Dimple User Manual
Java Interface

Version 0.07

October 27, 2014

Contents

1 What is Dimple?

Dimple is an open-source API for probabilistic modeling and inference. Dimple allows the user to specify probabilistic models in the form of graphical models (factor graphs, Bayesian networks, or Markov networks), and performs inference on the model using a variety of supported algorithms.

Probabilistic graphical models unify a great number of models from machine learning, statistical text processing, vision, bioinformatics, and many other fields concerned with the analysis and understanding of noisy, incomplete, or inconsistent data. Graphical models reduce the complexity inherent in complex statistical models by dividing them into a series of logically (and statistically) independent components. By factoring the problem into sub-problems with known and simple interdependencies, and by adopting a common language to describe each sub-problem, one can considerably simplify the task of creating complex probabilistic models. A brief tutorial on graphical models can be found in Appendix A.

An important attribute of Dimple is that it allows the user to construct probabilistic models in a form that is largely independent of the algorithm used to perform inference on the model. This modular architecture benefits those who create probabilistic models by freeing them from the complexities of the inference algorithms, and it benefits those who develop new inference algorithms by allowing these algorithms to be implemented independently from any particular model or application.

Key features of Dimple:

- Supports both undirected and directed graphs.
- Supports a variety of *solvers* for performing inference, including sum-product and Gaussian belief propagation (BP), min-sum BP, particle BP, discrete junction tree, linear programming (LP), and Gibbs sampling.
- Supports both discrete and continuous variables.
- Supports arbitrary factor functions as well as a growing library of standard distributions and mathematical functions.
- Supports nested graphs.
- Supports rolled-up graphs (repeated HMM-like structures).
- Growing support for parameter estimation (including the EM algorithm).
- Supports both MATLAB[1] and Java[2] API.

[1]MATLAB is a registered trademark of The Mathworks, Inc.
[2]Java is a registered trademark of Oracle and/or its affiliates.

2 Installing Dimple

2.1 Installing Binaries

Users can follow these instructions to install Dimple.

1. Requires Java 7.
2. Download the latest version of Dimple from http://dimple.probprog.org. The latest binaries can be found at http://dimple.probprog.org/download listed as "Dimple Binaries".
3. Extract the Dimple zip file
4. If you are developing using Eclipse
   (a) If you don't already have a Java project, create one.
   (b) Open Project->Properties.
   (c) Select Java Build Path
   (d) Select Libraries
   (e) Select "Add External JARs..."
   (f) browse to <dimple directory>/solvers/lib, select all the jars and click open.
   (g) browse to <dimple directory>/solvers/non-maven-jars, select all the jars and click open.
   (h) You should now be able to instantiate and use Dimple classes in your project.
5. If you are compiling from the command line, add the following directories to the Java classpath:
   - <dimple directory>/solvers/lib

2.2 Installing from Source

Developers interested in investigating Dimple source code, helping with bug fixes, or contributing to the source code can install Dimple from source. Developers only interested in using Dimple should install from binaries (described in the previous section).

1. Download the source from https://github.com/AnalogDevicesLyricLabs/dimple
2. Install Gradle from http://www.gradle.org/. (Gradle is a Java build tool that pulls down jars from Maven repositories.)
3. Change to root directory
4. Run gradle by typing "gradle"

If you want to edit java files with Eclipse:

1. From eclipse, Import->Existing Projects Into Workspace
2. Browse to the dimple directory, select sovers/java, and click Finish.

3 Getting Started: Basic Examples

The following sections demonstrate Dimple with four basic examples. The first example is a simple hidden Markov model. The second models a 4-bit XOR constraint. The third demonstrates how to use nested graphs. The final example is a simple LDPC code.

All java examples referenced in this section can be found in /solvers/java/src/main/java/-com/analog/lyric/dimple/examples if Dimple has been installed from source.

3.1 A Hidden Markov Model

We consider a very simple hidden Markov model (HMM), the Rainy/Sunny HMM illustrated in the Wikipedia article about HMMs. Two friends who live far apart, Alice and Bob, have a daily phone conversation during which Bob mentions what he has been doing during the day. Alice knows that Bob's activity on a given day depends solely on the weather on that day, and knows some general trends about the evolution of weather in Bob's area.

Alice believes she can model the weather in Bob's area as a Markov chain with two states 'Sunny' and 'Rainy'. She remembers hearing that on the first day of last week it was quite likely (70% chance) that it was sunny in Bob's town. She also knows that a sunny day follows another sunny day with 80% chance, while a sunny day succeeds a rainy day with 50% chance.

She knows Bob pretty well, and knows that Bob only ever does one of three things: stay inside to read a book, go for a walk, or cook. She knows that if it is sunny, Bob will go for a walk with 70% probability, cook with 20% probability, and stay inside with 10% probability. Conversely, if it is rainy, Bob will go for a walk with 20% probability, cook with 40% probability, and stay inside with 40% probability.

Bob told Alice that he went for a walk on Monday, Tuesday, and Thursday, cooked on Wednesday and Friday, and read a book on Saturday and Sunday.

Alice wants to know what the most likely weather is for every day of last week. The above naturally defines an HMM which can easily be modeled within Dimple

Creating the Factor Graph

The first thing to do in many Dimple programs, is to create a factor graph. This is easily done by instantiating a FactorGraph.

See solvers/java/src/main/java/com/analog/lyric/dimple/examples/HMM.java.

```
FactorGraph HMM = new FactorGraph();
```

Declaring Variables

Once a Factor Graph has been defined, we can define the variables of the factor graph. In our case, there will be seven variables, MondayWeather to SundayWeather. The syntax to create a variable is new Discrete(domain,dimensions). In our case, the domain will consist of the two distinct values: Sunny and Rainy.

```
DiscreteDomain domain = DiscreteDomain.create("sunny", "rainy");
Discrete MondayWeather = new Discrete(domain);
Discrete TuesdayWeather = new Discrete(domain);
Discrete WednesdayWeather = new Discrete(domain);
Discrete ThursdayWeather = new Discrete(domain);
Discrete FridayWeather = new Discrete(domain);
Discrete SaturdayWeather = new Discrete(domain);
Discrete SundayWeather = new Discrete(domain);
```

Adding Factors

We now add the different factors of the factor graph. We will first add the factors corresponding to the Markov Chain structure. This is done with addFactor, which is a method of the factor graph previously defined.

The method has syntax addFactor(factorFunction,arguments), where factorFunction is an instance of a class that inherits from FactorFunction. The arguments are the variables involved in the factor being defined (in the same order as the inputs of the real function of the Factor Function). The number of inputs of the function referred to by the function handle has to be equal to the number of arguments of addFactor.

In our case, we define a class called TransitionFactorFunction. Note that the evalEnergy method returns the negative log of the factor value, rather than the value itself[3].

```
  public static class TransitionFactorFunction extends FactorFunction
  {
    @Override
    public final double evalEnergy(Value[] args)
    {
      String state1 = (String)args[0].getObject();
      String state2 = (String)args[1].getObject();
      double value;

      if (state1.equals("sunny"))
      {
        if (state2.equals("sunny"))
        {
          value = 0.8;
        }
        else
        {
          value = 0.2;
        }
      }
      else
      {
        value = 0.5;
      }
      return -Math.log(value);
    }
  }
```

[3]The negative log of a probability value can be interpreted as energy, borrowing the terminology from physics.

We can now add the factor to the factor graphs by using the addFactor method:

```
TransitionFactorFunction trans = new TransitionFactorFunction();
HMM.addFactor(trans, MondayWeather,TuesdayWeather);
HMM.addFactor(trans, TuesdayWeather,WednesdayWeather);
HMM.addFactor(trans, WednesdayWeather,ThursdayWeather);
HMM.addFactor(trans, ThursdayWeather,FridayWeather);
HMM.addFactor(trans, FridayWeather,SaturdayWeather);
HMM.addFactor(trans, SaturdayWeather,SundayWeather);
```

The java code first instantiates the factor function before connecting all of the variables. It would be possible to create a new instance of the factor function for each addFactor call, but this would use unnecessary memory.

We now need to add factors corresponding to the observations of each day. As it happens, when using an addFactor method, the arguments need not be all random variables—some can be declared as constants. We see now how to use this fact to easily add the observations to each day. We first declare an observation function

see the ObservationFactorFunction from solvers/java/src/main/java/com/analog/lyric/dimple/examples/HMM.java.

```
  public static class ObservationFactorFunction extends FactorFunction
  {
    @Override
    public final double evalEnergy(Value[] args)
    {
      String state = (String)args[0].getObject();
      String observation = (String)args[1].getObject();
      double value;

      if (state.equals("sunny"))
      {
        if (observation.equals("walk"))
          value = 0.7;
        else if (observation.equals("book"))
          value = 0.1;
        else // cook
          value = 0.2;
      }
      else
      {
        if (observation.equals("walk"))
          value = 0.2;
        else if (observation.equals("book"))
          value = 0.4;
        else // cook
          value = 0.4;
      }

      return -Math.log(value);
    }
  }
```

Adding the observations is then very easy:

```
ObservationFactorFunction obs = new ObservationFactorFunction();
HMM.addFactor(obs,MondayWeather,"walk");
HMM.addFactor(obs,TuesdayWeather,"walk");
HMM.addFactor(obs,WednesdayWeather,"cook");
HMM.addFactor(obs,ThursdayWeather,"walk");
HMM.addFactor(obs,FridayWeather,"cook");
HMM.addFactor(obs,SaturdayWeather,"book");
HMM.addFactor(obs,SundayWeather,"book");
```

As we can see, though the function itself depends on two variables, each factor only depends on one random variable (the other argument being set as a constant during the addFactor call). This in effect creates a factor connected only to one variable of the factor graph.

Specifying Inputs

The last step consists in adding the prior for the MondayWeather variable. We could, as above, use a factor with a single variable. Let us introduce a new property to easily add a single variable factor— the input property (on variables). When referring to ”properties” in java, this implies there is a setter and getter so, in the case of inputs setInput and getInput.

For a vector of probabilities (i.e., nonnegative numbers which sums up to one), Variable.setInput adds a single factor which depends on this variable only, with values equal to those given by the vector.

In our case, we do:

```
MondayWeather.setInput(0.7,0.3);
```

The Input property can be used in several different ways. Some notes of interest regarding the Input property:

- The Input method can typically be used for prior probabilities of the root variables in a Bayes Net, or for the initial node of a Markov Chain or HMM.
- The Input property can also be used for any factors with only one variable, for instance, for observation factors (see the introduction to factor graphs on how to remove observations and turn them into single node factors).
- The ObservationFactorFunction in the above example was not entirely required (see below)—we could have used the input property instead.
- IMPORTANT: Unlike the addFactor method, the Input property can be used only once for each variable. That is, once you have specified an input for a variable, re-specifying this input will destroy the previous factor and create a new one. In the

example above, using only the Input property, it would not have been possible to separately incorporate both the prior on Monday's weather and the factor corresponding to Mondays observation. However, this feature is very useful when Input is used to specify external information, and when the user wants to see the effect of external information. Say for instance that Bob mentions to Alice that it rained on Wednesday. Alice can simply use the call WednesdayWeather.setInput(0,1). If Bob corrects himself and says he misremembered, and that it actually was sunny that day, Alice can correct the information using again the call WednesdayWeather.setInput(1,0).

Solving the Factor Graph

Finally, we explain how to solve the factor graph by using the solve, iterate, and NumIterations factor graph methods.

The typical way of solving a factor graph will be by choosing the number of iterations and then running the solver for the desired number of iterations. In our case, this is simply done by typing the following code:

```
HMM.setOption(BPOptions.iterations, 20);
HMM.solve();
```

IMPORTANT:

By default, the solver will use either a Flooding Schedule or a Tree Schedule depending on whether the factor-graph contains cycles. A loopy graph (one containing at least one cycle) will use a Flooding Schedule by default. This schedule can be described as:

- Compute all variable nodes
- Compute all factor nodes
- Repeat for a specified number of iterations

If the factor-graph is a tree (one that contains no cycles), the solver will automatically detect this and use a Tree Schedule by default. In this schedule, each node is updated in an order that will result in the correct beliefs being computed after just one iteration. In this case, NumIterations should be 1, which is its default value.

Accessing Results

Once the solver has finished running, we can access the marginal distribution of each variable by using the Belief property:

```
double [] belief = TuesdayWeather.getBelief();
```

This returns a vector of probability of the same length as the domain total size (i.e., the product of its dimensions), with the probability of each domain variable. Another way to solve the factor graph is to use the Solver.iterate(n) method, which runs the factor graph for n iterations (without arguments, it runs for 1 iteration).

```
HMM.getSolver().iterate();
HMM.getSolver().iterate(5);
```

IMPORTANT: The iterate method is useful to access intermediate results (i.e, see how beliefs change through the iterations).

IMPORTANT: One distinction between the solve and iterate methods is that all messages and beliefs are reinitialized when starting the solve method. Running solve twice in a row is therefore identical to running it once, unlike iterate. When calling iterate() for the first time, first call initialize(), which reinitialize all messages.

3.2 A 4-bit XOR

The following example creates a simple factor graph with 4 variables tied through a 4-bit XOR, with ‘priors’ (we abuse language here and call ‘prior’ the probability distribution of each random variable if they were not tied through the 4-bit XOR).

Through this example, we will learn how to define arrays of random variables, see how to use MATLAB indexing within Dimple, see an example of a hard constraint in a factor graph, and see how to use the Bit type of random variable.

We consider a set of four random variables (B1,B2,B3,B4) taking values 0 or 1. The joint probability distribution is given by:

$$Pr(B_1, B_2, B_3, B_4) = \delta_{B_1+B_2+B_3+B_4=0(mod2)} P(B_1)P(B_2)P(B_3)P(B_4)$$

where the delta function is equal to 1 if the underlying constraint is satisfied, and 0 otherwise (this ensures that illegal assignment have probability zero). The $P(B_i)$ are single variable factors which help model which values of B_i are more likely to be taken (we call them ‘priors’, though, once again, this is an abuse of language. Typically, the factor will represent an observation of O_i of variable B_i, and the factor $P(B_i)$ is set equal to the normalized function $P(O_i|B_i)$ [4]

For our example, we will choose $P(B_1 = 1) = P(B_2 = 1) = P(B_3 = 1) = .8$ and $P(B_4 = 1) = 0.5$.

We will build our factor graph in two different ways. The first directly uses a 4-bit XOR, and uses mostly tools seen in the previous example, while the second introduces the Bit kind of random variable, and how to use an intermediate variable to reduce the degree of a factor graph with parity checks (i.e., XOR function).

The first way of building the factor graph uses an inefficient N-bit XOR function defined as follows

See solvers/java/src/main/java/com/analog/lyric/dimple/examples/BadNBitXor.java. Note that the evalEnergy method returns the negative log of the factor value, rather than the value itself. In this case, positive infinity corresponds to zero probability.

```
public static class BadNBitXorFactor extends FactorFunction
{
  @Override
  public final double evalEnergy(Value[] args)
  {
    int sum = 0;
    for (int i = 0; i < args.length; i++)
      sum += args[i].getInt();

```

[4] Normalizing $P(O_i|B_i)$ happens to be equal to $P(B_i|O_i)$ in a factor graph with only the two variables O_i and B_i with a prior on both values of B_i being equally likely.

```
        return (sum % 2) == 0 ? 0 : Double.POSITIVE_INFINITY;
      }
    }
```

Using everything we have learned in the previous example, the sequence of instructions we use is simply:

```
FactorGraph fourBitXor = new FactorGraph();
DiscreteDomain domain = DiscreteDomain.bit();
Discrete b1 = new Discrete(domain);
Discrete b2 = new Discrete(domain);
Discrete b3 = new Discrete(domain);
Discrete b4 = new Discrete(domain);
fourBitXor.addFactor(new BadNBitXorFactor(),b1,b2,b3,b4);
b1.setInput(0.2,0.8);
b2.setInput(0.2,0.8);
b3.setInput(0.2,0.8);
b4.setInput(0.5,0.5);
fourBitXor.solve();
System.out.println(b4.getValue());
System.out.println(Arrays.toString(b4.getBelief()));
```

IMPORTANT: We also introduce the Discrete method 'Value', which returns the most likely assignment of that random variable.

Often, we will find it useful to have random Bits. For that purpose, one can directly create random Bits with the Bit class. A Bit is simply a Discrete with Domain [0, 1]. Also, in order to simplify Input declarations, for a Bit variable named BitVar, BitVar.Input takes a single number, the probability of 1.

The second version of the 4-bit XOR will uses Bit variables to represent our random variables. Furthermore, it will decompose the 4 bits XOR into two 3-bits XOR. It is indeed easy to prove that $B_1 + B_2 + B_3 + B_4 = 0(mod2)$ is equivalent to

$$\begin{aligned} B_1 + B_2 + C &= 0(mod2) \\ B_3 + B_4 + C &= 0(mod2) \end{aligned}$$

for an internal bit C. While the reduction from one 4-bit to two 3-bit XORs might not seem tremendous, it is easy to see that more generally, any N-bit XOR can be reduced to a tree of 3-bit XORs, with depth log(N). Because the complexity of running Belief Propagation is exponential in the degree of factors, using this technique leads to dramatic speed improvements.

Using all the techniques mentioned above, we derive a new factor graph for the 4-bit XOR, equivalent to the one previously defined.

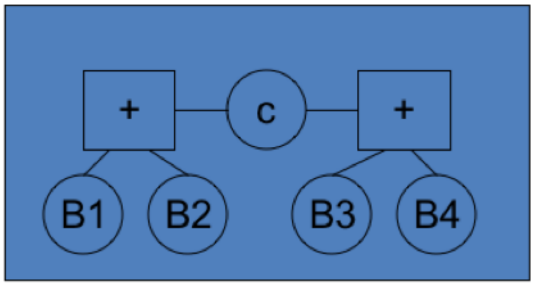
From solvers/java/src/main/java/com/analog/lyric/dimple/examples/ThreeBitXor.java

```
public class ThreeBitXor extends FactorFunction
{
  @Override
  public final double evalEnergy(Value[] args)
  {
    int arg0 = args[0].getInt();
    int arg1 = args[1].getInt();
    int arg2 = args[2].getInt();

    return (arg0 ^ arg1 ^ arg2) == 0 ? 0 : Double.POSITIVE_INFINITY;
  }
}
```

From solvers/java/src/main/java/com/analog/lyric/dimple/examples/FourBitXor.java

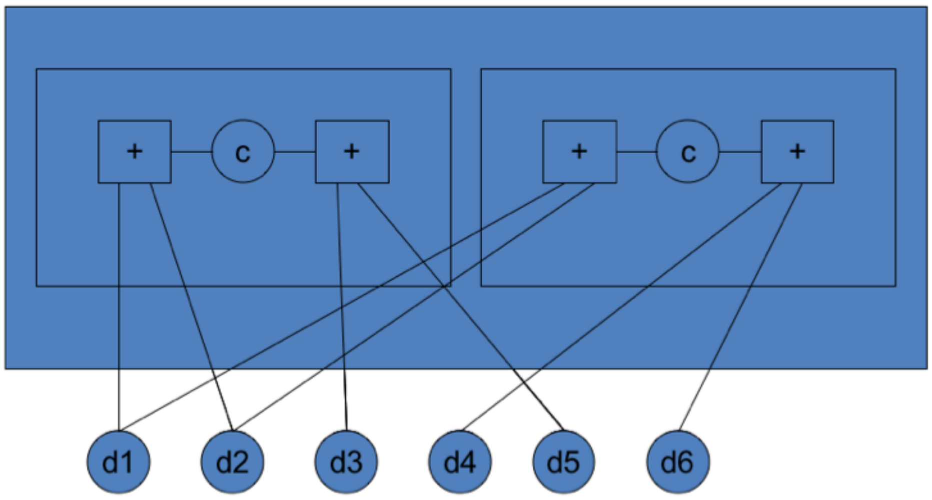
```
  FactorGraph xorGraph = new FactorGraph();
  Bit [] b = new Bit[4];
  for (int i = 0; i < b.length; i++)
    b[i] = new Bit();
  Bit c = new Bit();

  ThreeBitXor xd = new ThreeBitXor();

  xorGraph.addFactor(xd,b[0],b[1],c);
  xorGraph.addFactor(xd,b[2],b[3],c);

  double [] inputs = new double [] {.8, .8, .8, .5};
  for (int i = 0; i < inputs.length; i++)
    b[i].setInput(inputs[i]);

  xorGraph.solve();

  for (int i = 0; i < b.length; i++)
    System.out.println(b[i].getP1());
  for (int i = 0; i < b.length; i++)
    System.out.println(b[i].getValue());
```

The following figure represents the Factor Graph that is created and solved in this example.



3.3 Nested Graphs

Suppose we wish to use two 4-bit XORs from the previous example to create a 6-bit code. The following diagram shows our desired Factor Graph.



Dimple provides a way to replicate multiple copies of a Factor Graph and nest these instances in a containing Factor Graph. A nested Factor Graph can be seen as a special factor function between a set of variables (‘connector variables’), which, when ‘zoomed in,’ is in fact another factor graph, with factors involving both the connector variables, and other ‘internal variables.’ In the second version of the XOR example, we created a 4-bit XOR connected to 4 variables, by also creating an extra Bit C. What if we wanted to use that construction as a 4-bit XOR for potentially many different sets of 4 bits, overlapping or not, by replicating the factor graph as needed? Nested factor graphs provide an elegant solution to this problem.

A nestable factor graph is created by specifying first a set of ”connector” random variables, and instantiating a FactorGraph with these variables passed in to the constructor.

IMPORTANT: The factor graph defined this way is still a proper factor graph, and in principle, we can run BP on it. However, in practice, it is used as a ”helper” factor graph (and are ”helper” random variables), which will mostly be replicated in the actual factor graph of interest.

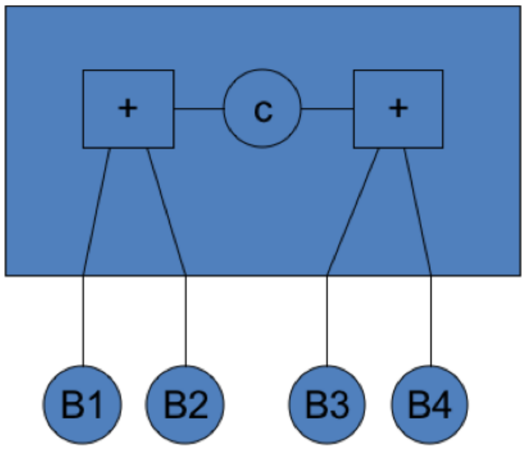
The following code creates a nestable factor graph, with connector variables (B_1, B_2, B_3, B_4)

See solvers/java/src/main/java/com/analog/lyric/dimple/examples/Nested.java

```
///////////////////////////////////////
// Define 4 bit xor from two 3 bit xors
///////////////////////////////////////
Bit [] b = new Bit[4];
for (int i = 0; i < b.length; i++)
  b[i] = new Bit();
```

```
FactorGraph xorGraph = new FactorGraph(b);

Bit c = new Bit();
ThreeBitXor xd = new ThreeBitXor();

xorGraph.addFactor(xd,b[0],b[1],c);
xorGraph.addFactor(xd,b[2],b[3],c);
```

IMPORTANT: In principle, variables attached to a Factor Graph can be defined before or after defining the factor graph (but obviously always before the factors they are connected to). However, connector variables naturally need to be defined before the nestable factor graph.



Consider a nestable factor graph NestableFactorGraph(connectvar1, connectvar2,..,vark) with k connector variables. Consider also an actual factor graph of interest, FactorGraph, containing (among others) k variables of interest (var1,..,vark). By using the addFactor method, we can replicate the NestableFactorGraph and use it to connect the variables (var1,..,vark) in the same way the connector variables are connected in the nestable factor graph: FactorGraph.addFactor(NestableFactorGraph,var1,..,vark).

In essence, the nestable factor graph represents a factor between the dummy connector variables, and the addFactor method is adding this factor to the desired actual variables.

IMPORTANT: Nested factor graphs support arbitrary levels of nesting. That is, a nested factor graph can be composed of nested factor graphs.

Armed with this tool, we can very simply use our custom 4-bit XOR to implement the factor graph described at the beginning of the section:

```
///////////////////////////////////////
// Create graph for 6 bit code
///////////////////////////////////////
Bit [] d = new Bit[6];
for (int i = 0; i < d.length; i++)
  d[i] = new Bit();
```

```
FactorGraph myGraph = new FactorGraph(d);
myGraph.addFactor(xorGraph,d[0],d[1],d[2],d[4]);
myGraph.addFactor(xorGraph,d[0],d[1],d[3],d[5]);
//////////////////////////////////////
// Set input and Solve
//////////////////////////////////////
double [] inputs = new double [] {.75, .6, .9, .1, .2, .9};
for (int i = 0; i < inputs.length; i++)
  d[i].setInput(inputs[i]);
myGraph.setOption(BPOptions.iterations, 20);
myGraph.solve();
for (int i = 0; i < d.length; i++)
  System.out.println(d[i].getValue());
```

4 How to Use Dimple

4.1 Defining Models

4.1.1 Overview of Graph Structures

While Dimple supports a variety of graphical model forms, including factor graphs, Bayesian networks, or Markov networks, the Dimple programming interface is oriented toward the language of factor graphs. Factor graphs are normally defined to be bipartite graphs—that is, graphs consisting of two types of nodes: *factor nodes* and *variable nodes*, connected by *edges*. Factor nodes connect only to variable nodes and vice versa. A mathematical overview of factor graphs can be found in Appendix A.

The model creation aspect of Dimple primarily involves defining variables and connecting these variables to factors. While dimple supports basic factor graphs, in which variables and factors are connected directly with no hierarchy, Dimple also supports more complex structures such as *nested* factor graphs, and *rolled-up* factor graphs, described below.

Basic factor graph: This is a graph in which all variables and factors are connected directly, with no hierarchy. Basic graphs are inherently finite in extent.

Nested factor graph: When a portion of a factor graph is used more than once, it may be convenient to describe that portion once, and then place copies of that sub-graph within a larger graph. Nested graphs can be used as a form of modularity that allows abstracting the details of a sub-graph from the description of the outer graph. A sub-graph can be thought of as a single factor in the outer graph that happens to be described as a factor graph itself. In Dimple, graphs may be nested to an arbitrary depth of hierarchy. Creation of a nested sub-graph within another graph is described in section 4.1.4.3.

Rolled-up factor graph: In some cases, a structure in a factor graph is repeated a great many times, or an unbounded or data-dependent number of times. In such cases, Dimple allows creation of rolled-up factor graphs, where only one copy of the repeated section is described in the model. The result is a factor graph that is implicitly unrolled when inference is performed. The amount of unrolling depends on a potentially unbounded set of data sources. Creation of rolled-up factor graphs is described in section 4.3.

4.1.2 Creating a Graph

To create a new factor graph, call the FactorGraph constructor and assign it to a variable. For example,

```
FactorGraph myGraph = new FactorGraph();
```

The created factor graph does not yet contain any variables or factors. Once a graph is created, then variables and factors can be added to it, as described in the subsequent sections.

Creating a factor graph in this way corresponds to a basic factor graph, or the top-level graph in a nested-graph hierarchy. To create a factor graph that will ultimately be used as a sub-graph in a nested graph hierarchy, the constructor must include arguments that refer to the "boundary variables" that will connect it to the outer graph. For example,

```
FactorGraph mySubGraph = new FactorGraph(varA, varB, varC);
```

In this case, the variables listed in the constructor, must already have been created. Creation of variables is described in section 4.1.3.

Creation of sub-graphs is described in more detail in section 4.1.4.3.

4.1.3 Creating Variables

4.1.3.1 Types of Variables

Dimple supports several types of variables:

Discrete: A Discrete variable is one that has a finite set of possible states. The domain of a discrete variable is a list of the possible states. These states may be anything—numbers, strings, arrays, objects, etc.

Bit: A Bit is a special-case of a discrete variable that has exactly two states: 0 and 1. Note that when using Bit variables, there are some differences in the API versus using Discrete variables. These differences are noted in the appropriate sections of this manual.

Real: A Real variable is a continuous variable defined on the real line, or some subset of the real line.

RealJoint: A RealJoint variable is a multidimensional continuous variable, defined on $\mathbb{R}^\mathbb{N}$ or a subset of $\mathbb{R}^\mathbb{N}$. Unlike a vector of Real variables, the components of a RealJoint variable cannot be connected independently in a graph.

Complex: A Complex variable is a special-case of a RealJoint variable with two dimensions.

FiniteFieldVariable: A FiniteFieldVariable is a special-case of a discrete variable that represents a finite field with $N = 2^n$ elements. These fields are often used, for example, in error correcting codes. These variables can be used along with certain custom factors that are implemented more efficiently for sum-product belief propagation than the alternative using discrete variables and factors implemented directly. See section 4.4 for more information on how these variables are used.

A factor graph may mix any combination of these variable types, though there are some limitations in the use of certain variable types by some solvers. Specifically, some solvers do not currently support Real variables.

A variable of a given type can be created using the corresponding object constructor. For example,

```
Discrete myDiscreteVariable = new Discrete(1,2,3,4,5,6,7,8,9,10);
Bit myBitVariable = new Bit();
Real myRealVariable = new Real();
```

In this case, each of the above lines created a single variable of the corresponding type. In the case of a Discrete variable, a constructor argument is required, which defines the domain of the variable. In the above example, the domain is the set of integer numbers 1 through 10.

Note that variables may be created prior to creation of the factor graph that they will ultimately become part of[5]. A variable is not yet associated with any factor graph until it is connected to at least one factor, as described in section 4.1.4.

4.1.3.2 Specifying Variable Domains

4.1.3.2.1 Discrete Variables

When creating a Discrete variable, the domain of that variable must be specified[6]. Once created, the domain of a variable cannot change.

The domain can be passed as a comma separated list, an object array, or a DiscreteDomain instance.

Some examples:

```
Object [] domain = new Object[10];
for (int i = 1; i <= domain.length; i++)
  domain[i-1] = i;
Discrete var = new Discrete(domain);
Discrete v2 = new Discrete(1.5, 1/3, 27.4327, -13.6, Math.sqrt(2), Math.sin(
    Math.PI/7));
Discrete v3 = new Discrete("Sun", "Clouds", "Rain", "Snow");
Discrete v4 = new Discrete(new double [][] {
            new double [] {1,2},
            new double [] {3,4}},
          new double [][] {
```

[5]For nested graphs, at least the boundary variables *must* be created prior to creation of the factor graph.

[6]Note that for Bit variables, the domain is implicit and is not specified in the constructor.

```
             new double [] {5,6,7},
             new double [] {8,9,10}},
           3.7);
Discrete v5 = new Discrete(DiscreteDomain.range(1,3));
```

WARNING: The following code:

```
Discrete v6 = new Discrete(new int [] {1,2,3});
```

Creates a discrete variable with a single domain object which is an array of three integers.

Even though the domains are discrete, the actual values of the domain states may be anything, including real numbers. Though, when using real numbers and objects, care must be taken, for example, when making equality comparisons.

The values of a domain need not be defined in-line in the constructor, but may be defined elsewhere in the program. For example:

```
Object [] domain = new Object[10];
...
for (int i = 1; i <= domain.length; i++)
  domain[i-1] = i;
```

Alternatively, the domain can be defined as a DiscreteDomain object, which provides an object wrapper around the domain. For example:

```
DiscreteDomain myDomain =  DiscreteDomain.range(1,3);
...
Discrete myVariable = new Discrete(myDomain);
```

In this case, the myDomain object has properties that can be queried, such as ".getElements()" , which provides a list of the elements of the domain.

To see the domain of a variable, you can query the Domain property. For a discrete variable, this returns a DiscreteDomain object, which you can query as described above to see the elements of the domain. For example:

```
System.out.println(Arrays.toString(myVariable.getDomain().getElements()));
```

The Domain property could be used, for example, to take one variable and create others that have the same domain. For example,

```
Discrete newVariable = new Discrete(otherVariable.getDomain());
```

4.1.3.2.2 Real Variables

When creating a Real variable, specifying a domain is optional. If no domain is specified, the domain is assumed to be the entire real line from $-\infty$ to ∞.

The domain of a Real variable is an array of two real numbers: the first specifies the lower bound on the variable, and the second specifies the upper bound (the upper bound must be greater than or equal to the lower). Either or both of the values may be $-\infty$ or ∞.

As with discrete variables, the domain may be specified ahead of time, and may be created by defining in this case a RealDomain object, which takes two arguments: the lower and upper bound, respectively. Some examples:

```
Real r1 = new Real();
Real r2 = new Real(0.0,Double.POSITIVE_INFINITY);
Real r3 = new Real(-Math.PI,Math.PI);
RealDomain myDomain = RealDomain.create(-2.6,12.4);
Real r4 = new Real(myDomain);
```

The Domain property of a Real variable returns in this case a RealDomain object, which contains two properties, getLowerBound() and getUpperBound(), which correspond to the lower bound and upper bound, respectively.

4.1.3.3 Naming Variables

Variables can be named, which can be useful in debugging as well as displaying a factor graph visually. A single variable can be named by setting the Name property:

```
myVariable.setName("My variable");
```

The variable name can be retrieved by referencing the Name property.

All variables in a factor graph must have unique names. Sometimes it's desirable to give multiple variables the same label for plotting or displaying. In this case, users can set the Label property. If the Label property is not set, plotting and displaying uses the name. If the Label property is set, the Label is used when displaying and plotting.

```
myVar.setLabel("myvar");
```

4.1.4 Creating Factors and Connections to Variables

4.1.4.1 Basic Factor Creation

Creation of a factor graph involves defining the variables of that graph, and then successively creating factors that connect to these variables.

Before creating a factor, the following must already have been created:

- The factor-graph into which the factor will be placed.
- All of the variables that the factor will connect to.

The most basic way of creating a factor is by calling the "addFactor" method on the factor graph object. For example:

```
myGraph.addFactor(factorSpecification, myVariable1, myVariable2);
```

Resulting from this method call, the specified factor is created, the factor is connected to the variables listed in the arguments, and the factor and all connected variables become part of the designated factor graph (if they are not already).

The factor specification is one or more arguments that define the factor. Dimple supports a variety of ways of specifying a factor, each of which is described in more detail in subsequent sections. A factor specification may be one of the following:

- A factor-table
- A built-in factor (or any instance of a class that extends FactorFunctions)
- The name of a built-in factor
- A sub-graph

After the factor specification argument(s), the remaining arguments are all treated as variables or constants.

The number and type of variables that can be connected to a factor depend on the type of factor being created. In some cases, a factor is defined flexibly to accommodate an arbitrary number of connected variables, while in other cases there are restrictions. In most cases, the order of variables matters. The definition of a factor generally requires a specific order of variables, where each variable or set of variables may be required to be of a specific type. In such cases, the arguments of the addFactor method call must appear in the required order.

For example, "Normal" is a built-in factor that describes a set of normally distributed variables with variable mean and precision. In this case, the first argument must be the mean, the second must be the precision, and the remaining one or more variables are the normally distributed variables. In this case, the factor creation could look like the following:

```
FactorGraph fg = new FactorGraph();
Real meanValue = new Real();
Real precisionValue = new Real(0.0,Double.POSITIVE_INFINITY);
Real [][] normalSamples = new Real[100][100];
for (int i = 0; i < 100; i++)
  for (int j = 0; j < 100; j++)
    normalSamples[i][j] = new Real();

Object [] vars = new Object[2 + 100*100];
vars[0] = meanValue;
vars[1] = precisionValue;
int index = 2;
for (int i = 0; i < 100; i++)
  for (int j = 0; j < 100; j++)
  {
    vars[index] = normalSamples[i][j];
    index++;
  }

fg.addFactor(new Normal(), vars);
```

In this example, the factor specification argument is an instance of the Normal factor function, and the normalSamples argument refers to the entire array of 100x100 normally distributed variables. The number of these sample variables is of arbitrary length only because of the way the “Normal” built-in factor was defined.

Note that these variables could have been listed explicitly as separate arguments. For example, if there were only two such variables, we could have written:

```
Real normalSample1 = new Real();
Real normalSample2 = new Real();
fg.addFactor(new Normal(), meanValue, precisionValue, normalSample1,
    normalSample2);
```

When creating a factor, it is sometimes convenient to supply a constant value to one or more of the factor’s arguments instead of a variable. This can be done simply by substituting a value that is not a Dimple variable. In this case, the value must be consistent with the possible values the particular factor is designed to accept. In the above example, if we wished to have a variable mean value, but define a fixed precision of 0.2, we could have written:

```
fg.addFactor(new Normal(), meanValue, 0.2, normalSample1, normalSample2);
```

Note that in this case, the “Normal” built-in factor requires the precision to be positive, so if we had provided a constant value of -0.2, for example, this would have resulted in an error. The particular requirements of a factor are specific to the definition of that factor.

4.1.4.2 Using Factor Tables

When Dimple creates a factor using a factor function, for some solvers, behind the scenes it translates that factor function into a table by enumerating all possible states of all connected variables. While steps are taken to make this efficient, including storing only non-zero values and reusing tables for identical factors, the time it takes to create the factor table can in some cases be very large. In some situations, a user may have specific knowledge of the structure of a factor that would allow them to create the same table much more efficiently. To accommodate such cases, Dimple allows factors to be specified using user-created factor tables.

A factor table consists of two parts: a two dimensional array of integers and a single dimensional array of doubles. Each row of the two dimensional table represents a combination of variable values for which the factor value is non-zero. Each column represents a variable connected to the factor. The values of this table specify an index into the discrete domain of a variable. Each row of the two dimensional table corresponds to one entry of the array of doubles, where that entry contains the value of the factor corresponding to the corresponding set of variable values.

Once the user has created the table, they can create a factor using this table in one of two ways. The first is to provide the two dimensional array of indices and vector of values directly as the first two arguments of the addFactor call, respectively:

```
FactorGraph fg = new FactorGraph();
Bit a = new Bit();
Bit b = new Bit();
int [][] indices = new int [][] {
  new int [] {0,0},
  new int [] {1,1}
};
double [] weights = new double [] {2.0,1.0};
FactorTable ft = FactorTable.create(indices, weights, a,b);
fg.addFactor(ft,a,b);
```

4.1.4.3 Using Sub-Graphs

In a nested graph, a factor at one layer of the graph hierarchy can correspond to an entire sub-graph. To add a sub-graph as a factor to another graph, first the sub-graph must have already been created. A sub-graph is created almost the same as any ordinary graph, with the exception of defining a subset of its variables to be “boundary” variables. These indicate how the sub-graph will connect to other variables in the outer graph.

To understand how sub-graph creation works, we first note that when a sub-graph is added to an outer graph, a new copy of the sub-graph is made, with entirely new variables and factors. The original sub-graph is used only as a template for creating the copies. This means that the actual variables used in the sub-graph are never directly used in the final nested graph. Internal variables within the sub-graph are created new when the sub-graph is added. Boundary variables, on the other hand, are connected to variables in the outer graph, which might already exist in that graph.

When a sub-graph is created, its boundary variables must be defined in the graph constructor. The boundary variables listed in the constructor must be of the identical type and have the identical domain (in the case of discrete variables) as the variables they will later connect when added to the outer graph. Additionally, the order of variables listed in creation of the sub-graph must match exactly the order of variables listed when adding the sub-graph to an outer graph.

For example, we define a subgraph as follows:

```
Discrete a = new Discrete(1,2,3);
Bit b = new Bit();
Bit x = new Bit();
FactorGraph mySubGraph = new FactorGraph(a, b);
mySubGraph.addFactor(exampleFactor1, a, b);
mySubGraph.addFactor(exampleFactor2, b, x);
```

To add this subgraph to an outer graph, we use addFactor , specifying the factor using the subgraph object.

```
int N = 5;
FactorGraph fg = new FactorGraph();
Discrete [] P = new Discrete[N];
Discrete [] Q = new Discrete[N];
for (int i = 0; i < P.length; i++)
{
  P[i] = new Discrete(1,2,3);
  Q[i] = new Bit();
}

for (int i = 0; i < N; i++)
    fg.addFactor(mySubGraph, P[i],Q[i]);
```

4.1.4.4 Using Built-In Factors

Dimple supports a set of built-in factors that can be specified when adding a factor to a graph. The complete list of available built-in factors can be found in section 5.9.

For example:

```
FactorFunction myFactorFunction = new Gamma(1,1);
MyGraph.addFactor(myFactorFunction, X1);
MyGraph.addFactor(myFactorFunction, X2);
```

4.1.4.5 Naming Factors

Just like variables, factors can be named, which can be useful in debugging as well as displaying a factor graph visually. To name a factor, the factor object must be accessible via a variable. When using addFactor, the result is the factor object, which can be assigned to a variable. A single factor can be named by setting the Name property of the factor object.

```
Factor myFactor = fg.addFactor(exampleFactor, a, b, c);
myFactor.setName("My factor");
```

And the factor name can be retrieved by referencing the getName() method.

Just like Variables, Factors also support the notion of a Label, which can be used to allow multiple factors to share the same label.

4.1.5 Modifying an Existing Graph

4.1.5.1 Removing a Factor

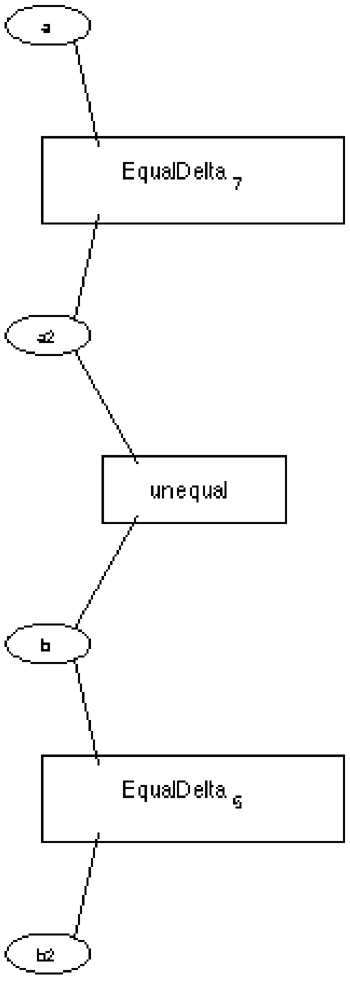
It is possible to remove a Factor from an existing FactorGraph:

```
FactorGraph fg = new FactorGraph();
 Bit [] b = new Bit[3];
 for (int i = 0; i < b.length; i++)
   b[i] = new Bit();
 Equals equals = new Equals();
 fg.addFactor(equals,b[0],b[1]);
 Factor f = fg.addFactor(equals,b[1],b[2]);
 double [] inputs = new double [] {0.8,0.8,0.6};
 for (int i = 0; i < inputs.length ; i++)
   b[i].setInput(inputs[i]);
 fg.setOption(BPOptions.iterations, 2);
 fg.solve();
 for (int i = 0; i < b.length; i++)
   System.out.println(b[i].getP1());
 fg.remove(f);
 fg.solve();
 for (int i = 0; i < b.length; i++)
   System.out.println(b[i].getP1());
```

4.1.5.2 Splitting Variables

It can be useful to make a copy of a variable and relate it to the old variable with an equals factor. The following code shows how to do this.

```
Bit a = new Bit();
Bit b = new Bit();

FactorGraph fg = new FactorGraph();
NotEquals ne = new NotEquals();
Factor f = fg.addFactor(ne,a,b);

Variable b2 = fg.split(b);
Variable a2 = fg.split(a,f);
```



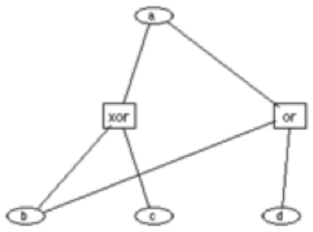
Note that the split method takes a list of factors as the second through nth argument. This is the list of factors that will be moved from the original variable to the copied variable. All unspecified factors will remain connected to the initial variable.

4.1.5.3 Joining Variables

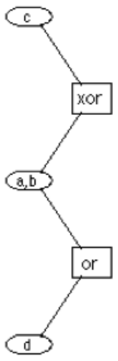
It is possible to join variables in an existing graph, which will create a new joint variable and modify all factors connected to the original variables to reconnect to the new joint variable. This can be useful in eliminating loops in a graph. The following code creates a loopy graph and then uses join to remove the loop.

```
Bit a = new Bit();
Bit b = new Bit();
Bit c = new Bit();
Bit d = new Bit();

FactorGraph fg = new FactorGraph();
ThreeBitXor xor = new ThreeBitXor();
XorYequalsZ xoryequalsz = new XorYequalsZ();
Factor f1 = fg.addFactor(xor,a,b,c);
Factor f2 = fg.addFactor(xoryequalsz ,a,b,d);
Variable newvar = fg.join(a,b);
```

The following is the loopy graph:



And after joining the variables we have:

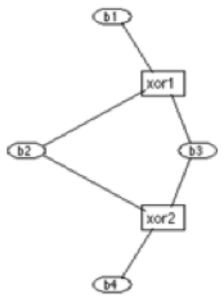


4.1.5.4 Joining Factors

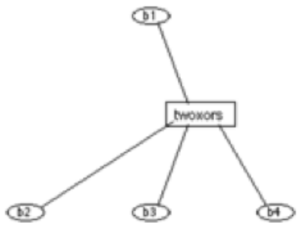
It is possible to remove loops by joining factors as well as by joining variables. (*NOTE: an easier way to eliminate loops is to use the Junction Tree solver (see 5.6.8), which will produce a transformed version of the graph without altering the original graph.*)

```
Bit [] b = new Bit[4];
for (int i = 0; i < b.length; i++)
     b[i] = new Bit();

FactorGraph fg = new FactorGraph();
ThreeBitXor xor = new ThreeBitXor();
Factor f1 = fg.addFactor(xor, b[0],b[1],b[2]);
Factor f2 = fg.addFactor(xor, b[1],b[2],b[3]);

Factor f3 = fg.join(f1,f2);
```

The following plot shows the graph with the loops:



And the following plot shows the graph after the factor is joined:



To join factors, Dimple does the following:

- Find the variables in common between two factors.
- Take the cartesian product of the tables but discard rows where the common variable indices differ.
- Consolidate the columns with common variables.
- Multiply the values for each row.

4.1.5.5 Changing Factor Tables

For factors connected only to discrete variables, the factors are stored in the form of a factor table (regardless of how the factor was originally specified). It is possible to modify some or all of the entries of that factor table in an existing factor graph.

```
int [][] indexList = new int [][] {
  new int[] {0, 0, 0},
  new int[][ {0, 1, 1},
  new int[]{ 1, 1, 0},
  new int[]{1, 0, 1}};
double [] weightList = new double []{0.2, 0.15, 0.4, 0.9};
factor.getFactorTable().change(indexList, weightList);
```

Here the indexList is is an array where each row represents a set of zero-based indices into the list of domain elements for each successive domain in the set of domains associated with the variables connected to this factor. The weightList is a one-dimensional array of real-valued entries in the factor table, where each entry corresponds to the indices given by the corresponding row of the indexList.

IMPORTANT: Identical factor tables are automatically shared between factors. If changing the factor table for a factor and if that factor is shared by other factors, then this changes it globally for all such factors.

4.1.6 Plotting a Graph

Plotting graphs is not currently supported in Java Dimple. (MATLAB Dimple does support plotting graphs).

4.1.7 Structuring Software for Model Portability

Because the specification of a model in Dimple is expressed in code, this code can be intermingled with other code that makes use of the model. However, it is recommended that code for defining a model be clearly separated from code that makes use of the model or performs other functions. Specifically, we recommend a structure whereby a graph (or subgraph) is encapsulated in a function that creates the graph and returns the graph object, optionally along with some or all of the variables in the graph. Depending on the application, the graph creation function may take some application-specific arguments. For example:

```
import com.analog.lyric.dimple.model.Bit;
import com.analog.lyric.dimple.model.FactorGraph;

public class MyGraphCreator {

  public static class Return
  {
    FactorGraph graph;
    Bit a;
    Bit b;
    Bit c;
  }

  public static Return createGraph()
  {
    Return r = new Return();

    r.graph = new FactorGraph();
    r.a = new Bit();
    r.b = new Bit();
    r.c = new Bit();
    r.graph.addFactor(new ThreeBitXor(),r.a,r.b,r.c);
    return r;
  }

  public static void main(String[] args)
  {
    Return r = createGraph();

    r.a.setInput(0.8);
    r.b.setInput(0.9);
    r.graph.solve();
    System.out.println(r.c.getP1());

  }

}
```

While it is possible to use Dimple to retrieve variables from the factor graph object itself, returning variables in the graph creation function allows them to be more easily managed and manipulated. The set of returned variables should include, at least, the variables that will subsequently be conditioned on input data and the variables that will be queried after performing inference. But since which variables will be used for these or other purposes may not be known ahead of time, it is often useful to simply return all variables. If more than one variable is to be returned, this could be done either by returning each as a separate return value, or combining them all into a single class, as shown in the above example.

In a nested graph, it is often preferable to use a structure like that shown above at each layer of nesting. In this way, a sub-graph creation function might be reused in more than one different outer graph.

In general, functions that create a Dimple model should not include operations that are related to performing inference. This includes choosing the solver, setting parameters that affect the solver behavior. There may in some cases be exceptions. For example, while conditioning variables on input data would normally not be considered part of the model, in some situations, it might be appropriate to consider this part of the model and to include it in the model creation code.

4.2 Performing Inference

Once a model has been created in Dimple, the user may then perform inference on that model. This typically involves first conditioning on input data, then performing the inference computation, and then querying the model to retrieve information such as beliefs (marginals), maximum *a posteriori* (MAP) value, or samples from the posterior distribution.

4.2.1 Choosing a Solver

To perform the inference computation, Dimple supports a variety of *solver* algorithms that the user can choose from. For each solver, there are a number of options and configuration parameters that the user may specify to customize the behavior of the solver.

At a high level, Dimple supports three categories of solver:

- Belief propagation (BP)
- Gibbs sampling
- Linear programming (LP)

The BP solvers further segment into solvers that are sum-product based—used primarily to compute marginals of individual variables in the model:

- SumProduct
- JunctionTree
- ParticleBP

and solvers that are min-sum based—used to compute the maximum *a posteriori* (MAP) value:

- MinSum
- JunctionTreeMAP

In either case, the result may be approximate, depending on the specific model and solver settings.

In the case of sum-product, the solvers further divide based on how continuous variables are dealt with (Real, Complex, and RealJoint variables). The SumProduct solver (which is the default solver) uses a Gaussian representation for messages passed to and from continuous variables, while the ParticleBP solver uses a particle representation[7]. In the current version of Dimple, the min-sum solvers support only discrete variables[8].

The two forms of Junction Tree solver are variants of BP that provide exact inference results by transforming a loopy graphical model into a non-loopy graph by joining factors and variables and then performing the sum-product or min-sum algorithm on the transformed model. This is only feasible for models that are "narrow" enough, i.e., ones in which a large number of variables would not have to be removed to eliminate any loops. The Junction Tree solvers currently support only discrete variables (and continuous variables that have been conditioned with fixed values). See section 5.6.8 for more details.

The Gibbs solver supports all variable types, and may be used to generate samples from the posterior distribution, or to determine marginals or approximate the maximum *a posteriori* value (see section 5.6.9).

The LP solver transforms a factor graph over discrete variables into an equivalent linear program then solves this linear program (see section 5.6.11). This solver is limited to factor graphs containing only discrete variables.

The Solver is a property of a factor graph. To set the Solver for a given graph, this property is set to the name of the solver. For example:

```
GibbsSolverGraph solverGraph = myGraph.setSolverFactory(new GibbsSolver());
```

Note that is often useful to save a reference to the created solver graph object returned by this method because it typically provides additional solver specific methods. The solver graph can also be obtained from the getSolver() method, but that requires an explicit downcast.

The currently supported set of solvers is:[9]

- SumProductSolver
- MinSumSolver
- JunctionTreeSolver
- JunctionTreeMapSolver
- ParticleBPSolver
- GibbsSolver

More detail on each of these solvers is provided in section 5.6.

If no solver is specified for a graph, Dimple will use the SumProduct solver by default.

[7]In the current version of Dimple, the ParticleBP solver does not support Complex or RealJoint variables.
[8]This restriction may be lifted in a future version.
[9]Dimple also supports and LP solver, but this solver is only accessible via the MATLAB API.

4.2.2 Conditioning on Input Data

In many cases, the model created in Dimple represents the prior, before conditioning on the data. In this case, then assuming inference on the posterior model is desired, then the user must condition the model on the data before performing inference.

There are two primary ways to condition on input data. In the first approach, the values actually measured are not included in the model, and instead the effect of the data is specified via a likelihood function for each variable that is indirectly influenced by the data. In the second approach, the variables that will be measured are included in the model, and the value of each is fixed to the actual measured data value.

4.2.2.1 Using a Likelihood Function as Input

Suppose a variable to be measured, y, depends only on another variable, x, and the conditional distribution $p(y|x)$ is known. Then conditioned on the measured value, $y = Y$, then the likelihood of x is given by $L(x) = p(y = Y|x)$. If our model includes the variable x, but does not include y, then we can indicate the effect of measuring $y = Y$ by specifying the likelihood function $L(x)$ as the "input" to the variable x using the Input property of the variable.

The particular form of the Input property depends on the type of variable. For a Discrete variable type, the Input property is a vector with length equal to the size of the variable domain. The values represent the (not necessarily normalized) value of the likelihood function for each element of the domain. For example,

```
Discrete v = new Discrete(1,2,3,4);
v.setInput(new double []{1.2, 0.6, 0, 0.8});
```

Notice that values in the Input vector may be greater than one—the Input is assumed to be arbitrarily scaled. All values, however, must be non-negative.

For a Bit variable, the Input property is specified differently. In this case, the Input is set to a scalar that represents a normalized version of the likelihood of the value 1. That is,

$$\frac{L(x = 1)}{L(x = 0) + L(x = 1)}$$

For example,

```
Bit b = new Bit();
b.setInput(0.3);
```

In this case, the value must be between 0 and 1, inclusive.

For a Real variable, the Input property is expressed in the form of a FactorFunction object that can connect to exactly one Real variable. The list of available built-in FactorFunctions is given in section 4.1.4.4. Typically, an Input would use one of the standard distributions included in this list. In this case, it must be one in which all the parameters can be fixed to pre-defined constants. For the Gibbs and ParticleBP solvers, any such factor function may be used as an Input. For the SumProduct solver, however, only a Normal factor function may be used. Below is an example of setting the Input for a Real variable:

```
Real r = new Real();
r.setInput(new Normal(measuredMean, measurementPrecision));
```

See section 5.2.6.2.3 for a description of how to set the Input on RealJoint (or Complex) variables.

4.2.2.2 Fixing a Variable to a Known Value

In some cases, the variable that will be measured is included in the model. In this case, once the value becomes known, the variable may be fixed to that specific value so that the remainder of the model becomes conditioned on that value. The FixedValue property is used to set a variable to a fixed value. For a single variable, the FixedValue is set to any value in the domain of that variable. For example:

```
v = Discrete(1:4);
v.FixedValue = 2;
```

```
v = Discrete([1.2, 5.6, 2.7, 6.94]);
v.FixedValue = 5.6;
```

```
b = Bit();
b.FixedValue = 0;
```

```
r = Real([-pi, pi]);
r.FixedValue = 1.7;
```

For Discrete variables, the FixedValue property is currently limited to variables with numeric domains, though the domains need not include only integer values[10].

```
Bit b = new Bit();
b.setFixedValue(1);
```

[10]This limitation may be removed in a future version.

To see if a FixedValue has been set on a variable, you can use the hasFixedValue method.

Because the Input and FixedValue properties serve similar purposes, setting one of these overrides any previous use of the other. Setting the Input property removes any fixed value and setting the FixedValue property removes any input[11].

4.2.2.3 Using a Data Source in a Rolled-Up Graph

In a rolled-up graph, the Input property of a variable can be set using a data source. Detail on how to do this can be found in section 4.3

4.2.3 Choosing a Schedule

All of the Dimple solvers operate by successively performing the inference computation on each element in the graph. In the case of BP solvers, both variable and factor nodes must be updated, and the performance of the inference can depend strongly on the order that these updates occur. Similarly, for the Gibbs solver, while variables must be updated in an order that maintains the requirements of valid Gibbs sampling, performance may depend on the particular order chosen.

The order of updates in Dimple is called a "schedule." The schedule may either be determined automatically using one of Dimple's built-in "schedulers," or the user may specify a custom schedule.

Each solver has a default scheduler, so if the user does not explicitly choose one, a reasonable choice is made automatically.

4.2.3.1 Built-in Schedulers

If no scheduler or custom schedule is specified, a default scheduler will be used. The default scheduler depends on the selected solver.

Another scheduler may be specified by setting the Scheduler property of a graph:

```
myGraph.setScheduler(new ExampleScheduler());
```

For the Junction Tree solvers, a tree schedule will always be used, so there is no need to choose a different scheduler. For the remaining BP solvers (SumProduct, MinSum, and ParticleBP), the following schedulers are available. More detail on each of these schedulers is provided in section 5.1.2.2.

[11]For implementation reasons, setting the fixed value of a Discrete or Bit variable also sets the Input to a delta function—with the value 0 except in the position corresponding to the fixed value that had been set.

- TreeOrFloodingScheduler
- TreeOrSequentialScheduler
- FloodingScheduler
- SequentialScheduler
- RandomWithoutReplacementScheduler
- RandomWithReplacementScheduler

In a nested graph, for most of the schedulers listed above (except for the random schedulers), the schedule is applied hierarchically. In particular, a subgraph is treated as a factor in the nesting level that it appears. When that subgraph is updated, the schedule for the corresponding subgraph is run in its entirety, updating all factors and variables contained within according to its specified schedule.

It is possible for subgraphs to be designated to use a schedule different from that of its parent graph. This can be done by specifying either a scheduler or a custom schedule for the subgraph prior to adding it to the parent graph. For example:

```
SubGraph.setScheduler(new SequentialScheduler());
ParentGraph.addFactor(SubGraph, boundaryVariables);
ParentGraph.setScheduler(new FloodingScheduler());
```

For the TreeOrFloodingScheduler and the TreeOrSequentialScheduler, the choice of schedule is done independently in the outer graph and in each subgraph. In case that a subgraph is a tree, the tree scheduler will be applied when updating that subgraph even if the parent graph is loopy. This structure can improve the performance of belief propagation by ensuring that the effect of variables at the boundary of the subgraph fully propagates to all other variables in the subgraph on each iteration.

For the RandomWithoutReplacementScheduler and RandomWithReplacementScheduler, if these are applied to a graph or subgraph, the hierarchy of any lower nesting layers is ignored. That is, the subgraphs below are essentially flattened prior to schedule creation, and any schedulers or custom schedules specified in lower layers of the hierarchy are ignored.

Because of the differences in operation between the Gibbs solver and the BP based solvers, the Gibbs solver supports a distinct set of schedulers. For the Gibbs solver, the following schedulers are available. More detail on each of these schedulers is provided in section 5.1.2.2.

- GibbsSequentialScanScheduler
- GibbsRandomScanScheduler

Because of the nature of the Gibbs solver, the nested structure of a graph is ignored in creating the schedule. That is, the graph hierarchy is essentially flattened prior to schedule creation, and only the scheduler specified on the outermost graph is applied.

4.2.3.2 Custom Schedules

Dimple supports user defined custom schedules created with a list of nodes and/or edges. A custom schedule is specified using the Schedule method. Specifying a custom schedule overrides any scheduler that the graph would otherwise use.

The following code demonstrates this feature:

```
Equals eq = new Equals();
FactorGraph fg = new FactorGraph();
Bit a = new Bit();
Bit b = new Bit();
Bit c = new Bit();
Factor eq1 = fg.addFactor(eq,a,b);
Factor eq2 = fg.addFactor(eq,b,c);

//define schedule
FixedSchedule fs = new FixedSchedule();
fs.add(b);
fs.add(eq1,a);
fs.add(eq2,c);
fs.add(a,eq1);
fs.add(c,eq2);
fs.add(eq1,b);
fs.add(eq2,b);

//set schedule
fg.setSchedule(fs);

//Set priors
a.setInput(0.6);
b.setInput(0.7);
c.setInput(0.8);

//Solve
fg.solve();
```

Dimple also supports nesting custom schedules and nesting in general. The following example demonstrates specifying nested graphs in a schedule.

```

Equals eq = new Equals();
Bit b1 = new Bit();
Bit b2 = new Bit();
FactorGraph nfg = new FactorGraph(b1,b2);
nfg.addFactor(eq,b1,b2);
Bit c1 = new Bit();
Bit c2 = new Bit();
Bit c3 = new Bit();
FactorGraph fg = new FactorGraph();
FactorGraph nf1 = fg.addFactor(nfg,c1,c2);
FactorGraph nf2 = fg.addFactor(nfg,c2,c3);
```

```
FixedSchedule schedule = new FixedSchedule();
schedule.add(c1,nf1,c2,nf2,c3);
c1.setInput(0.7);
fg.solve();
```

And finally we look at nesting a custom schedule:

```
//Now let's try nesting with a custom schedule on the nested graph.
Equals eq = new Equals();

//create a graph to nest and give it a funny schedule
// nfg: eb(1) - f1 - ib - f2 - eb(2)
Bit eb1 = new Bit();
Bit eb2 = new Bit();
Bit ib = new Bit();
FactorGraph nfg = new FactorGraph(eb1,eb2);
Factor f1 = nfg.addFactor(eq,eb1,ib);
Factor f2 = nfg.addFactor(eq,ib,eb2);
//Set an input and solve
eb1.setInput(0.8);

nfg.solve();

//We expect the output to be equal to the input since the tree
//scheduler passes the info along.
System.out.println(eb2.getP1() - eb1.getInput()[0]);

//Now we create a schedule that will not propagate the info.
FixedSchedule schedule = new FixedSchedule();
schedule.add(ib);
schedule.add(f1,eb1);
schedule.add(f2,eb2);
schedule.add(eb1);
schedule.add(eb2);
schedule.add(f1);
schedule.add(f2);
nfg.setSchedule(schedule);
nfg.solve();

System.out.println(eb2.getP1() - 0.5);


//Nest it and see if the schedule is preserved
Bit b1 = new Bit();
Bit b2 = new Bit();
FactorGraph fg = new FactorGraph();
FactorGraph g = fg.addFactor(nfg,b1,b2);

FixedSchedule schedule2 = new FixedSchedule();
schedule2.add(b1,b2,g);
fg.setSchedule(schedule2);

b1.setInput(0.8);
fg.solve();
System.out.println(b2.getP1()-0.5);
```

4.2.4 Running the Solver

Once a factor graph has been created and conditioned on any input data, inference may be performed on the graph by calling the solve method:

```
myGraph.solve();
```

The solve method performs all necessary computation, leaving the results available to be subsequently accessed. The behavior of the solve method is determined by the chosen schedule as well as by any solver-specific configuration parameters.

For example, for all of the BP solvers, the number of iterations can be set. By default, the number of iterations is 1, but for a loopy factor graph, generally multiple iterations should be performed. To set the number of iterations prior to solving, the BPOptions.iterations option may be set on the graph:

```
myGraph.setOption(BPOptions.iterations, 10);
myGraph.solve();
```

For the Gibbs solver, the BPOptions.iterations option isn't used and will be ignored if set; other options specific to this solver should be used instead. For example, to set the number of Gibbs samples to run before stopping (assuming the solver has been set to 'Gibbs'):

```
myGraph.setOption(GibbsOptions.numSamples, 1000);
myGraph.solve();
```

Note that in most cases solver options can be set directly on the factor graph and the values will only be used if the applicable solver is in use for that graph. Option values will usually not take effect until the solver objects have been initialized, but this is done automatically when the solve() method is run. If you need to interact directly with the solver representation of the factor graph, you can access it using the getSolver() method . Note that if the solver method is not part of the ISolverFactorGraph interface, you will need to cast the return value to the appropriate sovler class (e.g. GibbsSolverGraph) or you can simply save the return value from the setSolverFactory method:

```
GibbsSolverGraph mySGraph = myGraph.setSolverFactory(new GibbsSolver());
```

In general, each solver has a series of custom options and methods that can be used to configure the behavior of the solver and query its state. A complete list of these can be found in section 5.6.

In some cases, it is useful to observe the intermediate behavior of the solver before it has completed. For the BP solvers, this can be done by using the solver-specific iterate method instead of the solve method. When called without any arguments, this results in running one iteration. An optional argument allows specifying the number of iterations to run. Successive calls to iterate do not reset the state of the solver, allowing it to be called multiple times in succession. However, before running iterate for the first time, the initialize method must be called in order to reset the state before beginning. For example, here we run one iteration at a time, displaying the belief of a particular variable after each iteration:

```
myGraph.initialize();
for (int i=0; i <  numberOfIterations; i++)\
{
  myGraph.getSolver().iterate();
  System.out.println(Arrays.toString(someVariable.getBelief()));
}
```

If instead we wanted to run 5 iterations at a time, the iterate call would be replaced with:

```
myGraph.getSolver().iterate(5);
```

For the Gibbs solver, a similar method allows running one or a specified number of samples at a time, skipping initialization as well as any burn-in or random restarts. This is the sample method, which behaves the same as the iterate method.

4.2.4.1 Multithreading

Some solvers support multithreading. The following option can be used to turn on multithreading:

```
fg.setOption(SolverOptions.enableMultithreading, true);
```

By default, multithreading is turned off. Once multithreading is turned on, for large graphs or large factors, users can see acceleration up to N times where N is the number of cores in their machine.

4.2.5 Getting the Results of Inference

Once the solver has been run, the results of inference can be obtained from the elements of the graph. The kinds of results that may be desired vary with the application, and the kinds of results that are available depend on the particular solver and other factors.

One of the most common types of results are beliefs on individual variables in the graph. The belief of a variable is an estimate of the marginal distribution of that variable given the graph and any conditioning data.

When available, the belief is accessed via the getBeliefObject() method. If the variable is a Discrete variable, a double array can be retrieved using the getBelief() method.

```java
double [] d = myVariable.getBelief();
Object o = myVariable.getbeliefObject();
```

The particular form of the Belief property depends on the type of variable, and in some cases on the solver. For a Discrete variable type, the Belief property is a vector with length equal to the size of the variable domain. The values represent the normalized value of the approximate marginal probability of each element of the domain. For a Bit variable, the Belief property is a single number that represents the marginal probability of the value 1.

For Real variables when using the SumProduct solver, the Belief is represented the mean and precision parameters of a Normal distribution and for RealJoint variables they are represented as a mean vector and covariance matrix of a multivariate Normal distribution. Using the ParticleBP solver, beliefs are available for Real variables, but a different interface is used to obtain the beliefs in a useful form. This is summarized in section 5.6.10.4. Beliefs for Real variables are not currently supported in the Gibbs solver.

Another useful inference result returned by the Value property of a variable. This property returns a single value from the domain of the variable that corresponds to the maximum value of the belief:

```java
v = myVariable.getValue();
```

When using the Gibbs solver, there are additional inference results that may be useful. For example, for a given real variable, you can request the best sample value that has been seen during inference.

```java
double bestValue = ((GibbsReal)myVariable.getSolver()).getBestSample();
```

This is defined as the value of that variable associated with the sample over all variables that resulted in the most probably configuration observed given the graph and any conditioning data. Considering the graph as a potential function over the configuration space of all variables, this corresponds to the lowest energy configuration that had been observed.

By default, the Gibbs solver doesn't save all samples, but if so configured for a given variable (or all variables) prior to running the solver, the solver will save all samples, allowing the entire set of samples (post burn-in) to be obtained.

```java
myVariable.saveAllSamples();
myGraph.solve();
double [] allSamples = ((GibbsReal)myVariable.getSolver()).getAllSamples();
```

There are a number of other useful results that can be obtained from the Gibbs solver, which are detailed in section 5.6.9.

It is also possible to retrieve beliefs from factors. The belief of a factor is defined as the joint belief over all joint states of the variables connected to that factor. In the current version of Dimple, this works only for factors connected exclusively to discrete variables. The beliefs can be extracted using one of two properties of a factor:

```
double [] fb = ((DiscreteFactor)myFactor).getBelief();
```

Using the Belief property results in a compact representation of the belief that leaves out values corresponding to zero values of the factor.

4.2.6 Explicit Scheduling and Retrieving Message Values

Dimple supports the ability to retrieve and set messages as well as to explicitly update edges, factors and variables.

```
//OK, first we create a simple Factor Graph with a single xor connecting two
//variables.
Equals eq = new Equals();
FactorGraph fg = new FactorGraph();
Bit b1 = new Bit();
Bit b2 = new Bit();
Factor f = fg.addFactor(eq,b1,b2);
//We can go ahead and set some inputs
b1.setInput(0.8);
b1.setInput(0.7);

//we can examine some edges

System.out.println(f.getPorts().get(0).getInputMsg());
System.out.println(f.getPorts().get(0).getOutputMsg());

//we can even set some edge messages
f.getPorts().get(0).setInputMsgValues(new double [] {0.6,0.4});

//we can update a node
b1.update();
b2.update();

//or a specific edge
b1.updateEdge(f);

//but updating via portNum is quicker
b1.updateEdge(0);

//of course, if we don't know the portNum, we can get it
int portNum = b1.getPortNum(f);
b1.updateEdge(portNum);
```

```
//We can do the same kind of stuff with factors
f.updateEdge(b1);
f.updateEdge(f.getPortNum(b2));

//Let's look at some messages again
System.out.println(b1.getPorts().get(0).getInputMsg());
System.out.println(b2.getPorts().get(0).getInputMsg());

//and some beliefs
System.out.println(b1.getBelief());
```

4.3 Using Rolled Up Factor Graphs

4.3.1 Markov Model

In our first rolled up graph we build a simple Markov model describing an infinite stream of variables.

```
/////////////////////////////////
//Markov Model
/////////////////////////////////

/////////////////////////////////
//Build nested graph
/////////////////////////////////

//Here we build a graph that connects two variables by an xor
//equation An xor equation with only two variables is the
//equivalent of an equals constraint.
Equals eq = new Equals();
Bit in = new Bit();
Bit out = new Bit();
FactorGraph ng = new FactorGraph(in,out);
ng.addFactor(eq,in,out);


/////////////////////////////////
//create rolled up graph.
/////////////////////////////////

//Now we build a FactorGraph that creates an infinite chain of %variables.
BitStream bs = new BitStream();
FactorGraph fg = new FactorGraph();


//Passing variable streams as arguments to addFactor will result
//in a rolled up graph.  Passing in a slice of a variable stream
//specifies a relative offset for where the nested graph should
//be connected to the variable stream.
FactorGraphStream fs = fg.addRepeatedFactor(ng,bs,bs.getSlice(2));

/////////////////////////////////
//create data source
/////////////////////////////////
double [][] data = new double[10][];
for (int i = 0; i < 10; i++)
  data[i] = new double [] {0.4,0.6};

DoubleArrayDataSource ds = new DoubleArrayDataSource(data);
DoubleArrayDataSink dsink = new DoubleArrayDataSink();
bs.setDataSource(ds);
bs.setDataSink(dsink);

/////////////////////////////////
//solve
/////////////////////////////////
```

```
fg.solve();

/////////////////////////////////
//get Beliefs
/////////////////////////////////
while (dsink.hasNext())
  System.out.println(Arrays.toString(dsink.getNext()));
```

Let's talk about some of the aspects of rolled up graphs illustrated in this example in more detail:

4.3.1.1 Variable Streams and Slices

Variable Streams represent infinite streams of variables. When instantiating a VariableStream, users have to specify a domain. The following are examples of instantiating Discrete VariableStreams:

- new DiscreteStream(DiscreteDomain.range(0,2));
- DiscreteStream(0,1,2); - equivalent to the previous example
- new DiscreteStream(0,1); - A stream of bits
- new BitStream(); - Shorthand for the previous example.

When adding a repeated FactorGraph, users need to be able to specify how a nested graph connects to the variable streams. Slices can be used to indicate where in the stream a nested graph should connect. Slices are essentially references to a Variable Stream with a specified offset and increment. Slices have two main methods:

- hasNext() – Returns true if the next variable referenced by the slice can be retrieved.
- getNext() – Returns the next variable in the slice if it can be referenced. Otherwise throws an error.

Users typically shouldn't use these methods. Slices should primarily be used as arguments to addFactor.

First we need to instantiate a variable stream:

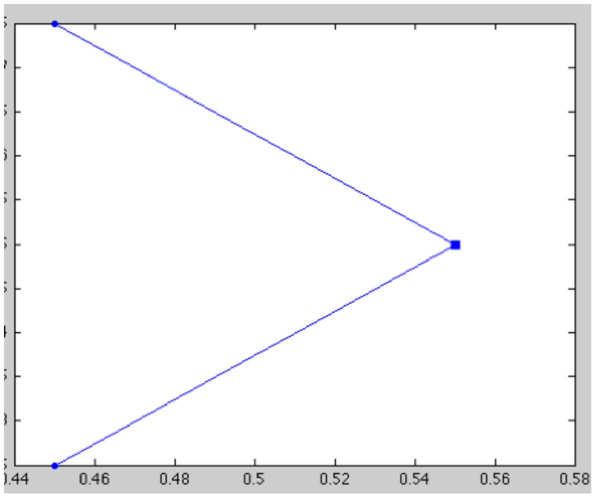
```
BitStream S = new BitStream();
```

We can get slices in the following way:

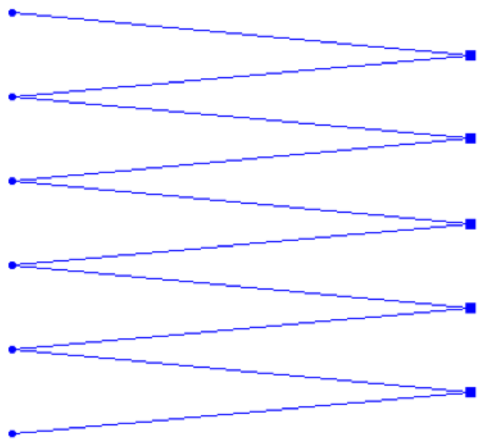
- S.getSlice(start); – Start specifies this slices starting point as an index into the variable stream.

4.3.1.2 Buffer Size

Users can specify a bufferSize for each FactorGraph stream. A FactorGraphStream is instantiated every time addFactor is called with a nestedGraph and VariableStreams or Slices. The default bufferSize is 1. Solving a graph with bufferSize one will result in a forward only algorithm. The bufferSize indicates how many nested graphs to instantiate for one step. In our Markov Model example, when buffer size is set to 1 and we plot the graph before solving we see this:



We see one factor and two instantiated variables. If we set bufferSize to 5 and plot we get:



We see five factors and 6 variables. After the first time we call 'advance' BlastFromThePast factors will be added to the oldest variable. These factors contain messages from the past. There are two ways to set the BufferSize for a FactorGraph stream:

- Fg.addFactor(ng,bufferSize,stream,slice,etc...); - Specified as second argument to addFactor.
- Fgs = fg.addFactor(ng,stream,slice,etc...); fgs.setBufferSize(bufferSize); - Set on the

FactorGraphStream directly.

4.3.1.3 DataSources

Users can create data sources and attach them to VariableStreams. As variables are created, data is retrieved from the DataSources and applied as inputs to the variables. If a VariableStream is never connected to a DataSource, hasNext() will always return true for that VariableStream. When a VariableStream is connected to a data source, hasNext() only returns true when there's more data in the DataSource.

DataSources implement the hasNext and getNext methods. Methods include:

- DoubleArrayDataSource(); – Constructor that creates an empty data source
- DoubleArrayDataSource(data); – Constructor that expects a two dimensional array as input. The data should be formatted such that each row provides input data for a step in the rolled up graph.
- add(data); – Users can add more data to the end of the data source. Data should have the following dimensions: NumSteps x SizeOfInputData.

Dimple also has support for MultivariateDataSources.

- MultivariateDataSource(); – Creates a data source object and assumes there is a single variable per step.
- add(means,covariance) – Means is the vector of means and covariance should contain a 2d array representing the covariance matrix.

4.3.1.4 DataSink

Users can retrieve their data using data sinks.

- DoubleArrayDataSink(); – Create a double array data sink
- MultivariateDataSink() – Created a multivariate data sink
- hasNext() – Is there more data in the data sink?
- getNext() – Retrieve the next chunk of data. For DoubleArrays, this returns data in the same form as is supplied to data sources. The MultivariateDataSink object returns MultivariateNormalParameters objects.
- varStream.setDataSink(dataSink); – Data sinks can be assigned to variable streams.

4.3.1.5 Accessing Variables

In the absence of data sinks, users need a way to retrieve variables to get beliefs. The following methods allow the user to do that:

```
BitStream Vs = new BitStream();
```

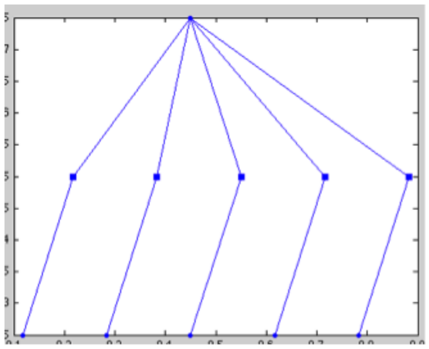
- Vs.setSize(size) – Number of variables in the buffer.
- Vs.get(index) – Retrieves a variable of the specified index. This is a 1-based value.

4.3.2 Markov Model with Parameter

When adding a repeated graph, it is possible to specify some variables as streams and others as individual variables. We sometimes call these individual variables parameters. Using this feature is straightforward:

```
Equals eq = new Equals();
FactorGraph ng = new FactorGraph(a,b);
ng.addFactor(eq,a,b);
Bit p = new Bit();
BitStream s = new BitStream();
FactorGraph fg = new FactorGraph();
FactorGraphStream fgs = fg.addRepeatedFactor(ng,p,s);
fgs.setBufferSize(5);
```

This code results in the following graph:



4.3.3 Real Variables

Rolled up graphs work with real variables as well. Here we create another Markov Model. We use the SumProduct solver and a built-in 'Product' factor. We create a data source that only has information about the first variable. The means beliefs are growing by 110% as we iterate through the stream because the factor provides a constraint that each variable is 110% of the previous variable.

```
/////////////////////////////////////////
//Build graph
/////////////////////////////////////////
Real a = new Real();
Real b = new Real();

FactorGraph ng =  new FactorGraph(a,b);
ng.addFactor("Product", b,a,1.1);

FactorGraph fg = new FactorGraph();
RealStream s = new RealStream();

fg.addRepeatedFactor(ng,s,s.getSlice(2));

/////////////////////////////////////////
//set data
/////////////////////////////////////////

double [][] data = new double[11][];
data[0] = new double [] {1,0.1};
for (int i = 1; i < 11; i++)
  data[i] = new double [] {0,Double.POSITIVE_INFINITY};

DoubleArrayDataSource dataSource = new DoubleArrayDataSource(data);

s.setDataSource(dataSource);
s.setDataSink(new DoubleArrayDataSink());


/////////////////////////////////////////
//Solve
/////////////////////////////////////////
fg.solve();

/////////////////////////////////////////
//get belief
/////////////////////////////////////////
DoubleArrayDataSink dds = (DoubleArrayDataSink)s.getDataSink();
while (dds.hasNext())
{
  double [] tmp = dds.getNext();
    System.out.println(Arrays.toString(tmp));
}
```

This produces the following output:

```
[1.0, 0.1]
[1.1, 0.11000000000000001]
[1.2100000000000002, 0.12100000000000002]
[1.3310000000000004, 0.13310000000000002]
[1.4641000000000006, 0.14641000000000004]
[1.6105100000000008, 0.16105100000000006]
[1.771561000000001, 0.17715610000000007]
[1.9487171000000014, 0.1948717100000001]
```

4.3.4 Manually Advancing

By default, rolled up graphs will advance until there is no data left in a DataSource. Users may override this behavior by either using FactorGraph.solveOneStep or setting the FactorGraph.NumSteps parameter. By default, NumSteps is set to Inf. By setting this to a finite number, N, users can examine the graph at every N steps of the rolled up graph. This allows the user to pull Beliefs off of any portion of the graph.

```java
//simple Markov Model with larger buffer size
Equals eq = new Equals();

//Create the data
int N = 10;
double [][] data = new double[N][];
for (int i = 0; i < N; i++)
  data[i] = new double [] {0.4,0.6};

//Create a data source
DoubleArrayDataSource dataSource = new DoubleArrayDataSource(data);

//Create a variable stream.
DiscreteStream vs = new DiscreteStream(0,1);
vs.setDataSource(dataSource);

//Create our nested graph
Bit in = new Bit();
Bit out = new Bit();
FactorGraph ng = new FactorGraph(in,out);
ng.addFactor(eq,in,out);

//Create our main factor graph
FactorGraph fg = new FactorGraph();

//Build the repeated graph
int bufferSize = 2;
FactorGraphStream fgs = fg.addRepeatedFactorWithBufferSize(ng,
    bufferSize,vs,vs.getSlice(2));


//Initialize our messages
fg.initialize();

while (true)
{
    //Solve the current time step
    fg.solveOneStep();

    //Get the belief for the first variable
    double [] belief = ((double[])vs.get(2).getBeliefObject());
        System.out.println(Arrays.toString(belief));

    if (fg.hasNext())
      fg.advance();
    else
      break;
}
```

In this code snippet, the user initializes the graph and calls advance until there is no data left. At each step, the user retrieves the beliefs from the third instance of the variable in the variable stream.

The user can also progress N steps:

```
//Initialize our messages
fg.initialize();
fg.setNumSteps(2);

while (true)
{
    //Solve the current time step
    fg.continueSolve(); //This method is need to avoid initialization

    //Get the belief for the first variable
    //Get the belief for the first variable
    double [] belief = ((double[])vs.get(2).getBeliefObject());
        System.out.println(Arrays.toString(belief));

    if (fg.hasNext())
      fg.advance();
    else
      break;
}
```

4.4 Using Finite Field Variables

4.4.1 Overview

Dimple supports a special variable type called a FiniteFieldVariable and a few custom factors for these variables. They represent finite fields with $N = 2^n$ elements. These fields find frequent use in error correcting codes. Because Dimple can describe any discrete distribution, it is possible to handle finite fields simply by describing their factor tables. However, the native FiniteFieldVariable type is much more efficient. In particular, variable addition and multiplication, which naively require $\mathcal{O}(N^3)$ operations, are calculated in only $\mathcal{O}(N \log N)$ operations.

4.4.2 Optimized Finite Field Operations

Rather than building finite field elements from scratch, a user can use a built-in variable type and associated set of function nodes. These native variables are much faster, both for programming and algorithmic reasons. All of these operations are supported with the SumProduct solver.

4.4.2.1 FiniteFieldVariables

Dimple supports a FiniteFieldVariable variable type, which takes a primitive polynomial (to be discussed later) and dimensions of the matrix as constructor arguments:

```
FiniteFieldVariable v = new FiniteFieldVariable(prim_poly);
```

This would create a finite field Variable with the given primitive polynomial.

4.4.2.2 Addition

Users can use the following syntax to create an addition factor node with three variables:

```
myFactorGraph.addFactor("FiniteFieldAdd",x,y,z);
```

The factor function must have the ”FiniteFieldAdd” name in order for the SumProduct solver to use the correct custom factor.

Adding this variable take $\mathcal{O}(1)$ time and solving takes $\mathcal{O}(N \log N)$ time, where N is the size of the finite field domain.

4.4.2.3 Multiplication

Similarly, the following syntax can be used to create a factor node with three variables for multiplication:

```
myFactorGraph.addFactor("FiniteFieldMult",x,y,z);
```

Under the hood this will create one of two custom factors, CustomFiniteFieldConstMult or CustomFiniteFieldMult. The former will be created if x or y is a constant and the latter will be created if neither is a constant. This allows Dimple to optimize belief propagation so that it runs in O(N) for multiplication by constants and O(Nlog(N)) in the more general case.

4.4.2.4 Projection

Elements of a finite field with base 2 can be represented as polynomials with binary coefficients. Polynomials with binary coefficients can be represented as strings of bits. For instance, $x^3 + x + 1$ could be represented in binary as 1011. Furthermore, that number can be represented by the (decimal) integer 11. When using finite fields for decoding, we are often taking bit strings and re-interpreting these as strings of finite field elements. We can use the FiniteFieldProjection built-in factor to relate n bits to a finite field variable with a domain of size 2^n.

The following code shows how to do that:

```
Object [] vars = new Object[n*2];

for (int j = 0; j < n; j++)
{
  vars[j*2] = j;
  vars[j*2+1] = new Bit();
}
myFactorGraph.addFactor("FiniteFieldProjection",vars);
```

4.4.3 Primitive Polynomials

See Wikipedia for a definition.

4.4.4 Algorithmics

Dimple interprets the domains as integers mapping to bit strings describing the coefficients of polynomials. Internally, the FiniteFieldVariable contains functions to map from this representation to a representation of powers of the primitive polynomial. This operation is known as the discrete log. Similarly Dimple FiniteFieldVariables provide a function to map the powers back to the original representation (i.e., an exponentiation operator).

- The addition code computes $x+y$ by performing a fast Hadamard transform of the distribution of both x and y, pointwise multiplying the transforms, and then performing an inverse fast Hadamard transform.
- The generic multiplication code computes xy by performing a fast Fourier transform on the distribution of the non-zero elements of the distribution, pointwise multiplying the transforms, performing an inverse fast Fourier transform, and then accounting for the zero elements.
- The constant multiplication code computes x by converting the distribution of the non-zero values of x to the discrete log domain (which corresponds to reshuffling the array), adding the discrete log of modulo $N-1$ (cyclically shifting the array), and exponentiating (unshuffling the array back to the original representation).

4.5 Parameter Learning

Dimple currently has two supported parameter learning algorithms: Expectation-Maximization on directed graphs and PseudoLikelihood Parameter Estimation on undirected graphs. Both of these algorithms are provided as early stage implementations and the APIs will likely change in the next version of Dimple.

4.5.1 PseudoLikelihood Parameter Estimation on Undirected Graphs

The PseudoLikelihood Parameter Estimation uses the following as its objective function:

$$\ell_{PL}(\theta) = \frac{1}{M}\sum_m \sum_i \sum_{a\sim i}(x_{-i}^{(m)}, x_i^{(m)}) - \frac{1}{M}\sum_m \sum_i logZ(x_{N(i)};\theta)$$

Currently it uses a very naive gradient descent optimizer. Future versions will likely have pluggable optimizers for each learning algorithm. (Likely including algorithms like BFGS).

4.5.1.1 Creating a parameter learner

The following creates a learner and initializes a few variables:

```
PseudoLikelihood pll = new PseudoLikelihood(fg, tables, vars2);
```

Arguments:

- factorGraph - the Factor Graph of interest
- factorTables - an array of factor tables for which to learn parameters.
- variables - an array of variable matrices (the order must match your data ordering).

4.5.1.2 Learning

The following method runs pseudo likelihood gradient descent. After it is run, the factor tables will contain the values of the learned parameters. For now the optimizer is simply a routine that multiplies the gradient by a scale factor and applies that change to the parameters. In the future, optimizers will be first class citizens and can be plugged into learners.

```
pll.learn(data, numSteps, scaleFactor);
```

Arguments:

- data - An MxN array where M is the number of samples and N is the number of variables. Variable data must be specified in the same order the variables were specified in the learner's constructor. For now, this data specifies the domain indices, not the domain values. This should be fixed in the future (so the user can do either). In reality, we'll probably split out training data into a more interesting data structure. (Same with the optimizer)
- numSteps - How many gradient descent steps should the optimizer run.
- scaleFactor - The value by which we multiply the gradient before adding to the current parameters. oldParams = oldParams + scaleFactor*gradient

4.5.1.3 Batch Mode

Users can divide their samples into subsets to run pseudo likelihood parameter learning in "batch" mode. Assuming users have their samples stored in a array of matrices, they could iterate over the array as follows:

```
for (int i = 0; i < samples.length; i++)
    pl.learn(samples[i],scaleFactor);
```

4.5.1.4 Setting Data

When calling the learn routine, users can set the data. However, if users want some visibility into the gradient or the numerical gradient, they must first set the data using the setData method

```
pl.setData(samples)
```

Arguments:

- samples - Takes the same form as in the learn method.

4.5.1.5 Calculating the Pseudo Likelihood

Users can retrieve the pseudo likelihood given the currently set samples using the following code:

```
likelihood = pl.calculatePseudoLikelihood();
```

Return value:

- likelihood - The log pseudo likelihood.

4.5.1.6 Calculating the Gradient

For debugging purposes, the user can retrieve the gradient given the current sample set and parameter settings.

```
result = pl.calculateGradient()
```

Return values:

- result - MxN matrix where M is the number of factor tables being learned and N is the number of weights per factor table.

4.5.1.7 Calculating the Numerical Gradient

For debugging purposes, the user can return a numerical gradient

```
pl.calculateNumericalGradient(table, weightIndex, delta)
```

Arguments:

- table - Which table to modify
- weightIndex - Which weight index to modify
- delta - the delta (in the log domain) of the parameter.

4.5.2 Expectation-Maximization on Directed Graphs

See the FactorGraph.baumWelch method in the API section. see section 5.1.3.8

4.6 Graph Libraries

Dimple provides a few graphs that are useful as nested graphs.

4.6.1 Multiplexer CPDs

Suppose you wanted a factor representing a DAG with the following probability distribution:

$$p(Y = y|a, z_1, z_2, ...) \propto \delta(y = z_a)$$

where all variables are Discrete.

You could code this up in Dimple as follows:

```java
public class Main
{
  public static class MyFunc extends FactorFunction
  {
    @Override
    public double evalEnergy(Value[] args)
    {
      Object y = args[0].getObject();
      int a = args[1].getInt();

      Object z = args[2+a].getObject();

      return y.equals(z) ? 0 : Double.POSITIVE_INFINITY;
    }
  }

  public static void main(String[] args)
  {
    Discrete y = new Discrete(1,2);
    Discrete a = new Discrete(0,1,2);
    Discrete z1 = new Discrete(1,2);
    Discrete z2 = new Discrete(1,2);
    Discrete z3 = new Discrete(1,2);

    FactorGraph fg = new FactorGraph();

    fg.addFactor(new MyFunc(),y,a,z1,z2,z3);
  }
}
```

However, to build this FactorTable takes $O(NM^{N+1})$ where N is the number of Zs and M is the domain size of the Zs. Runtime is almost as bad at $O(NM^N)$. However, there is an optimization that can result in $O(MN^2)$ runtime and graph building time. Dimple provides a MultiplexerCPD graph that can be used as a nested graph to achieve this optimization.

```
MultiplexerCPD cpd = new MultiplexerCPD(new Object [] {1,2},3);
Discrete Y = new Discrete(1,2);
Discrete A = new Discrete(0,1,2);
Discrete Z1 = new Discrete(1,2);
Discrete Z2 = new Discrete(1,2);
Discrete Z3 = new Discrete(1,2);

FactorGraph fg = new FactorGraph();
fg.addFactor(cpd,Y,A,Z1,Z2,Z3);
```

Dimple supports each Z having different domains. In this case, Y's domain must be the sorted union of all the Z domains

```
Object [][] domains = new Object [][] {
  new Object [] {1,2},
  new Object [] {1,2,3},
  new Object [] {2,4}
};

MultiplexerCPD cpd = new MultiplexerCPD(domains);
Discrete Y = new Discrete(1,2,3,4);
Discrete A = new Discrete(0,1,2);
Discrete Z1 = new Discrete(1,2);
Discrete Z2 = new Discrete(1,2,3);
Discrete Z3 = new Discrete(2,4);

FactorGraph fg = new FactorGraph();
fg.addFactor(cpd,Y,A,Z1,Z2,Z3);
```

Note that when using the SumProduct solver, a custom implementation of the built-in 'Multiplexer' factor function (see section 5.9) exists that is even more efficient than using the MultiplexerCPD graph library function. When using other solvers with discrete variables, the MultiplexerCPD graph library function should be more efficient. When the Z variables are real rather than discrete, the 'Multiplexer' factor function is the only option.

5 API Reference

The following section describes the classes, properties, options and methods that comprise the Dimple API for Java. Standard HTML-based Java API documentation is also available in the doc/javadoc/ subdirectory of the install kit and from the dimple.probprog.org website and will often provide more detailed information.

Note that in order to maintain compatibility with the MATLAB version of this document, the term ”property” is used to refer to elements of the API even though the Java language does not support the concept of implicit getter/setter method calls using field access syntax. When this document refers to a ”property” Foo with a given Type, it implies there is a setter and getter method of the form:

- getter - Type getFoo();
- setter - void setFoo(Type value);

5.1 FactorGraph

The FactorGraph class represents a single factor graph and contains a collection of all factors and variables associated with that factor graph.

5.1.1 Constructor

```
new FactorGraph(Variable ... boundaryVariables)
```

For a basic factor graph, the constructor can be called without arguments.

For a nested factor graph (one that may be used as a sub-graph within another graph), the constructor must include a list of the boundary variables of the graph. When used as a sub-graph, the boundary variables are dummy variables with the same specification as the variables in the outer graph that will ultimately connect to the sub-graph. A graph defined with boundary variables may alternatively be used as a top-level graph, in which case the boundary variables are used directly.

5.1.2 Properties

5.1.2.1 Solver

Read-write. Indicates the choice of solver to be used for performing inference on the graph. The default solver is SumProduct.

When setting the solver, the solver is given by an instance of IFactorGraphFactory specific to that kind of solver. By convention, these should have a name ending with the suffix Solver (e.g. MinSumSolver or GibbsSolver). The method setSolverFactory() sets the solver and returns a newly constructed ISolverFactorGraph instance specific to this graph and solver. The specific return type will depend on the solver.

```
SolverGraph sgraph = fg.setSolverFactory(new SolverClass());
```

The current set of valid solver classes are:[12]

- SumProductSolver
- MinSumSolver
- JunctionTreeSolver

[12]Dimple also supports and LP solver, but this solver is only accessible via the MATLAB API.

- JunctionTreeMAPSolver
- ParticleBPSolver
- GibbsSolver

A description of each of these solvers is given in section 5.6.

Note that the solver can be modified at any time. After running the solver on a graph, the solver may be modified and the new solver run using the same graph[13].

5.1.2.2 Scheduler

Read-write. Indicates the scheduler to be used for performing inference on the graph (unless a custom schedule is specified instead). A scheduler defines a rule that determines the update schedule of a factor graph when performing inference.

When setting the scheduler, the scheduler is given by a string representing the name of the scheduler. The scheduler name is case *sensitive.*

```
fg.setScheduler(new com.analog.lyric.dimple.schedulers.
    TreeOrSequentialScheduler());
```

Each scheduler is applicable only to a certain subset of solvers. The list of all available built-in schedulers and a description of their behavior can be found in section 5.5.

5.1.2.3 Schedule

Read-write. Specifies a custom schedule to be used for performing inference. Schedules must implement the ISchedule interface which provides an iterator over schedule entries. Users can instantiate a FixedSchedule object and add nodes and/or edges to the fixed schedule to provide an order of updates. Examples of using custom schedules are given in section 4.2.3.2.

For BP solvers, any of these entries may be included, and have the following interpretation[14].

FixedSchedule.add(variable) Update messages for all outgoing edges of that variable.

FixedSchedule.add(factor) Update messages for all outgoing edges of that factor.

FixedSchedule.add(variable,factor) Update a single outgoing edge of the variable in the direction connecting to the specified factor.

[13]In this case, care must be taken to set any solver-specific parameters to the new values after changing the solver.

[14]When using the JunctionTree or JunctionTreeMAP solvers, the specified Schedule is ignored.

FixedSchedule.add(factor,variable) Update a single outgoing edge of the factor in the direction connecting to the specified variable.

For BP solvers, a check is made when a custom schedule is set to ensure that all edges in the graph are updated at least once.

For the Gibbs solvers, the Schedule should include only variable entries. Any other entries will be ignored.

If a custom schedule is set on a factor graph (either an entire graph or a sub-graph), this schedule is used instead of any built-in scheduler that may have previously been set (or the default scheduler).

In a nested graph, the Schedule property at each nesting level may be set independently. For some built-in schedulers, the user may mix custom schedules at some nesting layers, while using built-in schedulers at others. The particular built-in schedulers that support such mixing are described in section 5.1.2.2.

5.1.2.4 NumIterations

Read-write. The NumIterations property sets the number of iterations BP will to run when using the solve method. This only applies to solvers that use BP, which are the SumProduct, MinSum, and ParticleBP solvers.

The default value is 1. For a factor graph with a tree-structure, when using the default scheduler, one iteration is appropriate. Otherwise, it would normally be appropriate to set the number of iterations to a larger value.

5.1.2.5 NumSteps

Read-write. This property is used for rolled-up graphs (see section 4.3). This property determines the number of steps over which to perform inference when using the solve or continueSolve methods (see sections 5.1.3.3 and 5.1.3.4). A *step* corresponds to a single run of the solver over the current portion of the rolled-up graph, followed by advancing the graph to the next position. By default, this property is infinite, resulting in no limit to the number of steps to be run (running until there is no more source data).

5.1.2.6 Name

Read-write. When read, retrieves the current name of the factor graph. When set, modifies the name of the factor graph to the corresponding value. The value set must be a string.

```
fg.setName('string');
```

5.1.2.7 Label

Read-write. All variables and factors in a Factor Graph must have unique names. However, sometimes it is desirable to have variables or factors share similar strings when being plotted or printed. Users can set the Label property to set the name for display. If the Label is not set, the Name will be used for display. Once the label is set, the label will be used for display.

5.1.2.8 Score

Read-only. When read, computes and returns the score (energy) of the graph given a specified value for each of the variables in the graph. The score represents the energy of the graph given the specified variable configuration, including all factors as well as all Inputs to variables (which behave as single-edge factors). The score value is relative, and may be arbitrarily normalized by an additive constant. The value of the score corresponds to the sum over factors and variables of their corresponding scores (see sections 5.3.1.1.4 and 5.2.2.1.6).

The value of each variable used when computing the Score is the Guess value for that variable (see section 5.2.2.1.5). If no Guess had yet been specified for a given variable, the value with the most likely belief (which corresponds to the Value property of the variable) is used[15].

For a rolled-up graph, the Score property represents only the score for only the portion of the graph in the current buffer.

5.1.2.9 BetheFreeEnergy

Read-only. (Only applies to the Sum Product Solver). When read returns:

$$BetheFreeEnergy = InternalEnergy - BetheEntropy$$

```
double bfe = fg.getBetheFreeEnergy();
```

[15]For some solvers, beliefs are not supported for all variable types; in such cases there is no default value, so a Guess must be specified.

5.1.2.10 Internal Energy

Read-only. (Only applies to the Sum Product Solver). When read returns:

$$InternalEnergy = \sum_{a \in F} InternalEnergy(a) + \sum_{i \in V} InternalEnergy(i)$$

Where F is the set of all Factors and V is the set of all variables. If Dimple treated inputs as single node Factors, this method would only sum over factors.

For a definition of a Factor's InternalEnergy, see sections 5.3.1.1.5. For a definition of a Variable's InternalEnergy, see section 5.2.2.1.7.

```
double ie = fg.getInternalEnergy();
```

5.1.2.11 Bethe Entropy

Read-only. (Only applies to the Sum Product Solver). When read returns:

$$BetheEntropy = \sum_{a \in F} BetheEntropy(a) - \sum_{i \in V} (d_i - 1) BetheEntropy(i)$$

Where F is the set of all Factors, V is the set of all variables, and d_i is the degree of variable i (i.e. the number of factors i is connected to).

For a definition of a Factor's BetheEntropy, see sections 5.3.1.1.6. For a definition of a Variable's InternalEnergy, see section 5.2.2.1.8.

```
be = fg.getBetheEntropy();
```

5.1.3 Methods

5.1.3.1 addFactor

```
MyGraph.addFactor(factorSpecification, variableList);
```

The addFactor method is used to add a factor to a factor-graph, connecting that factor to a specified set of variables. There are several ways of specifying the factor. The method takes a factorSpecification argument followed by a comma-separated list of variables or a single variable array.

The list of variables indicates which variables to connect to the factor. The order of the variables listed must correspond to the order of edges of the factor.

Some of the variables may be replaced by constants. In this case, no variable is created, but instead the specified constant value is used in place of a variable for the corresponding edge of the factor. The value of a constant must be a valid value given the definition of the corresponding factor.

The factorSpecification may be specified in one of a number of different ways. The following table lists the various ways the factorSpecification can be specified:

Factor Specification	Description
FactorFunction	A FactorFunction object as described in section 5.3.3. This form can be used to specify a built-in factor that requires constructor arguments. The list of Dimple built-in factors is given in section 5.9.
builtInFactorName	String indicating the name of a built-in factor. The list of Dimple built-in factors is given in section 5.9. Referring to a built-in factor by name assumes no constructor arguments are needed for the corresponding FactorFunction. Built-in factor names are case sensitive.
FactorTable	A FactorTable object as described in section 5.3.4. Specifying a factor in this form is supported only if all connected variables are discrete.
FactorGraph	A sub-graph to be nested within this graph. The number and order of the variables listed in the variableList must correspond to the number and order of the boundary variables declared when the sub-graph was created. When adding a nested graph within an outer graph, the specified sub-graph is used as a template to create a new factor graph that is actually added to the outer graph. Copies are made of all of the variables and factors in the specified sub-graph.

5.1.3.2 initialize

```
MyGraph.initialize();
```

The initialize method resets the state of the factor graph and its associated solver. When performing inference incrementally, for example using the iterate method, the initialize method must be called before the first iterate call. When using the solve method to perform inference, there is no need to call initialize first. The initialize method takes no arguments.

5.1.3.3 solve

```
MyGraph.solve();
```

The solve method runs the solver on the factor graph for the specified duration. Calling solve initializes the graph prior to solving.

For BP-based solvers, the solver runs the number of iterations specified by the NumIterations property. For the Gibbs solver, it runs for the specified number of samples (see section 5.6.9).

For rolled-up factor graphs, the solver runs the solver over multiple steps of the graph. A *step* corresponds to a single run of the solver over the current portion of the rolled-up graph, followed by advancing the graph to the next position. It performs the number of steps of inference specified by the NumSteps property or until there is no more data in a data source, whichever comes first.

5.1.3.4 continueSolve

This method is used for manually advancing a rolled-up graph (see section 4.3.4). This method takes no arguments and returns no value. When called, it performs the number of steps of inference specified by the NumSteps property or until there is no more data in a data source, whichever comes first. A *step* corresponds to a single run of the solver over the current portion of the rolled-up graph, followed by advancing the graph to the next position. The initialize method should be called prior to calling this method for the first time on an entire rolled-up graph, but should not be called before calling this method again to run additional steps.

5.1.3.5 solveOneStep

This method is used for manually advancing a rolled-up graph (see section 4.3.4). This method takes no arguments and returns no value. When called, it performs inference on the current portion of the rolled-up graph. Inference is performed on this section of the graph using whatever solver-specific parameters had previously been specified. The graph is not re-initialized prior to performing inference, starting instead from the final state resulting from inference on the previous graph position. The initialize method should be called prior to calling this method for the first time on an entire rolled-up graph, but should not be called after each advance to the next section.

5.1.3.6 advance

This method is used for manually advancing a rolled-up graph (see section 4.3.4). This method takes no arguments and returns no value. When called, it advances the graph to the next position. Advancing the graph involves getting the next value of all data sources, and writing the next available value to all data sinks.

5.1.3.7 hasNext

This method is used for manually advancing a rolled-up graph (see section 4.3.4). This method takes no arguments. It returns a boolean value indicating whether or not it is possible to advance the graph to the next position. This will be true only if all of the data sources have at least one more available value.

5.1.3.8 baumWelch

```
fg.baumWelch(factorList, numRestarts, numSteps);
```

The baumWelch method performs the Expectation-Maximization (EM) algorithm on a factor graph (the specific case of EM on an HMM graph is called the Baum-Welch algorithm, though EM in Dimple can be applied to any graph structure).

This method has the following limitations:

- The factors on which parameter estimation is being performed must be directed (see section 5.3.1.1.3).
- The factors on which parameter estimation is being performed must be connected only to discrete variables.
- The Solver must be set to use the SumProduct solver (the default solver).

The factorList argument is either a single Factor or FactorTable, or an array of Factors or FactorTables. The weight values in these factor tables are the parameters to be estimated. All other factors in the graph remain fixed and unmodified. When including a FactorTable, if that table is used in more than one factor in the graph, the entries in the table are tied, estimating them jointly for all factors containing that table.

The numRestarts argument indicates the number of times the EM process is repeated using distinct random restart values for the parameters. When numRestarts is greater than 1, the EM process is repeated with different random initialization, and the final resulting parameter values are chosen from the run that resulted in the smallest Bethe free energy for the graph.

The numSteps argument indicates (for each restart), how many steps of the EM algorithm are to be run. A single step of the EM algorithm is one run of belief propagation followed by re-estimation of the parameter values.

Note that the number of iterations for each run of belief propagation is determined from the NumIterations property. If the graph is a tree, the number of iterations should be 1 (the default value). If the graph is loopy, it should be set to a larger value (in this case, the EM algorithm is only approximate).

Upon completion of this method, the result appears in the factor tables that were listed in the factorList argument. That is, the factor table weights contain the resulting estimated values. To read these values, the Weights property of the factor table can be read (see section 5.3.4). For a factor, the factor table can be extracted using the FactorTable property (see section 5.3.4), and then the weights can be read from that.

5.1.3.9 join

The join method can be used to join a set of existing variables or a set of existing factors in a graph. In the current version of Dimple this method is supported only for discrete variables, and factors connected only to discrete variables.

When joining variables, the join method is called with a comma-separated list of variables to be joined.

```
fg.join(variableList);
```

The result is a new variable with a domain that is the Cartesian product of the domains of all of the variables being joined.

When joining factors, the join method is called with a comma-separated list of factors to be joined.

```
fg.join(factorList);
```

The result is a new factor with a factor table that corresponds to the product of the factors being joined. The new factor connects with the union of all variables that were previously connected to any of the joined variables.

5.1.3.10 split

The split method splits a variable in an existing graph into two variables connected by an Equality factor.

```
fg.split(variable, [factorList]);
```

The method takes an optional comma-separated list of factors. This list of factors identifies factors already connected to the variable that are to be moved to the new instance of the variable. All unspecified factors remain connected to the original instance.

5.1.3.11 removeFactor

```
fg.removeFactor(factor);
```

This method removes the specified factor from an existing factor graph that contains it. This also removes all edges that connect this factor to neighboring variables.

5.1.3.12 addBoundaryVariables

```
fg.addBoundaryVariables([factorList]);
```

This method takes a comma separated list of variables. The listed variables can then be used as boundary variables in a nested graph.

```
FactorGraph ng = new FactorGraph();
Bit a = new Bit();
Bit b = new Bit();

Equals eq = new Equals();

ng.addFactor(eq,a,b);

ng.addBoundaryVariables(a,b);

FactorGraph fg = new FactorGraph();
Bit a2 = new Bit();
Bit b2 = new Bit();
fg.addFactor(ng,a2,b2);
```

5.1.4 Introspection

The FactorGraph class provides several feature for inspecting aspects of the graph. The ability to nest graphs complicates things a bit. Nested FactorGraphs can be considered Factors. All of the introspection features allow the user to view nested graphs as leaf factors or to descend into them and operate on the children of the nested graphs. Each feature provides several methods:

- <FeatureName>(int relativeNestingDepth) – The relativeNestingDepth specifies how deep to descend into the nested FactorGraphs before treating deeper NestedGraphs as Factors. Specifying 0 will treat the top level nested Graphs as factors. Specifying a large enough number will descend all the way to the leaf factors. Specifying something between 0 and the FactorGraph's maximum depth will descend as far as this parameter specifies before considering NestedGraphs to be factors. The parameter contains the word "relative" because users can retrieve nested graphs. They can call one of the feature's methods on that nested graph.
- <FeatureName>Flat() – equivalent of <FeatureName>(max int)
- <FeatureName>Top() – equivalent of <FeatureName>(0)
- <FeatureName>() – equivalent of <FeatureName>Flat(). It was thought that users will most often want to operate on the FactorGraph in its flattened form.

Now, on to the specific features.

5.1.4.1 Retrieving All Factors

Users can retrieve Factors and/or NestedGraphs associated with a graph using the Factors methods and properties:

- Fg.getFactors()
- Fg.getFactorsFlat()
- Fg.getFactorsTop()
- Fg.getFactors(relativeNestingDepth)

When the user specifies a relativeNestingDepth or calls FactorsTop, the resulting array will contain a mix of leaf factors and Nested Graphs.

5.1.4.2 Retrieving Factors but Not Nested Factor Graphs

The FactorGraph class provides the following:

- getNonFactorGraphFactors()
- getNonFactorGraphFactorsFlat()

- getNonFactorGraphFactorsTop()
- getNonFactorGraphFactors(relativeNestingDepth)

As the name implies, this will behave similar to the Factors properties and methods but will exclude nested graphs.

5.1.4.3 Retrieving Variables

The FactorGraph class provides the following:

- getVariables() – calls getVariablesFlat()
- getVariablesFlat() – Returns a list of all the Variables in the graph, including those contained by nested graphs.
- getVariablesTop() – Returns only those variables contained in the top level of the graph.
- getVariables(relativeNestingDepth,forceIncludeBoundaryVariables) – Returns all variables contained in the FactorGraph from which the method is called as Variables that are as deep as the specified relativeNestingDepth. The second parameter is optional and defaults to false. When false, boundary variables are only included by the root graph. When true, boundary variables are included regardless of whether a graph is a root or nested graph.

5.1.4.4 Retrieving All Nodes

The FactorGraph provides the following:

- getNodes()
- getNodesFlat()
- getNodesTop()
- getNodes(relativeNestingDepth,forceIncludeBoundaryVariables)

These methods call the Factor and Variable methods and concatenate the results together.

5.1.4.5 Determining if a FactorGraph is a tree

The FactorGraph class provides the following:

- isTree(relativeNestingDepth)
- isTreeTop()
- isTreeFlat()

isTree – Users can call <factor graph name>.isTree() to determine if a FactorGraph is a tree. If the graph contains cycles, this method will return false. Like the other methods, the relativeNestingDepth determines at what point to consider NestedGraphs to be leaf nodes.

5.1.4.6 Retrieving an Adjacency Matrix

All of the following methods return a 2D int array A, where A is a square connectivity matrix.

- getAdjacencyMatrix(relativeNestingDepth,forceIncludeBoundaryVariables) – relativeNestingDepth behaves the same as in other methods that take this parameter. So does forceIncludeBoundaryVariables. forceIncludeBoundaryVariables has a default value of false.
- getAdjacencyMatrix(nodes,forceIncludeBoundaryVariables) – Users can specify a specific subset of nodes in which they are interested. This method will return an adjacency matrix with only those nodes. Nodes are considered connected only if there is an edge directly connecting them.
- getAdjacencyMatrixTop() – equivalent to getAdjacencyMatrix(0,false)
- getAdjacencyMatrixFlat() – equivalent to getAdjacencyMatrix(intmax,false)

FactorGraph also provides an AdjacencyMatrix Property:

- AdjacencyMatrix – equivalent to getAdjacencyMatrixFlat

An example of getAdjacencyMatrix:

```
FactorGraph fg = new FactorGraph();
Bit b1 = new Bit();
b1.setName("b1");
Bit b2 = new Bit();
```

```
b2.setName("b2");
Equals ne = new Equals();
Factor f = fg.addFactor(ne,b1,b2);
f.setName("f");
MapList<INode> nodes = fg.getNodesFlat();
int [][] matrix = fg.getAdjacencyMatrix(nodes);
for (INode n : nodes)
  System.out.println(n);
for (int i = 0; i < matrix.length; i++)
  System.out.println(Arrays.toString(matrix[i]));
```

results in:

```
b1
b2
f
[0, 0, 1]
[0, 0, 1]
[1, 1, 0]
```

5.1.4.7 Depth First Search

- depthFirstSearch(node, searchDepth, relativeNestingDepth) –
  - node – Specifies the node from which to initiate the search
  - searchDepth – specifies how far from node the search should go.
  - relativeNestingDepth – determines how deep to go down the NestedGraphs before considering NestedGraphs to be leaf nodes.
- depthFirstSearchFlat(node, searchDepth) – equivalent of depthFirstSearch(node,searchDepth,maxint)
- depthFirstSearchThop(node, searchDepth) – equivalent of depthFirstSearch(node,searchDepth,0)

An example:

```
FactorGraph fg = new FactorGraph();
Bit [] b = new Bit[6];
for (int i = 0; i < b.length; i++)
{
  b[i] = new Bit();
  b[i].setName("B" + i);
}
Factor f1 = fg.addFactor(new XorDelta(), b[0],b[1],b[2],b[3]);
f1.setName("f1");
Factor f2 = fg.addFactor(new XorDelta(), b[3],b[4],b[5]);
f2.setName("f2");
```

```
MapList<INode> nodes = fg.depthFirstSearch(b[0],3);

for (INode n : nodes)
{
  System.out.println(n);
}
```

results in:

```
B0
f1
B1
B2
B3
f2
```

As you might guess fg.depthFirstSearch(b[0],3) will return a collection of six nodes: b0, f1, b1, b2, b3, and f2. It will not include b5 and b6 since those are at a depth of four from b1.

5.2 Variables and Related Classes

5.2.1 Variable Types

The following variable types are defined in Dimple. Some variable types are supported by only a subset of solvers. The following table lists the Dimple variable types and the solvers that support them.

Variable Type	Supported Solvers
Discrete	all
Bit	all
Real	SumProduct, Gibbs, ParticleBP
RealJoint	SumProduct, Gibbs
Complex	SumProduct, Gibbs
FiniteFieldVariable	all[16]

5.2.2 Common Properties and Methods

The following properties and methods are common to variables of all types.

5.2.2.1 Properties

5.2.2.1.1 Name

Read-write. When read, retrieves the current name of the variable or array of variables. When set, modifies the name of the variable to the corresponding value. The value set must be a string.

```
var.setName('string');
```

5.2.2.1.2 Label

Read-write. All variables and factors in a Factor Graph must have unique names. However, sometimes it is desirable to have variables or factors share similar strings when being plotted or printed. Users can set the Label property to set the name for display. If the Label is

[16]The performance enhanced implementations of built-in factors for FiniteField variables are only available when using the SumProduct solver.

not set, the Name will be used for display. Once the label is set, the label will be used for display.

```
var.setLabel('string');
```

5.2.2.1.3 Domain

Read-only. Returns the domain in a form that depends on the variable type, as summarized in the following table:

Variable Type	Domain Data Type
Discrete	DiscreteDomain (see section 5.2.9)
Bit	DiscreteDomain (see section 5.2.9)
Real	RealDomain (see section 5.2.10)
RealJoint	RealJointDomain (see section 5.2.11)
Complex	ComplexDomain (see section 5.2.12)
FiniteFieldVariable	FiniteFieldDomain (see section 5.2.13)

5.2.2.1.4 Solver

Read-only. Returns the solver-object associated with the variable, to which solver-specific methods can be called. See section 5.6, which describes the solvers, including the solver-specific methods for each solver.

5.2.2.1.5 Guess

Read-write. Specifies a value from the variable to be used when computing the Score of the factor graph (or of the variable or neighboring factors). The Guess must be a valid value from the domain of the variable.

If the Guess had not yet been set, its value defaults to the most likely belief (which corresponds to the Value property of the variable)[17].

5.2.2.1.6 Score

Read-only. When read, computes and returns the score (energy) of the Input to this variable, which is treated as a single-edge factors, given a specified value for the variable. The score

[17]For some solvers, beliefs are not supported for all variable types; in such cases there is no default value, so a Guess must be specified in order to compute the Score.

value is relative, and may be arbitrarily normalized by an additive constant.

The value of the variable used when computing the Score is the Guess value for this variable (see section 5.2.2.1.5). If no Guess had yet been specified, the value with the most likely belief (which corresponds to the Value property of the variable) is used[18].

5.2.2.1.7 Internal Energy

Read-only. (Only applies to the Sum Product Solver). When read, returns:

$$InternalEnergy(i) = \sum_{d \in D} B_i(d) * (-log(Input(d)))$$

Read-only. When read returns:

Where D is variable i's domain, Input is the variable's input, and B_i is the variable Belief.

```
double ie = v.getInternalEnergy();
```

5.2.2.1.8 Bethe Entropy

Read-only. (Only applies to the Sum Product Solver). When read, returns:

$$BetheEntropy(i) = -\sum d \in DB_i(d) * log(B_i(d))$$

Where D is variable i's domain and B_i is the variable Belief.

```
double be = v.getBetheEntropy();
```

5.2.2.1.9 Ports

Read-only. Retrieves an array containing a list of Ports connecting the variable to its neighboring factors.

[18]For some solvers, beliefs are not supported for all variable types; in such cases there is no default value, so a Guess must be specified.

5.2.3 Discrete

A Discrete variable represents a variable that can take on a finite set of distinct states. The Discrete class corresponds to either a single Discrete variable or a multidimensional array of Discrete variables. All properties/methods can either be called for all elements in the collection or for individual elements of the collection.

5.2.3.1 Constructor

The Discrete constructor can be used to create an N-dimensional collection of Dimple Discrete variables. The constructor is called with the following arguments (arguments in brackets are optional).

```
Discrete(domain)
```

- domain is a required argument indicating the domain of the variable. The domain may either be an object array of domain elements, a comma separated list, or a DiscreteDomain object (see section 5.2.3.1.1).

For example:

```
Object [] domain = new Object [] {0,1,2};
Discrete w = new Discrete(domain);
Discrete x = new Discrete(0,1,2);
DiscreteDomain dd = DiscreteDomain.create(0,1,2);
Discrete y = new Discrete(dd);
```

We examine each of these arguments in more detail in the following sections.

5.2.3.1.1 Domain

Every Discrete random variable has a domain associated with it. A domain is a set. Elements of the set may be any object type. For example, the following are Discrete variables with valid domains:

```
Discrete a = new Discrete(1, 2, 3);
Discrete b = new Discrete(new double [] {1,0,0,1},1);
Discrete c = new Discrete("red","blue",2);
```

The domain may also be specified using a DiscreteDomain object. In that case, the domain of the variable consists of the elements of this object. For example:

```
DiscreteDomain myDomain = DiscreteDomain.create(0,1,2);
Discrete a = new Discrete(myDomain);
```

See section 5.2.9 for more information about the DiscreteDomain class.

5.2.3.2 Properties

5.2.3.2.1 Belief

Read-only. For any single variable, the Belief method returns a vector whose length is the total number of elements of the domain of the variable. When called after running a solver to perform inference on the graph, each element of the vector contains the estimated marginal probability of the corresponding element of the domain of the variable. The results are undefined if called prior to running a solver.

For an array of variables, the Belief method will return an array of vectors (that is, an array one dimension larger than the variable array) containing the beliefs of each variable in the array.

5.2.3.2.2 Value

Read-only. In some cases, one may wish to retrieve the single most likely element of a variable's domain. The Value property does just that.

For any single variable, the Value method returns a single value chosen from the domain of the variable. When called after running a solver to perform inference on the graph, the value returned corresponds to the element in the variable's domain that has the largest estimated marginal probability[19]. The results are undefined if called prior to running a solver.

For an array of variables, the Value method will return an array of values, each from the domain of the corresponding variable representing the largest estimated marginal probability for that variable.

5.2.3.2.3 Input

Read-write. For any variable, the Input method can be used to set and return the current input of that variable. An input behaves much like a single edge factor connected to

[19]If more than one domain element has identical marginal probabilities that are larger than for any other value, a single value from the domain is returned, chosen arbitrarily among these.

the variable, and is typically used the represent the likelihood function associated with a measured value (see section 4.2.2.1).

When read, for a single variable returns an array of values with each value representing the current input setting for the corresponding element of the variable's domain. The length of this array is equal to the total number of elements of the domain. When read, for an array of variables, the result is an array with dimension one larger than the dimension of the variable array. The additional dimension represents the current set of input values for the corresponding variable in the array.

When written, for a single variable, the value must be an array of length equal to the domain of the variable. The values in the array must all be non-negative, and non-infinite, but are otherwise arbitrary. When written, for an array of variables, the values must be a multidimensional array where the first set of dimensions exactly match the dimensions of the array (or the portion of the array) being set, and length of the last dimension is the number of elements in the variable's domain.

5.2.3.2.4 FixedValue

Read-write. For any variable, the FixedValue property can be used to set the variable to a specific fixed value, and to retrieve the fixed-value if one has been set. This would generally be used for conditioning a graph on known data without modifying the graph (see section 4.2.2.2).

Reading this property results in an error if no fixed value has been set. To determine if a fixed value has been set, use the hasFixedValue method (see section 5.2.3.3.1).

When setting this property on a single variable, the value must be a value included in the domain of the variable. The fixed value must be a value chosen from the domain of the variable. For example:

```
Discrete a = new Discrete(1,2,3);
a.setFixedValue(3);
```

When setting this property on a variable array, the value must be an array of the same dimensions as the variable array, and each entry in the array must be an element of the domain.

Because the Input and FixedValue properties serve similar purposes, setting one of these overrides any previous use of the other. Setting the Input property removes any fixed value and setting the FixedValue property replaces the input with a delta function—the value 0 except in the position corresponding to the fixed value that had been set.

5.2.3.3 Methods

5.2.3.3.1 hasFixedValue

This method takes no arguments. When called for a single variable, it returns a boolean indicating whether or not a fixed-value is currently set for this variable. When called for a variable array, it returns a boolean array of dimensions equal to the size of the variable array, where each entry indicates whether a fixed value is set for the corresponding variable.

5.2.4 Bit

A Bit is a special kind of Discrete with domain [0 1].

5.2.4.1 Constructor

```
Bit()
```

5.2.4.2 Properties

5.2.4.2.1 Belief

Read-only. For a single Bit variable, the Belief property is a single number that represents the estimated marginal probability of the value one.

For an array of Bit variables, the Belief property is an array of numbers with size equal to the size of the variable array, with each value representing the estimated marginal probability of one for the corresponding variable.

5.2.4.2.2 Value

See section 5.2.3.2.2.

5.2.4.2.3 Input

Read-write. For setting the Input property on a single Bit variable, the value must be a single number in the range 0 to 1, which represents the normalized likelihood of the value one (see section 4.2.2.1). If $L(x)$ is the likelihood of the variable, the Input should be set to $\frac{L(x=1)}{L(x=0)+L(x=1)}$.

For setting the Input property on an array of Bit variables, the value must be an array of normalized likelihood values, where the array dimensions must match the dimensions of the array (or the portion of the array) being set.

5.2.4.2.4 FixedValue

See section 5.2.3.2.4.

5.2.4.3 Methods

5.2.4.3.1 hasFixedValue

See section 5.2.3.3.1.

5.2.5 Real

A Real variable represents a variable that takes values on the real line, or on a contiguous subset of the real line.

5.2.5.1 Constructor

```
Real()
Real(RealDomain domain)
Real(double lowerBound, double upperBound)
```

- domain specifies a bound on the domain of the variable. It can either be specified as two elements or a RealDomain object (see section 5.2.10). If specified as two values, the first element is the lower bound and the second element is the upper bound. Negative infinity and positive infinity are allowed values for the lower or upper bound, respectively. If no domain is specified, then a domain from $-\infty$ to ∞ is assumed.

Examples:

- Real() specifies a scalar real variable with an unbounded domain.
- Real(-1, 1) specifies a scalar real variable with domain from -1 to 1.

- Real(Double.NEGATIVE_INFINITY,0) specifies a variable with the domain from negative infinity to zero.
- Real(RealDomain.create(-Math.PI, Math.PI)) specifies a scalar real variable with domain from $-\pi$ to π.

5.2.5.2 Properties

5.2.5.2.1 Belief

Read-only. The behavior of this property for Real variables is solver specific. Some solvers do not support this property at all and will return an error when read. See section 5.6 for more detail on each of the supported solvers.

For the SumProduct solver, this property returns the estimated marginal distribution of the variable in the form of a NormalParameters object (see section 5.2.14), which includes a mean and precision value. The results are undefined if called prior to running a solver.

5.2.5.2.2 Value

Read-only. The behavior of this property for Real variables is solver specific. Some solvers do not support this property at all and will return an error when read. See section 5.6 for more detail on each of the supported solvers.

For the SumProduct solver, the Value corresponds to the mean value of the belief.

5.2.5.2.3 Input

Read-write. For a Real variable, the Input property is expressed in the form of a FactorFunction object that can connect to exactly one Real variable. The list of available built-in FactorFunctions is given in section 4.1.4.4. Typically, an Input would use one of the standard distributions included in this list. In this case, it must be one in which all the parameters can be fixed to pre-defined constants. For the Gibbs and ParticleBP solvers, any such factor function may be used as an Input. For the SumProduct solver, however, only a Normal factor function may be used. Below is an example of setting the Input for a Real variable:

```
Real r = new Real();
r.setInput(new Normal(measuredMean, measurementPrecision));
```

To remove an Input that had previously been set, the Input may be set to null.

5.2.5.2.4 FixedValue

Read-write. The behavior of the FixedValue property for a Real variable is nearly identical to that of Discrete variables (see section 5.2.3.2.4). When setting the FixedValue of a Real variable, the value must be within the domain of the variable, that is greater than or equal to the lower bound and less than or equal to the upper bound. For example:

```
Real a = new Real(-Math.PI,Math.PI);
a.setFixedValue(1.7);
```

Because the Input and FixedValue properties serve similar purposes, setting one of these overrides any previous use of the other. Setting the Input property removes any fixed value and setting the FixedValue property removes the input.

5.2.5.3 Methods

5.2.5.3.1 hasFixedValue

See section 5.2.3.3.1.

5.2.6 RealJoint

A RealJoint variable is a tightly coupled set of real variables that are treated by a solver as a single joint variable rather than a separate collection of variables. For example, in the SumProduct solver, the messages associated with RealJoint variables involve joint mean and covariance matrix rather than an individual mean and variance for each variable.

Like other variables, the RealJoint class can represent either a single RealJoint variable (representing a collection of real values) or an array of RealJoint variables.

5.2.6.1 Constructor

```
RealJoint(int size)
RealJoint(RealJointDomain domain)
```

The arguments are defined as follows:

- size specifies the number of joint real-valued elements.

- domain specifies the domain of the RealJoint variable using a RealJointDomain object (see 5.2.11). Using this version of the constructor allows bounds to be specified in some or all dimensions of the domain.

5.2.6.2 Properties

5.2.6.2.1 Belief

Read-only. The behavior of this property for RealJoint variables is solver specific. Some solvers do not support this property at all and will return an error when read. See section 5.6 for more detail on each of the supported solvers.

For the SumProduct solver, this property returns the estimated marginal distribution of the variable in the form of a MultivariateNormalParameters object (see section 5.2.15), which includes a mean vector and covariance matrix. The results are undefined if called prior to running a solver.

5.2.6.2.2 Value

Read-only. The behavior of this property for Real variables is solver specific. Some solvers do not support this property at all and will return an error when read. See section 5.6 for more detail on each of the supported solvers.

For the SumProduct solver, this property returns the mean vector, with dimension equal to the dimension of the RealJoint variable.

5.2.6.2.3 Input

Read-write. For a RealJoint variable, the Input property is expressed in one of two forms: either a single FactorFunction object that can connect to exactly one RealJoint variable, or a set of FactorFunction objects that can each connect to exactly one Real variable. The latter case corresponds to a likelihood function where each dimension is independent. The list of available built-in FactorFunctions is given in section 4.1.4.4. Typically, an Input would use one of the standard distributions included in this list. In this case, it must be one in which all the parameters can be fixed to pre-defined constants. For the Gibbs and ParticleBP solvers, any such factor function may be used as an Input. For the SumProduct solver, however, only a MultvariateNormal factor function or a set of Normal factor functions may be used.

Below is an example of setting the Input for a RealJoint variable using a single multivariate factor function:

```
Real r = new Real();
r.setInput(new MultivariateNormal(measuredMeanVector,
    measurementCovarianceMatrix));
```

To remove an Input that had previously been set, the Input may be set to null.

To specify a set of univariate factor functions the value of this property must be an array of FactorFunction objects, one for each dimension of the RealJoint variable.

5.2.6.2.4 FixedValue

Read-write. The behavior of the FixedValue property for a RealJoint variable is similar to that of Discrete variables (see section 5.2.3.2.4). When setting the FixedValue of a Real variable, the value must be within the domain of the variable. When setting a fixed value, the value must be in an array with dimension equal to the dimension of the RealVariable. For example:

```
RealJoint a = new RealJoint(4);
a.setFixedValue(new double[]{1.7, 2.0, 0, -1.2});
```

Because the Input and FixedValue properties serve similar purposes, setting one of these overrides any previous use of the other. Setting the Input property removes any fixed value and setting the FixedValue property removes the input.

5.2.6.3 Methods

5.2.6.3.1 hasFixedValue

See section 5.2.3.3.1.

5.2.7 Complex

Complex is a special kind of RealJoint variable with exactly two joint elements.

The behavior of all properties and methods is identical to that of RealJoint variables.

5.2.7.1 Constructor

```
Complex()
Complex(ComplexDomain domain)
```

The arguments are defined as follows:

- domain specifies the domain of the Complex variable using a ComplexDomain object (see 5.2.12). Using this version of the constructor allows bounds to be specified in some or all dimensions of the domain.

5.2.7.2 Properties

5.2.7.2.1 Belief

Read-only. See section 5.2.6.2.1.

5.2.7.2.2 Value

Read-only. See section 5.2.6.2.2.

5.2.7.2.3 Input

Read-write. See section 5.2.6.2.3.

5.2.7.2.4 FixedValue

Read-write. See section 5.2.6.2.4.

5.2.7.3 Methods

5.2.7.3.1 hasFixedValue

See section 5.2.3.3.1.

5.2.8 FiniteFieldVariable

Dimple supports a special variable type called a FiniteFieldVariable, which represent finite fields with $N = 2^n$ elements. These fields find frequent use in error correcting codes. These variables are used along with certain custom factors that are implemented more efficiently for sum-product belief propagation than the alternative using discrete variables and factors implemented directly. See section 4.4 for more information on how these variables are used.

The behavior of all properties and methods is identical to that of Discrete variables.

5.2.8.1 Constructor

```
FiniteFieldVariable(primitivePolynomial)
```

The arguments are defined as follows:

- primitivePolynomial the primitive polynomial of the finite field. The format of the primitive polynomial follows the same definition used by MATLAB in the `gf` function. See the MATLAB help on the `gf` function for more detail.

5.2.9 DiscreteDomain

The DiscreteDomain class represents a domain with a finite fixed set of elements. It is the type of Domain used by Discrete variables. DiscreteDomain objects are immutable.

5.2.9.1 Construction

Construction of a DiscreteDomain object is done using the static create method.

```
DiscreteDomain.create(Object[] elementList)
```

The elementList argument is an array of domain elements. Every entry of the array is considered an element of the domain, regardless of the number of dimensions it has. For an array of Objects, each object in the array is considered an element of the domain regardless of the object type. For a numeric array, every entry in the array must be numeric.

5.2.9.2 Properties

5.2.9.2.1 Elements

Read-only. This property returns the set of elements in the discrete domain in the form of a one-dimensional array.

5.2.10 RealDomain

The RealDomain class is used to refer to the domain of Real variables.

5.2.10.1 Constructor

Construction of a RealDomain object is done using the static create method.

```
RealDomain.create()
RealDomain.create(lowerBound,upperBound)
```

- lowerBound indicates the lower bound of the domain. The value must be a scalar numeric value. It may be set to negative infinity to indicate that there is no lower bound. If the create method with no arguments is used, the default lower bound is negative infinity.
- upperBound indicates the upper bound of the domain. The value must be a scalar numeric value. It may be set to positive infinity to indicate that there is no upper bound. If the create method with no arguments is used, the default upper bound is positive infinity.

5.2.10.2 Properties

5.2.10.2.1 LowerBound

Read-only. This property returns the value of the lower bound. The default value is negative infinity.

5.2.10.2.2 UpperBound

Read-only. This property returns the value of the upper bound. The default value is positive infinity.

5.2.11 RealJointDomain

The RealJointDomain class is used to refer to the domain of RealJoint variables.

5.2.11.1 Constructor

Construction of a RealJointDomain object is done using the static create method.

```
RealJointDomain.create(int size)
RealJointDomain.create(RealDomain... domains)
RealJointDomain.create(RealDomain domain, int size)
```

- size indicates the number of dimensions in the domain of the RealJoint variable. If the form of create that specifies only the size is called, then all dimensions are assumed to be unbounded.
- domains is a list or array of RealDomain objects, one for each dimension. Each RealDomain in the list indicates the domain of the corresponding dimension of the RealJoint variable. The number of entries determines the number of dimensions.
- domain indicates a single RealDomain that is to be used for all dimensions of the RealJoint variable. In this form of create, the size must also be specified to indicate the number of dimensions of the RealJoint variable.

5.2.11.2 Properties

5.2.11.2.1 RealDomains

Read-only. Returns the collection of RealDomains that correspond to each dimension of the RealJointDomain.

5.2.11.3 Methods

5.2.11.3.1 getNumVars

```
domain.getNumVars();
```

Returns the number of elements in the RealJointDomain, which corresponds to the number of dimensions of an associated RealJoint variable.

5.2.11.3.2 getRealDomain

```
domain.getRealDomain(dimensionIndex);
```

Returns the RealDomain that correspond to the dimension corresponding to the specified `dimensionIndex`.

5.2.12 ComplexDomain

The ComplexDomain class, a subclass of the RealJointDomain class, is used to refer to the domain of Complex variables.

5.2.12.1 Constructor

Construction of a ComplexDomain object is done using the static create method.

```
ComplexDomain.create()
ComplexDomain.create(RealDomain... domains)
ComplexDomain.create(RealDomain domain)
```

- domains is an array of exactly two RealDomain objects, one for each dimension. Each RealDomain in the list indicates the domain of the corresponding dimension of the Complex variable (real followed by imaginary). If no domains argument is specified, then both dimensions are assumed to be unbounded.
- domain is a single RealDomain object. When only one RealDomain object is specified, the same RealDomain is used for both dimensions.

5.2.12.2 Properties

5.2.12.2.1 RealDomains

Read-only. Returns the collection of RealDomains that correspond to each dimension of the ComplexDomain (real followed by imaginary).

5.2.12.3 Methods

5.2.12.3.1 getNumVars

```
domain.getNumVars();
```

Returns the number of elements in the ComplexDomain, which should always equal two.

5.2.12.3.2 getRealDomain

```
domain.getRealDomain(dimensionIndex);
```

Returns the RealDomain that correspond to the dimension corresponding to the specified `dimensionIndex`.

5.2.13 FiniteFieldDomain

The FiniteFieldDomain class represents the domain of a FiniteFieldVariable. FiniteFieldDomain objects are immutable.

5.2.13.1 Construction

Construction of a FiniteFieldDomain object is done using the static create method.

```
FiniteFieldDomain.create(primitivePolynomial)
```

- primitivePolynomial is an integer representation of the primitive polynomial of the finite field.

5.2.13.2 Properties

5.2.13.2.1 Elements

Read-only. This property returns the set of elements in the discrete domain in the form of a one-dimensional array.

5.2.13.2.2 PrimitivePolynomial

Read-only. This property returns the primitive polynomial associated with the domain using an integer representation.

5.2.13.2.3 N

Read-only. This property returns the number of bits in the finite field. The size of the Elements property is 2^N.

5.2.14 NormalParameters

The NormalParameters class is used to specify the parameters of a univariate Normal distribution, as used in the SumProduct solver.

5.2.14.1 Constructor

```
NormalParameters(mean, precision)
```

- mean is the mean value of the distribution.
- precision is the precision of the distribution, which is the inverse of the variance.

5.2.14.2 Properties

5.2.14.2.1 Mean

Read-only. Returns the mean value.

5.2.14.2.2 Precision

Read-only. Returns the precision value.

5.2.14.2.3 Variance

Read-only. Returns the variance (inverse of the precision).

5.2.14.2.4 StandardDeviation

Read-only. Returns the standard deviation (square root of the variance).

5.2.15 MultivariateNormalParameters

The MultivariateNormalParameters class is used to specify the parameters of a multivariate Normal distribution, as used in the SumProduct solver.

5.2.15.1 Constructor

```
MultivariateNormalParameters(meanVector, covarianceMatrix)
```

- meanVector indicates the mean value of each element in a joint set of variables. The value must be a one-dimensional numeric array.
- covarianceMatrix indicates the covariance matrix associated with the elements of a joint set of variables. The value must be a two-dimensional numeric array with each dimension identical to the length of the meanVector.

5.2.15.2 Properties

5.2.15.2.1 Mean

Read-only. Returns a vector of values, where each value indicates the mean value of the corresponding element in a joint set of variables.

5.2.15.2.2 Covariance

Read-only. Returns a two-dimensional array of values, representing the covariance matrix of a joint set of variables.

5.2.15.2.3 InformationVector

Read-only. Returns a vector of values representing the information matrix, defined as $\Sigma^{-1}\mu$, where Σ is the covariance matrix and μ is the mean vector.

5.2.15.2.4 InformationMatrix

Read-only. Returns a two-dimensional array of values representing the information matrix, defined as Σ^{-1}, where Σ is the covariance matrix.

5.3 Factors and Related Classes

5.3.1 Factor

The Factor class can represent either a single factor or a multidimensional array of factors. The Factor class is never created directly, but is the result of using the addFactor or addFactorVectorized (or related) methods on a FactorGraph.

5.3.1.1 Properties

5.3.1.1.1 Name

Read-write. When read, retrieves the current name of the factor or array of factors. When set, modifies the name of the factor to the corresponding value. The value set must be a string.

```
factor.setName("string");
```

5.3.1.1.2 Label

Read-write. All variables and factors in a Factor Graph must have unique names. However, sometimes its desirable to have variables or factors share similar strings when being plotted or printed. Users can set the Label property to set the name for display. If the Label is not set, the Name will be used for display. Once the label is set, the label will be used for display.

5.3.1.1.3 DirectedTo

Read-write. The DirectedTo property indicates a set of variables to which the factor is directed. The value may be either a single variable, a comma separated list of variables, a VariableList, or an array of integers specifying the edge indexes to which the factor is directed. The DirectedTo property is used by some solvers, and in some cases is required for proper operation of certain features. Such cases are identified elsewhere in this manual.

For example, if a factor F corresponds to a function $F(a, b, c, d)$, where a, b, c, and d are variables, then the factor is directed toward c and d if $\sum_{c,d} F(a, b, c, d)$ is constant for all values of a and b. In this case, we may set:

```
F.setDirectedTo(c,d);
```

In some cases, the set of DirectedTo variables can be automatically determined when a factor is created. In this case it need not be set manually by the user. This includes many built-in factors supported by Dimple. If this property is set by the user, then in the case of factors connected only to discrete variables, Dimple will check that the factor is in fact directed in the specified direction.

5.3.1.1.4 Score

Read-only. When read, computes and returns the score (energy) of the factor given a specified value for each of the neighboring variables to this factor. The score represents the energy of the factor given the specified variable configuration. The score value is relative, and may be arbitrarily normalized by an additive constant.

The value of each variable used when computing the Score is the Guess value for that variable (see section 5.2.2.1.5). If no Guess had yet been specified for a given variable, the value with the most likely belief (which corresponds to the Value property of the variable) is used[20].

The variable energy is normalized by the maximum input probability.

5.3.1.1.5 InternalEnergy

Read-only. (Only applies to the Sum Product Solver). When read returns:

$$InternalEnergy(a) = \sum_{\vec{x} \in \vec{X}} B_a(\vec{x}) * (-log(Weight(\vec{x})))$$

Where a is an instance of a Factor, X is the set of variables connected to a, Weight is the FactorTable entry for the specified set of variable values, and B_a is the belief of that factor node.

```
double ie = f.getInternalEnergy();
```

5.3.1.1.6 Bethe Entropy

Read-only. (Only applies to the Sum Product Solver). When read returns:

[20]For some solvers, beliefs are not supported for all variable types; in such cases there is no default value, so a Guess must be specified.

$$BetheEntropy(a) = -\sum_{\vec{x} \in domain(\vec{X})} B_a(\vec{x}) * log(B_a(\vec{x}))$$

Where a is an instance of a Factor, X is the set of variables connected to a, and B_a is the belief of that factor node.

5.3.1.1.7 Belief

Read-only. (Only applies to the Sum Product Solver). To support the Bethe Free Energy property, Dimple provides getBelief associated with a Factor.

$$Belief_a(\vec{x}) = Weight(\vec{x}) \prod_{i=0}^{N} \mu_{X_i \rightarrow a}(x_i)$$

Where $\vec{x} \in domain(\vec{X})$ and $\vec{X}$ is the set of variables connected to the factor a.

```
double b = f.getBelief();
```

5.3.1.1.8 Ports

Read-only. Retrieves an array containing a list of Ports connecting the factor to its neighboring variables.

5.3.2 DiscreteFactor

When all variables connected to a Factor are discrete, a DiscreteFactor is created.

5.3.2.1 Properties

5.3.2.1.1 Belief

Read-only. The belief of a factor is the joint belief over all joint states of the variables connected to that factor. There are two properties that represent the belief in different ways:

Belief and FullBelief. Reading the Belief property after the solver has been run[21] returns the belief in a compact one-dimensional vector that includes only values that correspond to non-zero entries in the factor table. This form is used because in some situation, the full representation over all possible variable values (as returned by the FullBelief property) would result in a data structure too large to be practical.

```
double [] fb = myFactor.getBelief();
```

The result is a vector of belief values, where the order of the vector corresponds to the order of the factor table entries. The order of factor table entries can be determined from the factor using:

```
int [][] indices = f.getFactorTable().getIndices()
```

This returns a two-dimensional array, where each row corresponds to one entry in the factor table, and where each column-entry in a row indicates the index into the domain of the corresponding variable (where the order of the variable is the order used when the factor was created).

5.3.2.1.2 FullBelief

Read-only. Reading the FullBelief property after the solver has been run[22] returns the belief in a multi-dimensional array, where each dimension of the multi-dimensional array ranges over the domain of the corresponding variable (the order of the dimensions corresponds to the variable order used when the factor was created).

```
double [] fb = myFactor.getFullBelief();
```

5.3.3 FactorFunction

The FactorFunction class is an abstract class that is used as the base class for all FactorFunctions passed to the addFactor call. To create new FactorFunctions, users must override the FactorFunction class.

[21] In the current version of Dimple, the Belief property is only supported for factors connected exclusively to discrete variables, and is supported only by the SumProduct solver. These restrictions may be removed in a future version.

[22] In the current version of Dimple, the Belief property is only supported for factors connected exclusively to discrete variables, and is supported only by the SumProduct solver. These restrictions may be removed in a future version.

5.3.3.1 Constructors

- FactorFunction()
- FactorFunction(String name);

5.3.3.2 Methods

5.3.3.2.1 evalEnergy

This method must be overridden in a derived class. The method takes an array of Values and returns the energy (negative log of the weight value) associated with the corresponding values.

```
double evalEnergy(Value[] values);
```

5.3.3.2.2 isDirected

This method takes no arguments and returns a boolean indicating whether the factor function is directed. If directed, then there are a set of directed outputs for which the marginal distribution for all possible input values is a constant. This method may be overridden in a derived class. If not overridden, it is assumed false.

5.3.3.2.3 getDirectedToIndices

If a factor function is directed, this method indicates which edge indices are the directed outputs (numbering from zero), returning an array of integers. There are two forms of this method, which may be used depending on whether the set of directed outputs depends on the number of edges in the factor that uses this factor function (many factor functions support a variable number of edges). If `isDirected` is overridden and can return `true`, then this method must also be overridden. If the result depends on the number of edges connecting to the factor, then the second form must be overridden, otherwise the first form may be overridden instead.

```
int[]  getDirectedToIndices();
```

```
int[] getDirectedToIndices(int numEdges);
```

5.3.3.2.4 isDeterministicDirected

This method takes no arguments and returns a boolean indicating whether a factor function is both directed and deterministic. If deterministic and directed, then it is in the form of a deterministic function such that for all possible settings of the input values there is exactly one output value the results in a non-zero weight (or, equivalently, a non-infinite energy)[23]. This method may be overridden in a derived class. If not overridden, it is assumed false.

5.3.3.2.5 evalDeterministic

If a factor function is directed and deterministic, this method evaluates the values considered the inputs of the deterministic function and returns the resulting values for the corresponding outputs. Note that these are not the weight or energy values, but the actual values of the associated variables that are considered outputs of the deterministic function. The resulting values are returned in-place to the values passed in the argument list. If `isDeterministicDirected` is overridden and can return `true`, then this method must also be overridden.

```
evalDeterministic(Value[] arguments);
```

5.3.3.2.6 eval

The method takes an array of Values and returns the weight associated with the corresponding values. This method need not be overridden in a derived class as the default implementation uses `evalEnergy` to compute the result.

```
double eval(Value[] values);
```

5.3.4 FactorTable

The FactorTable class is used to explicitly specify a factor table in lieu of Dimple creating one automatically from a factor function. This is sometimes useful in cases where the factor table is highly structured, but automatic creation would be time consuming due to a large number of possible states of the connected variables.

5.3.4.1 Constructor

[23]The indication that a factor function is deterministic directed is used by the Gibbs solver, and is necessary for such factor functions to work when using the Gibbs solver.

- FactorTable.create(DiscreteDomain ... domains) - Creates a full table with with the specified domains.
- FactorTable.create(int [][] indices, double [] weights,DiscreteDomain ... domains) - Creates a table with the specified indices and weights with the specified domains
- FactorTable.create(int [][] indices, double [] weights, Discrete ... variables) - Same as the previous method but extracts the domains from the variables.
- FactorTable.create(Object table, DiscreteDomain [] domains) - Assumes table is a tensor (number of dimensions equal to number of domains).

A FactorTable is constructed by specifying the table values in one of two forms, or by creating an all-zeros FactorTable to be filled in later using the set method. The first form, specifying an indexList and weightList, is useful for sparse factor tables in which many entires are zero and need not be included in the table. The second form, specifying a weightMatrix, is useful for dense factor tables in which most or all of the entries are non-zero.

- indices is an array where each row represents a set of zero-based indices into the list of domain elements for each successive domain in the given set of domains.
- weights is a one-dimensional array of real-valued entries in the factor table, where each entry corresponds to the indices given by the corresponding row of the indexList.
- table is an N dimensional array of real-valued entries in the factor table. The number of dimensions, N, must correspond to the number of discrete domain elements given in the subsequent arguments, and the number of elements in each dimension must equal the number of elements in the corresponding domain element.

5.3.4.2 Properties

5.3.4.2.1 Indices

Read-write. When read, returns an array of indices of the factor table corresponding to entries in the factor table. Each row represents a set of zero-based indices into the list of domain elements for each successive domain in the given set of domains.

When written, replaces the previous array of indices with a new array. When writing using this property, the number of rows in the table must not change since this must equal the number of entires in the Weights. To change both Indices and Weights simultaneously, use the change method.

5.3.4.2.2 Weights

Read-write. When read, returns a one-dimensional array of real-valued entries in the factor table, where each entry corresponds to the indices given by the corresponding row of the indexList.

When written, replaces the previous array of weights. When writing using this property, the number of entries must not change since this must equal the number of rows in the Indices. To change both Indices and Weights simultaneously, use the change method.

5.3.4.2.3 Domains

Read-only. Returns an array of DiscreteDomain objects, each of which represents the corresponding domain specified in the constructor, in the same order as specified in the constructor.

5.3.4.3 Methods

5.3.4.3.1 set

This method allows setting individual entries in the factor table. For each entry to be set, the indices are specified followed by the weight value.

```
ft.set(new int [] {0,1,1},1.0);
```

5.3.4.3.2 get

This method retrieves the weight associated with a particular entry in the factor table. The entry is specified by a comma-separated list of indices. For example:

```
double w = ft.get(new int [] {0,1,1});
```

5.3.4.3.3 change

This method allows simultaneously replacing both the array of Indices and Weights in the factor table.

```
ft.change(indexList, weightList);
```

The arguments indexList and weightList are exactly as described in the FactorTable constructor.

5.4 Options

Dimple provides an option mechanism used to configure the runtime behavior of various aspects of the system. This section describes the Dimple option system and how it is used. Only the most widely useful parts of the API are described here: for complete documentation of the Java API please consult the HTML Java API documentation. Individual options are described in more detail later in this document.

Options were introduced in version 0.07 and replace earlier mechanisms based on solver-specific method calls. Those methods are now deprecated and will be removed in a future release. Users with existing Dimple code using such methods should switch to using options as soon as it is convenient to do so.

5.4.1 Option Keys

Options are key/value pairs that can be set on Dimple factor graph, variable, or factor objects to configure their behavior. An option key uniquely identifies an option, along with its type and default value.

In the Java API, option keys are singleton instances of the IOptionKey interface that are declared as public static final fields of a publicly accessible class. To refer to an option key, you only need to import the class in which it is declared and refer to the field, e.g:

```java
import com.analog.lyric.dimple.options.SolverOptions;
...
graph.setOption(BPOptions.iterations, 12);
```

The IOptionKey interface defines a number of methods that can be used to query the name(), type(), and defaultValue() as well as methods for converting and validating values for that option. Users should have little reason to invoke any of these directly. Details may be found in the HTML Java API documentation.

5.4.2 Setting Options

Options may be set on any FactorGraph, Factor, or Variable object or their solver-specific counterparts. Options may also be set on the DimpleEnvironment object, which is described in more detail below. (Options may in fact be set on any object that implements the local option methods of the IOptionHolder interface, but this should not matter to most Dimple users.) Options set on graphs will be applied to all factors, variables, and subgraphs contained in the graph unless overridden on one of those members. Likewise options set on a model object will be applied to an associated Solver object to it unless overridden directly in the Solver object. In most cases, it should not be necessary to set options directly on Solver objects.

Options can be set either using the setOption method (defined in the IOptionHolder interface) or through the set method of the option key itself. For example:

```
// These both do the same thing:
graph.setOption(BPOptions.damping, .9);
BPOptions.damping.set(graph, .9);
```

Some option keys may define additional set methods that may be more convenient to use than the setOption method:

```
// These both do the same thing:
factor.setOption(BPOptions.nodeSpecificDamping,
    new OptionDoubleList(.7, .8, .9));
BPOptions.nodeSpecificDamping.set(factor, .7, .8, .9);
```

All of these methods will ensure that the type of the option values are appropriate for that key and may also validate the value. For instance when setting the BPOptions.damping option, the value must be a double in the range from 0.0 to 1.0. If a value is not valid for its key an OptionValidationException will be thrown.

Options may be unset on any object on which they were previously set using the unset method:

```
graph.unsetOption(BPOptions.damping);
```

or the unset method on the option key can be used:

```
BPOptions.damping.unset(graph);
```

All options may be unset on an object using the clearLocalOptions method:

```
graph.clearLocalOptions();
```

5.4.3 Looking up Option Values

There are a number of methods for retrieving option values from objects on which they can be set. Most users will only need to use these to debug their option settings.

The option value that applies to a given object is determined hierarchically, based on an order that depends on the structure of the graph. An option value specified at any level applies to all objects below it in the hierarchy, unless specifically specified for an object lower in the hierarchy. At any level, the option value overrides the value specified at a higher level. When querying an object to determine what option value will be used, the hierarchy is searched in the following order[24]:

1. Search the object itself.
2. If the object is a solver object, next look at the corresponding model object.
3. If the object has a parent graph, then recursively search that graph, otherwise the DimpleEnvironment for that object will be searched (there is usually only one environment).

There are a number of methods defined on the IOptionKey and IOptionHolder interfaces that can be used to lookup the value of the option for that object using the lookup algorithm that was just described:

- IOptionKey.get(obj) and IOptionHolder.getOption(key) - both return the value of the option for the given object or else null if not set.

```
// These are equivalent:
Double damping1 = graph.getOption(BPOptions.damping);
Double damping2 = BPOptions.damping.get(graph);
```

- IOptionKey.getOrDefault(obj) and IOptionholder.getOptionOrDefault(key) - return the default value defined by the key instead of null when the option has not been set anywhere.

```
// These are equivalent:
double damping1 = graph.getOptionOrDefault(BPOptions.damping);
double damping2 = BPOptions.damping.getOrDefault(graph);
```

There are also methods for querying the values of only those options that are set directly on an object:

- IOptionHolder.getLocalOption(key) - returns the value of the option or else null if not set directly on the object.

```
Double damping = graph.getLocalOption(BPOptions.damping);
```

- IOptionHolder.getLocalOptions() - returns a read-only collection that provides a view of the option settings for that object in the form of IOption objects.

[24]The algorithm is actually slightly more complicated than this but the details should only matter to those implementing custom factors or solvers. For details see the documentation for EventSourceIterator in the HTML Java API documentation.

```java
// Print options set on node.
for (IOption<?> option : node.getLocalOptions())
{
    System.out.format("%s = %s\n", option.key(), option.value());
}
```

5.4.4 Option Initialization

While option values are visible as soon as they are set on an object, they may not take effect until later because internal objects that are affected by the change may have cached state based on the previous settings, or may not yet exist. The documentation for individual options should indicate when changes to the settings are incorporated, but in most cases that will happen when the initialize() method is called on the affected object. Since this happens automatically when invoking the FactorGraph.solve() method, users will often not have to be concerned with this detail. But if you performing other operations, such as directly calling FactorGraph.iterate(), then you will probably need to invoke FactorGraph.initialize() for modified option settings to take effect.

5.4.5 Setting Defaults on the Dimple Environment

Sometimes you may want to apply the same default option settings across multiple graphs. While you can simply set the options on all of the graphs individually, another choice is to set it on the DimpleEnvironment object. The DimpleEnvironment holds shared state for a Dimple session. Typically there will be only one instance of this class. Because the environment is the last place searched for option lookup, you can use it as a place to set default values of options to override those defined by the option keys.

You can obtain a reference to the active global environment using the static DimpleEnvironment.active() method: and set default option values on it. For instance, to globally enable multithreading for all graphs, you could write:

```java
DimpleEnvironment env = DimpleEnvironment.active();
env.setOption(SolverOptions.enableMultithreading, true);
```

5.5 Schedulers

A scheduler defines a rule that determines the update schedule of a factor graph when performing inference. This section describes all of the built-in schedulers available in Dimple.

Each scheduler is applicable only to a certain subset of solvers. For the BP solvers (other than the Junction Tree solvers, that is, SumProduct, MinSum, and ParticleBP), the following schedulers are available:

Name	**Description**
DefaultScheduler	Same as the TreeOrFloodingScheduler, which is the default if no scheduler or custom schedule is specified.
TreeOrFloodingScheduler	The solver will use either a Tree Schedule or a Flooding Schedule depending on whether the factor-graph contains cycles. In a nested graph, this choice is applied independently in each subgraph. If the factor-graph is a tree, the scheduler will automatically detect this and use a Tree Schedule. In this schedule, each node is updated in an order that will result in the correct beliefs being computed after just one iteration. If the entire graph is a tree, NumIterations should be set to 1, which is its default value. If the factor-graph is loopy, the solver will instead use a Flooding Schedule (as described below).
TreeOrSequentialScheduler	The solver will use either a Tree Schedule (as described above) or a Sequential Schedule (as described below) depending on whether the factor-graph contains cycles. In a nested graph, this choice is applied independently in each subgraph.
FloodingScheduler	The solver will apply a Flooding Schedule. For each iteration, all variable nodes are updated, followed by all factor nodes. Because the graph is bipartite (factor nodes only connect to variable nodes, and vice versa), the order of update within each node type does not affect the result.
SequentialScheduler	The solver will apply a Sequential Schedule. For each factor node in the graph, first, for each variable connected to that factor, the edge connecting the variable to the factor is updated; then the factor node is updated. The specific order of factors chosen is arbitrary, and depends on the order that factors were added to the graph.
RandomWithoutReplacementScheduler	The solver will apply a Sequential Schedule with the order of factors chosen randomly without replacement. On each subsequent iteration, a new random order is chosen. Since the factor order is chosen randomly with replacement, on each iteration, each factor will be updated exactly once.

RandomWithReplacementScheduler	The solver will apply a Sequential Schedule with the order of factors chosen randomly with replacement. On each subsequent iteration, a new random order is chosen. The number of factors updated per iteration is equal to the total number of factors in the graph. However, since the factors are chosen randomly with replacement, not all factors are necessarily updated in a single iteration, and some may be updated more than once.

For the JunctionTree and JunctionTreeMAP solvers, only a Tree Schedule will be used. When using these solvers, the Scheduler setting will be ignored.

In a nested graph, for most of the schedulers listed above (except for the random schedulers), the schedule is applied hierarchically. In particular, a subgraph is treated as a factor in the nesting level that it appears. When that subgraph is updated, the schedule for the corresponding subgraph is run in its entirety, updating all factors and variables contained within according to its specified schedule.

It is possible for subgraphs to be designated to use a schedule different from that of its parent graph. This can be done by specifying either a scheduler or a custom schedule for the subgraph prior to adding it to the parent graph. For example:

```
SubGraph.setScheduler(new com.analog.lyric.dimple.schedulers.
    SequentialScheduler());
ParentGraph.addFactor(SubGraph , a,b);
ParentGraph.setScheduler(new com.analog.lyric.dimple.schedulers.
    TreeOrFloodingScheduler());
```

For the TreeOrFloodingScheduler and the TreeOrSequentialScheduler, the choice of schedule is done independently in the outer graph and in each subgraph. In case that a subgraph is a tree, the tree scheduler will be applied when updating that subgraph even if the parent graph is loopy. This structure can improve the performance of belief propagation by ensuring that the effect of variables at the boundary of the subgraph fully propagates to all other variables in the subgraph on each iteration.

For the RandomWithoutReplacementScheduler and RandomWithReplacementScheduler, if these are applied to a graph or subgraph, the hierarchy of any lower nesting layers is ignored. That is, the subgraphs below are essentially flattened prior to schedule creation, and any schedulers or custom schedules specified in lower layers of the hierarchy are ignored.

Because of the differences in operation between the Gibbs solver and the BP based solvers, the Gibbs solver supports a distinct set of schedulers. For the Gibbs solver, the following schedulers are available:

Name	Description
GibbsDefaultScheduler	Same as the GibbsSequentialScanScheduler, which is the default when using the Gibbs solver.
GibbsSequentialScanScheduler	The solver will apply a Sequential Scan Schedule. For each scan, each variable is resampled in a fixed order. The specific order of variables chosen is arbitrary, and depends on the order that variables were added to the graph.
GibbsRandomScanScheduler	The solver will apply a Random Scan Schedule. Each successive variable to be resampled is chosen randomly with replacement. The number of variables resampled per scan is equal to the total number of variables in the graph, but not all variables are necessarily resampled in a given scan, and some may be resampled more than once.

Because of the nature of the Gibbs solver, the nested structure of a graph is ignored in creating the schedule. That is, the graph hierarchy is essentially flattened prior to schedule creation, and only the scheduler specified on the outermost graph is applied.

Schedulers are not applicable in the case of the LP solver.

5.6 Solvers

5.6.1 Solver-Specific Options

Each solver supports a number of options specific to that solver. These are described in the following sections. Solver-specific options may be used to configure how overall inference works for that solver or may be used to configure the behavior of individual factors or variables for that solver. Solver-specific options are typically set on the applicable model object (factor graph, factor, or variable) and the values will be observed and used to configure the corresponding solver objects:

```
model-object.setOption(SolverOptionClass.optionName, option-value)
```

Solver-specific options may be set at any time, even before the solver for a graph has been specified. Options that are not applicable to the object on which it is set or that are not applicable to the active solver will simply be ignored. For more details on the options mechanism see subsection 5.4 of this document.

5.6.2 Solver-Specific Methods

Each solver also may support solver-specific methods, which are described in the following sections. As with options, solver-specific methods may be available for various objects: a factor-graph, variable, or factor. In each case, to call a solver-specific method, the method is applied to the solver object, returned by the Solver property. For example:

```
((SolverSpecificGraph)fg.getSolver()).solverSpecificMethod(arguments);
```

```
((SolverSpecificVar)variable.getSolver()).solverSpecificMethod(arguments);
```

```
((SolverSpecificFactor)factor.getSolver()).solverSpecificMethod(arguments);
```

The downcast in the first cast can be avoided if you save the return value from setSolver-Factory and call the methods through that. For example:

```
JunctionTreeSolverGraph sgraph =
    fg.setSolverFactory(new JunctionTreeSolver());
sgraph.useConditioning(true);
```

The downcasts are also not required for common methods available to all solvers described in the next section.

Some solver-specific methods return results, while others do not. Some solver-specific methods require arguments, while others do not.

5.6.3 Common Options

A few options are applicable to multiple solvers and are therefore described in this subsection.

5.6.3.1 SolverOptions.enableMultithreading

Type	boolean
Default	false
Affects	graph
Description	Controls whether to use multithreading for this solver. Multithreading is currently only supported by the MinSum and SumProduct solvers but will eventually be implemented in others. This value will be ignored if not applicable.

5.6.3.2 DimpleOptions.randomSeed

Type	64-bit integer
Default	N/A
Affects	graph
Description	When set, this option specifies a random seed that may be used by solvers that use a random number generator. The seed will only be used if explicitly set; the default value is not used. This can be used to ensure repeatable behavior during testing or profiling but should not be used for normal operation.

5.6.4 Common Methods

There are also some methods that are common to all solvers. These are:

5.6.4.1 getMultithreadingManager

Dimple users can retrieve a MultithreadingManager on which to perform additional actions.

```
fg.getSolver().getMultithreadingManager()
```

Users can configure both the multithreading mode and the number of workers using the MultithreadingManager.

5.6.4.1.1 Multithreading Modes

Dimple provides various multithreading algorithms that have different speed advantages depending on the size of the user's graph and FactorTables. In the future Dimple should be modified to automatically detect the best threading algorithm. Currently, however, it defaults to the "Phase" multithreading mode and requires the user manually set the mode to change this. For a given graph, users can try both modes and see which is faster.

The currently supported multithreading modes are:

- Phase - Divides the schedule into "phases" where each phase contains schedule entries that are entirely independent of one another. These phases are then easy to parallelize.
- SingleQueue - Uses a single queue and a dependency graph to pull off work for each thread on the fly.

The following methods can be used for getting and setting modes:

- fg.getSolver().getMultithreadingManager().getModes() - Returns an array of enums specifying the valid modes.
- fg.getSolver().getMultithreadingManager().setMode(ModeName) - Allows users to set the mode by string. Currently "Phase" or "SingleQueue" will work.
- fg.getSolver().getMultithreadingManager().setMode(enum) - Allows users to set the mode by the enums returned by the getModes method or with MultithreadingMode.¡PhaseName¿.

5.6.4.1.2 Setting Number of Threads and Workers

Dimple provides a ThreadingPool as a singleton for multithreading. It sets the number of threads in this pool to the number of virtual cores in the user's machine by default. Users can override this default value. In addition, Dimple allows users to specify the number of "workers" for a given FactorGraph. This "NumWorkers" is also set to the number of virtual cores on the user's machine by default. Whereas NumThreads specifies how many threads are in the threadPool, NumWorkers specifies how work is divided up across the graph. These workers are run by the thread pool. Best performance is achieved when NumWorkers and NumThreads are the same. However, NumThreads is global and shared by all graphs where NumWorkers is specific to a given FactorGraph.

The following methods can be used to change number of workers:

- fg.getSolver().getMultithreadingManager().getNumWorkers()
- fg.getSolver().getMultithreadingManager().setNumWorkers(num)

- fg.getSolver().getMultithreadingManager().setNumWorkersToDefault()

The following global methods can be used to set the number of threads in the ThreadPool

- ThreadPool.getNumThreads()
- ThreadPool.setNumThreads(numThreads)
- ThreadPool.setNumThreadsToDefault()

5.6.5 Common Belief Propagation Options

There are a number of options that are applicable to multiple solvers that are based on some form of message-passing belief propagation. These include the Sum-Product, Min-Sum, Particle BP, and Junction Tree solvers. These options are defined in the BPOptions class. The following options are supported:

5.6.5.1 BPOptions.iterations

Type	integer
Default	1
Affects	graph
Description	Controls how many iterations to perform when running solve(). This is not applicable to all solvers. It is currently only used by the SumProduct, MinSum and ParticleBP solvers. It only makes sense to set this to a value greater than one if the graph is not singly connected or "loopy", that is when there is more than one unique path between two or more nodes in the graph. You can tell if a graph is loopy using the FactorGraph method isForest(), which will be false if the graph is not singly connected.

5.6.5.2 BPOptions.damping

Type	double
Default	0.0
Affects	variables and factors
Description	The belief propogation based solvers supports damping, in which messages are damped by replacing each message by a weighted sum of the computed message value and the previous value of that message (when the corresponding edge was most-recently updated). In the current version of Dimple, damping is supported only in discrete variables and factors that connect only to discrete variables[25].

The damping parameter specifies a weighting value in the range 0 through 1:

$$\text{message} = \text{computedMessage} \cdot (1 - D) + \text{previousMessage} \cdot D$$

where D is the damping value. So that a value of 0 means that the previous message will not be considered, effectively turning off damping.

This option applies the same damping parameter to all edges connected to the variable or factor on which it is set. If you want different values for different edges, you need to use the BPOptions.nodeSpecificDamping option.

5.6.5.3 BPOptions.nodeSpecificDamping

Type	OptionDoubleList
Default	*empty*
Affects	variables and factors
Description	This is the similar to the BPOptions.damping option but allows you specify different weights for different edges. Unlike the simple damping option, this usually makes no sense to set on the graph itself since factors and variables will typically have different numbers and arrangements of edges. The value must either be an empty list, indicating that damping should be turned off, or a list of weights with the same length as the number of siblings of the affected variable and factor. The damping weights will be applied in the order in which the siblings are declared. This option takes precedence over the simple damping option if both are specified for the same node.

5.6.5.4 BPOptions.maxMessageSize

Type	integer
Default	integer max
Affects	discrete factors
Description	This specifies the maximum size of the outgoing messages on the discrete factors on which it is set. If this number K is less than the full size of the message (i.e. the size of the domain of the target variable), then only the K-best values – those with the largest weights – will be included in the message. This can results in a faster but more approximate form of inference and is most suited to graphs with very large-dimension variables. IMPORTANT: k-best and damping are not compatible with each other[26]

5.6.5.5 BPOptions.updateApproach

Type UpdateApproach enum
Default AUTOMATIC
Affects discrete factors
Description This option controls which update algorithm is applied to discrete factors. The option can take one of three values:

- NORMAL - Perform updates using just the factor tables. Do not use the optimized update technique.
- OPTIMIZED - Use the optimized update algorithm. Note that factors that have only one edge, or factors that do not have all of their edges updated simultaneously by the schedule, ignore this setting and use the normal approach.
- AUTOMATIC - Automatically determine whether to use the optimized algorithm. The automatic selection algorithm can be tuned through the BPOptions.automaticExecutionTimeScalingFactor and BPOptions.automaticMemoryAllocationScalingFactor options.

5.6.5.6 BPOptions.automaticExecutionTimeScalingFactor

Type double
Default 1.0
Affects discrete factors
Description This option is an execution time scaling factor used when the BPOptions.updateApproach option is set to AUTOMATIC. It controls how execution time costs are weighed. The value must be a positive number.

5.6.5.7 BPOptions.automaticMemoryAllocationScalingFactor

Type double
Default 10.0
Affects discrete factors
Description This option is an memory allocation scaling factor used when the BPOptions.updateApproach option is set to AUTOMATIC. It controls how memory allocation costs are weighed. The value must be a positive number.

5.6.5.8 BPOptions.optimizedUpdateSparseThreshold

Type	double
Default	1.0
Affects	discrete factors
Description	This option controls the representation of the auxiliary tables used by the optimized update algorithm, which is controlled through the BPOptions.updateApproach. Internally, the optimized algorithm creates multiple factor tables to perform the update. This option specifies a density, below which an auxiliary table uses a sparse representation. It must be a number in the range [0.0, 1.0]. The value 1.0 (the default), indicates that a sparse representation should be used if there are any zero-entries in the table. The value 0.0 will prevent the sparse representation from being used entirely. Sparse tables typically decrease execution time, but they use more memory. When the update approach is set to AUTOMATIC, this option impacts both the execution time and memory allocation estimates used to choose the update approach.

5.6.6 Sum-Product Solver

Use of the sum-product solver is specified by calling:

```
SumProductSolverGraph sfg = fg.setSolverFactory(new SumProductSolver());
```

If no solver is specified, the SumProduct solver is used by default.

The SumProduct solver supports both discrete and continuous variables. The SumProduct solver uses the sum-product form of belief propagation to perform inference on a graph. For discrete variables, each message to or from a factor is in the form of a vector of length equal to the domain size of the variable. For continuous variables, messages are represented using a Gaussian parameterization. In some cases, this is an approximation to the exact message. For Real variables, a message is in the form of a pair of values representing the mean and variance of the corresponding Normal distribution. For Complex and RealJoint variables, a message is in the form of a vector and matrix, representing the mean and covariance of the corresponding multivariate Normal distribution.

While the Gaussian representation of messages for continuous variables is sometimes an approximation, there are some specific built-in factors for which exact Gaussian messages are computed. This can be done when a factor preserves the Gaussian form of the distribution on each edge. The following table is a list of such lists built-in factors. See section 5.9 for more information on built-in factors.

Built-in Factor	Variable Types	Notes
Normal	Real	Applies only if mean and precision parameters are constants and all connected variables are Real and unbounded[27].
MultivariateNormal	Complex or RealJoint	All connected variables must be RealJoint or Complex and unbounded.
Sum	Real	All connected variables must be Real and unbounded.
Subtract	Real	All connected variables must be Real and unbounded.
Negate	Real	All connected variables must be Real and unbounded.
ComplexSum	Complex	All connected variables must be Complex and unbounded.
ComplexSubtract	Complex	All connected variables must be Complex and unbounded.
ComplexNegate	Complex	All connected variables must be Complex and unbounded.
RealJointSum	RealJoint	All connected variables must be RealJoint of the same dimension and unbounded.
RealJointSubtract	RealJoint	All connected variables must be RealJoint of the same dimension and unbounded.

[27]Unbounded means that the domain of the variable must not have finite upper or lower bounds

Built-in Factor	Variable Types	Notes
RealJointNegate	RealJoint	All connected variables must be RealJoint of the same dimension and unbounded.
Product	Real, Constant	Applies only if the product is one unbounded Real variable times one scalar constant to produce an unbounded Real variable.
MatrixRealJoint VectorProduct	RealJoint, Constant	Applies only if the product is one unbounded RealJoint variable times a constant matrix to produce an unbounded RealJoint variable.
LinearEquation	Real	Linear equation. All connected variables must be Real and unbounded.

For factors that are neither discrete-only or listed in the above table, an approximate computation is used in computing messages from such a factor. This includes any factor that connects to both discrete and continuous variables as well as factors that connect only to continuous variables but do not appear in the list above. The approximate method is sample based and uses Gibbs sampling to sample from the factor, allowing approximate messages to be computed from the sample statistics. Several methods described below allow control over the behavior of these sampled factors.

For discrete-only factors, two factor update algorithms are available: normal and optimized. The optimized algorithm can be applied only to factors with more than one edge, and only when the schedule updates all of the factor's edges simultaneously. The optimized algorithm computes the outbound message update with fewer operations than the normal algorithm, which can decrease execution time; however, it also uses more memory and increases initialization time. Several options, described below, influence which algorithm is used. Key among them is the updateApproach option, which can be set to normal, optimized, or automatic. When set to automatic, Dimple makes an estimate of the memory usage and execution time of each algorithm in order to select one.

5.6.6.1 GibbsOptions for Sampled Factors

Factors connected to continuous variables that do not support exact message computation, instead use a sampled approximation (see section 5.6.6) where the sampling is performed using the Gibbs solver.

For all such factors in a graph, you may set any of the Gibbs solver options described in paragraph 5.6.9.1 to control how the sampling will be done. The most important of these options have different default values when used with Sum-Product. This is accomplished by setting the options on the solver graph object when it is constructed. In order to override these defaults, it is necessary to set them on the solver graph (to apply to all such factors, using `graph.Solver.setOption(...)`), or on the factor specific factor object (to apply to a single factor, using `factor.setOption(...)`).

These options and their SumProduct-specific default values are:

- GibbsOptions.numSamples: 1000 (See SampledFactor.DEFAULT_SAMPLES_PER_UPDATE)
- GibbsOptions.burnInScans: 10 (See SampledFactor.DEFAULT_BURN_IN_SCANS_PER_UPDATE)
- GibbsOptions.scansPerSample: 1 (See SampledFactor.DEFAULT_SCANS_PER_SAMPLE)

5.6.7 Min-Sum Solver

Use of the MinSum solver is specified by calling:

```
SFactorGraph sgraph = fg.setSolverFactory(new MinSumSolver());
```

Unlike the Sum-Product solver, the Min-Sum solver supports only discrete variables. It only uses the standard BP Options described in Common Belief Propagation Options.

5.6.8 Junction Tree Solver

There are two distinct forms of Junction Tree solver in Dimple: the sum-product form used for computing exact marginal variable beliefs and the min-sum form used for computing MAP values. The Junction Tree solvers support only discrete variables.

Use of one of the two forms of Junction Tree solver by calling either:

```
JunctionTreeSolverGraph sgraph =
    fg.setSolverFactory(new JunctionTreeSolver());
```

for the sum-product version or

```
JunctionTreeMAPSolverGraph sgraph =
    fg.setSolverFactory(new JunctionTreeMAPSolver());
```

for the min-sum version.

The Junction Tree solvers are useful for performing exact inference on loopy discrete graphical models for which the standard sum-product or min-sum algorithms will only produce approximate results. This works by transforming the model into a corresponding non-loopy model, building a proxy solver layer that connects the original model to the transformed version and doing inference on that model. If your model already is non-loopy then you can simply use sum-product or min-sum directly for exact inference. To test to see if a graph is loopy or not, use isForest():

```
useJunctionTree = !fg.isForest();
```

One significant limitation when using this solver is that the cost of inference and amount of memory needed to store the factor tables is proportional to the size of the tables, which in turn is exponential in the number of variables represented in a table. The Junction Tree algorithm may be unable to determine an equivalent tree structure that has small enough factors either to fit in memory or to perform inference on in an acceptable amount of time. The typical failure mode in such cases is to get an OutOfMemoryError when attempting to run solve. The Junction Tree algorithm works best when used with smaller graphs or larger graphs that have few loops or are loopy but long and narrow.

5.6.8.1 Junction Tree Options

The following options are available for use with both versions of the Junction Tree solver.

Because exact inference can be done in a single iteration, the Junction Tree solver fixes the number of iterations to one and will ignore attempts to set it to another value. Because damping would result in inexact inference, the Junction Tree solver does not provide options for using damping.

5.6.8.1.1 JunctionTreeOptions.useConditioning

Type	boolean
Default	false
Affects	graph
Description	Specifies whether to use conditioning when constructing the transformation of the model. When true, then any variables in the model that have a fixed value will be disconnected from the rest of the graph in the transformed version and its value will be incorporated in the factors of the transformed model. This will produce a more efficient transformed model when there are fixed values in the original model. Using this will require that a new transformation be computed every time a fixed value changes.

5.6.8.1.2 JunctionTreeOptions.maxTransformationAttempts

Type	integer
Default	1
Affects	graph
Description	Specifies the maximum number of times the junction tree transformer should try to determine an optimal transformation. Each attempt uses a greedy "variable elimination" algorithm using a randomly chosen cost function and random choices to break ties, so more iterations could produce a more efficient tree transformation.

5.6.9 Gibbs Solver

Use of the Gibbs solver is specified by calling:

```
GibbsSolverGraph sfg = fg.setSolverFactory(new GibbsSolver());
```

The Gibbs solver supports both discrete and continuous variables.

This solver performs Gibbs sampling on a factor graph. It supports a variety of output information on the sampled graph, including the best joint sample (lowest energy), marginals of each variable (discrete variables only), and a full set of samples for a user-selected set of variables. The solver supports both sequential and randomized scan, and it supports tempering with an exponential annealing schedule.

The Gibbs solver supports several user selectable generic samplers (those that don't require specific conjugacy relationships). The following table lists the available generic samplers, and the variable types supported by each.

Sampler	Variable Type	Description
CDFSampler	Discrete[28]	Samples from the full conditional distribution of the variable. This is the default sampler for discrete variables.
SliceSampler	Real[29]	Slice sampler using the doubling procedure. See Neal, Slice Sampling (2000). This is the default sampler for real variables.
MHSampler	Discrete & Real	Metropolis-Hastings sampler. For discrete variables, the default proposal kernel is uniform over values other than the current value. For real variables, the default proposal kernel is Normal with standard deviation 1 (the standard deviation is user settable). Alternate proposal kernels are also available (see below).
SuwaTodoSampler	Discrete	Suwa-Todo sampler. See Suwa, Todo, Markov Chain Monte Carlo Method without Detailed Balance (2010).
BlockMHSampler	Discrete & Real	Block Metropolis-Hastings sampler. Allows block proposals for a collection of more than one variable at a time. In the current version of Dimple, there are no built-in general purpose block proposal kernels. To use this sampler, a custom block proposal kernel must be written in Java, and used by specifying a block schedule entry that references this proposal kernel. See section 5.6.9.8.1 for more information.

[28]In this table, Discrete support implies any of the discrete variable types, including Discrete and Bit.

[29]In this table, Real support implies any of the continuous variable data types, including Real, Complex, and RealJoint.

In cases where the factors of the graph support a conjugate distribution, the solver will automatically determine this and use the appropriate conjugate sampler. The following table lists the supported conjugate samplers and the corresponding factors that support them. The corresponding sampler will be used for a given variable if all of the edges connected to that variable support the same sampler[30].

Sampler	Built-in Factor	Edge
BetaSampler	Beta	value
	Binomial	ρ
DirichletSampler	Dirichlet	value
	Categorical	α
	DiscreteTransition	α
GammaSampler	Gamma	value, β
	NegativeExpGamma	β
	Normal	τ
	LogNormal	τ
	Poisson	λ
	CategoricalUnnormalizedParameters	α
	DiscreteTransitionUnnormalizedParameters	α
NegativeExpGammaSampler	NegativeExpGamma	value
	CategoricalEnergyParameters	α
	DiscreteTransitionEnergyParameters	α
NormalSampler	Normal	value, μ
	LogNormal	μ

Additionally, conjugate sampling is supported across a subset of applicable deterministic functions. In the current version of Dimple, this includes the following factors:

Factor Function	Edges	Condition
Multiplexer	in, out	If any of the *in* variables would support the same conjugate sampler as the *out* variable, then those *in* variables will use conjugate sampling.

The Gibbs solver automatically performs block-Gibbs updates for variables that are deterministically related. The Gibbs solver automatically detects deterministic relationships associated with built-in deterministic factor functions (see section 5.9 for a list of these functions).

For user-defined factors specified by factor functions or factor tables, the Gibbs solver will detect if they are deterministic functions as along as the factor is marked as the directed outputs are indicated using the DirectedTo property, as described in section 5.3.1.1.3.

The Gibbs solver also automatically performs block-Gibbs updates for certain built-in factors that Gibbs sampling on individual variables would fail due to the dependencies between variables imposed by the factor. In these cases, a custom proposal distribution ensures that

[30]Additionally, for the conjugate sampler to be used, the domain of the variable must not be bounded to a range smaller than the natural range of the corresponding distribution.

proposals are consistent with the constraints of the factor. However, the custom proposals are not assured to result in efficient mixing. In the current version of Dimple, the following built-in factors automatically implement block-Gibbs updates:

- Multinomial
- MultinomialUnnormalizedParameters
- MultinomialEnergyParameters

Most Dimple methods work more-or-less as normal when using the Gibbs solver, but in some cases the interpretation is slightly different than for other solvers. For example, the .Belief method for a discrete variable returns an estimate of the belief based on averaging over the sample values.

NOTE: The setNumIterations() method is not supported by the Gibbs solver as the term "iteration" is ambiguous in this case. Instead, the method setNumSamples() should be used to set the length of the run. The Solver.iterate() method performs a single-variable update in the case of the Gibbs solver, rather than an entire scan of all variables.

The following sections list the solver-specific aspects of the API for the Gibbs solver.

5.6.9.1 Gibbs Options

The following options affect the behavior of various aspects of the Gibbs solver:

5.6.9.1.1 GibbsOptions.numSamples

Type	integer
Default	1
Affects	graph
Description	Specifies the number of samples to be generated when solving the graph post burn-in. (This value times the value of GibbsOptions.numRandomRestarts plus one determines the total number of samples that will be produced.)

5.6.9.1.2 GibbsOptions.scansPerSample

Type	integer
Default	0
Affects	graph
Description	Specifies sampling rate in terms of the number of scans[31] of the graph to perform for each sample.

5.6.9.1.3 GibbsOptions.burnInScans

Type	integer
Default	0
Affects	graph
Description	Specifies the number of scans of the graph to perform during the burn-in phase before generating samples.

5.6.9.1.4 GibbsOptions.numRandomRestarts

Type	integer
Default	0
Affects	graph
Description	Specifies the number of random restarts (zero by default, which means run once and don't restart). For a value greater than zero, the after running the specified number of samples, the solver is restarted with the variable values randomized, and re-run (including burn-in). The sample values (the best sample value, or all samples, if requested) are extracted across all runs.

5.6.9.1.5 GibbsOptions.saveAllSamples

Type	boolean
Default	false
Affects	graph
Description	Specifies whether to save all sample values for variables when running Gibbs. Note that this is practical only if the number of variables in the graph times the number of samples per variable is reasonably sized.

5.6.9.1.6 GibbsOptions.saveAllScores

Type	boolean
Default	false
Affects	graph
Description	Specifies whether to save scores for all generated samples in Gibbs. If true, then for each sample the total energy/log-likelihood a.k.a. *score* of the graph will be saved. The saved scores can later be retrieved by the getAllScores() method described below.

5.6.9.1.7 GibbsOptions.discreteSampler

Type	IGenericSampler class
Default	CDFSampler
Affects	variables
Description	Specifies the default sampler to use for discrete variables when a conjugate sampler is not suitable. The sampler may be configured by use of additional options defined by each sampler type. See subsubsection 5.6.13 for more details.

5.6.9.1.8 GibbsOptions.realSampler

Type	IGenericSampler class
Default	SliceSampler
Affects	variables
Description	Specifies the default sampler to use for real-valued variables (including Complex and RealJoint) when a conjugate sampler is not suitable. The sampler may be configured by use of additional options defined by each sampler type. See subsubsection 5.6.13 for more details.

5.6.9.1.9 GibbsOptions.enableAutomaticConjugateSampling

Type	boolean
Default	true
Affects	variables
Description	Specifies whether to use conjugate sampling when available for a given variable. Note that if a specific sampler has been specified for a particular variable (by setting the GibbsOptions.realSampler option directly on the model or solver variable object) then a conjugate sampler will not be used regardless.

5.6.9.1.10 GibbsOptions.computeRealJointBeliefMoments

Type	boolean
Default	false
Affects	variables
Description	Specifies whether to compute the belief moments (mean vector and covariance matrix) for RealJoint (and Complex) variables while sampling. To minimize computation, these are not computed by default for RealJoint variables (Real variables always compute similar statistics, and do not have a corresponding option to enable them). If true, the belief moments are computed for each sample on-the-fly (without saving all samples). The computed moments can later be retrieved by the getSampleMean and getSampleCovariance solver-specific methods for the variable (see section 5.6.9.6)

5.6.9.1.11 GibbsOptions.enableAnnealing

Type	boolean
Default	false
Affects	graph
Description	Specifies whether to use a tempering and annealing process when running Gibbs.

5.6.9.1.12 GibbsOptions.annealingHalfLife

Type	double
Default	1.0
Affects	graph
Description	Specifies the rate at which the temperature will be lowered during the tempering and annealing process. This rate is specified in terms of the number of samples required for the temperature to be lowered by half. This value is only used if annealing has been enabled as specified by the enableAnnealing option.

5.6.9.1.13 GibbsOptions.initialTemperature

Type	double
Default	1.0
Affects	graph
Description	Specifies the initial temperature to use when annealing is enabled (as specified by the enableAnnealing option).

5.6.9.2 Graph Methods

The following methods are available on a graph set to use the Gibbs solver:

Variable initialization (both on the first run and subsequent restarts) is randomized whenever possible. For a discrete variable, the value is sampled from the Input (uniform if an input is not specified). For a real variable, if an Input is specified and the Input supports one of the conjugate samplers listed above, that sampler is used to initialize the variable. If bounds are also specified for the variable domain, the values is truncated to fall within the bounds. If only bounds are specified (which are finite above and below), then the value is uniformly sampled from within the bounds. If no finite bounds are specified and there is no input, the variable is initialized to zero (or the value specified by setInitialSampleValue) on the initial run, and left at the final value of the previous run on restart.

Enable or disable the use of tempering, or determine if tempering is in use.

```
sfg.setTemperature(T);
sfg.getTemperature();
```

Set/get the current temperature. Setting the current temperature overrides the current annealing temperature.

```
sfg.getAllScores();
```

Returns an array including the score value for each sample. This method only returns a non-empty value if the GibbsOptions.saveAllScores option was set to true on the graph before generating samples.

```
sfg.getTotalPotential();
```

After running the solver, returns the total potential (score) over all factors of the graph (including input priors on variables) given the most recent sample values.

```
sfg.sample(numSamples)
```

This method runs a specified number of samples without re-initializing, burn-in, or random-restarts (this is distinct from iterate(), which runs a specified number of single-variable updates). Before running this method for the first time, the graph must be initialized using the initialize() method.

```
sfg.burnIn()
```

Run the burn-in samples independently of using solve (which automatically runs the burn-in samples). This may be run before using sample() or iterate().

```
sfg.getRejectionRate()
```

Get the overall rejection rate over the entire graph. The rejection rate is the ratio of the number of MCMC proposals that were rejected to the total number of sampler updates. Rejections only occur in MCMC samplers, such as the MHSampler. In other samplers, such as conjugate samplers or the CDFSampler, rejection doesn't occur and the rate for variables that use these samplers is zero (in these cases, sampling the same value twice in a row is not considered rejection). When getting the rejection rate for the entire graph, both the number of rejections and number of updates is counted for all variables, as well as all blocks of variables over which a block sampler is used. These counts are accumulated from the time the graph is initialized, which automatically occurs when running solve(). This includes both burn-in and subsequent sampling. These counts can also be reset explicitly using the resetRejectionRateStats method (see below), which allows these values to be determined, for example, during a specific set of samples.

```
sfg.resetRejectionRateStats()
```

Explicitly reset the rejection-rate statistics. These are automatically reset when the graph is initialized, which automatically occurs when running solve(), but may be reset manually at other times using this method.

5.6.9.3 Variable Methods

```
Object d = ((GibbsDiscrete)variable.getSolver()).getCurrentSample();
double r = ((GibbsReal)variable.getSolver()).getCurrentSample();
double[] rj = ((GibbsRealJoint)variable.getSolver()).getCurrentSample();
Value val= ((ISolverVariableGibss)variable.getSolver()).
    getCurrentSampleValue();
```

Returns the current sample value for a variable.

```
Object[] ds = ((GibbsDiscrete)variable.getSolver()).getAllSamples();
double[] rs = ((GibbsReal)variable.getSolver()).getAllSamples();
double[][] rjs = ((GibbsRealJoint)variable.getSolver()).getAllSamples();
```

Returns an array including all sample values seen so far for a variable. Over multiple variables, samples with the same index correspond to the same joint sample value. This method only returns a non-empty value if the GibbsOptions.saveAllSamples options was enabled for the variable when sampling was performed.

```
Object d = ((GibbsDiscrete)variable.getSolver()).getBestSample();
double r = ((GibbsReal)variable.getSolver()).getBestSample();
double[] rj = ((GibbsRealJoint)variable.getSolver()).getBestSample();
```

Returns the value of the best sample value seen so far, where best is defined as the sample with the minimum total potential over the graph (sum of -log of the factor values and input priors). When getting the best sample from multiple variables, they all correspond to the same sample in time, thus should be a valid sample from the joint distribution.

```
variable.getSolver().setInitialSampleValue(initialSampleValue)
variable.getSolver().getInitialSampleValue()
```

Set/get the initial sample value to be used as the starting value for this variable. This value is used only on the first run (not subsequent restarts). Setting this value overrides any randomization of the starting value on the first run.

```
variable.getSolver().setSampler(samplerName);
variable.getSolver().getSampler();
variable.getSolver().getSamplerName();
```

Set/get the sampler to be used for this variable. Setting the sampler for a given variable overrides the default sampler for the given variable type, and also overrides any conjugate sampler that might otherwise be used. Using this method the sampler may be set only to one of the generic samplers appropriate for the given variable type.

The getSampler method returns the sampler object, while the getSamplerName method returns a string indicating the name of the sampler being used for this variable. Automatic assignment of a conjugate sampler is done at graph initialization time, so in order to determine what sampler will actually be used, these methods must be called either after a call to the graph initialize method, or after running the solver.

```
variable.getSolver().getRejectionRate()
```

Get the rejection rate for the sampler used for a specific variable. The rejection rate is the ratio of the number of MCMC proposals that were rejected to the total number of sampler updates. Rejections only occur in MCMC samplers, such as the MHSampler. In other samplers, such as conjugate samplers or the CDFSampler, rejection doesn't occur and the rate for variables that use these samplers is zero (in these cases, sampling the same value twice in a row is not considered rejection). These counts are accumulated from the time the graph is initialized, which automatically occurs when running solve(). This includes both burn-in and subsequent sampling. These counts can also be reset explicitly using the resetRejectionRateStats method (see below), which allows these values to be determined, for example, during a specific set of samples.

```
variable.getSolver().getNumScoresPerUpdate()
```

Returns the average number of score computations performed per update when sampling from this variable. Use of an MCMC sampler requires computation of the score (energy) for specific settings of the variables. For some samplers, such as the slice sampler, the number of times the score is computed varies, and depends on the particular values and the form of the distribution. The returned value indicates the average number of times the score has been computed. If a non-MCMC-based sampler is used, the returned value will be zero. The count is accumulated from the time the graph is initialized, which automatically occurs when running solve(). This includes both burn-in and subsequent sampling. The count can also be reset explicitly using the resetRejectionRateStats method (see below), which allows this value to be determined, for example, during a specific set of samples.

```
variable.getSolver().resetRejectionRateStats()
```

Explicitly reset the rejection-rate statistics for a specific variable (the statistics for computing the rejection rate as well as the number of scores per update). These are automatically reset when the graph is initialized, which automatically occurs when running solve(), but may be reset manually at other times using this method.

5.6.9.4 Discrete-Variable-Specific Methods

The following methods apply only to Discrete, Bit, and FiniteField variables when using the Gibbs solver.

```
variable.getSolver().getSampleIndex();
```

Returns the index of the current sample for a variable, where the index refers to the index into the domain of the variable.

```
variable.getSolver().getAllSampleIndices();
```

Returns an array including the indices of all samples seen so far for a variable.

```
variable.getSolver().getBestSampleIndex();
```

Returns the index of the best sample seen so far.

```
variable.getSolver().setInitialSampleIndex(initialSampleIndex)
variable.getSolver().getInitialSampleIndex()
```

Set/get the initial sample index associated with the starting value for this variable. The value associated with this index is used only on the first run (not subsequent restarts). Setting this index overrides any randomization of the starting value on the first run.

5.6.9.5 Real-Variable-Specific Methods

The following methods apply only to Real variables when using the Gibbs solver.

```
variable.getSolver().getSampleMean();
```

Returns the mean value of all samples that have been collected. This is and estimate of the mean of the belief for the corresponding variable.

```
variable.getSolver().getSampleVariance();
```

Returns the variance of all samples that have been collected. This is and estimate of the variance of the belief for the corresponding variable.

5.6.9.6 RealJoint-Variable-Specific Methods

The following methods apply only to RealJoint and Complex variables when using the Gibbs solver.

```
variable.getSolver().getSampleMean();
```

Returns the mean vector of all samples that have been collected. This is and estimate of the mean of the belief for the corresponding variable. This method is only available if, prior to performing inference, the option GibbsOptions.computeRealJointBeliefMoments is set to true.

```
variable.getSolver().getSampleCovariance();
```

Returns the covariance matrix computed over all samples that have been collected. This is and estimate of the covariance of the belief for the corresponding variable. This method is only available if, prior to performing inference, the option GibbsOptions.computeRealJointBeliefMoments is set to true.

5.6.9.7 Factor Methods

```
factor.getSolver().getPotential();
```

Returns the potential value of a factor given the current values of its connected variables.

```
factor.getSolver().getPotential(values);
```

Get the potential value of a factor given the variable values specified by the argument vector. The argument must be a vector with length equal to the number of connected variables. For a table-factor (connected exclusively to discrete variables), each value corresponds the index into the domain list for that variable (not the value of the variable itself). For a real-factor (connected to one or more real variables), each value corresponds to the value of the variable.

5.6.9.8 Schedulers and Schedules

The built-in schedulers designed for belief propagation are not appropriate for the Gibbs solver. Instead, there are two built-in schedulers specifically for the Gibbs solver:

- GibbsSequentialScanScheduler
- GibbsRandomScanScheduler

The GibbsSequentialScanScheduler chooses the next variable for updating in a fixed order. It updates all variables in the graph, completing an entire scan, before repeating the same fixed order. (In Gibbs literature this seems to be known as a sequential-scan, systematic-scan, or fixed-scan schedule.)

The GibbsRandomScanScheduler randomly selects a variable for each update (with replacement).

The default scheduler when using the Gibbs solver is the GibbsSequentialScanScheduler, which is used if no scheduler is explicitly specified.

The user may specify a custom schedule when using the Gibbs solver. In this case, the schedule should include only Variable node updates (not specific edges), and no Factor updates (any Factor updates specified will be ignored).

To explicitly specify a scheduler, use the Scheduler or Schedule property of the FactorGraph (see sections 5.1.2.2 and 5.1.2.3).

5.6.9.8.1 Block Schedule Entries

The Gibbs solver allows a schedule to optionally include block entries that allow a group of variables to be updated at once. A block schedule entry can either be included in a custom schedule or added to the schedule produced by one of the Gibbs-specific built-in schedulers. A block schedule entry includes two pieces of information:

- A reference to a block sampler, which is used to perform the update
- A list of variables to be included in this block

In the current version of Dimple, the only built-in block sampler is the BlockMHSampler, which implements block Metropolis-Hastings sampling for the variables included in the block. The BlockMHSampler requires a block proposal kernel to be specified. In the current version of Dimple, there are no built-in general purpose block proposal kernels. To use this sampler, a custom block proposal kernel must be written (see section B.2).

To add a block schedule entry to a custom schedule, the the argument to the `add` method is a reference to a sampler object, and the subsequent entries are the variables to be included in the block. For example:

```
FixedSchedule s = new FixedSchedule();
s.add(new BlockScheduleEntry(new BlockMHSampler(new MyProposalKernel()), a,
    b, c));
s.add(new NodeScheduleEntry(d));
s.add(new NodeScheduleEntry(e));
fg.setSchedule(s);
```

In the above example, we create a block schedule entry that updates variables a, b, and c together, with separate schedule entries for variables d and e. The constructor for the BlockMHSampler requires a proposal kernel. In the above example, "MyProposalKernel" is a user-provided custom proposal class .

Block schedule entries can also be used with either of the Gibbs-specific built-in schedulers described above. When a block entry is added in this way, for each of the variables included in a block entry, the individual variable entries that would have been present in the schedule are removed. That is, those variables are only included in the corresponding block entry (or entries) and are not also updated independently. In case of the GibbsRandomScanScheduler, each update selects an entry randomly from among all blocks plus all variables that are not in a block.

A block schedule entry can be added when using a built-in Gibbs-specific scheduler using:

```
((IGibbsScheduler)fg.getScheduler()).addBlockScheduleEntry(new
    BlockScheduleEntry(blockSampler, listOfVariables));
```

The following example shows adding a block schedule entry that includes the variables a, b, and c.

```
((IGibbsScheduler)fg.getScheduler()).addBlockScheduleEntry(new
    BlockScheduleEntry(new BlockMHSampler(new MyProposalKernel()), a, b, c))
    ;
```

Note that to use the addBlockScheduleEntry method, the scheduler must first have been explicitly set to one of the Gibbs-specific schedulers.

To implement a custom block proposal kernel, a new Java class must be created that implements the IBlockProposalKernel interface. See section B.2 for more detail.

5.6.10 Particle BP Solver

Use of the particle BP solver is specified by calling:

```
ParticleBPSolverGraph sfg = fg.setSolverFactory(new ParticleBPSolver());
```

The following lists the solver-specific options for the ParticleBP solver.

5.6.10.1 Particle BP Options

The following options affect the behavior of various aspects of the ParticleBP solver:

5.6.10.1.1 ParticleBPOptions.numParticles

Type	integer
Default	1
Affects	real variables
Description	Specifies the number of particles used to represent the variable. This option takes affect when the solver variable is constructed and when it is initialized.

5.6.10.1.2 ParticleBPOptions.resamplingUpdatesPerParticle

Type	integer
Default	1
Affects	real variables
Description	For variables on which it is set, specifies the number of updates per particle to perform each time the particle is resampled.

5.6.10.1.3 ParticleBPOptions.iterationsBetweenResampling

Type	integer
Default	1
Affects	graph
Description	Specifies the number of iterations between re-sampling all of the variables in the graph. Default is 1, meaning resample between every iteration.

5.6.10.1.4 ParticleBPOptions.initialParticleRange

Type	two-element OptionDoubleList
Default	[-infinity infinity]
Affects	variables
Description	Set the range over which the initial particle values will be defined. The initial particle values are uniformly spaced between the min and max values specified. If the range is specified using this method, it overrides any other initial value. Otherwise, if a finite domain has been specified, the initial particle values are uniformly spaced between the lower and upper bound of the domain. Otherwise, all particles are initially set to zero.

```
// Set particle range to the unit interval for all variables in the graph:
ParticleBPOptions.initialParticleRange.set(graph, 0.0, 1.0);
```

5.6.10.1.5 ParticleBPOptions.proposalKernel

Type	IProposalKernel class
Default	NormalProposalKernel
Affects	real variables
Description	Specifies the type of proposal kernel to use for the specified variables. The selected proposal kernel may have additional options that can be used to configure its behavior. These can also be set on the variables.

5.6.10.1.6 ParticleBPOptions.enableAnnealing

Type	boolean
Default	false
Affects	graph
Description	Determines whether to use a tempering and annealing process during inference.

5.6.10.1.7 ParticleBPOptions.annealingHalfLife

Type	double
Default	1.0
Affects	graph
Description	Specifies the rate at which the temperature will be lowered during the tempering and annealing process. This rate is specified in terms of the number of iterations required for the temperature to be lowered by half. This value is only used if annealing has been enabled as specified by the enableAnnealing option.

5.6.10.1.8 ParticleBPOptions.initialTemperature

Type	double
Default	1.0
Affects	graph
Description	Specifies the initial temperature to use when annealing is enabled (as specified by the enableAnnealing option).

5.6.10.2 Graph Methods

The following solver-specific methods are available on the solver graph. (It is assumed that sfg is a variable of type ParticleBPSolverGraph obtained when the solver was set or by casting the result of the getSolver() method.)

```
sfg.setTemperature(newTemperature);
double temp = sfg.getTemperature();
```

Set/get the current temperature. Setting the current temperature overrides the current annealing temperature. This should rarely be necessary.

5.6.10.3 Variable Methods

The Particle BP solver supports both discrete and real variables. For discrete variables, the solver uses sum-product BP as normal, and all of the corresponding methods for the sum-product solver may be used for discrete variables. For real variables, several solver-specific methods are defined, as follows.

5.6.10.4 Real-Variable-Specific Methods

```
variable.getSolver().setProposalStandardDeviation(stdDev);
variable.getSolver().getProposalStandardDeviation();
```

Set/get the standard deviation for a Gaussian proposal distribution (the default is 1).

```
((ParticleBPReal)variable.getSolver()).getParticleValues();
```

Returns the current set of particle values associated with the variable.

```
((ParticleBPReal))variable.getSolver()).getBelief(valueSet);
```

Given a set of values in the domain of the variable, returns the belief evaluated at these points. The result is normalized relative to the set of points requested so that the sum over the set of returned beliefs is 1.

NOTE: the generic variable method Belief (or getBeliefs() with no arguments) operates similarly to the discrete-variable case, but the belief values returned are those at the current set of particle values. Note that this representation does not represent a set of weighted particles. That is, the particle positions are distributed approximately by the belief and the belief values represent the belief. It remains to be see if this should be the representation of belief that is used, or if an alternative representation would be better. The alternative solver-specific getBelief(valueSet) method allows getting the beliefs on a user-specified set of values, which may be uniform, and would not have this unusual interpretation.

5.6.11 LP Solver

Use of the linear programming (LP) solver is specified by calling:

```
fg.setSolverFactory(new LPSolver());
```

The LP solver transforms a factor graph MAP estimation problem into an equivalent linear program, which is solved using a linear programming software package. The solver can either be a linear programming solver (in which case the MAP is estimated using an LP relaxation, with no guarantees of correctness), or by an integer linear programming (ILP) solver, in which case the solution is guaranteed to be the MAP. Because this solver release on an external package, you will need to install and configure the specified package before using this solver.

The LP solver supports only discrete variables.

In the current version of Dimple (version 0.7), there is no support for rolled-up graphs when using the LP solver.

The following options are applicable to the LP solver:

5.6.11.1 LP Options

The following options affect the behavior of various aspects of the LP solver:

5.6.11.1.1 LPOptions.LPSolver

Type	string
Default	‘’
Affects	graph
Description	Selects which external LP solver will be used to solve the linear program. Valid values include ‘matlab’, ‘CPLEX’, ‘GLPK’, ‘Gurobi’, ‘LpSolve’, ‘MinSate’, ‘Mosek’, and ‘SAT4J’. The default value is synonomous with specifying ‘matlab’ and will delegate the solver specified by the MatlabLPOption that will be run from the MATLAB frontend. This will obviously only work when running Dimple from MATLAB. None of these solvers are included with Dimple and must be installed and configured separately. The interface for the non-MATLAB based solvers is provided by the third-party Java ILP package. See javailp.sourceforge.net for more information about configuring various solvers.

5.6.11.1.2 LPSolver.MatlabLPSolver

Type	string
Default	''
Affects	graph
Description	Selects which LP solver will be run from MATLAB to solve the linear program. This option is only relevant if the LPSolver option has been set to 'matlab' and Dimple is being run from MATLAB. The choices for the string solvername are 'matlab', 'glpk', 'glpkIP', 'gurobi', and 'gurobiIP'. The 'matlab', 'glpk', and 'gurobi' solvers are linear programming solvers, while 'glpkIP' and 'gurobiIP' are ILP solvers. The default value is synonomous with 'matlab'. Using the matlab LP solver requires the the MATLAB Optimization Toolbox. Using 'glpk' or 'glpkIP' requires glpkmex to be in the matlab path, and 'gurobi' and 'gurobiIP' require the gurobi matlab interface to be in the matlab path; in either case the appropriate packages will need to be obtained and installed.

5.6.12 Proposal Kernels

The following proposal kernels are provided by Dimple for use in samplers. Additional kernels may be added by creating new proposal kernel Java classes that implement the appropriate interfaces.

5.6.12.1 NormalProposalKernel

The following options may be used to configure this kernel:

5.6.12.1.1 NormalProposalKernel.standardDeviation

Type	double
Default	1.0
Affects	variables
Description	Specifies the standard deviation to use on NormalProposalKernel instances attached to the variables that are affected by the option setting. The value must be non-negative.

5.6.12.2 CircularNormalProposalKernel

The CircularNormalProposalKernel makes proposals from a Normal distribution on a circularly wrapping range of the real line. For example, setting the bounds of the range to $-\pi$ and π would create proposals representing angles on a circle.

Since this is a subclass of NormalProposalKernel, the standardDeviation option defined for that class will also affect this one. The following additional options may also be used:

5.6.12.2.1 CircularNormalProposalKernel.lowerBound

Type	double
Default	$-\pi$
Affects	variables
Description	Specifies lower bound to use on CircularNormalProposalKernel instances attached to the variables that are affected by the option setting.

5.6.12.2.2 CircularNormalProposalKernel.upperBound

Type	double
Default	$+\pi$
Affects	variables
Description	Specifies upper bound to use on CircularNormalProposalKernel instances attached to the variables that are affected by the option setting.

5.6.12.3 UniformDiscreteProposalKernel

This kernel does not have any configurable options.

5.6.13 Samplers

Samplers are used by sampling solvers (e.g. Gibbs) to generate samples for variables either singly or in blocks.

The following non-conjugate single variable samplers are currently available:

5.6.13.1 CDFSampler

The CDFSampler can be used with discrete variable types. It samples from the full conditional distribution of the variable. It is the default sampler used for discrete variables in the Gibbs solver.

It does not support any configuration options.

5.6.13.2 MHSampler

The MHSampler may be used for discrete or real variables. It implements the Metropolis-Hastings sampling algorithm. The sampler may be configured to use different proposal kernels for discrete and real variables as described below.

The following options may be used to configure the MHSampler for a given variable:

5.6.13.2.1 MHSampler.discreteProposalKernel

Type	IProposalKernel class
Default	UniformDiscreteProposalKernel
Affects	variables
Description	Specifies the proposal kernel to use when using the MHSampler on discrete variables for which this option setting is visible. Depending on the kernel selected, it may also be configured by additional option settings. See subsection 5.6.12 for more details.

5.6.13.2.2 MHSampler.realProposalKernel

Type	IProposalKernel class
Default	NormalProposalKernel
Affects	variables
Description	Specifies the proposal kernel to use when using the MHSampler on real variables for which this option setting is visible. Depending on the kernel selected, it may also be configured by additional option settings. See subsubsection 5.6.12 for more details.

5.6.13.3 SliceSampler

The slice sampler may be used with real variables. It implements the algorithm described in Neal's paper "Slice Sampling" 2010. It is the default sampler used for real variables in the Gibbs solver.

The following options may be used to configure the SliceSampler for a given variable:

5.6.13.3.1 SliceSampler.initialSliceWidth

Type	double
Default	1.0
Affects	variables
Description	The size of the initial slice to use when sampling for SliceSampler instances used by variables that are affected by the option setting. For variables with a natural range that is much smaller or much larger than the default value of one, it may be beneficial to modify this value.

5.6.13.3.2 SliceSampler.maximumDoublings

Type	integer
Default	10
Affects	variables
Description	The maximum number of doublings used during the doubling phase of the slice sampler. The maximum slice interval is on the order of $initialSliceWidth \cdot 2^{maximumDoublings}$

5.6.13.4 SuwaTodoSampler

The Suwo-Todo sampler can be used for discrete variables. It implements the algorithm described in Suwa and Todo's paper "Markov Chain Monte Carlo Method without Detailed Balance" (2010).

This sampler does not support any configuration options.

5.7 Streaming Data

5.7.1 Variable Stream Common Properties and Methods

Dimple supports several types of variable streams:

- DiscreteStream
- BitStream
- RealStream
- RealJointStream
- ComplexStream

Each of these share common properties and method listed in the following sections.

5.7.1.1 Properties

5.7.1.1.1 DataSource

Read-write. When written, connects the variable stream to a data source (see section 5.7.8). The data sink must be of a type appropriate for the particular variable stream type.

5.7.1.1.2 DataSink

Read-write. When written, connects the variable stream to a data sink (see section 5.7.12). The data sink must be of a type appropriate for the particular variable stream type.

5.7.1.1.3 Size

Read-only. Indicates the number of elements in the variable stream that are actually instantiated. Each element corresponds to one copy of the variable or variable array at a specific point in the stream. Dimple instantiates the minimum number of contiguous elements to cover the slices of the stream that are actually used in factors (see section 5.7.1.2.1), plus the number of additional elements to cover the indicated BufferSize (see section 5.7.7.1.1).

5.7.1.1.4 Variables

Read-only. Returns a variable array containing all of the currently instantiated variables in the stream.

5.7.1.1.5 Domain

Read-only. Returns the domain in a form that depends on the variable type, as summarized in the following table:

Stream Type	Domain Data Type
DiscreteStream	DiscreteDomain (see section 5.2.9)
BitStream	DiscreteDomain (see section 5.2.9)
RealStream	RealDomain (see section 5.2.10)
RealJointStream	RealJointDomain (see section 5.2.11)
ComplexStream	ComplexDomain (see section 5.2.12)

5.7.1.2 Methods

5.7.1.2.1 getSlice

```
varStream.getSlice(startIndex);
```

The getSlice method is used to extract a *slice* of the stream, which means a version of the stream that may be offset from the original stream itself. This is generally used for specifying streams to connect to a factor when calling addFactor.

Takes a single numeric argument, startIndex, which indicates the starting position in the stream. The resulting stream slice is essentially a reference to the stream offset by startIndex. For example, a startIndex of 1 returns a slice offset by 1, such that the first location in the slice corresponds to the second location in the original stream. A startIndex of 0 returns a slice identical to the original stream.

5.7.2 DiscreteStream

5.7.2.1 Constructor

The DiscreteStream constructor is used to create a stream of Discrete variables or arrays of Discrete variables.

```
DiscreteStream(domain);
```

- domain is a required argument indicating the domain of the variable. The domain may either be a comma separated list, an object array, or a DiscreteDomain object (see section 5.2.3.1.1).

5.7.3 BitStream

5.7.3.1 Constructor

The BitStream constructor is used to create a stream of Bit variables or arrays of Bit variables.

```
BitStream();
```

5.7.4 RealStream

5.7.4.1 Constructor

The RealStream constructor is used to create a stream of Real variables or arrays of Real variables.

```
RealStream();
RealStream(RealDomain domain)
RealStream(double lowerBound, double upperBound)
```

- domain specifies a bound on the domain of the variable. It can either be specified as two elements or a RealDomain object (see section 5.2.10). If specified as two values, the first element is the lower bound and the second element is the upper bound. Negative infinity and positive infinity are allowed values for the lower or upper bound, respectively. If no domain is specified, then a domain from $-\infty$ to ∞ is assumed.

5.7.5 RealJointStream

5.7.5.1 Constructor

The RealJointStream constructor is used to create a stream of RealJoint variables.

```
RealJointStream(int numJointVariables)
RealJointStream(RealJointDomain domain)
```

The arguments are defined as follows:

- numJointVariables specifies the number of joint real-valued elements.
- domain specifies the domain of the RealJoint variable using a RealJointDomain object (see 5.2.11). Using this version of the constructor allows bounds to be specified in some or all dimensions of the domain.

5.7.6 ComplexStream

5.7.6.1 Constructor

The ComplexStream constructor is used to create a stream of Complex variables.

```
ComplexStream()
ComplexStream(ComplexDomain domain)
```

The arguments are defined as follows:

- domain specifies the domain of the ComplexStream using a ComplexDomain object (see 5.2.12). Using this version of the constructor allows bounds to be specified in some or all dimensions of the domain.

5.7.7 FactorGraphStream

A FactorGraphStream is constructed automatically and returned as the result of adding a factor to a graph using the addFactor method where one or more of the arguments are variable streams.

5.7.7.1 Properties

5.7.7.1.1 BufferSize

Read-write. When written, modifies the number of instantiated elements in the FactorGraphStream to include the specified number of copies of the corresponding factor and connected variables. By default, the BufferSize is 1. When running the solver on one step of the overall factor graph, the solver uses the entire buffer. Making the buffer size larger means using more of the history in performing inference for each step. The results of inference run on previous steps that is beyond the size of the buffer is essentially frozen, and is no longer updated on subsequent steps of the solver.

5.7.8 Data Source Common Properties and Methods

Dimple supports several types of streaming data sources. A data source is a source of Input values to the variables within a variable stream[32]. When performing inference, as each step that the graph is advanced, the next source value is read from the data source, and the earlier values are shifted back to earlier time-steps in the graph.

Each source type corresponds to a particular format of input data. Each type is appropriate only to a specific type of variable stream and solver. The following table summarizes these requirements.

Data Source	Variable Stream Type	Supported Solvers
DoubleArrayDataSource	DiscreteStream, BitStream	all
	RealStream	SumProduct
MultivariateDataSource	RealJointStream	SumProduct
FactorFunctionDataSource	RealStream	SumProduct, Gibbs, ParticleBP

Each of these share common properties and method listed in the following sections.

5.7.8.1 Properties

5.7.9 DoubleArrayDataSource

5.7.9.1 Constructor

- DoubleArrayDataSource() - Create data source with no initial data
- DoubleArrayDataSource(ArrayList <double[] >initialData) - Specify some initial data.
- DoubleArrayDataSource(double [][] initialData) - Specify some initial data. The first dimension indexes the step and the second dimension indexes the single step distribution.

[32]In the current version of Dimple, data sources are limited to providing Inputs to variables. A future version of Dimple may expand this capability to allow sourcing FixedValues or other types of input data.

5.7.9.2 Methods

5.7.9.2.1 add

```
dataSource.add(data);
```

This method appends the data source with the specified data. The data argument is a multidimensional array, where the first dimensions correspond to the dimension of the variable stream this will feed, the next dimensions corresponds to the length of the Input vector for each variable (the domain size for discrete variable streams, and 2 for real variable streams used with the SumProduct solver), and the final dimension is the number of time-steps of data to provide. For single variable streams, the first dimensions are omitted.

5.7.10 MultivariateDataSource

5.7.10.1 Constructor

```
MultivariateDataSource();
```

5.7.10.2 Methods

5.7.10.2.1 add

- dataSource.add(MultivariateNormalParameters msg) - Add data for a single step.
- dataSource.add(MultivariateNormalParameters[] msgs) - Add multiple steps of data.
- dataSource.add(double [] means, double [][] covariances) - Add data for a single step.

5.7.11 FactorFunctionDataSource

5.7.11.1 Constructor

```
FactorFunctionDataSource();
```

5.7.11.2 Methods

5.7.11.2.1 add

```
dataSource.add(data);
```

This method appends the data source with the a single time-step of data (multiple time-steps must be added using successive calls to this method).The data argument is a multidimensional array, with dimension equal to the corresponding dimensions of the variable stream this will feed. Each element is a FactorFunctions (see section 5.3.3), which is to represent the Input of the corresponding variable.

5.7.12 Data Sink Common Properties and Methods

Dimple supports several types of streaming data sinks:

Dimple supports several types of streaming data sinks. A data sink is a data structure used to store successive results of inference from the variables with a variable stream. Specifically, it stores the Belief values of these variables[33]. When performing inference, as each step that the graph is advanced, the Belief value for the earliest element of the variable stream is stored in the data sink.

Each sink type corresponds to a particular format of output data. Each type is appropriate only to a specific type of variable stream and solver. The following table summarizes these requirements.

Data Sink	Variable Stream Type	Supported Solvers
DoubleArrayDataSink	DiscreteStream, BitStream	all
	RealStream	SumProduct
MultivariateDataSink	RealJointStream	SumProduct

Each of these share common properties and method listed in the following sections.

5.7.12.1 Properties

5.7.12.1.1 Dimensions

[33]In the current version of Dimple, data sinks are limited to the Beliefs to variables. A future version of Dimple may expand this capability to allow sinking other types of result data.

Read-only. Indicates the dimensions of the data sink. The dimensions correspond to the size of the variable array at each position in the stream that the data sink will be fed from.

5.7.12.2 Methods

5.7.12.2.1 hasNext

```
hasNext = dataSink.hasNext();
```

Used in connection with the getNext method (described in the sections below), this method takes no arguments and returns a boolean indicating whether or not there are any more time steps in the dataSink that have not yet been extracted.

5.7.13 DoubleArrayDataSink

5.7.13.1 Constructor

```
DoubleArrayDataSink();
```

5.7.13.2 Properties

5.7.13.2.1 Array

Read-only. Extracts the entire contents of the data sink as an array. The first dimensions of the array correspond to the Dimensions of the data sink and the final dimension corresponds to the number of time steps that had been gathered. For discrete variables, the dimension of the belief array corresponds to the domain sizes, while for real variables used with the SumProduct solver the dimension is 2, where the elements correspond to the mean and standard deviation, respectively.

5.7.13.3 Methods

5.7.13.3.1 getNext

```
double [] b = dataSink.getNext();
```

This method takes no arguments, and returns the belief values from the next time-step. The returned value is an array of beliefs for the variable for the given step. For discrete variables, the dimension of the belief array corresponds to the domain sizes, while for real variables used with the SumProduct solver the dimension is 2, where the elements correspond to the mean and standard deviation, respectively.

5.7.14 MultivariateDataSink

5.7.14.1 Constructor

```
MultivariateDataSink();
```

5.7.14.2 Methods

5.7.14.2.1 getNext

```
b = dataSink.getNext();
```

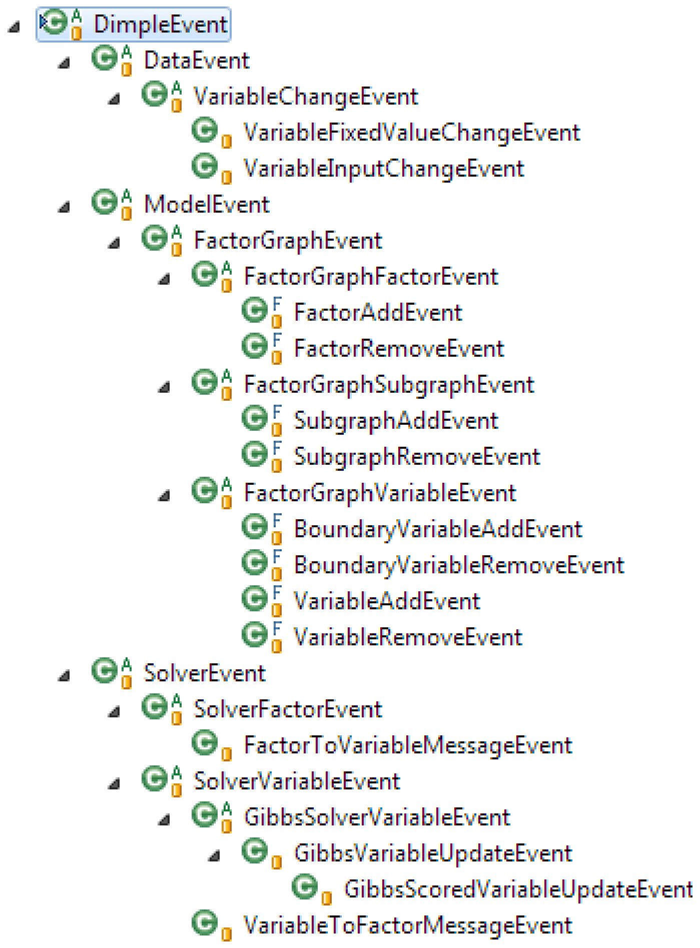
This method takes no arguments, and returns the belief values from the next time-step. The returned value is a MultivariateNormalParameters object (see section 5.2.15) that contains the mean vector and covariance matrix.

5.8 Event Monitoring

Sometimes it can be useful to monitor the actions Dimple takes as the model or data changes or as inference is performed. Such monitoring can be helpful when debugging your model, when trying to determine whether inference has converged while using belief propogation on a loopy graph, or when attempting to determine whether a graph has been adequately mixed when using the Gibbs solver. To address this need, Dimple provides an event-based system consisting of events that can be triggered when various actions of interest occur and associated event handlers and an event listener that handles dispatching of events. The event system is designed to have no effect on the performance of inference when it has not been enabled, but may have a noticeable effect when it is being used.

The full power of the event system is only available directly in the Java API but the MATLAB API does provide a simple event logging interface that allows events to be logged to the console or an external log file.

5.8.1 Event types



```
DimpleEvent
  DataEvent
    VariableChangeEvent
      VariableFixedValueChangeEvent
      VariableInputChangeEvent
  ModelEvent
    FactorGraphEvent
      FactorGraphFactorEvent
        FactorAddEvent
        FactorRemoveEvent
      FactorGraphSubgraphEvent
        SubgraphAddEvent
        SubgraphRemoveEvent
      FactorGraphVariableEvent
        BoundaryVariableAddEvent
        BoundaryVariableRemoveEvent
        VariableAddEvent
        VariableRemoveEvent
  SolverEvent
    SolverFactorEvent
      FactorToVariableMessageEvent
    SolverVariableEvent
      GibbsSolverVariableEvent
        GibbsVariableUpdateEvent
          GibbsScoredVariableUpdateEvent
      VariableToFactorMessageEvent
```

Figure 1: Dimple event hierarchy

Dimple Events are organized hierarchically, and the current version of the full hierarchy is shown in Figure 1. Note that any event types marked with an 'A' in the diagram are abstract super types and are used to organize the events by category. Actual event instances will belong to non-abstract types, which are for the most part leaf-nodes in the diagram. Events are subdivided into three categories:

- ModelEvent: includes changes to the structure of the model including adding and removing variables, factors and subgraphs. The concrete model event types are:
  - FactorAddEvent and FactorRemoveEvent: raised when a factor is added or removed from the model.
  - VariableAddEvent and VariableRemoveEvent: raised when a variable is added or removed from the model.
  - SubgraphAddEvent and SubgraphRemoveEven: raised when a subgraph is added or removed from the model.

    - BoundaryVariableAddEvent and BoundaryVariableRemoveEvent: raised when a boundary variable is added or removed.

- DataEvent: includes changes to data including changes to fixed values and inputs. The concrete data event types are:
    - VariableFixedValueChangeEvent: raised when a fixed value is set or unset on a variable.
    - VariableInputChangeEvent: raised when an input distribution is set or unset on a variable.
- SolverEvent: includes solver-specific events of interest that occur while running inference. The concrete event types are:
    - FactorToVariableMessageEvent and VariableToFactorMessageEvent: raised when edge messages are updated in belief propagation solvers. Currently only the sumproduct and minsum solvers generate these messages.
    - GibbsVariableUpdateEvent and GibbsScoredVariableUpdateEvent: raised when sample values are changed by the Gibbs solver. The two event types are the same except that the scored version adds information about the change in score induced by the sample update.

Additional events may be added in future releases. New event types may also be added by developers who have extended Dimple with their own solvers or custom factors in Java.

When specifying event types in the Java API, use the event class itself:

```
DimpleEventLogger logger = new DimpleEventLogger();
logger.log(FactorToVariableMessageEvent.class, fg);
```

5.8.2 Event logging

The easiest way to monitor events in Dimple is through an event logger. Given an event logger instance, you can configure it to log either to the console or to a text file, you can configure how verbose the output should be, and specify which events for which model objects should be logged.

Event loggers are instances of the DimpleEventLogger class. You may create a new one using the constructor:

```
DimpleEventLogger logger = new DimpleEventLogger();
```

Newly constructed loggers will output to standard error by default and will have a default verbosity of zero, which will produce the most terse output. You may configure the logger to change the verbosity level or to direct output to a different target. For example:

```
// Use more verbose log output.
logger.verbosity(2);

// Append output to a file in the working directory.
logger.open(new File("event-logger.txt"));

// ... or output to standard output
logger.open(System.out);
```

Usually a single logger will be sufficient, but you can create multiple logger objects that direct output to different targets.

To enable logging for a particular class of events on your model, use the log method with the event type of interest. If the event type is abstract (is annotated with the letter A in Figure 1), then all event subtypes will be logged. If the event type is not abstract, then only that particular event type will be logged. In particular, if you specify SolverEvent on a graph using the Gibbs solver, you will see GibbsScoredVariableUpdateEvent messages but if you specify GibbsVariableUpdateEvent you will get only messages for that specific class and will not get scored messages.

```
// Log all solver events for given model.
logger.log(SolverEvent.class, model);

// Log unscored Gibbs update messages
logger.log(GibbsVariableUpdateEvent.class, model);

// Log variable to factor messages for a single variable
logger.log(VariableToFactorMessageEvent.class, x);
```

You may remove previously created log registration either by using the unlog method to remove individual entries or the clear method to remove all entries. When using unlog, the arguments must match the original arguments passed to log. (Note that setting the verbosity to a negative value or closing the output will also turn off logging it will not prevent Dimple from creating the underlying events, so make sure to use the clear() method when you are done with logging if you do not want to slow down inference.)

```
// Disable a previous log registration
logger.log(SolverEvent.class, model);

// Disable all logging
logger.clear();
```

When using logging, it is usually very helpful to give the variables, factors and subgraphs unique, easy to read, names. This will make your log output much easier to understand.

5.8.3 Advanced event handling

The DimpleEventLogger class provides an easy way to monitor changes when running inference in Dimple, but it does not let you interact directly with the event objects or take actions other than simple logging. A much wider range of options is available by using the event handler and listener framework upon which the logger is based. This is described in more detail in the next sections.

5.8.4 Event listeners

An object of type DimpleEventListener may be attached to the root FactorGraph of the model by the FactorGraph.setEventListener method and will be used to dispatch all events generated by the model and its associated data and solver layers. If there is no listener on the root graph, then no events should be generated. By default there is no listener for newly created graphs. When creating a listener, you can either create a new instance or use the global default instance that is lazily created by DimpleEventListener.getDefault(). Note that the DimpleEventLogger automatically adds the default listener to the root graph of models that don't already have a listener when a new log specification is registered for a child of the graph, but you will need to do this explicitly if you are not using that class.

Once a listener has been associated with the graph, then one or more handlers may be registered to handle events on the graph. The registration must specify the base class of the type of event to be handled and the root object on which events will be raised. It will be easiest to register for events at the root graph, but it may sometimes be desirable to set up handlers for specific variables, factors or subgraphs. For instance, to register handlers for various belief propagation messages on a graph, you could write:

```java
DimpleEventListener listener = DimpleEventListener.getDefault();
fg.setEventListener(listener);
listener.register(factorMessageHandler,
    FactorToVariableMessageEvent.class, fg);
listener.register(variableMessageHandler,
    VariableToFactorMessageEvent.class, fg);
```

Handlers can be removed by one of the various unregister methods on the listener; see the Java API doc for that class for details. Changes to event registration or the value of the root listener are not guaranteed to take effect until the next time the affected objects have been initialized or the IDimpleEventSource.notifyListenerChanged() method has been invoked on the object that generates the event. This is important for model events in particular since model changes typically occur prior to initialization.

5.8.5 Event handlers

Dimple event handlers are objects that implement the IDimpleEventHandler interface. In practice most handlers should simply extend the abstract base class DimpleEventHandler. For example, here is a simple handler class that simply prints events out to the console with a verbosity of one:

```
public class EventPrinter extends DimpleEventHandler<DimpleEvent>
{
    public void handleEvent(DimpleEvent event)
    {
        event.println(System.out, 1);
    }
}
```

Handler classes that are specific to a particular event subclass can be parameterized appropriately to avoid the need for downcasts. For example, here is a simple handler that keeps a running total of the total graph score during Gibbs sampling based on sample score differences:

```
public class RunningScoreHandler extends DimpleEventHandler<
   GibbsScoredVariableUpdateEvent>
{
    public double score;

    RunningScoreHandler(double startingScore)
    {
        score = startingScore;
    }

    public void handleEvent(GibbsScoredVariableUpdateEvent event)
    {
        score += event.getScoreDifference();
    }
}
```

5.9 List of Built-in Factors

The following table lists the current set of built-in Dimple factors. For each, the name is given, followed by the set of variables that would be connected to the factor, followed by any constructor arguments. Optional variables and constructor arguments are in brackets. And an arbitrary length list or array of variables is followed by ellipses. The allowed set of variable data-types for each variable is given in parentheses (B = Bit, D = Discrete, F = FiniteFieldVariable, R = Real, C = Complex, RJ = RealJoint).

Name	Variables	Constructor	Description

Abs	out(D,R)
	in(D,R)
		[smoothing]	Deterministic absolute value function, where out = abs(in). An optional smoothing value may be specified as a constructor argument[34].

ACos	out(D,R)
	in(D,R)
		[smoothing]	Deterministic arc-cosine function, where out = acos(in). An optional smoothing value may be specified as a constructor argument[34].

AdditiveNoise	out(R)
	in(B,D,R)
		σ	Add Gaussian noise with a known standard deviation, σ, specified in constructor.

And	out(B)
	in...(B)
		-	Deterministic logical AND function, where out = AND(in...).

ASin	out(D,R)
	in(D,R)
		[smoothing]	Deterministic arc-sine function, where out = asin(in). An optional smoothing value may be specified as a constructor argument[34].

ATan	out(D,R)
	in(D,R)
		[smoothing]	Deterministic arc-tangent function, where out = atan(in). An optional smoothing value may be specified as a constructor argument[34].

Bernoulli	[ρ](R) x...(B)	[ρ]	Bernoulli distribution, $p(x\|\rho)$, where ρ is a parameter indicating the probability of one, and x is an array of Bit variables. There can be any number of x variables, all associated with the same parameter value. The conjugate prior for the parameter, ρ, is a Beta distribution[35]. The parameter, ρ, can be a variable or a constant specified in the constructor.

[34]If smoothing is enabled, the factor function becomes $e^{-(\mathrm{out}-F(\mathrm{in}))^2/\mathrm{smoothing}}$ (making it non-deterministic) instead of $\delta(\mathrm{out} - F(\mathrm{in}))$, where F is the deterministic function associated with this factor. This is useful for solvers that do not work well with deterministic real-valued factors, such as particle BP, particularly when annealing is used.

[35]It is not necessary to use the conjugate prior, but in some cases there may be a benefit.

Name	Variables	Constructor	Description
Beta	$[\alpha]$(R) $[\beta]$(R) value...(R)	$[\alpha]$ $[\beta]$	Beta distribution. There can be any number of value variables, all associated with the same parameter values. Parameters α and β can be variables, or if both are constant they can be specified in the constructor.
Binomial	$[N]$(D) ρ(R) x(D)	$[N]$	Binomial distribution, $p(x\|N, \rho)$, where N is the total number of trials, ρ is a parameter indicating the success probability, and x is a count of success outcomes. Parameter N can be a Discrete variable with positive integer values or a constant integer value specified in the constructor. The domain of x must include integers from 0 through N, or if N is a variable, through the maximum value in the domain of N. The conjugate prior for the parameter, ρ, is a Beta distribution[35].
Categorical	$[\alpha]$(RJ) x...(D)	$[\alpha]$	Categorical distribution, $p(x\|\alpha)$, where α is a vector of parameter variables and x is an array of discrete variables. The number of elements in α must equal the domain size of x. There can be any number of x variables, all associated with the same parameter values. The α parameters are represented as a normalized probability vector. The conjugate prior for this representation is a Dirichlet distribution[35]. In the current implementation, the domain of the x variable must be zero-based contiguous integers, $0...N-1$[36].

[36]This limitation may be lifted in a future version.

Name	Variables	Constructor	Description
Categorical EnergyParameters	$[\alpha]$...(R) x...(D)	N, $[\alpha]$	Categorical distribution, $p(x\|\alpha)$, where α is a vector of parameter variables and x is an array of discrete variables. The number of elements in α and the domain size of x must equal the value of the constructor argument, N. There can be any number of x variables, all associated with the same parameter values. In this alternative version of the Categorical distribution, the α parameters are represented as energy values, that is, $\alpha = -\log(\rho)$, where ρ are unnormalized probabilities. The conjugate prior for this representation is such that each entry of α is independently distributed according to a negative exp-Gamma distribution, all with a common β parameter[35]. In the current implementation, the domain of the x variable must be zero-based contiguous integers, $0...N-1$[36].
Categorical Unnormalized Parameters	$[\alpha]$...(R) x...(D)	N, $[\alpha]$	Categorical distribution, $p(x\|\alpha)$, where α is a vector of parameter variables and x is an array of discrete variables. The number of elements in α and the domain size of x must equal the value of the constructor argument, N. There can be any number of x variables, all associated with the same parameter values. In this alternative version of the Categorical distribution, the α parameters are represented as a vector of unnormalized probability values. The conjugate prior for this representation is such that each entry of α is independently distributed according to a Gamma distribution, all with a common β parameter[35]. In the current implementation, the domain of the x variable must be zero-based contiguous integers, $0...N-1$[36].
ComplexAbs	out(R) in(C)	[smoothing]	Deterministic complex absolute value, where out = $\sqrt{Re(\text{in}) + Im(\text{in})}$. An optional smoothing value may be specified as a constructor argument[34].
ComplexConjugate	out(C) in(C,R)	[smoothing]	Deterministic complex conjugate function, where out = in*. An optional smoothing value may be specified as a constructor argument[34].

Name	Variables	Constructor	Description
ComplexDivide	quotient(C) dividend(C,R) divisor(C,R)	[smoothing]	Deterministic complex divide function, where quotient = $\frac{\text{dividend}}{\text{divisor}}$. An optional smoothing value may be specified as a constructor argument[34].
ComplexExp	out(C) in(C,R)	[smoothing]	Deterministic complex exponentiation function, where out = exp(in). An optional smoothing value may be specified as a constructor argument[34].
ComplexNegate	out(C) in(C,R)	[smoothing]	Deterministic complex negation function, where out = -in. An optional smoothing value may be specified as a constructor argument[34].
ComplexProduct	out(C) in...(C,R)	[smoothing]	Deterministic complex product function, where out = $\prod$in. An optional smoothing value may be specified as a constructor argument[34].
ComplexSubtract	out(C) posIn(C,R) negIn...(C,R)	[smoothing]	Deterministic complex subtraction function, where out = posIn $-\sum$negIn. An optional smoothing value may be specified as a constructor argument[34].
ComplexSum	out(C) in...(C,R)	[smoothing]	Deterministic complex summation function, where out = $\sum$in. An optional smoothing value may be specified as a constructor argument[34].
ComplexTo RealAndImaginary	outReal(R) outImag(R) in(RJ)	[smoothing]	Deterministic conversion of a Complex variable to two Real variables, with the first representing the real component and the second representing the imaginary component. An optional smoothing value may be specified as a constructor argument[34].
Cos	out(D,R) in(D,R)	[smoothing]	Deterministic cosine function, where out = cos(in). An optional smoothing value may be specified as a constructor argument[34].
Cosh	out(D,R) in(D,R)	[smoothing]	Deterministic hyperbolic-cosine function, where out = cosh(in). An optional smoothing value may be specified as a constructor argument[34].
Dirichlet	[α](RJ) value...(RJ)	[α]	Dirichlet distribution. There can be any number of value variables, all associated with the same parameter values. Parameter vector α can be a RealJoint variable or a constant specified in the constructor. The dimension of α and each of the value variables must be identical.

Name	Variables	Constructor	Description

DiscreteTransition	y(D)
	x(D)
	A...(RJ)
		-	Parameterized discrete transition factor, $p(y\|x, A)$, where x and y are discrete variables, and A is a matrix of transition probabilities. The transition matrix is organized such that columns correspond to the output distribution for each input state. That is, the transition matrix multiplies on the left. Each column of A corresponds to a RealJoint variable. The number of columns in A must equal the domain size of x, and the dimension of each element of A must equal the domain size of y.
			Each element of A corresponds to a normalized probability vector. The conjugate prior for this representation is such that each element of A is distributed according to a Dirichlet distribution[35].
			In the current implementation, the domain of the x variable must be zero-based contiguous integers, $0...N-1$[36].

DiscreteTransition EnergyParameters	y(D)
	x(D)
	A...(R)
		$N_y, N_x\|$
		N
			Parameterized discrete transition factor, $p(y\|x, A)$, where x and y are discrete variables, and A is a matrix of transition probabilities. The transition matrix is organized such that columns correspond to the output distribution for each input state. That is, the transition matrix multiplies on the left. The number of columns in A and the domain size of x must equal the value of the constructor argument, N_x and the number of rows in A and the domain size of y must equal the value of the constructor argument N_y. If N_x and N_y are equal, a single constructor argument, N, may be used.
			The elements of the matrix A are represented as energy values, that is, $A_{i,j} = -\log(\rho_{i,j})$, where ρ is an unnormalized transition probability matrix. The conjugate prior for this representation is such that each entry of A is independently distributed according to a negative exp-Gamma distribution, all with a common β parameter[35].
			In the current implementation, the domain of the x variable must be zero-based contiguous integers, $0...N-1$[36].

Name	Variables	Constructor	Description
DiscreteTransition Unnormalized Parameters	y(D) x(D) A...(R)	$N_y, N_x\|$ N	Parameterized discrete transition factor, $p(y\|x, A)$, where x and y are discrete variables, and A is a matrix of transition probabilities. The transition matrix is organized such that columns correspond to the output distribution for each input state. That is, the transition matrix multiplies on the left. The number of columns in A and the domain size of x must equal the value of the constructor argument, N_x and the number of rows in A and the domain size of y must equal the value of the constructor argument N_y. If N_x and N_y are equal, a single constructor argument, N, may be used. The elements of the matrix A are represented as unnormalized probability values. The conjugate prior for this representation is such that each entry of A is independently distributed according to a Gamma distribution, all with a common β parameter[35]. In the current implementation, the domain of the x variable must be zero-based contiguous integers, $0...N-1$[36].
Divide	quotient(D,R) dividend(D,R) divisor(D,R)	[smoothing]	Deterministic divide function, where quotient $= \frac{\text{dividend}}{\text{divisor}}$. An optional smoothing value may be specified as a constructor argument[34].
Equality	value...(B,D,R)	[smoothing]	Deterministic equality constraint. An optional smoothing value may be specified as a constructor argument[34].
Equals	out(B) in...(B,D,R)	-	Deterministic equals function, where out = (in(1) == in(2) == ...).
ExchangeableDirichlet	[α](R) value...(RJ)	N, [α]	Exchangeable Dirichlet distribution. This is a variant of the Dirichlet distribution parameterized with a single common parameter for all dimensions. There can be any number of value variables, all associated with the same parameter value. Parameter α can be a Real variable or a constant specified in the constructor. The dimension of each of the value variables must be identical and equal to the value of N, specified in the constructor.

Name	Variables	Constructor	Description

Exp	out(D,R)
	in(D,R)
		[smoothing]	Deterministic exponentiation function, where out = exp(in). An optional smoothing value may be specified as a constructor argument[34].

FiniteFieldAdd	out(F)
	in1(F)
	in2(F)
		-	Deterministic finite field two-input addition. See section 4.4 for a description of how to use finite field variables.

FiniteFieldMult	out(F)
	in1(F)
	in2(F)
		-	Deterministic finite field two-input multiplication. See section 4.4 for a description of how to use finite field variables.

FiniteFieldProjection	fieldVar(F)
	indices(const)
	bits...(B)
		-	Deterministic projection of a finite field variable onto a set of bit variables corresponding to the bits of the field value. The indices argument is a constant array, which must be a permutation of 0 through $N-1$, where N is the number of bits in the finite field value. The indices represent the order of the projection of the bits in the finite field value onto the corresponding Bit variable in the list of bits. See section 4.4 for a description of how to use finite field variables.

Gamma	[α](R) [β](R) value...(R)	[α] [β]	Gamma distribution. There can be any number of value variables, all associated with the same parameter values. Parameters α and β can be variables, or if both are constant they can be specified in the constructor.

GreaterThan	out(B)
	in1(B,D,R)
	in2(B,D,R)
		-	Deterministic greater-than function, where out = in1 $>$ in2.

InverseGamma	[α](R) [β](R) value...(R)	[α] [β]	Inverse Gamma distribution. There can be any number of value variables, all associated with the same parameter values. Parameters α and β can be variables, or if both are constant they can be specified in the constructor.

LessThan	out(B)
	in1(B,D,R)
	in2(B,D,R)
		-	Deterministic greater-than function, where out = in1 $<$ in2.

Name	Variables	Constructor	Description

LinearEquation	out(D,R)
	in(B,D,R)
		weights
		[smoothing]
			Deterministic linear equation, multiplying an input vector by a constant weight vector to equal the output variable. The weight vector is specified in the constructor. The number of *in* variables must equal the length of the weight vector. An optional smoothing value may be specified as a constructor argument[34].

Log	out(D,R)
	in(D,R)
		[smoothing]	Deterministic natural log function, where out = log(in). An optional smoothing value may be specified as a constructor argument[34].

LogNormal	[μ](R) [τ](R) value...(R)	[μ] [τ]	Log-normal distribution. There can be any number of value variables, all associated with the same parameter values. Parameters μ (mean) and $\tau = \frac{1}{\sigma^2}$ (precision) can be variables, or if both are constant then fixed parameters can be specified in the constructor.

MatrixProduct	C(D,R)
	A(D,R)
	B(D,R)
		N_r
		N_x
		N_c
		[smoothing]
			Deterministic matrix product function, $C = AB$, where A, B, and C are matrices. Constructor arguments, N_r specifies the number of rows in A and C, N_x specifies the number of columns in A and number of rows in B, and N_c specifies the number of columns in B and C. An optional smoothing value may be specified as a constructor argument[34].

MatrixVectorProduct	y(D,R)
	M(D,R)
	x(D,R)
		N_x
		N_y
		[smoothing]
			Deterministic matrix-vector product function, $y = Mx$, where x and y are vectors and M is a matrix. Constructor arguments, N_x and N_y, specify the input and output vector lengths, respectively. The matrix dimension is $N_y \times N_x$. An optional smoothing value may be specified as a constructor argument[34].

MatrixRealJoint
VectorProduct
	y(RJ)
	M(D,R)
	x(RJ)
		N_x
		N_y
		[smoothing]
			Deterministic matrix-vector product function, $y = Mx$, where x and y are RealJoint values and M is a matrix. Constructor arguments, N_x and N_y, specify the input and output vector lengths, respectively. The matrix dimension is $N_y \times N_x$. An optional smoothing value may be specified as a constructor argument[34].

Name	Variables	Constructor	Description
Multinomial	$[N]$(D) α(RJ) x...(D)	$[N]$	Multinomial distribution, $p(x\|N, \alpha)$, where N is the total number of trials, α is a vector of parameter variables, and x is a count of outcomes in each category. Parameter N can be a Discrete variable with positive integer values or a constant integer value specified in the constructor. The number of elements in α must exactly match the number of elements of x. The domain of each x variable must include integers from 0 through N, or if N is a variable, through the maximum value in the domain of N. The α parameters are represented as a normalized probability vector. The conjugate prior for this representation is a Dirichlet distribution^{35}.
Multinomial EnergyParameters	$[N]$(D) α...(R) x...(D)	D, $[N]$	Multinomial distribution, $p(x\|N, \alpha)$, where N is the total number of trials, α is a vector of parameter variables, and x is a count of outcomes in each category. Parameter N can be a Discrete variable with positive integer values or a constant integer value specified in the constructor. The number of elements in α must exactly match the number of elements of x, which must match the value of the constructor argument, D. The domain of each x variable must include integers from 0 through N, or if N is a variable, through the maximum value in the domain of N. In this alternative version of the Multinomial distribution, the α parameters are represented as energy values, that is, $\alpha = -\log(\rho)$, where ρ are unnormalized probabilities. The conjugate prior for this representation is such that each entry of α is independently distributed according to a negative exp-Gamma distribution, all with a common β parameter^{35}.

Name	Variables	Constructor	Description
Multinomial Unnormalized Parameters	$[N]$(D) α...(R) x...(D)	D, $[N]$	Multinomial distribution, $p(x\|N,\alpha)$, where N is the total number of trials, α is a vector of parameter variables, and x is a count of outcomes in each category. Parameter N can be a Discrete variable with positive integer values or a constant integer value specified in the constructor. The number of elements in α must exactly match the number of elements of x, which must match the value of the constructor argument, D. The domain of each x variable must include integers from 0 through N, or if N is a variable, through the maximum value in the domain of N. In this alternative version of the Multinomial distribution, the α parameters are represented as a vector of unnormalized probability values. The conjugate prior for this representation is such that each entry of α is independently distributed according to a Gamma distribution, all with a common β parameter[35].
Multiplexer	out(any) select in...(any)	[smoothing]	Deterministic multiplexer[37]. The selector must be a discrete variable that selects one of the inputs to pass to the output. The data type of all inputs must be identical to that of the output. For RealJoint variables, the dimension of all inputs must equal that of the output. The with domain of the selector variable must be zero-based contiguous integers, $0...N-1$, where N is the number of input variables. An optional smoothing value may be specified as a constructor argument[34].

[37]Note that for the SumProduct solver, an optimized custom implementation of this factor function is used automatically, which avoids creation of a corresponding factor table.

Name	Variables	Constructor	Description
MultivariateNormal	value...(RJ)	μ Σ	Multivariate Normal distribution. There can be any number of value variables, all associated with the same parameter values. Parameters μ (mean vector) and Σ (covariance matrix) are constant that must be specified in the constructor[38]. The dimension of the mean vector, both dimensions of the covariance matrix, and the dimension of each value variable must be identical.

Negate	out(D,R)
	in(D,R)
		[smoothing]	Deterministic negation function, where out = -in. An optional smoothing value may be specified as a constructor argument[34].

NegativeExpGamma	[α](R) [β](R) value...(R)	[α] [β]	Negative exp-Gamma distribution, which is a distribution over a variable whose negative exponential is Gamma distributed. That is, this is the negative log of a Gamma distributed variable. There can be any number of value variables, all associated with the same parameter values. Parameters α and β can be variables, or if both are constant they can be specified in the constructor, and correspond to the parameters of the underlying Gamma distribution.
Normal	[μ](R) [τ](R) value...(R)	[μ] [τ]	Normal distribution. There can be any number of value variables, all associated with the same parameter values. Parameters μ (mean) and $\tau = \frac{1}{\sigma^2}$ (precision) can be variables, or if both are constant then fixed parameters can be specified in the constructor.

Not	out(B)
	in(B)
		-	Deterministic logical NOT of function, where out = in.

NotEquals	out(B)
	in...(B,D,R)
		-	Deterministic not-equals function, where out = ∼(in(1) == in(2) == ...).

Or	out(B)
	in...(B)
		-	Deterministic logical OR function, where out = OR(in...).

[38]In this version of Dimple, there is no support for variable parameters in the MultivariateNormal distribution.

Name	Variables	Constructor	Description
Poisson	$[\lambda]$(R) k(D)	$[\lambda]$	Poisson distribution, $p(k\|\lambda)$, where λ is the rate parameter, and k is the discrete output. While the value of k for a Poission distribution is unbounded, the domain should be set to include integers from 0 through a maximum value. The maximum value should be a multiple of the maximum likely value of λ[39]. The conjugate prior for the parameter, λ, is a Gamma distribution[35].

Power	out(D,R)
	base(D,R)
	power(D,R)
		[smoothing]	Deterministic power function, where out = base $^{\text{power}}$. An optional smoothing value may be specified as a constructor argument[34].

Product	out(D,R)
	in...(B,D,R)
		[smoothing]	Deterministic product function, where out $=$ $\prod$in. An optional smoothing value may be specified as a constructor argument[34].

Rayleigh	$[\sigma]$(R) value...(R)	$[\sigma]$	Rayleigh distribution. There can be any number of value variables, all associated with the same parameter value. Parameter σ can be a variable, or if constant, can be specified in the constructor.

RealAndImaginary ToComplex	out(C)
	inReal(R)
	inImag(R)
		[smoothing]	Deterministic conversion of two Real variables to a Complex variable, where the first input represents the real component and the second represents the imaginary component. An optional smoothing value may be specified as a constructor argument[34].

RealJointNegate	out(RJ)
	in(RJ)
		[smoothing]	Deterministic negation function for RealJoint variables, where out = -in. An optional smoothing value may be specified as a constructor argument[34].

RealJointProjection	out(R)
	in(RJ)
		index
		[smoothing]
			Deterministic conversion of a RealJoint variable to a Real variable corresponding to one specific element of the RealJoint variable. The *index* constructor argument indicates which element of the RealJoint variable to be used (using zero-based numbering). An optional smoothing value may be specified as a constructor argument[34].

[39]If the maximum value is 5 times larger than the largest value of λ, then less than 0.1 of the probability mass would fall above this value.

Name	Variables	Constructor	Description

RealJointSubtract	out(RJ)
	posIn(RJ)
	negIn...(RJ)
		[smoothing]	Deterministic subtraction function for RealJoint variables, where out = posIn − $\sum$ negIn. An optional smoothing value may be specified as a constructor argument[34].

RealJointSum	out(RJ)
	in...(RJ)
		[smoothing]	Deterministic summation function for RealJoint variables, where out = $\sum$ in. An optional smoothing value may be specified as a constructor argument[34].

RealJointTo RealVector	out...(R)
	in(RJ)
		[smoothing]	Deterministic conversion of a RealJoint variable to a vector of Real variables. An optional smoothing value may be specified as a constructor argument[34].

RealVectorTo RealJoint	out(RJ)
	in...(R)
		[smoothing]	Deterministic conversion of a vector of Real variables to a RealJoint variable. An optional smoothing value may be specified as a constructor argument[34].

Sin	out(D,R)
	in(D,R)
		[smoothing]	Deterministic sine function, where out = sin(in). An optional smoothing value may be specified as a constructor argument[34].

Sinh	out(D,R)
	in(D,R)
		[smoothing]	Deterministic hyperbolic-sine function, where out = sinh(in). An optional smoothing value may be specified as a constructor argument[34].

Sqrt	out(D,R)
	in(D,R)
		[smoothing]	Deterministic square root function, where out = sqrt(in). An optional smoothing value may be specified as a constructor argument[34].

Square	out(D,R)
	in(D,R)
		[smoothing]	Deterministic square function, where out = in^2. An optional smoothing value may be specified as a constructor argument[34].

Subtract	out(D,R)
	posIn(B,D,R)
	negIn...(B,D,R)
		[smoothing]	Deterministic subtraction function, where out = posIn − $\sum$ negIn. An optional smoothing value may be specified as a constructor argument[34].

Sum	out(D,R)
	in...(B,D,R)
		[smoothing]	Deterministic summation function, where out = $\sum$ in. An optional smoothing value may be specified as a constructor argument[34].

Tan	out(D,R)
	in(D,R)
		[smoothing]	Deterministic tangent function, where out = tan(in). An optional smoothing value may be specified as a constructor argument[34].

Tanh	out(D,R)
	in(D,R)
		[smoothing]	Deterministic hyperbolic-tangent function, where out = tanh(in). An optional smoothing value may be specified as a constructor argument[34].

Name	Variables	Constructor	Description

VectorInnerProduct	z(D,R)
	x(D,R,RJ)
	y(D,R,RJ)
		[smoothing]	Deterministic vector inner product function, $z = x\dot{y}$, where x and y are vectors and z is a scalar. Each vector input may be either an array of scalar variables, or a single RealJoint variable. The number of elements in x and y must be identical. An optional smoothing value may be specified as a constructor argument[34].

VonMises	[μ](R) [τ](R) value...(R)	[μ] [τ]	Von Mises distribution. There can be any number of value variables, all associated with the same parameter values. Parameters μ (mean) and $\tau = \frac{1}{\sigma^2}$ (precision) can be variables, or if both are constant then fixed parameters can be specified in the constructor. The distribution is non-zero for value variables in the range $-\pi$ to π.

Xor	out(B)
	in...(B)
		-	Deterministic logical XOR function, where out = XOR(in...).

Dimple also includes some built-in helper functions to create structured graphs, combining smaller factors to form an efficient implementation of a larger subgraph. Specifically, the following functions are provided:

- MultiplexerCPD(domains) - See section 4.6.1
- MultiplexerCPD(domain, numZs) - See section 4.6.1

5.10 Other Top Level Functions

5.10.1 setSolver

```
Model.getInstance().setDefaultGraphFactory(graphFactory);
```

This function changes the default solver to the solver designated by the argument. (see section 5.1.2.1 for the list of valid solver names, and section 5.6 for a description of each solver).

5.10.2 dimpleVersion

Dimple provides a method to return a string describing the current Dimple version.

```
version = Model.getVersion();
```

This will produce something of the form:

```
<Release Number> <Git Branch Name> YYY-MM-DD HH:MM:SS <Timezone Offset>
```

For example:

```
0.04 master 2013-10-11 14:00:05 -0400
```

The date in the version string represents the last git commit date associated with the compiled code.

A A Short Introduction to Factor Graphs

We introduce factor graphs, a powerful tool for statistical modeling. Factor graphs can be used to describe a number of commonly used statistical models, such as hidden Markov models, Markov random fields, Kalman filters, and Bayes nets.

Suppose that we are given a set of n discrete random variables: $a_1, ..., a_n$. The random variables have some joint probability distribution: $p(a_1, a_2, ..., a_n)$. Suppose that the joint probability distribution factors, in the following sense: there exist subsets $S_1, ..., S_k \subseteq \{1, 2, ..., n\}$ where $S_j = \{S^j_1, s^j_2, ..., s^j_{t(j)}\}$ and such that $p(a_1, a_2, ..., a_n) = \prod_{j=1}^{k} f_j(a_{s^j_1}, a_{s^j_2}, ..., a_{s^j_{t(j)}})$.

For example, if the a_i form a Markov chain, then the joint probability can be factored as

$$\begin{aligned} p(a_1, a_2, ..., a_n) &= p(a_1) \prod_{j=1}^{n-1} p(a_{j+1}|a_j) && (1) \\ &= f_0(a_1) \prod_{j=1}^{n-1} f_j(a_j, a_{j+1}) && (2) \end{aligned}$$

The factors above are normalized, in the sense that as the a_i vary, the probabilities sum to one. We will define our factors more generally and ask only that they are proportional to the joint probability. So, we call the f_j a collection of factors of p if

$$p(a_1, a_2, ..., a_n) \propto \prod_{j=1}^{k} f_j(a_{s^j_1}, a_{s^j_2}, ..., a_{s^j_{t(j)}})$$

The product of the factors then differs from the joint probability only by multiplication by a normalizing constant.

When a probability distribution can be expressed as a product of small factors (i.e., $|S_j|$ is small for all j), then if is possible to invoke a host of powerful tools for modeling and inference, as we will soon see.

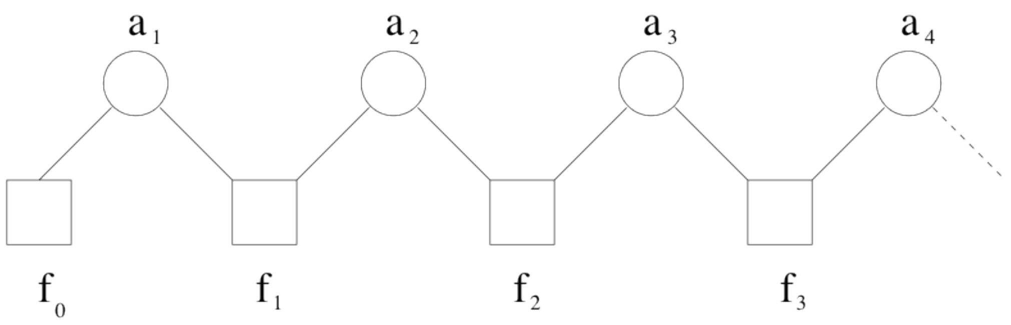
Suppose that we are given a factored representation of a joint probability distribution. It is possible to describe the structure of the factors as a graph. We can represent each variable a_i and each function f_j by a node in the graph, and place an (undirected) edge between node a_i and node f_j if and only if the variable a_i is an argument in the function f_j. These two types of nodes are referred to as factor nodes and variable nodes. Because all edges lie between the two disjoint classes of nodes, the resulting graph is bipartite. This graph is called a factor graph.

In the remainder of this documentation, we slightly abuse notation and use f_j and a_i to refer both to the nodes of the factor graph and to the underlying factors and variables (i.e., both the graphical representation of these entities and the mathematical entities underlying them).

To understand what factor graphs look like, we will construct several examples. First, let us continue with a Markov chain. Equation 1 expressed a Markov chain in factored form, where

$$f_j(a_j, a_{j+1}) = p(a_{j+1}|a_j)$$

We display the corresponding factor graph in the following figure:



Next, let us consider a hidden Markov model (HMM). We can construct the corresponding factor graph by extending the previous example. An HMM contains a Markov chain transiting from state a_i to a_{i+1}. There is also an observation b_i made of each state; if we are given a_i, then b_i is conditionally independent of all other variables. We can incorporate this probability by using a factor:

$$g_i(a_i) = Pr(b_i|a_i)$$

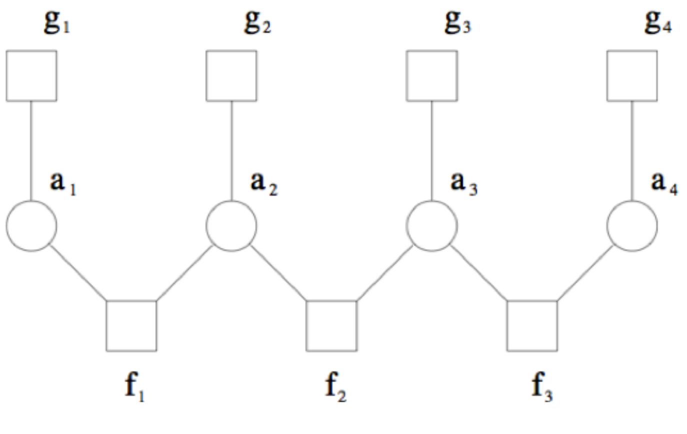
The product of our factors is then

$$\begin{aligned} f_0\left(\prod_{j=1}^{n-1} f_j(a_j, a_j+1)\right)\prod_{j=1}^{n} g_j(a_j) &= Pr(a_1)\left(\prod_{j=1}^{n-1} Pr(a_{j+1}|a_j)\right)\prod_{j=1}^{n} Pr(b_j|a_j) \\ &= Pr(a_1, ..., a_n, b_1, ..., b_n) \end{aligned}$$

Since the b_i observed, then $Pr(b_1, ..., b_n)$ is a constant. Therefore

$$\begin{aligned} Pr(a_1, ..., a_n, b_1, ..., b_n) &\propto \frac{Pr(a_1, ..., a_n, b_1, ..., b_n)}{Pr(b_1, ..., b_n)} \\ &= Pr(a_1, ..., a_n|b_1, ..., b_n) \end{aligned}$$

as desired.



The resulting factor graph takes the following form illustrated in the figure above. Note that the variables b_i need not appear explicitly in the factor graph; we have incorporated their effect in the g_i factors.

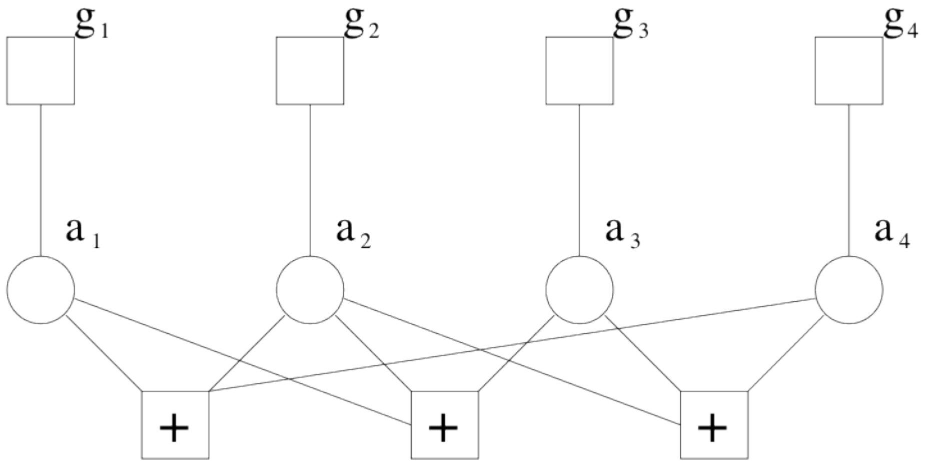
Generalizing from a Markov chain to an HMM illustrates a very powerful feature of factor graphs. Complicated mathematical models are often composed of simpler parts. When these models are expressed as factor graphs, we can frequently reuse the simpler factor graphs to construct the more complicated ones. This can be done simply in Dimple by using the nested graphs feature (see section 3.3 on Nested Graphs).

As a final example, we will construct a factor graph for error correction (for this more advanced topic, we will assume the reader is familiar with LDPC codes). Suppose that we receive a codeword from a 4-bit LDPC error-correcting code that has been corrupted by noise. The sender wishes to communicate a four-bit codeword (a_1, a_2, a_3, a_4) satisfying some parity check equations, but the receiver only observes the corrupted values (b_1, b_2, b_3, b_4). (The domain of the b_i is determined by the communication channel. For instance, if we have a discrete binary symmetric channel, then the b_i will be bits; if we have a continuous additive white Gaussian noise channel and some modulation scheme, the b_i will be real-valued.) Let H be the parity check matrix of the LDPC code used, i.e., the codeword (a_1, a_2, a_3, a_4) verifies the equation $Ha = 0 \bmod 2$.

For instance, suppose that H is the following parity check matrix:

$$H = \begin{pmatrix} 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \end{pmatrix}$$

Suppose that the error is independent, i.e., if we condition on a_i, b_i is conditionally independent of the other variables. Then, the following factor graph represents the decoding task at hand.



The construction above applies to any linear binary ECC. However, if every row and column of H is sparse (as would be the case with an LDPC code), then every factor is small, and every node in the factor graph will be of small degree.

Given a factor graph, the objective is often to compute the marginal distribution of the random variables a_i of the graph (this also allows us to find the most likely value taken by each variable, by maximization of the marginal probability). Dimple provides an implementation of belief propagation (BP) (in its sum-product version), in order to approximately compute the marginal distribution of each random variable.

BP is an iterative message-passing algorithm where messages pass along the edges of the factor graph. A "message" can be viewed as an un-normalized probability distribution. The algorithm comes in a number of variants depending on the message update rule and the order of the message updates.

The sum-product form of BP generalizes a number of algorithms, including the "forward-backward" algorithm of HMMs, and the BCJR algorithm of coding theory. It always gives the exact answer when the underlying factor graph is a tree (if the graph contains no cycles). Although it is not an exact algorithm for general graphs, BP has been found to give excellent results for a wide variety of factor graphs, and runs particularly fast on sparse factor graphs (i.e., factor graphs of low node degree).

B Creating Custom Dimple Extensions

B.1 Creating a Custom Factor Function

When a factor function function is needed to support continuous variables that is not available as a Dimple built-in factor, then it is necessary to create a custom factor function.

To create a custom factor function, you must create a Java class that extends the Dimple `FactorFunction` class. When extending the `FactorFunction` class, the following method must be overwritten:

- `evalEnergy`: Evaluates a set of input values and returns an energy value (negative log of a weight value).

The user may extend other methods, as appropriate:

- Constructor: If a constructor is specified (for example, to pass constructor arguments), it must call the constructor of the super class.
- `isDirected`: Indicates whether the factor function is directed. If directed, then there are a set of directed outputs for which the marginal distribution for all possible input values is a constant. If not overridden, this is assumed false.
- `getDirectedToIndices`: If a factor function is directed, indicates which edge indices are the directed outputs (numbering from zero), returning an array of integers. There are two forms of this method, which may be used depending on whether the set of directed outputs depends on the number of edges in the factor that uses this factor function (many factor functions support a variable number of edges). If `isDirected` is overridden and can return `true`, then this method must also be overridden.
- `isDeterministicDirected`: Indicates whether a factor function is both directed and deterministic. If deterministic and directed, then it is in the form of a deterministic function such that for all possible settings of the input values there is exactly one output value the results in a non-zero weight (or, equivalently, a non-infinite energy)[40]. If not overridden, this is assumed false.
- `evalDeterministic`: If a factor function is directed and deterministic, this method evaluates the values considered the inputs of the deterministic function and returns the resulting values for the corresponding outputs. Note that these are not the weight or energy values, but the actual values of the associated variables that are considered outputs of the deterministic function. If `isDeterministicDirected` is overridden and can return `true`, then this method must also be overridden.
- `eval`: Evaluates a set of input values and returns a weight instead of an energy value. Overriding this method would only be useful if implementing this method can be done

[40]The indication that a factor function is deterministic directed is used by the Gibbs solver, and is necessary for such factor functions to work when using the Gibbs solver.

significantly more computationally efficiently than the default implementation, which calls evalEnergy and then computes $\exp(-energy)$.

The following is a very simple example of a custom factor function:

```java
import com.analog.lyric.dimple.factorfunctions.core.FactorFunction;

/*
 * This factor enforces equality between all variables and weights
 * elements of the domain proportional to their value
 */
public class BigEquals extends FactorFunction
{
    @Override
    public final double evalEnergy(Value[] input)
    {
        if (input.length == 0)
            return 0;

        Value firstVal = input[0];

        for (int i = 1; i < input.length; i++)
            if (!input[i].valueEquals(firstVal))
                return Double.POSITIVE_INFINITY;

        return 0;
    }
}
```

B.2 Creating a Custom Proposal Kernel

In some cases, it may be useful to add a custom proposal kernel when using the Gibbs solver with a Metropolis-Hastings sampler. In particular, since the *block* Metropolis-Hastings sampler does not have a default proposal kernel, it is necessary to add a custom proposal kernel in this case.

To create a custom proposal kernel, you must create a Java class that implements either the Dimple `IProposalKernel` interface in the case of a single-variable proposal kernel, or the `IBlockProposalKernel` interface in the case of a block proposal kernel.

These interfaces define the following methods that must be implemented:

- `next` This method takes the current value(s) of the associated variable(s) along with the corresponding variable domain(s), and returns a proposal. The proposal object returned (either a `Proposal` object for a single-variable proposal or a `BlockProposal:` object for a block proposal) includes both the proposed value(s) as well as the forward and reverse proposal probabilities (the negative log of these probabilities).
- `setParameters`: Allows the class to include user-specified parameters that can be set to modify the behavior of the kernel. This method must be implemented but may be empty if no parameters are needed.
- `getParameters`: Returns the value of any user-specified parameters that have been specified. This method must be implemented but may return null if no parameters are needed.