

# How to perform correct and efficient Bayesian inference

- Previous methods focussed on either *correct* or *efficient*, not both
  - *correctness*: general purpose probabilistic programming languages
  - *efficiency*: programming from scratch

# Blang

- Our effort: Blang, a Bayesian modelling language focusing on supporting correct and efficient combinatorial space sampling
  - Open source
  - Project page: <https://www.stat.ubc.ca/~bouchard/blang/>

Collaborators: Davor Cubranic, Sahand Hosseini, Matteo Lepur, Kevin Chern, Zihui Ouyang

# Declarative model construction

- model: a collection of probability distributions (law's) indexed by parameters (i.e. a family)
- one way to build a model is to declare (conditional) probabilities for each random variable (syntax inspired by WinBUG/JAGS)

$x \mid \lambda \sim \text{Poisson}(\lambda)$
- motivation: stay close to mathematical notation

```
model SimplePhyloExample {  
    random RealVar shape ?: latentReal, rate ?: latentReal  
    random SequenceAlignment observations  
    random UnrootedTree tree ?: unrootedTree(observations.observedTreeNodes)  
    param EvolutionaryModel evoModel ?: kimura(observations.nSites)  
  
    laws {  
        shape ~ Exponential(1.0)  
        rate ~ Exponential(1.0)  
        tree | shape, rate ~ NonClockTreePrior(Gamma::distribution(shape, rate))  
        observations | tree, evoModel ~ UnrootedTreeLikelihood(tree, evoModel)  
    }  
}
```

# Model composition

Language	Language used for its standard library
Win / Open BUGS	Pascal
RevBayes	C++
JAGS	C++
Stan	C++
Blang	<b>Blang</b>

# Model composition

- model: a collection of distributions indexed by parameters
- can be specified using conditional distributions
- a conditional distribution can be built from a parameterized collection of distributions  
(parameter ↠ conditional)
- ⇒ can build a model using other model's

```
model SimplePhyloExample {  
...  
  
laws {  
    shape ~ Exponential(1.0)  
    ...  
}  
}
```

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

```
model Gamma {  
    random RealVar realization  
    param RealVar shape  
    param RealVar rate  
  
    laws {  
        logf(shape, rate, realization) {  
            if (shape <= 0.0 || rate <= 0)  
                return NEGATIVE_INFINITY  
            if (realization <= 0.0)  
                return NEGATIVE_INFINITY  
            ...  
        }  
        ...  
    }  
}
```

# Composition via parameters

- Pushing this idea a bit further, parameters themselves can be models (distributions)
- Other examples: distribution of branch lengths, Dirichlet process' base measure, etc.

```
model IntMixture {  
    param Simplex proportions  
    param List<IntDistribution> components  
    random IntVar realization  
  
    laws {  
        logf(proportions, components, realization) {  
            var sum = 0.0  
            if (components.size !== proportions.nEntries) throw new RuntimeException  
            for (i : 0 ..< components.size) {  
                val prop = proportions.get(i)  
                if (prop < 0.0 || prop > 1.0) return NEGATIVE_INFINITY  
                sum += prop * exp(components.get(i).logDensity(realization))  
            }  
            return log(sum)  
        }  
    }  
  
    generate (rand) {  
        val category = rand.categorical(proportions.vectorToArray)  
        return components.get(category).sample(rand)  
    }  
}
```

```
...  
observation | proportions, lambda, rho  
~ IntMixture(  
    proportions,  
    #[  
        Poisson::distribution(lambda),  
        YuleSimon::distribution(rho)  
    ]  
)  
...
```

Usage example

Example: a mixture of discrete distributions, taking a list of distributions as parameter: the mixture components

- Each Blang model is turned into an inference program (currently command line, but more interfaces under development; currently working on the r interface)
- This program takes as input observed data and outputs posterior samples for the unobserved variables  
Concretely, inputs are currently command line arguments for each variable (organized hierarchically, most with sensible default values. try --help).  
Outputs are *tidy* csv files (Wickham, 2013).

Try at home!  
Needed: Oracle  
Java 8, git,  
POSIX

```
> git clone https://github.com/UBC-Stat-ML/blangExample.git
[cloning]

> ./gradlew installDist
[downloading dependencies and compiling]

> ./build/install/example/bin/example \
--model.observations.file data/primates.fasta \
--model.observations.encoding DNA \
--engine SCM \
--engine.nThreads Max \
--excludeFromOutput observations

Preprocessing started
4 samplers constructed with following prototypes:
RealScalar sampled via: [RealSliceSampler]
UnrootedTree sampled via: [SingleNNI, SingleBranchScaling]
Sampling started
[sampling progress report]
Normalization constant estimate: -1216.1211229417504
Final rejuvenation started
Preprocessing time: 141.4 ms
Sampling time: 2.304 min
executionMilliseconds : 138405

```

# Open type system

- Blang allows object-oriented development of custom random variable datatypes (combinatorial objects)

```
/**  
 * An unrooted phylogenetic tree.  
 * @author Alexandre Bouchard (alexandre.bouchard@gmail.com)  
 */  
@Samplers(SingleNNI, SingleBranchScaling) →  
class UnrootedTree {  
    val Map<UnorderedPair<TreeNode, TreeNode>, Double> branchLengths  
    val UndirectedGraph<TreeNode, UnorderedPair<TreeNode, TreeNode>> topology  
    ...  
}
```

```
class SingleBranchScaling extends MHSampler {  
    @SampledVariable UnrootedTree variable  
  
    override propose(Random rand, Callback callback) {  
        val allEdges = newArrayList(variable.topology.edgeSet)  
        val edge = sample(allEdges, rand)  
        val oldValue = variable.getBranchLength(edge)  
        val m = exp(2.0 * log(2.0) * (rand.nextDouble - 0.5))  
        callback.setProposalLogRatio(log(m))  
        variable.updateBranchLength(edge, m * oldValue)  
        if (!callback.sampleAcceptance)  
            variable.updateBranchLength(edge, oldValue)  
    }  
}
```

- For scalability, user may need to write a sampler, but..
  - user can write using the same syntax
  - Bar is lower in terms of efficiency : thanks to advanced posterior simulation methods (beyond MCMC)
  - Blang helps you checking correctness of implementation

# Correctness

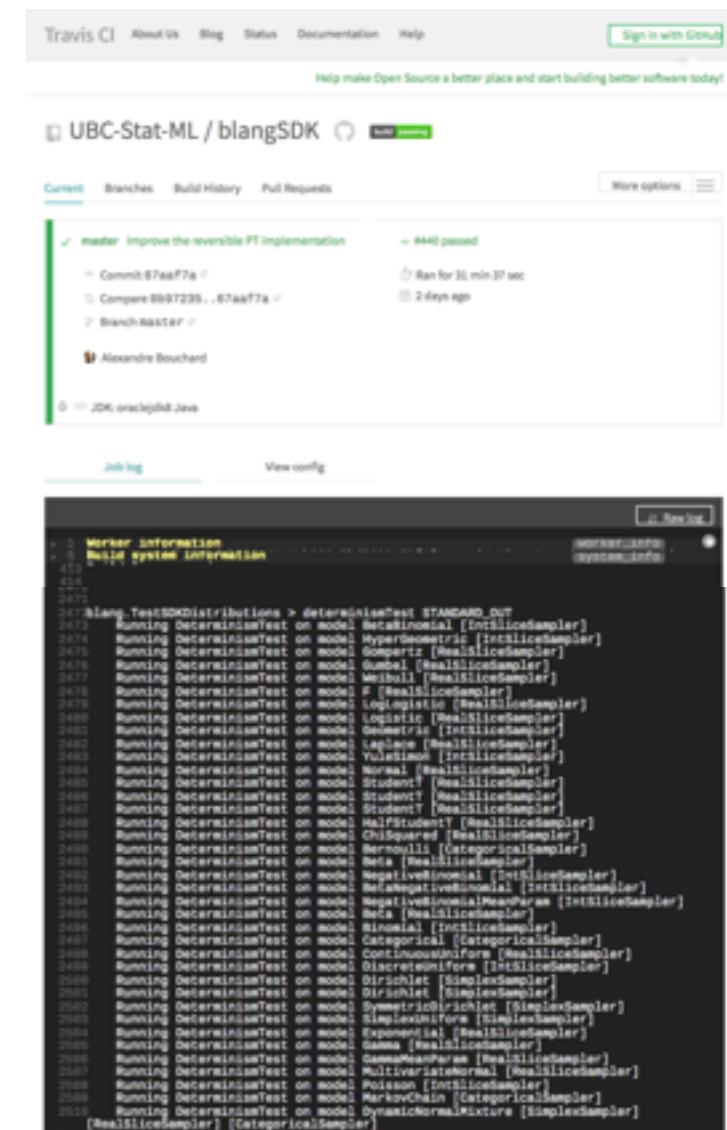
- How to check MCMC code is invariant and/or irreducible?

build passing

- Seems hard; surprisingly very good tests can be constructed

- Example: for discrete model's, Blang's DiscreteMCTest utility will:

- enumerate all the execution traces for each associated samplers
- compute the posterior  $\pi$  by enumeration
- build an explicit transition matrix  $M$  for each sample
- check that  $\pi = M\pi$ , as well as irreducibility
- Many more tests, existing and novel, including tests for continuous models (e.g., Geweke, 2004)



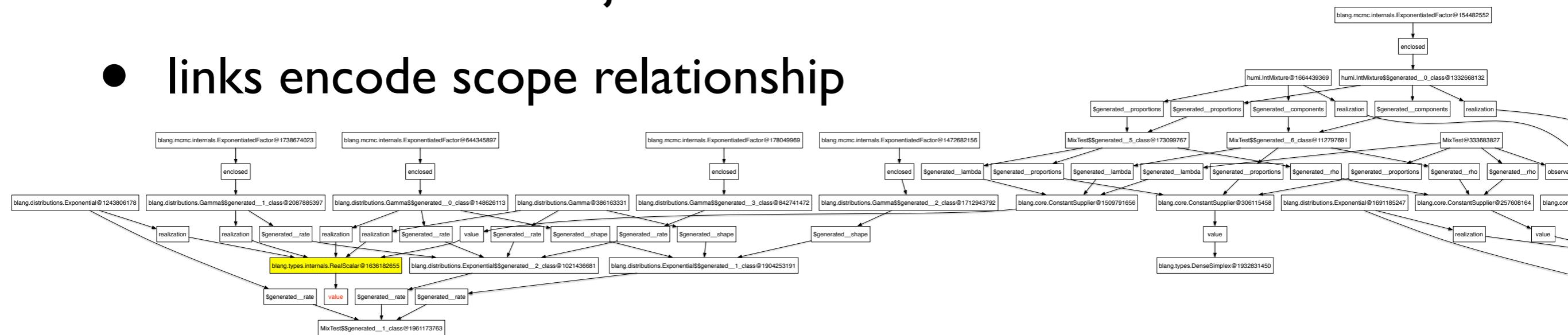
JUnit integration

# Posterior inference

- Blang uses the declarative syntax to automatically build a sequence of distributions instead of just the posterior distribution
- Key to the advanced *inference engines* available
  - *Non-reversible Parallel Tempering*
  - *Sequential Change of Measure*

# Exploiting sparsity

- Samplers can be much faster if they avoid recomputing all model components
  - these savings are formalized by sparse *factor graphs* (Clifford, 1990)
- Idea: building factor graphs using *scoping information*
- Details: get factor graph by running linear time graph algorithms on an *accessibility graph*:
  - in which vertices objects
  - links encode scope relationship



# Language details

- Built using Xtext, (framework for designing programming languages)
- Allowed us to “quickly” (3yrs) build an extensive multi-paradigm language
  - in addition to declarative capability, supports functional, generic and object programming, static typing, just-in-time compilation, garbage collection, etc
  - advanced IDE support
  - Runs on JVM, interoperates with Java
  - fast (just-in-time compiled)
  - can exploit arbitrarily many cores

```
grammar ca.ubc.stat.blang.BlangDsl with
org.eclipse.xtext.xbase.annotations.XbaseWithAnnotations

generate blangDsl "http://www.ubc.ca/stat/blang/BlangDsl"

import "http://www.eclipse.org/xtext/common/JavaVMTypes" as jvmTypes
import "http://www.eclipse.org/Xtext/Xbase/XAnnotations" as xAnnotations

BlangModel:
    {BlangModel}
    ('package' package=QualifiedName)?
    importSection=XImportSection?
    (Annotations += XAnnotation)*
    'model' name=ID '{'
    (variableDeclarations += VariableDeclaration)*
    'laws' '{' (lawNodes += LawNode)* '}'
    ('generate' '(' generationRandom = ValidID ')') generationAlgorithm =
XBlockExpression )?
    (variableDeclarations += VariableDeclaration)*
    '}'
;
...
...
```

# Workflows

- Blang IDE:

- leverages static types (eg smart links, call/type hierarchies, refactoring)



- debugging, profiling, code coverage analysis, etc

- Web IDE

- Command line

SILICO

Commands - Run cached model - Clone History - studentbouchard2 Sign out

Input

```
1001011  
0100010  
1010101  
- new-test
```

Code

```
new-test  
- MyMix.bl  
- configuration.txt  
- data.csv
```

Output

```
new-test  
results  
std_out_err.txt
```

17 random List<RealVal> means 7: listOfRealVals(2),  
18 variances 7: listOfRealVals(2)  
19 param Matrix concentration 7: denseCopy([1.0, 1.0])  
20  
21 lows {  
22 pi | concentration ~ Dirichlet(concentration)  
23 // priors on each mixture component mean and variance  
24 for (int mixIdx : 0 ..< means.size) {  
25 means.get(mixIdx) ~ Normal([0.0], [1.0])  
26 variances.get(mixIdx) ~ Gamma([1.0], [1.0])  
27 }  
28  
29 for (int obsIdx : 0 ..< observations.size) {  
30 // prior over mixture indicators  
31 clusterIndicators.get(obsIdx) | pi ~ Categorical(pi)  
32 // likelihood:  
33  
34  
35 Pass 0 (computed 0 moves so far)  
Pass 1 (computed 1005 moves so far)  
Pass 2 (computed 2010 moves so far)  
Pass 3 (computed 3015 moves so far)  
Pass 4 (computed 4020 moves so far)  
Pass 5 (computed 5025 moves so far)  
Pass 6 (computed 6030 moves so far)  
Pass 7 (computed 7035 moves so far)  
Pass 8 (computed 8040 moves so far)  
Pass 9 (computed 9045 moves so far)  
Pass 10 (computed 10050 moves so far)  
Pass 11 (computed 11055 moves so far)  
Pass 12 (computed 12060 moves so far)  
Pass 13 (computed 13065 moves so far)

# Reproducibility & dissemination

- Code fully deterministic
  - even in multithread mode
  - checked via test units
- Create versioned packages containing models/samplers that others can import
  - built on Maven
  - smart dependency resolution

dependencies.txt

```
ca.ubc.stat:nowellpack:1.0.5  
ca.ubc.stat:conifer:2.0.4
```

```
+--- ca.ubc.stat.blang:ca.ubc.stat.blang:3.25.2  
+--- org.eclipse.jdt:org.eclipse.jdt.core:3.10.0  
|   +--- org.eclipse.core:org.eclipse.core.resources:3.7.100  
|   |   +--- org.eclipse.core:org.eclipse.core.expressions:3.4.300  
|   |   \--- org.eclipse.core:org.eclipse.core.runtime:3.7.0  
|   |       +--- org.eclipse.osgi:org.eclipse.osgi:3.7.1  
|   |       +--- org.eclipse.equinox:org.eclipse.equinox.common:3.6.0  
|   |       +--- org.eclipse.core:org.eclipse.core.jobs:3.5.100  
|   |       \--- org.eclipse.equinox:org.eclipse.equinox.common:3.6.0  
|   +--- org.eclipse.equinox:org.eclipse.equinox.registry:3.5.101  
|   \--- org.eclipse.equinox:org.eclipse.equinox.common:3.6.0  
+--- org.eclipse.equinox:org.eclipse.equinox.preferences:3.4.1  
|   +--- org.eclipse.equinox:org.eclipse.equinox.common:3.6.0  
|   +--- org.eclipse.core:org.eclipse.core.contenttype:3.4.100  
|   |   +--- org.eclipse.equinox:org.eclipse.equinox.preferences:3.4.1 (*)  
|   |   +--- org.eclipse.equinox:org.eclipse.equinox.registry:3.5.101 (*)  
|   |   \--- org.eclipse.equinox:org.eclipse.equinox.common:3.6.0  
+--- org.eclipse.core:org.eclipse.equinox.app:1.3.100  
+--- org.eclipse.core:org.eclipse.core.filesystem:3.6.0  
+--- org.eclipse.equinox:org.eclipse.equinox.common:3.6.0  
|   +--- org.eclipse.osgi:org.eclipse.osgi:3.7.100  
|   \--- org.eclipse.core:org.eclipse.core.runtime:3.7.0 (*)  
+--- org.eclipse.core:org.eclipse.core.filesystem:3.5.101 (*)  
+--- org.eclipse.core:org.eclipse.core.runtime:3.7.0 (*)  
+--- org.eclipse.core:org.eclipse.core.filesystem:1.3.100 (*)  
+--- org.eclipse.text:org.eclipse.text:3.5.101  
|   +--- org.eclipse.core:org.eclipse.core.commands:3.6.0  
|   \--- org.eclipse.equinox:org.eclipse.equinox.common:3.6.0  
+--- org.eclipse.xtext:org.eclipse.xtext:2.12.0  
|   +--- org.eclipse.xtext:org.eclipse.xtext.util:2.12.0  
|   |   +--- org.eclipse.xtext:org.eclipse.xtext.lib:2.12.0  
|   |   |   \--- com.google.guava:guava:[14.0,19) -> 19.0  
|   |   \--- org.eclipse.xtext:org.eclipse.xtext.lib.macro:2.12.0  
|   |       \--- org.eclipse.xtext:org.eclipse.xtext.xbase.lib:2.12.0 (*)  
|   +--- log4j:log4j:1.2.16  
+--- org.eclipse.emf:org.eclipse.emf.common:[2.10.1,3) -> 2.15.0  
+--- org.eclipse.emf:org.eclipse.emf.ecore:[2.10.2,3) -> 2.15.0  
|   \--- org.eclipse.emf:org.eclipse.emf.common:[2.15.0,3.0.0) -> 2.15.0  
+--- com.google.inject:guice:3.0  
|   +--- javax.inject:javax.inject:1  
\--- aopalliance:aopalliance:1.0
```

# Documentation

- Available at <https://www.stat.ubc.ca/~bouchard/blang/>
- Getting started & a few examples
- Setting up workflows
- Complete syntax
- Reference for SDK distributions, engines, tests, io, etc

## Discrete

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

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

Binomial: A sum of  $n$  iid Bernoulli variables. Values in  $[0, n]$ .

▪ **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  $\{0, 1, 2, \dots, n - 1\}$ .

▪ **param SimplexProbabilities**: 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).  $M \in [m, \infty)$

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

▪ **param RealVar mean**: Mean parameter  $\lambda$ ,  $\lambda > 0$

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 result).  $r > 0$

▪ **param RealVar p**: Probability of success of each experiment.  $p \in (0, 1)$

YuleSimon: An exponential-geometric mixture.

▪ **param RealVar rho**: The rate of the mixing exponential distribution.

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  $|K|$

GeneralizedBinomial: Generalized Binomial Distribution with Beta priors

▪ **param IntVar numberOfTrials**: The number  $n$  of Bernoulli variables being

## Continuous

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

▪ **param RealVar min**: The left end point  $m$  of the interval.  $m \in (m, M)$

▪ **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  $\lambda$ , inversely proportional to the mean.  $\lambda > 0$

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

▪ **param RealVar mean**: Mean  $\mu \in \mathbb{R}$

▪ **param RealVar variance**: Variance  $\sigma^2, \sigma^2 > 0$

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.  $\beta > 0$

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

▪ **param RealVar shape**: The shape  $\alpha$  is proportional to the mean and variance.  $\alpha > 0$

▪ **param RealVar rate**: The rate  $\lambda$  is inverse proportional to the mean and quadratically inverse proportional to the variance.  $\lambda > 0$

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

▪ **param RealVar nu**: The degrees of freedom  $\nu, \nu > 0$

▪ **param RealVar mu**

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

▪ **param RealVar nu**: A degree of freedom parameter  $\nu, \nu > 0$

▪ **param RealVar sigma**: A scale parameter  $\sigma, \sigma > 0$

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

▪ **param IntVar nu**: The degrees of freedom  $\nu, \nu > 0$

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

▪ **param RealVar location**: The mean parameter.

▪ **param RealVar scale**: The scale parameter  $b$ , equal to the square root of half of the variance.  $b > 0$

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

▪ **param RealVar location**: The center of the PDF. Also the mean, mode and median.  $\mu \in \mathbb{R}$

▪ **param RealVar scale**: The scale parameter.  $\sigma > 0$

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

▪ **param RealVar scale**: The scale parameter  $\sigma$  and also the median.  $\sigma > 0$

▪ **param RealVar shape**: The shape parameter  $\beta, \beta > 0$

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  $d_2$ .  $d_1, d_2 > 0$

Weibull: The Weibull Distribution. Values in  $[0, +\infty)$ .

## Overview of the syntax reference

In the following, we assume the reader has already looked at the [Getting started page](#).

This document attempts to be as self-contained as possible. Blang is designed to be usable with very minimal past programming exposure. However, to fully master the advanced features of Blang, familiarity with a modern multi-paradigm statically typed language is recommended.

Additionally, for readers familiar with Java and using the Blang IDE, you can right click on any Blang file and select [Open generated file](#) to see the Java code generated by Blang.

### Types

The information stored by programs into a computer's memory is highly structured into chunks. Each chunk is called an object. Some of the objects have shared properties. A property that allows us to group objects is called a type.

Concretely, types in Blang are equivalent to Java types (a terminology that encompasses Java classes, interfaces, primitives, enumerations and annotation interfaces). This means 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.

### Comments

Single line comments use the syntax `// some comment spanning the rest of the line`.

Multi-line comments use `/ * many lines can go here */`.

In the following, we use comments to give contextual explanation on syntax examples.

### Models

The syntax for Blang models is as follows:

```
1 package my.namespace // optional
2
3 // import statements
4
5 model NameOfModel {
6
7     // variables declarations
8
9     lows {
10        // lows declaration
11    }
12
13    generate(NameOfRandomObject) { // optional
14        // generate block
15    }
16 }
```

## Testing strategies

### Exhaustive tests

We provide a non-standard replacement implementation of `bayonet.distributions.Random` which can be used to enumerate all the probability traces used by an arbitrary discrete random process. In particular, many inference engines' code manipulate models through interfaces that are agnostic to the model being continuous or discrete, so we can achieve code coverage of the inference engines using discrete models. See `bayonet.distributions.ExhaustiveRandom`.

We use this for example to test the unbiasedness of the normalization constant estimate provided by our SMC implementation.

```
1 package Blang.validation
2
3 import java.util.function.Supplier
4 import bayonet.distributions.ExhaustiveDebugRandom
5
6 class UnbiasednessTest {
7     def static double expectedZEstimate(Supplier<Double> log2Expectation) {
8         var expectation = 0.0
9         var nProgramTraces = 0
10        while (ExhaustiveRand.hasNext()) {
11            val log2 = log2Expectation.get()
12            expectation += Math.exp(log2) * exhaustiveland.lastProgramTraceProbability
13            nProgramTraces++
14        }
15        println("nProgramTraces = " + nProgramTraces)
16        return expectation
17    }
18 }
```

This can be called with a small finite model, e.g. a short HMM here, but making it large enough to achieve code coverage (Blang and the BlangDl are compatible with the ecmcmc code coverage tool).

The output of the test has the form:

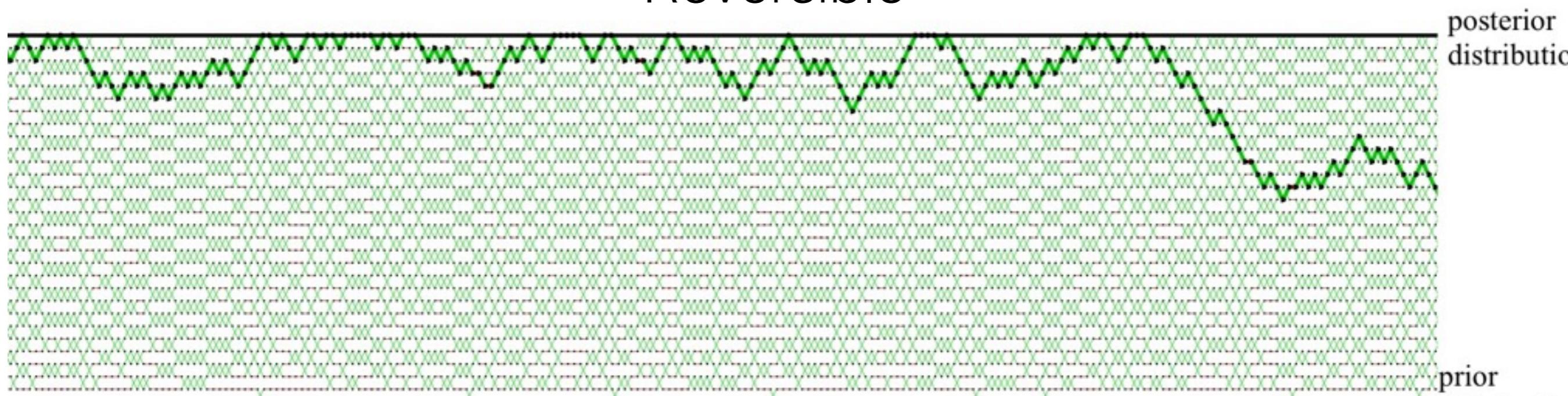
```
1 nProgramTraces = 23864
2 true normalization constant Z: 0.345
3 expected Z estimate over all traces: 0.34500000000000564
```

Showing that indeed our implementation of SMC is unbiased.

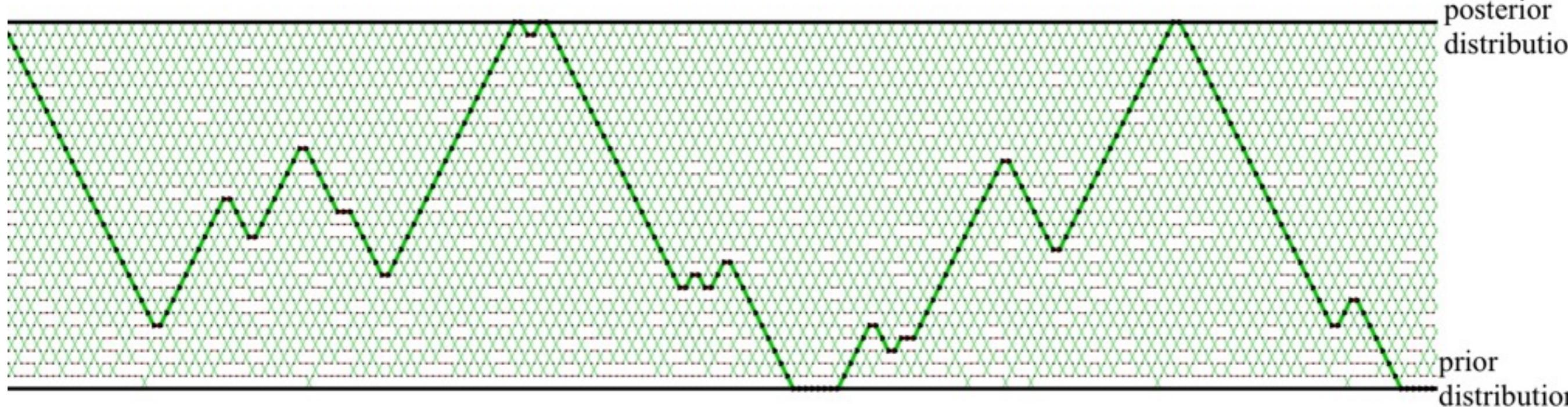
This can also be used to test numerically that transition probabilities of small state

# Visualization: paths

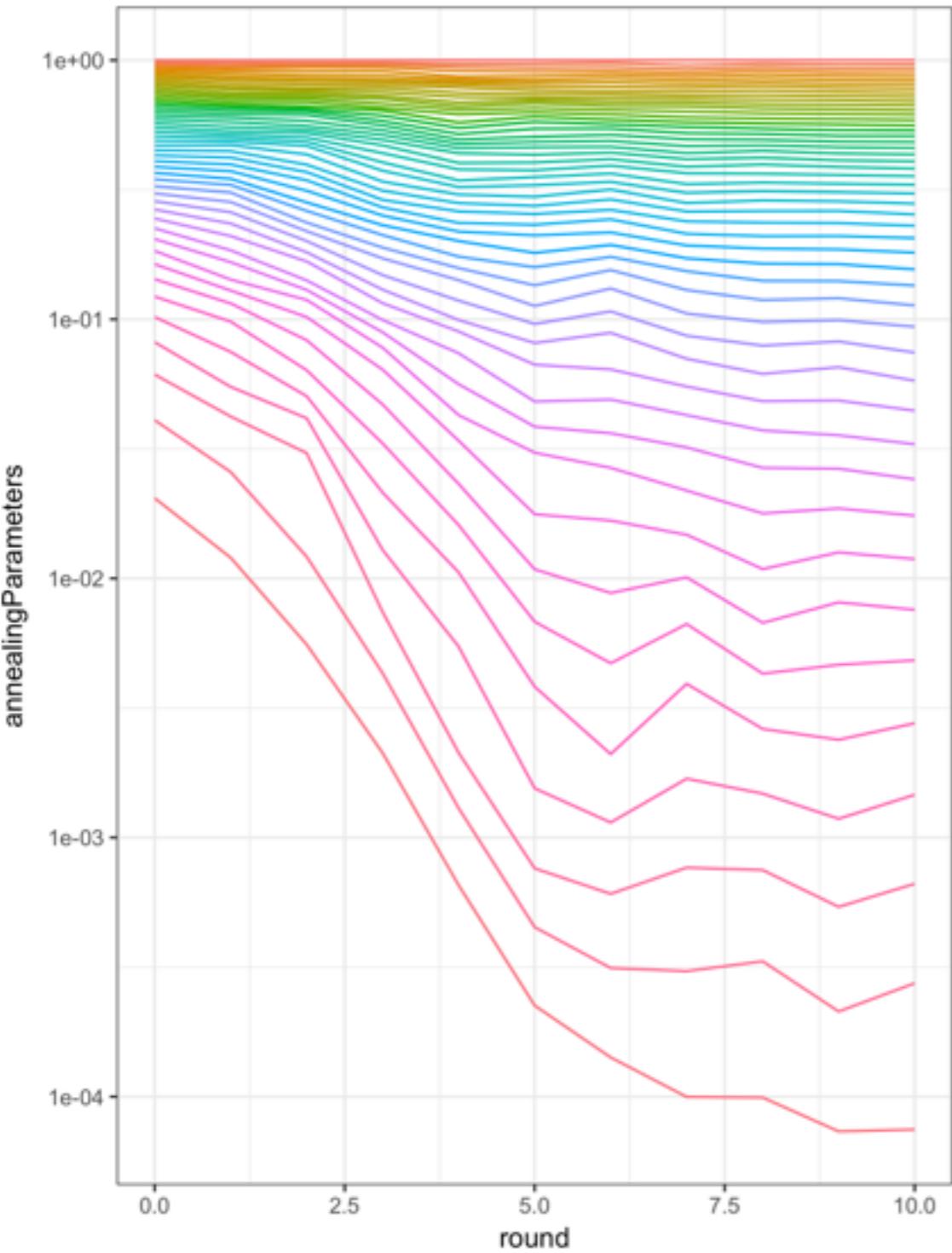
Reversible



Non-reversible



# Visualization: adaptation



factor(chain)

0	17	34
1	18	35
2	19	36
3	20	37
4	21	38
5	22	39
6	23	40
7	24	41
8	25	42
9	26	43
10	27	44
11	28	45
12	29	46
13	30	47
14	31	48
15	32	
16	33	

