Yet Another Simulation Language

Nikhil Bhat (nb2490)

September 25, 2013

1. Introduction

Yet Another Simulation Language (YASL) aims to be a simple intuitive language for to perform Monte Carlo simulation. Simulation is widely used to analyze behavior of systems which have unpredictable behavior. For example, stock price fluctuations are unpredictable and hence they are studied using mathematical formalism of random variables. Further to make predictions about these events aggregate statistics such and the expected value and higher order moments are analyzed. This is essentially performing integrations over the probability distributions. These can be done in the following way:

- **Analytical:** Use calculus to find 'closed form' solution to the integration problem. E.g. suppose head = 1 and tail = 0 in a fair coin, the average reward is $0.5 = 0.5 \times 1 + 0.5 \times 0$.
- **Simulation:** Draw samples of the random variable and average. E.g. we can flip a sequence of coins in this made up example and find the average. As the number of samples increases, the average is likely to be close to 0.5.

The first approach will typically yield faster algorithms but can be applied to relatively simple situations. Indeed, a lot of research goes into finding closed form solutions to such integration problems. The second approach is slower but also unavoidable for complex scenarios. Simulation is a mature field of applied mathematics and clever ways to simulate distributions have been devised.

YASL will be a simple language to perform simulation of multivariate distributions. In this language one will be able to use some well known distributions as building blocks to compose more complex random variables. The following code gives a taste of YASL:

```
Random normal = new Normal(mean = 0.0, std = 1.0);
Random binom = new Binomial(n = 10, p = 0.2);
Generator MyExp(Float lmd; Random U = new Uniform(0.0, 1.0)) : Random {
return -ln(1 - U) /. lmd;
}
Random myexp = new MyExp(lmd = 3.0)
Random fancy = new MyExp(lmd = abs(normal))

Func g(Float x, Float y) : Float =  {
return x**.3 +. exp(-y);
}

Random fancy2 = g(normal, binom);

Stat ThirdMom(Random XX) : Float {
```

```
return E(XX**.3);
}

Matrix mat = [[0.1, 0.3],[0.3, 1.0]];
Matrix vec = [[0.2, 0.1]];
Random[] normalvec = new NormalMult(mean = vec, covar = mat);

Simulate
(seed = 123, trials = 10000)
{
E(fancy);
PR(fancy2 in [10, inf]);
Var(normal);
E(fancy + fancy2 | binom = 3);
ThirdMom(fancy);
PR(normalvec[1] in [0.0, 1.0]);
}
```

2. Language Sketch

2.1. Primitive Types

The language will have the following primitive types. We divide the types into random and non-random types in the obvious way.

Type	Description
`Int`	Integers
`Float`	Floating point numbers
`Boolean`	True or False
`Matrix`	A two dimensional array of Floats
`Random`	Random number of Float type
`RandInt`	Random number of Int type
`RandBool`	Random Boolean
`RandMatrix`	Random Matrix
`T[]`	An array of a primitive type T, except an array

2.2. Functions

Functions take a tuple of primitive types and return a single primitive type. For example the function `g` mentioned in the code is of the type `Float` $\times$ `Float` $\rightarrow$ `Float`. We can 'infect' these function to change their type. For example when we do `g(X, Y)` where `X` and `Y` are `Random`, the output is also `Random`. But if X is a `RandInt`, we get a compiler error.

2.3. Generators

The keyword `Generator`is used to generate new source of randomness. Apart from the arguments the generators have randomness sources. The arguments and The language will provide certain built in `Generator`s, for example `Uniform` and `Normal`. The `Generator`s can take primitive types

as arguments. In a normal use the arguments will be of non-random parameters. But nothing stops you from using a `Random` instead of a `Float`, etc. Generators must return random types.

As an aside. we note that as long as we have the `Uniform` generator and functions, we do not really need `Generator`s. These can be thought of as syntactic sugar. For example look at the `MyExp` generator. This is how we would write a Generator for the exponential random number. The arguments usually have some semantic significance and hence Generators can be called with named arguments.

2.4. Stats

`Stat`s are the opposite of `Generator`s in that these are used to average over the randomness. For example look at the `ThirdMom Stat`. We build on the built in `Stat E` to build more complex ones. `Stat` must return non-random types.

2.5. Control Flow

The YASL will have C-like `if, for` and `while` control flow tools.

2.6. The `E` and `Pr` Stats

`E` is a special operator which takes any primitive type and computes the average for it by Monte Carlo. In addition we can take conditional expectations. By doing something like `E(X | Y = a)`. Similarly we have the `Pr` operator which can be used with the `in` keyword. Again, note that `Pr` keyword is not necessarily needed since we can do something like `E(I(X, 0, inf))` instead of `Pr(X in [0, inf])`, where `I` is 1 if `X` is non-negative.

2.7. Simulate Block

Finally using the `Simulate` block we can run the Monte Carlo to compute some statistics of interest. The `Simulate` has a block with parenthesis which can be used to specify the simulation parameters and then a block which lists properties of interest. All there statements must be calls to various stats. The output can be printed on command line or a file.