

# **SMILE: Structural Modeling, Inference, and Learning Engine**

## **Programmer's Manual**

**Version 1.2.1.R4, Built on 2/27/2019**  
**BayesFusion, LLC**

This page is intentionally left blank.

<b>1. Introduction</b>	<b>7</b>
<b>2. Licensing</b>	<b>9</b>
<b>3. Compiler and Linker Options</b>	<b>11</b>
3.1 Visual C++ .....	12
3.2 gcc .....	13
<b>4. Hello, SMILE!</b>	<b>15</b>
4.1 Success/Forecast model .....	16
4.2 The program .....	16
4.3 hello.cpp .....	18
4.4 VentureBN.xdsl .....	19
<b>5. Using SMILE</b>	<b>21</b>
5.1 Main header file .....	22
5.2 Naming conventions .....	22
5.3 Error handling .....	22
5.4 Networks, nodes and arcs .....	23
5.4.1 Network .....	23
5.4.2 Nodes .....	23
5.4.3 Arcs .....	24
5.5 Anatomy of a node .....	25
5.5.1 Multidimensional arrays .....	25
5.5.2 Node definition .....	26
5.5.3 Node value & evidence .....	27
5.5.4 Other node attributes .....	28
5.6 Input and output .....	29
5.7 Inference .....	29
5.8 User properties .....	30
5.9 Cases .....	31
5.10 Sensitivity analysis .....	31
5.11 Deterministic nodes .....	35
5.12 Canonical nodes .....	36
5.12.1 Noisy-MAX .....	36
5.12.2 Noisy-Adder .....	38
5.13 Influence diagrams .....	39

<b>5.14</b>	<b>Dynamic Bayesian networks</b>	<b>40</b>
5.14.1	Unrolling	40
5.14.2	Temporal definitions	43
5.14.3	Temporal evidence	43
5.14.4	Temporal beliefs	44
<b>5.15</b>	<b>Continuous models</b>	<b>44</b>
5.15.1	Equation-based nodes	44
5.15.2	Continuous inference	45
<b>5.16</b>	<b>Hybrid models</b>	<b>47</b>
<b>5.17</b>	<b>Datasets</b>	<b>48</b>
5.17.1	Text file I/O	48
5.17.2	Discrete and continuous variables	50
5.17.3	Generating data from network	51
5.17.4	Discretization	52
<b>5.18</b>	<b>Learning</b>	<b>52</b>
5.18.1	Learning network structure	52
5.18.2	Learning network parameters	54
5.18.3	Validation	55
<b>6.</b>	<b>Tutorials</b>	<b>57</b>
<b>6.1</b>	<b>main.cpp</b>	<b>58</b>
<b>6.2</b>	<b>Tutorial 1: Creating a Bayesian Network</b>	<b>59</b>
6.2.1	tutorial1.cpp	61
<b>6.3</b>	<b>Tutorial 2: Inference with a Bayesian Network</b>	<b>63</b>
6.3.1	tutorial2.cpp	66
<b>6.4</b>	<b>Tutorial 3: Exploring the contents of a model</b>	<b>68</b>
6.4.1	tutorial3.cpp	70
<b>6.5</b>	<b>Tutorial 4: Creating the Influence Diagram</b>	<b>72</b>
6.5.1	tutorial4.cpp	74
<b>6.6</b>	<b>Tutorial 5: Inference in an Influence Diagram</b>	<b>76</b>
6.6.1	tutorial5.cpp	77
<b>6.7</b>	<b>Tutorial 6: Dynamic model</b>	<b>80</b>
6.7.1	tutorial6.cpp	82
<b>6.8</b>	<b>Tutorial 7: Continuous model</b>	<b>84</b>
6.8.1	tutorial7.cpp	87
<b>6.9</b>	<b>Tutorial 8: Hybrid model</b>	<b>90</b>
6.9.1	tutorial8.cpp	92
<b>7.</b>	<b>Reference Manual</b>	<b>97</b>

7.1	DSL_network .....	98
7.2	DSL_node .....	108
7.3	DSL_nodeDefinition .....	109
7.4	DSL_noisyMAX .....	111
7.5	DSL_noisyAdder .....	112
7.6	DSL_truthTable .....	114
7.7	DSL_equation .....	115
7.8	DSL_nodeValue .....	116
7.9	DSL_valEqEvaluation .....	120
7.10	DSL_Dmatrix .....	121
7.11	DSL_dataset .....	125
7.12	DSL_dataGenerator .....	129
7.13	DSL_validator .....	130
7.14	DSL_em .....	133
7.15	DSL_bs .....	135
7.16	DSL_pc .....	137
7.17	DSL_tan .....	138
7.18	DSL_abn .....	139
7.19	DSL_nb .....	140
7.20	DSL_bkgndKnowledge .....	141
7.21	DSL_pattern .....	141
7.22	DSL_progress .....	143
7.23	Equations .....	143
7.23.1	Operators .....	143
7.23.2	Probability Distributions .....	145
7.23.3	Arithmetic Functions .....	151
7.23.4	Combinatoric Functions .....	153
7.23.5	Trigonometric Functions .....	154
7.23.6	Hyperbolic Functions .....	154
7.23.7	Logical/Conditional functions .....	155
<b>8.</b>	<b>Acknowledgments</b>	<b>157</b>
	<b>Index</b>	<b>0</b>

This page is intentionally left blank.

# Introduction

## 1 Introduction

---

Welcome to SMILE Programmer's Manual, version 1.2.1.R4, built on 2/27/2019. For the most recent version of this manual, please visit <https://support.bayesfusion.com/docs/>.

SMILE (Structural Modeling, Inference, and Learning Engine) is the software library for performing Bayesian inference, written in C++, available in compiled form for a variety of platforms, including multiple versions of Visual C++ for 32-bit and 64-bit Windows, macOS (formerly known as OS X) and Linux. We assume that the reader has basic knowledge of the C++ language. We also provide wrappers exposing SMILE functionality to programs written in Java (jSMILE), Python (PySMILE), .NET (Smile.NET) or using COM (SMILE.COM, targeted for use with Microsoft Excel). However, this manual is for C++ programmers and does not cover the interoperability issues. The documentation for wrappers is provided in the separate manual.

If you are new to SMILE and would like to start with an informal, tutorial-like introduction, please start with the [Hello SMILE!](#)<sup>[16]</sup> section. If you are an advanced user, please browse through the Table of Contents or search for the topic of your interest.

This manual refers to a good number of concepts that are assumed to be known to the reader, such as probability, utility, decision theory and decision analysis, Bayesian networks, influence diagrams, etc. Should you want to learn more about these, please refer to GeNIe manual. SMILE is GeNIe's Application Programmer's Interface (API) and practically every elementary operation performed with GeNIe translates to calls to SMILE methods. Being familiar with GeNIe may prove extremely useful in learning SMILE. Understanding some of SMILE's functionality may be easier when performed interactively in GeNIe.

The source code of SMILE is proprietary. If the operating system and/or compiler you want to use SMILE with is not in the list of binaries available at our software download website at <https://download.bayesfusion.com>, contact us.

If you have used SMILE before SMILE 1.2 (released in November 2017), here are three important changes introduced in version 1.2:

- programs linking with SMILE are required to define and initialize the specific licensing key variables; see the [Licensing](#)<sup>[10]</sup> section below
- the learning library (SMILearn) is included in SMILE now (so there's just one library). The functionality previously included in the `smilearn.h` header file is now accessed through `smile.h`; `smilearn.h` is no longer available.
- the global `ErrorH` object was replaced by the global `DSL_errorH` function.



# Licensing

## 2 Licensing

---

SMILE library is a commercial product that requires a development license to use. There are two types of development licenses: Academic and Commercial. Academic license is free of charge for research and teaching use by those users affiliated with an academic institution. All other use requires a commercial license, available for purchase from BayesFusion, LLC.

Deployment of SMILE library, i.e., embedding it into user programs, requires a deployment license. There are two types of deployment licenses: Server license, which allows a program linked with SMILE to be deployed on a computer server, and end-user program license, which allows distribution of user programs that include SMILE. Please contact BayesFusion, LLC, for details of the licenses and pricing.

The licensing system is implemented as two variables (DSL\_LIC1 and DSL\_LIC2), which must be declared and initialized in order to successfully link your program with SMILE. The definitions for these variables are included in your license key header file (usually `smile_license.h`). **This file is not included in SMILE distribution**, it is personalized by BayesFusion, LLC, for you or your organization. 6-month academic and free 30-day evaluation license keys can be obtained directly at <https://download.bayesfusion.com>.

6-month academic license should be sufficient for most coursework. If you need SMILE Academic for a research project and would like a longer license, please email us at [support@bayesfusion.com](mailto:support@bayesfusion.com) from your university email account.

# Compiler and Linker Options

## 3 Compiler and Linker Options

SMILE is distributed as a C++ library along with the set of header files. To compile your program, you need to include the `smile.h` header file. You also need to ensure that your binary is linked with SMILE.

BayesFusion, LLC can provide binaries built with settings different from these described below on request.

### 3.1 Visual C++

We support compiled SMILE library for multiple versions of Visual C++. Each zip file contains libraries for both x86 (32-bit) and x64 (64-bit) architectures. In addition, for each architecture, we provide three libraries built to use with different versions of Visual C++ runtime. For example, SMILE for Visual Studio 2015 includes the following libraries:

- `smile_dbg_vc_140x64.lib`: debug build for dynamic debug CRT, 64-bit
- `smile_dbg_vc_140x86.lib`: debug build for dynamic debug CRT, 32-bit
- `smile_vc_140x64.lib`: release build for static CRT, 64-bit
- `smile_vc_140x86.lib`: release build for static CRT, 32-bit
- `smile_dyn_vc_140x64.lib`: release build for dynamic (DLL) CRT, 64-bit
- `smile_dyn_vc_140x86.lib`: release build for dynamic (DLL) CRT, 32-bit

The number '140' in the list of libraries above corresponds to Microsoft's internal toolkit version, which is 140 for Visual C++ included in the Visual Studio 2015 product.

**Only one of these libraries needs to be linked with your executable. The `smile.h` header contains `#pragma` directives which automatically select the proper library depending on the selected architecture and current build configuration of your project.**

NOTE: due to auto-linking implemented in `smile.h`, you **don't need** to add `smile*.lib` files to the list of libraries in project's linker settings. Doing that is very likely to cause linker errors.

However, you still need to tell linker the location of the directory containing SMILE's `.lib` files. There are two independent ways to do that:

1. Go to Project Settings | Linker | General and add SMILE directory to ‘Additional Library Directories’, or
2. Go to Project Settings | VC++ Directories and add SMILE directory to ‘Library Directories’.

There are also two independent ways of specifying the SMILE include directory:

1. Go to Project Settings | C++ | General and add SMILE directory to ‘Additional Include Directories’, or
2. Go to Project Settings | VC++ Directories and add SMILE directory to ‘Include Directories’.

Starting with version 1.2.0 released in November 2017, SMILE no longer requires `_SECURE_SCL=0` to be defined in preprocessor settings.

## 3.2 gcc

SMILE for Unix-based and Unix-like systems (Linux, FreeBSD) is compiled with the gcc toolchain. SMILE for Apple operating systems (iOS and macOS) is compiled with Apple clang. In both cases, we compile public binaries with `-O3` and `-DNDEBUG`.

To use SMILE you’ll need the following on your g++ or clang command line:

- Specify directory containing `libsmile.a` using `-L` option: `-L<smiledir>`
- Specify directory containing `smile.h` using `-I` (uppercase I) option: `-I<smiledir>`
- Add `libsmile.a` to libraries used by your program with `-l` (lowercase L) option: `-lsmile`. Note that the ‘lib’ prefix and ‘a’ suffix are implied.

Example:

We assume that SMILE’s headers and libraries are located in the subdirectory ‘smile’ and your source code is in file(s) with the `.cpp` extension.

```
g++ -DNDEBUG -O3 -I./smile -L./smile -lsmile *.cpp
```

This command compiles the program, then links it with `libsmile.a`. The output executable binary has the default filename `a.out`.

The archive (`.tar.gz`) with SMILE binaries includes the `build.txt` file, which contains the details of the build environment used to compile the library, including the version of the compiler. Please make sure you are using the library compatible with your compiler.



**Hello, SMILE!**

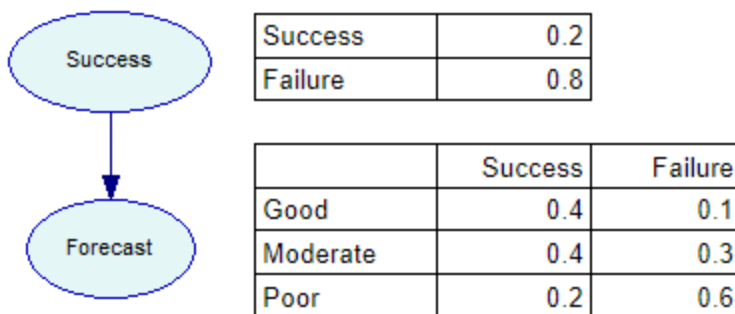
## 4 Hello, SMILE!

In this section, we will show how SMILE can load and use a model created in GeNIe to perform useful work. We will use the model developed in the GeNIe on-line help (Section *Hello GeNIe!*). The model for this problem is available as one of the example networks (file `VentureBN.xdsl`). If you have GeNIe installed, you can copy the file into your working directory. The same file is also included in the zip file containing all source code for tutorials, available from <http://support.bayesfusion.com/docs>. Alternatively, create a file named `VentureBN.xdsl` with any text editor by copying the content of the [VentureBN.xdsl](#)<sup>[19]</sup> section below.

The complete source code of the program is also provided in this chapter in the [hello.cpp](#)<sup>[18]</sup> section.

### 4.1 Success/Forecast model

The model encodes information pertaining to a problem faced by a venture capitalist, who considers a risky investment in a startup company. A major source of uncertainty about her investment is the success of the company. She is aware of the fact that only around 20% of all start-up companies succeed. She can reduce this uncertainty somewhat by asking expert opinion. Her expert, however, is not perfect in his forecasts. Of all start-up companies that eventually succeed, he judges about 40% to be good prospects, 40% to be moderate prospects, and 20% to be poor prospects. Of all start-up companies that eventually fail, he judges about 10% to be good prospects, 30% to be moderate prospects, and 60% to be poor prospects.



### 4.2 The program

We will show how to load this model using SMILE, how to enter observations (evidence), how to perform inference, and how to retrieve the results of SMILE's calculations. The complete source code is included below. Note that you'll need to #include your SMILE license key. See the [Licensing](#)<sup>[10]</sup> section of this manual if you want to obtain your academic or trial license key.



The program starts with redirecting the error and warning messages to the standard output. We don't expect to see any messages. If `VentureBN.xdsl` is not in the current directory, you'll get notified.

```
DSL_errorH().RedirectToFile(stdout);
```

Our network object is declared as local variable, then we read the file. We proceed only if the file loaded correctly.

```
DSL_network net;
int res = net.ReadFile("VentureBN.xdsl");
if (DSL_OKAY != res)
{
    return res;
}
```

We want to set the evidence on the *Forecast* node to *Moderate*. SMILE uses integer handles to identify nodes, so we need to convert known node identifier ("*Forecast*") to handle. Handles are always non-negative; if node identifier is not found in the network the value returned by the `FindNode` method will be less than zero.

```
int handle = net.FindNode("Forecast");
if (handle < 0)
{
    return handle;
}
```

With node handle we can proceed to retrieve the index of the outcome by searching for outcome identifier ("*Moderate*"). Again, if the identifier is not found, the return value from `FindPosition` method will be negative.

```
DSL_node *f = net.GetNode(handle);
int idx = f->Definition()->GetOutcomesNames()->FindPosition("Moderate");
if (idx < 0)
{
    return idx;
}
```

Now we can set the evidence and update the network.

```
f->Value()->SetEvidence(idx);
net.UpdateBeliefs();
```

Note that in the real-world program we would test the status codes returned by both `SetEvidence` and `UpdateBeliefs`. Our sample program only checks for errors caused by missing network file or nodes/outcomes not present in the network.

After network update we can read the posterior probabilities of the *Success* node. We again convert node identifier to handle and iterate over its outcomes, displaying the probability of each outcome in the loop.

```
DSL_node *s = net.GetNode(handle);
const DSL_Dmatrix &beliefs = *s->Value()->GetMatrix();
const DSL_idArray &outcomes = *s->Definition()->GetOutcomesNames();
for (int i = 0; i < outcomes.NumItems(); i++)
{
```

```
    printf("%s=%g\n", outcomes[i], beliefs[i]);
}
```

This ends the source of the program. If you compile and run it, the output should be:

```
Success=0.25
Failure=0.75
```

“Success” and “Failure” are outcomes of the *Success* node.

We will build upon the simple network described in this chapter in the [Tutorials](#)<sup>[58]</sup> section of this manual.

## 4.3 hello.cpp

---

```
// hello.cpp

#include <cstdio>
#include <smile.h>
#include "smile_license.h" // your licensing key

int main()
{
    DSL_errorH().RedirectToFile(stdout);
    DSL_network net;
    int res = net.ReadFile("VentureBN.xdsl");
    if (DSL_OKAY != res)
    {
        return res;
    }

    int handle = net.FindNode("Forecast");
    if (handle < 0)
    {
        return handle;
    }

    DSL_node *f = net.GetNode(handle);
    int idx =
        f->Definition()->GetOutcomesNames()->FindPosition("Moderate");
    if (idx < 0)
    {
        return idx;
    }
    f->Value()->SetEvidence(idx);

    net.UpdateBeliefs();

    handle = net.FindNode("Success");
    if (handle < 0)
    {
        return handle;
    }
}
```

```

DSL_node *s = net.GetNode(handle);
const DSL_Dmatrix &beliefs = *s->Value()->GetMatrix();
const DSL_idArray &outcomes =
    *s->Definition()->GetOutcomesNames();
for (int i = 0; i < outcomes.NumItems(); i++)
{
    printf("%s=%g\n", outcomes[i], beliefs[i]);
}

return DSL_OKAY;
}

```

## 4.4 VentureBN.xdsl

---

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<smile version="1.0" id="VentureBN" numsamples="1000">
  <nodes>
    <cpt id="Success">
      <state id="Success" />
      <state id="Failure" />
      <probabilities>0.2 0.8</probabilities>
    </cpt>
    <cpt id="Forecast">
      <state id="Good" />
      <state id="Moderate" />
      <state id="Poor" />
      <parents>Success</parents>
      <probabilities>
        0.4 0.4 0.2 0.1 0.3 0.6
      </probabilities>
    </cpt>
  </nodes>
  <extensions>
    <genie version="1.0" app="GeNIe 2.1.1104.2"
      name="VentureBN"
      faultnameformat="nodestate">
      <node id="Success">
        <name>Success of the venture</name>
        <interior color="e5f6f7" />
        <outline color="0000bb" />
        <font color="000000" name="Arial" size="8" />
        <position>54 11 138 62</position>
      </node>
      <node id="Forecast">
        <name>Expert forecast</name>
        <interior color="e5f6f7" />
        <outline color="0000bb" />
        <font color="000000" name="Arial" size="8" />
        <position>63 105 130 155</position>
      </node>
    </genie>
  </extensions>
</smile>

```

This page is intentionally left blank.

## Using SMILE

## 5 Using SMILE

---

### 5.1 Main header file

---

To use SMILE you need to include single header file, namely `smile.h`. As described earlier, with Visual C++ this file also takes care of specifying the correct variant of the library (debug vs. release with static CRT vs. release with CRT in DLL).

The second required header, usually named `smile_license.h` contains your licensing information. You need to include this file in exactly one of your `.cpp` files. See the [Licensing](#)<sup>10</sup> section for more details, including information on how to obtain evaluation or academic license from our website.

### 5.2 Naming conventions

---

The names of the majority of SMILE classes and functions start with the `DSL_` prefix and use camel case afterwards. For example:

```
DSL_network  
DSL_node
```

Constants and #defined symbols start with the same `DSL_` prefix followed by all-uppercase letters, for example:

```
DSL_OKAY  
DSL_CPT
```

### 5.3 Error handling

---

Most of the SMILE functions and methods return an integer status code. Negative numbers are used for specific error codes. See `errors.h` in SMILE's distribution directory for the list of codes.

Sometimes SMILE emits the warning or error message. These are stored by default in the global object of the class `DSL_errorStringHandler`, access to which you can get by calling the `DSL_errorH` function. By default, your program is not notified about these messages: you need to query `DSL_errorH`'s stored codes and strings.

Alternatively, your program can redirect these messages to `FILE*` (including `stdout` and `stderr`):

```
DSL_errorH().RedirectToFile(stdout);
```

For the complete control over the messages, use `DSL_errorStringHandler::Redirect` method. Its only argument is a pointer to the instance of the class derived from `DSL_errorStringRedirect`.

```
class MyRedirect : public DSL_errorStringRedirect
{
public:
    void LogError(int code, const char *message)
    {
        // do something with the code and message
    }
};
// ...
MyRedirect myRedir;
DSL_errorH().Redirect(&myRedir);
```

Tutorials included in this manual redirect errors to `stdout`.

## 5.4 Networks, nodes and arcs

---

The most important class defined by SMILE is `DSL_network`. The objects of this class act as containers for nodes and are responsible for node creation and destruction. Nodes and arcs are always created and destroyed by invoking `DSL_network`'s methods.

Nodes are created by the `DSL_network::AddNode` method and destroyed by `DSL_network::DeleteNode`.

The arcs are created by the `DSL_network::AddArc` method and destroyed by `DSL_network::RemoveArc`.

### 5.4.1 Network

To create a network, simply use a default constructor. You can create `DSL_network` as a local variable on the stack, an object on the heap or a member of another class, depending on your specific needs. Many programs will call the `ReadFile` or `ReadString` methods to initialize the content of the network after creating it. The `UpdateBeliefs` method invokes the inference algorithm on the network and sets the node values. The `CalcProbEvidence` method computes the probability of evidence currently set in the network.

### 5.4.2 Nodes

Nodes are represented by the `DSL_node` object. Your program should always use the `DSL_network::AddNode` method to create a node within the network and never use `DSL_node` constructor directly. To delete the node, use `DSL_network::DeleteNode`; never try to use operator `delete`.

Within the network, the nodes are uniquely identified by their handle. The node handle is a non-negative integer, which is preserved when network is copied using copy constructor or operator=. Most of DSL\_network methods dealing with nodes uses node handles as input arguments. In general, the node handles may change when you write the network to file and read it later. This means you should not rely on handles as persistent identifiers.

The values of the handles are not guaranteed to be consecutive or start from any particular value. To iterate over nodes in the network, use DSL\_network::GetFirstNode and GetNextNode:

```
for (int h = net.GetFirstNode(); h >= 0; h = net.GetNextNode(h))
{
    DSL_node *node = net.GetNode(h);
    printf("Node handle: %d, node id: %s\n", h, node->GetId());
}
```

Note the loop exit condition - we stop when GetNextNode returns a negative value.

In addition to handles, each node has an unique (in the context of its containing network), persistent, textual identifier. This identifier is specified as an argument to DSL\_network::AddNode method at node creation (it may be changed later). The identifiers in SMILE start are case-sensitive, start with a letter and contain letters, digits and underscores. Node's identifier can be converted to node handle with a call to DSL\_network::FindNode method.

```
int handle = net.FindNode("myNodeId");
if (handle >= 0)
{
    printf("Handle of myNodeId is: %d\n", handle);
}
else
{
    printf("There's no node with ID=myNodeId\n");
}
```

Note that FindNode has  $O(n)$  complexity, it simply compares its input argument with all node identifiers in the loop. On the other hand, GetNode is  $O(1)$ , as it performs an index lookup in the array.

Nodes may be marked as targets with DSL\_network::SetTarget method. Target nodes are always guaranteed to be updated by the inference algorithm. Other nodes, i.e., nodes that are not designated as targets, may be updated or not, depending on the internals of the algorithm used, but are not guaranteed to be updated. Focusing inference on the target nodes can reduce time and memory required to complete the calculation. When no targets are specified, SMILE assumes that all nodes are of interest to the user.

### 5.4.3 Arcs

SMILE does not define a class representing an arc between nodes. When you call DSL\_network::AddArc method, the internal data structures are updated to keep the relationship between the parent and child node. To remove the arc, call DSL\_network::RemoveArc.



The graph defined by nodes and arcs in `DSL_network` is a directed acyclic graph (DAG) at all times. If your call to `AddArc` would result in a cycle in the graph, the method returns an error code. To check if adding an arc would introduce a cycle, you can call `DSL_network::RemainsAcyclic` method.

To inspect the graph structure, use `DSL_network::GetParents` and `GetChildren` methods:

```
int nodeHandle = ...;
const DSL_intArray& parents = net.GetParents(nodeHandle);
for (int i = 0; i < parents.NumItems(); i++)
{
    printf("Parent %d: %d %s\n", i, parents[i],
           net.GetNode(parents[i]->GetId()));
}
const DSL_intArray& children = net.GetChildren(nodeHandle);
for (int i = 0; i < children.NumItems(); i++)
{
    printf("Child %d: %d %s\n", i, children[i],
           net.GetNode(children[i]->GetId()));
}
```

## 5.5 Anatomy of a node

---

The instance of `DSL_node` object managed by `DSL_network` aggregates other objects representing different aspects of the node. The most important are the definition and value objects. Additionally, the node contains the description attributes which do not affect inference, but determine node's location, color, name, etc.

### 5.5.1 Multidimensional arrays

To represent conditional probability tables (CPTs) SMILE uses `DSL_Dmatrix` class. CPT describes the interaction between a node and its immediate predecessors. The number of dimensions and the total size of a conditional probability table are determined by the number of parents, the number of states of each of these parents, and the number of states of the child node. Essentially, there is a probability for every state of the child node for every combination of the states of the parents. Nodes that have no predecessors are specified by a prior probability distribution table, which specifies the prior probability of every state of the node.

The conditional probability tables are stored as vectors of doubles that are a flattened version of multidimensional tables with as many dimensions as there are parents plus one for the node itself. The order of the coordinates reflects the order in which the arcs to the node were created. The most significant (leftmost) coordinate will represent the state of the first parent. The state of the node itself corresponds to the least significant (rightmost) coordinate.

The image below is an annotated screenshot of GeNIe's node properties window open for the *Forecast* node in the model created in [Tutorial 1](#)<sup>[59]</sup>. *Forecast* has three outcomes and two parents: *Success of the venture* and *State of the economy*, with two and three outcomes, respectively.

Therefore, the total size of the CPT is  $2 \times 3 \times 3 = 18$ . The annotation arrows in the image (not part of the actual GeNIe window) show the ordering of the entries in the linear buffer which DSL\_Dmatrix uses internally. The first (or rather, the zero-th, because all indexes in SMILE are zero-based) element, the one with the value of 0.7 and yellow background, represents  $P(\text{Forecast}=\text{Good} \mid \text{Success of the venture}=\text{Success} \ \& \ \text{State of the economy}=\text{Up})$ . It is followed by the probabilities for *Moderate* and *Poor* outcomes given the same parent configuration. The next parent configuration is Success of the venture=Success & State of the economy=Flat, and so on.

Success of the venture		Success			Failure		
State of the economy		Up	Flat	Down	Up	Flat	Down
▶ Good		0.7	0.65	0.6	0.15	0.1	0.05
Moderate		0.29	0.3	0.3	0.3	0.3	0.25
Poor		0.01	0.05	0.1	0.55	0.6	0.7

In addition to linear indexing, DSL\_Dmatrix allows access to its elements through coordinate system by overloading operator[] and Subscript methods. The coordinates are specified by DSL\_intArray object containing values for each parent and node for which CPT is defined. For example, the element for

- Success of the venture=Failure
- State of the economy=Up
- Forecast=Poor

would have the coordinates [1, 0, 2]. It's linear index is 11. DSL\_Dmatrix provides CoordinatesToIndex and IndexToCoordinates methods for conversion between coordinates and linear indices and vice versa. The NextCoordinates and PrevCoordinates methods can be used to shift the coordinates forward or backward in odometer-like fashion (with the rightmost entry representing node for which CPT is defined moving fastest.)

Note that DSL\_Dmatrix is not used exclusively for CPTs. Other uses of this class in SMILE include (but are not limited to) expected utility tables and marginal probability distributions.

### 5.5.2 Node definition

The definition of the node specifies how it interacts with other nodes in the network. The node definition is written as part of the network by DSL\_network::WriteFile and WriteString. For general chance node the definition consists of conditional probability table (CPT) and list of state names. To access it use DSL\_node::Definition method, which returns a pointer to the instance of the class derived from the DSL\_nodeDefinition.

```
int nodeHandle = ...;
DSL_node *node = net.GetNode(nodeHandle);
DSL_nodeDefinition *def = node->Definition();
```

The definition object is managed by the network containing the node. SMILE provides a number of classes derived from `DSL_nodeDefinition`, specialized to represent different node types (CPT, Noisy-MAX, decision, etc.) The choice of the definition object class associated with given node is based on the node type parameter passed to `DSL_network::AddNode`. As SMILE is compiled with RTTI disabled, you cannot use `dynamic_cast` to check for actual type of the object returned by `DSL_node::Definition`. However, you can use `DSL_nodeDefinition::GetType` and `GetType` methods:

```
int nodeHandle = net.AddNode(DSL_CPT, "myNodeId");
DSL_nodeDefinition *def = net.GetNode(nodeHandle)->Definition();
printf("Type of the definition: %d %s\n",
    def->GetType(), def->GetType());
```

In the example above, the 'def' variable points to the object of the `DSL_cpt` class derived from `DSL_nodeDefinition`. While it is possible to `static_cast` (or use old-style C cast) and obtain an access to type-specific functionality of the object, SMILE provides general purpose virtual methods defined in `DSL_nodeDefinition` and overridden in derived classes, which makes casting unnecessary most of the time. Two of these methods are `DSL_nodeDefinition::GetMatrix` and `GetOutcomesNames`, which give access to node's parameters and state names. Note that some of the node types do not support all of the operations. For example, the decision node does not have any numeric parameters, therefore its definition object of `DSL_list` class returns NULL from its `GetMatrix` method.

### 5.5.3 Node value & evidence

The value of the node contains the values (typically, the marginal probability distribution or the expected utilities) calculated for the node by the inference algorithm. Unlike the definition, the value is not written as part of the network by `DSL_network::WriteFile` and `WriteString`. Like the definition, the value object is managed by the network. The actual value object, which you can access through `DSL_node::Value` method, is an instance of the class derived from `DSL_nodeValue`. SMILE chooses the class appropriate for the node type during node creation.

In addition to the numeric output of the inference algorithm, the node value object may contain the evidence for the node, which is (along with node definition) part of the input to the inference algorithm. To set and remove the evidence, use `DSL_nodeValue::SetEvidence` and `ClearEvidence` methods. As with the definition part of node, most of the time there is no need to cast the pointer returned by `DSL_node::Value` to specific class, because the base class provides the set of general purpose virtual functions overridden by derived value classes.

```
int evidenceNodeHandle = ...
DSL_nodeValue *evVal = net.GetNode(evidenceNodeHandle)->Value();
evVal.SetEvidence(1); 1 is the 0-based outcome index
net.UpdateBeliefs();
int beliefNodeHandle = ...;
DSL_nodeValue *beVal = net.GetNode(beliefNodeHandle)->Value();
if (beVal->IsValueValid())
{
```

```

    const DSL_Dmatrix *m = beVal->GetMatrix();
    // read the matrix contents
}

```

Note that before accessing the actual numeric value of the node with `GetMatrix` we need to check if the value is valid by calling `IsValidValue`. The value will not be valid if inference algorithm was not called yet, or some definition or evidence has changed after the last inference call.

It is also possible to specify the virtual evidence using `DSL_nodeValue::SetVirtualEvidence` method. Virtual evidence allows for entering uncertain observation (in form of probability distribution over the possible states of the observation) directly into the normally unobservable variable. `SetVirtualEvidence` requires a `std::vector<double>` with a size equal to the number of node outcomes. The following snippet show how to set virtual evidence for a node with tree outcomes:

```

int evidenceNodeHandle = ...
DSL_nodeValue *evVal = net.GetNode(evidenceNodeHandle)->Value();
std::vector<double> virtEv(3);
virtEv[0] = 0.2; virtEv[1] = 0.7; virtEv[2] = 0.1;
evVal->SetVirtualEvidence(virtEv);

```

### 5.5.4 Other node attributes

Node's attributes not related to inference are grouped in the `DSL_nodeInfo` object, accessible through `DSL_node::Info` method. In turn, the `DSL_nodeInfo` provides access to the following objects:

- header: textual attributes, like node identifier, name and description, through `DSL_nodeInfo::Header` method returning a reference to `DSL_header` object.
- screen information: position, colors, border thickness, etc., through `DSL_nodeInfo::Screen` method returning a reference to `DSL_screenInfo` object.
- user properties: a list of key/value pairs used for application-specific purposes, through `DSL_nodeInfo::UserProperties` method returning a reference to `DSL_userProperties` object.

```

int nodeHandle = ...;
DSL_node *node = net.GetNode(nodeHandle);
DSL_nodeInfo &info = node->Info();
printf("Node name: %s\n", info.Header().GetName());
const DSL_rectangle &pos = info.Screen().position;
printf("Node center: (%d, %d), size: (%d,%d)\n",
pos.center_X, pos.center_Y,
pos.width, pos.height);

```

Note that for convenience `DSL_node` allows direct read access to its identifier by `DSL_node::GetId` method, which calls `Info().Header().GetId()`.

## 5.6 Input and output

---

SMILE supports two types of network I/O:

- string-based, with `DSL_network::WriteString` and `ReadString`
- file-based, with `DSL_network::WriteFile` and `ReadFile`

The native format for SMILE networks is XDSL. The format is XML-based and the definition schema is available at BayesFusion documentation website (<http://support.bayesfusion.com/docs/>). When writing network in this format to file, the `.xdsl` extension should be used.

XDSL is the only format supported by string I/O methods. File-based methods can read and write other formats. Depending on the feature parity between SMILE and the 3rd party software using other file types, some of the information may be lost. For complete list of supported file types, see the reference section for `DSL_network::ReadFile`.

By default, the file I/O methods infer the file type based on the filename extension. The file type can be explicitly specified in the call if needed.

```
DSL_errorH().RedirectToFile(stdout);
DSL_network net;
int res = net.ReadFile("my_network.xdsl");
if (DSL_OKAY != res)
{
    printf("ReadFile failed.\n");
}
```

In case of read failure the program listed above will display the specific error message on the standard output (due to earlier `RedirectToFile(stdout)` call).

## 5.7 Inference

---

SMILE includes functions for several popular Bayesian network inference algorithms, including the clustering algorithm, and several approximate stochastic sampling algorithms. To run the inference, use `DSL_network::UpdateBeliefs` method.

The default algorithm for discrete Bayesian Networks is clustering over network preprocessed with relevance. To switch between Bayesian Network algorithms, call `DSL_network::SetDefaultBNAlgorithm` and pass the algorithm identifier as its parameter. For

influence diagrams, the method is `DSL_network::SetDefaultIDAlgorithm`. Available algorithms are listed in the reference section for these methods.

`DSL_network::UpdateBeliefs` returns `DSL_OUT_OF_MEMORY` error code if the temporary data structures required to complete the inference take too much memory. In such case, or if the inference takes too long, consider taking advantage of SMILE's relevance reasoning layer. Relevance reasoning runs as a preprocessing step, which can lessen the complexity of later stages of inference algorithms. Relevance reasoning takes the target node set into account, therefore, to reduce the workload you should reduce the number of nodes set as targets if possible. Note that by default all nodes are targets (this is the case when no nodes were marked as such). If your network has 1,000 nodes and you only need the probabilities of 20 nodes, by all means call `DSL_network::SetTarget` on them.

If changing the model to use [Noisy-MAX](#)<sup>[36]</sup> nodes is possible, then it's definitely worth trying. The inference can be performed very efficiently on the networks with Noisy-MAX nodes when Noisy-MAX decomposition is enabled. To enable it, call `DSL_network::EnableNoisyDecomp`. If enabled, the Noisy-MAX decomposition runs in the relevance layer and reduces the complexity of the subsequent phases of the inference algorithm. To further control the decomposition you can call `DSL_network::SetNoisyDecompLimit`, which controls the maximal number of parents in the temporary structures managed by SMILE during inference.

The stochastic inference algorithms can be controlled by setting the number of generated samples with the `DSL_network::SetNumberOfSamples` method. Obviously, the more samples are generated, the more time it takes to complete the inference.

To obtain probability of evidence currently set in the network, call `DSL_network::CallProbEvidence`.

## 5.8 User properties

To integrate data specific to your application with SMILE, you can use SMILE's user properties. The user properties are lists of key/value pairs available at the `DSL_network` and `DSL_node` level. While it's possible to use data structure not managed by SMILE (like `std::map<DSL_node *, YourObject>`) to extend the set of attributes associated with the nodes, that external data is not written by `DSL_network::WriteFile` and `WriteString`. On the other hand, the user properties are stored in the XDSL files.

```
DSL_node *node = net.GetNode(handle);
// if that node has user properties,
// the loop below will print them out
DSL_userProperties &props = node->Info().UserProperties();
for (int i = 0; i < props.GetNumberOfProperties(); i++)
```

```

{
    printf("%s=%s\n",
        props.GetProperty(i),
        props.GetProperty(i));
}

```

Note that the property name is unique for the set of properties defined in given node. Property name follows the convention of SMILE identifiers: is case-sensitive, starts with a letter and contains letters, digits and underscores.

To get access to network-level user properties, call `DSL_network::UserProperties` method.

## 5.9 Cases

---

SMILE includes management of cases, which allow users to save a partial or a complete evidence set as a case and retrieve this case at a later time. Cases are saved alongside the model (XDSL file format only), so when the model is loaded at a later time, all cases are going to be available.

Internally, the case information is stored in the `DSL_simpleCase` objects, which are managed by `DSL_network` (so your program does not create instances of this class directly). To add the case to the network, use `DSL_network::AddCase`, which returns a pointer to the newly created instance of the `DSL_simpleCase` object. To retrieve the case information, use `DSL_network::GetCase`.

Each case has a name (required), a category and a description (optional). Once the case is created, your program can add evidence to the case with `DSL_simpleCase::AddEvidence`. Alternatively, you can copy current network evidence into the case with `DSL_simpleCase::NetworkToCase`.

To apply the evidence defined in the case, use `DSL_simpleCase::CaseToNetwork`.

```

DSL_simpleCase *c = net.GetCase("my_case");
for (int i = 0; i < c->GetNumberOfEvidence(); i++)
{
    int handle, outcome;
    printf("%d %d %d\n", i, handle, outcome);
}
c->CaseToNetwork();
net.UpdateBeliefs();

```

## 5.10 Sensitivity analysis

---

`DSL_sensitivity` class contains an implementation of a sensitivity analysis algorithm proposed by Kjaerulff and van der Gaag (2000).

Given a set of target nodes, the algorithm calculates a complete set of derivatives of the posterior probability distributions over the target nodes over each of the numerical parameters of the Bayesian network. Note that these derivatives are dependent on the evidence currently set in the network and they give an indication of importance of precision of network numerical parameters for calculating the posterior probabilities of the targets. If the derivative is large for a parameter  $x$ , then a small change in  $x$  may lead to a large difference in the posteriors of the targets. If the derivative is small, then even a large change in the parameter will make little difference in the posteriors.

For each node, the sensitivity can be obtained as an actual value of the derivative, and also as a set of coefficients, which define the dependency between the target node posterior and the specific CPT parameter. The general form of the function is linear rational:

$$y = (Ax + B) / (Cx + D)$$

where  $y$  is the target posterior (calculated by inference algorithm) and  $x$  is the value of the specific CPT parameter. SMILE calculates the coefficients  $A$ ,  $B$ ,  $C$ ,  $D$  and the derivative  $y'$  at the current value of  $x$ . The formula for the derivative is:

$$y' = (AD - BC) / (Cx + D)^2$$

The denominator is positive, therefore the sign of the derivative is constant for all  $x$ , and hence the function is monotonic or constant. By substituting 0 and 1 for  $x$  in the first formula (because  $x$  is a probability) we can calculate how much the posterior will change if  $x$  is modified in its CPT. The range is defined by:

$$y_1 = B / D, \quad y_2 = (A + B) / (C + D)$$

The sign of  $AD - BC$  determines which value is minimum and which is maximum.

As an illustration, let Us use the *HeparII.xdsl* model, which is distributed with GeNIe and is also available from BayesFusion's online model repository at <https://repo.bayesfusion.com>. The choice of the target node, evidence and CPT parameter is motivated only by the large value of the derivative:

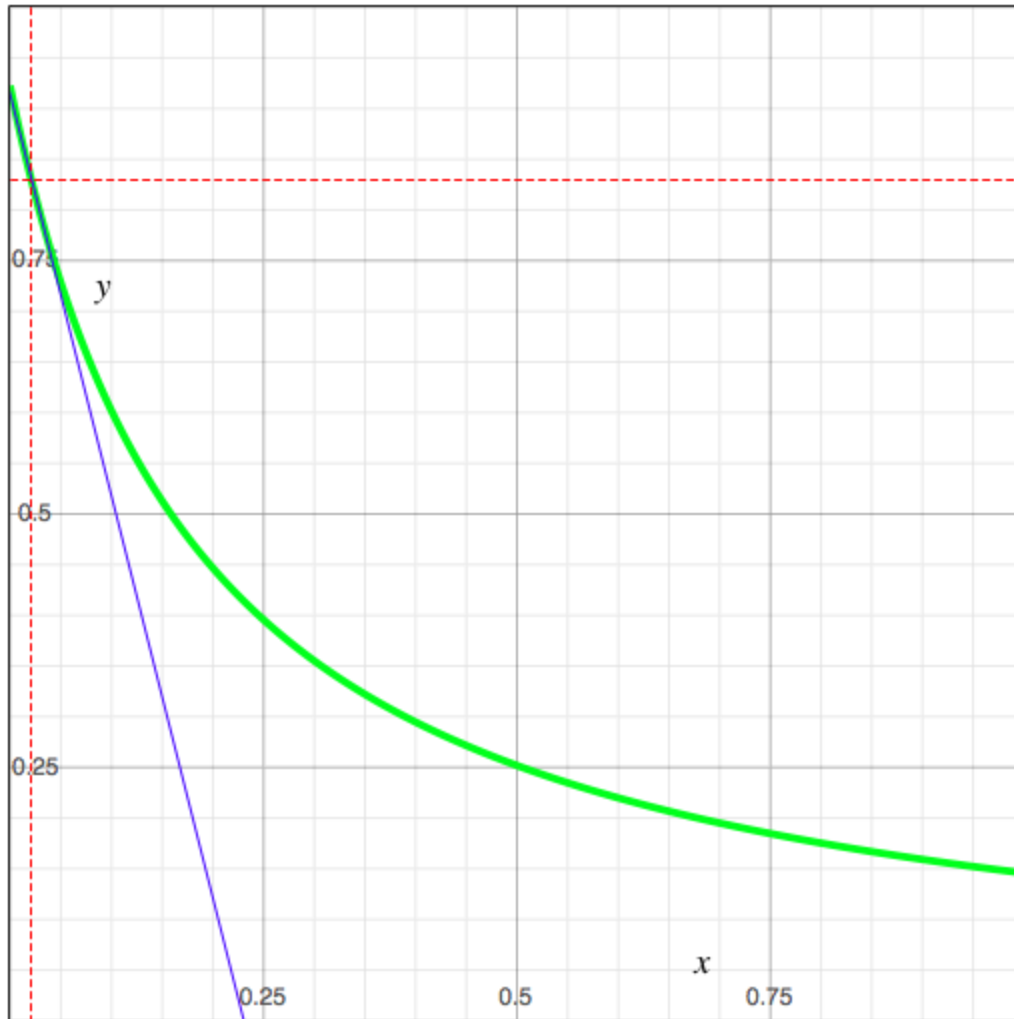
$$x = P(\text{bilirubin}=a19\_7 \mid \text{Hyperbilirubinemia}=absent, \text{PBC}=absent, \text{Cirrhosis}=absent, \\ \text{gallstones}=absent, \text{ChHepatitis}=absent) \\ y = P(\text{PBC}=present \mid \text{ast}=149\_40, \text{irregular\_liver}=present, \text{bilirubin}=a19\_7, \text{proteins}=a5\_2)$$

Note that the right side of the vertical bar in the definition of  $x$  determines the position of the parameter within the CPT of the *bilirubin* node, while the right side of the vertical bar in the definition of  $y$  represents the evidence set in the network. The sensitivity output for  $x=0.02126152$  (the current value of the parameter in the CPT) and  $y=0.82906518$  (the calculated posterior probability) is:



$$y' = -3.96207, A=0, B=6.18863e-5, C=0.00035673, D=6.70612e-5$$

With known coefficients we can now visualize the relationship between  $x$  and  $y$ :



The green curve in the image represents the function  $y(x)$ , the blue line represents the tangent of the green curve at the current value of  $x$  (its slope being equal to  $y'$ ) and the red dashed lines make it easier to locate the current  $(x, y)$  on the curve.

To perform sensitivity analysis, start with setting the target(s) in the network, instantiate the `DSL_sensitivity` object, and invoke its `Calculate` method. Next, read the values of derivatives and/or the function coefficients. The code snippet below uses the *HeparII.xdsl* network and calculates the sensitivity for the same CPT parameter and target node. To save space, it uses the `SetEvidenceById` helper function from [Tutorial 2](#)<sup>63</sup>.

```
DSL_network net;
int res = net.ReadFile("HeparII.xdsl");
```

```

if (DSL_OKAY != res) return res;

SetEvidenceById(net, "ast", "a149_40");
SetEvidenceById(net, "irregular_liver", "present");
SetEvidenceById(net, "bilirubin", "a19_7");
SetEvidenceById(net, "proteins", "a5_2");

DSL_sensitivity sens;
res = sens.Calculate(net);
if (DSL_OKAY != res) return res;

int target = net.FindNode("PBC");
if (target < 0) return target;
DSL_sensitivity::Target targetNodeAndOutcome(target, 0);

int nodeUnderStudy = net.FindNode("bilirubin");
if (nodeUnderStudy < 0) return nodeUnderStudy;

// the CPT parameter index is specified as linear, but
// it's of course possible to obtain it by passing the outcomes
// of nodeUnderStudy and its parents to DSL_Dmatrix::CoordinatesToIndex
const int paramIndex = 285;

// just to ensure we have right paramIndex
const DSL_Dmatrix *cpt =
    net.GetNode(nodeUnderStudy)->Definition()->GetMatrix();
printf("x=%g\n", (*cpt)[paramIndex]);

const DSL_Dmatrix *yPrim =
    sens.GetSensitivity(nodeUnderStudy, targetNodeAndOutcome);
double slope = (*yPrim)[paramIndex];

std::vector<const DSL_Dmatrix *> coeffs;
sens.GetCoefficients(nodeUnderStudy, targetNodeAndOutcome, coeffs);

printf("y'=%g\nA=%g B=%g C=%g D=%g\n", slope,
    (*coeffs[0])[paramIndex], (*coeffs[1])[paramIndex],
    (*coeffs[2])[paramIndex], (*coeffs[3])[paramIndex]);

```

The program outputs is:

```

x=0.0212615
y'=-3.96207
A=0 B=6.18863e-05 C=0.00035673 D=6.70612e-05

```

The `DSL_sensitivity` object owns the matrices containing the derivatives and the parameters - do not call delete on the pointers returned from its getter methods.

For performance reasons, the sensitivity algorithm works on a network processed by SMILE's relevance algorithms. Those nodes that do not affect target node's posteriors are dropped early by the relevance. The matrix objects returned for these nodes by `GetSensitivity` and `GetCoefficients` will be empty (`DSL_Dmatrix::GetSize` will return zero). Unlike the example above, your code should include the check for the matrix size and assume that target is not sensitive to changes in the node under study if the sensitivity matrix is empty.

In the usual case, the programs calculating sensitivity do not focus on the specific parameter. Instead, they iterate over the coefficients and derivatives over the entire CPT for one or more nodes. If your program only needs the maximum sensitivity value, use the overloaded `DSL_sensitivity::GetMaxSensitivity` methods. The returned numbers represent the maxima for the entire network, a single target node, a single node under study or a combination of the two.

Sensitivity analysis can be performed on influence diagrams. In such case, the terminal utility node becomes the target for sensitivity calculations. The calculated coefficients depend on the outcomes for the utility indexing parents. Use the `DSL_sensitivity::GetConfigCount` and `SetConfig` to switch between different combinations of indexing parent outcomes.

## 5.11 Deterministic nodes

Deterministic nodes can be created by using the `DSL_TRUTHTABLE` node type identifier when calling `DSL_network::AddNode`. They behave like CPT nodes, with the exception that their probability table contains only zeros and ones. Deterministic nodes are defined as having no noise in their definition, so there is no uncertainty about the outcome of a deterministic node once all its parents are known.

While it is possible to model a deterministic variable using a chance node with its CPT filled with zeros and ones, the deterministic nodes are displayed as double ellipses in GeNIe, making the modeler's intent explicit and, hence, possible modeling errors less likely.

The definition of a deterministic node is represented by the object of `DSL_truthTable` class. The pointer returned from `DSL_node::Definition` should be cast appropriately to access its methods. The example code below assumes a deterministic node with three outcomes and two binary parents. Its definition is set with a call to `DSL_truthTable::SetResultingStates`. For each column of the truth table, we specify one resulting state.

```
DSL_truthTable *tt = static_cast<DSL_truthTable *>(net.GetNode(h)->Definition());
const int resStates[] = { 2, 0, 1, 1 };
tt->SetResultingStates(resStates);
```

The result will look as follows in GeNIe:

p1		State0		State1	
p2		State0	State1	State0	State1
State0					
State1					
State2					

The `SetResultingStates` method is overloaded - see the reference sections for details.

## 5.12 Canonical nodes

Canonical probabilistic nodes, such as Noisy-MAX/OR, Noisy-MIN/AND, and Noisy-Adder gates, implemented by SMILE, are convenient knowledge engineering tools widely used in practical applications. In case of a general CPT binary node with  $n$  binary parents, the user has to specify  $2^n$  parameters, a number that is exponential in the number of parents. This number can quickly become prohibitive: when the number of parents  $n$  is equal to 10, we need 1,024 parameters, when it is equal to 20, the number of parameters is equal to 1,048,576, with each additional parent doubling it. A Noisy-OR model allow for specifying this interaction with only  $n+1$  parameters, one for each parent plus one more number. This comes down to 11 and 21 for  $n$  equal to 10 and 20 respectively.

Canonical models are not only great tools for knowledge engineering - they also lead to significant reduction in computation through the independences that they model implicitly. Using canonical gates makes thus model construction easier but also leads to models that are easier to solve.

To create canonical nodes, pass `DSL_NOISY_MAX` or `DSL_NOISY_ADDER` to `DSL_network::AddNode`. Each node type exposes its own specific attributes through its definition object, but otherwise works in exactly the same way as a CPT node (has at least two outcomes, can be used as a parent or a child wherever the CPT node can, etc).

### 5.12.1 Noisy-MAX

Noisy-MAX is a generalization of the popular canonical gate Noisy-OR and is capable of modeling interactions among variables with multiple states. If all the nodes in question are binary, a Noisy-MAX node reduces to a Noisy-OR node. The Noisy-MAX, as implemented in SMILE, includes an equivalent of negation. By DeMorgan's laws, the OR function (or its generalization, the MAX function) along with a negation, is capable of expressing any logical relationship, including the AND (and its generalization, MIN). This means that SMILE's Noisy-MAX can be used to model the Noisy-AND/MIN functions, as well as other logical relationships.

SMILE's inference algorithm contains special code path for networks with Noisy-MAX nodes, which can speed up computations significantly. See the [Inference](#)<sup>[29]</sup> section of this manual for details.

To create a Noisy-MAX node, use the `DSL_NOISY_MAX` type with `DSL_network::AddNode`:

```
int h = net.AddNode(DSL_NOISY_MAX, "node1");
```

In such case, `AddNode` creates a node with a definition object of `DSL_noisyMAX` class. In addition to information about node outcomes, the definition contains the following Noisy-MAX specific data:

- a set of conditionally independent probabilities
- for each of node's parents, the vector of parent outcome strengths.

The conditional probabilities are stored in the two-dimensional `DSL_Dmatrix` object, accessible through `DSL_noisyMAX::GetCiWeights`. First dimension of the matrix has the size which is the sum of the number of parent outcomes plus one (for the leak column). The second dimension is the number of the node's own outcomes. The last column for each of the parents should be constrained and have zero probabilities in all but its last element.

Parent outcome strengths enable control of the order of states of the parent nodes, as they enter their relation with the child. A Noisy-MAX table always follows the order of strengths.

As an example, consider a binary Noisy-MAX node with two parents, each with three outcomes. The following snippet modifies the probabilities and outcome strengths for the second parent (with zero-based index 1).

```
DSL_noisyMAX *maxDef = static_cast<DSL_noisyMAX *>(net.GetNode(h)->Definition());
DSL_Dmatrix &p = maxDef->GetCiWeights();
int BASE = 2 * 3;
p[BASE] = 0.1;
p[BASE + 1] = 0.9;
p[BASE + 2] = 0.3;
p[BASE + 3] = 0.7;
DSL_intArray strengths;
strengths.Add(2);
strengths.Add(0);
strengths.Add(1);
maxDef->SetParentOutcomeStrengths(1, strengths);
```

The value of `BASE` is calculated as a product of node outcome count and preceding parents' outcome counts (in this case, there is just one preceding parent with three outcomes). The probabilities for the parent are written directly into `DSL_Dmatrix` returned by reference from `GetCiWeights`. The next step changes the order of parents' outcomes in relationship to the Noisy-MAX node. The `DSL_intArray` object is created and initialized with parent outcome indices. After the `DSL_noisyMAX::SetParentOutcomeStrengths` call, the first column of probabilities for parent with index 1 (the one with 0.1 and 0.9) represents the probabilities for the parent outcome with index 2 (because 2 is the first element in the `strengths` array). Note that this does not modify the parent node in any way and the ordering is valid only in the context of this particular parent-child relationship. Assuming that we started with the default uniform probabilities in the table, our modifications yields the following Noisy-Max definition, as viewed in GeNIe:

Parent		p1			p2			LEAK
State		State0	State1	State2	State2	State0	State1	
▶ State0		0.5	0.5	0	0.1	0.3	0	0.5
State1		0.5	0.5	1	0.9	0.7	1	0.5

The outcomes of both parent are  $\{State0, State1, State2\}$ . However, by using `DSL_noisyMAX::SetParentOutcomeStrengths` for parent  $p2$ , its outcomes are seen by the Noisy-MAX child node as  $\{State2, State0, State1\}$ . Other Noisy-MAX nodes in the same network can set up their own parent outcome ordering if  $p2$  becomes their parent.

### 5.12.2 Noisy-Adder

The Noisy-Adder model is described in the doctoral dissertation of Adam Zagorecki (2010), Section 5.3.1 *Non-decomposable Noisy-average*. Essentially, it is a non-decomposable model that derives the probability of the effect by taking the average of probabilities of the effect given each of the causes in separation.

To create a Noisy-Adder ode, use the DSL\_NOISY\_ADDER type with DSL\_network::AddNode:

```
int h = net.AddNode(DSL_NOISY_MAX, "node1");
```

The class of the definition object associated with the new node is DSL\_noisyAdder. In addition to information about node outcomes, the definition contains the following Noisy-Adder specific data:

- index of node's own distinguished state
- for each of node's parents, the index of the parent's distinguished state
- weights for each of the node's parents and for the leak

Consider a binary Noisy-Adder node with two parents, each with three outcomes. The following snippet modifies the probabilities, the weight, and the distinguished state for the second parent (with zero-based index 1). Node's own distinguished state is set to 0.

```
DSL_noisyAdder *addDef = static_cast<DSL_noisyAdder *>(net.GetNode(h)->Definition());
addDef->SetDistinguishedState(0);
DSL_Dmatrix &p = addDef->GetCiWeights();
int BASE = 2 * 3;
p[BASE + 2] = 0.2;
p[BASE + 3] = 0.8;
p[BASE + 4] = 0.4;
p[BASE + 5] = 0.6;
addDef->SetParentDistinguishedState(1, 0);
addDef->SetParentWeight(1, 2.5);
```

The image below shows the Noisy-Adder definition table (as viewed in GeNIe) after the modifications applied by the code above. The modified parent's weight is shown in parenthesis after its identifier (*p2*). Distinguished states are marked by bold font.

Parent (Weight)	p1 (1)			p2 (2.5)			LEAK (1)
State	State0	State1	State2	State0	State1	State2	
State0	0.5	0.5	1	1	0.2	0.4	0
State1	0.5	0.5	0	0	0.8	0.6	1

Parent distinguished states and weights are part of the child Noisy-Adder definition. If *p2* has other Noisy-Adder children, each of these nodes can specify its own distinguished state and weight for *p2*.

### 5.13 Influence diagrams

---

Influence diagrams use two additional node types next to chance (CPT and canonical) and deterministic nodes:

- Decision nodes represent variables that are under control of the decision maker and model available decision alternatives, modeled explicitly as possible states of the decision node. They have no numerical parameters, only a discrete set of outcomes. Decision nodes can be children of decision and chance nodes. The node type identifier passed to `DSL_network::AddNode` is `DSL_LIST` (short for "List of decisions").
- Value nodes, i.e., a measure of desirability of the outcomes of the decision process. They are quantified by the utility of each of the possible combinations of outcomes of the parent nodes, specified as an `DSL_Dmatrix` object. Value nodes can be children of decision and chance nodes. Pass `DSL_TABLE` to `DSL_network::AddNode` to create a value node.
- Multi-attribute utility (MAU) nodes, which combine value nodes to form a multi-attribute utility function. The function can be specified as a set of weights of a linear function (in such case, the node becomes an additive linear utility, `ALU`) or any expression that refers to identifiers of the value node parents. See the [Equations](#)<sup>[143]</sup> section of the reference manual for a list of available functions. MAU nodes can be children of decision, value, and other MAU nodes. If decision parents exist, the definition of the MAU node contains a separate set of weights or expressions for each combination of decision parents. Use `DSL_MAU` with `DSL_network::AddNode` to create a MAU node.

As is the case with Bayesian networks, the values calculated by influence diagram inference algorithms are stored in node values and can be accessed through the same APIs. However, the interpretation of the numbers stored in `DSL_Dmatrix` objects retrieved with `DSL_nodeValue::GetMatrix` is extended. The matrices are indexed by the set of nodes called indexing parents. These are unobserved decision nodes that precede the current node or unobserved chance nodes that are predecessors of decision nodes and should have been observed before the decisions can be made. Call `DSL_nodeValue::GetIndexingParents` to retrieve indexing parents. The set of outcomes of indexing parents is called a policy. After a successful inference in an influence diagram, node values are:

- for chance and deterministic nodes: posterior probabilities for each policy
- for decision nodes: expected utilities for all outcomes and for each policy
- for value and MAU nodes: expected utility for each policy

See [Tutorial 4](#)<sup>[72]</sup> for a simple influence diagram demo program.

## 5.14 Dynamic Bayesian networks

---

A Bayesian network is a snapshot of the system at a given time and is used to model systems that are in some kind of equilibrium state. Unfortunately, most systems in the world change over time and sometimes we are interested in how these systems evolve over time more than we are interested in their equilibrium states. Whenever the focus of our reasoning is change of a system over time, we need a tool that is capable of modeling dynamic systems.

A dynamic Bayesian network (DBN) is a Bayesian network extended with additional mechanisms that are capable of modeling influences over time (Murphy, 2002). The temporal extension of BNs does not mean that the network structure or parameters changes dynamically, but that a dynamic system is modeled. In other words, the underlying process, modeled by a DBN, is stationary. A DBN is a model of a stochastic process.

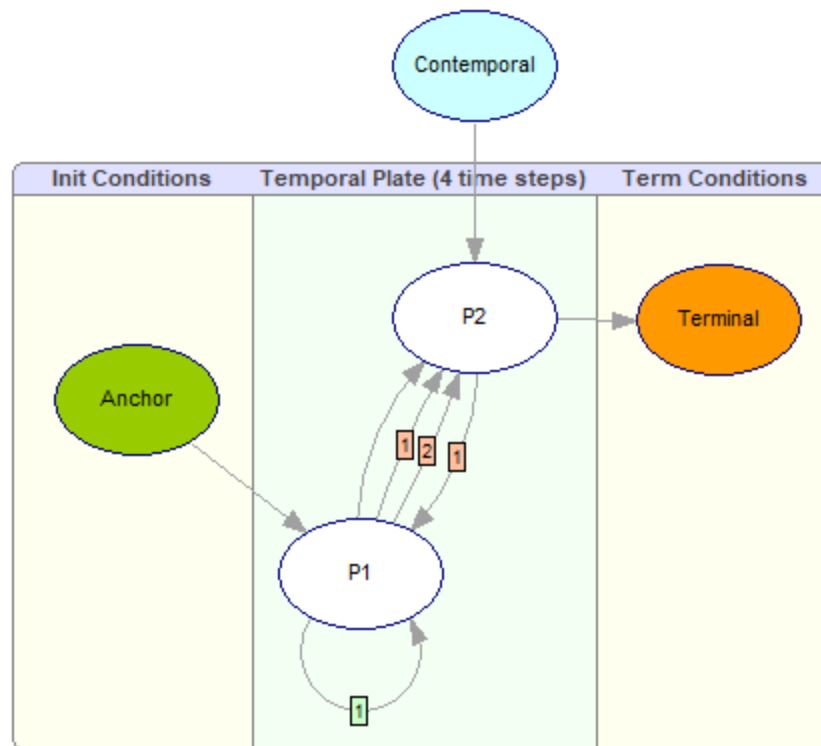
The implementation of DBNs in SMILE allows for use both chance (CPT and canonical) and deterministic node types in dynamic models.

[Tutorial 6](#)<sup>[80]</sup> contains a complete program demonstrating the use of DBNs.

### 5.14.1 Unrolling

SMILE's inference algorithm for dynamic Bayesian networks converts DBNs (unrolls them over time) to temporary static networks and then solves these networks. Results of this inference are copied back into the node values in the original DBNs. The structure of the following DBN (in GeNIe format) does not represent any real system, it is created just for demonstration purposes.

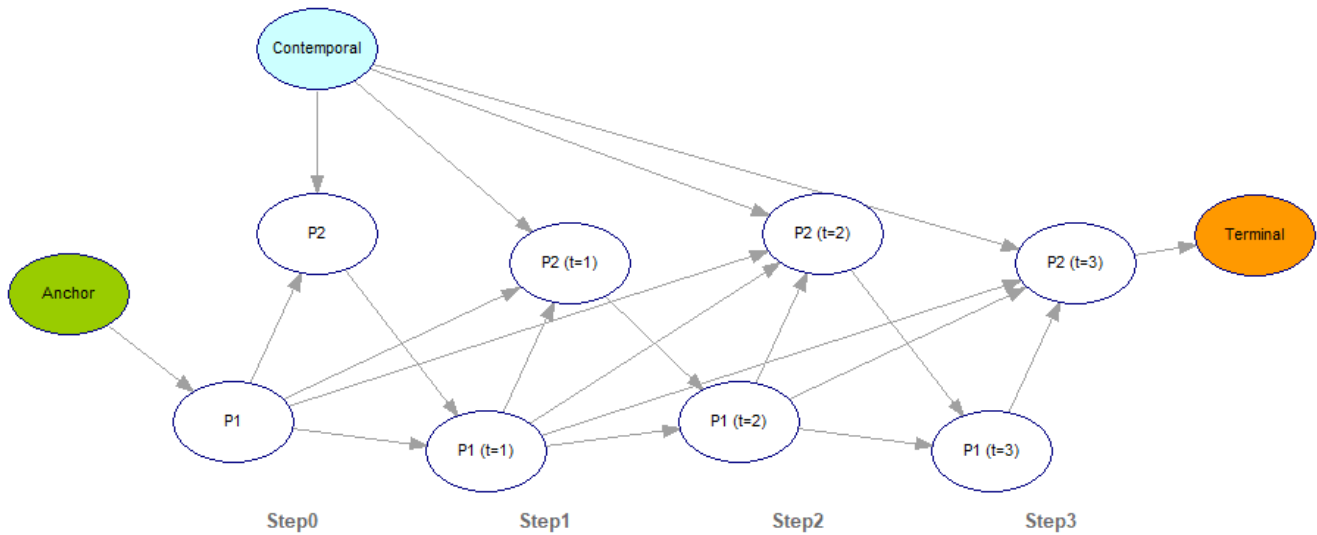




Different colors of the nodes represent their location relative to the abstract "temporal plate." There are four temporal types of nodes, defined by the following enum:

```
enum dsl_temporalType { dsl_normalNode, dsl_anchorNode, dsl_terminalNode, dsl_plateNode };
```

By default, the nodes in the network are set to be `dsl_normalNode`. By using `DSL_network::SetTemporalType` you can change their temporal type assignment. In the example model above, the plate nodes are white, the anchor node is green, the terminal node is orange and the blue node is normal. For simplicity, we use only single anchor, terminal, and normal node. Note the presence of multiple arcs between two plate nodes and the arc linking *P1* with itself. Some of the arcs between plate nodes have a tag with a number; these are *temporal arcs*, which are created by `DSL_network::AddTemporalArc`. The number on the tag is a *temporal order* of an arc. The arcs without the tag were added by `DSL_network::AddArc`. The result of unrolling the above DBN (this is performed automatically by the DBN inference algorithm; it is possible to create such network with `DSL_network::UnrollNetwork`) is the following:



To reduce the size of the unrolled network, we changed the number of time steps (slices) from the default value of 10 to 4 with a call to `DSL_network::SetNumberOfSlices`. The colors of the original nodes were carried over to the unrolled model, which is extended by creating new copies of the plate nodes. The structure of this network shows how the temporal arcs are used to express the conditional dependency of plate nodes in time step  $i$  on the plate nodes in time step  $j$ , where  $i > j$ .

Consider the (temporal) arc between  $P1$  and  $P2$  with temporal order 2. It was copied twice to link  $P1$  with  $P2(t=2)$ , and  $P1(t=1)$  with  $P2(t=3)$ . Note that there is no copy of this arc starting from  $P1(t=2)$ , because its child would be in  $t=2+2=4$ , and (zero-based) time stops at 3 in our example. On the other hand, all three temporal arcs with temporal order 1 are copied three times. The relationship represented by an arc linking  $P1$  with itself in the DBN is clearly visible between  $P1$  and  $P1(t=1)$ ,  $P1(t=1)$  and  $P1(t=2)$  and  $P1(t=2)$  and  $P1(t=3)$ .

Note the difference between the arcs originating in *Contemporal* and *Anchor* nodes. The *Anchor* node has children only in the initial slice, while the *Contemporal* node is linked to all time slices. The *Terminal* node has parents only in the last slice.

To ensure that unrolled network remains a directed acyclic graph, the following arcs are forbidden in DBNs:

- from plate nodes to normal and anchor nodes
- from terminal nodes to non-terminal nodes

Unrolling is performed automatically during inference. For debug/explanatory purposes it is also possible to obtain an unrolled network by calling `DSL_network::UnrollNetwork`. The unrolled network created this way is an independent model, which means that changes made to the DBN after `UnrollNetwork` call are not propagated into the unrolled network.

### 5.14.2 Temporal definitions

Consider node  $P_2$  from the example in the previous section. It has four incoming arcs:

- normal arc from *Contemporal*
- normal arc from  $P_1$
- two temporal arcs from  $P_1$  with temporal orders 1 and 2

However, in the unrolled network's slice for  $t=0$ ,  $P_2$  has only two incoming arcs. This is because there are no slices representing timepoints before  $t=0$ .  $P_2$  in the slice for  $t=1$  has three incoming arcs, because it's now possible to link  $P_1$  at  $t=0$  with  $P_2$  at  $t=1$ . Finally, starting with slice for  $t=2$ ,  $P_2$  has four incoming arcs. The structure of the unrolled network requires  $P_2$  to have separate CPTs for  $t=0$ ,  $t=1$  and  $t \geq 2$ . Generally speaking, if a plate node has an incoming arc of order  $x$ , it will require  $x+1$  separate definitions. To get and set the temporal definitions for CPT nodes, use `DSL_nodeDefinition::GetTemporalDefinition` and `SetTemporalDefinition`. The node definition classes for canonical and deterministic nodes have methods for accessing temporal definitions. For example, the `DSL_noisyMAX` defines `Get/SetTemporalCiWeights` and `Get/SetTemporalParentOutcomeStrengths`. See the reference section for details; the method names always start with `GetTemporal` or `SetTemporal`.

All parents for the temporal definition with a specified temporal order can be retrieved by calling `DSL_network::GetUnrolledParents`. Note that `DSL_network::GetTemporalParents` called for the same temporal order returns only a subset of parents indexing this temporal definition. This is caused by unrolling: in the unrolled network node  $P_2$  has four incoming arcs in slices for  $t \geq 2$ , but only two of these are actually arcs with temporal order 2.

### 5.14.3 Temporal evidence

To specify evidence for the plate nodes in the DBN, use `DSL_nodeValue::SetTemporalEvidence`. The code snippet below sets the temporal evidence in slices 5 and 7. The latter is virtual evidence.

```
int evidenceNodeHandle = ...
DSL_nodeValue *val = net.GetNode(evidenceNodeHandle)->Value();
val.SetTemporalEvidence(5, 1); 1 is the 0-based outcome index
std::vector<double> virtEv(nodeOutcomeCount);
// fill in virtEv here
val.SetTemporalEvidence(7, virtEv);
```

To retrieve the temporal evidence, use `DSL_nodeValue::GetTemporalEvidence`. The method has two overloads, one for normal and another for virtual evidence.

Other useful methods are `DSL_nodeValue::HasTemporalEvidence` and `IsTemporalEvidence`, which check whether a node has any temporal evidence or temporal evidence in specified temporal order respectively.

### 5.14.4 Temporal beliefs

For plate nodes in the DBN, the inference algorithm calculates the temporal beliefs, which are marginal posterior probability distributions as a function of time. To access temporal beliefs, use the `DSL_nodeValue::GetMatrix` method. We described the same method earlier in [Node value & evidence](#)<sup>[27]</sup> section, where it was used to retrieve non-temporal beliefs. The dependency of the beliefs on time makes the temporal beliefs matrix larger. If a node has  $X$  outcomes and the slice count was set to  $Y$ , the matrix will have  $X * Y$  elements. The elements representing a single time slice are adjacent. Therefore, elements with indices  $[0..X-1]$  in the matrix are the beliefs for  $t=0$ , elements  $[X..2*X-1]$  are the beliefs for  $t=1$  and so on. The code snippet below iterates over the temporal beliefs and displays a single line of numbers for each time step. The numbers are probabilities for node outcomes.

```
DSL_node *node = ...;
int outcomeCount = node->Definition()->GetNumberOfOutcomes();
int sliceCount = node->Network()->GetNumberOfSlices();
const DSL_Dmatrix *mtx = node->Value()->GetMatrix();
for (int sliceIdx = 0; sliceIdx < sliceCount; sliceIdx++)
{
    printf("\ttt=%d:", sliceIdx);
    for (int i = 0; i < outcomeCount; i++)
    {
        printf(" %f", (*mtx)[sliceIdx * outcomeCount + i]);
    }
    printf("\n");
}
```

Only the plate nodes have the temporal beliefs. Other temporal node types have normal beliefs (the number of elements in the belief array is equal to their outcome count).

## 5.15 Continuous models

---

Graphical models, such as Bayesian networks, are not necessarily consisting of only discrete variables. They are, in fact, close relatives of systems of simultaneous structural equations. SMILE allows for constructing models consisting of equation nodes that are alternative, graphical representations of systems of simultaneous structural equations.

[Tutorial 7](#)<sup>[84]</sup> contains a complete program demonstrating the use of continuous models.

### 5.15.1 Equation-based nodes

To create an equation node, use `DSL_EQUATION` node type when creating a node with `DSL_network::AddNode`. The node equation will be initialized to  $id=0$ , where  $id$  is the node

identifier passed to `AddNode`. The definition of the equation node is represented by an object of the `DSL_equation` class. The code snippet below changes the default equation of the freshly created node ( $x=0$ ) to an equation representing the standard Gaussian distribution:  $x=Normal(0, 1)$ .

```
DSL_network net;
int hx = net.AddNode(DSL_EQUATION, "x");
DSL_equation* eqDefX =
    static_cast<DSL_equation*>(net.GetNode(hx)->Definition());
eqDefX->SetEquation("x=Normal(0,1)");
```

SMILE defines many functions for use in node equations. The complete list of functions is available in the [Equations](#)<sup>[143]</sup> section in the reference chapter of this manual. Among these functions, there are probability distributions, which generate a single sample based on the passed parameters. In the example above, the `Normal` is the name of SMILE's probability distribution function.

Node equations can reference other nodes by using their identifiers. Continuing with our code snippet:

```
int hy = net.AddNode(DSL_EQUATION, "y");
DSL_equation* eqDefY =
    static_cast<DSL_equation*>(net.GetNode(hy)->Definition());
eqDefY->SetEquation("y=2*x");
```

Second node is added and its equation is set to  $y=2*x$ , where  $x$  is a reference to previously defined node. SMILE **adds an arc** between  $x$  and  $y$  automatically and it is not necessary to call `DSL_network::AddArc` before setting the equation referencing other nodes. Calling `AddArc` will change the child node equation by adding a term representing the parent node as a last term on the right hand side of the child equation. Assuming network with equation nodes  $a$ ,  $b$ ,  $c$  and  $d$  and  $d$ 's equation set to  $d=Normal(a, 1)+Normal(b, 2)$ , calling `AddArc` to with node handles of  $c$  and  $d$  would rewrite  $d$ 's equation to  $d=Normal(a, 1)+Normal(b, 2)+c$ .

If an arc is removed, either by calling `DSL_network::RemoveArc` or `DSL_network::DeleteNode` on one of the parents, the node equation will be rewritten as sum of the remaining parents. This ensures that equations and arcs are always in sync. If node  $b$  would be removed with `DeleteNode`, or an arc from  $b$  to  $d$  would be removed by `RemoveArc`,  $d$ 's equation would become  $d=a+c$ .

Node identifier changes are propagated into the equations. If the identifier of the first node from the code snippet above was changed from  $x$  to  $x\_prime$ , the equation of node  $y$  would change to  $y=2*x\_prime$ .

### 5.15.2 Continuous inference

To run the inference in continuous model, use `DSL_network::UpdateBeliefs`; the same method which is used in discrete networks.

The inference in continuous networks is based on stochastic sampling when there is no evidence in the network, or the evidence is specified only for nodes without parents. Otherwise, the inference is performed on a temporary discrete network, derived from the original continuous model. The

definitions for the temporary discrete nodes are derived from the discretization intervals defined in each continuous node.

To set evidence in an equation node, use `DSL_nodeValue::SetEvidence(double)` overload. In the snippet below we are assuming that `evidenceNodeHandle` is the handle of the equation node.

```
int evidenceNodeHandle = ...
DSL_nodeValue *evVal = net.GetNode(evidenceNodeHandle)->Value();
evVal.SetEvidence(1.5); // 1.5 is the evidence value
```

Be careful and avoid passing integer literals when you want to set continuous evidence without the fractional part. If instead of 1.5 the evidence should be equal to 2, the literal has to have ".0" suffix.

```
evVal.SetEvidence(2.0); // setting evidence to 2
```

Without the suffix the compiler would use `DSL_nodeValue::SetEvidence(int)` method, which is not applicable for equation nodes. Of course this problem does not happen when evidence value is stored in a variable of type `double`.

Both types of inference algorithm use the lower and upper bounds defined for each equation node. To set the bounds, use `DSL_equation::SetBounds` method. Stochastic sampling can reject a sample when its value falls outside of the bounds defined for the node. You can control this behavior by `DSL_network::EnableRejectOutlierSamples` method. By default, outlier rejection is disabled.

Stochastic sampling can be controlled by setting the sample count with `DSL_network::SetNumberOfDiscretizationSamples`. Large number of samples provides better approximation of the solution to the set of equations, which continuous network represents, but requires more time to complete.

To set the discretization intervals, use `DSL_equation::SetDiscIntervals`. The method accepts a vector of intervals as its argument. The vector consists of string/double pairs. The first element of the pair is the interval identifier, which is not used during inference. The second element of the pair is the upper bound of the discretization interval. The lower bound of the interval with index *j* is defined by upper interval of the interval with index *j*-1. The lower bound of the first interval is defined by the lower bound defined for the node with a `SetBounds` call.

Discretization intervals are used to obtain CPTs for the temporary discrete network. The CPTs are calculated by drawing a number of samples specified at the network level for each CPT column. Call `DSL_network::SetNumberOfDiscretizationSamples` to change this number, which defaults to 10000. The size of the discretized CPT is a product of the number of node intervals and parent node intervals. Note that this may lead to excessive memory use when the node has many parents.

The results of the inference in a continuous model can be either samples or discretized beliefs, depending on the inference algorithm. If the inference was performed by sampling, node values will have sample vectors with actual sample values. If the inference was performed on the temporary discrete network, the equation node values in the original continuous model will have discretized beliefs (discrete probability distributions over their discretization intervals). In contrast

to simply setting evidence, the access to samples, sample statistics and discretized beliefs requires a pointer cast from `DSL_nodeValue` to the derived class `DSL_valEqEvaluation`:

```
int evidenceNodeHandle = ...
DSL_nodeValue *v = net.GetNode(evidenceNodeHandle)->Value();
DSL_valEqEvaluation *eqVal = static_cast<DSL_valEqEvaluation *>(v);
const std::vector<double> &discBeliefs = eqVal->GetDiscBeliefs();
if (discBeliefs.empty())
{
    double mean, stddev, vmin, vmax;
    eqVal->GetStats(mean, stddev, vmin, vmax);
    // use statistics here
}
```

## 5.16 Hybrid models

Hybrid models are networks with both discrete and continuous nodes. The arcs in a hybrid network can link all combinations of node types, so it is possible to add an arc from a continuous to a discrete node, and from a discrete to a continuous node. Inference in hybrid models follows the rules defined for continuous nodes (sampling when there is no evidence in nodes with parents, discretization otherwise). Basically, hybrid models can be treated as continuous models with discrete nodes representing the specialized function, namely conditional probability table specified by an array of numbers.

Adding arcs from discrete to continuous nodes is performed by including discrete node identifier in the continuous node equation (as is the case for the continuous to continuous arcs). The following discussion assumes that reader is familiar with the functions from which the node equations are built, described in detail in the [Equations](#)<sup>143</sup> section of the reference part of this manual.

For example, assuming that a node *c* is continuous and a node *d* is discrete, the equation for *c* may look like this: *c*=*Normal*(*If*(*d*=1, 1, -1), 5). When a sample for node *c* is evaluated, one of its inputs will be the value of its parent node *d*. Discrete node values are numbers drawn from the interval  $[0 .. N-1]$ , where *N* is the number of discrete node outcomes. Therefore, the example equation for *c* above says that *c* should be drawn from a normal distribution with standard deviation equal to 5, but with mean depending on the parent node *d*. If *d* is in its (zero-based) state with index 1, the mean will be 1 and -1 otherwise.

To improve the readability of the equation, the outcomes of the parent discrete nodes can also be represented as text literals. If *d* has three outcomes *High*, *Medium* and *Low*, then the equation from the preceding example could be rewritten as *c*=*Normal*(*If*(*d*="Medium", 1, -1), 5).

In addition to the function *If* or its counterpart, the ternary operator *?:*, the common functions to use with discrete nodes are *Switch* and *Choose*. For example, the equation for node *c* with three possible means of its normal distributions can look like this: *c*=*Normal*(*Switch*(*d*, "High", 3.2, "Medium", 2.5, "Low", 1.4), 5). An alternative notation would be *c*=*Normal*(*Choose*(*d*, 3.2, 2.5, 1.4), 5).

**Important:** SMILE will not modify the text literals representing the outcomes of discrete parent nodes if the outcome identifiers change. If the text literal cannot be associated with any parent node outcome, its value is evaluated as -1 (minus one).

Generally speaking, discrete nodes can appear anywhere in the equation where numbers can be used:  $c = \log(1+d)$  or  $c = 2^d$ . This kind of equation does not use the text literals representing the discrete node outcomes (because there is no comparison involved in evaluation the equation).

To add an arc from a continuous to a discrete node, use `DSL_network::AddArc` (the method used in discrete models). In such case, the discretization intervals of the parent node are considered to be the equivalent of the outcomes of discrete parent.

[Tutorial 8](#)<sup>90</sup> contains a complete program demonstrating the use of hybrid models.

## 5.17 Datasets

---

The data used by SMILE for learning and network validation is stored in the objects of the `DSL_dataset` class. The dataset is a table-like structure. Its columns are called *variables* and rows are called *records*.

The contents of the dataset can be loaded from a text file. Alternatively, your program can initialize the structure of the dataset, then add records containing actual data.

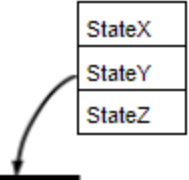
### 5.17.1 Text file I/O

You can load the contents of the dataset from a text file by calling `DSL_dataset::ReadFile`. For the illustration purposes, let's assume we have the comma-separated text file with the following data:

```
VarA,VarB,VarC
44.225,3,StateZ
26.913,0,StateY
24.379,2,*
*,3,StateX
76.681,*,StateZ
44.702,1,StateX
```

After the `DSL_dataset::ReadFile` call, the dataset will be structured like this:





	VarA	VarB	VarC
0	44.225	3	2
1	26.913	0	1
2	24.379	2	-1
3	sqrt(-1)	3	0
4	76.681	-1	2
5	44.702	1	0

- variable VarA is continuous. The missing value was replaced by sqrt(-1).
- variable VarB is discrete. The missing value was replaced by -1.
- variable VarC is also discrete, and has the string labels associated with its integer values. The missing value was replaced by -1.

You can fine-tune the parsing by passing a `DSL_datasetParseParams` structure to `ReadFile`. The following code should be used if the data file has no header line with the names of the columns, missing values are marked by a "N/A" string and missing values in the discrete columns should be replaced by 999:

```
DSL_dataset ds;
DSL_datasetParseParams params;
params.columnIdsPresent = false;
params.missingValueToken = "N/A";
params.missingInt = 999;
int res = ds.ReadFile("datafile.txt", &params);
```

To write the contents of the dataset to a text file, use `DSL_dataset::WriteFile`. You can customize the field separator, the missing value marker, etc. by passing a `DSL_datasetWriteParams` structure to `WriteFile`. The example below writes a comma-separated file, which includes a header line with column names and uses "(none)" as a marker for missing values:

```
DSL_dataset ds;
// create or load dataset here
DSL_datasetWriteParams params;
params.columnIdsPresent = true;
params.missingValueToken = "(none)";
params.separator = ',';
int res = ds.WriteFile("datafile.csv", &params);
```

### 5.17.2 Discrete and continuous variables

If the data for learning or validation comes from the source other than a text file, you'll need to programatically initialize the structure and the contents of the dataset. Consider the example dataset from the previous chapter - it had three variables, the first was continuous and other two were discrete. The code to create the structure of this dataset looks like this:

```
DSL_dataset ds;
ds.AddFloatVar("Var1");
ds.AddIntVar("Var2");
ds.AddIntVar("Var3");
vector<string> stateNames(3);
stateNames[0] = "StateX";
stateNames[1] = "StateY";
stateNames[2] = "StateZ";
ds.SetStateNames(ds.FindVariable("Var3"), stateNames);
```

`DSL_dataset::FindVariable` was used to get the index of the variable with a known identifier. An alternative approach in the example above would use a hardcoded index, as we know what variables were added to the dataset just prior to `DSL_dataset::SetStateNames`.

You can set the number of records in the dataset upfront with a call to `DSL_dataset::SetNumberOfRecords`, or call `AddEmptyRecord` for each record you're appending to the dataset.

```
ds.AddEmptyRecord();
int recIdx = ds.GetNumberOfRecords() - 1;
ds.SetFloat(0, recIdx, 44.225);
ds.SetInt(1, recIdx, 3);
ds.SetInt(2, recIdx, 2);
```

Note that depending on the type of the variable, you need to use either `DSL_dataset::SetFloat` or `SetInt`. In the example above, the last variable has associated state names, but they're not used for data entry.

To mark an element of the variable as missing, use `DSL_dataset::SetMissing`. After `AddEmptyRecord` all the elements of the last record in the dataset are missing.

The following code snippet displays content of any dataset. It uses `DSL_dataset::IsDiscrete` to determine the type of the variable. For discrete variables, the state names vector returned by `DSL_dataset::GetStateNames` is also checked. If the vector is empty, there are no strings associated with the integer variable values.

```
int varCount = ds.GetNumberOfVariables();
int recCount = ds.GetNumberOfRecords();
for (int r = 0; r < recCount; r++)
{
    for (int v = 0; v < varCount; v++)
    {
        if (v > 0) printf(",");
        if (ds.IsMissing(v, r))
        {
```

```

        printf("N/A");
    }
    else if (ds.IsDiscrete(v))
    {
        int x = ds.GetInt(v, r);
        const vector<string> &states = ds.GetStateNames(v);
        if (states.empty())
        {
            printf("%d", x);
        }
        else
        {
            printf("%s", states[x].c_str());
        }
    }
    else
    {
        printf("%f", ds.GetFloat(v, r));
    }
}
printf("\n");
}

```

### 5.17.3 Generating data from network

Given that a Bayesian network is a representation of the joint probability distribution over its variables, you can generate a dataset based on this distribution. Use the `DSL_dataGenerator` class for this purpose. The `DSL_dataGenerator::GenerateData` method has three overloads, which write the data to the following outputs:

- `DSL_dataset` object
- text file specified by a filename
- any object derived from the pure abstract class `DSL_dataGeneratorOutput`

The following code snippet generates the 200 records with 1/4 of the values marked as missing and writes the output to a text file:

```

DSL_network net;
// create or load network here
DSL_dataGenerator gen(net);
gen.SetNumberOfRecords(200);
gen.SetMissingValuePercent(25);
gen.GenerateData("out.txt");

```

#### 5.17.4 Discretization

To discretize the dataset variable, use the `DSL_dataset::Discretize` method. To convert variable to the discrete variable with 10 states use the following code:

```
vector<double> edges;  
int res = ds.Discretize(varIdx, DSL_dataset::Hierarchical, 10, "discState", edges);
```

Note that after the discretization the variable will have state names starting with the specified prefix ("discState" in this case). The names will be suffixed with numeric values derived from calculated discretization intervals. These intervals are also returned in the (optional) `edges` parameter.

The supported discretization methods are defined in the `DSL_dataset::DiscretizeAlgorithm` enum:

- Hierarchical
- UniformWidth
- UniformCount

Discretization works on both continuous and discrete variables.

### 5.18 Learning

---

Learning in SMILE can perform two tasks:

- structure learning: create a new network from a dataset
- parameter learning: refine parameters (CPTs) in an existing network

SMILE also supports network validation, which is frequently used after learning to evaluate the results.

#### 5.18.1 Learning network structure

The following classes can be used to learn `DSL_network` from `DSL_dataset`:

- `DSL_bs`: Bayesian Search, a hill climbing procedure guided by scoring heuristic with random restarts
- `DSL_nb`: Naive Bayes

- DSL\_tan: Tree Augmented Naive Bayes, semi-naive method based on the Bayesian Search approach
- DSL\_abn: Augmented Naive Bayes, another semi-naive method based on the Bayesian Search approach

In the simplest scenario, just declare the object representing learning algorithm and call its Learn method:

```
DSL_dataset ds;  
ds.ReadFile("myfile.txt");  
DSL_network net;  
DSL_bs baySearch;  
int res = baySearch.Learn(ds, net);
```

If algorithm succeeds, DSL\_OKAY is returned and the DSL\_network passed as an argument to Learn is the output of the actual learning. Note that every variable in the dataset takes part in the learning process. If your data comes from the text file and you want to exclude some variables, use DSL\_dataset::RemoveVar.

After learning the structure, each of the algorithms listed above performs parameter learning with EM.

The code example above used the default settings for Bayesian Search. To tweak the learning process, you can set some public data members in the learning object before calling its Learn method. The example below sets the number of iterations and maximum number of parents:

```
DSL_bs baySearch;  
baySearch.nrIteration = 10;  
baySearch.maxParents = 4;  
int res = baySearch.Learn(ds, net);
```

The settings for the learning algorithms are described in detail in the [Reference](#)<sup>[98]</sup> section.

SMILE also contains the DSL\_pc class, which implements the PC structure learning algorithm (algorithm name is an acronym derived from its inventors' names). This algorithm also uses DSL\_dataset as data source, but instead of DSL\_network learns the DSL\_pattern object, which is a graph with directed and undirected edges, which is not guaranteed to be acyclic.

DSL\_bs and DSL\_pc objects contain a public data member of DSL\_bkgndKnowledge type. It can be used to pass the background knowledge to the learning algorithm. The background knowledge influences the learned structure by:

- forcing arcs between specified variables
- forbidding arcs between specified variables

- ordering specified variables by temporal tiers: in the resulting structure, there will be no arcs from nodes in higher tiers to nodes in lower tiers

The example below forces the arc between X and Y and forbids the arc between Z and Y. It is assumed that dataset contains the variables with the identifiers used in the calls to `DSL_dataset::FindVariable`.

```
DSL_network net;
DSL_bs baySearch;
int varX = ds.FindVariable("X");
int varY = ds.FindVariable("Y");
int varZ = ds.FindVariable("Z");
baySearch.bkk.forcedArcs.push_back(make_pair(varX, varY));
baySearch.bkk.forbiddenArcs.push_back(make_pair(varZ, varY));
res = baySearch.Learn(ds, net);
```

### 5.18.2 Learning network parameters

To learn the parameters in the existing `DSL_network` object, you can use the EM algorithm implemented in `DSL_em` class. As with structure learning, the data comes in `DSL_dataset` object. However, the network and the data must be matched to ensure that learning algorithm knows the relationship between the dataset variables and network nodes. If the variables and nodes have identical identifiers, you can use the `DSL_dataset::MatchNetwork` method:

```
DSL_dataset ds;
DSL_network net;
// load network and data here
vector<DSL_datasetMatch> matching;
string errMsg;
res = ds.MatchNetwork(net, matching, errMsg);
if (DSL_OKAY == res)
{
    DSL_em em;
    res = em.Learn(ds, net, matching);
}
```

If your network and data cannot be automatically matched with `MatchNetwork`, you can build the vector of `DSL_datasetMatch` structures in your own code. `DSL_datasetMatch` has `node` and `column` members representing node handle and variable index, respectively. For each node/variable pair you need one element of the matching vector.

`DSL_em::Learn` can be fine-tuned with various algorithm settings - see the [DSL\\_em](#)<sup>133</sup> reference section for details. The method can also return log likelihood, ranging from minus infinity to zero, which is a measure of fit of the model to the data:

```
DSL_em em;
double logLik;
res = em.Learn(ds, net, matching, &logLik);
```

It is possible to exclude some nodes from the learning. These fixed nodes do not change their CPTs during parameter learning.

```
DSL_em em;
double loglik;
vector<int> fixedNodes;
// add node handles to fixedNodes here
res = em.Learn(ds, net, matching, fixedNodes, &loglik);
```

### 5.18.3 Validation

To evaluate the predictive quality of your network you can use the `DSL_validator` class.

The `DSL_validator` constructor requires a references to `DSL_dataset` and `DSL_network` objects to be specified. To properly match the network and data the constructor also requires the vector of `DSL_datasetMatch` objects (as did `DSL_em::Learn` method).

After the validator object is constructed, you need to specify which nodes in the network are considered class nodes by calling `DSL_validator::AddClassNode` method. Validation requires at least one class node.

For each record in the dataset during the validation, the variables matched to non-class nodes are used to set the evidence. The posterior probabilities are then calculated and for each class node the outcome with the highest probability is selected as a predicted outcome. The prediction is compared with an outcome in the dataset variable associated with the class node. The number of matches and calculated posteriors are used to obtain the accuracy, confusion matrix, ROC and calibration curves.

Validation can be either performed without parameter learning using `DSL_validator::Test` method, or with parameter learning using `DSL_validator::KFold` and `LeaveOneOut` methods. K-fold crossvalidation divides the dataset into K parts of equal size, trains the network on K-1 parts, and tests it on the last, Kth part. The process is repeated K times, with a different part of the data being selected for testing. Leave-one-out is an extreme case of K-fold, in which K is equal to the number of records in the data set.

The example below performs K-fold crossvalidation with 5 folds using one class node.

```
DSL_dataset ds;
DSL_network net;
vector<DSL_datasetMatch> matching;
// load network and dataset, create the matching here
DSL_validator validator(ds, net, matching);
int classNodeHandle = net.FindNode("someNodeIdentifier");
validator.AddClassNode(classNodeHandle);
DSL_em em;
// optionally tweak the EM options here
int res = validator.KFold(em, 5);
if (DSL_OKAY == res)
{
```

```
double acc;  
validator.GetAccuracy(classNodeHandle, 0, acc);  
printf("Accuracy=%f\n", acc);  
}
```

See the [DSL\\_validator](#)<sup>130</sup> reference for more details.



# Tutorials

## 6 Tutorials

---

Each tutorial in this section is contained in a single `cpp` file. The file defines a function named `TutorialN` (with `N` being an ordinal number of the tutorial) and one or more helper functions. We show how the tutorials work by interleaving actual code with textual explanation. The complete code ready to be copied and pasted is located at the end of each subsection. C++ 03 standard is used for the broad compatibility with users' compilers. To ensure that tutorials are self-contained, there is some duplicated code defined in functions declared as static. For example, the function `CreateCptNode` is present in tutorials 1, 6 and 8.

If you want to create a program containing single tutorial only, add a simple `main` function at the bottom of the `tutorialN.cpp` file, for example:

```
int main()
{
    return Tutorial1();
}
```

To run all tutorials you can use the `main.cpp` file listed below. This approach requires compiling and linking `main.cpp` AND all of the `tutorialN.cpp` files.

### 6.1 main.cpp

---

```
// main.cpp
// Run all SMILE tutorials.

// smile_license.h contains your personal license key
#include "smile_license.h"

int Tutorial1();
int Tutorial2();
int Tutorial3();
int Tutorial4();
int Tutorial5();
int Tutorial6();
int Tutorial7();
int Tutorial8();

int main()
{
    static int (* const f[])() =
    {
        Tutorial1, Tutorial2, Tutorial3,
        Tutorial4, Tutorial5, Tutorial6,
        Tutorial7, Tutorial8
    };
    for (int i = 0; i < sizeof(f) / sizeof(f[0]); i++)
    {
        int r = f[i]();
        if (r)
        {
            return r;
        }
    }
}
```

```

    }
  }
  return 0;
}

```

## 6.2 Tutorial 1: Creating a Bayesian Network

Consider a slight twist on the problem described in the [Hello SMILE!](#)<sup>16</sup> section of this manual.

The twist will include adding an additional variable *State of the economy* (with the identifier *Economy*) with three outcomes (*Up*, *Flat*, and *Down*) modeling the developments in the economy. These developments are relevant to the decision, as they are impacting expert predictions. When the economy is heading *Up*, our expert makes more optimistic predictions, when it is heading *Down*, the expert makes more pessimistic predictions for the same venture. This is reflected by a directed arc from the node *State of the economy* to the node *Expert forecast*. *State of the economy* also impacts the probability of venture being successful, which we model by adding another arc.



We will show how to create this model using SMILE and how to save it to disk. In subsequent tutorials, we will show how to enter observations (evidence), how to perform inference, and how to retrieve the results of SMILE's calculations.

We start by redirecting error and warning messages to the console and declaring our network variable. The three nodes in the network are subsequently created by calling a helper function `CreateCptNode` declared beforehand and defined below the `Tutorial1` function.

```

DSL_errorH().RedirectToFile(stdout);
DSL_network net;

const char *ECONOMY_OUTCOMES[] = {"Up", "Flat", "Down", NULL};
int e = CreateCptNode(net, "Economy", "State of the economy",
    ECONOMY_OUTCOMES, 160, 40);

const char *SUCCESS_OUTCOMES[] = {"Success", "Failure", NULL};
int s = CreateCptNode(net, "Success", "Success of the venture",
    SUCCESS_OUTCOMES, 60, 40);

const char *FORECAST_OUTCOMES[] = {"Good", "Moderate", "Poor", NULL};
int f = CreateCptNode(net, "Forecast", "Expert forecast",
    FORECAST_OUTCOMES, 110, 140);

```

Before connecting the nodes with arcs, let's have a look at the `CreateCptNode` function.

```
static int CreateCptNode(DSL_network &net,
    const char *id, const char *name,
    const char *outcomes[], int xPos, int yPos)
{
    int handle = net.AddNode(DSL_CPT, id);
    DSL_node *node = net.GetNode(handle);

    DSL_idArray ida;
    for (const char **p = outcomes; *p != NULL; p++)
    {
        ida.Add(*p);
    }
    DSL_nodeDefinition *def = node->Definition();
    def->SetNumberOfOutcomes(ida);

    node->Info().Header().SetName(name);

    DSL_rectangle &position = node->Info().Screen().position;
    position.center_X = xPos;
    position.center_Y = yPos;
    position.width = 85;
    position.height = 55;

    return handle;
}
```

The function creates a CPT node with specified identifier, name, outcomes and position on the screen. The handle returned by `AddNode` is converted to `DSL_node*` pointer with a call to `GetNode`. CPT nodes are created with two outcomes named *State0* and *State1*. To change the number of node outcomes and rename them, we will call the `DSL_nodeDefinition::SetNumberOfOutcomes` (we could also use `DSL_nodeDefinition::AddOutcome` or `DSL_nodeDefinition::InsertOutcome`). The single parameter passed to `SetNumberOfOutcomes` is a reference to `DSL_idArray` object containing the outcome identifier. After outcomes, we set node's name and position. Function returns the handle of the node it created.

Back in the `Tutorial1` function, we add arcs between nodes.

```
net.AddArc(e, s);
net.AddArc(s, f);
net.AddArc(e, f);
```

Next step is to initialize the conditional probability tables of the nodes. See the [Multidimensional arrays](#)<sup>[25]</sup> section in this manual for the description of the CPT memory layout. For each of three nodes in our network we obtain a pointer to node definition object. To change the numbers (probabilities) in the CPT we will call `DSL_nodeDefinition::SetDefinition` method and pass a reference to a `DSL_doubleArray` as its single argument. When constructing `DSL_doubleArray` object we initialize its size to be equal to the CPT size. Next, we fill the array with probabilities. The code shown below for the *Success* node. Two other nodes are initialized in the same way.

```
DSL_nodeDefinition *successDef = net.GetNode(s)->Definition();
DSL_doubleArray sp(successDef->GetMatrix()->GetSize());
sp[0] = 0.3; // P(Success=S|Economy=U)
```

```

sp[1] = 0.7; // P(Success=F|Economy=U)
sp[2] = 0.2; // P(Success=S|Economy=F)
sp[3] = 0.8; // P(Success=F|Economy=F)
sp[4] = 0.1; // P(Success=S|Economy=D)
sp[5] = 0.9; // P(Success=F|Economy=D)
res = successDef->SetDefinition(sp);
if (DSL_OKAY != res)
{
    return res;
}
res = successDef->SetDefinition(sp);

```

With CPTs initialized our network is complete. We write its contents to the tutorial1.xdsl file. [Tutorial 2](#)<sup>63</sup> will load this file and perform the inference. The split between tutorials is artificial, your program can use networks right after its creation without the need to write/read from the file system.

```
res = net.WriteFile("tutorial1.xdsl");
```

### 6.2.1 tutorial1.cpp

```

// tutorial1.cpp
// Tutorial1 creates a simple network with three nodes,
// then saves it as XDSL file to disk.

#include "smile.h"
#include <cstdio>

static int CreateCptNode(
    DSL_network &net, const char *id,
    const char *name, const char *outcomes[],
    int xPos, int yPos);

int Tutorial1()
{
    printf("Starting Tutorial1...\n");

    // show errors and warnings in the console
    DSL_errorH().RedirectToFile(stdout);

    DSL_network net;

    const char *ECONOMY_OUTCOMES[] = { "Up", "Flat", "Down", NULL };
    int e = CreateCptNode(net, "Economy", "State of the economy",
        ECONOMY_OUTCOMES, 160, 40);

    const char *SUCCESS_OUTCOMES[] = { "Success", "Failure", NULL };
    int s = CreateCptNode(net, "Success", "Success of the venture",
        SUCCESS_OUTCOMES, 60, 40);

    const char *FORECAST_OUTCOMES[] = { "Good", "Moderate", "Poor", NULL };
    int f = CreateCptNode(net, "Forecast", "Expert forecast",
        FORECAST_OUTCOMES, 110, 140);

    net.AddArc(e, s);
}

```

```

net.AddArc(s, f);
net.AddArc(e, f);

DSL_nodeDefinition *economyDef = net.GetNode(e)->Definition();
DSL_doubleArray ep(economyDef->GetMatrix()->GetSize());
ep[0] = 0.2; // P(Economy=U)
ep[1] = 0.7; // P(Economy=F)
ep[2] = 0.1; // P(Economy=D)
int res = economyDef->SetDefinition(ep);
if (DSL_OKAY != res)
{
    return res;
}

DSL_nodeDefinition *successDef = net.GetNode(s)->Definition();
DSL_doubleArray sp(successDef->GetMatrix()->GetSize());
sp[0] = 0.3; // P(Success=S|Economy=U)
sp[1] = 0.7; // P(Success=F|Economy=U)
sp[2] = 0.2; // P(Success=S|Economy=F)
sp[3] = 0.8; // P(Success=F|Economy=F)
sp[4] = 0.1; // P(Success=S|Economy=D)
sp[5] = 0.9; // P(Success=F|Economy=D)
res = successDef->SetDefinition(sp);
if (DSL_OKAY != res)
{
    return res;
}

DSL_nodeDefinition *forecastDef = net.GetNode(f)->Definition();
DSL_doubleArray fp(forecastDef->GetMatrix()->GetSize());
fp[0] = 0.70; // P(Forecast=G|Success=S,Economy=U)
fp[1] = 0.29; // P(Forecast=M|Success=S,Economy=U)
fp[2] = 0.01; // P(Forecast=P|Success=S,Economy=U)

fp[3] = 0.65; // P(Forecast=G|Success=S,Economy=F)
fp[4] = 0.30; // P(Forecast=M|Success=S,Economy=F)
fp[5] = 0.05; // P(Forecast=P|Success=S,Economy=F)

fp[6] = 0.60; // P(Forecast=G|Success=S,Economy=D)
fp[7] = 0.30; // P(Forecast=M|Success=S,Economy=D)
fp[8] = 0.10; // P(Forecast=P|Success=S,Economy=D)

fp[9] = 0.15; // P(Forecast=G|Success=F,Economy=U)
fp[10] = 0.30; // P(Forecast=M|Success=F,Economy=U)
fp[11] = 0.55; // P(Forecast=P|Success=F,Economy=U)

fp[12] = 0.10; // P(Forecast=G|Success=F,Economy=F)
fp[13] = 0.30; // P(Forecast=M|Success=F,Economy=F)
fp[14] = 0.60; // P(Forecast=P|Success=F,Economy=F)

fp[15] = 0.05; // P(Forecast=G|Success=F,Economy=D)
fp[16] = 0.25; // P(Forecast=G|Success=F,Economy=D)
fp[17] = 0.70; // P(Forecast=G|Success=F,Economy=D)

res = forecastDef->SetDefinition(fp);
if (DSL_OKAY != res)
{

```

```

        return res;
    }

    res = net.WriteFile("tutorial1.xdsl");
    if (DSL_OKAY != res)
    {
        return res;
    }

    printf("Tutorial1 complete: Network written to tutorial1.xdsl\n");
    return DSL_OKAY;
}

static int CreateCptNode(
    DSL_network &net, const char *id, const char *name,
    const char *outcomes[], int xPos, int yPos)
{
    int handle = net.AddNode(DSL_CPT, id);
    DSL_node *node = net.GetNode(handle);

    DSL_idArray ida;
    for (const char **p = outcomes; *p != NULL; p++)
    {
        ida.Add(*p);
    }
    DSL_nodeDefinition *def = node->Definition();
    def->SetNumberOfOutcomes(ida);

    node->Info().Header().SetName(name);

    DSL_rectangle &position = node->Info().Screen().position;
    position.center_X = xPos;
    position.center_Y = yPos;
    position.width = 85;
    position.height = 55;

    return handle;
}

```

## 6.3 Tutorial 2: Inference with a Bayesian Network

---

Tutorial 2 starts with the model we have previously created. We will perform multiple calls to Bayesian inference algorithm through the `DSL_network::UpdateBeliefs`, starting with network without any evidence. After each `UpdateBeliefs` call the posterior probabilities of nodes will be displayed.

The `Tutorial2` function itself is very simple and delegates work to helper functions declared at the top of the file and defined below `Tutorial2`.

The model is loaded with the `DSL_network::ReadFile`. We exit the program if the status code `ReadFile` returned is not `DSL_OKAY`.

```

DSL_network net;
int res = net.ReadFile("tutorial1.xdsl");
if (DSL_OKAY != res)
{
    printf("Load failed, did you run Tutorial1 before Tutorial2?\n");
    return res;
}

```

UpdateBeliefs is followed by the call to PrintAllPosteriors helper.

```

printf("Posteriors with no evidence set:\n");
net.UpdateBeliefs();
PrintAllPosteriors(net);

```

PrintAllPosteriors displays posterior probabilities calculated by UpdateBeliefs and stored in node values. To iterate over the nodes, DSL\_network::GetFirstNode and GetNextNode are used. In the body of the loop we call another locally defined helper function, PrintPosteriors.

```

for (int h = net.GetFirstNode(); h >= 0; h = net.GetNextNode(h))
{
    PrintPosteriors(net, h);
}

```

PrintPosteriors converts node handle to node pointer. Then it checks if node has evidence set; if this is the case the name of the evidence state is displayed. Otherwise the function iterates over all states and displays the posterior probability of each.

```

static void PrintPosteriors(DSL_network &net, int handle)
{
    DSL_node *node = net.GetNode(handle);
    const char *nodeId = node->GetId();
    const DSL_idArray &outcomeIds
        = *node->Definition()->GetOutcomesNames();
    DSL_nodeValue *val = node->Value();
    if (val->IsEvidence())
    {
        printf("%s has evidence set (%s)\n",
            nodeId, outcomeIds[val->GetEvidence()]);
    }
    else
    {
        const DSL_Dmatrix &posteriors = *val->GetMatrix();
        for (int i = 0; i < posteriors.GetSize(); i++)
        {
            printf("P(%s=%s)=%g\n", nodeId, outcomeIds[i], posteriors[i]);
        }
    }
}

```

Going back to Tutorial2 function, we repeatedly call another locally defined helper function to change evidence, update network and display posteriors:

```

printf("\nSetting Forecast=Good.\n");
ChangeEvidenceAndUpdate(net, "Forecast", "Good");
printf("\nAdding Economy=Up.\n");
ChangeEvidenceAndUpdate(net, "Economy", "Up");
printf("\nChanging Forecast to Poor, keeping Economy=Up.\n");
ChangeEvidenceAndUpdate(net, "Forecast", "Poor");

```



```
printf("\nRemoving evidence from Economy, keeping Forecast=Poor.\n");
ChangeEvidenceAndUpdate(net, "Economy", NULL);
```

Let us examine the ChangeEvidenceAndUpdate helper.

```
static int ChangeEvidenceAndUpdate(DSL_network &net,
    const char *nodeId, const char *outcomeId)
{
    int res = SetEvidenceById(net, nodeId, outcomeId);
    if (DSL_OKAY != res)
    {
        return res;
    }
    res = net.UpdateBeliefs();
    if (DSL_OKAY != res)
    {
        return res;
    }
    PrintAllPosteriors(net);
    return DSL_OKAY;
}
```

It is just a series of calls to SetEvidenceById, DSL\_network::UpdateBeliefs and PrintAllPosteriors. Let us check the SetEvidenceById helper now.

```
static int SetEvidenceById(DSL_network &net,
    const char *nodeId, const char *outcomeId)
{
    int handle = net.FindNode(nodeId);
    if (handle < 0)
    {
        return handle;
    }

    DSL_node *node = net.GetNode(handle);

    if (NULL == outcomeId)
    {
        return node->Value()->ClearEvidence();
    }
    else
    {
        DSL_nodeDefinition *def = node->Definition();
        int idx = def->GetOutcomesNames()->FindPosition(outcomeId);
        if (idx < 0)
        {
            return idx;
        }

        return node->Value()->SetEvidence(idx);
    }
}
```

The function converts human-readable node identifier passed as its 2nd argument to a node handle with a call to DSL\_network::FindNode. If the 3rd argument (the outcome identifier) is NULL, evidence for the node is cleared. Otherwise the identifier is converted to its index; the index is used to set the evidence with DSL\_nodeValue::SetEvidence.

### 6.3.1 tutorial2.cpp

```
// tutorial2.cpp
// Tutorial2 loads the XDSL file created by Tutorial1,
// then performs the series of inference calls,
// changing evidence each time.

#include "smile.h"
#include <cstdio>

static int ChangeEvidenceAndUpdate(
    DSL_network &net, const char *nodeId, const char *outcomeId);
static void PrintAllPosteriors(DSL_network &net);

int Tutorial2()
{
    printf("Starting Tutorial2...\n");

    DSL_errorH().RedirectToFile(stdout);

    // load the network created by Tutorial1
    DSL_network net;
    int res = net.ReadFile("tutorial1.xdsl");
    if (DSL_OKAY != res)
    {
        printf(
            "Network load failed, did you run Tutorial1 before Tutorial2?\n");
        return res;
    }

    printf("Posteriors with no evidence set:\n");
    net.UpdateBeliefs();
    PrintAllPosteriors(net);

    printf("\nSetting Forecast=Good.\n");
    ChangeEvidenceAndUpdate(net, "Forecast", "Good");

    printf("\nAdding Economy=Up.\n");
    ChangeEvidenceAndUpdate(net, "Economy", "Up");

    printf("\nChanging Forecast to Poor, keeping Economy=Up.\n");
    ChangeEvidenceAndUpdate(net, "Forecast", "Poor");

    printf("\nRemoving evidence from Economy, keeping Forecast=Poor.\n");
    ChangeEvidenceAndUpdate(net, "Economy", NULL);

    printf("\nTutorial2 complete.\n");
    return DSL_OKAY;
}

static int SetEvidenceById(
    DSL_network &net, const char *nodeId, const char *outcomeId)
{
    int handle = net.FindNode(nodeId);
```

```

    if (handle < 0)
    {
        return handle;
    }

    DSL_node *node = net.GetNode(handle);

    if (NULL == outcomeId)
    {
        return node->Value()->ClearEvidence();
    }
    else
    {
        int idx =
            node->Definition()->GetOutcomesNames()->FindPosition(outcomeId);
        if (idx < 0)
        {
            return idx;
        }

        return node->Value()->SetEvidence(idx);
    }
}

static void PrintPosteriors(DSL_network &net, int handle)
{
    DSL_node *node = net.GetNode(handle);
    const char *nodeId = node->GetId();
    const DSL_idArray &outcomeIds = *node->Definition()->GetOutcomesNames();
    DSL_nodeValue *val = node->Value();
    if (val->IsEvidence())
    {
        printf("%s has evidence set (%s)\n",
            nodeId, outcomeIds[val->GetEvidence()]);
    }
    else
    {
        const DSL_Dmatrix &posteriors = *val->GetMatrix();
        for (int i = 0; i < posteriors.GetSize(); i++)
        {
            printf("P(%s=%s)=%g\n", nodeId, outcomeIds[i], posteriors[i]);
        }
    }
}

static void PrintAllPosteriors(DSL_network &net)
{
    for (int h = net.GetFirstNode(); h >= 0; h = net.GetNextNode(h))
    {
        PrintPosteriors(net, h);
    }
}

static int ChangeEvidenceAndUpdate(

```

```

    DSL_network &net, const char *nodeId, const char *outcomeId)
{
    int res = SetEvidenceById(net, nodeId, outcomeId);
    if (DSL_OKAY != res)
    {
        return res;
    }
    res = net.UpdateBeliefs();
    if (DSL_OKAY != res)
    {
        return res;
    }
    PrintAllPosteriors(net);
    return DSL_OKAY;
}

```

## 6.4 Tutorial 3: Exploring the contents of a model

---

This tutorial will not perform any calculations. Instead, we will display the information about the network structure (nodes and arcs) and parameters (in this case, conditional probability tables). The Tutorial3 function itself is again very simple. We load the network created by Tutorial1 and for each node invoke the locally defined helper function PrintNodeInfo. This is where real work is done. The first node attribute displayed is its name, stored in node header object.

```

DSL_node *node = net.GetNode(nodeHandle);
printf("Node: %s\n", node->Info().Header().GetName());

```

Identifiers of the outcomes are next, retrieved from node definition

```

printf(" Outcomes:");
const DSL_idArray &outcomes = *node->Definition()->GetOutcomesNames();
for (int i = 0; i < outcomes.NumItems(); i++)
{
    printf(" %s", outcomes[i]);
}

```

The parents of the node follow.

```

const DSL_intArray &parents = net.GetParents(nodeHandle);
if (parents.NumItems() > 0)
{
    printf(" Parents:");
    for (int i = 0; i < parents.NumItems(); i++)
    {
        printf(" %s", net.GetNode(parents[i])->GetId());
    }
    printf("\n");
}

```

Node's children are next. The code fragment is virtually identical to the iteration over parents above.

Finally, the node definition is checked. If its type is DSL\_CPT (general chance node) or DSL\_TRUTHTABLE (deterministic discrete node), we retrieve the probabilities from node's definition

with the `DSL_nodeDefinition::GetMatrix`. Note that all the nodes in the network created in [Tutorial 1](#)<sup>[59]</sup> are `DSL_CPT`. The if statement is there to keep the function general enough to be copied and pasted into other program using SMILE.

```
DSL_nodeDefinition *def = node->Definition();
int defType = def->GetType();
printf(" Definition type: %s\n", def->GetTypeName());
if (DSL_CPT == defType || DSL_TRUTHTABLE == defType)
{
    const DSL_Dmatrix &cpt = *def->GetMatrix();
    PrintMatrix(net, cpt, outcomes, parents);
}
```

`PrintMatrix` is another locally defined helper function. It iterates over entries in the CPT. For each entry, the outcome of the node and its parents' outcomes are displayed along with the probability value. Let us inspect the main loop of the function:

```
for (int elemIdx = 0; elemIdx < mtx.GetSize(); elemIdx++)
{
    const char *outcome = outcomes[coords[dimCount - 1]];
    printf("    P(%s", outcome);

    if (dimCount > 1)
    {
        printf(" | ");
        for (int parentIdx = 0; parentIdx < dimCount - 1; parentIdx++)
        {
            if (parentIdx > 0) printf(",");
            DSL_node *parentNode = net.GetNode(parents[parentIdx]);
            const DSL_idArray &parentOutcomes =
                *parentNode->Definition()->GetOutcomesNames();
            printf("%s=%s", parentNode->GetId(),
                parentOutcomes[coords[parentIdx]]);
        }
    }
    double prob = mtx[elemIdx];
    printf(")=%g\n", prob);
    mtx.NextCoordinates(coords);
}
```

The loop uses both linear index (`elemIdx` variable) and multidimensional coordinates (`coords` variable of `DSL_intArray` type, initialized to zeros before the loop). Both are kept in sync, the equivalent of increasing linear index by one (`elemIdx++`) is a call to `NextCoordinates` (`mtx.NextCoordinates(coords)`). Note that we could use `elemIdx` to convert into coordinates during each iteration with `IndexToCoordinates`. Conversely, it is also possible to convert `coords` into linear index with `CoordinatesToIndex`. While not significant for this tutorial, the coordinate-based loop performance over large CPTs is better when `NextCoordinates` approach is used (`NextCoordinates` is more economical than `IndexToCoordinates`). Of course, the plain linear index will be even faster.

The node outcome is the rightmost entry in the coordinates. The parents' outcome indexes start from the left at index 0 in the coords. Part of the Tutorial3 output for the node *Forecast* is show below. All lines starting with "P"( were printed by PrintMatrix.

```
Node: Expert forecast
Outcomes: Good Moderate Poor
Parents: Success Economy
Definition type: CPT
P(Good | Success=Success,Economy=Up)=0.7
P(Moderate | Success=Success,Economy=Up)=0.29
P(Poor | Success=Success,Economy=Up)=0.01
P(Good | Success=Success,Economy=Flat)=0.65
P(Moderate | Success=Success,Economy=Flat)=0.3
P(Poor | Success=Success,Economy=Flat)=0.05
P(Good | Success=Success,Economy=Down)=0.6
P(Moderate | Success=Success,Economy=Down)=0.3
P(Poor | Success=Success,Economy=Down)=0.1
P(Good | Success=Failure,Economy=Up)=0.15
P(Moderate | Success=Failure,Economy=Up)=0.3
P(Poor | Success=Failure,Economy=Up)=0.55
P(Good | Success=Failure,Economy=Flat)=0.1
P(Moderate | Success=Failure,Economy=Flat)=0.3
P(Poor | Success=Failure,Economy=Flat)=0.6
P(Good | Success=Failure,Economy=Down)=0.05
P(Moderate | Success=Failure,Economy=Down)=0.25
P(Poor | Success=Failure,Economy=Down)=0.7
```

### 6.4.1 tutorial3.cpp

```
// tutorial3.cpp
// Tutorial3 loads the XDSL file and prints the information
// about the structure (nodes and arcs) and the parameters
// (conditional probabilities of the nodes) of the network.

#include "smile.h"
#include <cstdio>

static void PrintNodeInfo(DSL_network &net, int nodeHandle);

int Tutorial3()
{
    printf("Starting Tutorial3...\n");

    DSL_errorH().RedirectToFile(stdout);

    // load the network created by Tutorial1
    DSL_network net;
    int res = net.ReadFile("tutorial1.xdsl");
    if (DSL_OKAY != res)
    {
        printf(
            "Network load failed, did you run Tutorial1 before Tutorial3?\n");
        return res;
    }
}
```

```

    for (int h = net.GetFirstNode(); h >= 0; h = net.GetNextNode(h))
    {
        PrintNodeInfo(net, h);
    }

    printf("\nTutorial3 complete.\n");
    return DSL_OKAY;
}

// PrintMatrix displays each probability entry in the matrix in the separate
// line, preceded by the information about node and parent outcomes the entry
// relates to.
// The coordinates of the matrix are ordered as P1,...,Pn,S
// where Pi is the outcome index of i-th parent and S is the outcome of the node
// for which this matrix is the CPT.
static void PrintMatrix(
    DSL_network &net, const DSL_Dmatrix &mtx,
    const DSL_idArray &outcomes, const DSL_intArray &parents)
{
    int dimCount = mtx.GetNumberOfDimensions();
    DSL_intArray coords(dimCount);
    coords.FillWith(0);

    // elemIdx and coords will be moving in sync
    for (int elemIdx = 0; elemIdx < mtx.GetSize(); elemIdx++)
    {
        const char *outcome = outcomes[coords[dimCount - 1]];
        printf("    P(%s", outcome);

        if (dimCount > 1)
        {
            printf(" | ");
            for (int parentIdx = 0; parentIdx < dimCount - 1; parentIdx++)
            {
                if (parentIdx > 0) printf(",");
                DSL_node *parentNode = net.GetNode(parents[parentIdx]);
                const DSL_idArray &parentOutcomes =
                    *parentNode->Definition()->GetOutcomesNames();
                printf("%s=%s",
                    parentNode->GetId(), parentOutcomes[coords[parentIdx]]);
            }
        }

        double prob = mtx[elemIdx];
        printf(")=%g\n", prob);

        mtx.NextCoordinates(coords);
    }
}

// PrintNodeInfo displays node attributes:
// name, outcome ids, parent ids, children ids, CPT probabilities
static void PrintNodeInfo(DSL_network &net, int nodeHandle)
{

```

```

DSL_node *node = net.GetNode(nodeHandle);
printf("Node: %s\n", node->Info().Header().GetName());

printf(" Outcomes:");
const DSL_idArray &outcomes = *node->Definition()->GetOutcomesNames();
for (int i = 0; i < outcomes.NumItems(); i++)
{
    printf(" %s", outcomes[i]);
}
printf("\n");

const DSL_intArray &parents = net.GetParents(nodeHandle);
if (parents.NumItems() > 0)
{
    printf(" Parents:");
    for (int i = 0; i < parents.NumItems(); i++)
    {
        printf(" %s", net.GetNode(parents[i])->GetId());
    }
    printf("\n");
}

const DSL_intArray &children = net.GetChildren(nodeHandle);
if (children.NumItems() > 0)
{
    printf(" Children:");
    for (int i = 0; i < children.NumItems(); i++)
    {
        printf(" %s", net.GetNode(children[i])->GetId());
    }
    printf("\n");
}

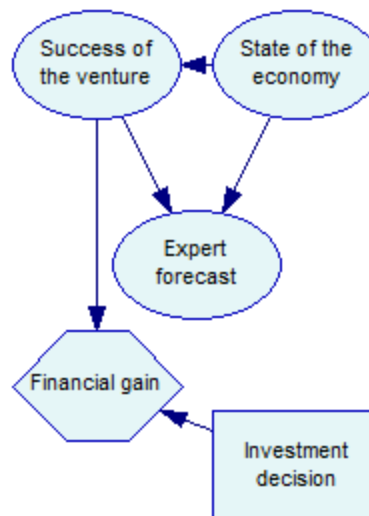
DSL_nodeDefinition *def = node->Definition();
int defType = def->GetType();
printf(" Definition type: %s\n", def->GetTypeNames());
if (DSL_CPT == defType || DSL_TRUTHTABLE == defType)
{
    const DSL_Dmatrix &cpt = *def->GetMatrix();
    PrintMatrix(net, cpt, outcomes, parents);
}
}

```

## 6.5 Tutorial 4: Creating the Influence Diagram

We will further expand the model created in [Tutorial 1](#)<sup>59</sup> and turn it into an influence diagram. To this effect, we will add a decision node *Investment decision* and a utility node *Financial gain*. The decision will have two possible states: *Invest* and *DoNotInvest*, which will be the two decision options under consideration. Which option is chosen will impact the financial gain and this will be reflected by a directed arc from *Investment decision* to *Financial gain*. Whether the venture succeeds or fails will also impact the financial gain and this will be also reflected by a directed arc from *Success of the venture* to *Financial gain*.





We will show how to create this model using SMILE and how to save it to disk. In the subsequent tutorial, we will show how to enter observations (evidence), how to perform inference, and how to retrieve the utilities calculated for the *Financial gain* node.

The program starts by reading the file, just like [Tutorial 2](#)<sup>63</sup> and [Tutorial 3](#)<sup>68</sup>. We convert the identifier of the node *Success* to node handle, which we will use later to create an arc between *Success* and *Gain*. Two new nodes will be created by calling a `CreateNode` helper function, which is a slightly modified version of the `CreateCptNode` from [Tutorial 1](#)<sup>59</sup>. The difference is that we now want to create different types of nodes. Therefore, `CreateNode` has one additional input parameter, an integer for specifying the node type. Another difference is that `CreateNode` needs to be able to add utility nodes, which do not have outcomes. The function checks for the value of its outcomes parameter and if it is `NULL`, the call to `DSL_nodeDefinition::SetNumberOfOutcomes` is skipped.

```

static int CreateNode(DSL_network &net, int nodeType, const char *id,
    const char *name, const char *outcomes[], int xPos, int yPos)
{
    int handle = net.AddNode(nodeType, id);
    DSL_node *node = net.GetNode(handle);
    if (NULL != outcomes)
        /... the rest of the function is unchanged
    }
}

```

Back in `Tutorial4` function, we add nodes and arcs:

```

const char *INVEST_DECISIONS[] = { "Invest", "DoNotInvest", NULL };
int i = CreateNode(net, DSL_LIST,
    "Invest", "Investment decision", INVEST_DECISIONS, 160, 240);

int g = CreateNode(net, DSL_TABLE,
    "Gain", "Financial gain", NULL, 60, 200);

net.AddArc(i, g);
net.AddArc(s, g);

```

Note that DSL\_LIST is the node type identifier for decision nodes. DSL\_TABLE is the node type identifier for utility nodes. Decision nodes do not have numeric parameters, but utility nodes do. The structure of the matrix in utility node's definition is similar to the CPT with the exception of last dimension being always set to one (as there are no outcomes). Node Gain has two parents with two outcomes each and size of its definition is  $2 \times 2 \times 1 = 4$ . The program specifies four numbers for the utilities.

```
DSL_nodeDefinition *gainDef = net.GetNode(g)->Definition();
DSL_doubleArray gu(gainDef->GetMatrix()->GetSize());
gu[0] = 10000; gu[1] = -5000; gu[2] = 500; gu[3] = 500;
res = gainDef->SetDefinition(gu);
```

The influence diagram is now complete. We write its contents to file and exit the function. [Tutorial 5](#)<sup>76</sup> will load the file and perform the inference.

### 6.5.1 tutorial4.cpp

```
// tutorial4.cpp
// Tutorial4 loads the XDSL file file created by Tutorial1
// and adds decision and utility nodes, which transforms
// a Bayesian Network (BN) into an Influence Diagram (ID).

#include "smile.h"
#include <cstdio>

static int CreateNode(
    DSL_network &net, int nodeType, const char *id, const char *name,
    const char *outcomes[], int xPos, int yPos);

int Tutorial4()
{
    printf("Starting Tutorial4...\n");

    DSL_errorH().RedirectToFile(stdout);

    DSL_network net;
    // load the network created by Tutorial1
    int res = net.ReadFile("tutorial1.xdsl");
    if (DSL_OKAY != res)
    {
        printf(
            "Network load failed, did you run Tutorial1 before Tutorial4?\n");
        return res;
    }

    int s = net.FindNode("Success");
    if (s < 0)
    {
        printf("Success node not found.");
        return s;
    }

    const char *INVEST_DECISIONS[] = { "Invest", "DoNotInvest", NULL };
```

```

int i = CreateNode(net, DSL_LIST, "Invest", "Investment decision",
    INVEST_DECISIONS, 160, 240);

int g = CreateNode(net, DSL_TABLE, "Gain", "Financial gain", NULL, 60, 200);

net.AddArc(i, g);
net.AddArc(s, g);

DSL_nodeDefinition *gainDef = net.GetNode(g)->Definition();
DSL_doubleArray gu(gainDef->GetMatrix()->GetSize());
gu[0] = 10000; gu[1] = -5000; gu[2] = 500; gu[3] = 500;
res = gainDef->SetDefinition(gu);
if (DSL_OKAY != res)
{
    return res;
}

res = net.WriteFile("tutorial4.xdsl");
if (DSL_OKAY != res)
{
    return res;
}

printf("Tutorial4 complete: Influence diagram written to tutorial4.xdsl\n");
return DSL_OKAY;
}

static int CreateNode(
    DSL_network &net, int nodeType, const char *id, const char *name,
    const char *outcomes[], int xPos, int yPos)
{
    int handle = net.AddNode(nodeType, id);
    DSL_node *node = net.GetNode(handle);

    if (NULL != outcomes)
    {
        DSL_nodeDefinition *def = node->Definition();
        DSL_idArray ida;
        for (const char **p = outcomes; *p != NULL; p++)
        {
            ida.Add(*p);
        }
        def->SetNumberOfOutcomes(ida);
    }

    node->Info().Header().SetName(name);

    DSL_rectangle &position = node->Info().Screen().position;
    position.center_X = xPos;
    position.center_Y = yPos;
    position.width = 85;
    position.height = 55;

    return handle;
}

```

## 6.6 Tutorial 5: Inference in an Influence Diagram

This tutorial loads the influence diagram that we have created in [Tutorial 4](#)<sup>[72]</sup>. We will perform multiple inference calls and display calculated utilities.

The tutorial starts with now-familiar sequence of redirecting error messages, loading the file and obtaining the handle to the node *Financial gain*. When we invoke `UpdateBeliefs` for the first time, the model has no evidence. A local helper function, `PrintFinancialGain`, is called to print out the utilities.

```
static void PrintFinancialGain(DSL_network &net, int gainHandle)
{
    DSL_node *node = net.GetNode(gainHandle);
    const char *nodeName = node->Info().Header().GetName();
    printf("%s:\n", nodeName);
    DSL_nodeValue *val = node->Value();
    const DSL_Dmatrix &mtx = *val->GetMatrix();
    const DSL_intArray &parents = val->GetIndexingParents();
    PrintMatrix(net, mtx, "Utility", NULL, parents);
}
```

The function prints the name of the node specified by its 2nd parameter (which points always to node *Financial gain* in this tutorial), then prepares the input arguments for the `PrintMatrix` function: the node value matrix (utilities) and an array with handles of nodes indexing the utilities (these are all uninstantiated decision nodes and all nodes that have not been observed but should have been observed because they have outgoing arcs that enter decision nodes). `PrintMatrix` in this manual is slightly modified version of the function from [Tutorial 3](#)<sup>[68]</sup>. The changes are needed to properly display utility matrix with its last dimension set to 1, as utility nodes have no outcomes.

The utilities without evidence suggest that we should not invest:

```
Financial gain:
Utility(Invest=Invest)=-1850
Utility(Invest=DoNotInvest)=500
```

Next, we model the analyst's forecast to be good by calling local helper function `SetEvidenceById`. It made its first appearance in [Tutorial 3](#)<sup>[68]</sup>, we are using it unchanged here.

```
SetEvidenceById(net, "Forecast", "Good");
if (DSL_OKAY != net.UpdateBeliefs())
{
    return res;
}
PrintFinancialGain(net, gain);
```

The utilities have changed; with good forecast the optimal decision is to invest:

```
Financial gain:
Utility(Invest=Invest)=4455.78
Utility(Invest=DoNotInvest)=500
```

Now we observe the state of the economy and conclude that it is growing.

```
SetEvidenceById(net, "Economy", "Up");
```

```

if (DSL_OKAY != net.UpdateBeliefs())
{
    return res;
}
PrintFinancialGain(net, gain);

```

Growing economy makes our chances even better:

```

Financial gain:
    Utility(Invest=Invest)=5000
    Utility(Invest=DoNotInvest)=500

```

This concludes Tutorial 5.

### 6.6.1 tutorial5.cpp

```

// tutorial5.cpp
// Tutorial5 loads the XDSL file created by Tutorial4,
// then performs the series of inference calls,
// changing evidence each time.

#include "smile.h"
#include <cstdio>

static int SetEvidenceById(
    DSL_network &net, const char *nodeId, const char *outcomeId);
static void PrintFinancialGain(DSL_network &net, int gainHandle);

int Tutorial5()
{
    printf("Starting Tutorial5...\n");

    DSL_errorH().RedirectToFile(stdout);

    DSL_network net;
    // load the network created by Tutorial4
    int res = net.ReadFile("tutorial4.xdsl");
    if (DSL_OKAY != res)
    {
        printf(
            "Network load failed, did you run Tutorial4 before Tutorial5?\n");
        return res;
    }

    int gain = net.FindNode("Gain");
    if (gain < 0)
    {
        printf("Gain node not found.");
        return gain;
    }

    printf("No evidence set.\n");
    if (DSL_OKAY != net.UpdateBeliefs())
    {
        return res;
    }
}

```

```

    }
    PrintFinancialGain(net, gain);

    printf("\nSetting Forecast=Good.\n");
    SetEvidenceById(net, "Forecast", "Good");
    if (DSL_OKAY != net.UpdateBeliefs())
    {
        return res;
    }
    PrintFinancialGain(net, gain);

    printf("\nAdding Economy=Up\n");
    SetEvidenceById(net, "Economy", "Up");
    if (DSL_OKAY != net.UpdateBeliefs())
    {
        return res;
    }
    PrintFinancialGain(net, gain);

    printf("\nTutorial5 complete.\n");
    return DSL_OKAY;
}

// PrintMatrix displays each probability entry in the matrix in the separate
// line, preceded by the information about node and parent outcomes the entry
// relates to.
// The coordinates of the matrix are ordered as P1,...,Pn,S
// where Pi is the outcome index of i-th parent and S is the outcome of the node.
// If node type is utility, then S collapses and the last coordinate is
// always zero.
static void PrintMatrix(
    DSL_network &net, const DSL_Dmatrix &mtx, const char *prefix,
    const DSL_idArray *outcomes, const DSL_intArray &parents)
{
    int dimCount = mtx.GetNumberOfDimensions();
    DSL_intArray coords(dimCount);
    coords.FillWith(0);

    // elemIdx and coords will be moving in sync
    for (int elemIdx = 0; elemIdx < mtx.GetSize(); elemIdx++)
    {
        if (NULL != outcomes)
        {
            const char *outcome = (*outcomes)[coords[dimCount - 1]];
            printf("    %s(%s", prefix, outcome);
        }
        else
        {
            printf("    %s(", prefix);
        }

        if (dimCount > 1)
        {
            if (NULL != outcomes)
            {

```

```

        printf("|");
    }

    for (int parentIdx = 0; parentIdx < dimCount - 1; parentIdx++)
    {
        if (parentIdx > 0) printf(",");
        DSL_node *parentNode = net.GetNode(parents[parentIdx]);
        const DSL_idArray &parentOutcomes =
            *parentNode->Definition()->GetOutcomesNames();
        printf("%s=%s",
            parentNode->GetId(), parentOutcomes[coords[parentIdx]]);
    }
}

double prob = mtx[elemIdx];
printf(")=%g\n", prob);

mtx.NextCoordinates(coords);
}
}

static void PrintFinancialGain(DSL_network &net, int gainHandle)
{
    DSL_node *node = net.GetNode(gainHandle);
    const char *nodeName = node->Info().Header().GetName();
    printf("%s:\n", nodeName);
    DSL_nodeValue *val = node->Value();
    const DSL_Dmatrix &mtx = *val->GetMatrix();
    const DSL_intArray &parents = val->GetIndexingParents();
    PrintMatrix(net, mtx, "Utility", NULL, parents);
}

static int SetEvidenceById(
    DSL_network &net, const char *nodeId, const char *outcomeId)
{
    int handle = net.FindNode(nodeId);
    if (handle < 0)
    {
        return handle;
    }

    DSL_node *node = net.GetNode(handle);

    if (NULL == outcomeId)
    {
        return node->Value()->ClearEvidence();
    }
    else
    {
        int idx =
            node->Definition()->GetOutcomesNames()->FindPosition(outcomeId);
        if (idx < 0)
        {
            return idx;
        }
    }
}

```

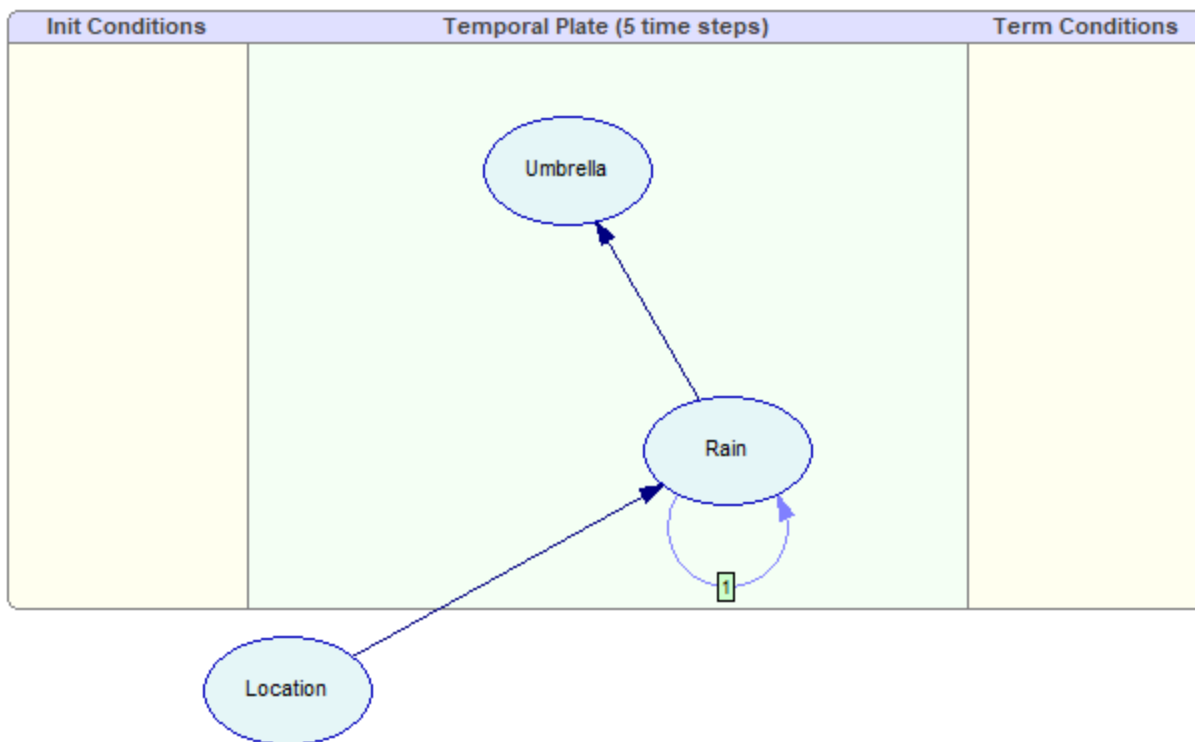
```

    return node->Value()->SetEvidence(idx);
  }
}

```

## 6.7 Tutorial 6: Dynamic model

Consider the following example, inspired by (Russell & Norvig, 1995), in which a security guard at some secret underground installation works on a shift of seven days and wants to know whether it is raining on the day of her return to the outside world. Her only access to the outside world occurs each morning when she sees the director coming in, with or without an umbrella. Furthermore, she knows that the government has two secret underground installations: one in Pittsburgh and one in the Sahara, but she does not know which one she is guarding. For each day  $t$ , the set of evidence contains a single variable  $Umbrella_t$  (observation of an umbrella carried by the director) and the set of unobservable variables contains  $Rain_t$  (a propositional variable with two states *true* and *false*, denoting whether it is raining) and  $Location$  (with two possible states: *Pittsburgh* and *Sahara*). The prior probability of rain depends on the geographical location and on whether it rained on the previous day. For simplicity, we do not use the initial and terminal condition nodes in this tutorial.



The contains three discrete chance nodes (CPT), created with a call to `CreateCptNode` helper function, just like in nodes in [Tutorial 1](#)<sup>[59]</sup>. The *Rain* and *Umbrella* nodes are marked as belonging to the temporal plate by calling `DSL_network::SetTemporalType`:

```

net.SetTemporalType(rain, dsl_temporalType::dsl_plateNode);
net.SetTemporalType(umb, dsl_temporalType::dsl_plateNode);

```



Two of the three arcs in the model are created by `DSL_network::AddArc`. The temporal arc expressing the dependency of  $Rain_t$  on  $Rain_{t-1}$  is created by the `AddTemporalArc` method. Please note the temporal order of the arc passed as the 3rd parameter:

```
net.AddArc(loc, rain);
net.AddTemporalArc(rain, rain, 1);
net.AddArc(rain, umb);
```

CPTs for the nodes are initialized with `DSL_nodeDefinition::SetDefinition`, as in Tutorial 1. However, the node *Rain* requires **two** CPTs, because it has an incoming temporal arc of order 1. The program fills the `DSL_doubleArray` object with probabilities and calls `DSL_nodeDefinition::SetTemporalDefinition`:

```
rainProbs.SetSize(rainDef->GetTemporalDefinition(1)->GetSize());
rainProbs[0] = 0.7; // P(Rain=true | Location=Pittsburgh, Rain[t-1]=true)
rainProbs[1] = 0.3; // P(Rain=false | Location=Pittsburgh, Rain[t-1]=true)
rainProbs[2] = 0.3; // P(Rain=true | Location=Pittsburgh, Rain[t-1]=false)
rainProbs[3] = 0.7; // P(Rain=false | Location=Pittsburgh, Rain[t-1]=false)
rainProbs[4] = 0.001; // P(Rain=true | Location=Sahara, Rain[t-1]=true)
rainProbs[5] = 0.999; // P(Rain=false | Location=Sahara, Rain[t-1]=true)
rainProbs[6] = 0.01; // P(Rain=true | Location=Sahara, Rain[t-1]=false)
rainProbs[7] = 0.99; // P(Rain=false | Location=Sahara, Rain[t-1]=false)
res = rainDef->SetTemporalDefinition(1, rainProbs);
```

Finally, we adjust the number of slices created during the network unrolling:

```
net.SetNumberOfSlices(5);
```

The network is complete. The program proceeds to the inference, first without any evidence in the network, then with two observations of the *Umbrella*, set in the time steps  $t=1$  and  $t=3$ :

```
net.GetNode(umb)->Value()->SetTemporalEvidence(1, 0);
net.GetNode(umb)->Value()->SetTemporalEvidence(3, 1);
```

In dynamic Bayesian networks, the plate nodes have their beliefs calculated for each time slice. The number of elements in the node value matrix for these nodes is a product of outcome count and slice count. The helper function `UpdateAndShowTemporalResults` iterates over this matrix using two nested loops in order to print the results:

```
const DSL_Dmatrix *mtx = node->Value()->GetMatrix();
for (int sliceIdx = 0; sliceIdx < sliceCount; sliceIdx++)
{
    printf("\ttt=%d:", sliceIdx);
    for (int i = 0; i < outcomeCount; i++)
    {
        printf(" %f", (*mtx)[sliceIdx * outcomeCount + i]);
    }
    printf("\n");
}
```

Since the probabilities for the same slice are adjacent, the inner loop uses the product of the outcome count and slice index from the outer loop as a base index.

### 6.7.1 tutorial6.cpp

```
// tutorial6.cpp
// Tutorial6 creates a dynamic Bayesian network (DBN),
// performs the inference, then saves the model to disk.

#include "smile.h"
#include <cstdio>

static int CreateCptNode(
    DSL_network &net, const char *id,
    const char *name, const char *outcomes[],
    int xPos, int yPos);

static void UpdateAndShowTemporalResults(DSL_network &net);

int Tutorial6()
{
    printf("Starting Tutorial6...\n");

    DSL_errorH().RedirectToFile(stdout);

    DSL_network net;

    const char *LOCATION_OUTCOMES[] = { "Pittsburgh", "Sahara", NULL };
    int loc = CreateCptNode(net, "Location", "Location",
        LOCATION_OUTCOMES, 160, 360);

    const char *BOOL_OUTCOMES[] = { "true", "false", NULL };
    int rain = CreateCptNode(net, "Rain", "Rain",
        BOOL_OUTCOMES, 380, 240);

    int umb = CreateCptNode(net, "Umbrella", "Umbrella",
        BOOL_OUTCOMES, 300, 100);

    net.SetTemporalType(rain, dsl_temporalType::dsl_plateNode);
    net.SetTemporalType(umb, dsl_temporalType::dsl_plateNode);

    net.AddArc(loc, rain);
    net.AddTemporalArc(rain, rain, 1);
    net.AddArc(rain, umb);

    DSL_nodeDefinition *rainDef = net.GetNode(rain)->Definition();
    DSL_doubleArray rainProbs(rainDef->GetMatrix()->GetSize());
    rainProbs[0] = 0.7; // P(Rain=true | Location=Pittsburgh)
    rainProbs[1] = 0.3; // P(Rain=false | Location=Pittsburgh)
    rainProbs[2] = 0.01; // P(Rain=true | Location=Sahara)
    rainProbs[3] = 0.99; // P(Rain=false | Location=Sahara)
    int res = rainDef->SetDefinition(rainProbs);
    if (DSL_OKAY != res)
    {
        return res;
    }

    rainProbs.SetSize(rainDef->GetTemporalDefinition(1)->GetSize());
}
```

```

rainProbs[0] = 0.7; // P(Rain=true | Location=Pittsburgh, Rain[t-1]=true)
rainProbs[1] = 0.3; // P(Rain=false | Location=Pittsburgh, Rain[t-1]=true)
rainProbs[2] = 0.3; // P(Rain=true | Location=Pittsburgh, Rain[t-1]=false)
rainProbs[3] = 0.7; // P(Rain=false | Location=Pittsburgh, Rain[t-1]=false)
rainProbs[4] = 0.001; // P(Rain=true | Location=Sahara, Rain[t-1]=true)
rainProbs[5] = 0.999; // P(Rain=false | Location=Sahara, Rain[t-1]=true)
rainProbs[6] = 0.01; // P(Rain=true | Location=Sahara, Rain[t-1]=false)
rainProbs[7] = 0.99; // P(Rain=false | Location=Sahara, Rain[t-1]=false)
res = rainDef->SetTemporalDefinition(1, rainProbs);
if (DSL_OKAY != res)
{
    return res;
}

DSL_nodeDefinition *umbDef = net.GetNode(umb)->Definition();
DSL_doubleArray umbProbs(umbDef->GetMatrix()->GetSize());
umbProbs[0] = 0.9; // P(Umbrella=true | Rain=true)
umbProbs[1] = 0.1; // P(Umbrella=false | Rain=true)
umbProbs[2] = 0.2; // P(Umbrella=true | Rain=false)
umbProbs[3] = 0.8; // P(Umbrella=false | Rain=false)
res = umbDef->SetDefinition(umbProbs);
if (DSL_OKAY != res)
{
    return res;
}

net.SetNumberOfSlices(5);

printf("Performing update without evidence.\n");
UpdateAndShowTemporalResults(net);

printf("Setting Umbrella[t=1] to true and Umbrella[t=3] to false.\n");
net.GetNode(umb)->Value()->SetTemporalEvidence(1, 0);
net.GetNode(umb)->Value()->SetTemporalEvidence(3, 1);
UpdateAndShowTemporalResults(net);

res = net.WriteFile("tutorial6.xdsl");
if (DSL_OKAY != res)
{
    return res;
}

printf("Tutorial6 complete: Network written to tutorial6.xdsl\n");
return DSL_OKAY;
}

static void UpdateAndShowTemporalResults(DSL_network &net)
{
    net.UpdateBeliefs();
    int sliceCount = net.GetNumberOfSlices();
    for (int h = net.GetFirstNode(); h >= 0; h = net.GetNextNode(h))
    {
        if (net.GetTemporalType(h) == dsl_temporalType::dsl_plateNode)
        {
            DSL_node *node = net.GetNode(h);
            int outcomeCount = node->Definition()->GetNumberOfOutcomes();

```

```

        printf("Temporal beliefs for %s:\n", node->GetId());
        const DSL_Dmatrix *mtx = node->Value()->GetMatrix();
        for (int sliceIdx = 0; sliceIdx < sliceCount; sliceIdx++)
        {
            printf("\ttt=%d:", sliceIdx);
            for (int i = 0; i < outcomeCount; i++)
            {
                printf(" %f", (*mtx)[sliceIdx * outcomeCount + i]);
            }
            printf("\n");
        }
    }
    printf("\n");
}

static int CreateCptNode(
    DSL_network &net, const char *id, const char *name,
    const char *outcomes[], int xPos, int yPos)
{
    int handle = net.AddNode(DSL_CPT, id);
    DSL_node *node = net.GetNode(handle);

    DSL_idArray ida;
    for (const char **p = outcomes; *p != NULL; p++)
    {
        ida.Add(*p);
    }
    DSL_nodeDefinition *def = node->Definition();
    def->SetNumberOfOutcomes(ida);

    node->Info().Header().SetName(name);

    DSL_rectangle &position = node->Info().Screen().position;
    position.center_X = xPos;
    position.center_Y = yPos;
    position.width = 85;
    position.height = 55;

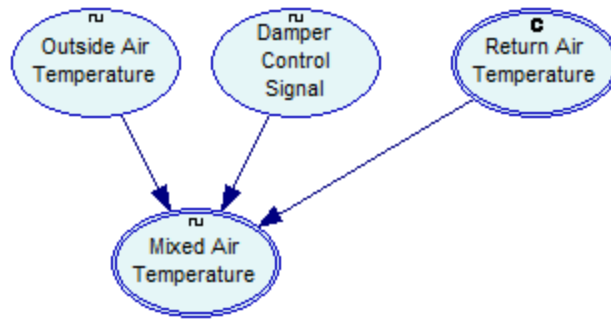
    return handle;
}

```

## 6.8 Tutorial 7: Continuous model

---

The continuous Bayesian network used in this tutorial focuses on a fragment of a forced air heating and cooling system. In order to improve the system's efficiency, return air is mixed with the air that is drawn from outside. Temperature of the outside air depends on the weather and is specified by means of a Normal distribution. Return air temperature is constant and depends on the thermostat setting. Damper control signal determines the composition of the mixture.



Temperature of the mixture is calculated according to the following equation:  $tma = toa * u_d + (tra - tra * u_d)$ , where  $tma$  is mixed air temperature,  $toa$  is outside air temperature,  $u_d$  is the damper signal, and  $tra$  is return air temperature.

The nodes in this model are created by the helper function `CreateEquationNode`. It is a modified version of the `CreateCptNode` used in the previous tutorials. The bold text marks the difference between two functions.

```

static int CreateEquationNode(
    DSL_network &net, const char *id, const char *name,
    const char *equation, double loBound, double hiBound,
    int xPos, int yPos)
{
    int handle = net.AddNode(DSL_EQUATION, id);
    DSL_node *node = net.GetNode(handle);

    DSL_equation *eq = static_cast<DSL_equation *>(node->Definition());
    eq->SetEquation(equation);
    eq->SetBounds(loBound, hiBound);

    node->Info().Header().SetName(name);

    DSL_rectangle &position = node->Info().Screen().position;
    position.center_X = xPos;
    position.center_Y = yPos;
    position.width = 85;
    position.height = 55;

    return handle;
}

```

Instead of discrete outcomes, we specify the node equation and the bounds for the node value. Another helper function, `SetUniformIntervals`, is used to define the discretization intervals. They will be used by the inference algorithm when the evidence is set for *Mixed Air Temperature* node (which has parents). The uniform intervals are chosen for simplicity here; in general case the choice of interval edges should be done based on the actual expected distribution of the node value (for example, in case of the Normal distribution, we might create narrow discretization intervals close to the mean.)

Note that while the model has three arcs, there are no calls to `DSL_network::AddArc` in this tutorial. The arcs are created implicitly by `DSL_equation::SetEquation` method (called by `CreateEquationNode` function).

The network is complete now and we can proceed to inference. Three inference calls are made, one without evidence and two with continuous evidence specified by calling `DSL_nodeValue::SetEvidence(double)`. Setting the *Outside Air Temperature* to 28.5 degrees (toa is the name of int variable holding the handle of the *Outside Air Temperature* node):

```
net.GetNode(toa)->Value()->SetEvidence(28.5);
```

This overload of the `SetEvidence` method is easy to confuse with the one used in previous tutorials, which accepts an integer as a parameter. If the temperature to set had no fractional part, we would need to ensure the literal is of type double by appending ".0":

```
net.GetNode(tma)->Value()->SetEvidence(21.0);
```

The program uses `UpdateAndShowStats` helper function for inference. The helper calls `DSL_network::UpdateBeliefs` and iterates over the nodes in the network and calls another helper, `ShowStats`, for each node. `ShowStats` first checks if the node has evidence set. If it does, the evidence value is printed, and the function returns:

```
DSL_valEqEvaluation *eqVal =
    static_cast<DSL_valEqEvaluation *>(net.GetNode(nodeHandle)->Value());

if (eqVal->IsEvidence())
{
    double v;
    eqVal->GetEvidence(v);
    printf("%s has evidence set (%g)\n", nodeId, v);
    return;
}
```

If node has no evidence, we need to check if its value comes from the sampling or discretized inference. If sampling was used, the `std::vector` returned from `DSL_valEqEvaluation::GetDiscBeliefs` is empty:

```
const std::vector<double> &discBeliefs = eqVal->GetDiscBeliefs();
if (discBeliefs.empty())
{
    double mean, stddev, vmin, vmax;
    eqVal->GetStats(mean, stddev, vmin, vmax);
    printf("%s: mean=%g stddev=%g min=%g max=%g\n",
        nodeId, mean, stddev, vmin, vmax);
}
```

In such case, the output for the node contains simple statistics from sampling retrieved by `DSL_valEqEvaluation::GetStats`. To avoid excessive output, we omit the actual sample values, which could be retrieved by `DSL_valEqEvaluation::GetSample[s]`.

If the network had evidence set for the *Mixed Air Temperature* node (which has parents), the inference algorithm would fall back to discretization, and the discretized belief vector would be non-empty. The else part of the if statement looks like this:

```
DSL_equation *eqDef =
    static_cast<DSL_equation *>(net.GetNode(nodeHandle)->Definition());
const DSL_equation::IntervalVector &iv = eqDef->GetDiscIntervals();
printf("%s is discretized.\n", nodeId);
double loBound, hiBound;
eqDef->GetBounds(loBound, hiBound);
double lo = loBound;
for (unsigned i = 0; i < discBeliefs.size(); i++)
{
    double hi = iv[i].second;
    printf("\tP(%s in %g..%g)=%g\n", nodeId, lo, hi, discBeliefs[i]);
    lo = hi;
}
```

Note how we need to read the discretization intervals specified in equation node definition to display the complete information about the discretized beliefs (both probability from node value and discretization interval edges from the node definition).

All tutorials redirect SMILE's diagnostic stream to the console, and during the execution of this tutorial the following message will appear on the screen when `UpdateAndShowStats` is called after setting *Mixed Air Temperature* to 21 degrees.

```
-19: Equation node u_d was discretized.
-4: Discretization problem in node toa: Underflow samples: 807, min=-39.8255 loBound=-10
Overflow samples: 275, max=72.2534 hiBound=40 Total valid samples: 8918 of 10000
-4: Discretization problem in node tma: Underflow samples: 15145, min=-9.65187 loBound=10
Overflow samples: 8203, max=39.9188 hiBound=30 Total valid samples: 76652 of 100000
```

The first line with status code -19 (`DSL_WRONG_NUM_STATES`) is a warning about missing discretization intervals for the *Damper Control Signal* node. In such case, SMILE uses two intervals dividing the node's domain into two equal half. Two intervals are adequate in this case, as the equation for this node uses Bernoulli distribution. Status code -4 (`DSL_INVALID_VALUE`) is a warning that some of the discretization samples fall outside of the node's domain (defined by lower and upper bounds set earlier by `DSL_equation::SetBounds`). Detailed information about underflow and overflow can help during the model building, however, with the long-tailed distributions some of the generated samples will inevitably fall out of bounds.

At the end of the tutorial, the model is saved to disk. [Tutorial 8](#)<sup>[90]</sup> will expand it into a hybrid network by adding CPT nodes.

### 6.8.1 tutorial7.cpp

```
// tutorial7.cpp
// Tutorial7 creates a network with three equation-based nodes
// performs the inference, then saves the model to disk.

#include "smile.h"
#include <cstdio>
```

```

static int CreateEquationNode(
    DSL_network &net, const char *id, const char *name,
    const char *equation, double loBound, double hiBound,
    int xPos, int yPos);

static void UpdateAndShowStats(DSL_network &net);

static void SetUniformIntervals(DSL_network &net, int nodeHandle, int count);

int Tutorial7()
{
    printf("Starting Tutorial7...\n");

    DSL_errorH().RedirectToFile(stdout);

    DSL_network net;
    net.EnableRejectOutlierSamples(true);
    int tra = CreateEquationNode(net,
        "tra", "Return Air Temperature",
        "tra=24", 23.9, 24.1,
        280, 100);
    int u_d = CreateEquationNode(net,
        "u_d", "Damper Control Signal",
        "u_d = Bernoulli(0.539)*0.8 + 0.2", 0, 1,
        160, 100);
    int toa = CreateEquationNode(net,
        "toa", "Outside Air Temperature",
        "toa=Normal(11,15)", -10, 40,
        60, 100);
    int tma = CreateEquationNode(net,
        "tma", "Mixed Air Temperature",
        "tma=toa*u_d+(tra-tra*u_d)", 10, 30,
        110, 200);

    SetUniformIntervals(net, toa, 5);
    SetUniformIntervals(net, tma, 4);

    printf("Results with no evidence:\n");
    UpdateAndShowStats(net);

    net.GetNode(toa)->Value()->SetEvidence(28.5);
    printf("Results with outside air temperature set to 28.5:\n");
    UpdateAndShowStats(net);

    net.GetNode(toa)->Value()->ClearEvidence();
    printf("Results with mixed air temperature set to 21:\n");
    net.GetNode(tma)->Value()->SetEvidence(21.0); // ensure it's a double value
    UpdateAndShowStats(net);

    int res = net.WriteFile("tutorial7.xdsl");
    if (DSL_OKAY != res)
    {
        return res;
    }

    printf("Tutorial7 complete: Network written to tutorial7.xdsl\n");
}

```



```

    return DSL_OKAY;
}

static int CreateEquationNode(
    DSL_network &net, const char *id, const char *name,
    const char *equation, double loBound, double hiBound,
    int xPos, int yPos)
{
    int handle = net.AddNode(DSL_EQUATION, id);
    DSL_node *node = net.GetNode(handle);

    DSL_equation *eq = static_cast<DSL_equation *>(node->Definition());
    eq->SetEquation(equation);
    eq->SetBounds(loBound, hiBound);

    node->Info().Header().SetName(name);

    DSL_rectangle &position = node->Info().Screen().position;
    position.center_X = xPos;
    position.center_Y = yPos;
    position.width = 85;
    position.height = 55;

    return handle;
}

static void ShowStats(DSL_network &net, int nodeHandle)
{
    const char *nodeId = net.GetNode(nodeHandle)->GetId();

    DSL_valEqEvaluation *eqVal =
        static_cast<DSL_valEqEvaluation *>(net.GetNode(nodeHandle)->Value());

    if (eqVal->IsEvidence())
    {
        double v;
        eqVal->GetEvidence(v);
        printf("%s has evidence set (%g)\n", nodeId, v);
        return;
    }

    const std::vector<double> &discBeliefs = eqVal->GetDiscBeliefs();
    if (discBeliefs.empty())
    {
        double mean, stddev, vmin, vmax;
        eqVal->GetStats(mean, stddev, vmin, vmax);
        printf("%s: mean=%g stddev=%g min=%g max=%g\n",
            nodeId, mean, stddev, vmin, vmax);
    }
    else
    {
        DSL_equation *eqDef =
            static_cast<DSL_equation *>(net.GetNode(nodeHandle)->Definition());
        const DSL_equation::IntervalVector &iv = eqDef->GetDiscIntervals();
        printf("%s is discretized.\n", nodeId);
    }
}

```

```

        double loBound, hiBound;
        eqDef->GetBounds(loBound, hiBound);
        double lo = loBound;
        for (unsigned i = 0; i < discBeliefs.size(); i++)
        {
            double hi = iv[i].second;
            printf("\tP(%s in %g..%g)=%g\n", nodeId, lo, hi, discBeliefs[i]);
            lo = hi;
        }
    }
}

static void UpdateAndShowStats(DSL_network &net)
{
    net.UpdateBeliefs();
    for (int h = net.GetFirstNode(); h >= 0; h = net.GetNextNode(h))
    {
        ShowStats(net, h);
    }
    printf("\n");
}

static void SetUniformIntervals(DSL_network &net, int nodeHandle, int count)
{
    DSL_equation *eq =
        static_cast<DSL_equation *>(net.GetNode(nodeHandle)->Definition());

    double lo, hi;
    eq->GetBounds(lo, hi);

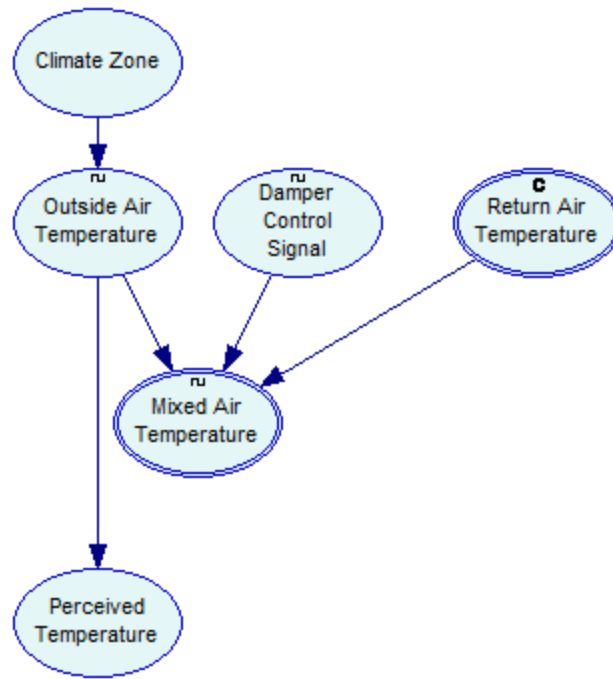
    DSL_equation::IntervalVector iv(count);
    for (int i = 0; i < count; i++)
    {
        iv[i].second = lo + (i + 1) * (hi - lo) / count;
    }

    eq->SetDiscIntervals(iv);
}

```

## 6.9 Tutorial 8: Hybrid model

We extend the model described in [Tutorial 7](#)<sup>84</sup> by adding two discrete nodes: *Climate Zone* and *Perceived Temperature*. *Climate Zone* refines the probability distribution of the outside air temperature and *Perceived Temperature* is an additional input originating from a subjective perception of the temperature, useful in case of a failure in the outside temperature sensor.



After loading the network created in the previous tutorial, the program adds two discrete nodes using the `CreateCptNode` helper function first seen in [Tutorial 1](#)<sup>[59]</sup>. The arc from *Climate Zone* (node identifier is *zone*) to *Outside Air Temperature* (node identifier is *toa*) is created by changing the equation of the latter:

```

DSL_equation *eq =
    static_cast<DSL_equation *>(net.GetNode(toa)->Definition());
eq->SetEquation("toa=If(zone=\"Desert\",Normal(22,5),Normal(11,10))");

```

In the C++ code, we need to escape the double quote characters in order to use the text literal representing the *Desert* outcome of the parent. The new equation switches between two normal distribution based on the outcome of the parent. The equation could be also written in the following ways, all being functionally identical (assuming that *Climate Zone* has two outcomes, *Temperate* and *Desert*):

```

eq->SetEquation("toa=zone=\"Desert\" ? Normal(22,5) : Normal(11,10)");
eq->SetEquation("toa=Switch(zone, \"Desert\",Normal(22,5),\"Temperate\",Normal(11,10))");
eq->SetEquation("toa=Choose(zone,Normal(11,10),Normal(22,5))");

```

To create the arc from *Outside Air Temperature* to *Perceived Temperature* the program calls `DSL_network::AddArc`. The model loaded from disk had 5 discretization intervals already defined for *Outside Air Temperature*. With 5 possible discretized outcomes of its single parent and its own 3 outcomes, the CPT for *Perceived Temperature* has  $3 \times 5 = 15$  entries. It is initialized by filling `DSL_doubleArray` object with numbers and passing it to `DSL_nodeDefinition::SetDefinition`, exactly the same approach that we used for discrete nodes in the previous tutorials. Note that for simplicity we do not change the default uniform distribution for the binary *Climate Zone* node.

The program performs inference for each of the *Climate Zone* outcomes set as evidence. The output displayed for the *Outside Air Temperature* node (*toa*) shows changing mean and standard deviation.

Finally, the network is saved to disk. This concludes the tutorial.

### 6.9.1 tutorial8.cpp

```
// tutorial8.cpp
// Tutorial8 loads continuous model from the XDSL file written by Tutorial7,
// then adds discrete nodes to create a hybrid model. Inference is performed
// and model is saved to disk.

#include "smile.h"
#include <cstdio>

static int CreateCptNode(
    DSL_network &net, const char *id,
    const char *name, const char *outcomes[],
    int xPos, int yPos);

static int SetEvidenceById(
    DSL_network &net, const char *nodeId, const char *outcomeId);

static void UpdateAndShowStats(DSL_network &net);

int Tutorial8()
{
    printf("Starting Tutorial8...\n");

    DSL_errorH().RedirectToFile(stdout);

    DSL_network net;
    int res = net.ReadFile("tutorial7.xdsl");
    if (DSL_OKAY != res)
    {
        printf(
            "Network load failed, did you run Tutorial7 before Tutorial8?\n");
        return res;
    }

    const char *ZONE_OUTCOMES[] = { "Temperate", "Desert", NULL };
    CreateCptNode(net,
        "zone", "Climate Zone", ZONE_OUTCOMES,
        60, 20);

    int toa = net.FindNode("toa");
    if (toa < 0)
    {
        printf("Outside air temperature node not found.\n");
        return toa;
    }
    DSL_equation *eq =
        static_cast<DSL_equation *>(net.GetNode(toa)->Definition());
```

```

eq->SetEquation("toa=If(zone=\"Desert\",Normal(22,5),Normal(11,10))");

const char *PERCEIVED_OUTCOMES[] = { "Hot", "Warm", "Cold", NULL };
int perceived = CreateCptNode(net,
    "perceived", "Perceived Temperature", PERCEIVED_OUTCOMES,
    60, 300);
net.AddArc(toa, perceived);

DSL_nodeDefinition *percDef = net.GetNode(perceived)->Definition();
DSL_doubleArray percProbs(percDef->GetMatrix()->GetSize());
percProbs[0] = 0;    // P(perceived=Hot | toa in -10..0)
percProbs[1] = 0.02; // P(perceived=Warm|toa in -10..0)
percProbs[2] = 0.98; // P(perceived=Cold|toa in -10..0)
percProbs[3] = 0.05; // P(perceived=Hot |toa in 0..10)
percProbs[4] = 0.15; // P(perceived=Warm|toa in 0..10)
percProbs[5] = 0.80; // P(perceived=Cold|toa in 0..10)
percProbs[6] = 0.10; // P(perceived=Hot |toa in 10..20)
percProbs[7] = 0.80; // P(perceived=Warm|toa in 10..20)
percProbs[8] = 0.10; // P(perceived=Cold|toa in 10..20)
percProbs[9] = 0.80; // P(perceived=Hot |toa in 20..30)
percProbs[10] = 0.15; // P(perceived=Warm|toa in 20..30)
percProbs[11] = 0.05; // P(perceived=Cold|toa in 20..30)
percProbs[12] = 0.98; // P(perceived=Hot |toa in 30..40)
percProbs[13] = 0.02; // P(perceived=Warm|toa in 30..40)
percProbs[14] = 0;    // P(perceived=Cold|toa in 30..40)
percDef->SetDefinition(percProbs);

SetEvidenceById(net, "zone", "Temperate");
printf("Results in temperate zone:\n");
UpdateAndShowStats(net);

SetEvidenceById(net, "zone", "Desert");
printf("Results in desert zone:\n");
UpdateAndShowStats(net);

res = net.WriteFile("tutorial8.xdsl");
if (DSL_OKAY != res)
{
    return res;
}

printf("Tutorial8 complete: Network written to tutorial8.xdsl\n");
return DSL_OKAY;
}

static int SetEvidenceById(
    DSL_network &net, const char *nodeId, const char *outcomeId)
{
    int handle = net.FindNode(nodeId);
    if (handle < 0)
    {
        return handle;
    }

    DSL_node *node = net.GetNode(handle);

```

```

    if (NULL == outcomeId)
    {
        return node->Value()->ClearEvidence();
    }
    else
    {
        int idx =
            node->Definition()->GetOutcomesNames()->FindPosition(outcomeId);
        if (idx < 0)
        {
            return idx;
        }

        return node->Value()->SetEvidence(idx);
    }
}

static void ShowStats(DSL_network &net, int nodeHandle)
{
    const char *nodeId = net.GetNode(nodeHandle)->GetId();

    DSL_valEqEvaluation *eqVal =
        static_cast<DSL_valEqEvaluation *>(net.GetNode(nodeHandle)->Value());

    if (eqVal->IsEvidence())
    {
        double v;
        eqVal->GetEvidence(v);
        printf("%s has evidence set (%g)\n", nodeId, v);
        return;
    }

    const std::vector<double> &discBeliefs = eqVal->GetDiscBeliefs();
    if (discBeliefs.empty())
    {
        double mean, stddev, vmin, vmax;
        eqVal->GetStats(mean, stddev, vmin, vmax);
        printf("%s: mean=%g stddev=%g min=%g max=%g\n",
            nodeId, mean, stddev, vmin, vmax);
    }
    else
    {
        DSL_equation *eqDef =
            static_cast<DSL_equation *>(net.GetNode(nodeHandle)->Definition());
        const DSL_equation::IntervalVector &iv = eqDef->GetDiscIntervals();
        printf("%s is discretized.\n", nodeId);
        double loBound, hiBound;
        eqDef->GetBounds(loBound, hiBound);
        double lo = loBound;
        for (unsigned i = 0; i < discBeliefs.size(); i++)
        {
            double hi = iv[i].second;
            printf("\tP(%s in %g..%g)=%g\n", nodeId, lo, hi, discBeliefs[i]);
            lo = hi;
        }
    }
}

```

```

}

static void UpdateAndShowStats(DSL_network &net)
{
    net.UpdateBeliefs();
    for (int h = net.GetFirstNode(); h >= 0; h = net.GetNextNode(h))
    {
        if (net.GetNode(h)->Definition()->GetType() == DSL_EQUATION)
        {
            ShowStats(net, h);
        }
    }
}

```

```

static int CreateCptNode(
    DSL_network &net, const char *id, const char *name,
    const char *outcomes[], int xPos, int yPos)
{
    int handle = net.AddNode(DSL_CPT, id);
    DSL_node *node = net.GetNode(handle);

    DSL_idArray ida;
    for (const char **p = outcomes; *p != NULL; p++)
    {
        ida.Add(*p);
    }
    DSL_nodeDefinition *def = node->Definition();
    def->SetNumberOfOutcomes(ida);

    node->Info().Header().SetName(name);

    DSL_rectangle &position = node->Info().Screen().position;
    position.center_X = xPos;
    position.center_Y = yPos;
    position.width = 85;
    position.height = 55;

    return handle;
}

```

This page is intentionally left blank.



# Reference Manual

## 7 Reference Manual

The information provided in this section covers the most important parts of SMILE's public API. For additional information, please refer to the header files included in your SMILE distribution - each class reference in this section begin with the header filename. The public members of SMILE classes are generally placed first within the class declarations.

Note that you don't need to include the headers one-by-one in your program; the main header `smile.h` does that.

### 7.1 DSL\_network

Header file: `network.h`

---

```
DSL_network();  
DSL_network(const DSL_network &src);  
~DSL_network();
```

Default constructor, copy constructor and destructor are implemented.

---

```
int ReadFile(const char *filename, int fileType=0, void *reserved=NULL);
```

Reads the network contents from the file specified by `filename`. If `fileType` is set to zero, the type of file is determined based on file's extension retrieved from the `filename`. The supported file type identifiers are:

Identifier	Extension	Description
DSL_XDSL_FORMAT	.xdsl	Native SMILE format
DSL_DSL_FORMAT	.dsl	Old, deprecated SMILE format
DSL_HUGIN_FORMAT	.net	Hugin
DSL_NETICA_FORMAT	.dne	Netica
DSL_INTERCHANGE_FORMAT	.dsc	Microsoft BN

Identifier	Extension	Description
DSL_ERGO_FORMAT	.erg	Ergo
DSL_KI_FORMAT	.dxp	DXpress

Returns DSL\_OKAY on success or negative error code otherwise. If the reason for failure is syntax error in the file, or the file is not found, the error message is logged to DSL\_errorH().

---

```
int WriteFile(const char *filename, int fileType=0, void *reserved=NULL);
```

Writes the network contents to the file system.

---

```
int ReadString(const char *xdslString, void *reserved=NULL);
```

Read the network content from xdslString, only XDSL format is supported.

---

```
int WriteString(std::string &xdslOut, void *reserved=NULL);
```

Writes the network content to xdslOut as XDSL.

---

```
int UpdateBeliefs();
```

Executes inference algorithm to update node values. Returns DSL\_OKAY on success or negative integer code on error.

---

```
void SetDefaultBNAlgorithm(int algorithm);
```

Sets the algorithm used for inference in Bayesian networks. Available algorithm identifiers are:

Identifier	Exact?	Description
DSL_ALG_BN_LAURITZEN	Yes	Clustering (default algorithm)
DSL_ALG_BN_RELEVANCEDECOMP	Yes	Linear Relevance Decomposition
DSL_ALG_BN_RELEVANCEDECOMP2	Yes	Recursive Relevance Decomposition

Identifier	Exact?	Description
DSL_ALG_BN_EPISSAMPLING	No	EPIS Sampling
DSL_ALG_BN_LBP	No	Loopy Belief Propagation
DSL_ALG_BN_AISSAMPLING	No	AIS Sampling
DSL_ALG_BN_LSAMPLING	No	Likelihood Sampling
DSL_ALG_BN_HENRION	No	Logic Sampling

---

```
void SetDefaultIDAlgorithm(int algorithm);
```

Sets the algorithm used for inference in influence diagrams. Available algorithm identifiers are:

Identifier	Description
DSL_ALG_ID_COOPERSOLVING	Policy evaluation
DSL_ALG_ID_SHACHTER	Best policy search

---

```
int InvalidateAllBeliefs();
```

Invalidates values in all of network's nodes. Returns DSL\_OKAY on success or negative integer code on error.

---

```
bool CalcProbEvidence(double &pe, bool forceChainRule = false);
```

Calculates the probability of evidence currently set in the network and stores the result in the pe argument. By default this method runs modified jointree algorithm to efficiently obtain the P(e). Set forceChainRule to true to enforce the slower chain rule algorithm. Returns true on success, false on error.

---

```
int GetNumberOfSamples() const;  
int SetNumberOfSamples(int numSamples);
```

Gets/sets the number of samples generated by sampling inference algorithms.

---

```
bool IsNoisyDecompEnabled() const;  
void EnableNoisyDecomp(bool enable);
```

Gets/sets the flag enabling noisyMAX decomposition.

---

```
int GetNoisyDecompLimit() const;  
void SetNoisyDecompLimit(int limit);
```

Gets/sets the limit on the noisyMAX parent nodes after the noisyMAX decomposition runs. The number of parent applies to temporary data structures managed by inference algorithm and does not modify the DSL\_network nor its nodes.

---

```
bool IsRejectOutlierSamplesEnabled() const  
int EnableRejectOutlierSamples(bool enable);
```

Gets/sets the flag controlling the sample rejection during stochastic inference in continuous and hybrid models.

---

```
int GetNumberOfDiscretizationSamples() const;  
void SetNumberOfDiscretizationSamples(int numSamples);
```

Gets/sets the number of samples generated for each CPT column during the discretization of continuous nodes.

---

```
bool IsZeroAvoidanceEnabled() const;  
void EnableZeroAvoidance(bool enable);
```

Gets/sets the flag controlling the discretization of non-deterministic continuous nodes. If set, the node discretization algorithm adds one artificial sample to each empty discretization bin. This only applies to nodes with equations explicitly including probability distribution functions.

---

```
int ClearAllEvidence();
```

Clears all evidence in the network.

---

```
DSL_node* GetNode(int handle);
```

Returns a pointer to the node identifier by handle, NULL otherwise.

---

```
int FindNode(const char *nodeId) const;
```

Returns a handle for the node with identifier equal to `nodeId`, negative integer with error code otherwise.

---

```
int GetFirstNode() const;
```

Returns a handle for the first node in the network, `DSL_OUT_OF_RANGE` if network has no nodes.

---

```
int GetNextNode(int handle) const;
```

Returns node handle following the specified handle. If the specified handle represented the last node in the network, returns `DSL_OUT_OF_RANGE`.

---

```
int GetLastNode() const;
```

Returns the last node handle by calling `GetFirstNode` followed by `GetNextNode` in the loop. If the network is empty, returns `DSL_OUT_OF_RANGE`.

---

```
int GetNumberOfNodes() const;
```

Returns the number of nodes in the network.

---

```
int AddNode(int nodeType, const char *nodeId);
```

Creates new node and returns its handle. If `nodeId` is `NULL`, the unique identifier will be created by the network. Supported node types are:

Identifier	Description
<code>DSL_CPT</code>	Conditional probability table
<code>DSL_TRUTHTABLE</code>	Discrete deterministic
<code>DSL_NOISY_MAX</code>	Noisy-MAX
<code>DSL_NOISY_ADDER</code>	Noisy-Adder
<code>DSL_LIST</code>	Decision
<code>DSL_TABLE</code>	Utility

Identifier	Description
DSL_MAU	Multi-attribute utility
DSL_EQUATION	Equation

If input parameters are invalid, the function returns negative status code.

---

```
int DeleteNode(int handle);
```

Deletes the node specified by handle. Returns DSL\_OKAY or negative status code.

---

```
int DeleteAllNodes();
```

Deletes all nodes in the network.

---

```
int AddArc(int parentHandle, int childHandle,  
          dsl_arcType layer = dsl_normalArc);
```

Adds an arc between nodes specified by parentHandle and childHandle. Returns DSL\_OKAY on success or negative status code on error. If adding an arc would result in cycle in the network, the status code is DSL\_CYCLE\_DETECTED.

---

```
int RemoveArc(int parentHandle, int childHandle,  
             dsl_arcType layer = dsl_normalArc);
```

Removes an arc. Returns DSL\_OKAY on success or negative status code on error.

---

```
const DSL_intArray& GetParents(int nodeHandle,  
                              dsl_arcType layer = dsl_normalArc) const;
```

Returns a reference to the DSL\_intArray containing handles of parent nodes. If nodeHandle is invalid the return value is undefined.

---

```
const DSL_intArray& GetChildren(int nodeHandle,  
                               dsl_arcType layer = dsl_normalArc) const;
```

Returns a reference to the DSL\_intArray containing handles of children nodes. If nodeHandle is invalid the return value is undefined.

---

```
int ReverseArc(int parentHandle, int childHandle);
```

Reverses an arc, preserving the joint probability distribution in the network. Returns DSL\_OKAY on success, negative status code otherwise.

---

```
int IsArcNecessary(int parentHandle, int childHandle,  
                  double epsilon, bool &necessary) const;
```

Checks if the arc between parentHandle and childHandle is necessary. The arc is considered to be necessary when child's conditional distributions for different parent states are different. The check is performed using Hellinger's distance with specified epsilon. The output of the check is returned in the necessary parameter. The return value of the function is DSL\_OKAY on success or negative status code otherwise.

---

```
int RemoveAllArcs();
```

Removes all arcs in the network.

---

```
int MarginalizeNode(int nodeHandle, DSL_progress *progress = NULL);
```

Removes node specified by nodeHandle, but preserves the joint probability distribution over the remaining nodes. Returns DSL\_OKAY on success or negative status code otherwise. The marginalization may take significant amount of time, therefore the caller may specify an instance of object derived from the DSL\_progress class to monitor the progress and optionally cancel the operation. If cancelled, the function returns DSL\_INTERRUPTED.

---

```
int MakeUniform();
```

Uniformizes all node definitions in the network.

---

```
int GetNumberOfTargets() const;
```

Returns the number of target nodes in the network.

---

```
int IsTarget(int nodeHandle);
```

For valid handle, returns nonzero if node was set as target, zero otherwise. If handle is invalid, it returns a negative status code.

---

```
int SetTarget(int nodeHandle, bool target);
```



Sets the target flag on node specified by `nodeHandle`. Returns `DSL_OKAY` if `nodeHandle` is valid and the flag has actually changed. Returns `DSL_INVALID_VALUE` if `nodeHandle` was valid, but the value of `target` is the same as the value of the current target flag of the node (i.e., when trying to set the target flag of a node that is already a target or when trying to clear the target flag of a node that is not a target).

---

**`int ClearAllTargets();`**

Resets the target flags on all nodes.

---

**`DSL_simpleCase* AddCase(const std::string & name);`**

Adds new case with specified name. Returns the pointer to the newly created case.

---

**`DSL_simpleCase* GetCase(int index) const;`**

Returns the pointer to the case specified by the zero-based case index. If index is negative or greater or equal to the number of cases, returns `NULL`.

---

**`DSL_simpleCase* GetCase(const std::string & name) const;`**

Returns the pointer to the case specified by name. If the case is not found, returns `NULL`.

---

**`int DeleteCase(int index);`**

Deletes the case specified by the index. Returns `DSL_OKAY` on success, negative status code otherwise.

---

**`void DeleteAllCases();`**

Removes all cases from the network.

---

**`int GetNumberOfCases() const;`**

Returns the number of cases defined for the network

---

**`void EnableSyncCases(bool sync);`**

Enables or disables the case synchronization. If enabled, structural changes in node definitions, like creation or removal of node outcomes, will be reflected in the data stored in cases. The case synchronization mode is enabled by default.

---

```
bool IsEnableSyncCases() const;
```

Returns true if case synchronization is enabled, false otherwise.

---

```
int UnrollNetwork(DSL_network &unrolled, std::vector<int> &unrollMap) const;
```

Creates unrolled network from a DBN. The unrollMap output parameter contains a mapping from unrolled network to the original DBN. Node with the handle *h* in the unrolled represents the original node with handle *unrollMap[h]*.

---

```
int GetMaxTemporalOrder() const;  
int GetMaxTemporalOrder(int nodeHandle) const;
```

Returns the maximum order of temporal arcs in the entire network or for specific node.

---

```
int GetTemporalOrders(int nodeHandle, DSL_intArray &orders) const;
```

Gets all temporal orders of

---

```
int GetNumberOfSlices() const;  
int SetNumberOfSlices(int slices);
```

Get/set the number of slices in the unrolled DBN.

---

```
int AddTemporalArc(int parent, int child, int order);
```

Adds temporal arc of a specified order between nodes with parent and child handles.

---

```
int RemoveTemporalArc(int parent, int child, int order);
```

Removes a temporal arc of a specified order between nodes with parent and child handles.

---

```
bool TemporalArcExists(int parent, int child, int order) const;
```

Returns true if a temporal arc of a specified order between nodes with parent and child handles.

---

```
int IsTemporalArcNecessary(int parent, int child,
    int order, double epsilon, bool &necessary) const;
```

Checks if the temporal arc with specified order between parent and child is necessary. The arc is considered to be necessary when child's conditional distributions for different parent states are different. The check is performed using Hellinger's distance with specified epsilon. The output of the check is returned in the necessary parameter. The return value of the function is DSL\_OKAY on success or negative status code otherwise.

---

```
dsl_temporalType GetTemporalType(int nodeHandle) const;
int SetTemporalType(int nodeHandle, dsl_temporalType type);
```

Get/set the temporal type of node. The temporal type is defined by the following enumeration:

```
enum dsl_temporalType { dsl_normalNode, dsl_anchorNode, dsl_terminalNode, dsl_plateNode };
```

---

```
int GetTemporalChildren(int parent,
    std::vector<std::pair<int, int> > &children) const;
```

Retrieves the information about the temporal children of the specified parent node. Each temporal arc originating in the parent has one corresponding element in the output vector. The element is the `std::pair<int,int>`, where first element is a child node handle, and second element is a temporal order. Returns DSL\_OKAY on success or negative status code otherwise.

---

```
int GetTemporalParents(int child, int order, DSL_intArray &parents) const;
```

Retrieves the handles of temporal parents of the specified temporal order. Note that parents with order  $x$  are in general only the subset of nodes which index temporal definition of order  $x$ . Use `GetUnrolledParents` to get all parents indexing the temporal definition. Returns DSL\_OKAY on success or negative status code otherwise.

---

```
int GetUnrolledParents(int child, int order,
    std::vector<std::pair<int, int> > &parents) const;
```

Retrieves the handles and orders of temporal parents indexing the temporal definition of specified order. Returns DSL\_OKAY on success or negative status code otherwise.

---

```
int GetUnrolledParents(int child,
    std::vector<std::pair<int, int> > &parents) const;
```

Retrieves the handles and orders of temporal parents. Returns DSL\_OKAY on success or negative status code otherwise.

---

## 7.2 DSL\_node

---

Header file: `node.h`

DSL\_node objects are always created and destroyed by their parent DSL\_network. Never use free or delete on a node pointer. To obtain a pointer to existing node, use DSL\_network::GetNode.

---

```
const char* GetId() const;
```

Returns node's identifier.

---

```
int SetId(const char *newId);
```

Sets node's identifier. Returns DSL\_OKAY on success or negative status code on failure.

---

```
DSL_nodeInfo &Info();
```

Returns a reference to DSL\_nodeInfo structure containing node attributes not directly related to calculations.

---

```
DSL_network* Network();
```

Returns a pointer to node's network.

---

```
int Handle();
```

Returns node's handle.

---

```
DSL_nodeDefinition* Definition();
```

Returns a pointer to node's definition. Never call delete on the returned pointer.

---

```
DSL_nodeValue* Value();
```

Returns a pointer to node's value. Never call delete on the returned pointer.

### 7.3 DSL\_nodeDefinition

---

Header file: `nodedef.h`

Objects of classes derived from `DSL_nodeDefinition` are created and destroyed by their parent `DSL_node`. Never use `free` or `delete` on a node definition pointer. Use `DSL_node::Definition` method to obtain a pointer to node's definition.

---

```
DSL_network* Network();
```

Returns a pointer to parent node's network.

---

```
int Handle() const;
```

Returns a handle of parent node.

---

```
virtual int GetType() const = 0;
```

Returns a numeric identifier of the definition type. See `DSL_network::AddNode` for the complete list of types.

---

```
virtual const char *GetTypeName() const = 0;
```

Returns a definition type name.

---

```
virtual int AddOutcome(const char *outcomeId);
```

Adds node's outcome with the specified identifier.

---

```
virtual int InsertOutcome(int outcomeIndex, const char *outcomeId);
```

Inserts node's outcome with the specified identifier at the specified index.

---

```
virtual int RemoveOutcome(int outcomeIndex);
```

Removes node's outcome at the specified index.

---

```
virtual int GetNumberOfOutcomes();
```

Returns the number of node's outcomes or a negative error code if node type does not have outcomes.

---

```
virtual int RenameOutcome(int outcomeIndex, char *newId);
```

Renames node's outcome at the specified index.

---

```
virtual int RenameOutcomes(DSL_stringArray &outcomeIds);
```

Changes all the outcome identifiers.

---

```
virtual DSL_idArray* GetOutcomesNames();
```

Returns the object representing outcome identifiers.

---

```
virtual int SetNumberOfOutcomes(int outcomeCount);
```

Sets the number of node's outcomes. If the outcome count increases, new outcomes will have automatically generated identifiers. Returns DSL\_OKAY when successful or a negative error code on failure. Note that this function cannot be called if node has children nodes.

---

```
virtual int SetNumberOfOutcomes(DSL_stringArray &outcomeIds);
```

Sets the number of node's outcomes and renames them by using specified identifiers. Returns DSL\_OKAY when successful or a negative error code on failure. Note that this function cannot be called if node has children nodes.

---

```
virtual int SetDefinition(DSL_doubleArray &def);
```

```
virtual int SetDefinition(DSL_Dmatrix &def);
```

Sets the numeric parameters of the node. Returns DSL\_OKAY on success or a negative failure code otherwise.

---

```
virtual int GetDefinition(DSL_doubleArray **def);
```

```
virtual int GetDefinition(DSL_Dmatrix **def);
```

Retrieves a pointer to an object representing node's numeric parameters. Returns DSL\_OKAY on success or a negative failure code otherwise.

---

```
DSL_Dmatrix* GetMatrix();
```

Returns a pointer to a `DSL_Dmatrix` object with node's numeric parameters, or `NULL` if node does not have numeric parameters.

---

```
int SetTemporalDefinition(int order, DSL_doubleArray& temporal);
```

For a plate node in a DBN, sets the temporal parameters of the node for the specified temporal order. Returns `DSL_OKAY` on success or a negative failure code otherwise.

---

```
const DSL_doubleArray* int GetTemporalDefinition(int order);
```

For a plate node in a DBN, returns a pointer to an array representing the temporal parameters of the node with the specified order, or `NULL` if node does not have temporal parameters of the specified order.

## 7.4 DSL\_noisyMAX

---

Header file: `defnoisymax.h`

When using Noisy-MAX node, call `DSL_node::Definition` method to obtain a pointer to node's definition, then cast it to a `DSL_noisyMAX` pointer. Never use `free` or `delete` on a Noisy-MAX node definition pointer.

---

```
DSL_Dmatrix &GetCiWeights();
```

Returns a reference to conditionally independent probability matrix.

---

```
const DSL_intArray &GetParentOutcomeStrengths(int parentPos) const;
```

Returns the outcome strengths for the specified parent.

---

```
int SetParentOutcomeStrengths(int parentPos, const DSL_intArray &newStrengths);
```

Sets the outcome strengths for the specified parent. Returns `DSL_OKAY` on success or a negative error code on failure.

---

```
const DSL_Dmatrix* GetTemporalCiWeights(int order) const;
```

For a plate node in a DBN, gets the conditionally independent probability matrix for a specified temporal order.

---

```
int SetTemporalCiWeights(int order, const DSL_Dmatrix& weights);
```

For a plate node in a DBN, sets the conditionally independent probability matrix for a specified temporal order.

---

```
int GetTemporalParentOutcomeStrengths(int order, std::vector<DSL_intArray>& strengths) con
```

For a plate node in a DBN, gets outcome strengths for all parents for a specified temporal order. Returns DSL\_OKAY on success or a negative error code on failure.

---

```
int SetTemporalParentOutcomeStrengths(int order, const std::vector<DSL_intArray>& strength
```

For a plate node in a DBN, sets outcome strengths for all parents for a specified temporal order. Returns DSL\_OKAY on success or a negative error code on failure.

## 7.5 DSL\_noisyAdder

---

Header file: **defnoisyadder.h**

When using Noisy-Adder node, call `DSL_node::Definition` method to obtain a pointer to node's definition, then cast it to a `DSL_noisyAdder` pointer. Never use `free` or `delete` on a Noisy-Adder node definition pointer.

---

```
DSL_Dmatrix &GetCiWeights();
```

Returns a reference to conditionally independent probability matrix.

---

```
int GetDistinguishedState();
```

Returns node's distinguished state.

---

```
int GetParentDistinguishedState(int parentPos);
```

Returns the distinguished state of the specified parent.



---

```
double GetParentWeight(int parentPos);
```

Returns the parent weight or, when parentPos is equal to the number of parents, the leak weight.

---

```
int SetDistinguishedState(int thisState);
```

Sets node's distinguished state.

---

```
int SetParentDistinguishedState(int parentPos, int newDState);
```

Sets the distinguished state of the specified parent.

---

```
int SetParentWeight(int parentPos, double value);
```

Setsthe parent weight or, when parentPos is equal to the number of parents, the leak weight.

---

```
const DSL_Dmatrix* GetTemporalCiWeights(int order) const;
```

For a plate node in a DBN, gets the conditionally independent probability matrix for specified temporal order.

---

```
int SetTemporalCiWeights(int order, const DSL_Dmatrix& weights);
```

For a plate node in a DBN, sets the conditionally independent probability matrix for specified temporal order.

---

```
int GetTemporalDistinguishedState(int order) const;
```

For a plate node in a DBN, gets the distinguished state for specified temporal order. Returns DSL\_OKAY on success or a negative error code on failure.

---

```
int SetTemporalDistinguishedState(int order, int state);
```

For a plate node in a DBN, sets the distinguished state for specified temporal order. Returns DSL\_OKAY on success or a negative error code on failure.

---

```
int GetTemporalParentInfo(int order,  
    DSL_doubleArray &weights, DSL_intArray &distStates) const;
```

---

For a plate node in a DBN, gets the Noisy-Adder parent information for a specified temporal order. Returns DSL\_OKAY on success or a negative error code on failure.

---

```
int SetTemporalParentInfo(int order,
    const DSL_doubleArray &weights, const DSL_intArray &distStates);
```

For a plate node in a DBN, sets the Noisy-Adder parent information for a specified temporal order. Returns DSL\_OKAY on success or a negative error code on failure.

## 7.6 DSL\_truthTable

---

Header file: **deftruthtable.h**

When using a deterministic node, call `DSL_node::Definition` method to obtain a pointer to node's definition, then cast it to a `DSL_truthTable` pointer. Never use `free` or `delete` on a deterministic node definition pointer.

---

```
int GetResultingStates(DSL_intArray &here) const;
int GetResultingStates(DSL_stringArray &here) const;
```

Fill the output array with indices or identifiers of resulting states.

---

```
int SetResultingStates(const int* states);
int SetResultingStates(const DSL_intArray &theStates);
int SetResultingStates(DSL_stringArray &theStates);
```

Set the resulting states using state indices or identifiers. Returns DSL\_OKAY on success or a negative error code on failure.

---

```
int GetTemporalResultingStates(int order, DSL_intArray &states) const;
```

For a plate node in a DBN, gets the resulting states for a specified temporal order. Returns DSL\_OKAY on success or a negative error code on failure.

---

```
int SetTemporalResultingStates(int order, const int* states);
```

For a plate node in a DBN, sets the resulting states for a specified temporal order. Returns DSL\_OKAY on success or a negative error code on failure.

## 7.7 DSL\_equation

---

Header file: **defequation.h**

DSL\_equation class is derived from DSL\_nodeDefinition. When using an equation node, call DSL\_node::Definition method to obtain a pointer to node's definition, then cast it to a DSL\_equation pointer. The virtual methods defined in DSL\_nodeDefinition related to discrete nodes (managing node outcomes and matrix-based parameters) return DSL\_WRONG\_NODE\_TYPE. Never use free or delete on a DSL\_equation pointer.

---

```
const DSL_generalEquation& GetEquation() const;
```

Returns a reference to the DSL\_generalEquation object, representing node equation in parsed form. Use DSL\_generalEquation::Write(std::string&) to obtain an equation in a textual form.

---

```
int SetEquation(const std::string &eq,  
               int *errPos = NULL, std::string *errMsg = NULL);
```

Sets the node equation based on the text passed in eq string. Optional parameters are useful for programs which parse user's input. errPos is the index of the character in the equation string where the parse error was encountered. errMsg is the human-readable error message.

---

```
bool ValidateEquation(const std::string &eq, std::vector<std::string> &vars,  
                     std::string &errMsg, int *errPos = NULL) const;
```

Validates the equation, returns true if the equation is valid (and using it in the SetEquation call would be successful). Checks for syntax errors and ensures that referenced variable and function names are valid. The identifiers of the variables in the equation are returned in the vars output parameter.

---

```
void GetBounds(double &low, double &high) const;
```

Gets the domain for the variable represented by the node.

---

```
void SetBounds(double low, double high);
```

Sets the domain for the variable represented by the node. Use plus or minus infinity to represent open bounds.

---

```
typedef std::vector<std::pair<std::string, double> > IntervalVector;
```

Convenience typedef. First element of the pair represents the interval id, is optional (can be empty) and used only for display purposes. The second element is the upper bound of the interval. The lower bound is the upper bound of the preceding interval, or the lower bound for the node for the first interval.

---

```
int SetDiscIntervals(const IntervalVector &intervals);  
int SetDiscIntervals(double lo, double hi, const IntervalVector &intervals);
```

Set the intervals. The second overload also sets the node bounds.

---

```
int ClearDiscIntervals();
```

Removes the intervals. Does not affect the node bounds.

---

```
const IntervalVector& GetDiscIntervals() const;
```

Returns the reference to the defined intervals (which may be an empty vector if not defined yet).

---

```
int GetDiscInterval(int intervalIndex, double &lo, double &hi) const;
```

Returns the bounds for the specified interval.

## 7.8 DSL\_nodeValue

---

Header file: **nodeval.h**

Objects of classes derived from **DSL\_nodeValue** are created and destroyed by their parent **DSL\_node**. Never use **free** or **delete** on a node value pointer. Use **DSL\_node::Value** method to obtain a pointer to node's value.

---

```
DSL_network *Network();
```

Returns a pointer to parent node's network.

---

**int Handle();**

Returns a handle of parent node.

---

**virtual int GetType() const = 0;**

Returns the numeric type identifier of the value object.

---

**DSL\_intArray& GetIndexingParents();**

Returns the reference to the DSL\_intArray object with handles of the parents indexing the value. This array is non-empty only for influence diagrams.

---

**int IsValueValid();**

Returns non-zero if node's value is valid.

---

**virtual int GetValue(DSL\_Dmatrix \*\*val);**

Retrieves the pointer to the DSL\_Dmatrix object with calculated node value. Returns DSL\_OKAY on success or a negative error code on failure.

---

**DSL\_Dmatrix\* GetMatrix();**

Returns the pointer to the DSL\_Dmatrix object with calculated node value. Call only if IsValueValid returns non-zero.

---

**const DSL\_intArray& GetIndexingParents() const;**

Returns the reference to the DSL\_intArray which contains handles of the indexing parents. Indexing parents are unobserved decision nodes that precede the current node or unobserved chance nodes that should have been observed. This function should be called only in influence diagrams.

---

**int IsEvidence() const;**

Returns non-zero if node has evidence.

---

**int IsRealEvidence() const;**

Returns non-zero if node has an evidence set, and the evidence is not a propagated evidence.

---

**int IsPropagatedEvidence() const;**

Returns non-zero if node has a propagated evidence set.

---

**int IsVirtualEvidence() const;**

Returns non-zero if node has a virtual evidence set.

---

**virtual int GetEvidence() const;**

Retrieves node's discrete evidence. Returns non-negative outcome index on success, or a negative error code on failure.

---

**virtual int GetEvidence(double &evidence) const;**

Retrieves node's continuous evidence. Valid only for DSL\_EQUATION nodes. Returns DSL\_OKAY on success, or a negative error code on failure.

---

**virtual int SetEvidence(double evidence);**

Sets the evidence as a continuous number. Valid only for DSL\_EQUATION nodes. Returns DSL\_OKAY on success, or a negative error code on failure.

---

**virtual int SetEvidence(int evidence);**

Sets the evidence as an integer index of node's outcome. Returns DSL\_OKAY on success, or a negative error code on failure.

---

**virtual int ClearEvidence();**

Removes all types of evidence from the node.

---

**virtual int ClearPropagatedEvidence();**

If node is set to propagated evidence, removes the evidence from the node.

---

**virtual int GetVirtualEvidence(std::vector<double> &evidence) const;**

Retrieves node's virtual evidence. Returns DSL\_OKAY on success, or a negative error code on failure.

---

**virtual int SetVirtualEvidence(const std::vector<double> &evidence);**

Sets node's virtual evidence. Returns DSL\_OKAY on success, or a negative error code on failure.

---

**bool HasTemporalEvidence() const;**

Returns true if node has any temporal evidence (regardless of the slice)

---

**bool IsTemporalEvidence(int slice) const;**

Returns true if node has a temporal evidence in the specified slice.

---

**int GetTemporalEvidence(int slice) const;**

Retrieves node's discrete temporal evidence for the specified slice. Returns non-negative outcome index on success, or a negative error code on failure.

---

**int GetTemporalEvidence(int slice, std::vector<double> &evidence) const;**

Retrieves node's virtual temporal evidence for the specified slice. Returns DSL\_OKAY on success, or a negative error code on failure.

---

**int SetTemporalEvidence(int slice, int evidence);**

Sets the temporal evidence for the specified slice as an integer index of node's outcome. Returns DSL\_OKAY on success, or a negative error code on failure.

---

**int SetTemporalEvidence(int slice, const std::vector<double> &evidence);**

Sets the temporal virtual evidence for the specified slice. Returns DSL\_OKAY on success, or a negative error code on failure.

---

**int ClearTemporalEvidence(int slice);**

Clears the temporal evidence for the specified slice.

## 7.9 DSL\_valEqEvaluation

---

Header file: `valequationevaluation.h`

`DSL_valEqEvaluation` class is derived from `DSL_nodeValue`. The virtual methods defined in `DSL_nodeValue` related to discrete nodes return `DSL_WRONG_NODE_TYPE`.

---

**`const std::vector<double>& GetDiscBeliefs() const;`**

Returns a reference to the discretized beliefs vector. If the vector is non-empty, the equation was evaluated over the temporary discrete network and contains no samples. Therefore, all sample-related methods should not be called and the only information about node probabilities is contained in the beliefs vector.

---

**`const std::vector<double>& GetSamples() const;`**

Returns a reference to the vector of samples generated for the node during stochastic inference. The size of the vector is no larger than a value passed to `DSL_network::SetNumberOfSamples`. If outlier rejection was enabled by `DSL_network::EnableRejectOutlierSamples`, some of the samples may be rejected if their value falls outside of the node bounds.

---

**`double GetSampleMean() const;`**

Gets the domain for the variable represented by the node.

---

**`double GetSampleStdDev() const;`**

Sets the domain for the variable represented by the node. Use plus or minus infinity to represent open bounds.

---

**`typedef std::vector<std::pair<std::string, double> > IntervalVector;`**

Convenience typedef. First element of the pair represents the interval id, is optional (can be empty) and used only for display purposes. The second element is the upper bound of the interval. The lower bound is the upper bound of the preceding interval, or the lower bound for the node for the first interval.

---

**`int SetDiscIntervals(const IntervalVector &intervals);`**  
**`int SetDiscIntervals(double lo, double hi, const IntervalVector &intervals);`**



Set the intervals. The second overload also sets the node bounds.

---

```
int ClearDiscIntervals();
```

Removes the intervals. Does not affect the node bounds.

---

```
const IntervalVector& GetDiscIntervals() const;
```

Returns the reference to the defined intervals (which may be an empty vector if not defined yet).

---

```
int GetDiscInterval(int intervalIndex, double &lo, double &hi) const;
```

Returns the bounds for the specified interval.

## 7.10 DSL\_Dmatrix

---

Header file: **dmatrix.h**

---

```
DSL_Dmatrix();
```

The default constructor, creates an empty matrix without any dimensions.

---

```
DSL_Dmatrix(const DSL_intArray &dimensions);
```

The constructor which creates the matrix with specified dimensions.

---

```
DSL_Dmatrix(const DSL_Dmatrix &mtx);
```

Copy constructor.

---

```
int operator=(const DSL_Dmatrix &mtx);
```

Assignment operator.

---

```
void Swap(DSL_Dmatrix &mtx);
```

Swaps the two matrices.

---

```
double &operator[](int index);  
double operator[](int index) const;  
double &Subscript(int index);
```

Access to the matrix elements with linear index.

---

```
double &operator[](const DSL_intArray &coords);  
double operator[](const DSL_intArray &coords) const;  
double &Subscript(DSL_intArray &coords);
```

Access to the matrix elements using multidimensional coordinates.

---

```
double &operator[] (const int * const * indirectCoords);
```

Access to the matrix elements using indirect multidimensional coordinates.

---

```
int IndexToCoordinates(int index, DSL_intArray &coords) const;
```

Converts the linear index to multidimensional coordinates. Returns DSL\_OKAY on success, or an error code on failure.

---

```
int CoordinatesToIndex(DSL_intArray &coords) const;  
int CoordinatesToIndex(const int * const * indirectCoords) const;
```

Converts the multidimensional coordinates to linear index. Returns a converted index, or a negative error code on failure.

---

```
int NextCoordinates(DSL_intArray &coords) const;
```

Modifies the specified multidimensional coordinates to be the equivalent of the next linear index. Returns DSL\_OKAY on success, or an error code on failure. Specifying the coordinates representing the last element of the matrix will cause an error code to be returned (as there's no next element).

---

```
int PrevCoordinates(DSL_intArray &coords) const;
```

Modifies the specified multidimensional coordinates to be the equivalent of the next linear index. Returns DSL\_OKAY on success, or an error code on failure. Specifying the coordinates representing the first element of the matrix will cause an error code to be returned (as there's no previous element).

---

```
DSL_doubleArray& GetItems();  
const DSL_doubleArray& GetItems() const;
```

Returns the reference the linear buffer containing the elements of the matrix.

---

```
const DSL_intArray& GetDimensions() const;
```

Returns the dimensions of the matrix.

---

```
const DSL_intArray& GetPreProduct() const;
```

Returns the reference to the preproduct array, which is useful for converting between linear and multidimensional coordinates.

---

```
int GetNumberOfDimensions() const;
```

Returns the number of dimensions of the matrix.

---

```
int GetLastDimension() const;
```

Returns the index of the last dimension of the matrix (not the *size* of the last dimension)

---

```
int GetSizeOfDimension(int dimIdx) const;
```

Returns the size of the specified matrix dimension.

---

```
int GetSize() const;
```

Returns the total size of the matrix (which is a product of the dimensions).

---

```
int Sum(const DSL_Dmatrix &a, const DSL_Dmatrix &b);
```

Adds two matrices and stores the result in this matrix. The matrix must have dimensions compatible with both operands, or an error is returned. Returns DSL\_OKAY on success.

---

```
int Subtract(const DSL_Dmatrix &a, const DSL_Dmatrix &b);
```

Subtracts two matrices and stores the result in this matrix. The matrix must have dimensions compatible with both operands, or an error is returned. Returns DSL\_OKAY on success.

---

---

```
int Add(const DSL_Dmatrix &m);
```

Adds the specified matrix to this matrix. The matrix must have dimensions compatible with both operands, or an error is returned. Returns DSL\_OKAY on success.

---

```
int Add(double x);
```

Adds a scalar value to all elements in the matrix. Returns DSL\_OKAY.

---

```
int Multiply(double x);
```

Multiplies all elements in the matrix by a scalar value. Returns DSL\_OKAY.

---

```
int Multiply(DSL_doubleArray &v);
```

Multiplies each element of the matrix with coordinates (a,b,c,...,z=N) by the Nth element of vector v.

---

```
int FillWith(double x);
```

Fills the matrix with a specified scalar value.

---

```
int AddDimension(int dimSize);
```

Adds a dimension with a specified size.

---

```
int AddDimensions(const DSL_intArray &newDimensions);
```

Adds multiple dimensions with specified sizes.

---

```
int InsertDimension(int dimIdx, int dimSize);
```

Inserts a dimensions with a specified size at a specified index.

---

```
int RemoveDimension(int dimIdx);
```

Removes the specified dimension. Uses the original elements with the coordinate value of zero at dimIdx.

---

```
int RemoveDimension(int dimIdx, int elemIdx);
```

Removes the specified dimension. Uses the original elements with the coordinate value of `elemIdx` at `dimIdx`.

## 7.11 DSL\_dataset

---

Header file: `dataset.h`

---

```
DSL_dataset();
DSL_dataset(const DSL_dataset &src);
DSL_dataset& operator=(const DSL_dataset &src);
~DSL_dataset();
```

Default constructor, copy constructor, assignment operator and destructor are defined.

---

```
int ReadFile(const std::string &filename,
             const DSL_datasetParseParams *params = NULL,
             std::string *errOut = NULL);
```

Reads the contents of the dataset from the text file. Returns `DSL_OKAY` on success or an error code on failure. If `errOut` is not `NULL`, the additional information about the error is returned.

The parser reads the first line from the file and searches for the following separator characters: tab, comma, semicolon, space (in this order). The first character found is considered to be the separator.

The types of dataset variables are determined as:

- if the data column in the file contains non-numeric entries, the corresponding dataset variable is string discrete
- if the data column in the file contains only numeric entries and there's at least one fractional value, the corresponding dataset variable is numeric continuous
- otherwise the dataset variable is numeric discrete

To customize parsing, you can pass the pointer to the `DSL_datasetParseParams` struct. The structure is declared in `dataset.h` as:

```
struct DSL_datasetParseParams
{
    DSL_datasetParseParams() :
        missingValueToken("*"),
```

```

        missingInt(DSL_MISSING_INT),
        missingFloat(DSL_MISSING_FLOAT),
        columnIdsPresent(true) {}
std::string missingValueToken;
int missingInt;
float missingFloat;
bool columnIdsPresent;
};

```

---

```

int WriteFile(const std::string &filename,
              const DSL_datasetWriteParams *params = NULL,
              std::string *errOut = NULL) const;

```

Writes the contents of the dataset to the text file. Returns DSL\_OKAY on success or an error code on failure. If errOut is not NULL, the additional information about the error is returned.

To customize parsing, you can pass the pointer to the DSL\_datasetWriteParams struct. The structure is declared in dataset.h as:

```

struct DSL_datasetWriteParams
{
    DSL_datasetWriteParams() :
        missingValueToken("*"),
        columnIdsPresent(true),
        useStateIndices(false),
        separator('\t'),
        floatFormat("%g") {}
    std::string missingValueToken;
    bool columnIdsPresent;
    bool useStateIndices;
    char separator;
    std::string floatFormat;
};

```

---

```

int MatchNetwork(const DSL_network &net,
                std::vector<DSL_datasetMatch> &matching,
                std::string &errMsg);

```

Attempts to match the contents of the dataset to the structure of the network specified as first argument (typically before parameter learning or network validation). May change the integer indices in the dataset to ensure the correct fit with outcome ids in the network nodes, therefore it is a non-const method.

On success, the vector of DSL\_datasetMatch objects is returned in the matching argument and the method returns DSL\_OKAY. To successfully match network and data, at least one node and dataset variable have to have identical identifier, and

- either both node and dataset variable are continuous, or

- both node and dataset variable are discrete, and all values in the dataset variable can be mapped onto node outcomes

When the network and the dataset cannot be matched, the error code is returned and additional human-readable information about the error is written to `errMsg` parameter.

---

```
int AddIntVar(const std::string id = std::string(),  
             int missingValue = DSL_MISSING_INT);
```

Adds discrete integer variable to the dataset. Note that you need to call `DSL_dataset::SetStateNames` later if you want to assign string values to integer indices. Returns `DSL_OKAY` on success or error code on failure.

Multiple variables with empty identifiers are allowed.

---

```
int AddFloatVar(const std::string id = std::string(),  
               float missingValue = DSL_MISSING_FLOAT);
```

Adds continuous, floating point variable to the dataset. Returns `DSL_OKAY` on success or error code on failure.

Multiple variables with empty identifiers are allowed.

---

```
int RemoveVar(int var);
```

Removes a variable from the dataset. Returns `DSL_OKAY` on success or error code on failure.

---

```
void AddEmptyRecord();
```

Appends a record with all values missing.

---

```
void SetNumberOfRecords(int numRecords);
```

Sets the number of records in the dataset. If the new number is greater than current, new records will have all values missing.

---

```
int RemoveRecord(int rec);
```

Removes the specified record from the dataset. Returns `DSL_OKAY` on success or error code on failure.

---

```
int FindVariable(const std::string &id) const;
```

Returns the index of the variable with the specified identifier, or a negative error code on failure.

---

```
int GetNumberOfVariables() const;
```

Returns the number of variables in the dataset.

---

```
int GetNumberOfRecords() const;
```

Returns the number of records in the dataset.

---

```
int GetInt(int var, int rec) const;
```

Returns the integer data value in the specified variable and record.

---

```
float GetFloat(int var, int rec) const;
```

Returns the floating data value in the specified variable and record.

---

```
void SetInt(int var, int rec, int value);
```

Sets the integer data value in the specified variable and record.

---

```
void SetFloat(int var, int rec, float value);
```

Sets the floating data value in the specified variable and record.

---

```
void SetMissing(int var, int rec);
```

Marks the data element in the specified variable and record as missing.

---

```
bool IsMissing(int var, int rec) const;
```

Returns true if the data element in the specified variable and record is missing.

---

```
int GetMissingInt(int var) const;
```

---



Returns the integer value representing missing data in the specified discrete variable.

---

```
float GetMissingFloat(int var) const;
```

Returns the floatvalue representing missing data in the specified continuous variable.

---

```
bool IsDiscrete(int var) const;
```

Returns true if specified variable is discrete.

---

```
enum DiscretizeAlgorithm { Hierarchical, UniformWidth, UniformCount };  
int Discretize(int var, DiscretizeAlgorithm alg, int intervals,  
    const std::string &statePrefix, std::vector<double> &edges);  
int Discretize(int var, DiscretizeAlgorithm alg, int intervals,  
    const std::string &statePrefix);
```

Discretizes a dataset variable. Returns DSL\_OKAY on success or error code on failure. The first overload also returns the values of discretization interval edges.

## 7.12 DSL\_dataGenerator

---

Header file: `datagenerator.h`

---

```
DSL_dataGenerator(DSL_network &net);
```

To create a DSL\_dataGenerator instance you need to pass a reference to DSL\_network, which will be used as a source probability distribution for the data generator.

---

```
int GenerateData(DSL_dataset &ds);
```

Generate data and store the results in the DSL\_dataset

---

```
int GenerateData(const char *filename,  
    const DSL_datasetWriteParams *params = NULL);
```

Generate data and write the results to the text file. To fine tune the output format pass the pointer to the DSL\_datasetWriteParams object.

---

```
int GenerateData(DSL_dataGeneratorOutput &out);
```

Generate data and write the results to the abstracted output. In order to use this method, create a class derived from `DSL_dataGeneratorOutput`, which is a pure abstract class declared in `datagenerator.h` header.

---

```
void SetNumberOfRecords(int numrec);  
int GetNumberOfRecords() const;
```

Set/get the number of records to generate

---

```
void SetRandSeed(int seed);  
int GetRandSeed() const;
```

Set/get the seed used to initialize the random generator. Defaults to zero, which causes the value based on system clock to be used as seed.

---

```
void SetMissingValuePercent(int perc);  
int GetMissingValuePercent() const;
```

Set/get the percentage of missing values. Defaults to zero.

---

```
void SetBiasSamplesByEvidence(bool bias);  
bool GetBiasSamplesByEvidence() const;
```

If set to true, generates a data file from the posterior joint probability distribution (i.e., biased by the observations) rather than from the original joint probability distribution. Defaults to false.

---

```
int SetSelectedNodes(const std::vector<int> &selection);  
const std::vector<int>& GetSelectedNodes() const;
```

Set/get the nodes included in the output from `GenerateData`. By default the selection vector is empty, which means that all nodes will be included.

## 7.13 DSL\_validator

---

Header file: **validator.h**

---

```
DSL_validator(  
    DSL_dataset& ds, const DSL_network &net,  
    const std::vector<DSL_datasetMatch> &matching,  
    const std::vector<int> *fixedNodes = 0);
```

The constructor requires a reference to the dataset, the network and the vector of `DSL_datasetMatch` objects. If `KFold` or `LeaveOneOut` are going to be used, it is also possible to specify which nodes in the network do not change their parameters by passing their handles in the `fixedNodes` argument.

---

```
int AddClassNode(int classNodeHandle);
```

Adds class node to the internal list of class nodes. Returns `DSL_OKAY` on success or an error code on failure.

---

```
int Test(DSL_progress *progress = 0);
```

Performs testing using the dataset specified in the constructor. The network does not change its parameters during the procedure. Returns `DSL_OKAY` on success or an error code on failure.

The optional argument `progress` can be used to stop the testing by returning false from `DSL_progress::Tick` method, which is called periodically within the main loop of the learning algorithm. In such case, the method returns `DSL_INTERRUPTED`.

---

```
int KFold(DSL_em &em, int foldCount, int randSeed = 0,  
    DSL_progress *progress = 0);
```

Performs the K-fold crossvalidation using the dataset specified in the constructor. Returns `DSL_OKAY` on success or an error code on failure.

The internal parameter learning is performed with the `em` object. The network specified in the constructor does not change its parameters; EM runs on a copy of the network.

The number of folds is specified with the `foldCount` parameter.

The folds are created by randomly splitting records in the datasets into subsets. The random generator is initialized with the `randSeed` parameter. The value of this parameter defaults to zero, which causes the value based on system clock to be used as seed.

The optional argument `progress` can be used to stop the testing by returning false from `DSL_progress::Tick` method, which is called periodically within the main loop of the learning algorithm. In such case, the method returns `DSL_INTERRUPTED`.

---

```
int LeaveOneOut(DSL_em &em, DSL_progress *progress = 0);
```

Performs the Leave-one-out crossvalidation using the dataset specified in the constructor. The network does not change its parameters during the procedure. Returns DSL\_OKAY on success or an error code on failure.

The internal parameter learning is performed with the em object. The network specified in the constructor does not change its parameters; EM runs on a copy of the network.

The optional argument progress can be used to stop the testing by returning false from DSL\_progress::Tick method, which is called periodically within the main loop of the learning algorithm. In such case, the method returns DSL\_INTERRUPTED.

---

```
int GetPosteriors(int classNodeHandle, int recordIndex,  
    std::vector<double> &posteriors) const;
```

Fills the posteriors vector with the probabilities calculated for classNodeHandle using the record from the dataset with the index specified by the recordIndex.

Returns DSL\_OKAY on success or an error code on failure.

---

```
int GetAccuracy(int classNodeHandle, int outcome, double &acc) const;
```

Gets the accuracy for the specified class node and its outcome. Returns DSL\_OKAY on success or an error code on failure.

---

```
int GetConfusionMatrix(int classNodeHandle,  
    std::vector<std::vector<int> > &matrix) const;
```

Gets the confusion matrix for the specified class node and its outcome. Returns DSL\_OKAY on success or an error code on failure.

---

```
int GetPredictedOutcome(int classNodeHandle, int recordIndex) const;
```

Gets the predicted outcome for the specified class node and record index. Returns DSL\_OKAY on success or an error code on failure.

---

```
int GetPredictedNode(int recordIndex) const;  
int GetPredictedNodeIndex(int recordIndex) const;
```

Gets the predicted node handle or index for the specified record index. The node prediction is based on the probabilities of most likely outcomes in class nodes. Returns DSL\_OKAY on success or an error code on failure.

---

```
int GetFoldIndex(int recordIndex) const;
```

Gets the fold to which the specified dataset record belongs. Returns DSL\_OKAY on success or an error code on failure.

---

```
void GetResultDataset(DSL_dataset &output) const;
```

Fills the output dataset with the content of the input dataset (specified in the constructor) and calculated class node probabilities and predicted outcomes.

---

```
int CreateROC(int classNodeHandle, int outcomeIndex,  
    std::vector<std::pair<double, double> > &curve,  
    std::vector<double> &thresholds) const;
```

Creates the ROC curve for the specified class node and its outcome. Returns DSL\_OKAY on success or an error code on failure.

---

```
int CalibrateByBinning(int classNodeHandle, int outcomeIndex, int binCount,  
    std::vector<std::pair<double, double> > &curve,  
    double &hosmerLemeshTest) const;  
int CalibrateByMovingAverage(int classNodeHandle, int outcomeIndex,  
    int windowSize, std::vector<std::pair<double, double> > &curve) const;
```

Create the calibration curve for the specified class node and its outcome. Returns DSL\_OKAY on success or an error code on failure.

## 7.14 DSL\_em

---

Header file: **em.h**

---

```
DSL_em();
```

The default constructor sets equivalent sample size to one, random seed to zero and parameter randomization to true.

---

```

int Learn(const DSL_dataset& ds, DSL_network& orig,
          const std::vector<DSL_datasetMatch> &matches,
          double *loglik = NULL, DSL_progress *progress = 0);
int Learn(const DSL_dataset& ds, DSL_network& orig,
          const std::vector<DSL_datasetMatch> &matches,
          const std::vector<int> &fixedNodes,
          double *loglik = NULL, DSL_progress *progress = 0);

```

Learns network parameters in the specified network using data from the dataset using the EM algorithm. Returns DSL\_OKAY on success or an error code on failure.

Network nodes and dataset variables are matched through the DSL\_datasetMatch vector specified through matches argument. Typically, this vector is obtained by a call to DSL\_dataset::MatchNetwork, but it can also be created by your program if node and variable identifiers do not match.

The second overload should be used when some nodes' parameters are assumed to be fixed. The handles of these nodes are passed in the fixedNodes argument.

The optional argument loglik can be used to obtain the log likelihood from the EM algorithm. This value, ranging from minus infinity to zero, is a measure of fit of the model to the data.

The optional argument progress can be used to stop the learning by returning false from DSL\_progress::Tick method, which is called periodically within the main loop of the learning algorithm. In such case, the Learn method returns DSL\_INTERRUPTED.

---

```

void SetRandomizeParameters(bool r);
bool GetRandomizeParameters() const;

```

Set/get the value of parameter randomization flag. If set to true, the network parameters will be randomized before entering the main loop of the EM algorithm. Defaults to true.

---

```

void SetUniformizeParameters(bool u);
bool GetUniformizeParameters() const;

```

Set/get the value of parameter uniformization flag. If set to true, the network parameters will be uniformized before entering the main loop of the EM algorithm. Defaults to false.

---

```

void SetSeed(int seed);
int GetSeed() const;

```

Set/get the seed used to initialize the random generator. Defaults to zero, which causes the value based on system clock to be used as seed.

---

```
void SetEquivalentSampleSize(float eqs);  
float GetEquivalentSampleSize() const;
```

Set/get the equivalent sample size. The equivalent sample size, also known as confidence, which can be interpreted as the number of records that the current network parameters are based on. The larger the value, the less weight is assigned to the new cases, which gives a mechanism for gentle refinement of model numerical parameters. The interpretation of this parameter is obvious when the entire network or its parameters have been learned from data - it should be equal to the number of records in the data file from which they were learned.

Defaults to 1. Call `SetRandomizeParameters(false)` and `SetUniformizeParameters(false)` if you want to use larger values as equivalent sample sizes.

## 7.15 DSL\_bs

---

Header file: `bs.h`

---

```
DSL_bs();
```

The default constructor.

---

```
int Learn(const DSL_dataset &ds_, DSL_network &net,  
    DSL_progress *progress = NULL, DSL_bsEvaluator *eval = NULL,  
    double *bestScore = NULL, int *bestIteration = NULL) const;
```

Creates network structure using Bayesian Search algorithm, then learns the parameters with EM using the specified dataset. Each variable in the dataset is represented by a node in the network after learning is complete. Returns `DSL_OKAY` on success or an error code on failure.

The optional argument `progress` can be used to stop the learning by returning false from `DSL_progress::Tick` method, which is called periodically within the main loop of the learning algorithm. In such case, the `Learn` method returns `DSL_INTERRUPTED`.

The optional argument `eval` may be used to provide an alternative structure evaluator.

The optional output arguments `bestScore` and `bestIteration` can be used to obtain the score for the network structure selected by the algorithm and the iteration index corresponding to that network structure.

---

**int nrIteration;**

Number of iterations to perform in the main structure learning loop. Each iteration starts with random structure and is refined until convergence. Defaults to 20.

---

**int maxParents;**

Maximum number of parents in the learned network. Defaults to 5.

---

**int maxSearchTime;**

Maximum search time (in seconds) for the structure learning to run. Elapsed time is checked after each iteration is complete. Defaults to zero, meaning no time limit.

---

**int seed;**

The seed used to initialize the random generator. Defaults to zero, which causes the value based on system clock to be used as seed.

---

**int priorSampleSize;**

Takes part in the score calculation, representing the inertia of the current parameters when introducing new data. Defaults to 50.

---

**double linkProbability;**

The parameter used when generating a random starting network at the outset of each of the iterations. It essentially influences the connectivity of the starting network. Defaults to 0.1.

---

**double priorLinkProbability;**

Influences the score, by offering a prior over all edges. It comes into the formula in the following way:

$$\log \text{Posterior score} = \log \text{marginal likelihood (i.e., the } BDeu) + |parents| * \log(pll) + (|nodes| - |parents| - 1) * \log(1 - pll)$$



Defaults to 0.001.

---

**DSL\_bkgndKnowledge bkk;**

Background knowledge used to constrain the network structures created by structure learning algorithm. Empty by default.

## 7.16 DSL\_pc

---

Header file: pc.h

---

**DSL\_pc();**

The default constructor.

---

**int Learn(const DSL\_dataset &ds, DSL\_pattern &pat,  
DSL\_progress \*progress = NULL) const;**

Based on the specified dataset, creates a graph using PC algorithm and stores the graph edges in the specified DSL\_pattern. Each variable in the dataset is represented by a node in the pattern after learning is complete. Returns DSL\_OKAY on success or an error code on failure.

The output of the PC algorithm - DSL\_pattern object - can be converted to DSL\_network with uniform probability distributions with a call to DSL\_pattern::ToNetwork.

The optional argument progress can be used to stop the learning by returning false from DSL\_progress::Tick method, which is called periodically within the main loop of the learning algorithm. In such case, the Learn method returns DSL\_INTERRUPTED.

---

**int maxAdjacency;**

Maximum number of neighbours of a node. Defaults to 8.

---

**int maxSearchTime;**

Maximum search time (in seconds) for the learning to run. Elapsed time is checked after each iteration is complete. Defaults to zero, meaning no time limit.

---

**double significance;**

Alpha value used in classical independence tests on which the PC algorithm rests. Defaults to 0.05.

---

**DSL\_bkgndKnowledge bkk;**

Background knowledge used to constrain the network structures created by structure learning algorithm. Empty by default.

## 7.17 DSL\_tan

---

Header file: **tan.h**

---

**DSL\_tan();**

The default constructor.

---

```
int Learn(DSL_dataset &ds, DSL_network &net,
          DSL_progress *progress = NULL,
          double *emLogLik = NULL) const;
```

Creates network structure using Tree Augmented Naive Bayes (TAN) algorithm, then learns the parameters with EM using the specified dataset. Each variable in the dataset is represented by a node in the network after learning is complete. Returns DSL\_OKAY on success or an error code on failure.

The algorithm produces an acyclic directed graph with the class variable being the parent of all the other (feature) variables and additional connections between the feature variables.

The optional argument `progress` can be used to stop the learning by returning false from `DSL_progress::Tick` method, which is called periodically within the main loop of the learning algorithm. In such case, the `Learn` method returns DSL\_INTERRUPTED.

---

**std::string classvar;**

Identifier of the class variable.

---

**int maxSearchTime;**

Maximum search time (in seconds) for the structure learning to run. Elapsed time is checked after each iteration is complete. Defaults to zero, meaning no time limit.

---

**int seed;**

The seed used to initialize the random generator. Defaults to zero, which causes the value based on system clock to be used as seed.

## 7.18 DSL\_abn

---

Header file: **abn.h**

---

**DSL\_abn();**

The default constructor.

---

```
int Learn(DSL_dataset &ds, DSL_network &net,
          DSL_progress *progress = NULL,
          double *bestScore = NULL, int *bestIteration = NULL,
          double *emLogLik = NULL) const;
```

Creates network structure using Augmented Naive Bayes (ABN) algorithm, then learns the parameters with EM using the specified dataset. Each variable in the dataset is represented by a node in the network after learning is complete. Returns DSL\_OKAY on success or an error code on failure.

The algorithm produces an acyclic directed graph with the class variable being the parent of all the other (feature) variables and additional connections between the feature variables.

The optional argument `progress` can be used to stop the learning by returning false from `DSL_progress::Tick` method, which is called periodically within the main loop of the learning algorithm. In such case, the `Learn` method returns DSL\_INTERRUPTED.

---

**std::string classvar;**

Identifier of the class variable.

---

**bool feature\_selection;**

If true, invokes an additional function that removes from the feature set those features that do not contribute enough to the classification

---

```
int maxParents;
int maxSearchTime;
int nrIteration;
double linkProbability;
double priorLinkProbability;
int priorSampleSize;
int seed;
```

See the [DSL\\_bs](#)<sup>[135]</sup> reference.

## 7.19 DSL\_nb

---

Header file: **nb.h**

---

```
DSL_nb();
```

The default constructor.

---

```
int Learn(DSL_dataset ds, DSL_network &net,
          DSL_progress *progress = NULL,
          double *emLogLikelihood = NULL) const;
```

Creates the naive Bayes network, then learns the parameters with EM using the specified dataset. Returns DSL\_OKAY on success or an error code on failure.

The structure of a Naive Bayes network is not learned but rather fixed by assumption: the class variable is the only parent of all remaining, feature variables and there are no other connections between the nodes of the network.

The optional argument progress can be used to stop the learning by returning false from DSL\_progress::Tick method, which is called periodically within the main loop of the learning algorithm. In such case, the Learn method returns DSL\_INTERRUPTED.

---

```
std::string classVariableId;
```

Identifier of the class variable.

## 7.20 DSL\_bkgndKnowledge

---

Header file: `bkgndknowledge.h`

---

```
typedef std::vector<std::pair<int, int> > IntPairVector;
```

Type defined for convenience.

---

```
IntPairVector forcedArcs;
```

Arcs which are required to be present in the learned network structure

---

```
IntPairVector forbiddenArcs;
```

Arcs which are forbidden in the learned network structure

---

```
IntPairVector tiers;
```

Enforces temporal tiers in the network: there will be no arcs from nodes in higher tiers to nodes in lower tiers. Each element of the vector contains node handle in `pair::first` and the index of the tier in the `pair::second` member.

---

## 7.21 DSL\_pattern

---

Header file: `pattern.h`

---

```
enum EdgeType {None,Undirected,Directed};
```

Edge type identifiers

---

```
void SetSize(int size);  
int GetSize() const;
```

Sets/gets the number of nodes in the pattern

---

---

**EdgeType GetEdge(int from, int to) const;**

Returns the edge type between from and to, where from and to are zero-based indices of the nodes in the pattern. If there's no edge, EdgeType::None is returned.

---

**void SetEdge(int from, int to, EdgeType type);**

Sets the edge type between from and to, where from and to are zero-based indices of the nodes in the pattern.. To remove the edge, set the type parameter to EdgeType::None

---

**bool HasDirectedPath(int from, int to) const;**

Returns true if there's a directed path between from and to, where from and to are zero-based indices of the nodes in the pattern.

---

**bool HasCycle() const;**

Returns true if the pattern contains a cycle

---

**bool IsDAG() const;**

Returns true if the pattern is a directed acyclic graph - there are no cycles and all edges are directed.

---

**bool ToDAG();**

Attempts the conversion of pattern to a directed acyclic graph. Returns true if successful.

---

**bool ToNetwork(const DSL\_dataset &ds, DSL\_network &net);**

Attempts the conversion of pattern to a directed acyclic graph. If successful, creates DSL\_network with node identifiers and outcomes based on the specified dataset. It is assumed that the dataset was used previously to obtain this pattern, therefore the order of variables in the dataset corresponds to the order of nodes in the pattern (and in the resulting network). Returns true if successful.

---

**bool HasIncomingEdge(int to) const;**  
**bool HasOutgoingEdge(int from) const;**

Return true if there's an edge incoming or outgoing to/from the specified node.

---

```
void GetAdjacentNodes(const int node, std::vector<int>& adj) const;  
void GetParents(const int node, std::vector<int>& par) const;  
void GetChildren(const int node, std::vector<int>& child) const;
```

Return the adjacent/parent/children nodes of the specified node.

## 7.22 DSL\_progress

---

Header file: `progress.h`

---

```
virtual bool Tick(double percComplete = -1, const char *msg = NULL) = 0;
```

DSL\_progress class declares single pure virtual function, which must be overridden in the derived classes. SMILE algorithms supporting DSL\_progress periodically call the Tick method with percComplete parameter specifying the percentage (0..100) of work performed so far. If the algorithm can't predict the number of iterations of its main loop, -1 is used as the percentage.

The string pointed to by msg parameter should be copied if the class derived from DSL\_progress uses it for display or logging purposes after the Tick call is finished.

Returning false from Tick causes the calling algorithm to quit with DSL\_INTERRUPTED error code.

## 7.23 Equations

---

The subsections of this chapter describe the notation used when specifying the definition for equation nodes and expression-based MAU nodes.

### 7.23.1 Operators

#### Arithmetic operators

**+** addition, e.g., if  $x=3$  and  $y=2$ ,  $x+y$  produces 5

- subtraction and unary minus, e.g., if  $x=3$  and  $y=2$ ,  $x-y$  produces 1 and  $-x$  produces -3
- $\wedge$  exponentiation ( $a^b$  means  $a^b$ ), e.g., if  $x=3$  and  $y=2$ ,  $x^y$  produces 9
- $*$  multiplication, e.g., if  $x=3$  and  $y=2$ ,  $x*y$  produces 6
- / division, e.g., if  $x=3$  and  $y=2$ ,  $x/y$  produces 1.5

## Comparison operators

- > greater than, e.g., if  $x=3$  and  $y=2$ ,  $x>y$  produces 1
- < smaller than, e.g., if  $x=3$  and  $y=2$ ,  $x<y$  produces 0
- >= greater or equal than, e.g., if  $x=3$  and  $y=2$ ,  $x>=y$  produces 1
- <= smaller or equal than, e.g., if  $x=3$  and  $y=2$ ,  $x<=y$  produces 0
- <> not equal, e.g., if  $x=3$  and  $y=2$ ,  $x<>y$  produces 1
- = equal, e.g., if  $x=3$  and  $y=2$ ,  $x=y$  produces 0

## Conditional selection operator

**? :** ternary conditional operator like in C, C++ or Java programming languages.

This operator is essentially a shortcut to the If function. For example,  $a=b?5:3$  is equivalent to  $\text{If}(a=b, 5, 3)$ .

## Order of calculation, operator precedence, and parentheses

Expressions are evaluated from left to right, according to the precedence order specified below (1 denotes the highest precedence).

Precedence order	Operator
1	- (unary minus)



Precedence order	Operator
2	$\wedge$ (exponentiation)
3	$*$ and $/$ (multiplicative operators)
4	$+$ and $-$ (additive operators)
5	$>$ , $<$ , $>=$ , $<=$ , $=$ (comparison operators)
6	<b>?:</b> (conditional selection)

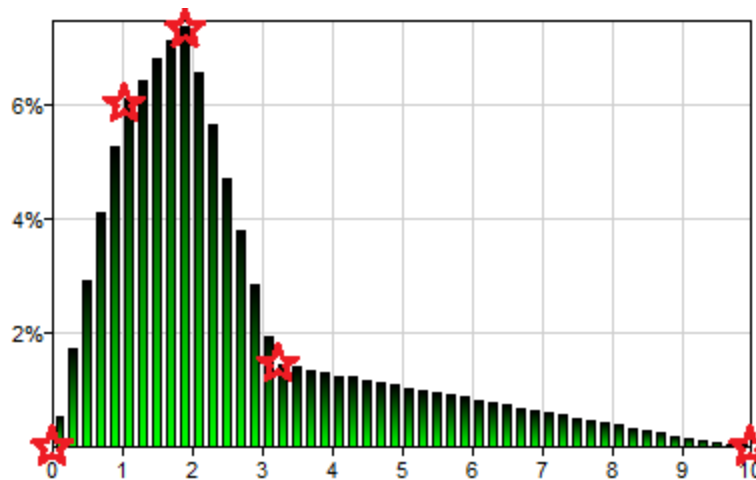
To change the order of calculation, enclose in parentheses those parts of the formula that should be calculated first. This can be done recursively, i.e., parentheses can be nested indefinitely. For example, if  $x=3$  and  $y=2$ ,  $2*y+x/3-y+1$  produces 4,  $2*(y+x)/(3-y)+1$  produces 11, and  $2*((y+x)/(3-y)+1)$  produces 12.

### 7.23.2 Probability Distributions

Probability distribution functions **each generate a single sample** from the distributions defined below. In most equations, they can be imagined as random noise that distort the equation. Because the fundamental algorithm for inference in continuous and hybrid models is stochastic simulation, it is possible to visualize what probability distributions these single errors result in for each of the variables in the model. Probability distributions are not allowed in expression-based MAU nodes.

#### CustomPDF(x1,x2,...y1,y2,...)

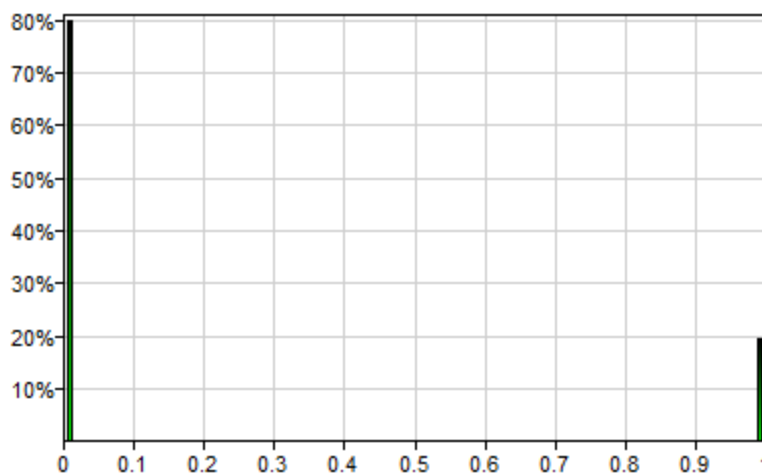
The CustomPDF distribution allows for specifying a non-parametric continuous probability distribution by means of a series of points on its probability density (PDF) function. Pairs  $(x_i, y_i)$  are coordinates of such points. The total number of parameters of CustomPDF function should thus to be even. Please note that  $x$  coordinates should be listed in increasing order. The PDF function specified does not need to be normalized, i.e., the area under the curve does not need to add up to 1.0. For example, `CustomPDF(0,1.02,1.9,3.2,10,0,4,5,1,0)` generates a single sample from the following distribution:



Stars on the plot mark the points defined by the CustomPDF arguments, i.e., (0, 0), (1.02, 4), (1.9, 5), (3.2, 1), and (10, 0).

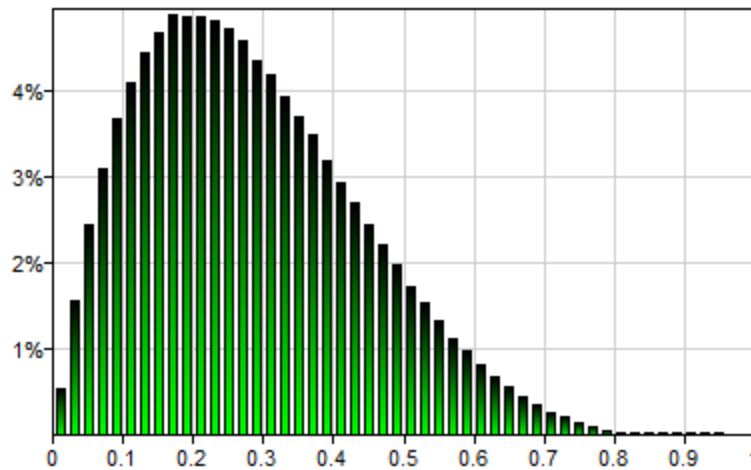
### **Bernoulli( $p$ )**

Bernoulli is a discrete distribution that generates 0 with probability  $1-p$  and 1 with probability  $p$ . `Bernoulli(0.2)` will generate a single sample (0 or 1) from the following distribution, i.e., 1 with probability 0.2 and 0 with probability 0.8:



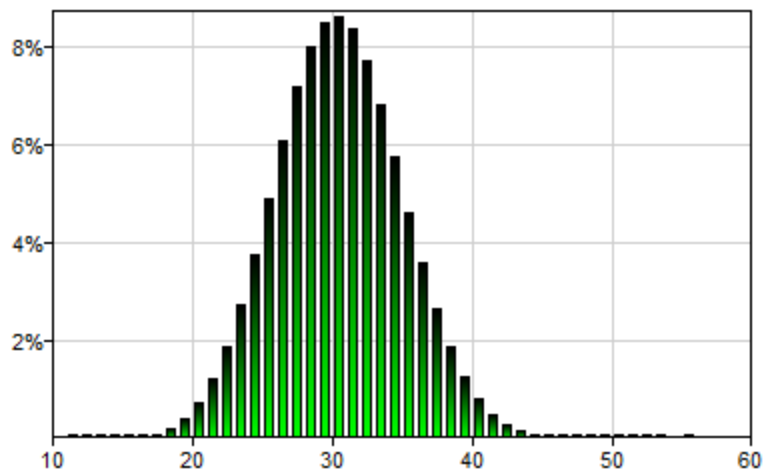
### **Beta( $a,b$ )**

The Beta distribution is a family of continuous probability distributions defined on the interval  $[0, 1]$  and parametrized by two positive shape parameters,  $a$  and  $b$  (typically denoted by  $\alpha$  and  $\beta$ ), that control the shape of the distribution. `Beta(2, 5)` will generate a single sample from the following distribution:



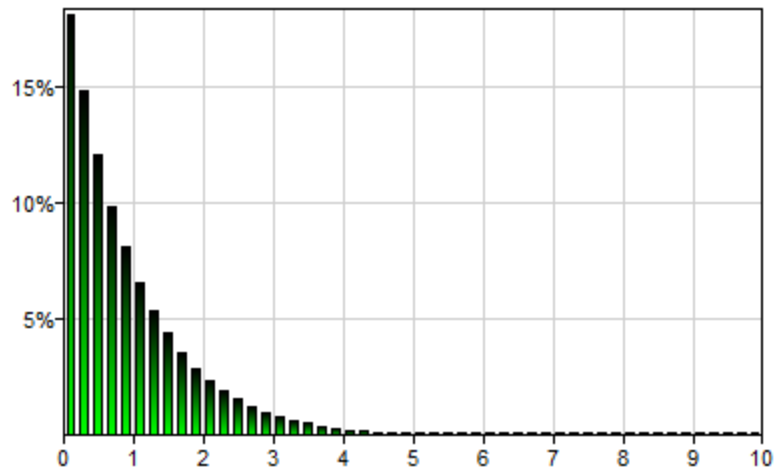
### Binomial( $n,p$ )

Binomial is a discrete probability distribution over the number of successes in a sequence of  $n$  independent trials, each of which yields a success with probability  $p$ . It will generate a single sample, which will be an integer number between 0 and  $n$ . A success/failure experiment is also called a Bernoulli trial. Hence,  $\text{Binomial}(1,p)$  is equivalent to  $\text{Bernoulli}(p)$ .  $\text{Binomial}(100,0.3)$  will generate a single sample from the following distribution:



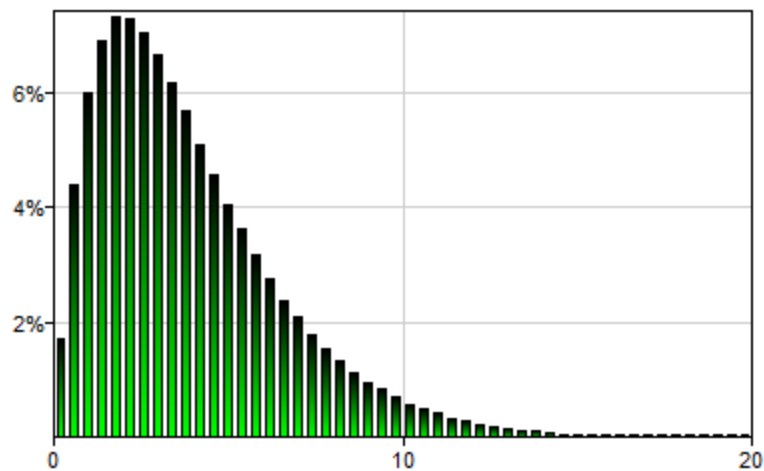
### Exponential( $\lambda$ )

The exponential distribution is a continuous probability distribution that describes the time between events in a Poisson process, i.e., a process in which events occur continuously and independently at a constant average rate. Its only real-valued, positive parameter  $\lambda$  (typically denoted by  $\lambda$ ) determines the shape of the distribution. It is a special case of the Gamma distribution.  $\text{Exponential}(\lambda)$  generates a single sample from the domain  $(0,\infty)$ .  $\text{Exponential}(1)$  will generate a single sample from the following distribution:



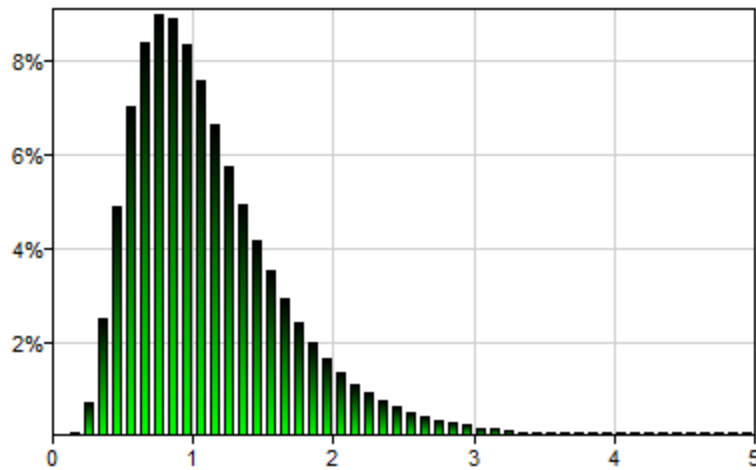
### Gamma(shape,scale)

The Gamma distribution is a two-parameter family of continuous probability distributions. There are different parametrizations of the Gamma distribution in common use. SMILE parametrization follows one of the most popular parametrizations, with *shape* (often denoted by  $k$ ) and *scale* (often denoted by  $\theta$ ) parameters, both positive real numbers.  $\text{Gamma}(2.0, 2.0)$  will generate a single sample from the following distribution:



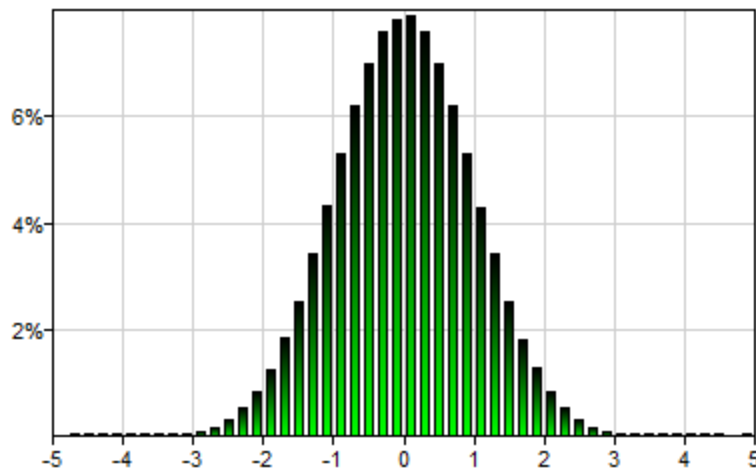
### Lognormal(mu,sigma)

The lognormal distribution is a continuous probability distribution of a random variable, whose logarithm is normally distributed. Thus, if a random variable  $X$  is lognormally distributed, then a variable  $Y = \ln(X)$  has a normal distribution. Conversely, if  $Y$  has a normal distribution, then  $X = e^Y$  has a lognormal distribution. A random variable which is lognormally distributed takes only positive values.  $\text{Lognormal}(0, 0.5)$  will generate a single sample from the following distribution:



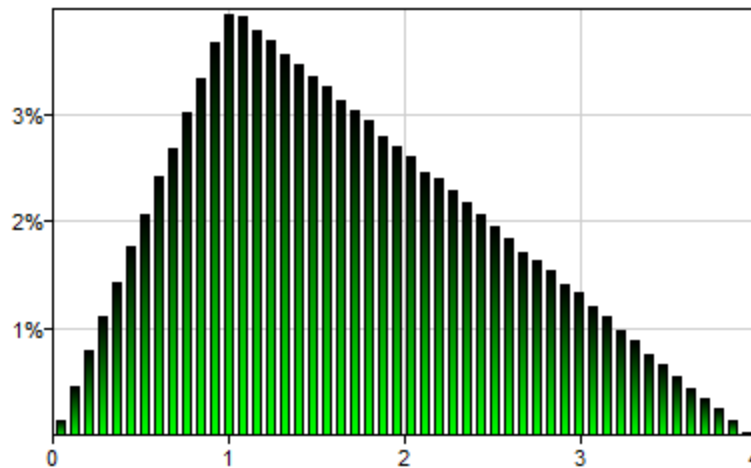
### Normal(mu,sigma)

Normal (also known as Gaussian) distribution is the most commonly occurring continuous probability distribution. It is symmetric and defined over the real domain. Its two parameters,  $\mu$  (mean,  $\mu$ ) and  $\sigma$  (standard deviation,  $\sigma$ ), control the position of its mode and its spread respectively. `Normal(0,1)` will generate a single sample from the following distribution:



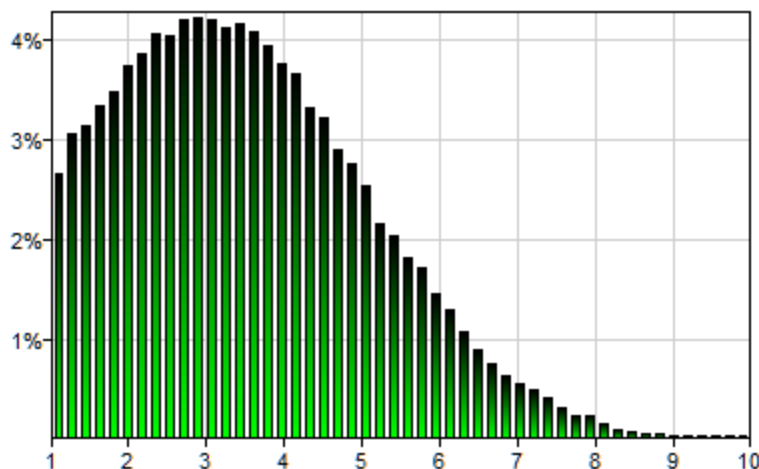
### Triangular(min,mod,max)

Triangular distribution is a continuous probability distribution with lower limit  $\min$ , upper limit  $\max$  and mode  $\text{mod}$ , where  $\min \leq \text{mod} \leq \max$ . `Triangular(0,1,3)` will generate a single sample from the following distribution:



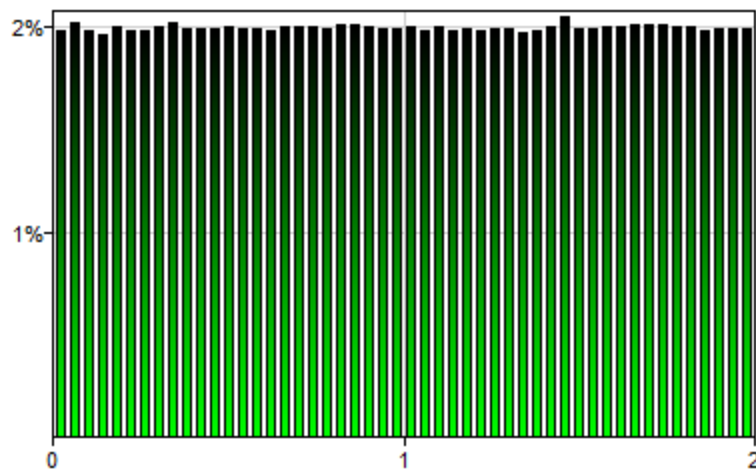
### **TruncNormal(mu,sigma,min)**

Truncated Normal distribution is essentially a Normal distribution that is truncated at the value  $min$  ( $min \leq mu$ ). This distribution is especially useful in situation when we want to limit physically impossible values in the model. The constraint  $min \leq mu$  is not restrictive in practice and allows to control efficiency of sample generation, which is performed by rejecting samples smaller than  $min$  from a  $Normal(mu, sigma)$  distribution. `TruncNormal(3,2,1)` will generate a single sample from the following distribution:



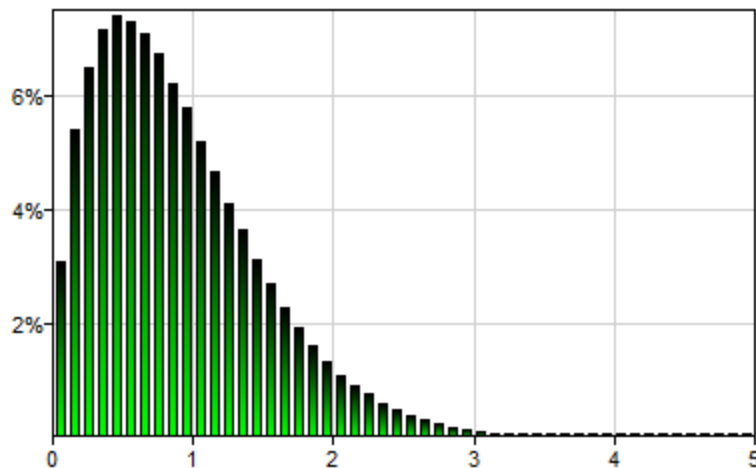
### **Uniform(a,b)**

The continuous uniform distribution, also known as the rectangular distribution, is a family of probability distributions under which any two intervals of the same length are equally probable. It is defined two parameters,  $a$  and  $b$ , which are the minimum and the maximum values of the random variable. `Uniform(0,2)` will generate a single sample from the following distribution:



### Weibull(lambda,k)

Weibull distribution is a continuous probability distribution named after a Swedish mathematician Waloddi Weibull, used in modeling such phenomena as particle size. It is characterized by two positive real parameters: the scale parameter  $\lambda$  and the shape parameter  $k$ . `Weibull(1,1.5)` will generate a single sample from the following distribution:



## 7.23.3 Arithmetic Functions

### Abs(x)

Returns the absolute value of a number, e.g.,  $\text{Abs}(5.3) = \text{Abs}(-5.3) = 5.3$

### Exp(x)

Returns  $e$  (Euler's number) raised to the power of  $x$ , e.g.,  $\text{Exp}(2.2) = e^{2.2} = 9.02501$

**GammaLn(x)**

Returns the natural logarithm of the Gamma function ( $\Gamma(x)$ ), e.g.,  $\text{GammaLn}(2.2)=0.0969475$

**GCD(n,k)**

Returns the greatest common integer divisor of its two integer arguments  $n$  and  $k$ , e.g.,  $\text{GCD}(15,25)=5$ . When the arguments are not integers, their fractional part is ignored.

**LCM(n,k)**

Returns the least common integer multiple of its two integer arguments  $n$  and  $k$ , e.g.,  $\text{LCM}(15,25)=75$ . When the arguments are not integers, their fractional part is ignored.

**Ln(x)**

Returns the natural logarithm of  $x$ , which has to be non-negative, e.g.,  $\text{Ln}(10)=2.30259$

**Log(x,b)**

Returns the base  $b$  logarithm of  $x$ , which has to be non-negative, e.g.,  $\text{Log}(10,2)=3.32193$

**Log10(x)**

Returns decimal logarithm of  $x$ , which has to be non-negative, e.g.,  $\text{Log10}(100)=2$

**Pow10(x)**

Returns 10 raised to the power of  $x$ , e.g.,  $\text{Pow10}(2)=10^2=100$

**Round(x)**

Returns the integer that is nearest to  $x$ , e.g.,  $\text{Round}(2.2)=2$ ,  $\text{Round}(3.5)=4$

**Sign(x)**

Returns 1 if  $x>0$ , 0 when  $x=0$ , and -1 if  $x<0$ , e.g.,  $\text{Sign}(2.2)=1$ ,  $\text{Sign}(0)=0$ ,  $\text{Sign}(-3.5)=-1$

**Sqrt(x)**

Returns the square root of  $x$ , which has to be non-negative, e.g.,  $\text{Sqrt}(2)=1.41421$

**SqrtPi(x)**



Returns square root of  $\pi$  multiplied by  $x$ , which has to be non-negative, e.g.  $\text{SqrtPi}(2)=\text{Sqrt}(\text{Pi}()*2)=2.50663$ . This function is provided for the sake of compatibility with Microsoft Excel.

### **Sum(x1,x2,...)**

Returns the sum of its arguments, e.g.,  $\text{Sum}(2.2, 3.5, 1.3)=7.0$ .  $\text{Sum}()$  requires at least two arguments.

### **SumSq(x1,x2,...)**

Returns the sum of squares of its arguments, e.g.,  $\text{SumSq}(2.2, 3.5, 1.3)=18.78$ .  $\text{SumSq}()$  requires at least two arguments.

### **Trim(x,lo,hi)**

Trims the value of the argument  $x$  to a value in the interval  $\langle lo, hi \rangle$ . If  $x \leq lo$ , the function returns  $lo$ , if  $x \geq hi$ , the function returns  $hi$ , if  $lo < x < hi$ , the function returns  $x$ . The function is a shortcut to two nested conditional functions  $\text{If}()$  and is equivalent to  $\text{If}(x < lo, lo, \text{If}(x > hi, hi, x))$ . For example,  $\text{Trim}(-0.5, 0, 1)=0$ ,  $\text{Trim}(0.5, 0, 1)=0.5$ ,  $\text{Trim}(1.5, 0, 1)=1$

### **Truncate(x)**

Returns the integer part of  $x$ , e.g.,  $\text{Truncate}(2.2)=2$

## **7.23.4 Combinatoric Functions**

### **Combin(n,k)**

Returns the number of combinations of distinct  $k$  elements from among  $n$  elements, e.g.,  $\text{Combin}(10, 2)=45$

### **Fact(n)**

Returns the factorial of  $n$ , e.g.,  $\text{Fact}(5)=5!=120$ .  $\text{Fact}$  of a negative number returns 0.

### **FactDouble(n)**

Returns the product of all even (when  $n$  is even) or all odd (when  $n$  is odd) numbers between 1 and  $n$ , e.g.,  $\text{FactDouble}(5)=15$ ,  $\text{FactDouble}(6)=48$ .  $\text{FactDouble}$  of a negative number returns 0.

### **Multinomial(n1,n2,...)**

Factorial of sum of arguments, divided by the factorials of all arguments, e.g.,  $\text{Multinomial}(2,5,3) = \text{Fact}(2+5+3) / (\text{Fact}(2) * \text{Fact}(5) * \text{Fact}(3)) = 10! / (2! * 5! * 3!) = 2520$ . All arguments of `Multinomial` have to be positive.

### 7.23.5 Trigonometric Functions

#### **Acos(x)**

Returns arccosine (*arcus cosinus*) of  $x$ , e.g.,  $\text{Acos}(-1) = 3.14159$

#### **Asin(x)**

Returns arcsine (*arcus sinus*) of  $x$ , e.g.,  $\text{Asin}(1) = 1.5708$

#### **Atan(x)**

Returns arctangent (*arcus tangens*) of  $x$ , e.g.,  $\text{Atan}(1) = 0.785398$

#### **Atan2(y,x)**

Returns arctangent (*arcus tangens*) from  $x$  and  $y$  coordinates, e.g.,  $\text{Atan2}(1,1) = 0.785398$

#### **Cos(x)**

Returns cosine (*cosinus*) of  $x$ , e.g.,  $\text{Cos}(1) = 0.540302$

#### **Pi()**

Returns constant  $\pi$ , e.g.,  $\text{Pi}() = 3.14159$

#### **Sin(x)**

Returns sine (*sinus*) of  $x$ , e.g.,  $\text{Sin}(1) = 0.841471$

#### **Tan(x)**

Returns tangent (*tangens*) of  $x$ , e.g.,  $\text{Tan}(1) = 1.55741$

### 7.23.6 Hyperbolic Functions

#### **Cosh(x)**

Returns the hyperbolic cosine of  $x$ , e.g.,  $\text{Cosh}(1) = 1.54308$

**Sinh(x)**

Returns the hyperbolic sine of  $x$ , e.g.,  $\text{Sinh}(1)=1.1752$

**Tanh(x)**

Returns the hyperbolic tangent of  $x$ , e.g.,  $\text{Tanh}(1)=0.761594$

**7.23.7 Logical/Conditional functions****And(b1,b2,...)**

Returns the logical conjunction of the arguments, which are all interpreted as Boolean expressions. If any of the expressions evaluates to a zero, `And` returns 0. For example,  $\text{And}(1=1, 2, 3)=1$ ,  $\text{And}(1=2, 2, 3)=0$

**Choose(index,v0,v1,...,vn)**

Returns  $v_i$  if *index* is equal to  $i$ . When *index* evaluates to a value smaller than 0 or larger than  $n-1$ , the function returns 0. Examples:

$\text{Choose}(0, 1, 2, 3, 4, 5)=1$

$\text{Choose}(4, 1, 2, 3, 4, 5)=5$

$\text{Choose}(7, 1, 2, 3, 4, 5)=0$

**If(cond,tval,fval)**

If *cond* evaluates to non-zero return *tval*, *fval* otherwise, e.g.,  $\text{If}(1=2, 3, 4)=4$ ,  $\text{If}(1, 5, 10)=5$

**Max(x1,x2,...)**

Returns the largest of the arguments  $x_i$ . Examples:

$\text{Max}(1, 2, 3, 4, 5)=5$

$\text{Max}(-3, 2, 0, 2, 1)=2$

**Min(x1,x2,...)**

Returns the smallest of the arguments  $x_i$ . Examples:

$\text{Min}(1, 2, 3, 4, 5)=1$

$\text{Min}(-3, 2, 0, 2, 1)=-3$

**Or(b1,b2,...)**

Returns the logical disjunction of the arguments, which are all interpreted as Boolean expressions. If any of the expressions evaluates to a non-zero value, Or returns 1. For example,  $\text{Or}(1=1, 0, 0)=1$ ,  $\text{Or}(1=0, 0, 0)=0$

### **Switch(x,a1,b1,a2,b2,...,[def])**

If  $x=a1$ , return  $b1$ , if  $x=a2$ , return  $b2$ , when  $x$  is not equal to any of  $as$ , return  $def$  (default value), which is an optional argument. When  $x$  is not equal to any of  $as$  and no  $def$  is defined, return 0.

Examples:

```
Switch(3,1,111,2,222,3,333,4,444,5,555,999)=333
Switch(8,1,111,2,222,3,333,4,444,5,555,999)=999
Switch(8,1,111,2,222,3,333,4,444,5,555)=0
```

### **Xor(b1,b2,...)**

Returns logical exclusive OR of all arguments, which are all interpreted as Boolean expressions. Xor returns a 1 if the number of logical expressions that evaluate to non-zero is odd and a 0 otherwise.

Examples:

```
Xor(0,1,2,-3,0)=1
Xor(1,1,0,2,2)=0
Xor(-5,1,1,-2,2)=1
```

## Acknowledgments

## 8 Acknowledgments

---

SMILE internally uses portions of the following two software libraries: micro-ECC and Expat. Both require an acknowledgment that we are reproducing below.

### **micro-ECC**

Copyright (c) 2014, Kenneth MacKay

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- \* Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

- \* Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

### **Expat**

Copyright (c) 1998-2000 Thai Open Source Software Center Ltd and Clark Cooper

Copyright (c) 2001-2017 Expat maintainers

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.