Journal of Statistical Software

MMMMMM YYYY, Volume VV, Issue II. *doi: 10.18637/jss.v000.i00*

Blang: Bayesian Declarative Modelling of Arbitrary Data Structures

Alexandre Bouchard-Côté
University of British Columbia

Kevin Chern
University of British Columbia

Davor Cubranic
University of British Columbia

Sahand Hosseini
University of British Columbia

Justin Hume
Third Foundation Labs Inc.

Matteo Lepur
University of British Columbia

Zihui Ouyang
University of British Columbia

Giorgio Sgarbi
University of British Columbia

Abstract

Consider a Bayesian inference problem where a variable of interest does not take values in a Euclidean space. These "non-standard" data structures are in reality fairly common. They are frequently used in problems involving latent discrete factor models, networks, and domain specific problems such as sequence alignments and reconstructions, pedigrees, and phylogenies. In principle, Bayesian inference should be particularly well-suited in such scenarios, as the Bayesian paradigm provides a principled way to obtain confidence assessment for random variables of any type. However, much of the recent work on making Bayesian analysis more accessible and computationally efficient has focused on inference in Euclidean spaces.

In this paper, we introduce Blang, a domain specific language (DSL) and library aimed at bridging this gap. Blang allows users to perform Bayesian analysis on arbitrary data types while using a declarative syntax similar to BUGS. Blang is augmented with intuitive language additions to invent data types of the user's choosing. To perform inference at scale on such arbitrary state spaces, Blang leverages recent advances in parallelizable, non-reversible Markov chain Monte Carlo methods.

Keywords: Bayesian modelling language, Bayesian inference, non-standard data structures, Blang.

1. Introduction

Blang is a probabilistic programming language (PPL) and software development kit (SDK) for performing Bayesian data analysis. Its design supports scalable inference over arbitrary datatypes, in particular, combinatorial spaces which are of central importance in areas such as computational biology. The design philosophy is centered around the day-to-day requirements of real-world data science. In the following, we put Blang in the context of the rich PPL and Bayesian modelling ecosystem.

Probabilistic programming has revolutionized applied Bayesian statistics in the past two decades; now being a part of the core toolbox of applied statistics. For example, packages such as BUGS (Lunn *et al.* 2000, 2009, 2012), JAGS (Plummer 2003), Stan (Carpenter *et al.* 2017), and PyMC3 (Salvatier *et al.* 2015) have been widely used in various applications ranging from ecology (Semmens *et al.* 2009), to astronomy (Greiner *et al.* 2016), and psychology (Burkner and Vuorre 2019).

In recent years, research in the area of Bayesian modelling software has focused on two main directions. On one hand, considerable progress has been made in designing general-purpose PPLs (Wood *et al.* 2014; Paige and Wood 2014; Milch *et al.* 2005; Goodman *et al.* 2012) which are able to represent any computable probability distributions (Ackerman *et al.* 2017). However, inference in these powerful languages often has to resort to algorithms such as non-Markovian Sequential Monte Carlo that can have poor scalability. Rapid progress is being made to lift this limitation (e.g., Paige and Wood (2016)) but general purpose PPLs are not yet able to handle common computational biology tasks, such as Bayesian inference over the topology of medium to large scale multiple-sequence alignment datasets. A second area of active development (Carpenter *et al.* (2017); Salvatier *et al.* (2015); Bingham *et al.* (2018), *inter alia*) has been to use automatic differentiation combined with Hamiltonian Monte Carlo (HMC) sampling (Duane *et al.* 1987; Neal 2012), which is highly efficient in problems defined on continuous state spaces. Naturally, algorithms based on HMC are not necessarily well-suited for inference problems defined on discrete and combinatorial state spaces.

In the past, efficient sampling of combinatorial spaces has been achieved by designing portfolios of specialized samplers in a case-by-case basis (see e.g., Lakner *et al.* (2008); Höhna *et al.* (2016)). This process is time consuming and error prone. There is an opportunity to simplify this process, and to speed-up and parallelize inference, thanks to new developments in computational statistics such as non-reversible Markov chain Monte Carlo (MCMC) methods (Syed *et al.* 2019) based on parallel tempering (PT) (Geyer 1991), and on a non-standard flavour of sequential Monte Carlo (SMC) method that we call Sequential Change of Measure (SCM) to avoid confusion with state-space SMC (Del Moral *et al.* 2006; Neal 2001), augmented with adaptive schemes (Zhou *et al.* 2016). All these schemes are based on a continuum of probability distributions, all defined on the same space and interpolating between the prior and posterior. The benefit of these methods is that a simplistic set of sampling algorithms can still achieve high sampling efficiency while exploiting parallel architectures. Blang fully automates the construction of interpolating probability distributions and therefore democratizes the use of high-performance Monte Carlo schemes such non-reversible PT and SCM.

Blang is designed to be efficient not only in computational terms but also for the user's development time. To achieve this goal, considerable effort has been put to facilitate model construction, testing, reuse and integration into existing data analysis pipelines, and to support reproducible data analysis. Instead of creating a language from scratch, Blang is built using

Xtext (Efftinge and Völter 2006), a powerful framework for designing programming languages. Owing to this infrastructure, Blang incorporates a feature set comparable to many modern, fully-fledged, multi-paradigm languages: functional, generic and object programming, static typing, just-in-time compilation, garbage collection, IDE support for static types, profiling, code coverage, and debugging.

Blang comes with a growing library of built-in models, which are themselves written in Blang (as done in Murray and Schön (2018)), moreover, users can share and maintain models via an established transitive dependency management and versioning system. Blang also implements a suite of existing and novel testing strategies for models and MCMC methods, blending them with unit testing and multiple testing tools.

The goal of this paper is to provide readers with an introduction to Blang. We begin with an outline of the language's goals in Section 2, followed by illustrative examples in Section 4. We formalize and detail Blang's declarative syntax, and structure in Sections 5 and 6. Next, we discuss how users can utilize Blang's Software Development Kit (SDK) to implement complex models in Section 7, followed by a description of its architecture as a whole in Section 9.

2. Goals

Blang's purpose is to provide Monte Carlo approximations of posterior distributions arising in Bayesian inference problems. The design of the language and its software development kit is guided by the following high-level goals:

Correctness: Bayesian inference software is notoriously difficult to implement. An example from the tip of the iceberg is shown in Geweke (2004), which identifies software bugs and erroneous results in earlier published studies. We address this issue using a marriage of statistical theory and software engineering methodology, such as compositionality and unit testing.

Ease of use: Blang uses a familiar BUGS-like syntax and it is designed to be integrated well in modern data science workflows (input in "tidy" format, samples output in tidy format).

Generality: The language is not limited to basic datatypes such as Euclidean or finite state spaces. This is achieved with a Turing-complete language equipped with an open type system, as well as facilities to quickly develop and test sampling algorithms for new types.

Computational scalability. The language is designed to ensure that state-of-the-art Monte Carlo methods can be utilized. In particular, we made certain trade-offs to ensure that a well-behaved continuum of distributions can be automatically created. This is complemented with methods that extend existing PPL strategies to combinatorial space, for example code scoping analysis to discover sparsity patterns with arbitrary types, as well as built-in support for parallelization to arbitrary numbers of cores.

3. License and source availability

Blang is free and open source. The language and SDK are available under a permissive

BSD 2-Clause license. The relevant GitHub repositories are linked at `https://github.com/UBC-Stat-ML/blangDoc`.

4. Tutorial

This section aims to introduce readers to Blang by presenting several illustrative examples. We begin with instructions for performing inference on a simple model using the command-line interface (CLI). This is followed by examples incorporating non-standard data types and an outline of custom sampler implementation.

4.1. Installing Blang's command-line interface

We provide instructions here for installing Blang via a CLI. Other installation options, including a graphical user interface (GUI), are available from the documentation website under the link `Tools`. The GUI is also described in Section 7.1. Additionally, an R and Python interface to Blang are currently under development.[1] The prerequisites for the CLI installation process are:

1. A UNIX-compatible environment running `bash`. This includes, in particular, Mac OS X, where `bash` is the default terminal interpreter when launching `Terminal.app`.

2. The `git` command.

3. The Java Software Development Kit (SDK), version 8 or more recent. The `Java` *runtime environment* is not sufficient, as compilation of models requires compilation into the Java Virtual Machine. Type `javac -version` to test if the `Java SDK` is installed. If not, the `Java SDK` is freely available at https://openjdk.java.net/.

The following installation process is most thoroughly tested on Mac OS X, which is the primary supported platform at the moment, however users have reported installing it successfully on certain Linux and Windows configurations and we plan to expand the set of officially supported platform to both in the near future.

To install the CLI tools, input the following commands in a bash terminal interpreter:

```
> git clone https://github.com/UBC-Stat-ML/blangSDK.git
> cd blangSDK
> source setup-cli.sh
```

The `git clone` command downloads the **blangSDK** repository, `cd` changes the current working directory, and `source setup-cli.sh` compiles and installs Blang (i.e., updates the `PATH` variable). If the user moves the **blangSDK** folder, the command `source setup-cli.sh` needs to be reran.

You may now use Blang from any directory by typing `blang` (use lower case for the CLI command as UNIX is case-sensitive).

[1]The interfaces and associated instructions will be hosted on `https://github.com/UBC-Stat-ML`

4.2. Basic CLI usage: Posterior inference

Consider the simplified Doomsday Argument (Carter Brandon and McCrea W. H. 1983) in Figure 1. Using a PPL for such a simple model is excessive but is useful for demonstrating the basic mechanics of Bayesian inference in Blang.

```
package toy

model Doomsday {
  random RealVar z
  random RealVar y
  param RealVar rate
  laws {
    z | rate ~ Exponential(rate)
    y | z ~ ContinuousUniform(0.0, z)
  }
}
```

Figure 1: Doomsday model.

While in **blangSDK**, create an empty directory called `project` and create another directory called `toy` nested within `blangSDK/project/`. Finally in `toy`, create a file called `Doomsday.bl` and paste the above code into it.

The code in Figure 1 illustrates four Blang keywords.

- `model`: there should be exactly one `model` per file. The keyword should be followed by an identifier, in this case `Doomsday`. Blang is a case-sensitive language and we use the convention that model names are capitalized.

- `random` and `param` are used to declare variables. Variables need to specify their type. For example, `random RealVar z` is of type `RealVar` and we give it the name `z`. As a convention, types are capitalized and variable names are not. The difference between `random` and `param` is explained in Section 5.

- Each model is required to have exactly one `laws` keyword followed by a code chunk surrounded by curly braces, called the *laws block*. The purpose of the laws block is to define joint distributions over the random variables. Here, we show one method to do so, which is inspired by the BUGS notation and its derivatives. For example,

  ```
  y | z ~ ContinuousUniform(0.0, z)
  ```

  denotes the conditional distribution of `y` given `z` is equal to a uniform distribution between 0 and `z`. In contrast to BUGS, we require specification of the random variables that we are conditioning on, here `| z`.

By default, the CLI utility `blang` approximates the posterior distribution over the latent `random` variables conditioning on the observed `random` variables. From the `project` directory, type the following command, in which we specify that `rate` and `y` are fixed to given values while `z` is unobserved:

```
> blang --model toy.Doomsday --model.rate 1.0 --model.y 1.2 --model.z NA
```

Alternatively, the same model can be run via a prepackaged repository of examples

```
> git clone https://github.com/UBC-Stat-ML/JSSBlangCode.git
> cd JSSBlangCode/example/
> blang --model jss.Doomsday --model.rate 1.0 --model.y 1.2 --model.z NA

Compilation {
  ...
} [ ... ]
Preprocess {
  Initialization {
    ...
  } [ ... ]
   ...
 } [ ... ]
Inference {
   ...
  Round(9/9) {
   ...
  } [ ... ]
 } [ ... ]
Postprocess {
   ...
 } [ ... ]
executionMilliseconds : 1037
outputFolder: ./JSSBlangCode/example/results/all/2019-06-27-14-13-21-RL.exec
```

Samples approximating the posterior distribution of `z` given the observation `y` are outputted in Tidy format (Wickham 2014) to `z.csv` located in the directory specified by `outputFolder`.

By default, posterior inference is done in two stages. The first stage, corresponding to the `Initialization` block in the standard output, uses an adaptive SCM algorithm that attempts to automatically identify configurations of positive density. In the second stage, an adaptive non-reversible PT algorithm is initialized from the output of the first stage and performs a series of adaptation rounds, corresponding to `Round(1/9)` through `Round(9/9)` blocks in the standard output. PT algorithms are known to perform well even in the face of difficult sampling problems such as those arising in multi-modal distributions or weakly identifiable models. We describe the inference algorithms and their configuration in detail in Section 9.1.

4.3. Inference on a non-standard data structure

Consider an inference problem where the data structure of interest is a phylogenetic tree. A phylogenetic tree is a branching process encoding evolutionary relationships between organisms. The following example illustrates how to perform inference on a phylogenetic tree model given sequence alignment data.

The Blang language itself does not contain tree-valued random variables. However, the language allows *creating* custom types of random variables. Moreover, these custom types can be packaged, published and imported. Here we begin with an example where we use a custom data types developed in a third-party library written in Blang.

First, we create a file called `dependencies.txt` at the root of the project directory. Each line in `dependencies.txt` encodes a versioned third-party library to be imported (along with its transitive set of dependencies). Here we use a Blang package providing phylogenetic centric data types (Zhao *et al.* 2015):

```
com.github.alexandrebouchard:conifer:2.0.4
```

We encode the Blang model, using the import data type and BUGS inspired syntax, as follows:

```
package demo
import conifer.*
import static conifer.Utils.*

model PhylogeneticTree {

  /* Block 1: variable and parameter declaration */
  random RealVar shape ?: latentReal()
  random RealVar rate ?: latentReal()
  random SequenceAlignment observations
  param EvolutionaryModel evoModel ?: kimura(observations.nSites)

  /* non-standard data structure */
  random UnrootedTree tree ?: unrootedTree(observations.observedTreeNodes)

  /* Block 2: define distributions and relationships */
  laws {
    shape ~ Exponential(1.0)
    rate ~ Exponential(1.0)
    tree | shape, rate ~ NonClockTreePrior(Gamma.distribution(shape, rate))
    observations | tree, evoModel ~ UnrootedTreeLikelihood(tree, evoModel)
  }
}
```

Here NonClockTreePrior and UnrootedTreeLikelihood are themselves Blang models defined in the imported package.

To run PhylogeneticTree we enter the following in the CLI:

```
> git clone https://github.com/UBC-Stat-ML/JSSBlangCode.git
> cd JSSBlangCode/example
> blang --model jss.phylo.PhylogeneticTree \
    --model.observations.file data/primates.fasta \
    --model.observations.encoding DNA \
    --engine PT \
    --engine.random 1

Preprocess {
  ...
  Initialization {
    ...
  } [ ... ]
} [ ... ]
Inference {
    ...
  Round(9/9) {
    ...
  } [ ... ]
} [ ... ]
Postprocess {
 ...
} [ ... ]
executionMilliseconds : ...
outputFolder :./JSSBlangCode/example/results/all/2019-06-18-09-42-15-sP.exec
```

We briefly explain the runtime arguments below, full descriptions are provided through `blang --model MyModelName --help`. Here `--model.observations.file` specifies the data path, `--model.observations.encoding` is a model-specific option to parse our data, `--engine PT` selects a PT algorithm as the inference engine, `--engine.nScans` sets the number of samples, and `--engine.random` sets a random seed. Samples are outputted to `shape.csv`, `rate.csv`, `tree.csv` located in the directory specified by `outputFolder`.

4.4. Spike and slab classification

We illustrate a detailed example of implementing and performing inference on a non-standard data type using a spike and slab model (Mitchell and Beauchamp 1988). In contrast to the previous section, we show how the non-standard data type is implemented.

The spike and slab model is a mixture of prior distributions commonly used for coefficients in a regression model. The non-standard data type SpikedRealVar is Blang's representation of the type of the coefficients in a spike and slab model. The data type `SpikedRealVar` is implemented as follows

```
package glms

import blang.core.RealVar
import blang.core.IntVar
import blang.types.StaticUtils

class SpikedRealVar implements RealVar {
  public val IntVar selected = StaticUtils::latentInt()
  public val RealVar continuousPart = StaticUtils::latentReal()

  override doubleValue() {
    if (selected.intValue < 0 || selected.intValue > 1)
      StaticUtils::invalidParameter()
    if (selected.intValue == 0) return 0.0
    else return continuousPart.doubleValue
  }
  override toString() { "" + doubleValue }
}
```

We declare its member variables, `selected` and `continuous`, as `IntVar` and `RealVar`. These variables will encode the spike and slab component values for each explanatory variable. Because these members are random, their values are initialized using `latentInt()` and `latentReal()`. We `override` `RealVar()`'s getter method `doubleValue()` to return the regression coefficient if the explanatory variable is selected.

We can now use this custom data type to build a simple classification model:

```
package glms

model SpikeSlabClassification {

  param GlobalDataSource data
  random RealVar activeProbability ?: latentReal
  random RealVar sigma ?: latentReal
  random RealVar intercept ?: latentReal

  // plate and plated variables
  param Plate<String> instances, features
  param Plated<Double> covariates
  random Plated<IntVar> labels
  random Plated<SpikedRealVar> parameters // custom data type
  laws {
    for (Index<String> instance : instances.indices) {
      labels.get(instance) | intercept,
      DotProduct dotProduct
      = DotProduct.of(features, parameters, covariates.slice(instance))
        ~ Bernoulli(logistic(intercept + dotProduct.compute))
    }

    for (Index<String> feature : features.indices) {
      parameters.get(feature).selected | activeProbability
        ~ Bernoulli(activeProbability)
      parameters.get(feature).continuousPart | sigma
        ~ StudentT(1.0, 0.0, sigma)
    }

    intercept | sigma ~ StudentT(1.0, 0.0, sigma)
    activeProbability ~ ContinuousUniform(0, 1)
    sigma ~ Exponential(1.0)
  }
}
```

In the above model, the random variable `parameters` is indexed by `instances` and `features`. This relationship is encoded using built-in types `Plated` and `Plate` variables; where a `Plated` variable is indexed by one or more `Plate` variable. Hence, `parameters` is of type `Plated<SpikedRealVar>` and both `instances` and `features` are of type `Plate<String>`. `Plate` and `Plated` variables are detailed in Section 7.4.1.

To perform inference on model `SpikeSlabClassification`, we call the following in the CLI:

```
> git clone https://github.com/UBC-Stat-ML/JSSBlangCode.git
> cd JSSBlangCode/example
> blang --model jss.glms.SpikeSlabClassification \
    --model.data data/titanic/titanic-covariates.csv \
    --model.instances.name Name \
    --model.instances.maxSize 200 \
    --model.labels.dataSource data/titanic/titanic.csv \
    --model.labels.name Survived \
    --engine PT \
    --engine.nChains 20 \
    --postProcessor DefaultPostProcessor

Preprocess {
    ...
```

```
  Initialization {
    ...
  } [ ... ]
} [ ... ]
Inference {
    ...
  Round(9/9) {
    ...
  } [ ... ]
} [ ... ]
Postprocess {
  Post-processing activeProbability
  Post-processing allLogDensities
  Post-processing energy
  Post-processing intercept
  Post-processing logDensity
  Post-processing parameters
  Post-processing sigma
  MC diagnostics
} [ ... ]
executionMilliseconds : ...
outputFolder :./JSSBlangCode/example/results/all/2019-06-18-10-05-29-ut.exec
```

The arguments `--model.data` and `--model.labels.dataSource` specify the data source, `--model.labels.name` and `--model.instances.name` specify the column names that `labels` and `instances` correspond to, and `--model.instances.maxSize` indicates the maximum size of the variable. The `--postProcessor` command creates additional summary statistics, posterior plots, trace plots, monitoring plots, and effective sample size in addition to the default outputs. These are further discussed below. For a preview, summary statistics for `SpikeSlabClassification` model's `parameters` are shown below, and can be found under the directory `summaries` in `results/latest`.

index	features	mean	sd	min	median	max
1	Age	−0.022	0.023	−0.097	−0.016	0.001
2	child	1.089	1.049	−1.145	0.950	5.958
3	Fare	0.000	0.001	−0.014	0.000	0.015
4	female	3.142	0.436	1.810	3.131	5.073
5	Parents.Children.Aboard	0.017	0.135	−0.640	0.000	0.951
6	Pclass	−0.479	0.287	−1.312	−0.499	0.130
7	Siblings.Spouses.Aboard	−0.705	0.281	−2.000	−0.693	0.000

All experiment outputs are stored in a `results` directory, within the working directory in which the `blang` CLI command is called. Generally, there are three categories of outputs: samples, post-processed statistics and plots, and monitoring statistics and plots.

Sample outputs are in Tidy format, with each variable stored in individual `.csv` files. Options for post-processing is handled via the `--postProccesor` runtime argument, accepting

`DefaultPostProcessor` or `NoPostProcessor` as arguments.

Currently, the `DefaultPostProcessor` option produces sets of trace and density plots, and provides summary statistics including effective sample size (ESS) metrics, based on a numerically robust version of the $\sqrt{n}$-size batch estimator described in Flegal and Jones (2010). Type information is used to select appropriate plotting strategies (e.g., probability mass functions for `IntVar` types, density estimates for `RealVar`). Figures 2 and 3 are examples of the density and trace plots respectively.

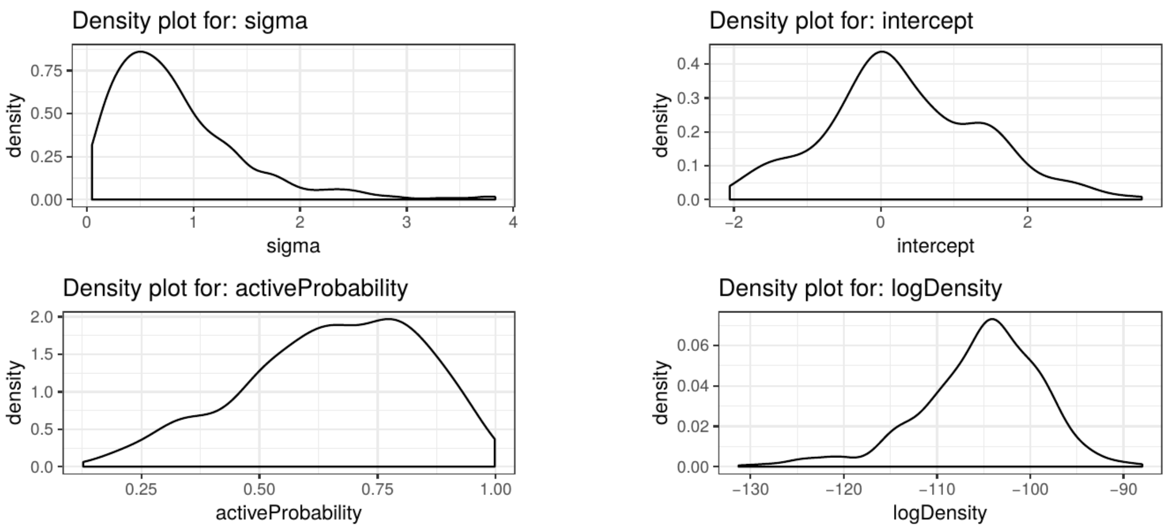


Figure 2: Posterior density plots for a subset of random variables in the spike and slab model.

Metadata for diagnosing and monitoring the performance of inference algorithms are also produced. Under the SCM algorithm (Section 9.1), propagation and resampling statistics are logged and output, we refer the reader to Del Moral *et al.* (2006); Zhou *et al.* (2016) for interpretation.

In PT (Section 9.1), several diagnostics are output to monitor algorithm performance, such as round trip rates, communication barriers, swap statistics, annealing parameters and schedule, to name a few. Visual summaries are often useful for interpretation and thus included in the `monitoringPlots` directory. Figures 4 to 6 showcase a number of these plots. For a comprehensive explanation of these statistics, we refer readers to Syed *et al.* (2019).

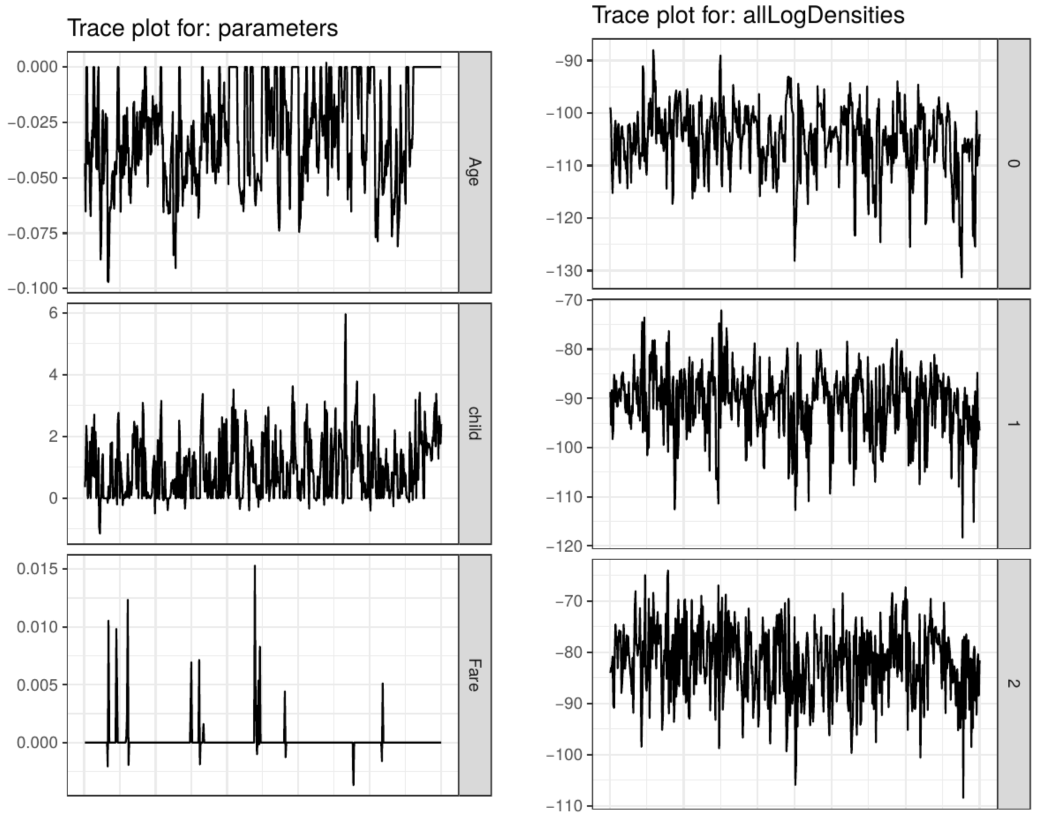


Figure 3: Trace plots for a subset of various random variables in the spike and slab model. Left: the coefficients visit zero with positive probability as expected. Right: log densities for 3 of the 20 tempered chains used in PT.

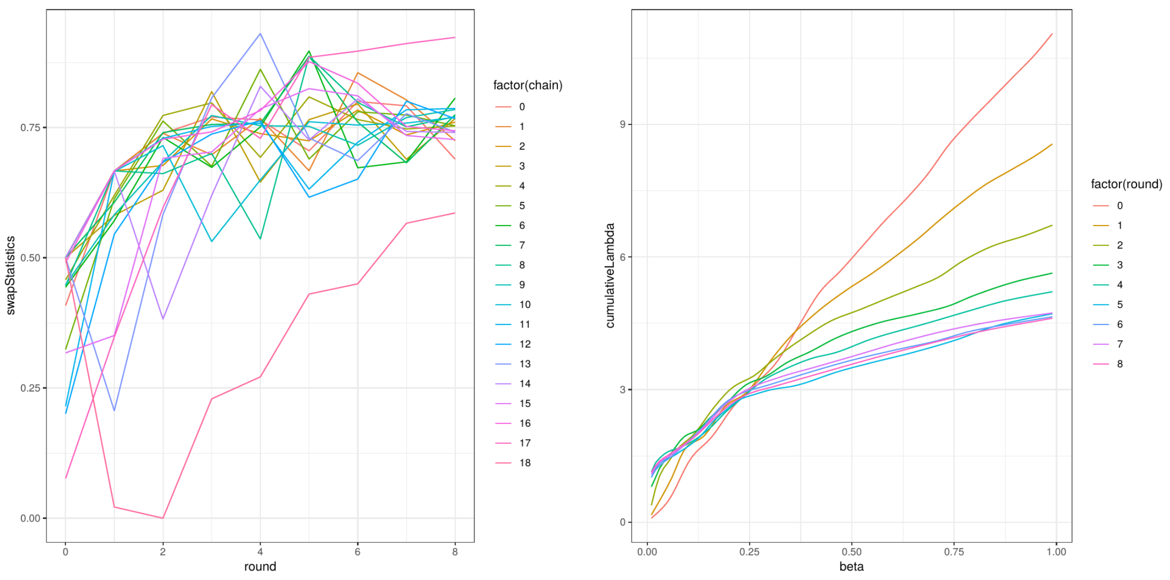


Figure 4: Monitoring plots: swap statistics progress across PT schedule adaptation rounds (left, in which colours encode tempered chains), estimation of the cumulative rejection rate used as the basis of PT schedule adaptation (right, in which colours encode adaptation rounds).

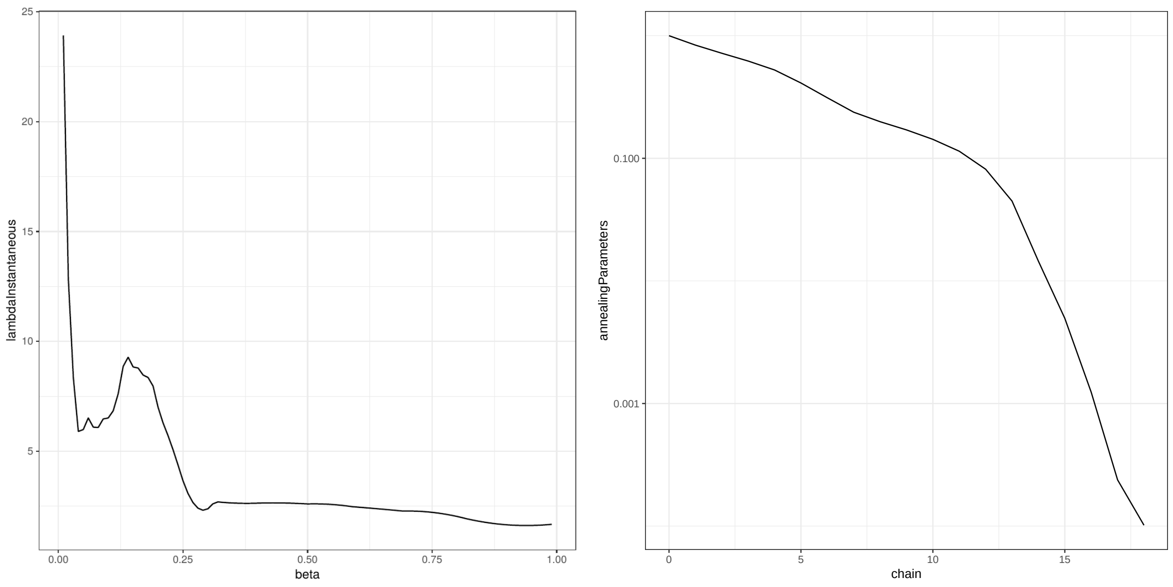


Figure 5: Monitoring plots: instantaneous rejection rate (left), annealing parameter values at different tempered chains (right).

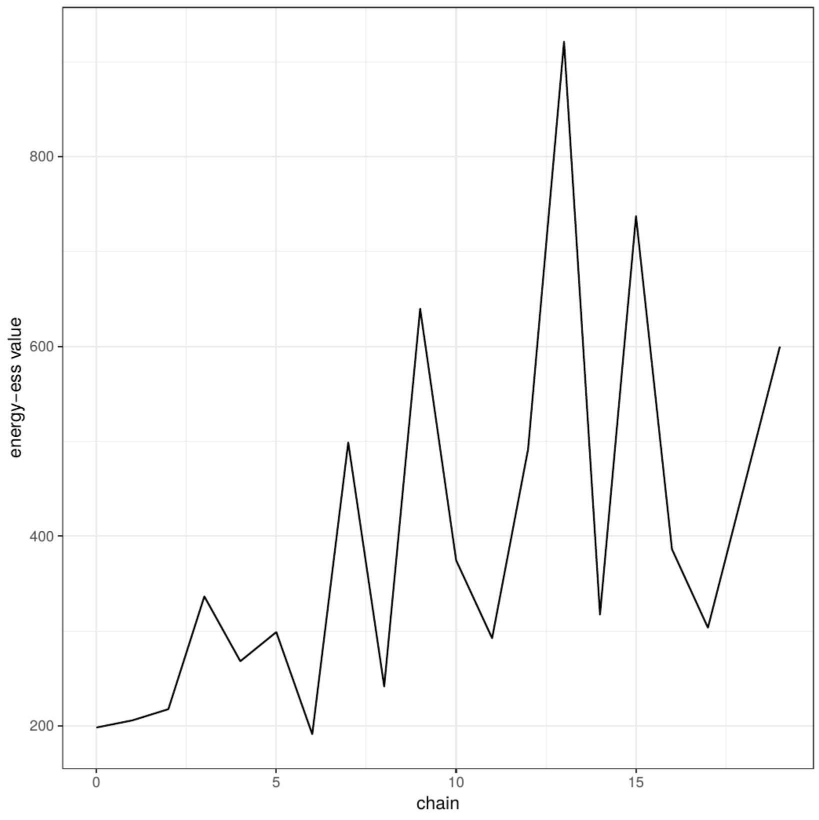


Figure 6: Monitoring plots: effective sample size of the energy statistics at different tempered chains.

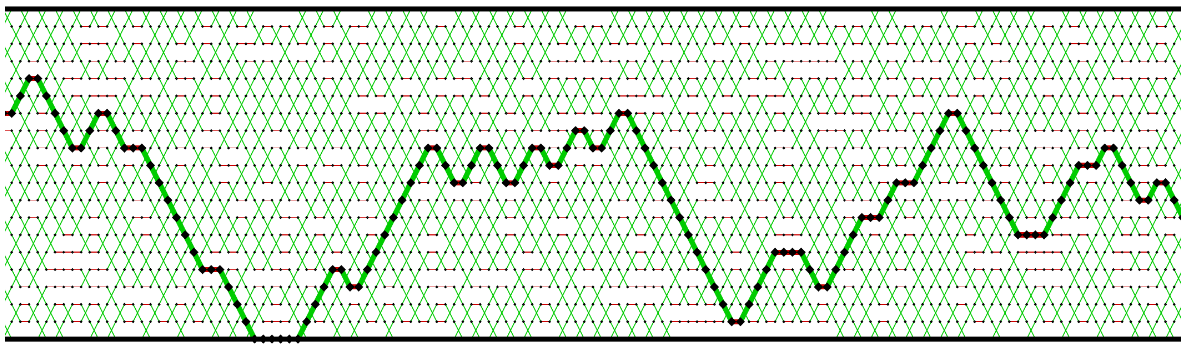


Figure 7: Plot of chain swap paths (cropped). The x-axis corresponds to PT iterations; the y-axis are chain indices.

4.5. Custom samplers for custom data structures

In the previous section, sampling `SpikedRealVar` variables could be handled via default samplers. This luxury is typically unavailable when working with complex state spaces such as trees, partitions, or permutation spaces. In such situation, Blang still assists the user in several ways described in more detail in Section 7. Here we focus on how Blang helps establish the correctness of the implemented sampler.

Consider a model with latent variables taking values in a set of permutations (perfect bipartite matching). For example, *record linkage* problems (Tancredi and Liseo 2011; Steorts *et al.* 2016) rely on this type of latent variables. In short, record linkage is the process of matching de-identified noisy records from multiple data sources that reference the same entity. In the following, we demonstrate how to implement a custom sampler for data of type permutation.

The first step is to implement a *class* describing how permutations will be represented. Without exposing distracting implementation details, we first note a permutation will be stored as a list of integers (full details are available in the reproduction materials located in the directory `reproduction_materials/others`):

```
...
@Samplers(PermutationSampler)
class Permutation {
  val List<Integer> connections
  ...
}
```

In the above, we show a partial implementation of the `Permutation` class. The annotation `@Samplers( ... )` informs the runtime engine to sample `Permutation` objects with an instance of `PermutationSampler`, which we define next. More than one samplers can be specified as a comma-separated list. All samplers must implement the `blang.mcmc.Sampler` interface, meaning that the samplers must define a method called `execute`:

```
class PermutationSampler implements Sampler {
  @SampledVariable Permutation permutation
  @ConnectedFactor List<LogScaleFactor> numericFactors

  override void execute(Random rand) {
    ...
  }
}
```

The field annotated with `@SampledVariable` will automatically be populated with the object to be sampled, in this example, an object of type `Permutation`. Similarly, the field annotated with `@ConnectedFactor` will automatically be populated with factors dependent on the sampled object, as inferred automatically via a factor graph built from scope analysis (described in detail in Section 9).

We now have the components necessary to implement the `execute` method.

```
override void execute(Random rand) {
   val n = permutation.componentSize
   val i = rand.nextInt(n)
   val j = rand.nextInt(n)

   val log_pi_current = logDensity()
   permutation.swapConnections(i, j)
   val log_pi_new = logDensity()

   val accept_prob = min(1.0, exp(log_pi_new - log_pi_current))
   val accept_proposal = rand.nextBernoulli(accept_prob)
   if (!accept_proposal) {
      permutation.swapConnections(i, j)
   }
}
```

We assume our readers are familiar with the Metropolis algorithm (Metropolis *et al.* 1953). The `logDensity()` method returns the sum of log factors. The `execute` method is invoked with each iteration of the inference algorithm (Section 9.1), and updates our variable of interest in-place. With this, the implementation of the custom sampler is completed, and is able to be used with distributions defined on a permutation space.

An example of a uniform distribution over the permutation space is implemented as follows:

```
model UniformPermutation {
  random Permutation permutation

  laws {
    logf(permutation) {
      - logFactorial(permutation.componentSize)
    }
  }

  generate(rand) {
    permutation.sampleUniform(rand)
  }
}
```

As with any distribution, it can be used in composition with more complex models.

```
model ComplicatedModel {

  random Permutation p
  ...

  laws {
    p ~ UniformPermutation()
    ...
  }
  ...
}
```

Finally, we now demonstrate some of the resources available to assist in testing the correctness of the newly created sampler. A first test utility is `DiscreteMCTest`, which is specialized to fully-combinatorial spaces. The idea behind `DiscreteMCTest` is that for small combinatorial spaces, we can explicitly form a sparse transition matrix and numerically check properties such as invariance and irreducibility. In our experience, many software defects can be found in problem just large enough to achieve code coverage. Setting up a `DiscreteMCTest` is done as follows:

```
  val test = new DiscreteMCTest(
    buildMatchingModelOfSize(5),
    [connectionsCopy(model)]
  )
  test.checkInvariance
  test.checkIrreducibility
```

When creating a `DiscreteMCTest` object, one should pass in the small discrete model to be tested as the first argument and as the second argument, a lambda function (denoted by square brackets) that accepts a model and creates a new object encoding the identity of the current configuration, identity being mediated by the `.equals()` function of the returned objection. As `DiscreteMCTest` is created, the samplers involved in the input model are automatically translated into explicit sparse transition matrices, via a type of non-standard evaluation of the sampling code. When invoking `checkInvariance`, the stationary distribution is computed as a vector, and multiplied to the sparse transition matrix. The product is expected to be equal up to numerical precision to the stationary matrix. Invoking `checkIrreducibility` uses graph algorithms to ensure that the sampler can reach all states.

Additional testing resources are discussed in Section 7.5, in particular to handle continuous state spaces.

5. Conceptual overview

We now describe more formally the semantics of our language's core construct: the `model`. The basic notation introduced here will be useful to describe the syntax in full detail in the next section.

5.1. Models

A Blang `model` encodes a set of densities $\{f_\theta(x) : \theta \in \Theta, x \in T\}$, and hence the distribution of a random object $X : \Omega \to T$. We use the term "density" in a generalized sense, encompassing discrete, continuous, and mixed models, by allowing it to be defined with respect to customizable reference measures.

We assume $x = (x_1, x_2, \dots, x_n)$ where $n < \infty$ is fixed. Formal expressivity is preserved as each x_i is permitted to be of random or infinite dimensionality, despite n being finite. The type of x_i is denoted by T_i. Hence $x_i \in T_i$ and $x \in T = T_1 \times T_2 \times \cdots \times T_n$. We also assume each type, T_i, is implicitly associated with a default reference measure μ_i. These default choices can be changed using the `is` keyword defined in Section 6.10. Once each reference measure, μ_i, is given, the densities are turned into distributions as follows:

$$\mathbb{P}_\theta(X \in A) = \int_A f_\theta(x) \prod_{i=1}^{n} \mu_i(\,\mathrm{d}x_i). \tag{1}$$

Similar to x, we assume a decomposition for the parameters $\theta = (\theta_1, \theta_2, \dots, \theta_m)$ where m is fixed and each coordinate θ_j has its type denoted by Θ_j. Hence, $\theta_j \in \Theta_j$ and $\theta \in \Theta = \Theta_1 \times \Theta_2 \times \cdots \times \Theta_m$. We use the terminology "model variable" to refer to x and θ collectively.

To understand how these mathematical concepts translate into Blang syntax, let us relate them via the Doomsday example from Section 4. The correspondence is shown in Figures 8 and 9. The variables marked with the `random` keyword are concatenated to form x, while those marked with `param` keyword are concatenated to form θ.

```
model Doomsday {

  param RealVar rate
  random RealVar y
  random RealVar z

  laws { ... }

}
```

Figure 8: Blang Syntax.

$$\begin{aligned} \theta &= (\texttt{rate}) \\ x &= (\texttt{y}, \texttt{z}) \\ \{f_\theta\} &= \{\texttt{Doomsday(rate)}\} \\ \Theta_1 = T_1 &= T_2 = \texttt{RealVar} \end{aligned}$$

Figure 9: Mathematical notation.

5.2. Interpretation of `laws` blocks

The `laws` block is responsible for computing the point-wise evaluation of $f_\theta(x)$ (in log scale) for any input x and θ. To do so, two methods are supported:

Composite laws, which use existing Blang models as building blocks to create a new one.

Atomic laws, which provide an arbitrary algorithm to compute the log density.

Both composite and atomic laws allow the user to express a known factorization of the density

$$f_\theta(x) = \prod_{k=1}^{K} f^{(k)}(x, \theta). \tag{2}$$

Such a factorization can then be used as the basis of automating key aspects of state-of-the-art Monte Carlo methods, such as the construction of a well-behaved continuum of auxiliary distributions and the detection of sparsity patterns.

5.3. Interpretation of atomic laws

In the case of an atomic law, for each $k \in \{1, 2, \dots, K\}$, an expression or algorithm is provided to compute the value of factor k in log scale, i.e., $\log\left(f^{(k)}(x, \theta)\right)$.

For example, consider the continuous uniform distribution, which can be factorized as

$$f_\theta^{\text{unif}}(x) = \underbrace{\frac{1}{\theta_2 - \theta_1}}_{f^{(1)}(x)} \; \underbrace{\mathbf{1}[\theta_1 \le x \le \theta_2]}_{f^{(2)}(x)},$$

where $\theta = (\theta_1, \theta_2) = (\texttt{min}, \texttt{max})$. The `model` defining a `ContinuousUniform` distribution in the Blang SDK, encodes this factorization as follows.

```
model ContinuousUniform {
  random RealVar realization
  param  RealVar min
  param  RealVar max

  laws {
    logf(min, max) {              // First factor
      if (max - min <= 0.0) return NEGATIVE_INFINITY
      return - log(max - min)
    }
    logf(realization, min, max) {  // Second factor
      if (min <= realization && realization <= max) return 0.0
      else return NEGATIVE_INFINITY
    }
  }
  ...
}
```

5.4. Interpretation of composite laws

In the case of a composite law, the decomposition in Equation 2 typically comes from an application of chain rule. In the Doomsday example, this is just:

$$f_\theta^{\text{Dooms}}(x) \;=\; \underbrace{\theta_1 \exp(-\theta_1 x_2)}_{\tilde f^{(1)}(x,\theta)} \; \underbrace{\frac{\mathbf{1}[0 \le x_1 \le x_2]}{x_2}}_{\tilde f^{(2)}(x,\theta)}. \tag{3}$$

To understand composite laws, notice the factors in this decomposition can often be retrieved from another existing `model`. In such a case, we say that a `model`, $\{f_\theta^{\text{caller}}(x) : x \in T, \theta \in \Theta\}$, *calls* another model, $\{f_{\theta'}^{\text{callee}}(x') : x' \in T', \theta' \in \Theta'\}$. This is illustrated in our running example as the `Doomsday` model, the caller, calls the `ContinuousUniform` model, the callee. Consequently allowing us the write the second factor in Equation (3) using the previously defined `ContinuousUniform` model via

$$\tilde f^{(2)}(x,\theta) \;=\; f_{t(x,\theta)}^{\text{unif}}(s(x)), \tag{4}$$

for $t(x,\theta)$ and $s(x)$ defined as follows.

First, $t : T \times \Theta \to \Theta'$ is a transformation from the *caller* model's variables into the *callee* model's parameters, in this case $t(x,\theta) = (0, x_2)$. The two entries in the list $(0, x_2)$ correspond to the two `param` variables, `min` and `max`, in the definition of `ContinuousUniform` shown in Section 5.3. We see that the order in which the `param` are declared is important when a `model` is to be used in a composite fashion.

Second, $s : T \to T'$ is a selection of a subset $i_1, \dots, i_{|x'|}$ of coordinates in x, so that $s(x) = (x_{i_1}, \dots, x_{i_{|x'|}})$. Hence, s selects which of the calling model's random variables are used as the callee model's random variables. Here $s(x) = (x_1)$, where the single entry, (x_1), corresponds to the `random` variable, `realization`, in the definition of `ContinuousUniform`. Again, if more than one random variable is selected, the order in which they are declared in the callee model determines how they are matched.

Considering now the Blang statement,

```
  y | z ~ ContinuousUniform(0.0, z)
```

we see that the left of the pipe symbol, `|`, encodes the selection s, and the expression in parentheses encodes the transformation t.

In summary, the two lines in the laws block of the Doomsday model,

```
  z | rate ~ Exponential(rate)
  y | z ~ ContinuousUniform(0.0, z)
```

have the same interpretation as they would in probability theory. However, by our definition of notation, with s and t, it can also be extended to useful novel patterns. We describe two PPL design patterns that emerge from our formal construction of composite laws in Section 8.

5.5. Model tree

Composite laws induce a directed tree over models, where a directed edge denotes a `model` calling another `model`. We call this tree the *model tree*. The root of this tree is called the *root model*.

5.6. Interpretation of `generate` blocks

In addition to the atomic and composite constructs available to specify a mandatory `laws` block, Blang provides an optional orthogonal way to specify $\mathbb{P}_\theta(X \in A)$, called a `generate` block. The `generate` block performs "forward simulation": it takes as input a random seed, $\omega \in \Omega$, and returns $X(\omega)$ such that Equation (1) holds.

The `generate` block is technically redundant, but is crucial to check software correctness by setting up statistical unit tests as described in Section 7.5. It is also used for various purposes during posterior inference, for example, by providing a form or regeneration in PT.

5.7. Normal form

Although syntactically optional, the main inference engines used in Blang make certain assumptions about availability of generate blocks. Some terminology: if a given `laws` block contains either only composite laws or only atomic laws, we say it is in *normal form*. We say a model is in *generative normal form* if it satisfies the following conditions. First, all models in the model tree should be in normal form. Second, all models in the model tree which are based on atomic laws attached to unobserved random variables should be equipped with a generate block.

We show in Section 8.1 how to rewrite a wide range of models into a generative normal form. If a model cannot be written in generative normal form, the user may still apply standard MCMC methods but not the more advanced PT and SCM schemes.

5.8. From Blang models to posterior inference

Any Blang model can be transformed into a posterior inference computer program. The inputs of this computer program consists of variables in the root model. All `param` variables in the root model become required inputs. In contrast, `random` variables in the root model can either be specified or left missing as latent. The target posterior distribution is then defined as the distribution of latent random variables given the variables that have been given an input value.

6. A complete tour of Blang's syntax

In this section we provide a more systematic survey of the Blang language. The formal definition of the language can be accessed in the **blangDSL** repository at https://github.com/UBC-Stat-ML/blangDSL.

6.1. Project organization

Blang projects are composed of three types of files: Blang files (`.bl`), Xtend files (`.xtend`), and Java files (`.java`). This section is devoted to the syntax of Blang files. Xtend and Java files are used to create supporting code for non-standard datatypes, samplers, and user-defined functions. The user can choose either Xtend or Java for creating supporting code. For users not familiar with Java, we recommend using Xtend because its syntax is consistent with Blang's syntax. This is a consequence of both languages being constructed with the Xtext language development framework.

6.2. Interoperability with Java

Blang, Xtend and Java are seamlessly interoperable as the first two are transpiled into Java. More precisely, any Java type can be imported and used in Blang, and any model defined in Blang can be imported and used in Java with no extra work needed.

As such, types in Blang are equivalent to *Java types*, a terminology that encompasses Java classes, interfaces, primitives, enumerations and annotation interfaces. At a high-level, a type can be thought as a group of *objects* (chunks of computer memory) that satisfy a certain set of properties (for example, they all support being passed in a certain function). We do not assume prior knowledge of the Java language, in fact, Blang and Xtend syntax is often simpler compared to Java's.

6.3. Comments

Single line comments use the syntax

```
// some comment spanning the rest of the line.
```

Multi-line comments use

```
/*
many commented lines
can go here
*/
```

6.4. Blang models: High-level syntax

A Blang file is organized as follows:

```
// package and import statements

model NameOfMyModel {

  // variables declarations

  laws {
    // laws declaration
  }

  generate(nameOfMyRandomObject) { // optional
    // generate block
  }
}
```

In the remainder, if a string such as `NameOfMyModel` contains the substring "`My`", or has an integer as suffix, it refers to an identifier that should be tailored to the context of the model being written.

As a general rule, identifiers (model names, variable names, etc) should start with a letter and only use letters, number, and underscores. Furthermore, as a convention we encourage users to capitalize model names as Blang is case-sensitive.

6.5. Packages and imports

The *packages* construct deals with the rare, but unavoidable, situation of wanting to use code from two developers that used the same name for a Blang `model`. Package declarations will disambiguate the two.

To declare a Blang `model` as part of a hierarchical group of related code, place the following declaration at the very beginning of the Blang file:

```
package myOrganization.myPackageName
```

This *package declaration* line is optional but recommended if you plan to share your code. The dot in `myOrganization.myPackageName` denotes a hierarchical organization going from broader to more specific from left to right. As a convention, package names are generally not capitalized.

To use your Blang `model` called `MyImportedModel` from a package named `my.imported.pack`, the package declaration shown previously should be followed by an `import` statement of the form:

```
import my.imported.pack.MyImportedModel
```

The same syntax can be used to import Java or Xtend classes. In the unlikely event of having to use two types with duplicated names within the same file, importing should be avoided and instead each instance of the type should be prefixed with the package name within the code, as such:

```
package myOrganization.myPackageName

model MyModel{
  random package1.DupedType var1
  random package2.DupedType var2

  laws {
    ...
  }
}
```

A related construct is the extension import mechanism, described in more detail in Section 6.11.

6.6. Automatic imports in Blang files

Any Blang file automatically imports:

- all the types in the following packages:
  `blang.core`,
  `blang.distributions`,
  `blang.io`,
  `blang.types`,
  `blang.mcmc`,
  `java.util`,
  `xlinear`;

- all the static function in the following files:
  `xlinear.MatrixOperations`,
  `bayonet.math.SpecialFunctions`,
  `org.apache.commons.math3.util.CombinatoricsUtils`,
  `blang.types.StaticUtils`;

- as static extensions all the static functions in the following files:
  `xlinear.MatrixExtensions`,
  `blang.types.ExtensionUtils`.

6.7. Model variables

Variables are declared using one of two methods:

```
// method 1: no default initialization
random Type1 name1
param Type2 name2

// method 2: initialize with XExpressions
random Type3 name3 ?: XExpression1
param Type4 name4 ?: XExpression2
```

Each variable is declared as `random` or `param` to specify a random variable or a parameter variable. The initialization blocks, denoted by `XExpression1` and `XExpression2`, are used in the model to provide default values in the absence of CLI arguments. Moreover, these initialization blocks can use values of previously listed variables. If a CLI argument is provided, then the initialization block will be overridden by it.

The expressions in initialization blocks are constructed with so called *XExpressions*. XExpressions are introduced in more detail in Section 6.11 and are used to construct several aspects of Blang programs. For now, think about XExpressions as chunks of code capable of performing arbitrary computations (loops, conditionals, creating temporary variables, calling other functions, etc), and returning one value.

6.8. Laws block

Composite laws

Composite laws, described conceptually in Section 5.4, have the following syntax in Blang:

```
variableExpression1, variableExpression2, ...
  | conditioning1, conditioning2, ...
    ~ MyDistributionName(argumentExpression1, argumentExpression2, ...)
```

In the above code block, `MyDistributionName` refers to another Blang model. Each element in `argumentExpression1, argumentExpression2, ...` is matched from left to right in the same order as the `param` variables are declared in the model `MyDistributionName`.

The list `(argumentExpression1, argumentExpression2, ...)` corresponds to the transformation $t : T \times \Theta \to \Theta'$ in the notation used in Section 5.4. This is implemented by allowing each element in `argumentExpression1, argumentExpression2, ...` to be an XExpression which is recomputed each time the value of the density $f_\theta(x)$ is queried.

Each element in `variableExpression1, variableExpression2, ...` is matched from left to right in the same order as the `random` variables are declared in model `MyDistributionName`.

To relate this to Section 5.4, the list `variableExpression1, variableExpression2, ...` corresponds to the output of the selection function $s : T \to T'$. This is implemented by allowing each `variableExpression` to be an XExpression which is executed only once, at initialization time. Often this XExpression is only a variable name, but it could also be an expression selecting an entry in a list or vector.

The conditioning block, `conditioning1, conditioning2, ...` is used to restrict what can be accessed by the transformation t. This is called the *scope* of the transformation t. It is useful to restrict the scope as much as possible since this restriction induces sparsity patterns in the model. Sparsity is then exploited by our efficient inference algorithms.

Specification of the scope is implemented as follows. Each item within `conditioning1, conditioning2, ...` can take one of two possible forms. First, it can be one of the variable names declared via the keyword `random` or `param`. For example, this first method is used in all conditionings of the Doomsday model (see Figure 8).

The second method to specify a conditioning is as follows:

```
MyType myConditioningVaribale = XExpression
```

where `MyType` is a type, `myConditioningVarible` is a local variable name, and the code in `XExpression` has access to all variables. The `XExpression` code is executed only once at initialization. We show in Section 8.2 an example of the typical use case for this initialization process, where in a model for a Markov Chain, this initialization is simply to select, in a list of random variables, the variable corresponding to the previous time step.

Atomic laws

Atomic laws, described conceptually in Section 5.3, have the following syntax in Blang:

```
logf(expression1, expression2, ...) { XExpression }
```

Recall that in Section 5.3, each atomic law was denoted as $\log\left(f^{(k)}(x,\theta)\right)$. Here the list `expression1, expression2, ...` is used to restrict the scope of $f^{(k)}$, with the same motivation and mechanism as for composite laws, described in the last section. Each item in the list `expression1, expression2, ...` follows the same syntax as the items in `conditioning1, conditioning2, ...` also described in the last section.

The `XExpression` is responsible for computing the numerical value of $\log\left(f^{(k)}(x,\theta)\right)$, and as such, should return a value of type Double. The `XExpression` is recomputed each time the value of the density $f_\theta(x)$ is queried.

Declarative loops

In practice, the factorization in Equation (2) may have a large number of factors. To assist the user in declaring these factors, we provide a "declarative loop" construct:

```
for (MyIteratorType myIteratorName : XExpression) { ... }
```

This will repeat all the declarations inside `...` be they atomic or composite. Loops can be nested with the expected cross product behaviour.

The `XExpression` should return an object of type `java.lang.Iterable`. Some important loop idioms:

- Simple loop from 0 (inclusively) to 10 (exclusively):
  `for (Integer i : 0 ..< 10) { ... }`,
- Loops based on a `Collection`, which offer a wide choice of data structures via Java's SDK[2] or Google's Guava project[3].
  An example iterating over a power set[4]:
  `for (Set<Integer> s : (0 ..< 5).powerSet) { ... }`
- Loops based on functional programming, either based on Xtend's utilities or Java's.[5]
  An example iterating over the first four even integers:
  `for (Integer i : (0 ..< 10).filter[it % 2 == 0]) { ... }`
- The `Plate` data structure supplied by the Blang SDK is described in more detail in Section 7.4.

The current runtime infrastructure assumes that the `XExpression` specifying the range should not be random, in particular, it should not change during sampling. As such it is only computed at initialization. Therefore, declarative loops, which *surround* atomic and composite laws, are different than the loops *within* `XExpressions`. Sampling of infinite dimensional objects can be handled by creating dedicated types and/or using `XExpression` loops *inside* a `logf` block.

[2]`https://docs.oracle.com/javase/tutorial/collections/index.html`
[3]`https://github.com/google/guava/wiki/CollectionUtilitiesExplained`
[4]This requires the import line `import static extension com.google.common.collect.Sets.powerSet`
[5]Documentation for Xtend's utilities available here and documentation for Java's streams is available here.

6.9. Generate block

We formalize the syntax used to encode the generate block introduced conceptually in Section 5.6.

The `generate` block is optional and uses the following syntax:

```
generate(myRandomSeed) {
  XExpression
}
```

The argument `myRandomSeed` can be interpreted as the outcome $\omega \in \Omega$, from which the `XExpression` should form the realization $X(\omega)$. No type for `rand` is provided because the argument is always a subtype of `java.util.Random`.

If the `model` has exactly one `random` variable of type `IntVar` or `RealVar`, then the `generate` block should return an `int` or `double` respectively, corresponding to the new realization. Otherwise, the generate block should modify the `random` variable(s) in-place.

6.10. Latent random variables and their reference measures

Each type of `random` variable which we would like to be latent is required to declare one or more sampling algorithms. This is done by adding the following *type annotation* in the Xtend or Java class for that data type:

```
@Samplers(MySampler1, MySampler2, ...)
class MyDataType {
  ...
}
```

Here each item in the list `MySampler1, MySampler2, ...` should be subtypes of the interface `blang.mcmc.Sampler`.

Implicitly, the samplers associate a default reference measure to the latent `random` variables. It may be necessary to overwrite these default reference measures for a particular `random` variable. In such cases, Blang provides a mechanism to change them by adding in the laws block, a line of the following form.

```
myVariableName is MyConstrained
```

In the above, `myVariableName` refers to the `random` variable name for which the default reference measure is to be changed and the type `MyConstrained` should be a subtype of `blang.core.Constrained`. Effectively, the intended behaviour is to disable samplers which would be inoperative with the alternate choice of reference measure. For example, this is used to disable the default coordinate-wise slice sampling for a Dirichlet distributed random variable: because of the simplex constraint, the default sampler would always reject proposals.

6.11. XExpressions

Syntax for XExpressions is provided by the Xtext language engineering framework. Here, we review the key aspects used commonly in Blang programs. We refer the reader to the Xtext documentation for more information.[6]

XExpressions can be either a single instruction as in the argument of the following Exponential composite law:

```
y | a, b, x ~ Exponential(exp(a * x + b))
```

or there can be several instructions nested in braces, as in this equivalent version of the above code

```
y | a, b, x ~ Exponential({
  val product = a * x
  exp(product + b)
})
```

Types

We classify types into three main categories: primitives, object references, and array references. The most common primitives are `boolean`, `int`, and `double`.[7] Object references can be thought of as an annotated address to a memory location, possibly `null`. Lastly, array references are rarely used directly in Blang. Instead, arrays are typically encapsulated in more convenient data structures.

[6]documentation page can be found at https://www.eclipse.org/Xtext/documentation/index.html

[7]They have the same characteristics as in Java, see https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html for technical details.

Literals

Examples of expressions that create constants:

- `boolean`: `true`, `false`
- `int`: e.g., `42`, `12000`
- `double`: e.g., `1.0`, `1.3e2`. Note be sure to include the decimal suffix or use scientific notation.
- String literals: either via `"A"`,
  or `'''This version allows "quotes inside" and more'''`
- type literals: e.g., `String` is equivalent to `Java`'s `String.class`.
- `List`: e.g., `#[true,false]`
- `Set`: e.g., `#{"A","C","G","T"}`
- `Pair` e.g., `"likelihood" -> -123.43`. Note can be used with arbitrary key and value types.
- `Map` e.g., `#"key1" -> 1, "key2" -> 2`

Declaring variables with XExpressions

Local variables have to be declared at their first occurrence. The main syntax variant to do so are:

```
var String myModifiableInt = 17
var typeInferred = [1,2,3]
val int myConstantInt = 17
```

In the example, `var` encodes a variable that is mutable whereas `val` encodes a variable that is immutable. The meaning of immutability is simple to understand in the case of a primitive, but it should be interpreted carefully in the context of references. In the latter, it means that the reference will always point to the same object in the heap, however the internal state of that object might change over time.

`typeInferred` illustrates that the type is inferred automatically, in our example a `List`. This automatic type inference is often, but not always, possible.

Conditionals

Conditional expressions have the following form:

```
if (condition) {
   // do something
}
```

Optionally, they can have an else clause. Also, the pair of if and else is an expression (i.e., returns a value):

```
val String variable = if (condition) "firstString" else "secondString"
```

If an else clause is not included, a `null` alternative is used implicitly to maintain an expression interpretation.

Scope

The *scope* of a variable is defined as the portion of code in which the variable applies and can be accessed. Scoping in Blang is similar to most languages where in order to find the scope of a variable we identify the parent braces and determine the region of the code where the variable can be accessed. For example, a local variable declared within a the body of a `for` loop (the regions between curly braces) cannot be accessed outside of the body. If one variable reference is in the scope of several variables declared with the same name, then the innermost braces has priority.

The only exception are the arguments of the atomic and composite laws. These laws require explicit identification of the variables to include in the scope where these variables should be identified at the right of the `|` symbol.

XExpression loops

In addition to allowing loops following the declarative loop syntax, loops within XExpressions allow the number of iterations to be random as well as a few syntactic alternatives:

1. Basic, C-like for loops:
   `for (var IteratorType iteratorName = init; condition; update) {...}`
   An example of which would be
   `for (var int i = 0; i <= 10; i++) {...}`.
2. While loops:
   `while (condition) {...}`.

Function calls

Functions are called as one would expect: `nameOfFunction(expression1, expression2)` where each element in `expression1` and `expression2` are XExpressions. These expressions are evaluated prior to being passed into the function (i.e., a form of "eager/greedy evaluation").

The only exception to this are composite laws, where the evaluation of an argument is delayed at initialization and instead repeated each time the density is evaluated during MCMC sampling (i.e., a form of "lazy evaluation" in this special case).

In all cases, the actual function call only involves copying a constant size register making these calls very cheap. For primitives, the value of the primitive is copied and therefore the original primitive can never suffer side effects from the call. For object references, the memory address in the reference is copied and hence the original reference cannot be changed, although the object it points to might have its state changed by the function call.

User defined functions

To create supporting functions, the user can create a separate Xtend or Java file. In Xtend, use the following template for the separate file, say `MyFunctions.xtend`:

```
package my.pack
class MyFunctions {
  def static ReturnType myFunction(ArgumentType1 arg1, ArgumentType2 arg2) {
    // some computation
    return result
  }
}
```

Back to the Blang file being developed, the user can then import the functions into the Blang file using `import static my.pack.MyFunction.*` allowing us to call `myFunction(arg1, arg2)`.

Extensions

Extension methods provide a kind of lightweight trait, i.e., adding methods to existing classes on demand.

Continuing the same example as last section, this is done by adding an extension import statement:

```
import static extension my.pack.MyFunctions.myFunction
```

Provided a variable, say `myVar`, of type `ArgumentType1` (the type of the first input argument to the function `myFunction` defined in the previous section), the user can then invoke the function via `myVar.myfunction(arg2)`.

As a concrete example of how this is used to create more succinct code, consider a typical `generate` snippet, showing here how a Yule Simon distributed variate can be generated as a mixture

```
generate(rand) {
  val w = rand.exponential(rho)
  return rand.negativeBinomial(1.0, 1.0 - exp(-w))
}
```

The underpinning of this code is that since Blang automatically imports `blang.distributions.Generators`, which contains the function

`def static double exponential(Random random, double rate)`

then we can call `rand.exponential(...)` on the variable `rand` of type `java.util.Random`.

Creating objects

An object of type `MyClass` is created by calling `new MyClass(argument1, ...)`. This can be shortened to `new NameOfClass` if there are no arguments. To find which argument(s) are necessary, look for the *constructor* in `MyClass`, which uses the keyword `new` in Xtend and the name `MyClass(...)` in Java.

In some libraries, for example in the package we use for linear algebra, **xlinear**, the call to `new` is wrapped inside a static function. In this case, just call the function to instantiate the object. For example, to create a new sparse matrix with 1 000 rows and 10 000 columns, use `sparse(1_000, 10_000)` (automatically imported from `xlinear.MatrixOperations`).

Using objects

Classes have *instance variables* or *fields*, which are variables associated with objects, as well as *methods*, which are functions associated with the object having access to the object's instance variables. Collectively, fields and methods are called *features.*

Features are accessed using the "dot" notation:
`myObject.myVariable` and `myObject.myMethod(...)`. When a method has no argument, the call can be shortened to `myObject.myMethod`.

The ability to call a feature is subject to Java visibility constraints. In short, only public features can be called from outside the file declaring a class.

Implicit variable `it`

The special variable `it` allows users to provide a default object for feature calls:

```
val it = myObject
doSomething // equivalent to it.doSomething
```

Lambda expressions

A *lambda expression* is a succinct way to write a function without having to give it a name. This construction makes it easy to call functions which take functions as argument (e.g., to apply the function to each item in a list, etc). Since they are so useful, many syntactic shortcuts are available.

The explicit syntax for lambda expressions is:

```
[Type1 argument1, ... | functionBody ]
```

For example, computing the norm of a matrix is implemented as follows in **xlinear**:

```
def static double norm(Matrix m) {
  val double sumOfSqrs = m.nonZeroEntries
                         .map([double value | value * value])
                         .sum
  return Math.sqrt(sumOfSqrs)
}
```

When there is a single input argument, you can skip declaring the argument, and instead the argument will be assigned to the implicit variable `it` (described in the previous section). This allows us to write:

```
  ...
  val double sumOfSqrs = m.nonZeroEntries.map([value * value]).sum
  ...
```

Finally, when the last argument of a function is a function, you can simply put the lambda after the parentheses of the function call. For example:

```
  ...
  val double sumOfSqrs = m.nonZeroEntries.map[value * value].sum
  ...
```

Lambda expressions can also access final variables (i.e., marked by `val`) that are in the scope. Moreover, lambda expressions can automatically be cast to interfaces having a single declared method. For example, a constant `RealVar` can be created as follows: `val RealVar = [3.14]`.

Boxing and unboxing

Boxing refers to wrapping a primitive such as `int` or `double` into an object such as `Integer` or `Double`. Deboxing is the reverse process. The `Integer` or `Double` objects are immutable data structures necessary as many data structures assume all their contents are references to objects rather than primitives. As in Java, the conversion between the two representations is automatic in the vast majority of the cases. Blang adds boxing/deboxing to and from `blang.core.IntVar` and `blang.core.RealVar`, which are mutable versions of `Integer` or `Double`. See Appendix A.3 for a discussion on why these mutable datastructures are necessary in Blang.

Operator overloading

Operator overloading is permitted. One important case to be aware of is `==` which is overloaded to `.equals(...)`. For the low-level equality operator that checks if the two sides are identical (point to the same object or in the case of primitive, have the same value) use `===` (with the exception of `Double.NaN` which, following IEEE convention, is never `===` to anything). When in the Blang IDE, command click on an operator to reveal its definition.

Some useful operators that are automatically imported:

- `0..10`, `0..<11`, `-1>..10`: These three expressions are range operators and return integers 0, 1, 2, ..., 10.
- `object => lambdaExpression`: calls the lambda expression with the input given by object e.g., `new ArrayList => [add("to be added in list")]`

When overloading operators of custom type refer to Xtend's official documentation (Xtend 2019).

Parameterized types

Types can be parameterized as in Java's `List` type. For example, we use `List<String>` to declare that a string will be stored, just as we would in Java and Blang. At the moment, models can use variables with type parameters but models themselves cannot have type parameters.

Throwing exceptions

Throw exceptions to signal abnormal behaviour and to terminate the Blang runtime with an informative message:

```
throw new MyException("Some error message.")
```

Here `MyException` should be of type `java.lang.Throwable`. A reasonable default choice is `java.lang.RuntimeException`. To signal that the current factor has invalid parameters return the value `NEGATIVE_INFINITY`. If not possible due to a particular code structure, one can also return the value `blang.types.StaticUtils.invalidParameter`. This will be caught and interpreted as a factor having zero probability. In contrast to Java, Blang exceptions are never required to be declared or caught. If the exceptions needs to be caught, the syntax is as follows:

```
try {
  // code that might throw an exception
} catch (ExceptionType exceptionName) {
  // process exception
} // optionally:
finally {
  // code executed whether the exception is thrown or not
}
```

7. Tools and software development kit

Blang comes with "batteries included": more than just a language, it is a suite of tools and libraries supporting common tasks in Bayesian data analysis. In this section, we present an overview of these libraries. Briefly, we start with a description of the Blang integrated development environment (IDE). Followed by a discussion on how input of data is handled in Blang. This includes an introduction to implementing plate and plated variables for plate notation used in traditional graphical models. Next, we discuss how samplers, distributions, and other components fit into the core inference algorithms' architecture. Finally, we conclude with brief discussions on post-processing options, monitoring logs, testing frameworks, and additional packages and dependencies.

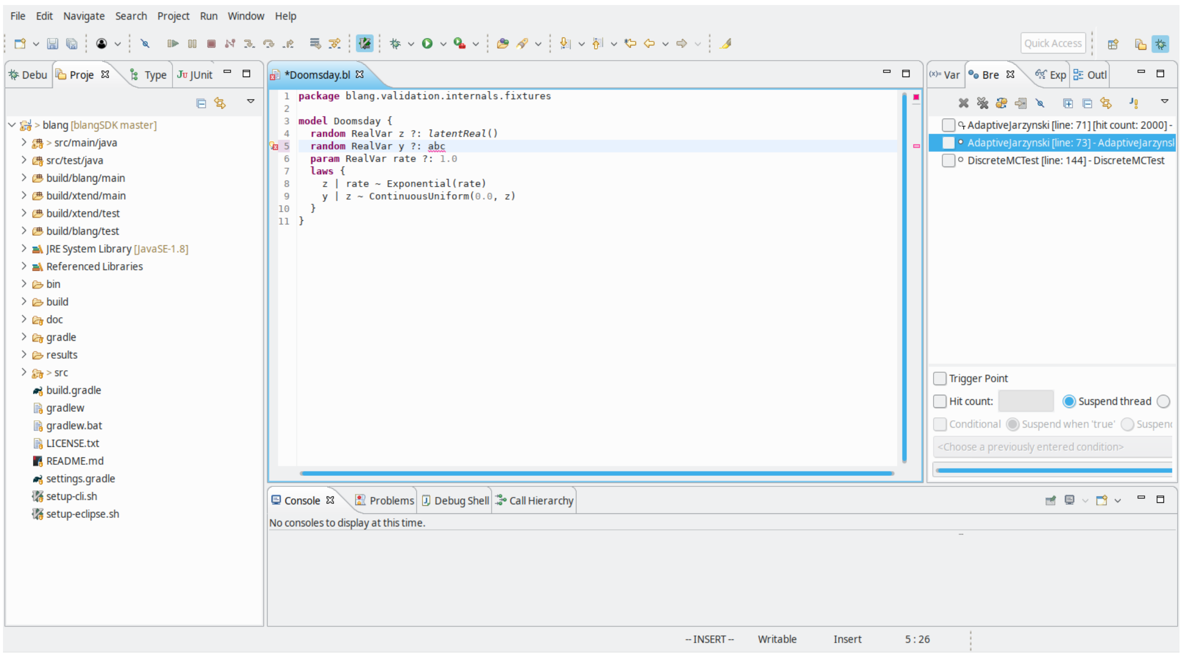
7.1. Integrated development environment (IDE)

Integrated development environments are software applications built for software construction. They are typically equipped with features such as syntax highlighting, code completion, refactoring, debugging and other tools that assist programmers in software development.

Desktop IDE

The Xtext framework provides a convenient workflow, allowing for the integration of the Eclipse IDE with the language being developed. The installation process for the Blang IDE that is most portable across platforms is the following:

1. Install *DSL tools for Eclipse*, which can be downloaded from the Eclipse website.
2. From Eclipse: select *Install New Software* from the *Help* menu.
3. Click *Add* and enter `https://www.stat.ubc.ca/~bouchard/maven/blang-eclipse-plugin-latest/` in the location field.
4. Click *Select All*, *Next*, then follow instructions as prompted.



```
package blang.validation.internals.fixtures

model Doomsday {
  random RealVar z ?: latentReal()
  random RealVar y ?: abc
  param RealVar rate ?: 1.0
  laws {
    z | rate ~ Exponential(rate)
    y | z ~ ContinuousUniform(0.0, z)
  }
}
```

```
AdaptiveJarzynski [line: 71] [hit count: 2000] -
AdaptiveJarzynski [line: 73] - AdaptiveJarzynski
DiscreteMCTest [line: 144] - DiscreteMCTest
```

```
No consoles to display at this time.
```

Figure 10: A preview of Blang IDE; Warning of syntactical error is underlined in red.

The key IDE features useful for development include:

- Ability to navigate a Blang code base and the Blang SDK by holding command while clicking on any symbol to jump to its definition, or hovering on it to see documentation. This and other related features are possible thanks to the static type system used by Blang.
- Incremental compilation in parallel in the background, which implies little time is spent waiting for compilation on modern multicore architectures. It also means that error messages appear interactively as the user type. See example in Figure 10.
- Quickly viewing the generated Java files, by right clicking anywhere in a Blang editor and selecting "Open Generated File".
- From any generated file, inference on the model can be launched by right-clicking "Run As... Java Application". After doing this the first time, a shortcut is accessible via the menu "Run > Run Configurations..". The Run Configurations allow setting the command-line arguments being passed to Blang.
- A full-feature debugger is built-in. Double clicking on the left margin of a Blang or Xtend file sets a break point. Use the menu "Debug > Debug Configurations" to start the debugger.
- Being built on Eclipse, the IDE also inherits Eclipse's comprehensive set of features, such as utilities for unit testing, code coverage analysis, git integration, visualization of call and type hierarchies among others.

More information on the Blang IDE is available from the Blang documentation page, `https://www.stat.ubc.ca/~bouchard/blang/Blang_IDE.html`.

Web IDE

To facilitate deployment on large number of cores on the cloud, for example in a teaching or reproducible research context, Blang is also available on the Web scientific platform Silico (`https://silico.io/`).

To setup a Blang project in Silico, create `Model` from the user profile page, and create a file with `.bl` extension. Command-line argument can be passed in by pasting them in a file called `configuration.txt`.

7.2. Data types provided in the SDK

The interfaces `blang.core.RealVar` and `blang.core.IntVar` are automatically imported. They can be either latent (unobserved, sampled), or fixed (conditioned upon). See Table 15 for commonly used functions to provide default initializations to these basic random variables.

Blang's linear algebra is based on **xlinear** which is in turn based on a portfolio of established libraries.

The basic classes available are `Matrix`, `DenseMatrix`, and `SparseMatrix`. Blang/XBase allows operator overloading, so it is possible to write expressions of the type `matrix1 * matrix2`, `2.0 * matrix`, and so on. Vectors do not have a distinct type, they are just $1 \times n$ or $n \times 1$ matrices. Standard operations are supported using unsurprising syntax, e.g., `identity(100_000)`, `ones(3,3)`, `matrix.norm`, `matrix.sum`, `matrix.readOnlyView`, `matrix.slice(1, 3, 0, 2)`, `matrix.cholesky`, etc.[8]

Blang augments **xlinear** with two specialized types of matrices: `Simplex`, vector of positive numbers summing to one, and `TransitionMatrix`. Refer to Table 16 for key functions related to these specialized types of matrices.

7.3. Distributions

A range of distributions are included in the SDK. See Appendix D for the current list. These distributions are themselves written in Blang. The SDK also contains tests covering all the included distributions. Our development workflow performs all the unit tests each time a commit is made in the Blang Github repository.

The implementation of the random number generators used in forward simulation of the SDK distributions are all grouped in the file `blang.distributions.Generators`.

[8]See `https://github.com/alexandrebouchard/xlinear` for more information on **xlinear**.

7.4. Input

Inputs are parsed and managed via the **inits** package's injection framework. Model variables can be provided a default initialization in the model's `.bl` file, or they can be initialized with arguments through the CLI. Should both methods exist, the latter takes precedence; an example exposing only the pertinent snippets of code is shown below.

```
model Name {
  param IntVar h ?: 3
  param IntVar a
  random Type1 p
  ...
}
```

The field `h` is initialized to `3` by default, but can be overridden by the command-line argument `--model.h 7`; field `a` must be assigned a value via the CLI through `--model.a 9`. For observations or custom data types such as `Type1`, a annotations can be easily added to control parsing. The following is an example of a constructor that can parse command-line arguments such as `--model.p.file abc.csv` and `--model.p.option 2`.

```
import blang.inits.ConstructorArg;
import blang.inits.DesignatedConstructor;
import blang.inits.GlobalArg;
import blang.runtime.Observations;

class Type1
{
  ...
  @DesignatedConstructor
  def static Type1 loadObservedData(
      @ConstructorArg(value = "file") File file,
      @ConstructorArg(value = "option") Integer x,
      @GlobalArg Observations observations)
  {
    val Type1 result = ... // Parse the file
    observations.markAsObserved(result)
    return result
  }
  ...
}
```

Additional information is provided at the relevant Blang documentation page https://www.stat.ubc.ca/~bouchard/blang/Input_and_output.html and in the **inits** repository https://github.com/UBC-Stat-ML/inits.

Plate notation

Simple collections of random variables can be built using Java's built-in `List` objects and related Collections Framework data structures, however, they are cumbersome in more complicated Bayesian models such as hierarchical models. To address this problem, Blang provides a specialized data structure based on *plates*.

The plate notation is a visual method for representing random variables repeated in a graphical model. Blang allows for easy transcription of graphical models involving collections of random variables into the language through its built-in types `Plate` and `Plated`.

Consider a hierarchical model representing failure rates of rocket launches, given data in the format shown below, and suppose we are primarily interested in comparing different rocket's failure rates for say US rockets, but wish to utilize launch data from other countries to borrow statistical strength.

countries	**rockets**	**nLaunches**	**nFails**
CHN	Chang Zheng 1	2	0
ESA	Ariane 44P	15	0
RUS	Molniya 8K78M	272	13
USA	Delta 2914	30	2
⋮	⋮	⋮	⋮

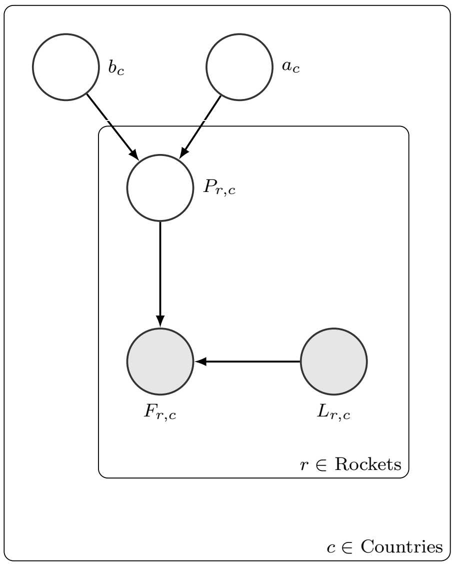
Figure 11 shows a Blang implementation for such a hierarchical model, alongside the corresponding graphical model.

We can run the rocket model via a prepackaged repository of examples

```
> git clone https://github.com/UBC-Stat-ML/JSSBlangCode.git
> cd JSSBlangCode/example/
> blang --model jss.hier.Rocket \
    --model.data data/rockets.csv \
    --model.countries.name Country \
    --model.rockets.name Rockets \
    --model.nLaunches.name nLaunches \
    --model.nFails.name nFails
```

In general, an object of type `Plate<K>` represents a plate, whose indices are of type `K` (typically `Integer` or `String`), while an object of type `Plated<T>` represents a random variable or parameter of type `T` enclosed in one or more `Plates`. Figure 11 demonstrates how a model comprised of nested plates can be implemented.

Each variable of type `Plate` and `Plated` will be put in correspondence with a column in a csv file in Tidy format. Command-line arguments can be used to set the csv file for each variable individually (use the argument `--model Rocket --help` for details). Alternatively, by declaring a dummy variable of type `GlobalDataSource`, here called `data`, we can set a default csv file that will be used by all `Plate` and `Plated` variables. In our example, specifying the default csv is then done via `--model.data pathToData/data.csv`.



```
model Rocket {
  param GlobalDataSource data
  param Plate<String> countries
  param Plate<String> rockets
  param Plated<IntVar> nLaunches
  random Plated<RealVar> a, b, p
  random Plated<IntVar> nFails
  laws {
  for(Index<String> c : countries.indices){
    a.get(c) ~ Gamma(1,1)
    b.get(c) ~ Gamma(1,1)
    for(Index<String> r : rockets.indices(c)){
      p.get(r, c) |
      RealVar a = a.get(c),
      RealVar b = b.get(c)
      ~ Beta(a, b)
      nFails.get(c, r) |
      RealVar p = p.get(r, c),
      IntVar nLaunch = nLaunches.get(r, c)
      ~ Binomial(nLaunch, p)
    }
   }
  }
}
```

Figure 11: Hierarchical model for rocket data with a graphical model representation (left) and the corresponding Blang model (right). Notice that `.get(r, c)` and `.get(c, r)` can be used interchangeably since the indices keep track of which plate they belong to.

By default, for each `Plate` and `Plated` variable, the data input algorithm will look for a column with the same name as the variable, but this default can also be changed via CLI arguments documented in `--help`.

PlatedMatrix

One special case of a `Plated` variable is the type `blang.types.PlatedMatrix`, which is a built-in type that facilitates easy representation of higher dimensional random variables such as random vectors or matrices.

`PlatedMatrix` can be used to represent both random vectors and matrices that are enclosed within a `Plate`. `PlatedMatrix` generally works in the same way as `Plated`, but provide specialized mechanisms to access vectors, matrices, simplices, etc. For example, to access a dense vector, use the method `myPlatedMatrix.getDenseVector(myRowPlate, myParentIndex1, myParentIndex2, ...)`. Here `myRowPlate` refers to the plate from which row indices will be constructed. The indices can be either of the form $0, 1, 2, \dots$, or, if they are of other types, say strings or non-consecutive integers, in which case a fixed correspondence with $0, 1, 2, \dots$ is maintained internally.

The other arguments, `myParentIndex1, myParentIndex2, ...` consist in the indices for the plates in which this vector belongs to. For example, a set of vectors can be obtained as follows:

```
model PlatedMatrixExample {
  param Plate<String> dims
  param Plate<String> replicates
  random PlatedMatrix vectors
  laws {
    for (Index<String> n : replicates.indices) {
      vectors.getDenseVector(dims, n) |
        int size = dims.indices.size
      ~ MultivariateNormal(dense(size), identity(size).cholesky)
    }
  }
}
```

7.5. Testing framework

There is considerable emphasis in the MCMC literatures on efficiency, but much less on correctness, in the sense of the implementation being ergodic with respect to the distribution of interest. In this section, we discuss the techniques available to test Blang (both its SDK and facilities to help users test their models).

Exhaustive random objects

We provide in `bayonet.distributions.ExhaustiveRandom` a non-standard replacement implementation of `bayonet.distributions.Random` which enumerates all the probability traces used by an arbitrary discrete random process.

Testing unbiasedness

We use `ExhaustiveRandom` to test the unbiasness of the normalization constant estimate provided by our SMC implementation. The code forming the basis of this test is shown below:

```
package blang.validation

import java.util.function.Supplier
import bayonet.distributions.ExhaustiveDebugRandom

class UnbiasednessTest {
  def static double expectedZEstimate(Supplier<Double> logZEstimator,
                                      ExhaustiveDebugRandom exhaustiveRand) {
    var expectation = 0.0
    var nProgramTraces = 0
    while (exhaustiveRand.hasNext) {
      val logZ = logZEstimator.get
      expectation += Math.exp(logZ) * exhaustiveRand.lastProbability
      nProgramTraces++
    }
    println("nProgramTraces = " + nProgramTraces)
    return expectation
  }
}
```

This can be called with a small finite model, e.g., a short HMM, but making it large enough to achieve code coverage. The output of a test based on this has the form:

```
nProgramTraces = 23868
true normalization constant Z: 0.345
expected Z estimate over all traces: 0.34500000000000164
```

Showing that indeed our implementation of SMC is unbiased.

Linear algebra based tests

In `blang.validation.DiscreteMCTest`, we provide algorithms that use `ExhaustiveRandom` to check via linear algebra if kernels on small discrete models are invariant and irreducible.

More precisely, `DiscreteMCTest` takes a model and kernel, and form the corresponding sparse transition matrix automatically. From this matrix it is then trivial to check numerically irreducibility and invariance. See `blang.TestDiscreteModels` for example of usage.

Exact invariance test

For continuous models, we provide a modified form of the method in Geweke (2004) which we call an exact invariance test (EIT). The general idea is that `blang` models that specify a `generate` block encode the same probability distribution in two different ways (forward and posterior). We then consider two sets of samples: one set based only on iid calls to `generate`, and another set that utilizes several MCMC traces which are, crucially, each initialized via `generate`. This is one of the rare scenarios where point null hypothesis testing makes sense: if the code is correct, the two sets are samples from the same distribution.

Let us assume we have a model called `MyModel` declaring a random variable called `realization` and that all variables have a default initialization. Then to test this model, use the following code:

```
import org.junit.Test
import blang.validation.ExactInvarianceTest
import blang.validation.Instance

class TestMyModel {
  val MyModel model = new MyModel.Builder().build

  @Test
  def void exactInvarianceTest() {
    val test = new ExactInvarianceTest
    test.nPosteriorSamplesPerIndep = 500
    test.add(
      new Instance(
        model,
        [realization.doubleValue]
      )
    )
    test.check
  }
}
```

A complete example of an exact invariance test for our permutation model (Section 4.5) is provided in the reproduction materials.

7.6. Package distribution and injection

Distributing and reusing predefined packages is standard practice in software development. Any user can create a model and publish it in a versioned fashion via GitHub.[9]

To use a package developed by another user, Blang projects compiled via the CLI automatically handle dependencies hosted on GitHub by parsing a file called `dependencies.txt` placed in the project root directory. For correct parsing, GitHub dependencies' format must be of the forms

`com.github.Username:Repository:Branch-CommitHash`
`com.github.Username:Repository:ReleaseTag`

where `CommitHash` may be replaced by `SNAPSHOT` to automatically select latest commits. For compilation through Eclipse IDE, users should manually input dependencies in the `build.gradle` file.

[9]Under the hood, the mechanism for dependency management is **Maven**. However, GitHub repository can seamlessly imported via **JitPack**. See `https://jitpack.io/` for details.

To distribute packages, users can create a project following the directory organization of the provided `blangTemplate` repository, and publish it in a separate, public GitHub repository.[10]

8. Design patterns

This section discusses design patterns specific to programming in Blang. The goal of these design patterns is to enable users to design models going beyond Bayes nets, improve computational efficiency, and improve code readability.

8.1. Undirected graphical models

The mechanisms in Blang's default inference engine require the models to be in generative normal form. In some cases, in particular for users interested in undirected graphical models or Markov random fields (MRF), this may appear a stringent condition, since forward simulation in these models is computationally intractable.

We illustrate here a construction based on a type of "pseudo-prior." Let $f_\theta(x) \propto \prod_{i\in I} \psi_\theta(x)$ denote a MRF. We rewrite the model as $f_\theta(x) \propto f_0(x) \prod_{i\in I} \tilde{\psi}_\theta(x)$, where $f_0(x)$ is a "tractable" pseudo-prior. By tractable, we mean that we can sample and compute the normalization constant of the pseudo-prior. Annealing is then automatically performed on the factors $\prod_{i\in I} \tilde{\psi}_\theta(x)$ only, not on the pseudo-prior, ensuring finite marginalization for all interpolating distributions.

For example, consider the Ising model (Ising 1925) which is a type of Markov random field (MRF). In this example, we can simply use a product of independent Bernoulli random variables as a pseudo-prior:

```
model Ising {
  param Double moment ?: 0.0
  param Double beta ?: log(1 + sqrt(2.0)) / 2.0 // critical point
  param Integer N ?: 5
  random List<IntVar> vertices ?: latentIntList(N*N)

  laws {
    // Pairwise potentials
    for (UnorderedPair<Integer, Integer> pair : squareIsingEdges(N)) {
      | IntVar first  = vertices.get(pair.getFirst),
        IntVar second = vertices.get(pair.getSecond),
        beta
      ~ LogPotential({
          if ((first < 0 || first > 1 || second < 0 || second > 1))
            return NEGATIVE_INFINITY
          else
            return beta*(2*first-1)*(2*second-1))
        })
```

[10]Template repository and instructions can be found at https://github.com/UBC-Stat-ML/blangTemplate

```
      }
      // Node potentials
      for (IntVar vertex : vertices) {
        vertex | moment ~ Bernoulli(logistic(-2.0*moment))
      }
    }
  }
```

Notice instead of using `logf` here for the likelihood, which would have violated the technical conditions for generative normal forms, we used the `LogPotential` utility in the SDK

```
model LogPotential {
  param RealVar logPotential
  laws {
    logf(logPotential) {
      return logPotential
    }
  }
```

Since `LogPotential` does not define random variables, when it is invoked there are no variables to the left of the conditioning symbol in `| IntVar first = ...`. This follows naturally from the formal definition of composite laws.

8.2. Delayed graphical model construction

The runtime engine is able to decrease computational expense when it can detect sparsity patterns in models. This is handled automatically for simple objects but requires user input for complex objects. For an example with a complex object `chain` consider the following Markov Chain:

```
model MarkovChain {

  param Simplex initialDistribution
  param TransitionMatrix transitionProbabilities
  random List<IntVar> chain

  laws {
    chain.get(0) | initialDistribution ~ Categorical(initialDistribution)

    for (int step : 1 ..< chain.size) {
      chain.get(step) | IntVar previous = chain.get(step - 1),
                        transitionProbabilities
        ~ Categorical({
            if (previous >= 0 && previous < transitionProbabilities.nRows)
              transitionProbabilities.row(previous)
            else
              invalidParameter
          })
    }
  }
}
```

We condition on the previous step instead of the whole chain, using `chain.get(step) | IntVar previous = chain.get(step - 1)` as opposed to `chain.get(step) | chain`. This prevents the runtime architecture from computing factors involving the full `chain` object, potentially improving computational efficiency by a scaling proportional to the chain size. In other words, here the exact specification of the graphical model is delayed until the data is available.

In general, it is optimal to condition objects on minimal parts of the data structure in comparison to the full structure. For example, suppose we have `SomeObject x` with conditional distribution on `ConditionalObject y`, where `y` has two `IntVar` fields `a` and `b`. If the distribution on `x` only requires the first field of `y`, `a`, then we should condition only on `a`. Hence, we use `x | IntVar v = y.a ~ Distribution(v)` as opposed to `x | y ~ Distribution(y.a)`. For a detailed understanding of this efficiency gain, we refer readers to Section 9.4.

8.3. Model reparameterization

It is often the case that distribution families can be written using different parameterizations, or that a family can be expressed as a special case of another family. Following "Don't Repeat Yourself" (DRY) coding principles, the following pattern shows what is the best practice to express such reparameterizations.

To illustrate the pattern, consider how the Exponential distribution is coded in the Blang SDK as a special case of the Gamma distribution:

```
model Exponential {
  random RealVar realization
  param  RealVar rate
  laws {
    realization | rate ~ Gamma(1.0, rate)
  }
}
```

8.4. Distributions as parameters

In many situations, it is useful to have one or several parameters of a model to be themselves distributions. Consider for example a mixture model: it takes as input a list of distributions as well as mixture proportions, and creates a new distribution from it. Here is an example of how this is implemented for mixtures of integer-valued distributions in Blang:

```
model IntMixture {
  param Simplex proportions
  param List<IntDistribution> components
  random IntVar realization

  laws {
    logf(proportions, components, realization) {
      var sum = 0.0
      if (components.size !== proportions.nEntries) {
        throw new RuntimeException
      }
      for (i : 0 ..< components.size) {
        val prop = proportions.get(i)
        if (prop < 0.0 || prop > 1.0) return NEGATIVE_INFINITY
        sum += prop * exp(components.get(i).logDensity(realization))
      }
      return log(sum)
    }
  }

  generate (rand) {
    val category = rand.categorical(proportions.vectorToArray)
    return components.get(category).sample(rand)
  }
}
```

And this is invoked from another model as follows, and in this case, to create a mixture of two Poisson distributions:

```
x | lambda1, lambda2, pi
  ~ IntMixture(
    pi,
    #[Poisson::distribution(lambda1), Poisson::distribution(lambda2)]
  )
```

Here `Poisson::distribution(...)` is a convenient shortcut generated automatically: any model with only one random variable is automatically endowed with a `distribution(...)` function taking as input the model's parameters. If that single random variable is of type `RealVar`, its return value of `distribution(...)` is of type `blang.core.RealDistribution`;

similarly `IntVar` is to `blang.core.IntDistribution`. If it is neither of the two cases, it is of the type `blang.core.Distribution`.

9. Inference

Blang efficiently samples from posterior distributions by detecting sparsity patterns in the model, matching variable types with their associated roles in inference, then sample using state-of-the-art Monte Carlo methods.

In the following sections, we detail intermediate steps in the process described above. We first assume that a continuum of probability distributions is available. On one end of the spectrum, we have the posterior distribution, and the prior on the other. The prior is a distribution from which we can sample from assuming the model is in generative normal form. Then we describe the technical details used to automatically construct this continuum of interpolating probability distributions, along with invariant Markov chain kernels for each distribution in the interpolation.

9.1. Inference algorithms

Blang currently focuses on two complementary inference algorithms: sequential change of measure (SCM), and non-reversible parallel tempering (PT). The former is an SMC algorithm and the latter a parallel MCMC algorithm.

A core concept present in both algorithms is the use of an adaptive sequence of tempered distributions extracted from a continuum of interpolating distributions. Through these tempering schemes, we are able to explore complex, multimodal distributions without the need for automatic differentiation, as such, these techniques are not limited to Euclidean spaces. For example, the default sampler for real and integer data types are their respective slice samplers,[11] which when used in a naive MCMC algorithm could perform poorly in highly correlated models. However in the context of SCM or PT, it is frequently the case that simple MCMC algorithms perform better than using specialized moves in a single chain (Ballnus *et al.* 2017).

Furthermore, due to the inherent characteristics of these algorithms, they are trivially parallelized for efficient computing, and provide computation of model evidence at negligible cost. For these reasons, SCM and PT are the default engine choice implemented for automatic inference on generalized state spaces. Algorithms can be used individually, but by default the SCM is used to initialize PT. This combination is motivated by the fact that SCM is better suited to quickly find a rough approximation. In particular it is able to find configurations of positive probability even in the presence of deterministic constraints. Then, to obtain high quality samples, we found SCM may require a number of particles larger than what can be fitted in memory. PT on the other hand can provide approximations of arbitrary high quality without asymptotically infinite memory consumption.

Parallel tempering

[11] More precisely, a doubling and shrinking procedure is used as an adaptive scheme, whose details and validity are described and proved by Neal (2003).

Parallel tempering (Geyer 1991) is an MCMC method that operates on product spaces. Informally, PT runs multiple Markov chains in parallel, concurrently targeting a sequence of tempered distributions. The full sample-update scheme consists of two components: an exploration step taking place within individual chains, and a swapping step taking place between chains. In the exploration step, each chain's state is updated via samplers discussed in Section 9.5. In the swapping step, moves are proposed as configuration swaps between chains. This swapping procedure provides opportunities for states to traverse across modes, by utilizing a tempered chain's variability. Accounting for information across all chains, we can estimate the model evidence, while pertaining a marginal distribution equivalent to the posterior distribution of interest.

In standard PT algorithms, increasing the number of parallel chains does not guarantee greater performance. As the number of chains is increased, the acceptance rates of swaps increases as there is less discrepancy between densities of neighbouring chains. However, this gain comes at the cost of an increase in expected number of swaps for information to propagate from end to end. In addressing this weakness, Syed *et al.* (2019) analyzed a class of non-reversible PT algorithms, which was shown to dominate its stochastic, reversible PT counterpart. Under this non-reversible regime, using a number of parallel chains at least as large as the number of parallel cores available was shown to strictly improve asymptotic performance.

In addition to its robustness in exploring complex posteriors, PT provides an inexpensive method to approximate the model evidence through thermodynamic integration (Ogata 1989). For models having full support, thermodynamic integration is automatically computed and included in the standard output. For models with hard constraints, the technical conditions underlying thermodynamic integration are not met so the user is prompted to use SCM instead to approximate the normalization constant, described in the next section.

Sequential change of measure

Informally, sequential change of measure initializes a population of particles from a prior distribution, and is iteratively perturbed and weighted across a sequence of tempered, bridging distributions to the posterior.

Formally, SCM is a special type of SMC sampler (Del Moral *et al.* 2006) following an adaptive tempering schedule described by Zhou *et al.* (2016). As described in these references, the method transforms the target distributions of fixed-dimensions into a form amenable to the standard SMC framework by introducing artificial, auxiliary backward kernels such that the target is defined on a product space.

In short, we adaptively construct a sequence of tempered, intermediate distributions bridging the prior and posterior distributions. Then, using invariant samplers and importance sampling, we perturb a population of weighted particles across this bridge of distributions. To prevent degeneracy of particles, resampling steps are invoked when the effective sample size (ESS) of weights fall beneath a threshold. The adaptive scheme determining annealing parameters is controlled by the ESS of weights. This resampling criterion can be adjusted via user input, and thus the annealed importance sampling algorithm (Neal 2001) can be recovered as a special case of the SCM by omitting resampling altogether.

Similar to the PT algorithm, by construction of the algorithm, an estimate of model evidence comes as a by-product and is provided in the outputs. In this case the theory behind its justification is far more generally applicable.

Constructing a sequence of measures

Both SCM and PT inference algorithms require a continuum of measures. To retain theoretical guarantees, we must ensure each measure in this sequence has a finite normalization constant. To achieve this, we factorize our joint density into what we call likelihood $l_i(x)$ and prior $p_j(x)$ factors. Assuming a Blang model in generative normal form, the construction of such a continuum of probability measures begins with an exhaustive unrolling of composite laws to identify all atomic laws, or log factors. Each factor belongs to a model and as such each of its dependencies can be classified as either corresponding to `random` or `param`. If its dependency is `random`, we direct the corresponding edge in the factor graph as out-going. Otherwise, if it is a `param`, we direct the edge as in-coming. Likelihood factors are then defined as factors whose outgoing edges, if any, all connect to an observed variable; factors are classified as priors otherwise.

Suppose we have factorized our posterior as follows

$$\pi(x) \propto \prod_{i=1}^{I} l_i(x) \prod_{j=1}^{J} p_j(x)$$

where $l_i(x)$, $p_j(x)$ denote likelihood and prior factors respectively. As opposed to raising the product of likelihood and prior factors to some $t \in [0, 1]$, which may not yield a probability distribution, it is preferable to exponentiate the likelihood factors.

Additionally, it is common to have configurations of zero probability when performing inference over discrete combinatorial objects. In some scenarios, for example in pedigree analysis, these zero-valued likelihood evaluations can create difficulties in building irreducible samplers, thus invalidating convergence guarantees. We alleviate this restriction using the annealing scheme shown below,

$$\pi_t(x) = \frac{\left(\prod_{i\in I}[(l_i(x))^t + \mathbb{I}(l_i(x) = 0)\epsilon_t]\right)p(x)}{Z_t} \tag{5}$$

where $\epsilon_t = \exp(-10^{100}t)\mathbb{I}(t < 1)$ and $p(x) = \prod_{j=1}^{J} p_j(x)$. By design, the interpolating chains have full support, while maintaining the guarantee of having a finite normalization constant for all annealing parameters:

$$\begin{aligned}
\int \pi_t(x)dx &= \int p(x)\prod_{i\in I}[(l_i(x))^t + \mathbb{I}(l_i(x) = 0)\epsilon_t]dx \\
&\leq \sum_{K:K\subset I} \epsilon_t^{|I|-|K|} \int p(x)(\prod_{i\in K} l_i(x))^t dx \\
&= \sum_{K:K\subset I} \epsilon_t^{|I|-|K|} \int p(x)(\prod_{i\in K} l_i(x))^t [I(\prod_{i\in K} l_i(x) \geq 1) + I(\prod_{i\in K} l_i(x) < 1)]dx \\
&\leq \sum_{K:K\subset I} \epsilon_t^{|I|-|K|} [\int p(x) \prod_{i\in K} l_i(x)dx + \int p(x)dx] \\
&< \infty
\end{aligned} \tag{6}$$

This proposed annealing scheme allows our sampler to traverse across multimodal distributions, preserve the correct marginal posterior distribution at room temperature, and guarantee convergence of normalizing constant estimates.

9.2. Parallelization

Due to the nature of PT and SCM algorithms, parallelization can be used to obtain significant performance improvements. In both PT and SCM, transition MCMC kernels are applied in parallel across particles/chains. In addition to parallelization of transition kernels, PT also performs its swap operations in parallel.

Blang uses lightweight threads to parallelize these operations (Friesen 2015). Specifically, it uses the algorithm described in Leiserson *et al.* (2012) as implemented in Steele and Lea (2013). This implementation allows each chain to pertain to its own random stream, consequently avoiding any blocking between threads. Furthermore, this implementation implies any numerical output will not be altered by the number of threads utilized given fixed random seeds.

9.3. Evidence

Computing the evidence for model selection is an integral part of the Bayesian paradigm. Both built-in inference algorithms PT and SCM compute the evidence at negligible costs. Parallel tempering algorithms allow for an estimate of the evidence through approximate thermodynamic integration (Ogata 1989). In sequential change of measure, the evidence comes as a by-product of the algorithm itself as detailed in Del Moral *et al.* (2006). In comparison with other probabilistic programming languages today, the need for additional packages and external dependencies is therefore not required. For example, to estimate a model's evidence in Stan, one would require additional post-processing with bridge sampling (Meng and Wong 1996) using packages such as **bridgesampling** (Gronau and Singmann 2018; Gronau *et al.* 2017).

9.4. Construction of invariant samplers

We first describe how a Blang model m is transformed into an efficient representation aware of m's sparsity patterns. The transformed representation is an instance of `blang.runtime.SampledModel`, a mutable object keeping track of the state space and offering methods to: 1) change the annealing parameter of the model, 2) apply a transition kernel in place targeting the current annealing parameter, 3) perform forward simulation in place, 4) obtain the joint log density of the current configuration, and 5) duplicate the state via a deep cloning library.

Preprocessing

The process of translating m into a `SampledModel` begins with the instantiation of model variables. After this is done, a list l of factors is recursively constructed. That is, we recursively search through m for sub-models, and terminate when we have identified and added all atomic laws to l.

The next phase of initialization consists of building an *accessibility graph* between all objects in a model, more concisely defined as follows: the set of vertices is the set of objects defined by a model, starting at the root model, and of the constituents of these objects recursively. Constituents are fields in the case of objects and integer indices in the case of arrays.[12]

[12]Exploration of a field can be skipped in the construction of the accessibility graph using the annotation

Constituents can also be customized, for example, in order to index entries of matrices. The directed edges of the accessibility graph connect objects to their constituents, and constituents to the object they resolve to, if any. We say that object o_2 is accessible from o_1 if there is a directed path from o_1 to o_2 in the accessibility graph.

Once the accessibility graph has been constructed, latent variables of m are extracted from the vertex set of the accessibility graph. These variables are the intersection of objects of a type annotated with `@Samplers` and objects that are mutable, or have accessible mutable children. In other words, latent variables are objects that have a designated sampler, and are or have access to non-final fields. Immutability is therefore the main mechanism used to define observed (fixed) values. Additionally, we can mark indices in matrices and arrays as observed. This is accomplished by the `blang.runtime.Observations` object. For example, `observationsObject.markAsObserved(mtx.getRealVar(i, j))` marks entry (i, j) of matrix `mtx` as observed. A separate mechanism is needed for matrices to take into account situations where only a proper subset of entries are observed.

Exploiting sparsity

Samplers can be made more efficient by avoiding unnecessary computation of model components; we exploit a model's sparsity pattern by building factor graphs via linear time graph algorithms on our accessibility graph.

Given a latent variable v and factor f, we can determine if the application of a sampling operator on v can change the numerical value of the factor f. This is accomplished by assessing v and f's co-accessbibility. Two objects o_1 and o_2 are said to be co-accessible if there is a mutable object o_3 such that o_3 is accessible form both o_1 and o_2.

Through this awareness of sparsity patterns, we can now perform sampling operations on variables without computing every factor involved in the model. The cost of the entire preprocessing procedure has negligible cost in comparison with the performance to be gained from its implications.

For a concrete example, we refer readers to the Markov chain example (Section 8.2).

9.5. Matching transition kernels (samplers)

Once sparsity patterns have been identified, samplers are matched to latent variables through the `@Samplers` annotation. We have seen examples of this annotation in the permutation example of Section 4.5. Here we provide more details on this process.

To implement a sampler, we turn our attention to the `interface blang.mcmc.Sampler`, which specifies two functions, `void execute(Random rand)`, and an optional `setup` function. When a sampler is instantiated, it first invokes the `setup` method to perform any required pre-computation. Then, during Monte Carlo sampling, the variable being sampled is updated in place through the invocation of `execute`.

A simple example of an implementation of this `Sampler` type is shown below.

```
...
class SimplexSampler implements Sampler {
```

`@SkipDependency`.

```
    @SampledVariable DenseSimplex simplex
    @ConnectedFactor List<LogScaleFactor> numericFactors
    @ConnectedFactor Constrained constrained

    override void execute(Random rand) { ... }
    override boolean setup(SamplerBuilderContext context) { ... }
  }
```

The field `simplex` annotated with `@SampledVariable` is automatically initialized with the sampled variable. Advantaging from the accessbility graph and detection of sparsity patterns, the field `@ConnectedFactor List<LogScaleFactor>` is automatically populated with log factors dependent of `simplex`. `Constrained` is used here to indicate that the sampler being constructed is aware of the constraints posed by simplex variables. If, and only if all the `@ConnectedFactor` can be populated will the sampler be automatically matched to the variable in the factor graph.

Computational Details

All the programs in this paper were run using **blangSDK** 1.33.2 on a Mac OS X version 10.14.5. The device used is a Macbook Pro (15-inch, 2018) with a 2.2 GHz 6-core Intel Core i7 processor and a 2.2 GHz Radeon Pro 555X graphics card and 32 GB of 2400 MHz DDR4 memory.

References

Ackerman NL, Freer CE, Roy DM (2017). "On Computability and Disintegration." *Mathematical Structures in Computer Science*, **27**(8), 1287–1314. doi:10.1017/S0960129516000098.

Ballnus B, Hug S, Hatz K, Görlitz L, Hasenauer J, Theis FJ (2017). "Comprehensive Benchmarking of Markov Chain Monte Carlo Methods for Dynamical Systems." *BMC Systems Biology*, **11**. ISSN 1752-0509. doi:10.1186/s12918-017-0433-1.

Bingham E, Chen JP, Jankowiak M, Obermeyer F, Pradhan N, Karaletsos T, Singh R, Szerlip P, Horsfall P, Goodman ND (2018). "Pyro: Deep Universal Probabilistic Programming." *Journal of Machine Learning Research*.

Burkner PC, Vuorre M (2019). "Ordinal Regression Models in Psychology: A Tutorial." *Advances in Methods and Practices in Psychological Science*, **2**(1), 77–101. doi:10.1177/2515245918823199.

Carpenter B, Gelman A, Hoffman MD, Lee D, Goodrich B, Betancourt M, Brubaker M, Guo J, Li P, Riddell A (2017). "Stan : A Probabilistic Programming Language." *Journal of Statistical Software*, **76**(1). ISSN 1548-7660. doi:10.18637/jss.v076.i01.

Carter Brandon, McCrea W H (1983). "The Anthropic Principle and its Implications for Biological Evolution." *Philosophical Transactions of the Royal Society of London. Series A, Mathematical and Physical Sciences*, **310**(1512), 347–363. doi:10.1098/rsta.1983.0096.

Del Moral P, Doucet A, Jasra A (2006). "Sequential Monte Carlo Samplers." *Journal of the Royal Statistical Society B*, **68**(3), 411–436. ISSN 13697412. doi:10.1111/j.1467-9868.2006.00553.x. 0212648.

Duane S, Kennedy A, Pendleton BJ, Roweth D (1987). "Hybrid Monte Carlo." *Physics Letters B*, **195**(2), 216–222. ISSN 03702693. doi:10.1016/0370-2693(87)91197-X.

Efftinge S, Völter M (2006). "oAW xText: A Framework for Textual DSLs." *Proceedings of Workshop on Modeling Symposium at Eclipse Summit*.

Flegal JM, Jones GL (2010). "Batch Means and Spectral Variance Estimators in Markov Chain Monte Carlo." *The Annals of Statistics*, **38**(2), 1034–1070. ISSN 00905364. doi:10.1214/09-AOS735.

Friesen J (2015). *Java Threads and the Concurrency Utilities*. Apress, Berkeley, CA. ISBN 978-1-4842-1699-6. doi:10.1007/978-1-4842-1700-9.

Geweke J (2004). "Getting It Right: Joint Distribution Tests of Posterior Simulators." *Journal of the American Statistical Association*, **99**(467), 799–804. ISSN 01621459.

Geyer CJ (1991). "Markov Chain Monte Carlo Maximum Likelihood." *Computing Science and Statistics: Proc. 23rd Symposium on the Interface, Interface Foundation, Fairfax Station, VA*, (1), 156–163.

Goodman ND, Mansinghka VK, Roy DM, Bonawitz K, Tenenbaum JB (2012). "Church: A Language for Generative Models." *CoRR*, **abs/1206.3255**. 1206.3255.

Greiner J, Burgess JM, Savchenko V, Yu HF (2016). "ON THEFERMI-GBM EVENT 0.4 s AFTER GW150914." *The Astrophysical Journal*, **827**(2), L38. doi:10.3847/2041-8205/827/2/l38.

Gronau QF, Singmann H (2018). ***bridgesampling**: Bridge Sampling for Marginal Likelihoods and Bayes Factors*. R package version 0.6-0, URL https://CRAN.R-project.org/package=bridgesampling.

Gronau QF, Singmann H, Wagenmakers EJ (2017). "**bridgesampling**: An R Package for Estimating Normalizing Constants." 1710.08162.

Höhna S, Landis M, Heath T, Boussau B, Lartillot N, Moore B, Huelsenbeck J, Ronquist F (2016). "RevBayes: Bayesian Phylogenetic Inference Using Graphical Models and an Interactive Model-Specification Language." *Systematic Biology*, **65**, 1–11. doi:10.1093/sysbio/syw021.

Ising E (1925). "Beitrag zur Theorie des Ferromagnetismus." *Zeitschrift für Physik*, **31**(1), 253–258. ISSN 0044-3328. doi:10.1007/BF02980577.

Lakner C, van der Mark P, Huelsenbeck JP, Larget B, Ronquist F (2008). "Efficiency of Markov Chain Monte Carlo Tree Proposals in Bayesian Phylogenetics." *Systematic Biology*, **57**(1), 86–103. ISSN 1063-5157. doi:10.1080/10635150801886156.

Leiserson CE, Schardl TB, Sukha J (2012). "Deterministic Parallel Random-Number Generation for Dynamic-Multithreading Platforms."

Lunn D, Jackson C, Best N, Thomas A, Spiegelhalter D (2012). *The BUGS Book Statistics The BUGS Book*. First edition. Chapman and Hall/CRC. ISBN 9781584888499.

Lunn D, Spiegelhalter D, Thomas A, Best N (2009). "The BUGS project: Evolution, Critique and Future Directions." *Statistics in Medicine*, **28**(25), 3049–3067. ISSN 0277-6715. doi:10.1002/sim.3680.

Lunn DJ, Thomas A, Best N, Spiegelhalter D (2000). "WinBUGS - A Bayesian Modelling Framework: Concepts, Structure, and Extensibility." *Statistics and Computing*, **10**(4), 325–337. ISSN 1573-1375. doi:10.1023/A:1008929526011.

Meng XL, Wong WH (1996). "Simulating Ratios of Normalizing Constants via a Simple Identity: A Theoretical Exploration." *Statistica Sinica*, **6**, 831–860.

Metropolis N, Rosenbluth AW, Rosenbluth MN, Teller AH, Teller E (1953). "Equation of State Calculations by Fast Computing Machines." *The Journal of Chemical Physics*, **21**(6), 1087–1092. ISSN 0021-9606. doi:10.1063/1.1699114.

Milch B, Marthi B, Russell S, Sontag D, Ong DL, Kolobov A (2005). "BLOG: Probabilistic Models with Unknown Objects." *IJCAI: Proceedings of the 19th International Joint Conference on Artificial Intelligence*, pp. 1352–1359.

Mitchell TJ, Beauchamp JJ (1988). "Bayesian Variable Selection in Linear Regression." *Journal of the American Statistical Association*, **83**(404), 1023–1032. ISSN 0162-1459. doi:10.1080/01621459.1988.10478694.

Murray LM, Schön TB (2018). "Automated Learning with a Probabilistic Programming Language: Birch." *Annual Reviews in Control*, **46**, 29–43. ISSN 1367-5788. doi:10.1016/j.arcontrol.2018.10.013.

Neal RM (2001). "Annealed Importance Sampling." *Statistics and Computing*, **11**(2), 125–139. ISSN 09603174. doi:10.1023/A:1008923215028. 9803008.

Neal RM (2003). "Slice Sampling." *The Annals of Statistics*, **31**(3), 705–741. ISSN 00905364. doi:10.1214/aos/1056562461.

Neal RM (2012). "MCMC using Hamiltonian Dynamics." 1206.1901.

Ogata Y (1989). "A Monte Carlo Method for High Dimensional Integration." *Numerische Mathematik*, **55**(2), 137–157. ISSN 0029-599X. doi:10.1007/BF01406511.

Paige B, Wood F (2014). "A Compilation Target for Probabilistic Programming Languages." In EP Xing, T Jebara (eds.), *Proceedings of the 31st International Conference on Machine Learning*, volume 32 of *Proceedings of Machine Learning Research*, pp. 1935–1943. PMLR, Bejing, China.

Paige B, Wood F (2016). "Inference Networks for Sequential Monte Carlo in Graphical Models." In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ICML'16, pp. 3040–3049. Event-place: New York, NY, USA.

Plummer M (2003). "JAGS: A Program for Analysis of Bayesian Graphical Models Using Gibbs Sampling." *Proceedings of the 3rd International Workshop on Distributed Statistical Computing*. ISSN 1609-395X.

Xtend (2019). "Xtend." URL `https://www.eclipse.org/xtend/documentation/203_xtend_expressions.html`.

Salvatier J, Wiecki T, Fonnesbeck C (2015). "Probabilistic Programming in Python Using PyMC." `1507.08050`.

Semmens BX, Ward EJ, Moore JW, Darimont CT (2009). "Quantifying Inter- and Intra-Population Niche Variability Using Hierarchical Bayesian Stable Isotope Mixing Models." *PLOS ONE*, **4**(7), 1–9. `doi:10.1371/journal.pone.0006187`.

Steele G, Lea D (2013). "Splittable Random Application Programming Interface." URL `https://docs.oracle.com/javase/8/docs/api/java/util/SplittableRandom.html`.

Steorts RC, Hall R, Fienberg SE (2016). "A Bayesian Approach to Graphical Record Linkage and Deduplication." *Journal of the American Statistical Association*, **111**(516), 1660–1672. ISSN 0162-1459. `doi:10.1080/01621459.2015.1105807`.

Syed S, Bouchard-Côté A, Deligiannidis G, Doucet A (2019). "Non-Reversible Parallel Tempering: an Embarassingly Parallel MCMC Scheme." `1905.02939`.

Tancredi A, Liseo B (2011). "A Hierarchical Bayesian Approach to Record Linkage and Population Size Problems." *The Annals of Applied Statistics*, **5**(2B), 1553–1585. ISSN 1932-6157. `doi:10.1214/10-AOAS447`.

Wickham H (2014). "Tidy Data." *Journal of Statistical Software, Articles*, **59**(10), 1–23. ISSN 1548-7660. `doi:10.18637/jss.v059.i10`.

Wood F, van de Meent JW, Mansinghka V (2014). "A New Approach to Probabilistic Programming Inference." In *Proceedings of the 17th International conference on Artificial Intelligence and Statistics*, pp. 1024–1032.

Zhao T, Cumberworth A, Wang Z, Gsponer J, de Freitas N, Bouchard-Côté A (2015). "Bayesian Analysis of Continuous Time Markov Chains with Application to Phylogenetic Modelling." *Bayesian Analysis*, **11**, 1203–1237.

Zhou Y, Johansen AM, Aston JA (2016). "Toward Automatic Model Comparison: An Adaptive Sequential Monte Carlo Approach." *Journal of Computational and Graphical Statistics*, **25**(3), 701–726. ISSN 15372715. `doi:10.1080/10618600.2015.1060885`. `1303.3123`.

A. Internal architecture

This section documents the high-level implementation decisions and trade-offs involved in the language construction. They may be skipped at first reading.

A.1. Language infrastructure

Blang is developed using Xtext, a mature framework for programming language design supported by the Eclipse Foundation and TypeFox. Thanks to the Xtext infrastructure, Blang incorporates a feature set comparable to many modern full fledge multi-paradigm language: functional, generic and object programming, static typing. Blang also automatically inherits state-of-the-art language development tools including a graphical integrated development environment (IDE) which leverages static types to provide insight into large Blang projects. The IDE also has a full-feature debugger, and plug-ins have been tested to perform profiling and code coverage analysis.

A.2. Choice of compilation target

Under the hood, Blang is compiled into Java, which in turn is compiled into Java Virtual Machine (JVM) bytecode. This *transpilation* step does not have marked effect on amortized compilation time since we use compilers supporting incremental compilation.

This is the default model in Xtext, which, in addition to greatly simplifying Xtext development by using most of the provided default behaviour, has for the user's perspective two advantages related to performance and production deployment. First, code running on modern JVM is fast. For example, on the leading crowd-sourced language performance benchmark,[13] as of June 2019, the geometric mean performance of Java is lower than C++, but higher than Julia, which itself outperforms the more common statistical computing choices such as R and Python by an order of magnitude or more. The performance gains of advanced compilers such as Java and Julia over R and Python are especially important when dealing with combinatorial spaces where vectorization is generally not possible. Other performance advantages include the JVM's high-performance multi-threading capacity and garbage collection algorithms, which greatly facilitated the development of advanced Monte Carlo algorithms, for example for the parallel computation and memory management of particle genealogies. The second advantage is related to production deployment. Java is currently the most used language according to the TIOBE index as of June 2019, and this state may ease deployment of Blang software into existing production environments.

An often cited downside of using Java is its verbosity. In our context, one specific concern is that more boilerplate code is typically needed to access high-performance computing libraries such as linear algebra libraries or random number generators. Fortunately, Blang and Xtend avoid the key issues that make Java code verbose: checked exception, bad default behaviour for constructor/accessors, and redundant type declaration. This brings Blang and Xtend code to a length similar to even non-statically typed language while preserving the advantages of the static type system. We use Blang and Xtend advanced language features combined with allowed operator overloading to wrap existing dense and sparse matrix libraries into xlinear, a new linear algebra library written in Xtend, which provide succinct linear algebra

[13]https://benchmarksgame-team.pages.debian.net/benchmarksgame/which-programs-are-fastest.html

expressions to Blang. Similarly, we wrap existing random generation libraries into convenient Xtend extension methods.

A.3. Choice of sampler state representation

The state of the sampler is modified in place. A priori, this choice appears in conflict with a popular doctrine in software engineering which is to avoid mutability and instead use functional-style idioms on immutable data structures. While we agree these functional patterns are often tremendously helpful, in the context of our samples' state representation, we found mutable data structures more useful for three reasons. First, the way we pre-compute a factor graph for efficient inference, via scoping analysis, assumes that certain references in the object graph stay invariant. These invariant objects allow us to gain information on the scope and hence dependencies. With functional style programming, we would trade immutability of the values into more mutability of the references making this scoping analysis complex. Second, since the state objects are assembled and used in a completely automated way (via Java reflection), the user simply does not face the traps of mutable data structures in this specific context. Third, there are computational complexity advantages to using mutable data structure: for example accessing or modifying array cost $O(1)$ instead of $O(\log n)$ for their functional copy on write counterparts.

B. Library dependencies

Contrary to many other PPLs, Blang's standard library uses its own language, and as such the majority of the dependencies were developed for Blang, and are handled automatically through **Maven** and Gradle.[14] Aside from libraries developed for Blang, **briefj**, **inits**, **bayonet**, **rejfree**, **binc**, **xlinear**, and **pxviz**, the language depends on three additional, external libraries: **Cloning**[15], **JGraphT**[16], and **Xbase**[17]. Figure 12 summarizes each of the aforementioned packages, while the remainder of this section expands on a select few that have been referenced earlier in the paper.

[14]`https://gradle.org/`
[15]`https://mvnrepository.com/artifact/uk.com.robust-it/cloning/1.9.6`
[16]`https://mvnrepository.com/artifact/org.jgrapht/jgrapht-core/0.9.0`
[17]`https://wiki.eclipse.org/Xbase`

Library	Description
briefj	Utilities for writing succint Java code.
inits	A framework to organize inputs and outputs of scientific simulations.
bayonet	Various low-level utilities for probabilistic inference.
rejfree	Implementation of a Bouncy Particle Sampler.
binc	An interface for calling binary programs from Java applications.
xlinear	Linear algebra package for Xtend and Java.
pxviz	A visualization library.
Cloning	Deep cloning library for Java.
JGraphT	Graph theory data structures and algorithms for optimizing samplers.
Xbase	Used as the base language for the DSL.

Figure 12: A summary of library dependencies.

B.1. bayonet

The **bayonet** `https://github.com/alexandrebouchard/bayonet` library contains utilities for performing probabilistic inference. Blang uses **bayonet.distribution.Random** as a replacement for **java.util.Random** for random number generation. This alternative is compatible with both Java and **Math Commons** random types. **bayonet.math.SpecialFunctions** provides several statistical utility functions that are used heavily in Blang.

B.2. inits

inits `https://github.com/UBC-Stat-ML/inits` is a framework for performing scientific simulations, and can be viewed as a dependency injection framework tailored to complex and hierarchical command-line arguments. Blang's CLI argument setup is automatically handled by **inits**.

B.3. xlinear

Blang's linear algebra is based on **xlinear** `https://github.com/alexandrebouchard/xlinear`, which itself relies on **Apache Commons**, **parallel COLT**, and **JEigen**. The simple API of **xlinear** and the operator overloading functionality is what is leveraged in Blang to augment the `DenseMatrix` and `SparseMatrix` types into `DenseSimplex` and `DenseTransitionMatrix`.

C. Output format

Output organization

Every Blang execution creates a unique directory. The path is outputted to standard out at the end of the program's execution/run. The latest run is also softlinked at `results/latest`.

The directory has the following structure:

- `arguments-details.txt`: a detailed list of all arguments and options.
- `arguments.tsv`: arguments used in current run.
- `executionInfo`: information for reproducibility (JVM arguments, version of the code, standard out, etc).
- `init`: information for simulation.
- `monitoring`: diagnostics for samplers.
- `samples`: samples from the target distribution. By default each random variable in the running model is output for each iteration (to disable this for some variables, e.g., those that are fully observed, use `--excludeFromOutput`).
- `logNormEstimate.txt`: estimate of the natural logarithm of the probability of the data (also known as the log of the normalization constant of the prior times the likelihood, integrating over the latent variables). Only available for certain inference engines such as SCM.

Additional files and directories if `--postProcessor DefaultPostProcessor` is specified:

- `ess`: information for ess and energy for each chain.
- `monitoringPlots`: sampler diagnostic plots
- `posteriorPlots`: plots of log-density and energy for each chain.
- `summaries`: summary statistics of log-density and energy for each chain.
- `tracePlots`: trace plots for the random variables, log-density, and energy for each chain with burn-in samples discarded.
- `tracePlotsFull`: trace plots with all samples included.

Format of the samples

The samples are stored in Tidy CSV files. For example, two samples for a `java.util.List` of two `RealVar`'s would look like:

index	**sample**	**value**
0	0	0.453
1	0	0.386
0	1	0.511
1	1	0.345

By default, the method `toString` is used to create the last column (value). This behaviour can be customized to follow the tidy philosophy by implementing the interface `TidilySerializable`.

Output options

The following command-line arguments can be used to tune the output:

- `--excludeFromOutput`: space-separated list of random variables to exclude from output.
- `--experimentConfigs.managedExecutionFolder`: set to false in order to output in the current folder instead of in the unique folder created in results/all.
- `--experimentConfigs.recordExecutionInfo`: set to false to skip recording the reproducibility information in executionInfo.
- `--experimentConfigs.recordGitInfo`: set to false to skip git repository lookup for the code.
- `--experimentConfigs.saveStandardStreams`: set to false to skip recording the standard out and err.
- `--experimentConfigs.tabularWriter`: by default set to `CSV`. Can set to `Spark` to organize tidy output into a hierarchy of directories each having a csv (with less columns, as many columns in this format are now inferable from the names of the parent directories). In certain scenarios this could save disk space. Inter-operable with Spark.

D. List of probability distributions in Blang's library

D.1. Discrete distributions

`Bernoulli`: Any random variable taking values in {0, 1}.

- `param RealVar probability`: Probability $p \in [0, 1]$ that the realization is one.

`BetaBinomial`: A sum of n iid Bernoulli variables, with a marginalized Beta prior on the success probability. Values in $(0, 1, 2, \ldots, n)$.

- `param IntVar numberOfTrials`: The number n of Bernoulli variables being summed. $n > 0$
- `param RealVar alpha`: Higher values brings mean closer to one. $\alpha > 0$
- `param RealVar beta`: Higher values brings mean closer to zero. $\beta > 0$

`Binomial`: A sum of n iid Bernoulli variables. Values in {0, 1, 2, …, n}.

- `param IntVar numberOfTrials`: The number n of Bernoulli variables being summed. $n > 0$
- `param RealVar probabilityOfSuccess`: The parameter $p \in [0, 1]$ shared by all the Bernoulli variables (probability that they be equal to 1).

`Categorical`: Any random variable over a finite set {0, 1, 2, …, $n - 1$}.

- `param Simplex probabilities`: Vector of probabilities $(p_0, p_1, \ldots, p_{n-1})$ for each of the n integers.

`DiscreteUniform`: Uniform random variable over the contiguous set of integers $\{m, m + 1, \ldots, M - 1\}$.

- `param IntVar minInclusive`: The left point of the set (inclusive). $m \in (-\infty, M)$
- `param IntVar maxExclusive`: The right point of the set (exclusive). $M \in (m, \infty)$

`Geometric`: The number of unsuccessful Bernoulli trials until a success. Values in $\{0, 1, 2, \ldots\}$

- `param RealVar p`: The probability of success for each Bernoulli trial.

`HyperGeometric`: Hyper-geometric distribution with population N and population satisfying certain condition K and drawing n samples.

- `param IntVar numberOfDraws`: number of samples n

- `param IntVar population`: number of population N
- `param IntVar populationConditioned`: number of population satisfying condition K

`NegativeBinomial`: Number of successes in a sequence of iid Bernoulli until (r) failures occur. Values in $\{0, 1, 2, \dots\}$.

- `param RealVar r`: Number of failures until experiment is stopped (generalized to the reals). $r > 0$
- `param RealVar p`: Probability of success of each experiment. $p \in (0, 1)$

`Poisson`: Poisson random variable. Values in $\{0, 1, 2, \dots\}$.

- `param RealVar mean`: Mean parameter λ. $\lambda > 0$

`YuleSimon`: An exponential-geometric mixture.

- `param RealVar rho`: The rate of the mixing exponential distribution.

D.2. Continuous Distributions

`Beta`: Beta random variable on the open interval (0, 1).

- `param  RealVar alpha`: Higher values brings mean closer to one. $\alpha > 0$
- `param  RealVar beta`: Higher values brings mean closer to zero. $\beta > 0$

`ChiSquared`: Chi Squared random variable. Values in $(0, \infty)$.

- `param  IntVar nu`: The degrees of freedom ν. $\nu > 0$

`ContinuousUniform`: Uniform random variable over a close interval $[m, M]$.

- `param  RealVar min`: The left end point m of the interval. $m \in (\infty, M)$
- `param  RealVar max`: The right end point of the interval. $M \in (m, \infty)$

`Exponential`: Exponential random variable. Values in $(0, \infty)$.

- `param  RealVar rate`: The rate λ, inversely proportional to the mean. $\lambda > 0$

`F`: The F-distribution. Also known as Fisher-Snedecor distribution. Values in $(0, \infty)$.

- `param RealVar d1, d2`: The degrees of freedom d_1 and d_2 . $d_1, d_2 > 0$

`Gamma`: Gamma random variable. Values in $(0, \infty)$.

- `param  RealVar shape`: The shape α is proportional to the mean and variance. $\alpha > 0$
- `param  RealVar rate`: The rate β is inverse proportional to the mean and quadratically inverse proportional to the variance. $\beta > 0$

`Gompertz`: The Gompertz distribution. Values in $[0, \infty)$.

- `param RealVar shape`: The shape parameter ν. $nu > 0$
- `param RealVar scale`: The scale parameter b. $b > 0$)

`Gumbel`: The Gumbel Distribution. Values in $\mathbb{R}$.

- `param RealVar location`: The location parameter μ. $\mu \in \mathbb{R}$
- `param RealVar scale`: The scale parameter β. $\beta > 0$

`HalfStudentT`: HalfStudentT random variable. Values in $(0, \infty)$.

- `param RealVar nu`: A degree of freedom parameter ν. $\nu > 0$
- `param RealVar sigma`: A scale parameter σ. $\sigma > 0$

`Laplace`: The Laplace Distribution over $\mathbb{R}$.

- `param RealVar location`: The mean parameter.
- `param RealVar scale`: The scale parameter b, equal to the square root of half of the variance. $b > 0$

`Logistic`: A random variable with a logistic probability distribution function. Values in $\mathbb{R}$.

- `param RealVar location`: The center of the PDF. Also the mean, mode and median. $\mu \in \mathbb{R}$
- `param RealVar scale`: The scale parameter. $s > 0$

`LogLogistic`: A log-logistic distribution is the probability distribution of a random variable.

- `param RealVar scale`: The scale parameter α and also the median. $\alpha > 0$
- `param RealVar shape`: The shape parameter β. $\beta > 0$

`Normal`: Normal random variables. Values in $\mathbb{R}$.

- `param RealVar mean`: Mean μ. $\mu \in \mathbb{R}$
- `param RealVar variance`: Variance σ^2. $\sigma^2 > 0$

`StudentT`: Student T random variable. Values in $\mathbb{R}$.

- `param RealVar nu`: The degrees of freedom ν. $\nu > 0$
- `param RealVar mu`: Location parameter μ. $\mu \in \mathbb{R}$
- `param RealVar sigma`: Scale parameter σ. $\sigma > 0$

`Weibull`: The Weibull Distribution. Values in $(0, \infty)$.

- `param RealVar scale`: The scale parameter λ. $\lambda \in (0, \infty)$
- `param RealVar shape`: The shape parameter k. $k \in (0, \infty)$

D.3. Multivariate distributions

`Dirichlet`: The Dirichlet distribution over vectors of probabilities $(p_0, p_1, \ldots, p_{n-1})$. $p_i \in (0, 1)$, $\sum_i p_i = 1$.

- `param Matrix concentrations`: Vector $(\alpha_0, \alpha_1, \ldots, \alpha_{n-1})$ such that increasing the ith component increases the mean of entry p_i.

`MultivariateNormal`: Arbitrary linear transformations of n iid standard normal random variables.

- `param Matrix mean`: An $n \times 1$ vector μ. $\mu \in \mathbb{R}^n$
- `param CholeskyDecomposition precision`: Inverse covariance matrix Λ, a positive definite $n \times n$ matrix.

`NormalField`: A mean-zero normal, sparse-precision Markov random field.

- `param Precision precision`: Precision matrix structure.

`SimplexUniform`: n dimensional Dirichlet with all concentrations equal to one.

- `param Integer dim`: The dimensionality n. $n > 0$

`SymmetricDirichlet`: n dimensional Dirichlet with all concentrations equal to $\frac{\alpha}{n}$.

- `param Integer dim`: The dimensionality n. $n > 0$
- `param RealVar concentration`: The shared concentration parameter α before normalization by the dimensionality. $\alpha > 0$

Miscellaneous

`LogPotential`: Not really a distribution, but rather a way to handle undirected model (or random fields).

- `param RealVar logPotential`: The log of the current value of this potential.

E. Frequently used functions

Any Java function can be called in Blang. The functions in Figure 13 and Figure 14 are automatically and statically imported for easy access. The functions below are the most useful of those imported and in addition to the functions, Blang also imports two fields from **java.lang.Math**, which are `E` and `PI`.

Function	Description
`abs(double value)`	absolute value
`acos(double a)`	arccosine
`asin(double a)`	arcsine
`atan(double a)`	arctangent
`cbrt(double a)`	cube root
`ceil(double a)`	ceiling
`cos(double a)`	cosine
`cosh(double a)`	hyperbolic cosine
`exp(double a)`	exponential base e
`floor(double a)`	floor
`log(double a)`	logarithm base e
`log10(double a)`	logarithm base 10
`max(double a, double b)`	maximum of `a` and `b`
`min(double a, double b)`	minimum of `a` and `b`
`pow(double a, double b)`	`a` to the power `b`
`signum(double a)`	signum function
`sin(double a)`	sine
`sinh(double a)`	hyperbolic sine
`sqrt(double a)`	square root
`tan(double a)`	tangent
`tanh(double a)`	hyperbolic tangent

Imported Fields	Description
`E`	Java's `double` value for e
`PI`	Java's `double` value for π

Figure 13: Functions imported from **java.lang.Math**. Note that all trigonometric operations use angles expressed in radians and that the return type of all functions listed above are `double`.

Function	Description
`erf(double a)`	error function
`inverseErf(double a)`	inverse error function
`logistic(double a)`	standard logistic function
`logit(double a)`	standard logit function
`logBinomial(int n, int k)`	logarithm of $\binom{n}{k}$
`lnGamma(double alpha)`	logarithm of the gamma function of `alpha`
`logFactorial(int input)`	logarithm of the factorial of `input`
`multivariateLogGamma(int dim, double a)`	logarithm of the multivariate gamma function

Figure 14: Functions imported from **bayonet.math.SpecialFunctions**. All return types are `double`.

Function	Description	Return Type
`latentInt()`	unobserved integer variable (initialized at zero)	`IntScalar`
`latentReal()`	unobserved real variable (represented as a double, initialized at zero)	`RealScalar`
`fixedInt(int value)`	fixed (constant or conditioned upon) integer scalar	`IntConstant`
`fixedReal(double value)`	fixed real scalar	RealConstant
`latentIntList(int size)`	size specifies the length of the list	`List<IntVar>`
`latentRealList(int size)`	size specifies the length of the list	`List<RealVar>`
`fixedIntList(int ... entries)`	list where the integer valued entries are fixed to the provided values	`List<IntVar>`
`latentVector(int n)`	an n-by-1 latent dense vector (initialized at zero)	`DenseMatrix`
`fixedVector(double ... entries)`	an n-by-1 fixed dense vector	`DenseMatrix`

Figure 15: Functions used to initialize random variables.

Function	Description	Return Type
`latentMatrix(int nRows, int nCols)`	an n-by-m latent dense matrix (initialized at zero)	`DenseMatrix`
`fixedMatrix(double [][] entries)`	a constant dense matrix	`DenseMatrix`
`latentSimplex(int n)`	latent n-by-1 matrix with entries summing to one (initialized at uniform)	`DenseSimplex`
`fixedSimplex(double ... probs)`	creates a constant simplex, also checks the provided list of number sums to one	`DenseSimplex`
`fixedSimplex(DenseMatrix probs)`	creates a constant simplex, also checks the provided vector sums to one	`DenseSimplex`
`latentTransitionMatrix(int nStates)`	latent n-by-n matrix with rows summing to one	`DenseTransitionMatrix`
`fixedTransitionMatrix(DenseMatrix probs)`	creates a constant transition matrix, also checks the provided rows all sum to one	`DenseTransitionMatrix`
`fixedTransitionMatrix(double [][] probs)`	creates a constant transition matrix, also checks the provided rows all sum to one.	`DenseTransitionMatrix`

Figure 16: Frequently used functions in Blang to complement the set of functions from the **xlinear** library.

Function	Description	Return Type
`getRealVar(Matrix m, int row, int col)`	View a single entry of a `Matrix` as a `RealVar`	`Realvar`
`getRealVar(Matrix m, int index)`	View a single entry of a 1-by-n or n-by-1 Matrix as a `RealVar`.	`RealVar`
`generator(java.util.Random random)`	Upgrade a `java.util.Random` into to the type of `Random` Blang uses, `bayonet.distributions.Random`.	`Random`
`setTo(Matrix one, Matrix another)`	Copy the contents of a matrix into another one.	`void`
`sum(Iterable<? extends Number> numbers)`		
`increment(Map<T, Double> map, T key, double value)`	Increment an entry of a map to double, setting to the value if the key is missing.	`void`
`isClose(double n1, double n2)`	Check if two numbers are within 1e-6 of each other.	`boolean`

Figure 17: Extension methods (automatically) imported from **blang.types.ExtensionUtils**.

Function	Description	Return Type
`getName()`	Human-readable name for the plate, typically automatically extracted from a `DataSource` column name.	`ColumnName`
`indices(Query parentIndices)`	Get the indices available given the indices of the parent (enclosing) plates. The parents can be provided in any order.	`Collection<Index<K»`
`ofIntegers(ColumnName columnName, int size)`	a plate with indices `0, 1, 2, ... size-1`	`Plate<Integer>`
`ofStrings(ColumnName columnName, int size)`	a plate with indices `category_0, category_1, ...`	`Plate<String>`
`ofStrings(String columnName, int size)`	a plate with indices `category_0, category_1, ...`	`Plate<String>`
`get(Index<?> ... indices)`	get the random variable or parameter indexed by the provided indices. The indices can be given in any order.	`T`
`entries()`	list all variables obtained through `get(...)` so far. Each returned entry contains the variable as well as the associated indices (`Query`).	`Collection<Entry<Quesry, T»`
`slice(Index<?> ... indices)`	a view into a subset of the plated variable.	`Plated<T>`
`latent(ColumnName name, Supplier<T> supplier)`	use the provided lambda expression ot initialize several latent variables.	`<T> Plated<T>`

Figure 18: Functions and methods related to `Plates` (above) and `Plated` (below) types.

Affiliation:

Alexandre Bouchard-Côté
Department of Statistics
Faculty of Science
University of British Columbia
3182 Earth Sciences Building, 2207 Main Mall Vancouver, BC Canada V6T 1Z4
E-mail: `bouchard@stat.ubc.ca`
URL: `https://www.stat.ubc.ca/~bouchard/`

Journal of Statistical Software `http://www.jstatsoft.org/`
published by the Foundation for Open Access Statistics `http://www.foastat.org/`
MMMMMM YYYY, Volume VV, Issue II *Submitted:* yyyy-mm-dd
`doi:10.18637/jss.v000.i00` *Accepted:* yyyy-mm-dd