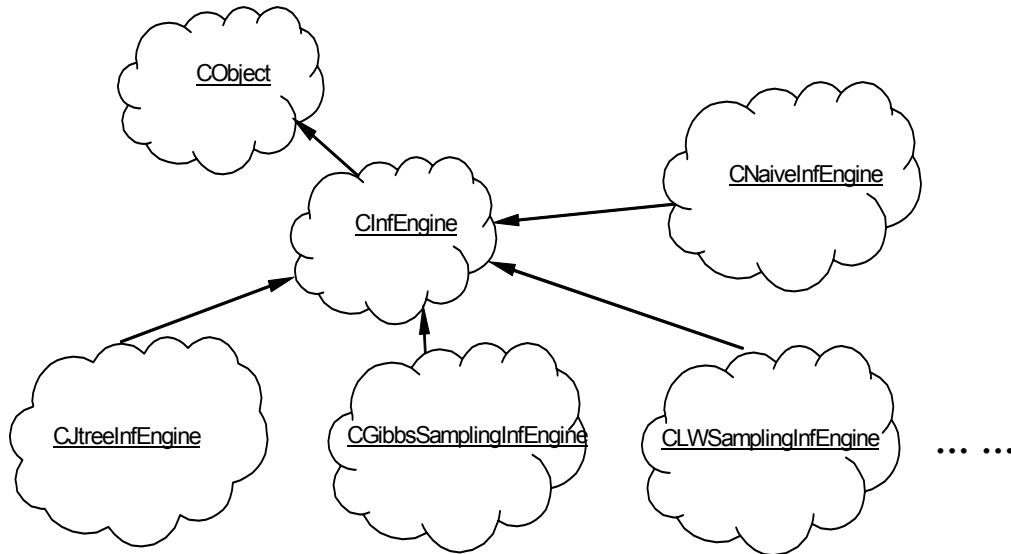
Class `CLWSamplingInfEngine` implements the particle-based inference engine for static `BNet` models.

Create

Creates particle-based inference engine for `BNet`.

```
static CLWSamplingInfEngine* Create( const CStaticGraphicalModel
    *pGraphicalModelIn, int particleCount= 400 );
```

Arguments

pGrModelIn	Static graphical model.
nParticleCount	Number of particles.

SetParemeter

Sets parameters of `LWSamplingInfEngine`.

```
void SetParemeter(  int nParticleCount = 400)
```

Arguments

nParticleCount   Number of particles.

Discussion

This function allows you to change particle counts of the `CLWSamplingInfEngine`.

LWSampling

Generates particles using observed evidence.

```
void  LWSampling(const CEvidence* pEvidenceIn = NULL)
```

Arguments

pEvidenceIn   The observed evidence.

Discussion

Both `LWSampling` and `EnterEvidence` generate particles. Unlike `EnterEvidence`, `LWSampling` cannot generate particle weights. To input particle weights you call `EnterEvidenceProbability()`.

EnterEvidenceProbability

Inputs likelihood of observations.

```
void  EnterEvidenceProbability( floatVector *pEvidenceProbIn )
```

Arguments

pEvidenceProbIn   Vector of particle weights.

Discussion

This function calculates automatically particle weights under observance.With this function you easily calculate the likelihood $\hat{\omega}$of observations.

GetCurSamples

Returns states of all current particles.

```
pEvidencesVector* GetCurSamples();
```

Discussion

This function returns all node values of particles.

GetParticleWeights

Gets weights of all current particles.

```
floatVector*      GetParticleWieghts()
```

Estimate

Estimates real values of nodes.

```
void Estimate(CEvidence *pEstimate)
```

Arguments

pEstimate   Given the estimated nodes as input.

Discussion

This function estimates real values (states) of nodes by all generated particles. It is similar to `GetMPE` but more convenient as it does not call `MarginalNodes`.The estimated values are stored in *pEstimate* as output.

GetNeff

Gets number of current effective particles.

```
float GetNeff()
```

Arguments

*float*   The current number of effective particles.

Discussion

This function acquires the number of current effeclive particles. If it is small enough, you re-initialize particle weights and values. Set by default, `LWSampling` detects the number of effective particles and re-initializes the particle weight if it is too small automatically.

Class CDynamicInfEngine

Class `CDynamicInfEngine` is a superclass for all classes that implement inference in dynamic graphical models. Figure 3-4 shows the underlying structure of the hierarchy

of `CDynamicInfEngine` class.

Figure 3-4   Structure of `CDynamicInfEngine` Class
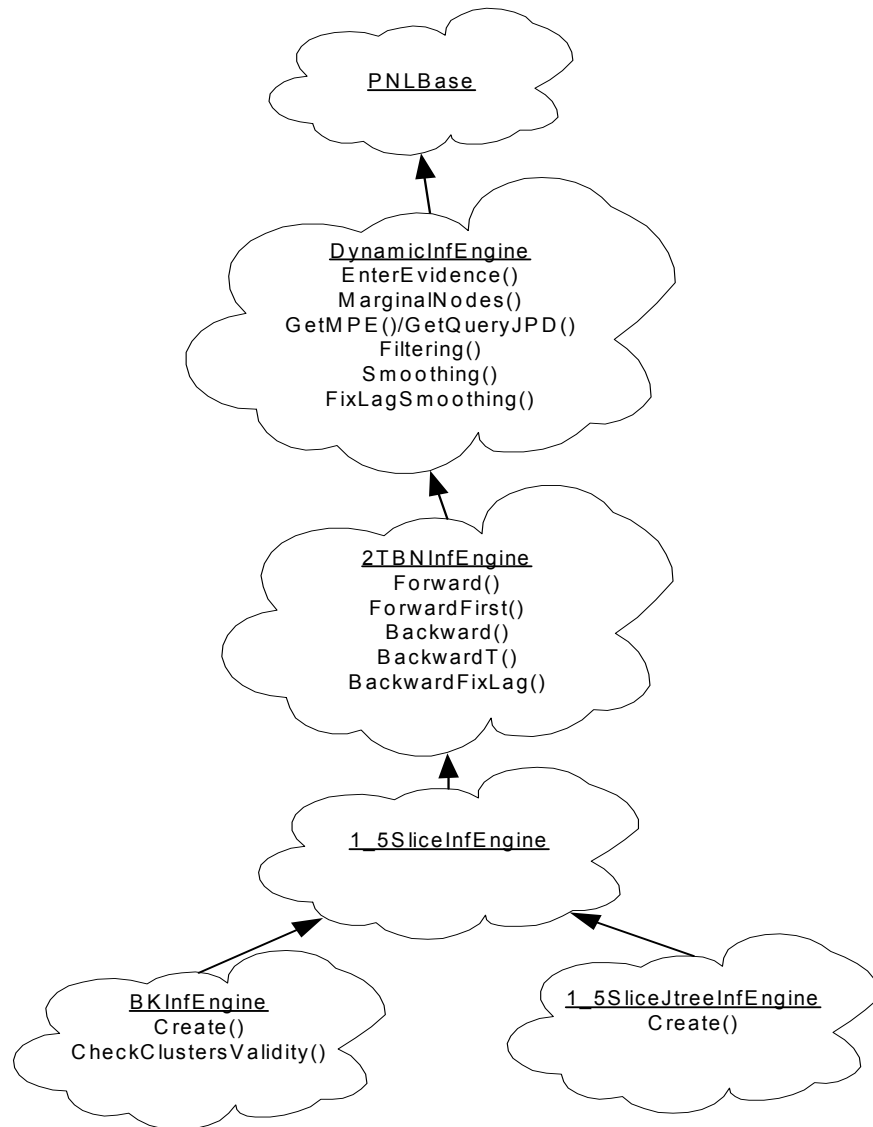
Public Member Functions

DefineProcedure

Defines type of inference procedure.

```
void CDynamicInfEngine::DefineProcedure( EProcedureTypes procedure, int lag );
```

Arguments

procedure	Type of the inference procedure that can be either `itFiltering`, `itSmoothing`, `itFixLagSmoothing` or `itViterbi`.
lag	Integer value, which corresponds to the value of lag for the problem of fixed-lag smoothing. Note that it must be equal to 0 for the filtering problem. For the problem of smoothing, this argument is the number of time slices.

Discussion

This member function defines the type of inference procedure as filtering, smoothing, fixed-lag smoothing or Viterby decoding.

EnterEvidence

Enters evidence to engine.

```
void CDynamicInfEngine::EnterEvidence( const CEvidence*const* evidences,  int
    numOfEvidences );
void CDynamicInfEngine::EnterEvidence(const pConstEvidenceVector& evidences);
```

Arguments

evidences	Pointer to the array of pointers to the collection of evidences for all the slices.
numOfEvidences	Size of array, that is, the number of evidences or the number of slices.

MarginalNodes

Marginalizes joint probability distribution to query nodes.

```
void CDynamicInfEngine::MarginalNodes( const int* query, int querySize, int
    timeSlice = 0, int notExpandJPD = 0 ) = 0;

void CDynamicInfEngine::MarginalNodes( const intVector& query, int timeSlice =
    0, int notExpandJPD = 0 );
```

Arguments

query	Array of nodes, which the user wants to appear in the query.
querySize	Number of nodes in the query.
timeSlice	Query slice number, which should be equal to 0 for the filtering and fixed-lag smoothing problems, because they are on-line procedures.
notExpandJPD	Flag of expanding.

Discussion

The call of this member function marginalizes the joint probability distribution to the set of nodes in slice given as the *query* input argument, at the time *timeSlice*. For filtering argument *timeSlice* must be equal to the current time and for fixed-lag smoothing it must be equal to time-lag.

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

GetQueryJPD

Returns joint probability distribution of query nodes.

```
virtual const CPotential* CDynamicInfEngine::GetQueryJPD() = 0;
```

GetMPE

Returns most probable explanation of query nodes.

```
virtual const CEvidence* CDynamicInfEngine::GetMPE() = 0;
```

Filtering

Performs filtering procedure.

```
void CDynamicInfEngine::Filtering( int timeSlice );
```

Arguments

`timeSlice`   Current time-slice number.

Discussion

This function performs filtering procedure.

Smoothing

Performs smoothing procedure.

```
void CDynamicInfEngine::Smoothing();
```

Discussion

This function performs smoothing procedure.

FixLagSmoothing

Performs fixed-lag smoothing procedure.

```
void CDynamicInfEngine::FixLagSmoothing( int timeSlice );
```

Arguments

timeSlice   Current time-slice number.

Discussion

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

FindMPE

Finds most probable explanation.

```
virtual const CEvidence* CDynamicInfEngine::GetMPE() = 0;
```

Discussion

This function performs the procedure of Viterbi decoding.

GetDynamicModel

Returns given dynamic model.

```
const CDynamicGraphicalModel* CDynamicInfEngine::GetDynamicModel() const;
```

GetProcedureType

Returns type of inference procedure.

```
EProcedureTypes CDynamicInfEngine::GetProcedureType() const;
```

Discussion

This function returns one of the following inference procedures: `ptFiltering`, `ptSmoothing`, `ptFixLagSmoothing`, `ptViterbi`.

Class C2TBNInfEngine
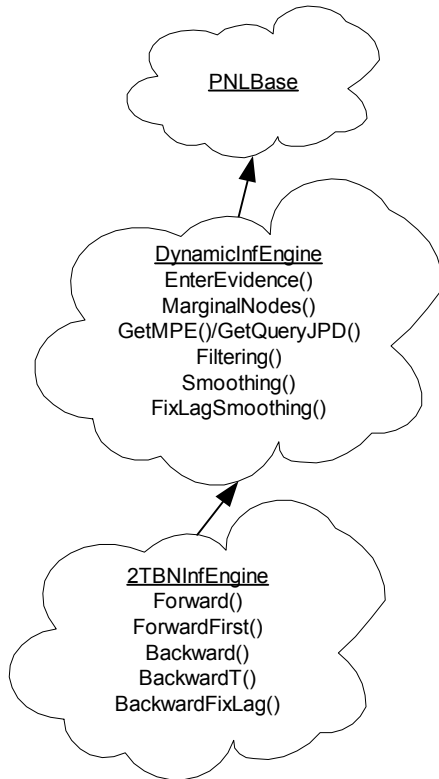
Class C2TBNInfEngine is a superclass for all dynamic inference engine classes, which use forward-backward operations between slices. With such structure of classes an inference procedure (filtering, smoothing, and so on) can be implemented with the help of combination of virtual member functions ForwardFirst, Forward, BackwardT, Backward, which are implemented in derived classes.

Public Member Functions

ForwardFirst

Performs forward operation for prior slice.

```
void C2TBNInfEngine::ForwardFirst( CEvidence* evidence );
```

Arguments

evidence   Pointer to evidence for the prior slice.

Forward

Performs forward operation.

```
void C2TBNInfEngine::Forward( CEvidence* evidence );
```

Arguments

evidence   Pointer to evidence for any but the prior slice.

BackwardT

Performs first backward operation after last forward operation.

```
void C2TBNInfEngine::BackwardT();
```

Backward

Performs backward operation.

```
void C2TBNInfEngine::Backward();
```

BackwardFixLag

Performs sequence of backward operations.

```
void C2TBNInfEngine::BackwardFixLag();
```

Discussion

This member function performs a sequence of backward operations for the fixed-lag smoothing problem, restoring data for the intermediate steps. The number of operations is equal to the value of the lag.

Class C1_ 5SliceInfEngine
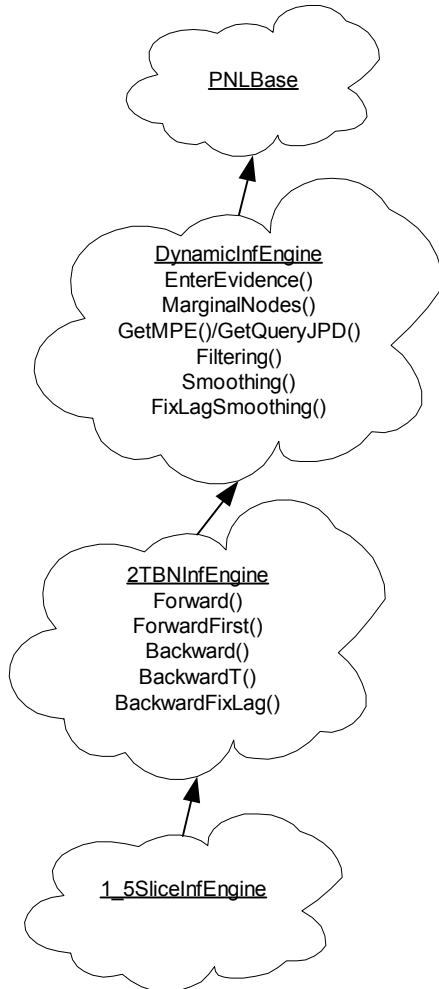
This class is basic for all inference procedures that carry out forward-backward operations between 1.5 slices.

Class C1_5SliceJTreeInfEngine

PNLBase

DynamicInfEngine
EnterEvidence()
MarginalNodes()
GetMPE()/GetQueryJPD()
Filtering()
Smoothing()
FixLagSmoothing()

2TBNInfEngine
Forward()
ForwardFirst()
Backward()
BackwardT()
BackwardFixLag()

1_5SliceInfEngine

1_5SliceJtreeInfEngine
Create()
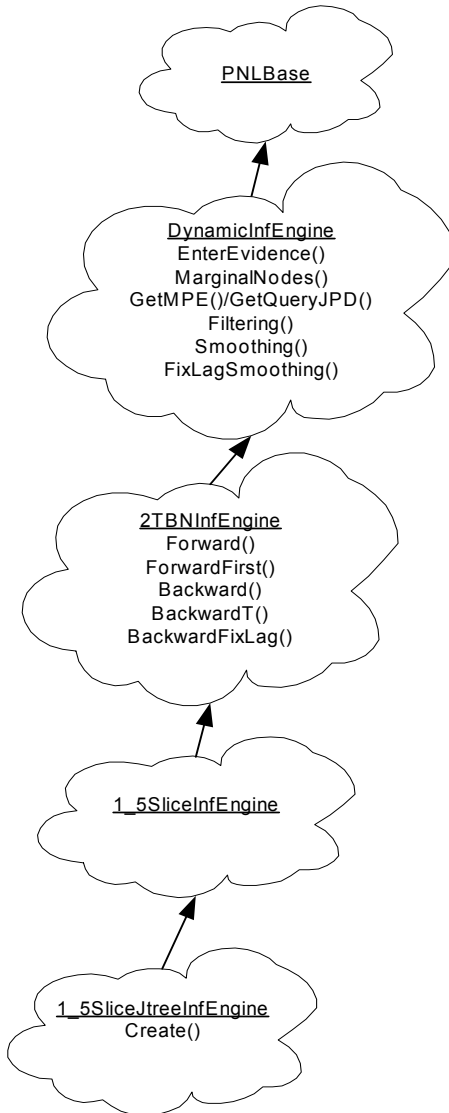
Create

Creates class object.

```
C1_5SliceJtreeInfEngine * C1_5SliceJtreeInfEngine::Create( const
    CDynamicGraphicalModel *pGrModel );
```

Arguments

pGrModel   Pointer to the dynamic graphical model for which inference
                              engine is created.

Class CBKInfEngine

PNLBase

DynamicInfEngine
EnterEvidence()
MarginalNodes()
GetMPE()/GetQueryJPD()
Filtering()
Smoothing()
FixLagSmoothing()

2TBNInfEngine
Forward()
ForwardFirst()
Backward()
BackwardT()
BackwardFixLag()

1_5SliceInfEngine

BKInfEngine
Create()
CheckClustersValidity()
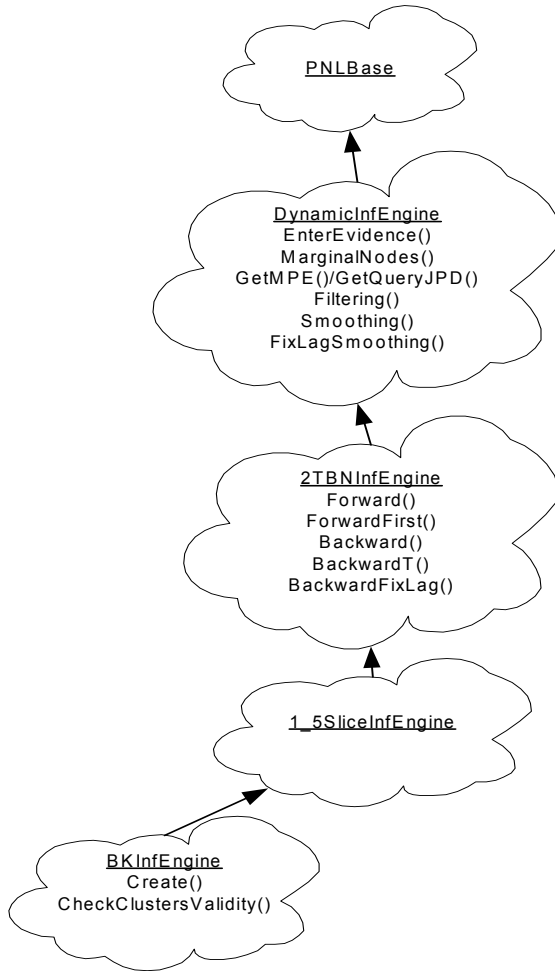
Create

Creates class object.

```
static CBKInfEngine* CBKInfEngine::Create( const CDynamicGraphicalModel
    *pGrModel, bool isFF = true );
static CBKInfEngine* CBKInfEngine::Create( const CDynamicGraphicalModel
    *pGrModel, intVecVector& clusters );
```

Arguments

pGrModel	Pointer to the Dynamic graphical model for which inference engine is created.
isFF	Flag of factorization. The 'true' value means that the inference is fully factorized, that is, each node belongs to a separate cluster. The 'false' value means that the inference is exact, so that nodes lie in one clique.
clusters	Array of nodes belonging to one cluster.

CheckClustersValidity

Checks validity of clusters.

```
static bool CBKInfEngine::CheckClustersValidity( intVecVector& clusters,
    intVector& interfNds );
```

Arguments

clusters	Vector of vectors of inference nodes. An inference node belongs only to one cluster.
interfNds	Vector of inference nodes.

intel®

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 fall into one and the same 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 they belong to the same cluster the inference is exact.

Class C2TPFInfEngine

Class `C2TPFIngEngine` implements the particle-based inference engine for dynamical `BNet` models.

Create

Creates particle-based inference for DBN model.

```
static C2TPFInfEngine * Create (const CDynamicGraphicalModel *pGrModelIn, int
    nParticleCount = 400);
```

Arguments

pGrModelIn	Dynamic graphical model of DBN.
nParticleCount	Number of particles.

SetParemeter

Sets parameters of `2TPFInfEngine`.

```
void SetParemeter(  int nParticleCount = 400, int nLowThreshold = 30);
```

Arguments

nParticleCount	Number of particles.

intel

nLowThreshold   Minimum number of effective particles.

Discussion

This function allows you to change parameters of `2TPFInfEngine`.

InitSlice0Particles

Initializes particles of first slice.

```
void InitSlice0Particles(CEvidence* pSlice0Evidence = NULL);
```

Arguments

pSlice0Evidence   Observed evidence.

EnterEvidence

Enters observed evidence.

```
void EnterEvidence( const CEvidence *pEvidenceIn );
```

Arguments

pEvidenceIn   Observed evidence of the current slice.

Discussion

This function provides observed evidence of the current slice for `2TPFInfEngine`, which updates the particle states and weights by `LWSampling`.

LWSampling

Updates particle states using obesrved evidence.

```
void  LWSampling(const CEvidence* pEvidenceIn = NULL);
```

Arguments

pEvidenceIn   Observed evidence of the current slice.

Discussion

This function updates particles in the current slice. Unlike `EnterEvidence` it cannot update particle weights by itself. To input particle weights you call `EnterEvidenceProbability`.

EnterEvidenceProbability

Inputs likelihood ω of observations in slice.

```
void  EnterEvidenceProbability( floatVector *pEvidenceProbIn );
```

Arguments

pEvidenceProbIn   The vector of particle weights

Discussion

This function calculates particle weights under observance automatically.This interface allows you to calculate likelihood ω of observations easily.

GetCurSamples

Returns states of all current particles.

```
pEvidencesVector* GetCurSamples( );
```

Arguments

*pEvidencesVector**   The returned pointer of particle states.

Discussion

This function returns node values of all particles and allows you to calculate the likelihood ωof observations.

GetParticleWeights

Gets weights of all current particles.

```
floatVector*      GetParticleWieghts( );
```

Arguments

*floatVector**   The returned pointer of particle weights.

Estimate

Estimates real values of nodes in current slice.

```
void Estimate(CEvidence *pEstimate);
```

Arguments

pEstimate   Estimated nodes provided for input.

Discussion

This function estimates real values (states) of nodes by all generated particles. It is similar to `GetMPE()` but more convenient because ot does not call `MarginalNodes()`.

GetNeff

Gets number of current effective particles.

```
float          GetNeff();
```

Arguments

float   The current number of effective particles.

Discussion

This function returns the number of current effective particles. If the number is too small, you should re-initialize the particle states and the particle weights. Set by default, `LWSampling()` detects the number of effective particles and re-initializes the particle weight, if it is too small, automatically.

Learning Engines

Figure 3-5   Structure of learning engines
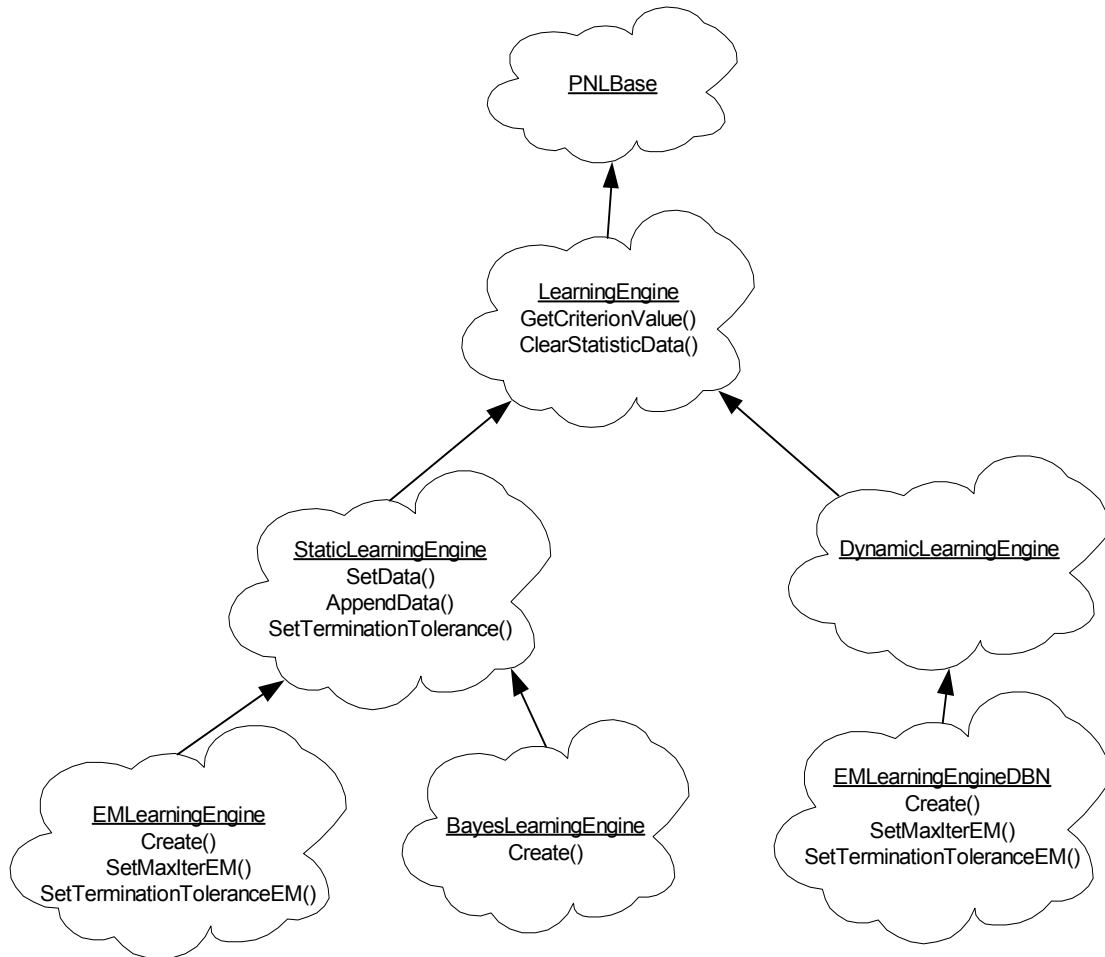
Class CLearningEngine

Learn

Performs learning.

```
virtual void CLearningEngine::Learn()=0;
```

Discussion

This function trains a graphical model by using the set data. In parameter learning the function upgrades factors using given evidences. In structure learning the function creates a new graphical model.

GetCriterionValue

Returns array of criterion values used in learning.

```
virtual float CLearningEngine::GetCriterionValue( int *numOfValues, const
    float **value ) const;
virtual float CStaticLearningEngine::GetScore( floatVector* value ) const;
```

Arguments

numOfValues   Returned parameter. Number of criterion values.

value   Returned parameter. Array of criterion values.

Discussion

This function returns numeric values of a criterion that is to be maximized during model learning.

ClearStatisticData

Clears statistical data.

```
inline void CLearningEngine::ClearStatisticData();
```

Class CStaticLearningEngine

SetData

Sets statistical data for learning.

```
virtual void CStaticLearningEngine::SetData( int size, const CEvidence* const*
    evidences );
virtual void CStaticLearningEngine::SetData(  const pConstEvidenceVector&
    evidences );
```

Arguments

size   Number of evidences.

evidences   Array of pointers to `CEvidence` objects.

Discussion

This function sets statistical data for learning. When the function is called all the prior statistical data is deleted.

AppendData

Appends evidence information.

```
virtual void CStaticLearningEngine::AppendData( int size, const CEvidence*
    const* evidences );
virtual void CStaticLearningEngine::AppendData(  const pConstEvidenceVector&
    evidences );
```

Arguments

size	Array size, equal to the number of rows in the new table of data.
evidences	Array of pointers to CEvidence objects.

Discussion

This function appends a set of evidences to the previously obtained data.

SetMaxIterIPF

Sets maximum iteration depth for Iterative Proportional Fitting.

```
void CStaticLearningEngine::SetMaxIterIPF( int maxIter = 10 );
```

Arguments

maxIter	Maximal number of iterations in Iterative Proportional Fitting.

Discussion

This function is used in Markov networks training and sets the maximal iteration depth allowed for IPF.

SetTerminationToleranceIPF

Sets exit condition for IPF.

```
void CStaticLearningEngine::SetTerminationToleranceIPF( float precision =
    0.001f );
```

Arguments

precision   Precision.

Discussion

This function sets the exit condition for IPF. IPF is over if the difference between clique parameters at the current step and at the previous step does not exceed a given value (*precision*).

GetStaticModel

Returns graphical model.

```
virtual inline CStaticGraphicalModel* CLearningEngine::GetStatisModal() const;
```

Class CEMLearningEngine

Class `CEMLearningEngine` is used in the learning of Bayesian networks with discrete or multivariate Gaussian nodes and in the learning of Markov networks with discrete nodes. The learning is based on *Expectation Maximization* (EM) algorithm.

Create

Creates class object.

```
CEMLearningEngine* CEMLearningEngine::Create(CStaticGraphicalModel* pGrModel,
    CInfEngine* pInfEng = NULL);
```

Arguments

pGrModel	Pointer to a graphical model for which the learning engine is created.
pInfEng	Pointer to the inference engine, which is used in the lerning procedure. By default the Junction tree inference engine is used.

SetMaxIterEM

Sets maximum iteration depth for Expectation Maximization.

```
void CEMLearningEngine::SetMaxIterEM( int numOfIter = 30 );
```

Arguments

numOfIter	Maximal iteration depth.

Discussion

This function sets the maximal number of iterations allowed in the learning process.

SetTerminationToleranceEM

Sets termination tolerance.

```
void CEMLearningEngine::SetTerminationToleranceEM( float precision = 0.001f );
```

Arguments

precision   Precision.

Discussion

This function sets the exit condition for EM. EM is over, if the difference between logarithm of likelihood value at the current step and at the previous step does not exceed a given value.

Class CBayesLearningEngine

Class `CBayesLearningEngine` is used to learn `BNets`, where parameters of a CPD are not fixed and have their own probability distributions (see User Guide). The current version of 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 (meaning) of pseudo counts which stand for an imaginary observed number of cases and assume any non-negative values.

Learning updates prior parameters. 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.

Create

Creates class object.

```
CBayesLearningEngine* CBayesLearningEngine::Create( CStaticGraphicalModel*
    pGrModel );
```

Arguments

pGrModel　　　　　Pointer to the graphical model for which the learning engine is
　　　　　　　　　　created.

Discussion

This function creates a class object. It applies only to `BNet` graphical models.

Class CBICLearningEngine

Class `CBICLearningEngine` is used in the learning of a Bayesian network with
discrete nodes in the case 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.

Create

Creates class object.

```
CBICLearningEngine* CBICLearningEngine::Create( CStaticGraphicalModel*
    pGrModel );
```

Arguments

pGrModel　　　　　Pointer to a graphical model for which learning procedure is
　　　　　　　　　　to be carried out.

Discussion

This function creates an object of `CBICLearningEngine` class.

GetGraphicalModel

Returns created graphical model.

```
const CStaticGraphicalModel * CBICLearningEngine::GetGraphicalModel() const;
```

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.

```
void CBICLearningEngine::GetOrder(intVector* reordering) const;
```

Arguments

reordering   Output parameter. Array of values corresponding to node numbers.

Discussion

This function returns the array of integer values. The array values correspond to node numbers in the source graphical model, which is passed as an argument when `CBicLearningEngine` is created, and the output model after the learning.

Class CMIStaticStructLearn
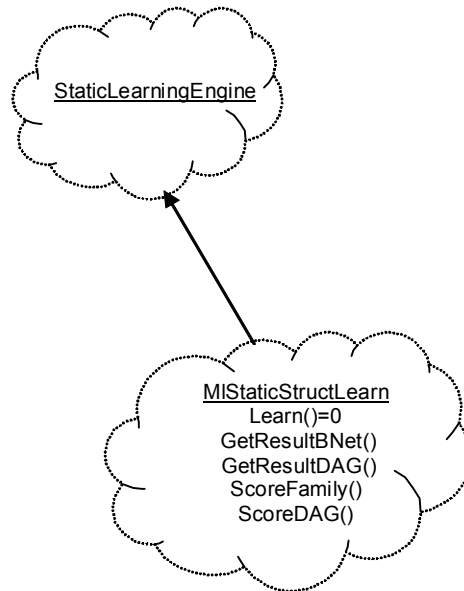
Class `CMIStaticStructLearn` is a virtual class. It defines the interface for all kinds of structure learning of a static `BNet` with complete data.

CreateResultBNet

Generates `BNet` *with structure and parameters.*

```
void  CMlStaticStructLearn::CreateResultBNet(CDAG* pDAG);
```

Arguments

pDAG   Pointer to the learned `DAG`.

Discussion

This function generates the result of learning, that is, a `BNet` object with its parameters and structure. The learned `DAG` sorts the given `DAG` topologically and creates a `BNet` for. The function computes all CPDs for the newly created `BNet`. The new object is stored in `m_pResultBNet`.

GetResultBNet

Gets handle of learned BN.

```
const CBNet* CMlStaticStructLearn::GetResultBNet() const;
```

CreateResultDAG

Gets handle of learned `DAG`

```
const CDAG*  CMlStaticStructLearn::GetResultDAG() const;
```

CreateResultRenaming

Gets handle of node ID mapping of result BN and
result DAG.

```
const int* CMlStaticStructLearn::GetResultRenaming()const;
```

Discussion

This function carries out topological sorting of the given `DAG` on the basis of the learned `DAG` data and creates a `BNet` for the sorted `DAG`. The function computes all CPDs for the created `BNet`. Node IDs of the sorted `DAG` and the resultant `BNet` are different. The renaming provides mapping between them.

ScoreDAG

Computes score of DAG.

```
float CMlStaticStructLearn::ScoreDAG(CDAG* pDAG, floatVector* familyScore);
```

Arguments

pDAG   Pointer to a `DAG` whose score is to be computed.

familyScore   Pointer to a float vector, that stores the score of a family.

Discussion

This function returns the whole score for the `DAG`. Score metric is defined in `m_ScoreType`.

ScoreFamily

Computes family score in DAG.

```
float CMlStaticStructLearn::ScoreFamily(intVector vFamily);
```

Arguments

vFamily   Integer vector. Stores a family in a `DAG`. The last element of the vector is the child node, all other nodes of the vector are parents of this child node.

Discussion

This function returns the score of a family. Score metric is defined in `m_ScoreType`.

SetInitGraphicalModel

Sets initial graphical model from which learning
procedure starts.

```
void CMlStaticStructLearn::SetInitGraphicalModel(CGraphicalModel* pGrModel);
```

Arguments

pGrModel   Pointer to a `CGraphicalModel`.

Discussion

This function sets the initial graphical model for learning. The initial graphical model set as the input *pGrModel* by default is `m_pGrModel`.

Class CMlStaticStructLearnHC
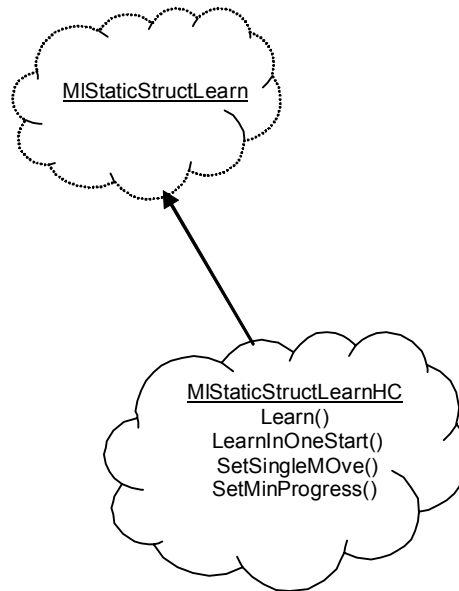
This class carries out the hill-climbing structure learning for a static `BNet`, under the condition that the input data is complete.

Create

Creates class object.

```
CMlStaticStructLearnHC* CMlStaticStructLearnHC::Create(CStaticGraphicalModel*
    pGrModel, ELearningTypes LearnType, EOptimizeTypes AlgorithmType,
    EScoreFunTypes ScoreType, int nMaxFanIn, intVector& vAncestor, intVector&
    vDescent, int nRestarts );
```

Arguments

pGrModel	Pointer to a `CStaticGraphicalModel` instance from which the hill-climbing procedure is to start.
LearnType	Type of learning.
AlgorithmType	Type of algorithm. In the current version of PNL the hill-climbing structure learning is performed only by `StructLearnHC`.
ScoreType	Score type.
nMaxFanIn	Maximum number of parents of a child node.
vAncestor	Ancestor vector. The vector is empty when learning is carried out for a static `BNet`.
vDescent	Descent vector. The vector is empty when learning is carried out for a static `BNet`.
nRestarts	Flag of hill-climbing search procedure restart with random initial structures. If it equals to 1 the procedure does not restart.

SetMinProgress

Sets control condition for search procedure.

```
void CMlStaticStructLearnHC::SetMinProgress(float minProgress);
```

Arguments

minProgress	Small float value.

Discussion

This function sets the control condition for the search procedure. If the improved rate of the best score in the current iteration is less than *minProgress* the search is stopped. The default value of *minProgress* is 0.0001.

SetSingleMove

Sets control condition for search procedure.

```
void CMlStaticStructLearnHC::SetSingleMove(bool SingleMove);
```

Arguments

SingleMove   Boolean value.

Discussion

This function sets control conditions for the search procedure. The 'true' value for *SingleMove* in every iteration of the hill-climbing search means that the change of one edge of the structure is permitted. The 'false' value of the same parameter means that numerous changes to the current structure are permitted. If *SingleMove* is set to 'true' only one edge of the structure is changed, if it is set to 'false' multiply changes to the structure are permitted. The default value is 'true'.

Class CDynamicLearningEngine

Class CDynamicLearningEngine is a superclass for all classes that implement learning for dynamic graphical models. The class contains functions that belong to its child classes.

Public Member Functions

SetData

Sets statistical data for learning.

```
virtual void CDynamicLearningEngine::SetData( int numOfTimeSeries, int
    *numOfSlices, const CEvidence* const* evidences );
```

```
virtual void CDynamicLearningEngine::SetData( const pEvidencesVecVector&
    evidences );
```

Arguments

numOfTimeSeries	Input argument. Array size, which is equal to the number of rows in the table of data.
numOfSlices	Input argument. Pointer to the array of numbers of slices for each time series.
evidences	Input argument. Sets of evidences for DBN slices.

Discussion

This function sets statistical data for learning. Data is represented as an array in which the number of rows is equal to the number of series and the number of elements in a row is equal to the number of time slices for each series.

GetDynamicModel

Returns pointer to model of learning engine.

```
virtual inline CDynamicGraphicalModel* GetDynamicModel() const;
```

Class CEMLearningEngineDBN

Class `CEMLearningEngineDBN` is used in the learning of Dynamic Bayesian Networks. The learning is based on *Expectation Maximization* (EM) algorithm.

Create

Creates class object.

```
static CEMLearningEngineDBN::Create( CDBN* pDBN, CDynamicInfEngine* pInfEng =
    NULL);
```

Arguments

pDBN	Graphical model to be trained.
pInfEng	Pointer to the inference engine which is used in the learning procedure. Set by default, the `C1_5SliceJtreeInfEngine` is used.

SetTerminationToleranceEM

Sets termination tolerance.

```
void CEMLearningEngineDBN::SetTerminationToleranceEM( float precision =
 0.001f );
```

Arguments

precision	Float value of precision, with which the difference between logarithms of likelihoods for two neighboring steps is compared to determine the breakpoint of the learning procedure.

Discussion

This function sets the exit condition for EM. EM procedure stops, if the difference between the logarithm of likelihood value at the current step and at the previous step does not exceed the input value of precision.

SetMaxIterEM

Sets maximum iteration depth for EM.

```
void CEMLearningEngineDBN::SetMaxIterEM(int nIter = 30);
```

Arguments

nIter   Maximal iteration depth.

Discussion

This function sets the maximal number of iterations allowed in the learning process.

Class CMlDynamicStructLearn
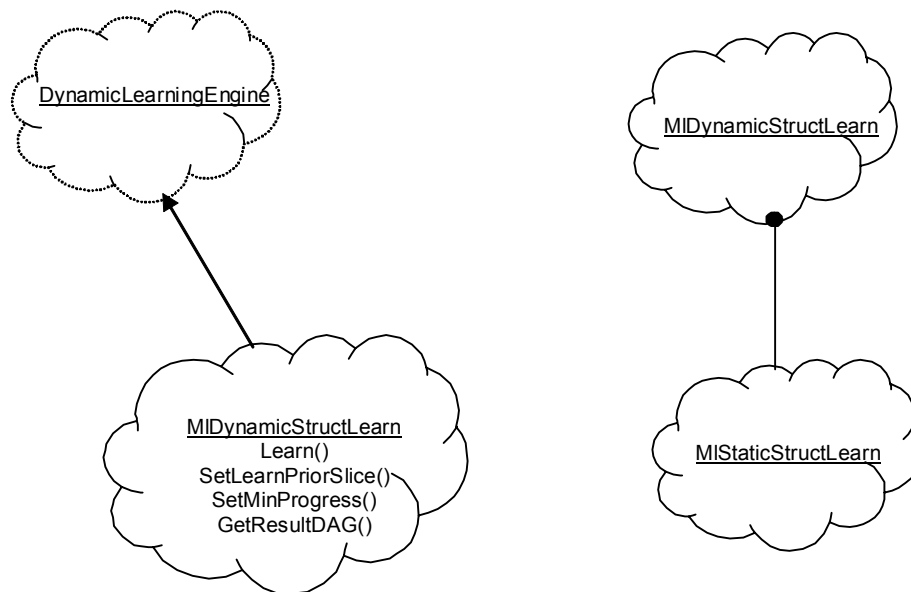
This class allows to learn the structure of DBN under the condition that the input data is complete. In the current version only the hill-climbing search algorithm is available. In the process of learning the algorithm creates one or two `MlStaticStructLearn` objects.

Create

Creates class object.

```
CMlDynamicStructLearn* CMlDynamicStructLearn::Create(CDBN*
    pGrModel,ELearningTypes LearnType, EOptimizeTypes AlgorithmType,
    EScoreFunTypes ScoreType, int nMaxFanIn, int nRestarts, int nMaxIters );
```

Arguments

pGrModel	Pointer to a `CDynamicGraphicalModel` instance from which the hill-climbing procedure is to start.
LearnType	Type of learning which is defined in `ppnlLearningEngine.hpp`.
AlgorithmType	Algorithm type. In the current version the hill-climbing structure learning is performed by `StructLearnHC`.
ScoreType	Score type.
nMaxFanIn	Maximum number of parents of a child node. Includes inter and intra slice-connections.
nRestarts	Flag of the hill-climbing search procedure restart with different random initial structures. If it equals to 1 the search procedure does not restart.
nMaxIters	Maximum number of iterations for the hill-climbing search procedure.

SetMinProgress

Sets control condition for search procedure.

```
void CMlDynamicStructLearn::SetMinProgress(float minProgress);
```

Arguments

minProgress   Small float value.

Discussion

This function sets the control condition for the search procedure. If the improved rate of the best score in the current iteration is less than *minProgress*, the search is stopped. The default value of *minProgress* is 0.0001.

CreateResultDAG

Gets handle of trained DAG

```
const CDAG*  CMlDynamicStructLearn::GetResultDAG() const;
```

SetLearnPriorSlice

Sets different structure for prior slice to be trained.

```
void CMlDynamicStructLearn::SetLearnPriorSlice(bool learnPriorSlice);
```

Arguments

learnPriorSlice   Flag of prior slice structure.

Discussion

This function sets a structure for a trained prior slice which is different from structures of other slices. The 'true' value means that the structure of the slice is different. The default value is 'false'.

Random Number Generation

Random number generation (RNG) is widely used in `PNL`. In the current version of PNL it has the following structure.

The principle element of RNG is the basic generator of uniform distributions (BG), which is a static variable, accessed from any part of the library. To generate samples of non-uniform distributions, corresponding functions also use BG.

The current implementation of RNG is not thread safe because it uses a static variable and does not feature access control. It is designed for internal use. You can reach it through the corresponding API formed of a number of global functions. The current version of PNL generates distributed numbers uniformly and normally.

Linking PNL to Intel Math Kernel Library.

To increase the speed of RNG you may use Intel MKL. The quality of RNG will be different because PNL and MKL use for RNG different maths.

To use MKL in PNL you should:

- define USE_VSL key in the compiler settings;
- link PNL to MKL (see MKL notes);
- rebuild PNL.

pnlSeed

Reinitializes random number generator.

```
void pnlSeed(int s);
```

intel₀

3-295

Arguments

s	Integer that reinitializes the internal state of the basic random number generator.

Discussion

This function reinitializes the random number generator. As RNG is initialised automatically on loading the library this function is called only if there are special need for it, such as, for example, the necessity to perform non-repeatable experiments in different calls of application.

Uniform distributions

pnlRand

Generates random numbers uniformly distributed
over specified numerical interval.

```
int  pnlRand(int left, int right);
void  pnlRand(int numElem, int* vec, int left, int right);
float pnlRand(float left, float right);
void pnlRand(int numElem, float* vec, float left, float right);
double pnlRand(double left, double right);
void pnlRand(int numElem, double* vec, double left, double right);
```

Arguments

left	Left boundary of the interval.
right	Right boundary of the interval.
numElem	Number of random numbers to be generated.
vec	Pointer to the array of random numbers to be generated. The array should be allocated externally and should contain not less than *numElem* elements.

Discussion

This function generates random numbers distributed uniformly over a specified numerical interval. `pnlRand` is a set of overloaded functions for generating uniformly distributed random numbers, both integer and floating point. In case of integer generation the function generates numbers on the interval `[left,right]` with boundaries of the interval included. In case of floating point generation the function generates numbers on the interval `(left,right)` with boundaries of the interval excluded.

There are two types of interfaces of the function: vector and scalar. In case of the scalar interface the function generates and returns only one random number. In case of the vector interface the function fills input array with a given number of random numbers.

Normal distributions

The multivariate Gaussian distribution, otherwise called normal distribution, is commonly expressed in terms of parameters μ and Σ , where μ is an $n \times 1$ vector or the mean vector in PNL terms and Σ is an $n \times n$ symmetric positive definite matrix or the covariance matrix in PNL terms. The formula of the multivariate distribution is:

$$f\langle x \rangle = \frac{1}{\sqrt{2\pi^n det\langle \Sigma \rangle}} \exp\left(-\frac{\langle x - \mu \rangle^T \Sigma^{-1} \langle x - \mu \rangle}{2}\right),$$

where $det\langle \Sigma \rangle$ is the determinant of Σ and n is the dimentionality of the vector x.

pnlRandNormal

Generates normally distributed random numbers.

```
float pnlRandNormal( float mean, float sigma, float variance );

double pnlRandNormal( double mean, double sigma , double variance);

void pnlRandNormal( int numElem, float* vec, float mean, float sigma, float
    variance );
```

```
void pnlRandNormal( int numElem, double* vec, double mean, double sigma ,
    double variance);
```

Arguments

mean	Mean of the normal distribution.
variance	Variance δ^2 of the normal distribution.
sigma	Standard deviation of the normal distribution.
numElem	Number of random numbers to be generated.
vec	Pointer to the array of random numbers to be generated. The array should be allocated externally and is to contain no less than *numElem* elements.

Discussion

This function generates random numbers normally distributed over the interval. `pnlRandNormal` is a set of overloaded functions for generating normally distributed random floating point numbers either with single or with double precision. A normal distribution is calculated according to the following formula:

$$f\langle\ x\rangle = \frac{1}{\sqrt{2\pi}\delta}\exp\langle\ -\frac{\langle\ x+\mu\rangle^2}{2\delta^2}\rangle$$

There are two types of interfaces of the function: vector and scalar. In case of the scalar interface the function generates and returns only one random number. In case of the vector interface the function fills input array with a given number of random numbers.

pnlRandNormal

Generates normally distributed multidimensional vector.

```
void pnlRandNormal(floatVector* vls, floatVector &mean, floatVector &sigma );
void pnlRandNormal(doubleVector* vls,doubleVector &mean,doubleVector &sigma );
```

intel

Arguments

mean	Mean of the multivariate normal distribution.
sigma	Covariance matrix of the normal distribution.
vls	Pointer to the output vector.

Discussion

This function generates a vector from a multivariate normal distribution.

Basic Data Structures

Class Value

This class stores inhomogeneous scalar data.

SetInt

Sets integer value to object.

```
void SetInt(int ivl);
```

Arguments

ivl	Integer value to be set to the Value object.

GetInt

Gets integer value from object.

```
int GetInt() const;
```

SetFlt

Sets float value to object.

```
void SetFlt(float fvl);
```

Arguments

fvl   Float value to be set to the `Value` object.

GetFlt

Gets float value from object.

```
int GetFlt() const;
```

Class pnlVector

```
template<class Type, class Allocator = GeneralAllocator<Type> >
class pnlVector: public std::vector<Type, Allocator>
```

This class is a template intended to store the vector data for PNL. Its implementation is based on `vector` object from STL. For the sake of brevity a number of `pnlVector` specializations was renamed in PNL.

Class valueVector

```
typedef pnlVector<Value>              valueVector;
```

Class pValueVector

```
typedef pnlVector<Value*>             pValueVector;
```

Class valueVecVector

```
typedef pnlVector<valueVector>        valueVecVector;
```

Class pConstValueVector

```
typedef pnlVector<const Value*>       pConstValueVector;
```

Class intVector
```
typedef pnlVector<int>              intVector;
```
Class intPVector
```
typedef pnlVector<int *>            intPVector;
```
Class pConstIntVector
```
typedef pnlVector<const int*>       pConstIntVector;
```
Class intVecVector
```
typedef pnlVector< intVector>       intVecVector;
```
Class intVecPVector
```
typedef pnlVector< intVector* >     intVecPVector;
```
Class floatVector
```
typedef pnlVector<float>            floatVector;
```
Class doubleVector
```
typedef pnlVector<double>           doubleVector;
```
Class floatVecVector
```
typedef pnlVector< floatVector>     floatVecVector;
```
Class boolVector
```
typedef pnlVector<bool>             boolVector;
```
Class nodeTypeVector
```
typedef pnlVector< CNodeType>       nodeTypeVector;
```
Class pNodeTypeVector
```
typedef pnlVector< CNodeType*>      pNodeTypeVector;
```
Class pConstNodeTypeVector
```
typedef pnlVector< const CNodeType*>  pConstNodeTypeVector;
```
Class pEvidencesVector
```
typedef pnlVector< CEvidence*>      pEvidencesVector;
```
Class pConstEvidenceVector
```
typedef pnlVector< const CEvidence*>  pConstEvidenceVector;
```

Class pConstEvidencesVecVector

```
typedef pnlVector<pConstEvidenceVector> pConstEvidencesVecVector;
```

Class pEvidencesVecVector

```
typedef pnlVector< pEvidencesVector>    pEvidencesVecVector;
```

Class pFactorVector

```
typedef pnlVector<CFactor*>             pFactorVector;
```

Class pConstFactorVector

```
typedef pnlVector<const CFactor*>       pConstFactorVector;
```

Class potsPVector

```
typedef pnlVector< CPotential*>         potsPVector;
```

Class potsPVecVector

```
typedef pnlVector< potsPVector>         potsPVecVector;
```

Class pConstCPDVector

```
typedef pnlVector<const CCPD *>         pConstCPDVector;
```

Class pGaussianCPDVector

```
typedef pnlVector<CGaussianCPD *>       pGaussianCPDVector;
```

Error Handling

Class CException

Class `CException` and its child classes are used to generate different types of exceptions.

Figure 3-6   Structure of Error Handling Classes
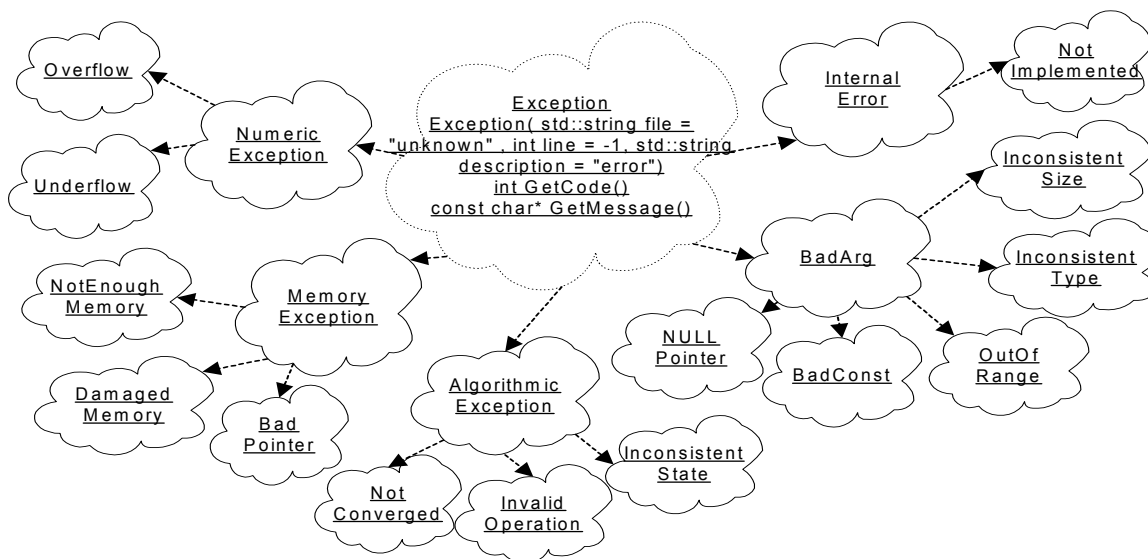
Each class in this hierarchy corresponds to a specific type of exception:

`NumericException` – One of the values exceeds its range, overflow and underflow respectively.

`MemoryException` – Possible memory problems:

`NotEnoughMemory`: available memory size is not enough for allocation.

`DamagedMemory`: the addressed memory location is damaged.

BadPointer: addressing the memory location at the given pointer address is invalid, memory is reserved for private purposes.

AlgorithmicException – Problems arising during algorithmic computations:

NotConverged: the iterative process has not converged within the maximal iteration depth.

InvalidOperation: the function called is not applicable to the calling object.

InconsistentState: the state of the object is not applicable for further computations. For example, a function for the object requires some of the object fields that are not specified.

BadArg – Function receives invalid arguments:

NULLPointer: a null pointer.

BadConst: enumeration fields pass as an argument a number that exceeds the enumeration size.

OutOfRange: numeric field is out of range, for example, negative counter value.

InconsistentType: argument type is inconsistent with the function, for example, when a function of a different class is called.

InconsistentSize: sizes of the input arguments are inconsistent with each, for example, in matrix multiplication.

InternalError: Exceptions related to incompleteness of the library:

NotImplemented: the called member function is not implemented.

The underlying class CException contains the functions that return exceptions. All child classes have no functions of their own, they are introduced only to track all exceptions of a certain type.

GetCode

Returns exception code.

```
int CException::GetCode() const;
```

Discussion

This function returns exception codes given in the file `pgmError.h`.

GenMessage

Generates exception message.

```
void CException::GenMessage();
```

Discussion

This function generates an exception message that contains the exception type, the file name, and the number of the line in the file where this exception was thrown.

GetMessage

Returns pointer to exception message.

```
const char* CEcxeption::GetMessage() const;
```

Discussion

This function returns pointer to an exception message that contains the exception type, the file name, and the number of the line in the file where this exception was thrown.

Log Subsystem

The subtasks of Log Subsystem fall into four categories:

- Decorating output;
- Optimizing output;
- Filtering control;
- Addition/substraction of output devices.

There are three classes to achieve these subtasks:

`Log` – decorates and optimizes the output;

`LogMultiplexor` – filters the output and configuring;

`LogDriver` – outputting device.

Interobject Data Flow

A `Log` object forms a string and passes it to `LogMultiplexor`. The latter passes the string to a corresponding driver.
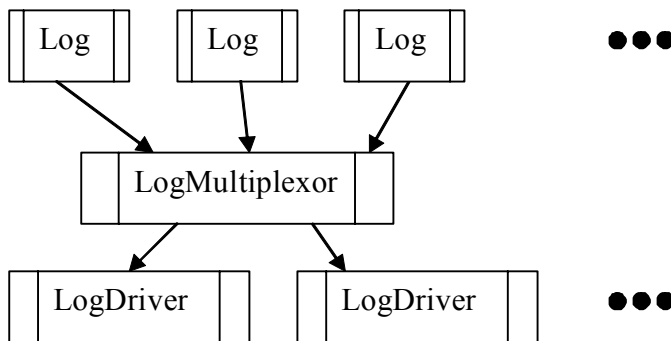
Level is a bitwise combination of the following values:

`eLOG_RESULT`   Resulting value.

`eLOG_SYSERR`   System error.

`eLOG_PROGERR`   Program error.

`eLOG_WARNING`	Warning message.
`eLOG_NOTICE`	Message that does not contain error conditions, but indicates that the information should possibly be handled specially.
`eLOG_INFO`	Information message.
`eLOG_DEBUG`	Message used in debugging.
`eLOG_ALL`	Combination of all the preceding values.

Service is a bitwise combination of the following values:

`eLOGSRV_LOG`	Log system.
`eLOGSRV_EXCEPTION_HANDLING`	Exception handling.
`eLOGSRV_PNL_POTENTIAL`	CPotential and derived classes.
`eLOGSRV_ALL`	Bitwise combination of all the preceding values.
`eLOGSRV_PNL`	Bitwise combination of `eLOGSRV_PNL_*`.

`enum LogDriver::EConfCmd` holds command to configure filtering.

`eADD`	Addition.
`eDELETE`	Subtraction.
`eSET`	Clearing of the old value and setting a new value.

Class Log

Methods of thie class serve as the dumping interface.

Log

Constructs class object.

```
Log::Log( const char *prefix, int level, int service );
```

Arguments

`prefix`	Prefix for each line dumped with this object.

level	Bitwise combination of levels.
service	Bitwise combination of services.

Log

Constructs object by signature.

```
Log::Log( const char *signature );
```

Arguments

signature	Signature string.

Discussion

This function constructs an object by signiture. To use a signature, you should register it with the `Register()` function. If the function finds no signiture, default signiture is used. You can change the default signature by registering it with an empty signature.

Register

Registres Log creating options.

```
void Log::Register( const char *signature ) const;
```

Arguments

signature	Signature string.

flush

Flushes output.

```
void Log::flush();
```

Discussion

Output is flushed automatically either during '\n' output or during Log destruction.

operator <<

Dumps primitive types.

```
inline Log& Log::operator<<(const char*);
inline Log& Log::operator<<(const unsigned char*);
inline Log& Log::operator<<(char);
inline Log& Log::operator<<(unsigned char);
inline Log& Log::operator<<(unsigned long);
inline Log& Log::operator<<(long);
inline Log& Log::operator<<(unsigned short);
inline Log& Log::operator<<(short);
inline Log& Log::operator<<(unsigned int);
inline Log& Log::operator<<(int);
inline Log& Log::operator<<(double);
```

printf

Dumps as printf *function.*

```
void Log::printf( const char* pFmt, ...);
```

Discussion

This function is similar to the printf function from Standard C Library.

Level

Returns current value of level.

```
int Log::Level() const;
```

Service

Returns current value of service.

```
int Log::Service() const;
```

SetLevel

Sets new value to level.

```
void Log::SetLevel(int newLevel);
```

Arguments

newLevel   New value of *level*.

SetService

Sets new value to `service`.

```
void Log::SetService(int newService);
```

Arguments

newService   New value of *service*.

Class LogMultiplexor

Class `LogMultiplexor` is the main commutator. Each of classes `Log` and `LogDriver` is linked with multiplexor. The multiplexor born to be single object. It is used for group configuration of drivers. Generally you do not create or delete this object.

Configure

Configures filtering of logged information.

```
void LogMultiplexor::Configure(EConfCmd command, int level = eLOG_ALL,
int service = eLOGSRV_ALL);
```

Arguments

command   Points at an operation (see description of `EConfCmd`).

level   Bitwise combination of levels.

service   Bitwise combination of services.

Discussion

This function configures filtering of logged information and is applied to all attached drivers.

WriteConfiguration

Dumps out configuration of attached driver to all
other attached drivers.

```
void LogMultiplexor::WriteConfigure() const;
```

Discussion

This function dumps out configuration of an attached driver to all other attached drivers. It is intended for debugging purposes only.

StdMultiplexor

Gets standard multiplexor.

```
static LogMultiplexor& LogMultiplexor::StdMultiplexor();
```

SetStdMultiplexor

Sets standard multiplexor.

```
static void LogMultiplexor::SetStdMultiplexor(LogMultiplexor *pMultiplexor);
```

Arguments

pMultiplexor   Multiplexor to become standard.

Discussion

This function sets the standard multiplexor.

The following member functions of `LogMultiplexor` are used by log subsystem internally.

LogMultiplexor

Constructs multiplexor.

```
LogMultiplexor::LogMultiplexor();
```

AttachDriver

Attaches driver to multiplexor.

```
int LogMultiplexor::AttachDriver(LogDriver* pDriver);
```

Arguments

pDriver   Driver to be attached.

Discussion

This function returns the identificator that must be an argument of the corresponding `DetachDriver` call.

AttachLogger

Attaches `Log` *object to multiplexor.*

```
int LogMultiplexor::AttachLogger(Log* pLog);
```

Arguments

pLog   Log to be attached.

Discussion

This function returns the identificator that must be an argument of the corresponding `DetachLogger` call.

DetachDriver

Detaches driver from multiplexor.

```
void LogMultiplexor::DetachDriver(LogDriver* pDriver, int iDriver);
```

Arguments

pDriver   Driver to be detached.

iDriver   Identificator returned by `AttachDriver()`.

DetachLogger

Detaches `Log` *object from multiplexor.*

```
void LogMultiplexor::DetachLogger(Log* pLog, int iLogger);
```

Arguments

pLog   `Log` to be detached.

iLogger   Id returned by `AttachLogger()`.

iUpdate

Gets updated identificator.

```
int LogMultiplexor::iUpdate() const;
```

Discussion

This function gets the updated identificator. The identificator changes when there is a change of the configuration. The latter changes after it is configured or after a driver is attached to/detached from it.

GetBDenyOutput

Returns TRUE *if output is possible with given*
`level` *and* `service`*.*

```
bool LogMultiplexor::GetBDenyOutput(int *piUpdate, int level, int service);
```

Arguments

`piUpdate`	Pointer to integer, where update id will be stored.
`level`	Bitwise combination of levels.
`service`	Bitwise combination of services.

Discussion

The function returns 'true' if the output is allowed within the given `level` and `service`, returns 'false' otherwise.

Returned value is valid if the 'update id' is not changed.

DriverReconfigured

Notifies multiplexor that driver was reconfigured.

```
void LogMultiplexor::DriverReconfigured(LogDriver *pDriver);
```

Arguments

pDriver   Driver was reconfigured.

WriteString

Writes string to drivers that allow output with
given `level` *and* `service`.

```
void LogMultiplexor::WriteString(int level, int service, const char* pStr, int
    strLen = -1);
```

Arguments

level   Bitwise combination of levels.

service   Bitwise combination of services.

pStr   String.

strLen   Length of the string. The length is calculated if it equals to - 1.

Discussion

This function writes a string to the drivers that allow output with the given *level* and *service*. This function is called by `Log::WriteString`.

Class LogDriver

Class `LogDriver` is a basic class. It the interface for dumping out logged strings.

Configure

Configures filtering of logged information.

```
virtual void LogDriver::Configure(EConfCmd command, int level = eLOG_ALL,
int service = eLOGSRV_ALL);
```

Arguments

command	Type of a set operation (see description of `EConfCmd`).
level	Bitwise combination of levels.
service	Bitwise combination of services.

isAllowedWriting

Returns true if writing is allowed by filtering for given level and service.

```
bool LogDriver::isAllowedWriting(int level, int service) const;
```

Arguments

level	Bitwise combination of levels.
service	Bitwise combination of services.

LogDriver

Constructs object.

```
LogDriver::LogDriver(int levelMask = 0, int serviceMask = 0)
```

Arguments

levelMask   Bitwise combination of levels.

serviceMask   Bitwise combination of services.

Discussion

LogDriver is a purely virtual class.

WriteConfigure

Dumps out configuration of driver to dumping device.

```
void LogDriver::WriteConfigure(const char *prefix, const LogDriver &driver);
```

Arguments

prefix   Prefix for each line.

driver   Driver whose configuration is dumped.

WriteString

Writes string.

```
virtual void LogDriver::WriteString(const char* pStr, int strLen = -1);
```

Arguments

pStr   String.

strLen   Length of the string. The length is calculated if it equals to -1.

Discussion

This function writes a string. This function is designed for the internal use.

Class LogDrvStream

This class is the implementation of the `LogDriver` class which is associated with a stream.

LogDrvStream

Constructs object by stream.

```
LogDrvStream::LogDrvStream(std::ostream* pStream, int levelMask = 0, int
    serviceMask = 0);
```

Arguments

pStream	Pointer to the stream.
levelMask	Bitwise combination of levels.
serveceMask	Bitwise combination of services.

Discussion

This function constructs a class object by a stream.The stream is not deleted after theobject is deleted.

LogDrvStream

Constructs object by file name.

```
LogDrvStream::LogDrvStream(const char *pFilename, int levelMask = 0, int
    serviceMask = 0);
```

Arguments

bafflement	Name of the file.
levelMask	Bitwise combination of levels.

serviceMask   Bitwise combination of services.

Discussion

This function creates a class object by the file name. The stream is created by the file name and is deleted after deleting the object.

Redirect

Redirects dumping to file.

```
void LogDrvStream::Redirect(const char *fname);
```

Arguments

fname   Name of the file.

Discussion

This function redirects dumping to a file. The stream is created by the file name and is deleted after deleting the object.

Redirect

Redirects dumping to stream.

```
void LogDrvStream::Redirect(std::ostream* pStream);
```

Arguments

pStream   Stream.

Discussion

This function redirects dumping to a stream. The stream is not deleted after the object is deleted.

WriteString

Implements `LogDriver` *method.*

```
virtual void LogDrvStream::WriteString(const char* pStr, int strLen = 0);
```

Arguments

pStr   String.

strLen   Length of the string (the length is calculated if it equals to - 1).

Discussion

This function implements the `LogDriver` method. It is intended for use in the logging subsystem only.

Class LogDrvSystem

This class is the implementation of the `LogDriver` class which is associated with a stream. It resembles the `LogDrvStream` but is configured through `ConfigureSystem()`, not through `Configure()`.

LogDrvSystem

Constructs object by stream.

```
LogDrvSystem::LogDrvSystem(std::ostream* pStream, int levelMask = 0, int
    serviceMask = 0);
```

Arguments

pStream   Pointer to the stream.

levelMask   Bitwise combination of levels.

serviceMask   Bitwise combination of services.

LogDrvSystem

Constructs object by file name.

```
LogDrvSystem::LogDrvSystem(const char *pFilename, int levelMask = 0, int
    serviceMask = 0);
```

Arguments

pFilename	Name of the file.
levelMask	Bitwise combination of levels.
serviceMask	Bitwise combination of services.

ConfigureSystem

Configures filtering of logged information.

```
void LogDrvSystem::ConfigureSystem(EConfCmd command, int level = eLOG_ALL,
int service = eLOGSRV_ALL);
```

Arguments

command	Type of a set operation (see description of `EConfCmd`).
level	Bitwise combination of levels.
service	Bitwise combination of services.

Discussion

This function configures filtering of logged information. This function is identical to `LogDriver::Configure`.

Configure

Performs no action, intended for
`LogMultiplexor::Configure` *only.*

```
virtual void LogDrvSystem::Configure(EConfCmd command, int level = eLOG_ALL,
int service = eLOGSRV_ALL);
```

Arguments

command	Type of a set operation.(see description of `EConfCmd`).
level	Bitwise combination of levels.
service	Bitwise combination of services.

Saving Models to File/Loading Models from File

Class `CContextPersistence` serves as the interface for saving a model to a file and for loading a model from a file. You can save following types of models: `BNet`, `DBN`, `MNet`, `MRF2`, if they do not contain one or more tree distribution functions. You cannot save following types of models: `FactorGraph`, `JunctionTree`.

- To save a model to a file put it into a `CContextPersistence` object by calling `Put` and call `SaveAsXML` indicating the name of the file.

Example 3-1   Saving a model to a file

```
CContextPersistence xmlContext;

xmlContext.Put(pGraphicalModel, "MyModel");

if(!xmlContext.SaveAsXML("myFavoriteObjects.xml"))
{
    // something goes wrong - can't create file, disk full or …
}
```

To load a model from the file call the `LoadXML` function and request the context objects by their names.

Example 3-2   Loading model from file

```
CContextPersistence xmlContext;

if(xmlContext.LoadXML("myFavoriteObjects.xml"))
{
    // can't open file or bad file content
}

CBNet *pGrModel = static_cast<CBNet*>(xmlContext.Get( "MyModel"));
```

Class CContextPersistence

This class serves as the external interface for the subsystem of saving and loading.

CContextPersistence

Creates class object.

```
CContextPersistence::CContextPersistence();
```

SaveAsXML

Saves object to XML file.

```
bool CContextPersistence::SaveAsXML(const std::string &filename) const;
```

Arguments

filename   Name of the file where an object is to be saved.

Discussion

This function saves a contextual object into a XML file. The function returns 'true' if saving is successful, returns 'false' otherwise.

LoadXML

Loads object from XML file.

```
bool CContextPersistence::LoadXML(const std::string &filename);
```

Arguments

filename   Name of the file from which the object is to be loaded.

Discussion

This function loads a PNL object from an XML file. The function returns 'true' if loading is successful, returns 'false' otherwise.

Class CContext

This is a basic class for all contexts. Stores the tree corresponding to object structures. Traverses a tree of objects.

CContext

Creates class object.

```
CContext::CContext();
```

Put

Puts object into context.

```
void CContext::Put(CPNLBase *pObj, const char *name, bool bAutoDelete =
    false);
```

Arguments

pObj	Pointer to the object.
name	Object name.
bAutoDelete	Flag of automatic deletion. If it is set to 'true' The object is to be deleted on deletion of the context.

Discussion

This function puts the object into the context by name. The object is requested from the context by the name through the `Get()` function.

Get

Returns pointer to object found by name.

```
CPNLBase *CContext::Get (const char *name);
```

Arguments

name	Name of the object.

Discussion

This function requests for a contextual object by name and returns the pointer to the first object found. If there is no object in the context the function returns NULL.

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.

Index

intel