Journal of Statistical Software

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

Blang: Bayesian Declarative Modelling of General Data Structures and Inference via Distribution Continua

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 sequential Monte Carlo and 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 data types, in particular, combinatorial spaces which are of central importance in areas such as computational biology. The design philosophy is centred 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). See van de Meent *et al.* (2018) for a recent survey.

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); Zhou *et al.* (2019); Ronquist *et al.* (2020)) but in typical applications that involve challenging combinatorial spaces, general purpose PPL inference engine are not yet able to match the performance of specialized samplers. 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 2011), 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 in combinatorial spaces has been achieved by designing portfolios of specialized samplers in a case-by-case basis (see e.g., Lakner *et al.* (2008)). This process is typically time consuming and error prone. There is an opportunity to simplify this process, minimize manual intervention of tuning algorithms, and to speed-up and parallelize inference. This is possible with 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 a non-standard flavour of sequential Monte Carlo (SMC) method that we call Sequential Change of Measure (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 as 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.

One of the existing PPLs most closely related to Blang is RevBayes (Höhna *et al.* 2014, 2016). RevBayes is a declarative PPL which provides extensive support for Bayesian inference over phylogenetic trees, an archetypical example of a challenging combinatorial space. Moreover, RevBayes supports interactive usage, a functionality currently not supported in Blang. However, as phylogenetic inference is the primary domain targeted by RevBayes, users interested in combinatorial spaces other than phylogenetic trees will benefit from Blang's abstractions which target arbitrary combinatorial spaces.

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 7. Next, we discuss how users can utilize Blang's Software Development Kit (SDK) to implement complex models in Section 10, followed by a description of its architecture as a whole in Section 12.

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 (Wickham 2014), samples output in Tidy format).

Generality: As a programming language, Blang is Turing-complete and equipped with an open type system, as well as facilities to quickly develop and test sampling algorithms for new types. By open type system, we mean that the set of types is not limited to integers and real numbers, and can be arbitrary classes. Blang does not fully automate the process of posterior sampling from user-defined types but instead greatly facilitates the development, composition and sharing of custom sampling algorithms.

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, source and documentation 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`. Online documentation is available at `https://www.stat.ubc.ca/~bouchard/blang/`, including Javadoc pages at `https://www.stat.ubc.ca/~bouchard/blang/Javadoc.html`.

4. Tutorial

This section aims to introduce readers to Blang by presenting a minimal working example. We begin with instructions for performing inference on a simple model using the command-line interface (CLI). Realistic applications are demonstrated in Sections 6 and 9. Advanced tutorials can be found in Appendix A.

4.1. Installing Blang's command-line interface

We provide instructions here for installing and using Blang via CLI. Alternative Blang interfaces include an integrated development environment (IDE), detailed in Section 10.1.1, as well as a Web interface (Section 10.1.2). Instructions are also available from the documentation website under the link `Tools`.[1] Additionally, an R and Python interface to Blang are currently under development.[2]

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 (at the time of publication, Open SDK 8 and 11 are tested). 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/`.

[1] Documentation for Blang is available at `https://www.stat.ubc.ca/~bouchard/blang/`
[2] The interfaces and associated instructions will be hosted on `https://github.com/UBC-Stat-ML`

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 platforms 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).

4.2. Posterior inference

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

`Doomsday.bl`[3]

```
package toy

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

The first line is a package declaration, which identifies the package in which the Doomsday model belongs to. The remaining code 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.

[3] `https://github.com/UBC-Stat-ML/JSSBlangCode/blob/master/example/jss/Doomsday.bl`. Note the package statements are different.

- `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`.[4]

By default, `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
```

The same model can be run via the Eclipse IDE, following instructions from Section 10.1, or 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 {
    ...
 } [ ... ]
Inference {
   ...
 } [ ... ]
Postprocess {
   ...
 } [ ... ]
executionMilliseconds : 1037
outputFolder: ./JSSBlangCode/example/results/all/2019-06-27-14-13-21-RL.exec
```

[4]There are several motivations behind this design choice deviating from BUGS. Technically, static analysis could identify the list of variables we are conditioning on. However the notation used here is closer to a mathematical notation used for example in the Bayesian non-parametric literature (e.g., Teh *et al.* (2006); Griffiths and Ghahramani (2011)). More importantly however, the explicit conditioning allows us to generalize the notation to handle complex dependencies. This is demonstrated in Sections 11.1 and 11.2.

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 SCM which 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 multimodal distributions or weakly identifiable models. We describe the inference algorithms and their configuration in detail in Section 12.1.

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 $x = (x_1, x_2, \ldots, x_n)$ where $n < \infty$ is fixed. Despite n being finite in this formalism, each x_i is permitted to be of random or infinite dimensionality. The *type* or space in which the x_i's lie in, 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 7.10. Once each reference measure μ_i is given, by definition the densities are turned into distributions as follows:

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

Where A is some event, or more formally, an element of the $\sigma-$algebra of T. We also assume a decomposition for the parameters $\theta = (\theta_1, \theta_2, \ldots, \theta_m)$ 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 variables* 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 1 and 2. The variables marked with the `random` keyword are concatenated to form x, while those marked with `param` keyword are concatenated to form θ.

5.2. Interpretation of `laws` blocks

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

```
model Doomsday {

  param RealVar rate
  random RealVar y
  random RealVar z

  laws { ... }

}
```

Figure 1: 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 2: Mathematical notation.

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

Atomic laws 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. Additionally this factorization enables efficient sampling of latent variables, as only a fraction of factors will require evaluation per variable.

5.3. Interpretation of atomic laws

In the case of an atomic law, for each $k \in \{1, 2, \ldots, K\}$, 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:

`ContinuousUniform.bl`[5]

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

[5]https://github.com/UBC-Stat-ML/JSSBlangCode/blob/master/example/jss/others/ContinuousUniformExample.bl

```
  param  RealVar max

  laws {
    logf(min, max) {
      if (max - min <= 0.0) return NEGATIVE_INFINITY
      return - log(max - min)
    }
    logf(realization, min, max) {
      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 the chain rule. In the Doomsday example, this is just:

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

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 to write the second factor in Equation (3) using the previously defined `ContinuousUniform` model via

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

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, our notation can also be extended to useful novel patterns (see Sections 11.1 and 11.2).

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 10.5. It is also used for various purposes during posterior inference, for example, by providing a form of regeneration in PT, and to initialize SCM samplers.

5.7. Normal form

A laws block containing either only composite laws or only atomic laws is said to be in *normal form*. For certain inference engines (PT and SCM), laws blocks are required to be in normal form for Blang's runtime architecture to distinguish between observed versus latent variables. Furthermore, 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. Generative normal form is only required if the inference engine is PT or SCM, as samples from the prior are exploited for initialization and/or regeneration.

We show in Section 11.1 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. Tutorial—a complete example

We illustrate an example of posterior inference for a Gaussian mixture model (GMM). We highlight and briefly discuss key components in implementing a model, and showcase a series of post-processed statistics and plots. After a formal introduction of the syntax (Section 7), we will return to this example in the form of a summary in Section 8.

Consider the following model:

$$
\begin{aligned}
&\text{concentration} & \alpha &= [1, 1] \\
&\text{proportions} & \pi \mid \alpha &\sim \text{Dirichlet}(\alpha) \\
&\text{labels} & z_i \mid \pi &\sim \text{Categorical}(\pi) \\
&\text{means} & \mu_k &\sim \text{Normal}(0, 10^2) \\
&\text{standard deviations} & \sigma_k &\sim \text{Uniform}(0, 10) \\
&\text{observations} & y_i \mid \mu, \sigma, z_i &\sim \text{Normal}(\mu_{z_i}, \sigma^2_{z_i})
\end{aligned}
$$

for $i \in \{1, 2, \ldots, n\}$ and $k \in \{1, 2\}$.

We encode this GMM in Blang as follows:[6]

`MixtureModel.bl`[7]

```
package jss.gmm

model MixtureModel {

  random List<RealVar>  y
  param  Integer        n   ?: y.size
  param  Matrix         a   ?: fixedVector(1.0, 1.0)
  random List<IntVar>   z   ?: latentIntList(n)

  param  Integer        K   ?: 2
  random Simplex        pi  ?: latentSimplex(K)
  random List<RealVar>  mu  ?: latentRealList(K)
  random List<RealVar>  sd  ?: latentRealList(K)

  laws {

    pi | a ~ Dirichlet(a)

    for (int k : 0 ..< K) {
```

[6]Complete and commented implementations in this section are available in the reproduction materials located in the directory `reproduction_materials/example`.

[7]`https://github.com/UBC-Stat-ML/JSSBlangCode/blob/master/example/jss/gmm/MixtureModel.bl`

```
      mu.get(k) ~ Normal(0.0, 100.0)
      sd.get(k) ~ ContinuousUniform(0.0, 10.0)
    }

    for (int i : 0 ..< n) {
      z.get(i) | pi ~ Categorical(pi)
      y.get(i) | mu, sd, IntVar k = z.get(i)
        ~ Normal(mu.get(k), pow(sd.get(k), 2.0))
    }
  }
}
```

We began by declaring variables as we did in the Doomsday model. In addition to declarations, we initialized them to their respective latent types. Default initializations are expressed using `?:` followed by an expression in a syntax called XExpression described in detail in Section 7.11.[8] Default initializations can be overridden from the CLI (command line interface). We discuss this mechanism in detail in Sections 7.7. In this example, interpret initializations as creating instances of latent objects.[9] A list of data types available for latent variables can be found in Figures 15 and 16.

In the next code block, the `laws` block, we declared the distribution of each latent variable. We used `for` loops to encode a set of declarations. For example, the following two implementations are equivalent:

```
for (int k : 0 ..< 2) {
  mu.get(k) ~ Normal(0.0, 100.0)
  sd.get(k) ~ ContinuousUniform(0.0, 10.0)
}
```

and

```
mu.get(0) ~ Normal(0.0, 100.0)
mu.get(1) ~ Normal(0.0, 100.0)
sd.get(0) ~ ContinuousUniform(0.0, 10.0)
sd.get(1) ~ ContinuousUniform(0.0, 10.0)
```

To perform posterior inference on `MixtureModel` based on observed y_i's, we invoke the following commands in the CLI:

```
> git clone https://github.com/UBC-Stat-ML/JSSBlangCode.git
> cd JSSBlangCode/example
> blang --model jss.gmm.MixtureModel \
        --model.y file data/obs1.txt \
```

[8]Those familiar with Java can think of XExpressions as "shorthand Java" for now.

[9]In Section 9, we create a constructor for objects of type permutation. Its application is helpful in painting a bigger picture on how these latent objects are used behind the scenes.

```
        --engine PT \
        --engine.nChains 36 \
        --engine.nScans 30000 \
        --postProcessor DefaultPostProcessor
```

```
Preprocess {
   ...
 } [ ... ]
Inference {
   ...
 } [ ... ]
Postprocess {
  Post-processing allLogDensities
  Post-processing energy
  Post-processing z
  Post-processing logDensity
  Post-processing mu
  Post-processing nOutOfSupport
  Post-processing pi
  Post-processing sd
  MC diagnostics
} [ ... ]
executionMilliseconds : ...
outputFolder :./JSSBlangCode/example/all/2020-12-31-23-59-03-N0PvDjdc.exec
```

In this example, `obs1.txt` is a new-line separated file formatted as follows:

`obs1.txt`

```
 3.2
-0.3
 1.7
  ⋮
```

More generally, information on the format used to input data can be obtained by *appending* `--help` to the command line arguments (the command line help is contextual, so the information given by appending `--help` to the model and inference engine specific arguments will be more detailed than only using `blang --help`). A more sophisticated method to input data, based on the *plate notation*, is discussed in Section 10.4.1. We briefly summarize the key CLI arguments for the example below:

Argument	Description
`--model.y file`	Specifies the file path to a newline-separated file with `y`'s values.
`--engine PT`	Specifies Parallel Tempering as the inference algorithm.
`--engine.nChains`	Controls the number of (annealed) parallel Markov chains.
`--engine.nScans`	Controls the number posterior samples to draw.
`--postProcessor`	Specifies the post-processor.

The details of how `--engine` arguments influence the performance of inference are discussed in Section 12.

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 (raw output), post-processed statistics/plots (summaries of the samples), and monitoring statistics/plots (to assess the quality of the posterior approximation). Options for post-processing is handled via the `--postProccesor` runtime argument, accepting `DefaultPostProcessor` or `NoPostProcessor` as arguments. Again use `--postProccesor DefaultPostProcessor --help` for more information.

Currently, the `DefaultPostProcessor` option produces trace and density plots,[10] and provides summary statistics including Highest Density credible Intervals (HDI, constructed using the method described in Chen and Shao (1999)) and effective sample size (ESS) estimates (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`). Examples of summary statistics for `MixtureModel`'s parameters are shown below, and can be found under the directory summaries in `results/latest`.[11]

index	parameter	mean	sd	min	median	max	HDI.lower	HDI.upper
0	mean	0.66	1.77	−29.40	1.21	30.96	−1.33	2.52
1	mean	0.63	1.85	−39.99	1.20	29.20	−1.46	2.38
0	variance	1.17	1.14	0.01	0.83	9.95	0.07	2.57
1	variance	1.16	1.15	0.00	0.81	9.98	0.04	2.53
0	pi	0.50	0.23	0.00	0.51	0.99	0.16	0.83
1	pi	0.50	0.23	0.00	0.49	0.99	0.16	0.83

Notice the posterior summaries are nearly identical for the two mixture components. Similarly, the marginal posterior plots in Figure 3 also exhibit this symmetry. This symmetry is to be expected in this example: it arises from the unidentifiability of the GMM parameters known as label switching (Jasra *et al.* 2005). Here the inference engine used, an adaptive non-reversible parallel tempering algorithm (abbreviated PT), is capable of capturing this symmetry despite the high-dimensional multimodality involved (all variables z_i's have to be flipped to switch mode).

[10]The `DefaultPostProcessor` requires R as well as the packages **dplyr** and **ggplot2**.

[11]Numerical values are truncated to fit in the page width.

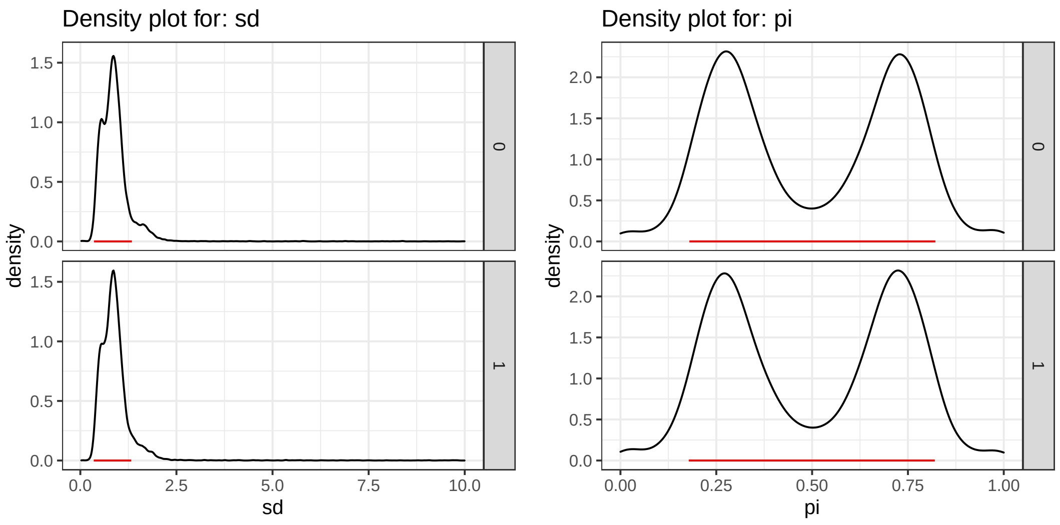


Figure 3: Posterior density plots for a subset of random variables in the GMM. The facets (rows) are indexed by the mixture components. Left: standard deviation parameters. Right: mixture proportion parameters. The two pairs of nearly-identical plots are indicative of successful label switching, showing that the multimodal posterior distribution is well approximated. By default, the 90% highest density interval is underlined in red.

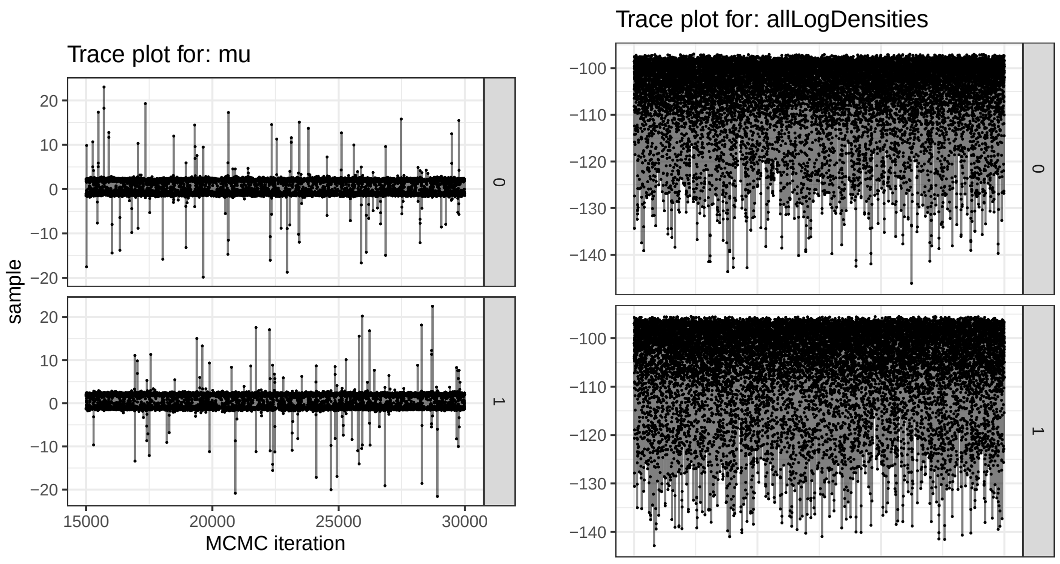


Figure 4: Left: Trace plot for cluster-specific location parameters. The two clusters are shown as facets. Right: log densities for two of the 36 tempered chains used in PT. Notice that the “jumps” between modes are densely distributed along the traces, i.e., they occur very frequently in this example. Other diagnostics produced will be discussed in Section 12.1.

Another statistic that is often of interest is the normalization constant (also known as model evidence, or marginal likelihood). The logarithm of this value is automatically output in `logNormalizationEstimate.csv`. The various methodologies available to estimate the log normalization constant are discussed in Section 12.4. Figure 5 illustrates the progression of estimates across PT adaptation rounds.

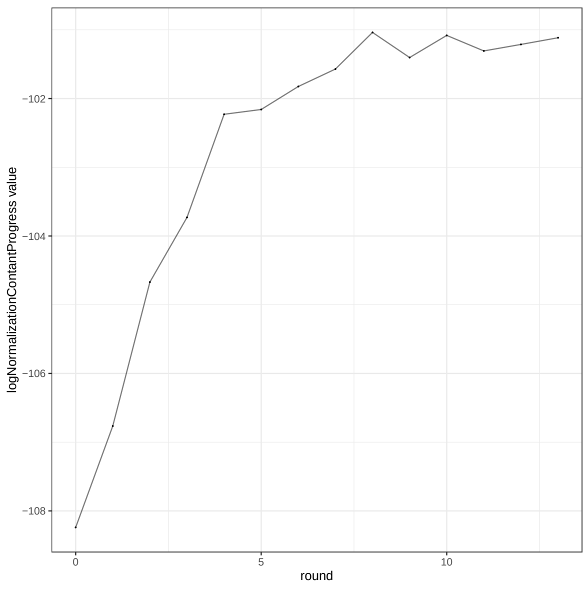


Figure 5: Log normalization constant estimates across adaptation rounds when the PT algorithm is used. The fact that these estimates plateaued supports that the allocated computational budget is sufficient for this inference task.

Output files for diagnosing and monitoring the performance of inference algorithms are also produced. We will describe them in Section 12.1.

7. 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.

7.1. 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 data types, 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.

7.2. 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 of 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.

7.3. 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
*/
```

7.4. Blang models: High-level syntax

A Blang file is organized as follows:

NameOfMyModel.bl

```
// package and import statements

model NameOfMyModel {

  // variables declarations

  laws {
    // laws declaration
  }

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

We briefly describe each code block as follows: package statements are responsible for defining the package in which a Blang model belongs to. Import statements are responsible for importing classes, functions, and models from other packages. The variables declarations block is responsible for declaring model variables, i.e., observed (constant) variables, latent variables, unknown parameters, known (constant) parameters. The laws block is used to declare the probability distribution associated with each of the (random) model variables (see

Section 7.8). The optional generate block is used for forward sampling from the model (see Section 7.9). It will also helpful to keep in mind that XExpressions (to be introduced) are imperative, while laws blocks are declarative.

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.

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

7.5. 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.

Packages in Blang work the same as in Java, and precede `import` statements. 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 another Blang `model` called `AnotherModel` from a package named `some.other.pack`, we can use `import` statements of the form:

```
import some.other.pack.AnotherModel
```

after the package declaration line. The same syntax can be used to import Java or Xtend classes, where `import static` is used to import a function, while a standalone `import` statement is used for types.

Package declarations effectively enable users to refer to specific objects of a package explicitly through import statements. In the example below we see why this would be useful. Suppose our model requires two data types from `package1` and `package2`, each of which contain an identically named but different implementation of `DupedType`. In the unlikely event of having to use two types with duplicated names within the same file, importing should be avoided (i.e., do not `import package1` nor `import package2`). Instead each instance of the type should be prefixed with the package name within the code, as such:

MyModel.bl

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

In contrast, here is an example of what *not* to do:

MyModel.bl

```
import package1.DupedType
import package2.DupedType

model MyModel{
  random DupedType var1
  random DupedType var2
  ...
}
```

A related construct is the extension import mechanism, described in more detail in Section 7.11.

7.6. Automatic imports in Blang files

Any Blang file automatically imports:[12]

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

[12]The relevant Javadocs can be found at `https://www.stat.ubc.ca/~bouchard/blang/Javadoc.html`.

7.7. Model Variables

Model variables encompass all observed (fixed) variables, latent variables, unknown parameters, and known (constant) parameters in a statistical model. Model variables are declared using one of two methods, declared with no default initialization:

```
random Type1 name1
param Type2 name2
```

or with default initialization:

```
random Type3 name3 ?: XExpression1
param Type4 name4 ?: XExpression2
```

Observed and latent random variables are declared with `random`, while parameters are declared with `param` (see Section 5.1). The initialization blocks, denoted by `XExpression1` and `XExpression2`, are imperative blocks of code surrounded by curly brackets used to provide default values in the absence of CLI arguments. If the block contains only one expression, the brackets can be omitted. 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 7.11 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.

7.8. Laws block

Laws blocks are used to *declare* the (conditional) probability distribution associated with each random variable. Note that unlike common programming languages used today for data analyses such as Python and R, the `laws` block is declarative. In particular, the interpretation of a model is invariant to the order in which the individual laws are declared in the code.

Composite laws

Described conceptually in Section 5.4, composite laws have the following syntax in Blang:

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

For example:

```
y | mu, variance  ~ Normal(mu + 123,  variance)
```

Where `variableExpression1`, `conditioning1`, `conditioning2`, `argumentExpression1`, and `argumentExpression2` correspond to `y`, `mu`, `variance`, `mu + 123`, and `variance` respectively.

`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 1).

The second method to specify a conditioning is as follows:

```
variableExpression1, variableExpression2, ...
  | MyType myConditioningVariable = XExpression1, ...
    MyDistributionName(argumentExpression1, argumentExpression2, ...)
```

where `MyType` is a type, `myConditioningVariable` is a local variable that exists only for the declaration of `variableExpression1, variableExpression2, ...`'s law. The code in `XExpression1` has access to all model variables. For example:

```
y | RealVar mu = manyMus.get(0) ~ Normal(mu, 1)
```

The `XExpression1` code is executed only once at initialization. We show in Section 11.2 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

Informally, atomic laws are used to compute factors, and are the building blocks for composite laws. Described conceptually in Section 5.3, atomic laws have the following syntax in Blang:

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

For example, $x \sim \text{Normal}(\mu, \sigma^2)$ (realization $\sim$ Normal(mean, variance)) would have the following encoding:

`Normal.bl`

```
laws {
  logf(mean, variance, realization) {
    if (variance < 0.0) return NEGATIVE_INFINITY
    return (- log(2*PI) / 2.0
            - 0.5 * log(variance)
            - 0.5 * pow(mean - realization, 2) / variance)
  }
}
```

It is recommended to separate factors with as few arguments together as possible, as this will help the runtime architecture determine dependencies and avoid redundant computation. For example, the `Normal.bl` implementation is recommended to be factorized as:

`Normal.bl` [13]

```
laws {
  logf() {
    - log(2*PI) / 2.0
  }
  logf(variance) {
    if (variance < 0.0) return NEGATIVE_INFINITY
    return - 0.5 * log(variance)
  }
  logf(mean, variance, realization)  {
    if (variance < 0.0) return NEGATIVE_INFINITY
    return - 0.5 * pow(mean - realization, 2) / variance
  }
}
```

[13] https://github.com/UBC-Stat-ML/blangSDK/blob/master/src/main/java/blang/distributions/Normal.bl

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[14] or Google's Guava project[15].
  An example iterating over a power set[16]:
  `for (Set<Integer> s : (0 ..< 5).powerSet) { ... }`
- Loops based on Xtend's or Java's utilities.[17] An example iterating over the first four even integers:
  `for (Integer i : (0 ..< 10).filter[it % 2 == 0]) { ... }`
  The keyword `it` is explained in section 7.11.12.
- The `Plate` data structure supplied by the Blang SDK is described in more detail in Section 10.4.

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

[14] https://docs.oracle.com/javase/tutorial/collections/index.html
[15] https://github.com/google/guava/wiki/CollectionUtilitiesExplained
[16] This requires the import line `import static extension com.google.common.collect.Sets.powerSet`
[17] Documentation for Xtend's utilities available at https://www.eclipse.org/xtend/documentation/203_xtend_expressions.html and documentation for Java's streams is available at https://docs.oracle.com/javase/tutorial/collections/streams/index.html.

laws, are different than the loops *within* `XExpressions`. Although Blang does not currently have built-in data types for sampling of infinite dimensional objects, they can be handled by creating dedicated types and/or using `XExpression` loops *inside* a `logf` block.

7.9. Generate block

The generate block is responsible for the forward generating mechanism of a model. This is optional in that it is only required when more sophisticated inference algorithms are desired, as discussed in Section 5.6. An important distinction from `laws` blocks is that `generate` blocks are imperative. Furthermore, they are not referentially transparent as random variables will be modified in-place. We formalize the syntax used to encode the generate block introduced conceptually in Section 5.6:

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

The argument `myRandomSeed` is the name of an input object of type `java.util.Random` (the type declaration for this input is skipped since this is the only possible type allowed). To connect this syntax with its interpretation described in Section 5.6, the input argument can be thought as an outcome $\omega \in \Omega$, from which the `XExpression` should form the realization $X(\omega)$.

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.

7.10. 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 `Sampler`.[18]

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:

[18] `https://www.stat.ubc.ca/~bouchard/blang/javadoc-sdk/blang/mcmc/Sampler.html`

```
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 `Constrained`.[19] Effectively, the intended behaviour is to disable samplers which would be inoperative with the alternate choice of reference measure.

To illustrate this necessity, consider a K-dimensional Dirichlet distributed random variable (i.e., $p = (p_1, p_2, \ldots, p_K)$). By default, Blang would automatically designate slice samplers for each of the coordinates $p_1, p_2, \ldots, p_K$, as they are of type `RealVar` variables (these variables are discussed in the following section). However, because of the simplex constraint requiring $\sum_{i=1}^{K} p_i = 1$, this would lead to proposal rejection almost surely. The keyword `Constrained` is used to prevent this automatic assignment of ineffective or incorrect samplers.

7.11. XExpressions

Syntax for XExpressions is provided by the Xtext language engineering framework. XExpressions are imperative expressions. Thus the `logf`, `generate`, and variable initialization blocks for example are imperative, while `laws` blocks are declarative.

Here we highlight key aspects commonly used in Blang programs. We refer the reader to the Xtext documentation for more information.[20]

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, with the last one providing the return value, 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`.[21] Object references can

[19] https://www.stat.ubc.ca/~bouchard/blang/javadoc-dsl/blang/core/Constrained.html

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

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

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.

Literals

Examples of expressions that create constants of type…

- `boolean`: `true`, `false`
- `int`: `42`, `12000`
- `double`: `1.0`, `1.3e2`, making sure to include the decimal suffix or to use scientific notation.
- String literals: either via `"A"`,
  or `'''This version allows "quotes inside" and more'''`
- type literals: `MyType`, which is equivalent to Java's `MyType.class`.
- `List`: `#[true,false]` (note the hash symbol `#` is not a comment as in other languages, it is used to construct lists, sets, and maps)
- `Set`: `#{"A","C","G","T"}`
- `Map`: `#{"key1" -> 1, "key2" -> 2}`
- `Pair`: `"likelihood" -> 1.43` (this example returns type `Pair<String, Double>`; this syntax can be used with arbitrary key and value types).

Declaring variables with XExpressions

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

```
var int 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.

In the above, `typeInferred` illustrates that the type can be inferred automatically, in this example a `List<Integer>`.

Conditionals

Conditional expressions have the following form:

```
val String variable = if (condition) value1 else value2
```

Conditional expressions return values depending on a condition, where `condition` evaluates to a boolean. When `(condition)` is `true`, `value1` is returned, otherwise `value2` is returned. The shorthand notation without `else`

```
if (condition) value
```

or

```
if (condition) {
   (value)
}
```

is equivalent to `if (condition) value else null`.

Scope

The *scope* of a variable is defined as the portion of code in which the variable 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 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 have priority.

The only exception is the arguments of the atomic and composite laws. Recall our example in Section 7.8.1 (repeated below),

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

These laws require explicit declaration of the variables to include in the scope, where these variables should be identified at the right of the `|` symbol. This design choice is primarily motivated by its flexibility in handling complex dependencies, to be demonstrated in Sections 11.1 and 11.2.

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 exceptions are composite laws, where the evaluation of an argument is delayed at initialization and instead repeated each time the density is evaluated during sampling (i.e., a form of "lazy evaluation"). To see why this is needed, consider a factor declaration of the form `y | x ~ Normal(2 * x, 1)`. Each time this factor is computed during inference, we would like the mean parameter `2 * x` to be recomputed. One way to think about lazy evaluation in this context is that when the factor graph is created, `2 * x` is converted into a lambda expression which is computed each time we are computing the value of the normal factor.

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`:

`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 in the 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 readable 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))
}
```

This can be equivalently written, more explicitly, as

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

The underpinning of this code is that since Blang automatically imports all functions in `Generators` as extension methods,[22] 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`).[23]

[22] https://www.stat.ubc.ca/~bouchard/blang/javadoc-sdk/blang/distributions/Generators.html
[23] https://www.stat.ubc.ca/~bouchard/blang/javadoc-xlinear/xlinear/MatrixOperations.html

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:[24]

```
val it = myObject
doSomething
```

This is merely used for shorthand notation for:

```
myObject.doSomething
```

and is used in lambda expressions which we discuss next.

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 a function to each item in a list). Since they are so useful, many syntactic shortcuts are available.

The explicit syntax for lambda expressions is:

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

For example, to capitalize words in a list, we can use the function `map(myFunction)` which applies `myFunction` to every entry in the list. Here `map()` is the function that takes in another function `myFunction` as an argument. More concretely, we have:

```
#["foo", "bar"].map([String s | s.toUpperCase])
```

[24]https://www.eclipse.org/xtend/documentation/203_xtend_expressions.html

When there is a single input argument in the lambda expression (i.e., in the above case `String s`), 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:

```
#["one", "two"].map([it.toUpperCase])
```

which further simplifies to:

```
#["one", "two"].map([toUpperCase])
```

Finally, when the last argument of a function (`map()` in this case) is a function, you can simply put the lambda after the parentheses of the function call (`map()`). For example:

```
#["one", "two"].map()[toUpperCase]
```

Which further simplifies to:

```
#["one", "two"].map[toUpperCase]
```

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 `IntVar` and `RealVar`,[25] which are mutable versions of `Integer` or `Double`. See Appendix B.3 for a discussion on why these mutable data structures are necessary in Blang.

Operator overloading

Operator overloading is permitted. When in the Blang IDE, command click on an operator to reveal its definition. 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).

Some useful operators that are automatically imported:

[25] https://www.stat.ubc.ca/~bouchard/blang/javadoc-dsl/blang/core/IntVar.html, and https://www.stat.ubc.ca/~bouchard/blang/javadoc-dsl/blang/core/RealVar.html

- "`0..10`", and "`0..<11`": These 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 `invalidParameter`.[26] 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 an exception 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
}
```

8. Cheatsheet interlude

Learning a language can be a time-consuming task with new grammar and syntax to remember. Before continuing with more examples, ideas, and patterns, we present a condensed summary of the concepts covered thus far in the form of a recipe. We will draw connections to

[26] `https://www.stat.ubc.ca/~bouchard/blang/javadoc-sdk/blang/types/StaticUtils.html`

the GMM example from Section 6 where appropriate. For convenience, we repeat the model below:

$$\begin{aligned}
&\text{concentration} & \alpha &= [1,1]\\
&\text{proportions} & \pi \mid \alpha &\sim \text{Dirichlet}(\alpha)\\
&\text{labels} & z_i \mid \pi &\sim \text{Categorical}(\pi)\\
&\text{means} & \mu_k &\sim \text{Normal}(0,10^2)\\
&\text{standard deviations} & \sigma_k &\sim \text{Uniform}(0,10)\\
&\text{observations} & y_i \mid \mu,\sigma,z_i &\sim \text{Normal}(\mu_{z_i},\sigma^2_{z_i})
\end{aligned}$$

for $i \in \{1,2,\ldots,n\}$ and $k \in \{1,2\}$.

The cheatsheet

1. Write down your package statement.
   Example:
   ```
   package jss.gmm
   ```

2. If it is already known which packages you will work with, then import them. We did not require additional packages for the GMM.
   Example:
   ```
   import some.other.pack
   ```

3. Name your model.
   Example:
   ```
   model MixtureModel { ... }
   ```

4. Identify all model variables: observed (constant) random variables (RV), latent RVs, unknown parameters, and known (constant) parameters.
   Example: observed RVs y_i, latent RVs z_i, unknown parameters π, μ_k, σ_k, and known parameter α.

5. Identify model variables' types, and values of known parameters.
   Example: y_i are real numbers, z_i are integers, π is a simplex, μ_k are real numbers, σ_k are real numbers, and $\alpha = [1,1]$.

6. Declare all observed and latent RVs, and unknown parameters with the keyword `random`; declare all known parameters with the keyword `param`.
   Example:
   ```
   random List<RealVar> observations
   random Simplex pi
   param Matrix concentrations
   ```

7. Initialize latent RVs and unknown parameters with their latent types, and initialize known parameter values. Realization of observations will be delayed until inference.
   Example:

```
random List<RealVar> observations
random Simplex pi ?: latentSimplex(2)
param Matrix concentrations ?: fixedVector(1.0, 1.0)
```

8. Next we declare their respective distributions in `laws{ ... }`.
   Example:

   ```
   pi | concentration ~ Dirichlet(concentration)
   ```

9. Use `for` loops to declare over a list of variables for cleaner code.
   Example:

   ```
   for (int k : 0 ..< means.size) {
     means.get(k) ~ Normal(0.0, pow(10.0, 2.0))
   }
   ```

10. Perform inference by using the CLI. Append `--help` to the CLI for model-specific input description.
    Example:

    ```
    blang --model jss.gmm.MixtureModel \
          --model.observations file "path/to/line_separated_values.txt"
    ```

9. Custom samplers for custom data structures

Although the focus of this section is on custom data structures and samplers, it will also provide insight to Blang's underlying sampling mechanisms.

In examples we have demonstrated thus far, sampling variables could be handled via default samplers. This luxury is typically unavailable when working with complex state spaces such as trees, partitions, permutation spaces, or more generally discrete, non-ordinal spaces. In such situations Blang still assists the user in several ways described in more detail in Section 10. Here we focus on how Blang helps implement a complete sampler for a model that consists of such custom data structures.

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 variable. 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.

We will implement a data type, `Permutation`, equipped with its tailor-made sampler, then apply it in the context of a model.[27] We begin by implementing a *class* describing how permutations will be encoded. Note a permutation can be represented with or stored as a list of integers. We present an implementation of the `Permutation` class below, and discuss each component individually.

[27] Complete and commented implementations in this section are available in the reproduction materials located in the directory `reproduction_materials/example`, or at `https://github.com/UBC-Stat-ML/JSSBlangCode/tree/master/PermutationExample/src`.

Permutation.xtend[28]

```
package jss.perm

import org.eclipse.xtend.lib.annotations.Data
import blang.mcmc.Samplers
import java.util.Random
import static java.util.Collections.sort
import static java.util.Collections.shuffle
import java.util.List

@Samplers(PermutationSampler)
@Data class Permutation {

  val List<Integer> connections

  new (int componentSize) {
    connections = (0 ..< componentSize).toList
  }

  def int componentSize() {
    return connections.size
  }

  def void sampleUniform(Random random) {
    sort(connections)
    shuffle(connections, random)
  }

  override String toString() {
    return connections.toString
  }
}
```

A first observation is the annotation `@Samplers(...)` which informs the runtime engine to sample `Permutation` objects with an instance of `PermutationSampler`.[29] We will discuss `PermutationSampler` later.

A second observation is the annotation `@Data`.[30] Briefly, this annotation should be interpreted as a *data class*, a terminology in object oriented programming (and an unfortunate clash with the conventional use of "data" in statistics), meaning the class can only declare final fields, and that `.equals`, `.hashcode` are automatically implemented in addition to other defaults.

Moving on to the main code block, as noted previously, we can represent the mathematical permutation object with a list of integers, where each element in the list is the permuted value of the element's index. Thus we encoded the permutation object with the field `connections`, and a constructor as repeated below:

[28] https://github.com/UBC-Stat-ML/JSSBlangCode/blob/master/PermutationExample/src/main/java/jss/perm/Permutation.xtend

[29] More than one sampler can be specified as a comma-separated list, more on this in Section 12.7.3.

[30] Complete documentation available at http://archive.eclipse.org/modeling/tmf/xtext/javadoc/2.9/org/eclipse/xtend/lib/Data.html.

```
val List<Integer> connections

new (int componentSize) {
  connections = (0 ..< componentSize).toList
}
```

This permutation constructor is to related to Section 6's `latentSimplex(K)` constructor for simplex variables. We will see its use later to construct a latent permutation to be sampled. Technically, this is all that is required to represent the permutation type. However, it will be convenient to define a few more helper functions, in particular a function `sampleUniform` to uniformly draw a realization of a permutation.

```
def void sampleUniform(Random random) {
  sort(connections)
  shuffle(connections, random)
}
```

Notice `sampleUniform` sorts connections, then shuffles connections in-place. The sorting is required from a computational perspective to ensure the sampling is not affected by the `connection`'s current state, thus uniform when shuffled. In other words, it enforces the contract that for a given random seed encoded in the `rand` object, the behaviour of the `generate` block is fully deterministic and not affected by the current state of the object. This behaviour is exploited to design test cases, as in `TestCompositeModel.xtend` described shortly. The sampling performed in-place is a technical requirement for the inference engine, detailed in Section 12.

Finally the last piece of the puzzle, the `toString` function. Its purpose is best illustrated by an example followed by an explanation. Here is what our sample output would read without overriding `toString`:

permutation.csv

```
sample,value
0,"Permutation [
  connections = ArrayList (
    2,
    0,
    1
  )
]"
1,"Permutation [
  connections = ArrayList (
    1,
    2,
    0
  )
]"
...
```

With the toString function, we have:

`permutation.csv`

```
sample,value
0,"[2, 0, 1]"
1,"[1, 2, 0]"
...
```

Thus we see that by overriding the default string output of our object, we enable the engine to output something more legible. One can customize this output to respect the Tidy philosophy, details of which we leave to Appendix D.0.2.

With all the pieces in place for our `Permutation` class, we are now ready to discuss samplers.

To perform posterior inference on permutation spaces, we need an invariant sampler designed specifically for the object `Permutation`. In this example, we assume familiarity with the Metropolis algorithm (Metropolis *et al.* 1953), and begin by presenting the full code below, followed by a breakdown of its main components:

`PermutationSampler.xtend`[31]

```
package jss.perm

import java.util.List
import bayonet.distributions.Random
import blang.core.LogScaleFactor
import blang.mcmc.ConnectedFactor
import blang.mcmc.SampledVariable
import blang.mcmc.Sampler
import blang.distributions.Generators
import static java.lang.Math.exp
import static java.lang.Math.min
import static extension java.util.Collections.swap

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

[31] https://github.com/UBC-Stat-ML/JSSBlangCode/blob/master/PermutationExample/src/main/java/jss/perm/PermutationSampler.xtend

```
  override void execute(Random rand) {
    val n = permutation.componentSize
    val i = Generators.discreteUniform(rand, 0, n)
    val j = Generators.discreteUniform(rand, 0, n)

    val currentLogDensity = logDensity()
    permutation.connections.swap(i,j)
    val newLogDensity = logDensity()

    val acceptProb = min(1.0, exp(newLogDensity - currentLogDensity))
    val accept = Generators.bernoulli(rand, acceptProb)
    if (!accept) {
      permutation.connections.swap(i, j)
    }
  }

  def double logDensity() {
    var double sum=0.0
    for (LogScaleFactor f : numericFactors) sum += f.logDensity()
    return sum
  }
}
```

A first observation is the *implementation* of the `Sampler` *interface*, and the two annotations `@SampledVariable` and `@ConnectedFactor`:[32]

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

Briefly, this implies the `PermutationSampler` class necessarily implements methods specified in the interface `Sampler`, namely `execute`. The `execute` method is invoked with each iteration of the inference algorithm (Section 12.1), and updates our variable of interest in-place. The field annotated with `@SampledVariable` will automatically be populated with an instance of the object to be sampled, in this example, an instance of `Permutation`. This annotation in tandem with `@Samplers` enables the linkage of variables and samplers.[33] Similarly, the field annotated with `@ConnectedFactor` will automatically be populated with factors dependent on the sampled object, which is inferred automatically via a factor graph built from scope analysis (described in detail in Section 12). By default once a type and its sampler have been implemented, variables of such type will be sampled with this sampler. We discussed how this default is altered in Section 7.10, and from another perspective in Section 12.7.3.

With this setup, we are ready to implement the Metropolis algorithm for permutations:

[32]For more on interfaces, see https://docs.oracle.com/javase/tutorial/java/concepts/interface.html.

[33]Sampling of multiple variables can also be performed. For example, the SDK incorporates an elliptic slice algorithm which samples many real variables at once, see https://github.com/UBC-Stat-ML/blangSDK/blob/master/src/main/java/blang/distributions/NormalField.bl and https://github.com/UBC-Stat-ML/blangSDK/blob/master/src/main/java/blang/mcmc/EllipticalSliceSampler.xtend

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

  val currentLogDensity = logDensity()
  permutation.connections.swap(i,j)
  val newLogDensity = logDensity()

  val acceptProb = min(1.0, exp(newLogDensity - currentLogDensity))
  val accept = Generators.bernoulli(rand, acceptProb)
  if (!accept) {
    permutation.connections.swap(i, j)
  }
}

def double logDensity() {
  var double sum=0.0
  for (LogScaleFactor f : numericFactors) sum += f.logDensity()
  return sum
  }
}
```

The implementation of `execute()` shown above is a standard Metropolis algorithm (Metropolis *et al.* 1953), which invokes `logDensity` when density evaluation is required. Since the field `numericFactors` is a list of all log factors dependent on our variable, the `logDensity` method merely returns the sum of log factors.

Notice the syntax `Generators.bernoulli(rand, acceptProb)` is used to determine acceptance of the proposal. This syntax equivalent to `Bernoulli.distribution(p).sample(rand)` (in fact the latter calls the former). However the second variant creates an intermediate object of type `IntDistribution` which could be a performance issue as the body of the sampling algorithm is in the inner loop of inference. In other contexts, having this intermediate object is useful, e.g., if one would like to provide a distribution as input parameter to another distribution, as in Section 11.4. As for the relationship with `... ~ Bernoulli(...)`, recall that the difference is that `... ~ Bernoulli(...)` is used in a declarative context, while the first two syntaxes are for imperative blocks such as MCMC samplers (of course, in all three variants there is no code duplications in the SDK, i.e., higher-level functions such as the declarative syntax call lower level implementations).

With our custom `Permutation` type and `PermutationSampler` in place, we are ready to apply them in models.

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

`UniformPermutation.bl`[34]

```
model UniformPermutation {
  random Permutation permutation
```

[34]https://github.com/UBC-Stat-ML/JSSBlangCode/blob/master/PermutationExample/src/main/java/jss/perm/UniformPermutation.bl

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

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

As we have seen before, the `logf` block provides a method for evaluating log densities, while the `generate` block provides a method for sampling permutations in place. In this case, `logf` returns the log density of a permutation with uniform distribution, and `generate` samples a permutation uniformly. As for `logFactorial`, which computes $\log(n!)$, it is part of the automatically imported functions described in Section 7.6 (a list of the most commonly used automatically imported functions can also be found in Appendix F).

As with any distribution, model `UniformPermutation` can be used in composition with other models. Here we present a minimal, illustrative example:

`CompositeModel.bl`[35]

```
package jss.perm

model CompositeModel {
  random List<RealVar> y ?: fixedRealList(2.1,-0.3,0.8)
  random Permutation permutation ?: new Permutation(y.size)

  laws {
    permutation ~ UniformPermutation
    for (int i : 0 ..< y.size) {
      y.get(i) | permutation, i
        ~ Normal(permutation.getConnections.get(i), 0.3)
    }
  }
}
```

This should look rather similar to other models, with the exception of the use of a custom constructor `new Permutation(y.size)` to instantiate the latent permutation variable. `CompositeModel` provides a toy example of how one can incorporate `UniformPermutation` into larger models. An example of custom data types with emphasis on a practical application using a spike and slab model (Mitchell and Beauchamp 1988) is presented in Appendix A.2.

This concludes our tutorial on creating custom data types and samplers. We dedicate the remainder of this section to showcasing some available resources that assist users in testing the correctness of samplers.

[35]https://github.com/UBC-Stat-ML/JSSBlangCode/blob/master/PermutationExample/src/main/java/jss/perm/CompositeModel.bl

A first test utility provided by the SDK is `DiscreteMCTest`, which is specialized to fully-discrete spaces. The idea behind `DiscreteMCTest` is that, for small discrete 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 problems just large enough to achieve code coverage.

To run tests, we need to setup a project directory, i.e., `create-blang-gradle-project --name PermutationExample`.[36] This will create a directory named "PermutationExample", with directory structure `src/main/java`. Place our implementations in this directory, and create additional directories `PermutationExample/src/test/java/`. Making sure the package names are matching, place `TestCompositeModel.xtend` in the testing directory `src/test/java/jss/perm/` with implementations as follows:

`TestCompositeModel.xtend`[37]

```
package jss.perm

import static blang.types.StaticUtils.*
import blang.runtime.SampledModel
import blang.validation.DiscreteMCTest
import com.rits.cloning.Cloner
import org.junit.Test
import static org.apache.commons.math3.util.CombinatoricsUtils.factorial
import static java.lang.Math.pow
import blang.runtime.internals.objectgraph.GraphAnalysis
import blang.runtime.Observations
import blang.types.ExtensionUtils

class TestCompositeModel {

  val static y = fixedRealList(2.1, -0.3, 0.8)
  val static CompositeModel compositeModel = new CompositeModel.Builder()
    .setY(y)
    .setPermutation(new Permutation(y.size))
    .build

  val static observations = {
    val Observations result = new Observations
    result.markAsObserved(y)
    result
  }

  val static DiscreteMCTest test =
    new DiscreteMCTest(
      new SampledModel(new GraphAnalysis(compositeModel, observations)),
      [
        val CompositeModel cm = model as CompositeModel
        return new Cloner().deepClone(cm.permutation)
      ]
    )
```

[36] Commented implementations on testing are available in the reproduction materials located in the directory `PermutationExample`.

[37] https://github.com/UBC-Stat-ML/JSSBlangCode/blob/master/PermutationExample/src/test/java/TestCompositeModel.xtend

```
  @Test
  def void stateSize() {
    test.verbose = true
    test.checkStateSpaceSize(factorial(y.size) as int)
  }

  @Test
  def void invariance() {
    test.verbose = true
    test.checkInvariance
  }

  @Test
  def void irreducibility() {
    test.verbose = true
    test.checkIrreducibility
  }
}
```

The test can be performed using `./gradlew test` while in the `PermutationExample` directory, or via the Eclipse IDE. When creating a `DiscreteMCTest` object, one should pass in the small discrete model to be tested as the first argument. The second argument is a lambda function (denoted by square brackets) that accepts a model and creates a new object encoding the identity of the current configuration, with identity being mediated by the `.equals()` function of the returned object. 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.[38] 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.

Detailed testing resources are discussed in Section 10.5, in particular to handle continuous state spaces.

10. 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.

[38]More information is available at `https://www.stat.ubc.ca/~bouchard/blang/Testing_Blang_models.html` under "Exhaustive tests."

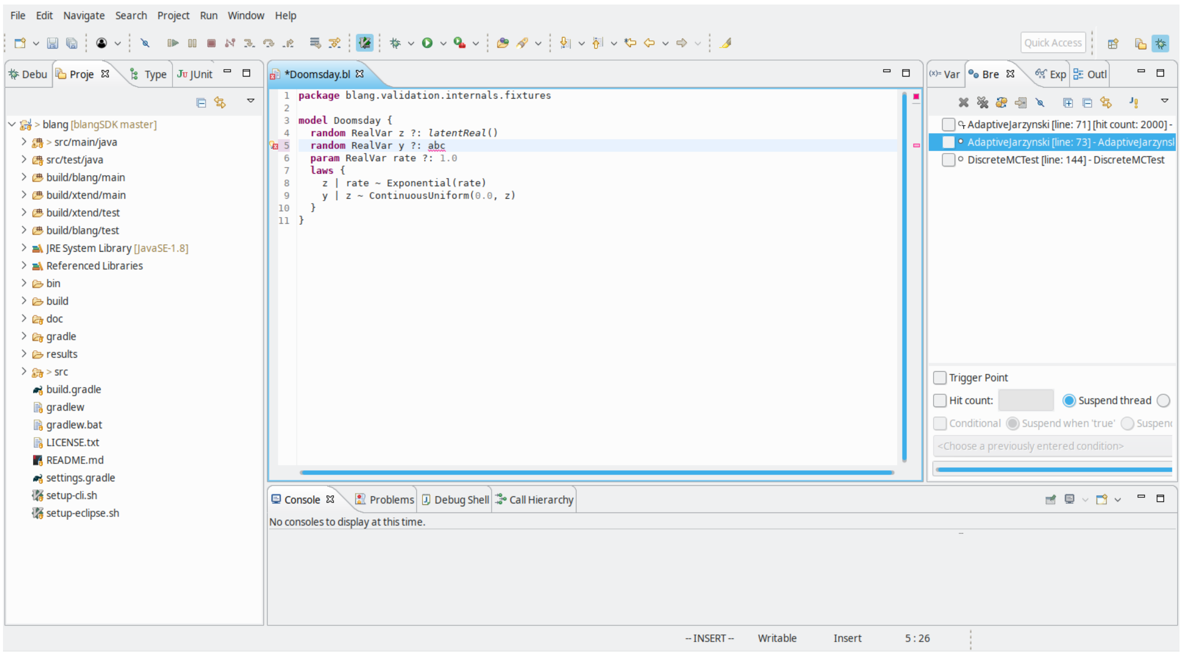
10.1. Integrated development environment (IDE)

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.[39]
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] - AdaptiveJarzynsl
DiscreteMCTest [line: 144] - DiscreteMCTest

No consoles to display at this time.

Figure 6: 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.

[39]`https://www.eclipse.org/downloads/packages/`

- 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 types. See example in Figure 6.
- 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..". 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`.

To setup a Blang project, use the command-line interface `create-blang-gradle-project` and follow its instructions. This will create a directory suitable for development using the IDE, and is the recommended option for working in Blang.

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

10.2. Data types provided in the SDK

The interfaces `RealVar` and `IntVar` are automatically imported.[40] 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** (for more information see Appendix C.3) 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

[40]`https://www.stat.ubc.ca/~bouchard/blang/javadoc-dsl/blang/core/IntVar.html`, and `https://www.stat.ubc.ca/~bouchard/blang/javadoc-dsl/blang/core/RealVar.html`

$1 \times n$ or $n \times 1$ matrices. Standard operations are supported using unsurprising syntax, e.g., `identity(100_000)` (underscore delimited 100,000), `ones(3,3)`, `matrix.norm`, `matrix.sum`, `matrix.readOnlyView`, `matrix.slice(1, 3, 0, 2)`, `matrix.cholesky`, etc.[41]

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.

10.3. Distributions

A range of distributions are included in the SDK. See Appendix E 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 `Generators`.[42]

10.4. 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:

`Name.bl`

```
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`, 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`.

`Type1.xtend`

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

[41] See `https://github.com/UBC-Stat-ML/xlinear` for more information on **xlinear**.

[42] `https://www.stat.ubc.ca/~bouchard/blang/javadoc-sdk/blang/distributions/Generators.html`

```
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 pages.[43]

Plate notation

Simple collections of random variables can be handled via Java's built-in `List` and related data structures. However, this can quickly become cumbersome and error-prone when working with sophisticated hierarchical Bayesian models. To address this problem, Blang provides a specialized data structure based on the *plate notation*, a method for representing repeated random variables in a graphical model. A concise encoding of these models can be achieved with built-in types `Plate` and `Plated`.

Consider the following rocket launching data in Tidy format:

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

Where "Country" is the origin of the rocket, "Rocket" is the name of rockets, "nLaunches" is the number of launches, and "nFails" is the number of failed launches. A toy model for this data set is the hierarchical model in Figure 7.

[43] https://www.stat.ubc.ca/~bouchard/blang/Javadoc.html.

$$
\begin{aligned}
\alpha_c &\sim \text{Gamma}(1,1) \\
\beta_c &\sim \text{Gamma}(1,1) \\
p_{r,c} \mid \alpha_c, \beta_c &\sim \text{Beta}(\alpha_c, \beta_c) \\
f_{r,c} \mid p_{r,c}, l_{r,c} &\sim \text{Binomial}(l_{r,c}, p_{r,c})
\end{aligned}
$$

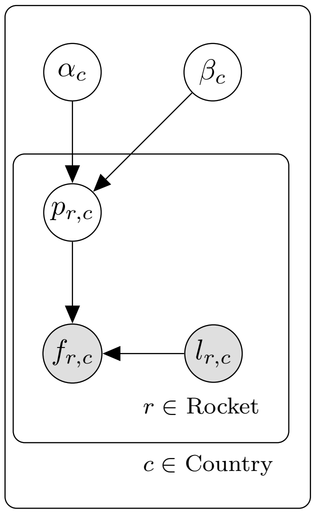


Figure 7: Left: a toy hierarchical model for the rocket launching data set. Indices c, r index countries and rockets respectively. Observations $f_{r,c}, l_{r,c}$ are the number of failures and launches for rocket r in country c respectively. Latent variables $p_{r,c}, \alpha_c, \beta_c$ are parameters of interest. Right: A graphical representation of the hierarchical rocket model with plate notation.

Its corresponding Blang encoding is as follows:[44]

Rocket.bl [45]

```
package jss.hier

model Rocket {

  param GlobalDataSource data

  param Plate<String> countries
  param Plate<String> rockets

  random Plated<RealVar> alpha
  random Plated<RealVar> beta
  random Plated<RealVar> prob
  random Plated<IntVar> nFails
  param Plated<Integer> nLaunches
```

[44]Complete code and data can be found in the prepackaged repository of examples.
[45]https://github.com/UBC-Stat-ML/JSSBlangCode/blob/master/example/jss/hier/Rocket.bl

```
  laws {
    for(Index<String> c : countries.indices()){
      alpha.get(c) ~ Gamma(1,1)
      beta.get(c) ~ Gamma(1,1)

      for(Index<String> r : rockets.indices(c)){
        prob.get(r, c) |
          RealVar a = alpha.get(c),
          RealVar b = beta.get(c)
          ~ Beta(a, b)
        nFails.get(c, r) |
          RealVar p = prob.get(r, c),
          Integer n = nLaunches.get(r, c)
          ~ Binomial(n, p)
      }
    }
  }
}
```

A first observation is the additional `param GlobalDataSource data`, which we have not seen in our previous models. We will discuss its function in more details shortly, at a high level, it is used to specify a CSV file from which many variables will be parsed.

A second observation is the use of `Plate<...>` and `Plated<...>` types.

```
param Plate<String> countries
param Plate<String> rockets

random Plated<RealVar> alpha
random Plated<RealVar> beta
random Plated<RealVar> prob
random Plated<IntVar> nFails
param Plated<IntVar> nLaunches
```

A `Plate` is a collection of indices, such as Country and Rocket indices (columns one and two of our example data set above). As they are non-random known indices, we declare plates with `param`. On the other hand `Plated` types, as its name suggests, are variables within plates. The usual rules for selection between `param` or `random` apply to plated variables (see Section 5.1).

With this setup, we are ready to examine the laws block.

```
for(Index<String> c : countries.indices()){
  alpha.get(c) ~ Gamma(1,1)
  beta.get(c) ~ Gamma(1,1)

  for(Index<String> r : rockets.indices(c)){
    prob.get(r, c) |
      RealVar a = alpha.get(c),
      RealVar b = beta.get(c)
      ~ Beta(a, b)
    nFails.get(c, r) |
      RealVar p = prob.get(r, c),
      Integer n = nLaunches.get(r, c)
      ~ Binomial(n, p)
  }
}
```

This should look rather similar to code we have presented thus far. We highlight the key differences: first, the set of index values is obtained by appending `.indices` to a `Plate` variable. Each index is of type `Index<T>`, where `T` is the same type as the corresponding `Plate<T>`. Plated variables can subsequently be retrieved by using `.get()`. Second, notice the syntax of the second `for` loop over rocket indices, in particular `rockets.indices(c)`. This syntax retrieves the set of rocket indices such that its country index is `c`. Lastly, we note the ordering of indices within `.get()` is exchangeable, for example, `nFails.get(c, r)` is equivalent to `nFails.get(r, c)`. This is possible since `Index<...>` objects keep track of which plate they belong to.

Additional methods available for types `Index<>` are described in Appendix 18.

With variables, parameters, and laws declared, we tie these concepts back to the promised discussion of `param GlobalDataSource data`. Its purpose becomes clear when we invoke `blang` and its corresponding arguments:

```
> 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 Rocket \
```

Each variable of type `Plate` and `Plated` will be put in correspondence with a column in a Tidy CSV file. Command-line arguments can be used to set the CSV file for *each* variable individually. *Alternatively* by declaring a dummy variable of type `GlobalDataSource`, here called `data`, we can set a default CSV file that will be used by default by all `Plate` and `Plated` variables. In our example, specifying the default CSV is achieved via `--model.data pathToData/data.csv`. For each `Plate` and `Plated` variable, the data input algorithm will match column names with variable names, but this can be overridden via CLI arguments. In our example, this is achieved via e.g., `--model.rockets.name Rocket`. Notice we did not require this argument for `nFails`, as the column name in the CSV file is also `nFails`.

When a plated variable is not found in the CSV file, it is assumed to be latent. Should a plate not correspond to a column in the CSV, then its `maxSize` should be set via for e.g., `--model.varName.maxSize 3`, or initialized in the model. An example using the CLI is presented in the advanced tutorial in Appendix A.2,[46] and an example using default initializations is shown below.

Recall the Gaussian mixture model from Section 6. We can implement the same model with plate syntax:

[46]Use the argument `--model Rocket --help` for full documentation.

MixtureModelPlated.bl[47]

```
package jss.gmm

model MixtureModelPlated {

  param GlobalDataSource data

  param  Integer        K  ?: 2
  param  Plate<Integer> N
  param  Plate<Integer> components ?: Plate.ofIntegers("components", K)

  random Plated<IntVar>  z
  random Plated<RealVar> y
  random Plated<RealVar> mu
  random Plated<RealVar> sd
  random Simplex         pi ?: latentSimplex(K)

  laws {
    pi | K ~ SymmetricDirichlet(K, 1.0)

    for (Index<Integer> k : components.indices) {
      mu.get(k) ~ Normal(0.0, 100.0)
      sd.get(k) ~ ContinuousUniform(0.0 ,10.0)
    }

    for (Index<Integer> i : N.indices) {
      z.get(i) | pi ~ Categorical(pi)
      y.get(i) | List<RealVar> muList = mu.asList(components),
                 List<RealVar> sdList = sd.asList(components),
                 IntVar k = z.get(i)
        ~ Normal(muList.get(k) , pow(sdList.get(k), 2.0))
    }
  }
}
```

A first observation is the use of `Plate.ofIntegers()` to initialize a plate with a predetermined size. The function `ofIntegers()` takes in for arguments a column name, and a maximum size. For other related functions, see Appendix 18.

A second observation is the `asList()` function, which returns the given plate (`components`) as a list.

```
y.get(i) | List<RealVar> muList = mu.asList(components),
           List<RealVar> sdList = sd.asList(components),
           IntVar k = z.get(i)
  ~ Normal(muList.get(k) , pow(sdList.get(k), 2.0))
```

Similar conversion utilities automatically imported from `ExtensionUtils` are documented in Appendix 17.[48] With this setup, our command line arguments for inference can be written

[47]https://github.com/UBC-Stat-ML/JSSBlangCode/blob/master/example/jss/gmm/MixtureModelPlated.bl

[48]https://www.stat.ubc.ca/~bouchard/blang/javadoc-sdk/blang/types/ExtensionUtils.html.

succinctly:

```
> git clone https://github.com/UBC-Stat-ML/JSSBlangCode.git
> cd JSSBlangCode/example/
> blang --model jss.gmm.MixtureModelPlated \
    --model.data data/obs1Plated.csv
```

PlatedMatrix

One special case of a `Plated` variable is the type `PlatedMatrix`,[49] which is a built-in type that facilitates easy representation of higher dimensional random variables such as random vectors or matrices, as well as lists and arrays of vectors and 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 $0, 1, 2, \ldots$, or, if they are of other types, say strings or non-consecutive integers, in which case a fixed correspondence with $0, 1, 2, \ldots$ 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:

`PlatedMatrixExample.bl`

```
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)
    }
  }
}
```

10.5. Testing framework

There is considerable emphasis in the MCMC literature on efficiency, but much less on correctness, in the sense of the implementation being ergodic with respect to the distribution of

[49] https://www.stat.ubc.ca/~bouchard/blang/javadoc-sdk/blang/types/PlatedMatrix.html

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 unbiasedness of the normalization constant estimate provided by our SMC implementation. The code forming the basis of this test is shown below:

UnbiasednessTest.xtend

```
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 `DiscreteMCTest`,[50] we provide algorithms that use `ExhaustiveRandom` to check via linear

[50] https://www.stat.ubc.ca/~bouchard/blang/javadoc-sdk/blang/validation/DiscreteMCTest.html

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 `TestDiscreteModels` for example of usage.[51]

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:

`TestMyModel.xtend`

```
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 9) is provided in the reproduction materials.

10.6. Package distribution and injection

[51]https://github.com/UBC-Stat-ML/blangSDK/blob/master/src/test/java/blang/TestDiscreteModels.xtend.

Distributing and reusing packages is standard practice in software development. Any user can create a model and publish it in a versioned fashion via GitHub.[52]

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:

```
dependencies.txt
```

```
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.

To distribute packages, users can create a Blang project with the command-line interface `create-blang-gradle-project`, and publish it in a GitHub repository.

11. 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.

11.1. 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 an MRF, where I denotes a set of cliques that factorizes the 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 MRF. In this case we use a product of independent Bernoulli random variables as a pseudo-prior. Note, we will make use of an "empty pipe symbol", i.e. "`| IntVar first = ...`" which is explained after the example:

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

Ising.bl[53]

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

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

Rather than 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 (shown below for reference). 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. A second observation worthy of note is the conditioning of `| IntVar first = vertices.get(pair.getFirst)` as opposed to `| vertices`. This prevents the runtime architecture from assuming that these factors depend on the full `vertices` object, hence improving computational efficiency by a scaling proportional to the size of `vertices`. We emphasize this computational advantage in Section 11.2 with a Markov chain example.

LogPotential.bl[54]

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

[53]https://github.com/UBC-Stat-ML/JSSBlangCode/blob/master/example/jss/others/IsingExample.bl

[54]https://github.com/UBC-Stat-ML/JSSBlangCode/blob/master/example/jss/others/LogPotentialExample.bl

11.2. 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:

`MarkovChain.bl`[55]

```
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 on the smallest possible scope. 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 12.7.

11.3. 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

[55] https://github.com/UBC-Stat-ML/JSSBlangCode/blob/master/example/jss/others/MarkovChainExample.bl

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:

Exponential.bl[56]

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

11.4. 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:

IntMixture.bl[57]

```
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)
```

[56]https://github.com/UBC-Stat-ML/JSSBlangCode/blob/master/example/jss/others/ExponentialExample.bl

[57]https://github.com/UBC-Stat-ML/JSSBlangCode/blob/master/example/jss/others/IntMixtureExample.bl

```
    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:

`PoissonPoissonMixtureExample.bl`[58]

```
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. The `distribution(...)` function returns a simplified application programming interface (API) for models having only one random variable. If that single random variable is of type `RealVar` (respectively, `IntVar`), the returned value of `distribution(...)` is of type `RealDistribution`[59] (respectively, `IntDistribution`[60]). If the type of the single random variable is neither `RealVar` nor `IntVar`, the returned value of `distribution(...)` is of the type `Distribution`.[61]

12. 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.

12.1. Inference algorithms

Blang currently focuses on two complementary inference algorithms: sequential change of measure (SCM), and non-reversible parallel tempering (PT). SCM infers the exact poste-

[58]https://github.com/UBC-Stat-ML/JSSBlangCode/blob/master/example/jss/others/PoissonPoissonMixtureExample.bl
[59]https://www.stat.ubc.ca/~bouchard/blang/javadoc-dsl/blang/core/RealDistribution.html
[60]https://www.stat.ubc.ca/~bouchard/blang/javadoc-dsl/blang/core/IntDistribution.html
[61]https://www.stat.ubc.ca/~bouchard/blang/javadoc-dsl/blang/core/Distribution.html

rior distribution asymptotically in memory, while PT infers the exact posterior distribution asymptotically in time. 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 interpolating from the prior to the posterior distribution. 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 (Neal 2003),[62] 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 good candidates for automatic inference on generalized state spaces. These two algorithms can be used individually, but by default the SCM is used to initialize PT. This combination is motivated by the fact that SCM appears to often be better suited to quickly find a crude approximation. In particular SCM is able to find configurations of positive probability even in the presence of deterministic constraints (i.e., configurations having zero posterior probability). However, to obtain high quality samples, 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.

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.

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

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,

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

where $\epsilon_t = \exp(-10^{100}t)\mathbb{I}(t < 1)$, $p(x) = \prod_{j=1}^{J} p_j(x)$ and we use the convention $0^0 = 0$ so that $\pi_0(x) = p(x)$. The conditions and effects of ϵ_t on the performance of algorithms have yet to be explored and is part of our future work. By design, the interpolating chains have a wide support (i.e., $p(x) > 0 \Rightarrow \pi_t(x) > 0$ for $t < 1$), 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.

In a given execution of the PT and SCM inference algorithms, the full continuum of distributions $\{\pi_t : t \in [0, 1]\}$ is only instantiated on a finite grid $0 = t_0 < t_1 < \cdots < t_N = 1$, called an *annealing schedule*. Since the performance of both PT and SCM are sensitive to the choice of annealing schedules, they each use a specialized algorithm to automatically optimize the annealing schedule (described in the next sections). To perform a continuous optimization over $(t_1, t_2, \ldots, t_{N-1})$ with monotonicity constraints, the algorithms rely on the fact that the discrete sequence of distributions is embedded in a continuum of distributions.

Sequential change of measure

Informally, Blang's SCM inference engine initializes a population of particles from a prior distribution, and iteratively perturbs and reweighs the particles. The number of particles used is 1 000 by default and can be set using the `--engine.nParticles` option. More precisely, SCM is a special case of the sequential Monte Carlo (SMC) sampler (Del Moral *et al.* 2006, Section 3.3.2.3) combined with an adaptive tempering schedule described by Zhou *et al.* (2016) (also called "annealed SMC", e.g., in Wang *et al.* (2020)). SMC *samplers* are an extension or generalization of SMC methods, which allow for sampling from a sequence of distributions defined on a fixed state space, as opposed to the more common SMC setup (Doucet and Johansen 2009) consisting of product spaces of increasing dimensionality.

As described in Del Moral *et al.* (2006), Section 3.3.2.3, the proposals we use consist in MCMC kernels targeting each of the intermediate distributions (see Section 12.7 for a detailed description on their construction). Del Moral *et al.* (2006) also justify in this context the incremental weight updates given by

$$w_t(x_{t-1}, x_t) = \frac{\gamma_t(x_{t-1})}{\gamma_{t-1}(x_{t-1})}, \tag{7}$$

where $\gamma_t(x)$ is the numerator in the right-hand side of Equation (5). Initialization is done using the prior sampler described in Section 12.6.

Due to the weight degeneracy problem reviewed in Doucet and Johansen (2009), a resampling procedure is required. Resampling prevents the population of sample weights from collapsing into a point mass. However, resampling injects additional noise into the sampling process. Motivated by the need to balance these two factors, we use a standard procedure to adaptively determine when resampling should be performed. The effective sample size (ESS) is computed at each iteration (Kong 1992). If the relative ESS—ESS divided by population size—falls beneath a predetermined threshold, resampling is performed. Figure 8 (left) illustrates the resampling procedure's effect on ESS. This threshold value defaults to 0.5 in `Blang`, and can be set, for example, to 0.4 using the command-line argument `--engine.resamplingESSThreshold 0.4`. By default, the resampling scheme used is the stratified sampling of Kitagawa (1996) (use `--engine.resamplingScheme MULTINOMIAL` for multinomial resampling).

SMC samplers rely on a discrete set of interpolating distributions $0 = t_1 < t_2 < \cdots < t_N = 1$. As initially proposed in Jasra *et al.* (2011) and improved in Zhou *et al.* (2016), instead of building this sequence a priori, we construct it incrementally and adaptively. At each step the next annealing parameter is determined so as to cause a fixed decay in the relative conditional ESS as defined in Zhou *et al.* (2016). Figure 8 (right) shows an example of a resulting adaptive annealing schedule. Finding the next annealing parameter is a simple univariate root finding problem. Since the weight update shown in Equation (7) does not depend on x_t, only the already available particles from the previous iteration, x_{t-1}, the computational cost of the root finding problem is negligible. By default, the targeted decay is set to 0.9999 and can be controlled via `--engine.temperatureSchedule.threshold`. Setting it to a lower value will speed up computation at the cost of a less accurate posterior distribution (and vice versa). One disadvantage of this adaptation scheme is that the running time of the method is random and may be hard to predict a priori. If the user requires a prespecified number of iterations, adaptive construction of the sequence of distribution can be turned off, for example to use a fixed number of 20 iterations and an equally spaced annealing schedule $0 = t_1 < t_2 < \cdots < t_{20} = 1$, use `--engine.temperatureSchedule FixedTemperatureSchedule --engine.temperatureSchedule.nTemperatures 20`. Custom mechanisms to control the schedule can be added by implementing the interface `TemperatureSchedule`.[63] If for example the user implements their own algorithm in a class called `MySchedule` located in package `mypackage`, to enable its use during inference, add the command line arguments `--engine.temperatureSchedule mypackage.MySchedule`.

[63] `https://www.stat.ubc.ca/~bouchard/blang/javadoc-sdk/blang/engines/internals/schedules/TemperatureSchedule.html`

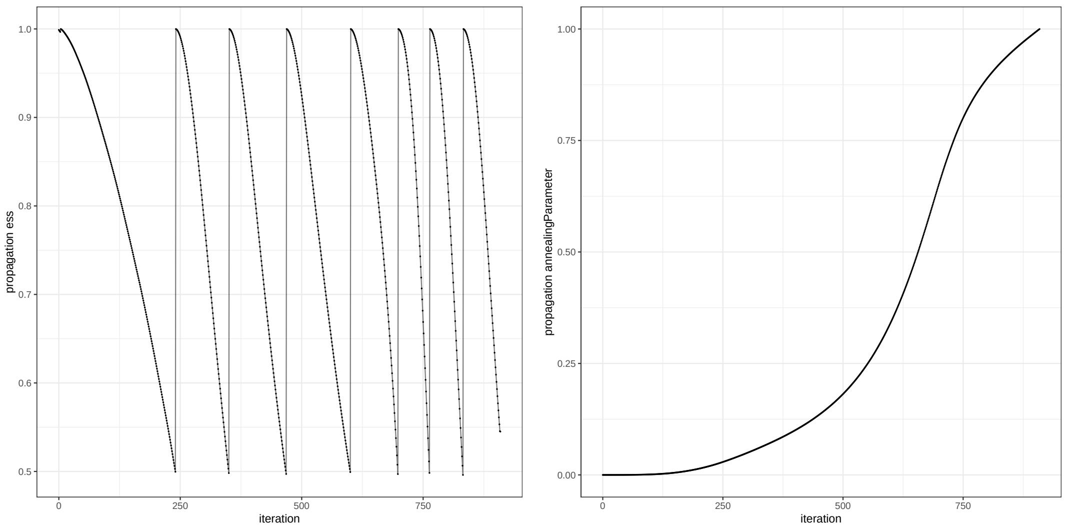


Figure 8: SCM monitoring plots automatically created when SCM is used and the default post-processing tool activated (`--engine SCM --postProcessor DefaultPostProcessor`). Here we show examples for the Gaussian mixture model in Section 6, using 10000 particles. The adaptive resampling scheme based on ESS estimates is performed by default for SCM. Left: ESS plotted against iterations, automatically created in `monitoringPlots/propagation-ess.pdf`. Each "spike" observed in this plot correspond to a resampling step taking place. When ESS falls beneath a predefined threshold, 0.5 here, particles are resampled to prevent weight degeneracy. This resampling procedure "refreshes" the relative ESS to 1.0. Right: The resulting adaptive annealing schedule, found in `monitoringPlots/propagation.pdf`. The y-axis corresponds annealing parameters, and x-axis the iteration number.

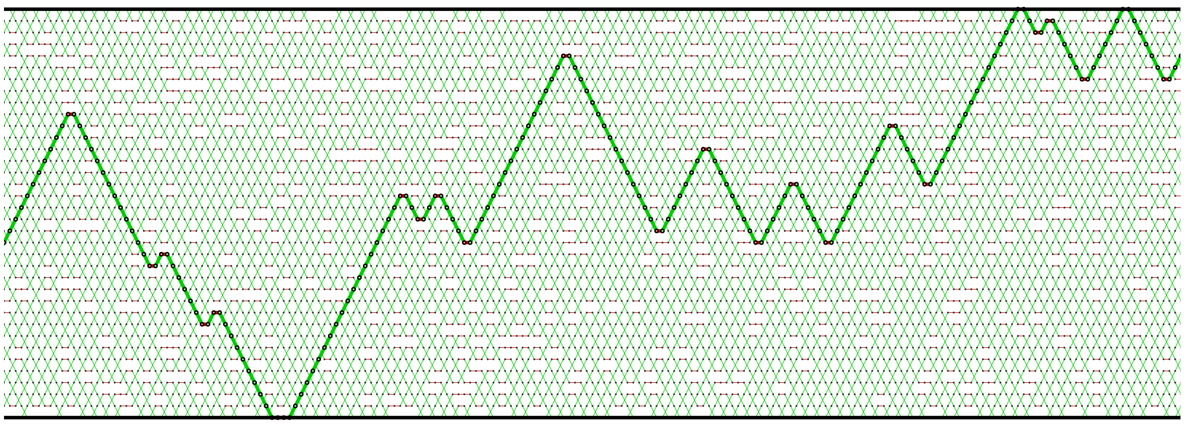


Figure 9: Visualization (cropped) of the chain swaps proposed while running non-reversible PT (add `--postProcessor.runPxviz true --postProcessor.boldTrajectory 1` to create this visualization). The x-axis corresponds to PT iterations, and y-axis corresponds to different parallel chains, with the one at the top corresponding the posterior distribution, the one at the bottom, to the prior distribution, and those in between interpolating between the two. When a swap is accepted (green line segments), two chains exchange their states, denoted by crossing lines. When a swap is rejected we use red line segments. An *index process* is obtained by considering a path formed by these line segments (one index process is shown as a bold line for ease of interpretation). An *annealed restart* is defined as a path segment within an index process which starts at the prior and ends at the posterior.

After SCM inference is performed, Blang performs one last round of resampling followed by 5 rounds of particle rejuvenation on each particle. This results in a set of equally weighted particles. The amount of rejuvenation to perform after the final resampling round can be controlled via `--engine.nFinalRejuvenations 10`.

Non-reversible parallel tempering

Blang incorporates a non-reversible, adaptive parallel tempering (PT) algorithm (Syed *et al.* 2019). PT (Geyer 1991) is an MCMC method that operates on product spaces. Informally, PT runs N Markov chains in parallel, each targeting a distribution from a sequence of tempered (i.e., annealed) distributions indexed by $0 \leq t \leq 1$ (Section 12.1.1). Each PT iteration consists of two phases: a local exploration phase taking place within individual chains, and a communication phase taking place between chains.

In the local exploration step, for chains with $t > 0$, the state is updated via samplers or MCMC kernels invariant with respect to the chain's target distribution (the construction of these kernels is detailed in Section 12.7). For the chain with $t = 0$, the local exploration step consists in an independent draw from the prior distribution (the construction of the independent sampler for the prior is detailed in Section 12.6). If the user requires using MCMC samplers for $t = 0$ instead of prior sampling, the option `--engine.usePriorSamples false` can be used.

In the communication phase, swaps between neighbour chains are proposed and accepted/rejected according to the Metropolis-Hastings ratio. Informally, this swapping procedure provides opportunities for states to traverse across modes, as the prior allows independent sampling and hence a form of regeneration. Even when sampling from the prior is not possible, annealing often yields MCMC kernels with better mixing rates.

In our implementation, both the exploration and communication phases are parallelized in the number of parallel chains N (see 12.3 for details). However to leverage this parallelism, following the theoretical analysis of Syed *et al.* (2019), special attention has been devoted (1) to the details of how the swap mechanism is performed, and (2) to the tuning of the annealing schedule $t_1, t_2, \dots, t_{N-1}$ introduced in the last section.

Point (1) is motivated by a sharp contrast between the performance of reversible and non-reversible flavours of PT. Performance in the following discussion is based on the notion of *annealed restarts*, defined along with the related notion of the index process in Figure 9. We define PT performance as the fraction of iterations where an annealed restart is just completed at the current iteration. This is called the restart rate, which we denote by τ, and it is equivalent (up to an additive factor of 1) to the notion of round trip rate popular in the PT literature (Katzgraber *et al.* 2006; Lingenheil *et al.* 2009).

Previous theoretical work has focused on reversible PT where the groups of chains to swap are selected at random. In the reversible regime, several lines of work (Rathore *et al.* 2005; Atchadé *et al.* 2011) have demonstrated that even when a high number of cores is available, one still has to ensure that N, and hence the number of cores leveraged, is not too large. More precisely, the performance of reversible PT collapses as N increases, even when communication and local exploration are fully parallel. For example the results in Atchadé *et al.* (2011) imply that $\tau_{\text{rev},N} \to 0$ as $N \to \infty$. Surprisingly, this performance collapse disappears when a non-reversible flavour of PT is used: Syed *et al.* (2019) identified conditions where $\tau_{\text{non-rev},N} \to c$ as $N \to \infty$, where the model-dependent constant $c > 0$ is discussed further below. Even more surprising is that algorithmically, the distinction needed to make PT non-reversible is minimal: it is simply the use of a deterministic alternation of two specific types of swap kernels, those swapping $i, i+1$ with i even, followed by similar swaps with i odd. This algorithm can be traced back to Okabe *et al.* (2001), however, only recently its non-reversible dynamics have been identified and used to prove the existence of a qualitative gap between the reversible and non-reversible flavours of PT. The gap can be established both non-asymptotically ($\tau_{\text{rev},N} < \tau_{\text{non-rev},N}$ for all N), and also asymptotically as $N \to \infty$, in which case the performance of non-reversible PT, $\tau_{\text{non-rev},N}$, is furthermore guaranteed to be monotonically increasing for N large enough.

More importantly, non-reversibility opens the door for highly parallel algorithms to optimize over the annealing schedule, hence addressing point (2) above. By default, Blang's PT engine uses the non-reversible schedule optimization from Syed *et al.* (2019) (labelled NRPT henceforth). In contrast, at the time of writing, mainstream probabilistic programming languages either lack support for parallel tempering (Plummer 2003; Lunn *et al.* 2012; Salvatier *et al.* 2015; Carpenter *et al.* 2017), or require manual input of the annealing parameters (Foreman-Mackey *et al.* 2013).

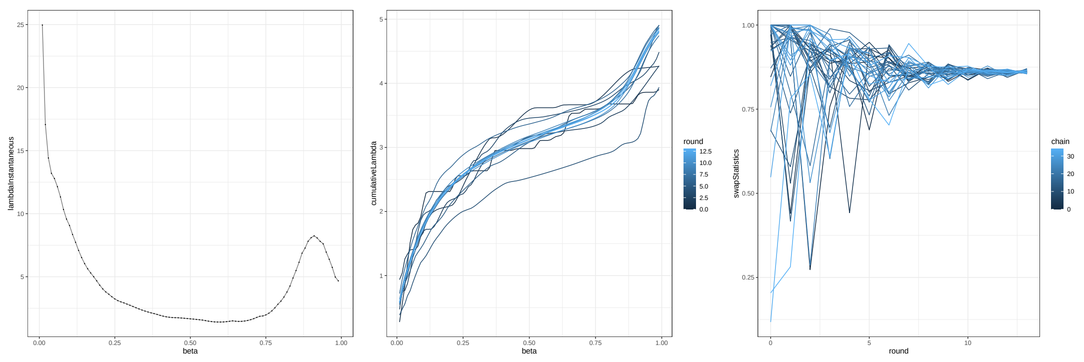


Figure 10: Left: final estimate of the local communication barrier $\hat{\lambda}$ in the GMM example of Section 6. The spiking phenomenon around $\beta = 0.9$ is indicative of a phase transition. This corresponds to the mixture indicator variables going from a disorganized configuration (the side of the peak at $\beta = 0.9$ closer to the prior on the left) to a clustered configuration (the side of the peak closer to the posterior). Middle: estimates of the cumulative communication barrier, $\hat{\Lambda}(t)$, with each colour corresponding to a different iterative round of the annealing schedule optimization algorithm. Right: here each line (colour) is one of the N chains, and the line tracks the average acceptance probability (ordinate) between that chain and its neighbour for each round of the schedule optimization algorithm (abscissa). In contrast to reversible PT, NRPT does not need to restrict swap acceptance probability to low values such as the 23% acceptance rule of Atchadé *et al.* (2011).

To outline how the NRPT algorithm works, we first outline the asymptotic distribution of a single index process (for example, the bold line in Figure 9) as $N \to \infty$. While for reversible PT this distribution converges to a diffusion, for non-reversible PT, it converges to a piecewise-deterministic Markov process (PDMP). See Davis (1993) for background on PDMPs. The rate parameter of this limiting PDMPs, λ, a positive function taking as input an annealing parameter $t \in [0, 1]$, can be interpreted as being proportional to the expected rejection rate for a swap between π_t and $\pi_{t+\epsilon}$. See Figure 10 (left) for an example of an estimate $\hat{\lambda}$ from the model in Section 6. Moreover, the constant c introduced earlier as the asymptotic non-reversible performance, $\tau_{\text{non-rev},N} \to c$ can be written as $c = (2 + \Lambda)^{-1}$, where $\Lambda(t) = \int_0^t \lambda(t')\,\mathrm{d}t'$. We call λ, $\Lambda(t)$, and $\Lambda = \Lambda(1)$ the local, cumulative and global communication barriers respectively.

Importantly, all three communication barriers can be estimated from the MCMC output, and used as the basis for tuning PT as described in (Syed *et al.* 2019). First, Λ can be used as a measure of the difficulty of PT-based inference for a given model: this is supported by its relation with the model-specific constant c described earlier. We recommend to use a number of chain proportional to Λ. Using at least 2Λ appears to provide a good starting point empirically. Second, NRPT uses $\Lambda(t)$ to optimize the annealing parameters using the following strategy. The algorithm iteratively estimates $\hat{\Lambda}(t)$, using a simple and asymptotically consistent rule $\hat{\Lambda}(t_i) = \sum_{j=1}^{i} \hat{r}^{(j-1,j)}$, and $\hat{\Lambda}(\cdot)$ interpolated using a monotone cubic spline between the t_i's, where $\{t_i\}$ are annealing parameters from the previous iterations, and $\hat{r}^{(j-1,j)}$ is the empirical swap rejection rates, also obtained from the previous iteration. The algorithm then computes univariate quantile of $t \mapsto \hat{\Lambda}(t)/\hat{\Lambda}(1)$ to update the annealing schedule. This is repeated using a doubling scheme, where the first round uses 1 iteration to estimate $\hat{\Lambda}(t)$, followed by annealing parameters update, the second round uses 2 iterations based on the updated schedule, followed by an update of the annealing parameters, then 4, 8, etc. See Figure 10 (middle) for an example of how estimates of $\hat{\Lambda}$ progress as the number of rounds increases. As a byproduct of the NRPT algorithm we obtain a burn-in mechanism: by default, all post-processing uses only the samples produced by the last optimization round which is equivalent to a 50% burn-in. The only exception is for the trace plot, which is shown for both the whole MCMC trace in the output folder `tracePlotsFull`, and for the post burn-in phase, `tracePlots`. The default of 50% burn-in can be customized via `--postProcessor.burnInFraction`.

Alternative mechanisms can be used to control the annealing parameters. By default, the initial annealing schedule is uniform, other initial values can be used, see `--engine PT --help` for various options. Optimization of the annealing parameters can be disabled with the argument `--engine.adaptFraction 0.0`. Custom mechanisms to control the initial schedule can be added by the user by implementing the interface `TemperatureLadder`.[64] If for example the user implements their own algorithm in a class called `MyLadder` located in package `mypackage`, to enable it, use `--engine.ladder mypackage.MyLadder`.

One tuning parameter that can be used to speed-up the execution of NRPT is the expected number of times each local exploration kernel should be used between two rounds of swap attempts, `--engine.nPassesPerScan` (fractional values are accepted). By default, this is set to 3, so that a theoretical assumption called Effective Local Exploration (ELE) Syed *et al.* (2019), is well approximated. However, we observed that performance was robust to this choice so if the local exploration kernels are reasonably efficient, lower values will lead to similar behaviour of the index processes for a lower computational budget. Conversely, if the local exploration kernels perform very poorly, it may be useful to explore higher values for the argument `--engine.nPassesPerScan`.

If the hard-drive space required to store the samples produced by PT becomes prohibitive, one option is to enable thinning by providing an input `--engine.thinning` greater than one. For example, `--engine.thinning 2` will store samples only once every two PT iterations. An alternative (available for all engines), is to compress the samples in `.gz` format, which is enabled using `--experimentConfigs.tabularWriter.compressed true`. All post-processing is compatible with the compressed samples format.

Initialization of PT is by default performed by first running SCM using an annealing schedule containing all annealing parameters in PT's initial schedule. The SCM initialization can be configured using the same arguments as those described in Section 12.1.2 but with the prefix `engine.scmInit`. For example, to increase the number of particles (set to 100 for the initialization run), use `--engine.scmInit.nParticles 200`.

Other inference engines

When a model is not in generative normal form, the PT and SCM engines cannot be used. In such case, the user can still a basic, single chain MCMC via `--engine MCMC`. This option is essentially a shortcut for setting the PT engine to use a single chain, to avoid using SCM for initialization, and to avoid other checks that assume a generative normal form. The PT command line arguments from Section 12.1.3 that are relevant to single-chain MCMC can still be used, in particular `--engine.nScans` and `--engine.thinning`.

[64] `https://www.stat.ubc.ca/~bouchard/blang/javadoc-sdk/blang/engines/internals/ladders/TemperatureLadder.html`

Another convenient shortcut is `--engine AIS` which uses SCM but with resampling disabled. This is known as the Annealed Importance Sampling algorithm Neal (2001). The SCM arguments relevant to AIS can still be used with this engine, namely `--engine.nParticles`, `--engine.nFinalRejuvenation`, and `--engine.temperatureSchedule.threshold`.

In cases where the user would like to sample independent and identically distributed realization from a model where no observation is present, the engine `--engine Forward` (for forward sampling) with option `--engine.nSamples 1` can be used.

When all random variables in a small model are discrete, the argument `--engine Exact` will enumerate all possible scenarios. Note that the `DefaultPostprocessor` should not be used to analyze the output of the exact engine. This is because the output in the folder `samples` have a different interpretation than with the other engines: instead of representing equally weighted samples, they represent weighted samples with weight indicated in a row called `logProbability`.

Finally, the inference engine can be customized. This is achieved by implementing the interface `PosteriorInferenceEngine`.[65] If for example the user implements their own algorithm in a class called `MyEngine` located in package `mypackage`, to enable its use during inference, add the command line arguments `--engine mypackage.MyEngine`.

12.2. Pseudo-random generator

The current pseudo-random generator is the Mersenne Twister Matsumoto and Nishimura (1998) as implemented in the **MathCommons** package. By default, the seed 1 is used. For inference engines based on randomized algorithms (all current algorithms except `Exact`), this can be changed using the command line argument `--engine.random` followed by an integer.

12.3. 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.

For controlling multi-threading, use `--engine.nThreads Max` to take advantage of as many threads as there are cores in the host machine, `--engine.nThreads Dynamic` to dynamically allocate threads based on the overall system usage (the default behaviour, which ensures analysts can smoothly carry other tasks while inference is running in the background), `--engine.nThreads Single` to force single-thread mode, and `--engine.nThreads Fixed --engine.nThreads.number 2` to fix a specific number of threads to use.

[65] https://www.stat.ubc.ca/~bouchard/blang/javadoc-sdk/blang/engines/internals/PosteriorInferenceEngine.html

12.4. Marginal likelihood computation

Standard Bayesian model selection requires computing the marginal likelihood, also known as the evidence. The marginal likelihood is conceptually simple: it is the probability or the density of the observed data. However computing or approximating this single scalar is often challenging. Fortunately, both PT and SCM automatically compute the marginal likelihood with no extra computational cost.

Our PT engine supports estimation of the marginal likelihood through two methods: thermodynamic integration (Ogata 1989), and the stepping stone estimator (Xie *et al.* 2011). For models with hard constraints (i.e., models whose likelihood is equal to zero for particular configurations of states proposed by the sampling algorithm), the technical conditions underlying thermodynamic integration may not be satisfied, and that estimator is automatically omitted in such cases. The stepping stone estimator can still be used in these cases. In SCM, the evidence comes as a by-product of the weights computed by the algorithm, see e.g., Del Moral *et al.* (2006).

In contrast to Blang, other mainstream probabilistic programming languages require additional packages and external dependencies to approximate the marginal likelihood. For example 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).

12.5. Diagnostics

We summarize here some diagnostic strategies that can be used to assess the quality of the posterior distribution approximation. With the PT inference engine, the key diagnostics are the ESS estimates (Section 4) and the number of annealed restarts (see Figure 9 for the definition, and `monitoring/actualTemperedRestarts.csv` for the estimates). Each annealed restart incorporates a unique independent draw from the prior chain successfully propagated to the posterior chain. This can be complemented with inspection of the trace plots (Section 4). Finally, another strategy is to monitor the marginal likelihood: a separate estimate is provided for each adaptation round in the PT engine (in `monitoring/logNormalizationContantProgress.csv`), so its convergence can be readily monitored, and moreover one can check the agreement of PT's estimate with the orthogonal marginal likelihood estimator used by SCM (either based on the automatic SCM initialization, or from a separate run; see `logNormalizationEstimate.csv`).

12.6. Construction of prior samplers

Consider the sequence of distribution in Equation (5) at $t = 0$, where we recover the prior distribution $p(x)$. When a model is in generative normal form (Section 5.7), Blang automatically constructs an efficient algorithm to sample from the prior distribution $p(x)$. Briefly, the normal form property guarantees that we can orient the factor graph over the latent variables into a directed graphical model. The generative normal form property enables the enumeration of forward samplers provided by the `generate` blocks. Finally, as a preprocessing step, we order these `generate` blocks according to a linearization of the directed graphical model.

12.7. 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 `SampledModel`,[66] 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, 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. 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, the latent variables in 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. Mutability here corresponds to the class having fields that are either arrays or that are *non-final* (the latter being the Java terminology, or equivalently, in Xtend, fields not marked by `val`). 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 `Observations` object.[67] For example, `observationsObject.markAsObserved(mtx.getRealVar(i, j))` marks entry (i, j) of matrix `mtx` as observed. In scenarios where objects or fields are accessible but unused in factors, the exploration of such objects and fields can be skipped. This can be handled in the construction of the accessibility graph by using the annotation `@SkipDependency`. With our accessibility graph constructed and latent variables identified, we can now exploit the model's sparsity patterns by constructing a factor graph.

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 whether the application of a sampling operator on v can change the numerical value of the factor f. This is accomplished by

[66] https://www.stat.ubc.ca/~bouchard/blang/javadoc-sdk/blang/runtime/SampledModel.html
[67] https://www.stat.ubc.ca/~bouchard/blang/javadoc-sdk/blang/runtime/Observations.html

assessing v and f's co-accessibility. Two objects o_1 and o_2 are said to be co-accessible if there is a mutable object (as defined previously) o_3 such that o_3 is accessible from 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 11.2).

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 9. Here we provide more details on this process.

To implement a sampler, we turn our attention to the `interface Sampler`,[68] 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 precomputation. 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.

`SimplexSampler.xtend`[69]

```
...
class SimplexSampler implements Sampler {
  @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 accessibility 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

[68] https://www.stat.ubc.ca/~bouchard/blang/javadoc-sdk/blang/mcmc/Sampler.html

[69] https://github.com/UBC-Stat-ML/blangSDK/blob/master/src/main/java/blang/mcmc/SimplexSampler.xtend

All the programs in this paper were run using **blangSDK** 2.10.3 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.

Funding

This research was supported by a Discovery Grant from the National Science and Engineering Research Council, a Canadian Statistical Sciences Institute Collaborative Research Team Project, and Michael Smith Foundation RRF Grant.

References

Ackerman NL, Freer CE, Roy DM (2017). “On Computability and Disintegration.” *Mathematical Structures in Computer Science*, **27**(8), 1287–1314.

Atchadé YF, Roberts GO, Rosenthal JS (2011). “Towards optimal scaling of Metropolis-coupled Markov chain Monte Carlo.” *Statistics and Computing*, **21**(4), 555–568. ISSN 0960-3174, 1573-1375. doi:10.1007/s11222-010-9192-1.

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.

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.

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.

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.

Chen MH, Shao QM (1999). “Monte Carlo Estimation of Bayesian Credible and HPD Intervals.” *Journal of Computational and Graphical Statistics*, **8**(1), 69–92. ISSN 1061-8600. doi:10.2307/1390921.

Davis MH (1993). *Markov Models & Optimization*. Chapman and Hall.

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. 0212648.

Doucet A, Johansen AM (2009). "A tutorial on particle filtering and smoothing: Fifteen years later." *Handbook of nonlinear filtering*, **12**(656-704), 3.

Duane S, Kennedy A, Pendleton BJ, Roweth D (1987). "Hybrid Monte Carlo." *Physics Letters B*, **195**(2), 216–222. ISSN 03702693.

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.

Foreman-Mackey D, Hogg DW, Lang D, Goodman J (2013). "emcee: The MCMC Hammer." *Publications of the Astronomical Society of the Pacific*, **125**(925), 306. ISSN 1538-3873. doi:10.1086/670067. Publisher: IOP Publishing.

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

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*, pp. 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.

Griffiths TL, Ghahramani Z (2011). "The Indian Buffet Process: An Introduction and Review." *Journal of Machine Learning Research*, **12**(32), 1185–1224. ISSN 1533-7928.

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, Heath TA, Boussau B, Landis MJ, Ronquist F, Huelsenbeck JP (2014). "Probabilistic Graphical Model Representation in Phylogenetics." *Systematic Biology*, **63**(5), 753–771. ISSN 1063-5157. doi:10.1093/sysbio/syu039.

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.

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

Jasra A, Holmes CC, Stephens DA (2005). "Markov chain Monte Carlo methods and the label switching problem in Bayesian mixture modeling." *Statistical Science*, pp. 50–67.

Jasra A, Stephens DA, Doucet A, Tsagaris T (2011). "Inference for LÃľvy-Driven Stochastic Volatility Models via Adaptive Sequential Monte Carlo." *Scandinavian Journal of Statistics*, **38**(1), 1–22. ISSN 1467-9469. doi:10.1111/j.1467-9469.2010.00723.x.

Katzgraber HG, Trebst S, Huse DA, Troyer M (2006). "Feedback-optimized parallel tempering Monte Carlo." *Journal of Statistical Mechanics: Theory and Experiment*, **2006**(03), P03018.

Kitagawa G (1996). "Monte Carlo Filter and Smoother for Non-Gaussian Nonlinear State Space Models." *Journal of Computational and Graphical Statistics*, **5**(1), 1–25. ISSN 1061-8600. doi:10.2307/1390750.

Kong A (1992). "A note on importance sampling using standardized weights." *Technical Report 348*, The University of Chicago.

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.

Leiserson CE, Schardl TB, Sukha J (2012). "Deterministic parallel random-number generation for dynamic-multithreading platforms." *ACM Sigplan Notices*, **47**(8), 193–204.

Lingenheil M, Denschlag R, Mathias G, Tavan P (2009). "Efficiency of exchange schemes in replica exchange." *Chemical Physics Letters*, **478**(1-3), 80–84.

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.

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.

Matsumoto M, Nishimura T (1998). "Mersenne Twister: A 623-dimensionally Equidistributed Uniform Pseudo-random Number Generator." *ACM Trans. Model. Comput. Simul.*, **8**(1), 3–30. ISSN 1049-3301.

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.

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.

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

Neal RM (2001). "Annealed Importance Sampling." *Statistics and Computing*, **11**(2), 125–139. ISSN 09603174. 9803008.

Neal RM (2003). "Slice Sampling." *The Annals of Statistics*, **31**(3), 705–741. ISSN 00905364.

Neal RM (2011). "MCMC using Hamiltonian dynamics." *Handbook of Markov chain Monte Carlo*, **2**(11).

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

Okabe T, Kawata M, Okamoto Y, Mikami M (2001). "Replica-exchange Monte Carlo method for the isobaric–isothermal ensemble." *Chemical physics letters*, **335**(5-6), 435–439.

Paige B, Wood F (2014). "A compilation target for probabilistic programming languages." *arXiv preprint arXiv:1403.0504*.

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.

Rathore N, Chopra M, de Pablo JJ (2005). "Optimal allocation of replicas in parallel tempering simulations." *The Journal of Chemical Physics*, **122**(2), 024111. doi:10.1063/1.1831273.

Ronquist F, Kudlicka J, Senderov V, Borgström J, Lartillot N, Lundén D, Murray L, Schön TB, Broman D (2020). "Probabilistic programming: a powerful new approach to statistical phylogenetics." *BioRxiv*.

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.

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.

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.

Teh YW, Jordan MI, Beal MJ, Blei DM (2006). "Hierarchical Dirichlet Processes." *Journal of the American Statistical Association*, **101**(476), 1566–1581. doi:10.1198/016214506000000302.

van de Meent JW, Paige B, Yang H, Wood F (2018). "An Introduction to Probabilistic Programming." *arXiv:1809.10756 [cs, stat]*. ArXiv: 1809.10756.

Wang L, Wang S, Bouchard-CÃťtÃľ A (2020). "An Annealed Sequential Monte Carlo Method for Bayesian Phylogenetics." *Systematic Biology*, **69**(1), 155–183. ISSN 1063-5157. doi:10.1093/sysbio/syz028.

Wickham H (2014). "Tidy Data." *Journal of Statistical Software, Articles*, **59**(10), 1–23. ISSN 1548-7660.

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.

Xie W, Lewis PO, Fan Y, Kuo L, Chen MH (2011). "Improving marginal likelihood estimation for Bayesian phylogenetic model selection." *Systematic biology*, **60**(2), 150–160.

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, Gram-Hansen BJ, Kohn T, Rainforth T, Yang H, Wood F (2019). "LF-PPL: A Low-Level First Order Probabilistic Programming Language for Non-Differentiable Models." In K Chaudhuri, M Sugiyama (eds.), *The 22nd International Conference on Artificial Intelligence and Statistics, AISTATS 2019, 16-18 April 2019, Naha, Okinawa, Japan*, volume 89 of *Proceedings of Machine Learning Research*, pp. 148–157. PMLR.

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. 1303.3123.

A. Advanced Tutorial

A.1. Inference on a non-standard data structures via third-party libraries

Consider an inference problem where the data structure or parameter 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, using a third-party library.

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.

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)[70]:

`dependencies.txt`

```
com.github.UBC-Stat-ML:conifer:2.1.3
```

We encode the Blang model, using the imported data type in the following code chunk

`PhylogeneticTree.bl`

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

model PhylogeneticTree {

  random RealVar shape ?: latentReal()
  random RealVar rate ?: latentReal()
  random SequenceAlignment observations
  param EvolutionaryModel evoModel ?: kimura(observations.nSites)

  random UnrootedTree tree ?: unrootedTree(observations.observedTreeNodes)

  laws {
    shape ~ Exponential(1.0)
    rate ~ Exponential(1.0)
    tree | shape, rate ~ NonClockTreePrior(Gamma.distribution(shape, rate))
    observations | tree, evoModel ~ UnrootedTreeLikelihood(tree, evoModel)
  }
}
```

[70] https://github.com/UBC-Stat-ML/conifer/tree/master/src/main/java/conifer

In the first block of code, unobserved variables of the type `RealVar` and IntVar are declared using the functions `latentReal()` and `latentInt()`.[71] This is no different from our previous examples. In the second block, `NonClockTreePrior` and `UnrootedTreeLikelihood` are themselves Blang models defined in the imported **conifer** package. `NonClockTreePrior` accepts a distribution as an argument, in the example an XExpression is used to pass in a Gamma distribution directly without the need to declare another variable in the model.

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

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

Here `--model.observations.file` specifies the data path, this is the standard Blang input method. `--model.observations.encoding` is a model-specific option to parse our data, provided by the third-party library. The usual outputs can be found in the `results` directory.

On the whole, to use third-party libraries or packages (not necessarily restricted to Blang), users just need to specify the dependencies in `dependencies.txt`, and include `import` statements as needed. Running the model can be done via the usual CLI. Inputs follow the same syntax, unless otherwise instructed by the third-party library (i.e., custom parsers). Outputs are also placed in the usual directories.

A.2. Spike and slab classification

[71] A summary of the most commonly used functions are listed in Figure 15 and Figure 16.

In this example, we focus on the implementation of a non-standard data type to handle a spike and slab model (Mitchell and Beauchamp 1988). 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 file `SpikedRealVar.xtend` shown in the code block below contains the implementation of the data type `SpikedRealVar` using Xtend.

`SpikedRealVar.xtend`

```
package jss.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 }
}
```

Because we want to use `RealVar` and `IntVar` types in our `SpikedRealVar` type (Xtend), we require the import statements of core Blang types, as the usual automatic imports are only for Blang (`.bl`) files. 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, this time using Blang. The code below is contained in `SpikeSlabClassification.bl`.

`SpikeSlabClassification.bl`

```
package glms

model SpikeSlabClassification {

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

  param Plate<String> instances, features
  param Plated<Double> covariates
  random Plated<IntVar> labels
  random Plated<SpikedRealVar> parameters

  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 10.4.1. Note this is not the only way to implement a spike and slab model. For instance, a user could define a distribution and sampler for the `SpikedRealVar` type itself. This example merely illustrates a minimal implementation that takes advantage of Blang's preexisting types, distributions, and samplers.

To perform inference on the model `SpikeSlabClassification`, we call the following using 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 \
    --engine.nScans 10000 \
    --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.

Figure 11 shows a subset of automatically post-processed trace plots. Summary statistics for `SpikeSlabClassification` model's `parameters` are shown below (with truncated values):

index	features	mean	sd	min	median	max	HDI.lower	HDI.upper
1	Age	−0.02	0.02	−0.11	−0.01	0.01	−0.05	0
2	child	1.11	1.04	−1.76	0.97	5.23	−0.11	2.79
3	Fare	0.00	0.00	−0.01	0	0.02	0	0
4	female	3.15	0.44	1.43	3.14	4.89	2.43	3.88
5	Par..Aboard	0.01	0.13	−0.79	0	0.89	−0.21	0.25
6	Pclass	−0.50	0.29	−1.62	−0.51	0.19	−0.88	0
7	Sib..Aboard	−0.71	0.26	−1.90	−0.70	0.07	−1.14	−0.27

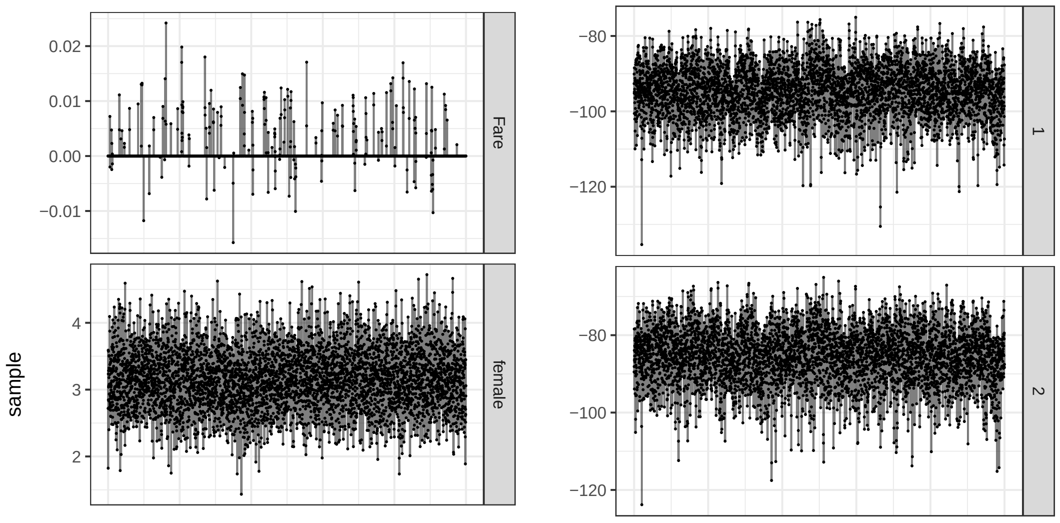


Figure 11: Trace plots for a subset of various random variables in the spike and slab model. Left: the coefficients visit zero with positive probability as expected. Right: log densities for two of the 20 tempered chains used in PT.

B. 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.

B.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-fledged 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.

B.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,[72] as of

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

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

B.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 precompute 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.

`html`

C. Library dependencies

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.[73] Aside from libraries developed for Blang, **briefj**, **inits**, **bayonet**, **rejfree**, **binc**, **xlinear**, and **pxviz**,[74] the language depends on three additional, external libraries: **Cloning**[75], **JGraphT**[76], and **Xbase**[77]. For users who require automatic post-processors, R with packages **dplyr** and **ggplot2** are required. 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.

Library	Description
briefj	Utilities for writing succinct Java code.
inits	A framework to organize inputs and outputs of scientific simulations.
bayonet	Various low-level utilities for probabilistic inference.
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.
MathCommons	The Apache Commons Mathematics Library.
Xbase	Used as the base language for the DSL.

Figure 12: A summary of library dependencies (automatically downloaded during installation, along with the transitive closure of these dependencies).

C.1. bayonet

The **bayonet** `https://github.com/UBC-Stat-ML/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.

C.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

[73] `https://gradle.org/`
[74] All of which are hosted on `https://github.com/UBC-Stat-ML/`.
[75] `https://mvnrepository.com/artifact/uk.com.robust-it/cloning/1.9.6`
[76] `https://mvnrepository.com/artifact/org.jgrapht/jgrapht-core/0.9.0`
[77] `https://wiki.eclipse.org/Xbase`

hierarchical command-line arguments. Blang's CLI argument setup is automatically handled by **inits**.

C.3. xlinear

Blang's linear algebra is based on **xlinear** `https://github.com/UBC-Stat-ML/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`.

D. 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 about the initialization process.
- `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`).
- `logNormalizationEstimate.csv`: 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).

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

- `ess`: information for ess and energy for each chain.
- `monitoringPlots`: sampler diagnostic plots
- `posteriorPlots`: posterior densities and probability mass functions.
- `summaries`: summary statistics such as posterior means, HDIs, etc.
- `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

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

index	**sample**	**value**
0	0	0.453
1	0	0.386
2	0	0.886
0	1	0.520
1	1	0.345
2	1	0.940

By default, the method `toString` is used to create the last column (value). How can this be modified to encompass arbitrary data types? For example, how do we output an object from permutation space (as in Section 9) as a Tidy CSV like below:

index	permutation_index	sample	value
0	0	0	2
0	1	0	0
0	2	0	1
1	0	0	1
⋮	⋮	⋮	⋮

This behaviour can be customized to adhere to the Tidy philosophy by implementing the interface `TidilySerializable` for a class of arbitrary data type.[78] The method `serialize` is invoked and passed an instance of `Context`.[79] Using `context.recurse(Object child, Object key, Object value )`, we can instruct the sampler to parse and output the custom data type:

`Permutation.xtend`

```
override void serialize(Context context) {
  for (int i : 0 ..< componentSize)
    context.recurse(connections.get(i), "permutation_index", i)
}
```

The `child` argument is the value to write. The `key` is the name of the key, for example `permutation_index`. The `value` is the value of the key, for example index `i` of value in object.

Additional examples can be found in `TestTidySerializer.xtend`.[80]

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.

[78] https://www.stat.ubc.ca/~bouchard/blang/javadoc-inits/blang/inits/experiments/tabwriters/TidilySerializable.html
[79] See the Context design pattern.
[80] https://github.com/UBC-Stat-ML/inits/blob/master/src/test/java/blang/inits/TestTidySerializer.xtend

- `--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 can now be inferred from the names of the parent directories). In certain scenarios this could save disk space. Inter-operable with Spark.

E. List of probability distributions in Blang's library

E.1. Discrete distributions

Random variables in this section are integer-valued, hence `IntVars`.

`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 $(0, 1, 2, \dots, n)$.

- `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 $(p_0, p_1, \dots, p_{n-1})$ for each of the n integers.

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

- `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 $\{0, 1, 2, \dots\}$

- `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.

E.2. Continuous Distributions

Random variables in this section are real-valued, hence `RealVar`s.
`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 centre 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. $k \in (0, \infty)$

E.3. Multivariate distributions

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

- `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. Random variables with this distribution are of type `Matrix`.

- `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. Random variables with this distribution are of type `Plated<RealVar>`.

- `param Precision precision`: Precision matrix structure.

`SimplexUniform`: n dimensional Dirichlet with all concentrations equal to one. Variables with this distribution are of type `Simplex`.

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

`SymmetricDirichlet`: n dimensional Dirichlet with all concentrations equal to $\frac{\alpha}{n}$. Random variables with this distribution are of type `Simplex`.

- `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`: A utility to handle undirected models (or random fields).

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

F. 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`
`asBool(int i)`	Returns `false` for `0` and `true` for `1`	`boolean`
`asBool(IntVar i)`	Returns `false` for `0` and `true` for `1`	`boolean`
`isBool(int i)`	Returns `true` if `i` is `0` or `1` and `false` otherwise.	`boolean`
`isBool(IntVar i)`	Returns `true` if `i` is `0` or `1` and `false` otherwise.	`boolean`
`asInt(boolean b)`	Returns `0` for `false` and `1` for `true`	`int`
`asList(Plated<T> plated, Plate<Integer> plate)`	Returns a the plated variable as a list.	`List<T>`
`asCollection(Plated<T> plated, Plate<Integer> plate)`	Returns a the plated variable as a collection.	`Collection<T>`
`asMap(Plated<T> plated, Plate<Integer> plate)`	Returns a the plated variable as a map.	`Map<K, T>`
`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»`
`index(K key)`	Get the index given a `key`.	`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<Query, 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 to 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