RiskModelica User's Guide

Michael Spiegel

May 1, 2008

Contents

1 Installation

In order the perform the RiskModelica installation, you must have a copy of the file jmodelica.jar. The archive file contains both the compiled RiskModelica compiler as well as additional Python scripts needed to run the RiskModelica execution infrastructure. If you want to run the SCEM-UA calibration module, you must install the following Python libraries: SciPy, NumPy, and Matplotlib. These libraries can be found online at: `http://www.scipy.org` and `http://matplotlib.sourceforge.net`. In order to use the Windows Dymola compiler and run RiskModelica on a Linux system, you must have Wine installed. Wine is an open source implementation of the Windows API on top of Unix. Wine can be found online at: `http://www.winehq.org`. We recommend running Dymola binaries under Wine because Dymola 6.0b produces executables that trigger a memory leak in the Windows OS (ie. memory is not reclaimed once the process dies). Wine will faithfully execute the executable under Linux without triggering a memory leak in the OS. On a Fedora Core Linux distribution you can install the required libraries using the packages numpy, scipy, lapack-devel, python-matplotlib, and wine. You must have the Java Runtime Environment (Java

version 1.4 or later) and the Python Programming Language (Python version 2.4 or later). An implementation of Java can be downloaded from `http://java.sun.com`, and Python can be downloaded from `http://www.python.org`.

1.1 Installing Python Scripts

Use the following command to extract the contents of jmodelica.jar:

```
jar xf jmodelica.jar
```

Copy the contents of the /python subdirectory into a location that you wish to serve as the RiskModelica library directory. After you have copied the /python subdirectory you can delete all the extracted files, but make sure to keep the archive file. For the purposes of this user guide, we will assume you have chosen the location /home/user/lib/riskmodelica as your RiskModelica library directory. Create an environment variable $RISKMODELICA_LIBS that points to this location.

2 Usage

We will assume that you are using the Dymola compiler version 6.0b or later for Windows. Create two versions of the Modelica source that you wish to compile. One version will include the RiskModelica language extensions to perform uncertainty analysis or model calibration. The second version is a plain Modelica variant of your RiskModelica program. Create the second version by removing all the RiskModelica features from your RiskModelica program. If you elide the RiskModelica keywords from a RiskModelica program you obtain a Modelica program, called the *deterministic elision* or *Modelica elision* of the RiskModelica program. The deterministic elision of a RiskModelica program is always a valid deterministic implementation in Modelica of the RiskModelica program. This concept is similar to the idea of serial elision in the parallel programming language Cilk [5].

You will compile the Modelica version using the Dymola compiler. Open the Modelica version in Dymola and select the "Simulate" tab (in the bottom right corner) to begin the compilation process. Under the "Simulation" window, select the "Setup..." dialog. Fill in the appropriate start time and stop time for your simulation. Under the "Output" tab make sure the "textual data format" and "double precision" options are selected. Under the "Simulation" window, select the "Translate" option to compile your Modelica program. In the filesystem you will see the files dymosim.exe and dsin.txt have been created. These files will be used to run a RiskModelica program.

You will compile your RiskModelica program using jmodelica.jar and the following command:

```
java -jar jmodelica.jar SampleModel.rmo SampleModel
```

The first argument, SampleModel.rmo, is the filename you wish to compile and the second argument, SampleModel, is the name of the model within the file you wish to execute. You may ignore any warning messages generated by the RiskModelica compiler. You will see the compiler has generated a python script named riskSampleModel.py in the filesystem. Run this script (using "python riskSampleModel.py") to execute your RiskModelica program.

a	=	{public, protected}	Access
c	=	{input, output, inputoutput}	Causality
f	=	{flow, nonflow}	Flow prefix
m	=	{replaceable, modifiable, final}	Modifiability
v	=	{constant, parameter, discrete, continuous}	Variability
o	=	{outer, inner, notouterinner}	Outer/Inner
p	=	{deterministic, sampling, calibration}	Program mode
r	=	{uncertain, measurement, certain}	Risk prefix

Figure 1: Type prefix classes in Modelica (upper region) and RiskModelica (lower region). Default prefixes are underlined.

3 RiskModelica Language

The RiskModelica language is a superset of the Modelica language. For an introduction to Modelica, we recommend the book "Principles of Object-Oriented Modeling and Simulation with Modelica 2.1" [6]. RiskModelica extends Modelica with the four type prefixes `uncertain`, `measurement`, `calibration`, and `sampling`. The `uncertain` type modifier is used to designate model parameters that have values drawn from either precise or imprecise probability distributions. The `measurement` type modifier denotes model output variables for which data sets are available in order to perform model calibration. The `calibration` and `sampling` keywords specify the start of equation blocks that are used for model calibration or uncertainty analysis, respectively. A RiskModelica model may have at most either one `calibration` equation block or one `sampling` equation block. We say that a RiskModelica program is running either in calibration mode or sampling mode, depending on which equation block is present. A model that has neither type of equation block is identical in program semantics to its deterministic elision. Uncertain variables can occur in either program mode although the semantics are slightly different depending on the current program mode (see section 3.1). Measurement variables can only occur in calibration mode; they do not have meaning when the program is running in sampling mode.

RiskModelica introduces two classes of type prefixes to the existing system of type prefix in Modelica. We refer the reader to "Types in the Modelica Language" [2] for a complete discussion on types in Modelica. A type qualifier is a form of subtyping where a supertype `T` is combined with a qualifier `Q` such that some semantic property is enforced on all instances of the subtype `Q T` [4]. Some canonical examples of type qualifiers are `const` and `volatile` from the ANSI C standard. A variable declared with the type qualifier `const` can be initialized but it cannot be modified. The type qualifier `volatile` means that the value of the instantiated variable can change asynchronously of the current execution thread. Type qualifiers can be used as a mechanism for implementing a variety of compile-time and runtime semantic properties. RiskModelica extends the rich type qualifier system available in the Modelica language. Modelica contains twelve type qualifiers, preserving semantic properties at compile-time such as encapsulation [`public`/`private`], causality [`input`/`output`], and variability [`constant`/`parameter`/`discrete`], etc.

Figure 1 extends figure 8 of Broman *et al.* with the additional type prefixes of RiskModelica. Following the convention of the original paper, we have introduced the new prefixes

`deterministic` and `certain` for the sake of completeness. These are not real type prefixes of the language, but rather they signify using the lack of a prefix for that category. The default prefixes for any category are marked by an underline. The horizontal separator is used to distinguish Modelica prefixes from RiskModelica prefixes.

Unlike Modelica, the RiskModelica type prefixes have context-sensitive restrictions on usage that extend beyond the object type and class type restrictions observed in Broman *et al.* The selection of program mode limits the set of available risk prefixes. More formally, there exists a function *program* that maps the choice of program modes to a set of allowable risk prefixes. This function is defined as follows:

$$
\begin{aligned}
&program: \ p \rightarrow 2^r \\
&program(\texttt{deterministic}) &&= \{\texttt{certain}\} \\
&program(\texttt{sampling}) &&= \{\texttt{uncertain}, \texttt{certain}\} \\
&program(\texttt{calibration}) &&= \{\texttt{uncertain}, \texttt{measurement}, \texttt{certain}\}
\end{aligned}
$$

Additionally the selection of variability limits the set of available risk prefixes. There exists a function *variable* that maps the choice of variability to a set of allowable risk prefixes. Note that program mode is a property defined on a whole program, whereas state variability is selected for each variable declaration. *variable* is defined as follows:

$$
\begin{aligned}
&variable: \ v \rightarrow 2^r \\
&variable(\{\texttt{continuous}, \texttt{discrete}\}) &&= \{\texttt{measurement}, \texttt{certain}\} \\
&variable(\texttt{parameter}) &&= \{\texttt{uncertain}, \texttt{certain}\} \\
&variable(\texttt{constant}) &&= \{\texttt{certain}\}
\end{aligned}
$$

In order to determine the set of allowable risk prefixes on a particular variable declaration, it is necessary to take the intersection of $program(p)$ and $variable(v)$.

3.1 Uncertain Parameters

The `uncertain` keyword is a type modifier than can be used on parameters for a program in either sampling mode or calibration mode. `uncertain` signifies that the parameter cannot be represented by a single discrete value. Instead the parameter is represented by either a crisp probability distribution or an imprecise distribution. When a program is in sampling mode, it is required that the shape of a parameter's precise or imprecise distribution is specified in the parameter definition:

```
uncertain(expr) parameter Real temperature = 98.6;
```

Once the program semantics for sampling mode have been defined (see Section 3.4), we will be able to describe the valid types for ***expr***. At the moment these semantics are undefined. When a program is in calibration mode, it is not required that the shape of the probability distribution is specified. This is because the shape of the prior and posterior distributions of a parameter are determined by the selection of a calibration algorithm. The selection of a calibration algorithm is performed in the `calibration` equation block using the

pre-declared variable "method" (see Section 3.3.1). In both program modes, the deterministic elision of a RiskModelica program results in a parameter that is represented by a single value. Following Modelica semantics, an equality statement at the variable's declaration leads to that value for the parameter (98.6 in our example). No equality statement leads to a default value for the parameter based on the variable's type. The equality statement is ignored in the RiskModelica program; it is a language component that exists solely for the deterministic elision.

3.2 Measurement Variables

The `measurement` keyword is a type modifier than can be used on either continuous or discrete variables for a program in sampling mode. `measurement` signifies that the program variable can be mapped to some external time series of data. This data set is compared to the variable output to evaluate model fitness. Model fitness is transformed into some likelihood measure in order to calibrate the `uncertain` parameters. It is relatively straightforward to declare `measurement` variables:

```
measurement Real temperature(start = 98.6);
```

As discussed in the next section, the external time series of data must be stored as sequences of (time, value) tuples. All Modelica programs use a logical time system of continuous values that starts at logical $time = 0.0$. Oftentimes we want to run our simulation for different durations of referent time. Therefore the datasets store referent time along with the values at those time points. We must use a variable in our RiskModelica program that maps logical time into referent time. This is specified by the pre-declared variable "timeField" in the `calibration` equation block. Now the mapping from logical time into referent time is specified only once in the program, and all the datasets can remain ignorant of program logical time.

3.3 Calibration Mode

Model calibration through Bayesian inference is a technique for updating *a priori* probability distributions based on the observation of data that are relevant to the variables of interest [1, 8]. Let us describe a generic application of Bayesian inference by assuming a model $y = f(x|\theta)$, where x is vector of input quantities to the model, θ is a vector of scalar parameters to the model, and y is vector of observable output quantities. The *a priori* beliefs about θ are expressed using a probability distribution function $p(\theta)$. On the basis of independent observations $X = (X_1, X_2, ..., X_n)$, the probability distribution function is then updated according to Bayes' Law: $p(\theta|X) = \frac{p(\theta)p(X|\theta)}{p(X)} = \frac{p(\theta)p(X|\theta)}{\int p(\theta)p(X|\theta)d\theta}$. For a fixed data set X, we may view the probability function $p(X|\theta)$ as a function of of θ instead of as a function of X. Under these circumstances the probability is viewed as the likelihood function and is written as $l(\theta|X)$. Additionally for a fixed data set, the probability $p(X)$ is independent of the choice of θ. Given these facts the posterior distribution is proportional to the likelihood function multiplied by the prior distribution: $p(\theta|X) \propto l(\theta|X)p(\theta)$.

When calibrating probability distributions through black-box simulation models, it is impossible or infeasible to analytically determine the posterior distribution. In these cases we must iteratively sample the posterior distribution in order to approximate its true shape. In order to numerically calculate high-dimensional integrals on large parameter spaces, a Markov chain Monte Carlo (MCMC) integration technique can be used. MCMC integration techniques are often more efficient than traditional Monte Carlo techniques in high-dimensional parameter spaces.

Calibration mode has been implemented using a modular interface that allows different calibration techniques to be defined and used. The employment of a MCMC algorithm is distinguished by the choice of a transition kernel that moves from one location to the next in the parameter space: $\theta^{t+1} \propto z(\theta|\theta^t)$. Currently only one calibration technique has been implemented, the shuffled complex evolution metropolis algorithm. This MCMC sampling technique is described in section 3.3.2.

Model calibration is enabled in RiskModelica by defining a calibration equation block in a model. A calibration equation block is an equation block that is declared by using the `calibration` keyword. Calibration equation blocks are used to state directives to the RiskModelica execution engine on how to perform model calibration. These calibration blocks are different from regular equation blocks in several aspects. First, calibration equation blocks cannot contain discrete-event statements (`when` statements) or the derivative operator. Second, all values used in calibration blocks must have a variability of either constant or parameter. In the current RiskModelica implementation, all values used in calibration models must have constant variability. Third, there are a number of variables that are declared by the RiskModelica language and to which the user must specify values. These variables exists in one of two categories: common variables that must be defined for all calibration schemes, and calibration scheme-specific variables that are different for each calibration algorithm. The common variables are defined in the next subsection.

The variable measurements specifies an array of filenames that will be used as datasets in the calibration scheme. Each file contains a set of (a,b) data points with one data point per line. The left column in the file indicates a time stamp for that measurement point. This time stamp must match up with the timeField variable that is specified in the model. The right column is the actual value of that measurement corresponding to the time stamp. All measurement files are ASCII text files where columns are tab deliminated, and rows are newline deliminated.

3.3.1 Common Variables

Name	Type	Value
method	`String`	The calibration module to use. Current valid strings: "SCEM-UA"
timeField	`String`	The variable name that is used to match timestamps on the measurement data sets. If the time field should be set to logical time, this can be specified using the Modelica built-in variable "time".

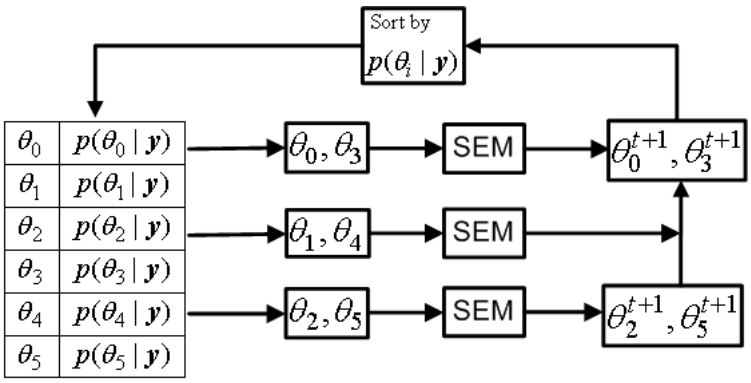

Figure 2: Shuffled Complex Evolution Metropolis (SCEM-UA) algorithm. Samples θ_i are stored in order of decreasing posterior density $p(\theta_i|\mathbf{y})$. Here the sample size is 6 and the complex size is 2 (resulting in $6/2 = 3$ complexes).

Name	Type	Value
executionPath	`String`	The absolute path to the directory containing the Modelica executable.
executableName	`String`	Assuming the working directory has been set to executionPath, the command string that will run the Modelica executable.
measurements	`Array` of `Strings`	Each String is a relative location (from executionPath) to one of the data sets used for calibration. The size of measurements should be equal to the number of `measurement` variables in the program. The order of elements in this array must match the order of `measurement` variables as they appear in the model.

3.3.2 SCEM-UA Algorithm

The shuffled complex evolution metropolis algorithm (SCEM-UA) is an adaptive sampling technique that uses covariance-based evolutionary selection for estimating the posterior distribution of a set of parameters [10]. Here we extend our generic model from the previous section into the form $y = f(x|\theta) + e$ with the addition of e, a vector of statistically independent errors with zero expectation and constant variance σ^2. SCEM-UA is an adaptation of the shuffled complex evolution global optimization algorithm [3] to a Bayesian scheme where parameters are represented as probabilities, instead of as a single optimal set of values.

An overview of the SCEM-UA algorithm is shown in Figure 2. A total of s samples from the parameter space θ are stored in order of decreasing posterior density $p(\theta|\mathbf{y})$. The sample points are partitioned into q complexes with the expectation that $q \ll s$. Partitioning is done by striping the total population across the complexes. More precisely, complex c contains all samples numbered i such that $i \bmod q = c$. After the samples have been divided into complexes, each complex is evolved independently using the sequence evolution metropolis (SEM) algorithm described in Algorithm 1. After the SEM algorithm completes,

all the updated complexes are combined together and stored in order of decreasing posterior density. This process repeats until the Gelman and Rubin convergence statistic is satisfied. The Gelman and Rubin convergence statistic [7] is a measure of both within and between complex variation. It is defined as $\sqrt{SR} = \sqrt{\frac{g-1}{g} + \frac{q+1}{q \cdot g}\frac{B}{W}}$, where g is the number of iterations within each complex, B is the variance between the q complex means, and W is the average of the q within-complex variances for the parameter under consideration. An $\sqrt{SR}$ score near unity for all parameters indicates convergence. However, because a score of unity is difficult to achieve, Gelman and Rubin recommend using a value of 1.2 or less to declare convergence.

The SEM algorithm contains the transition kernel, $\theta^{t+1} \propto z(\theta|\theta^t)$, that moves from one location in the parameter space to the next. This algorithm takes as input a complex, C, and a sequence, S. Both C and S have cardinality $m = s/q$. C contains the m current best samples for a complex. S contains the m most recent samples that have been accepted or rejected as candidates for C. The size of S is fixed to the size of C. Therefore when a new element is inserted into S (such as line 24 of the algorithm), then the oldest element of S is eliminated. At the start of the SCEM-UA algorithm, C and S contain the same elements.

Candidate points for S are generated using an adaptive multinormal distribution with mean and covariance structure identical to the current sequence S. However, if the mean posterior density of S is significantly smaller than the mean posterior density of C, then the center of the multinormal distribution is temporarily switched over to the mean of C (lines 9-12). This is to reduce the chances that S gets stuck in poor regions of the parameter space. After a candidate point is generated, it is accepted or rejected using the Metropolis-annealing criterion [9]. If a candidate point is accepted, then replace the best member of C with the new point. If a candidate point is rejected, then rollback to the most recently accepted candidate point in the search space (line 19). If (a) a rollback occurs, and (b) the posterior probability of the candidate point is greater than the worst member of C, and (c) the ratio of best member of C to worst member of C is greater than the predefined likelihood ratio, T, then replace the worst member of C with the candidate point (line 21). When the ratio of the best member of C to the worst member of C is too large, then the worst member of C should be replaced to facilitate convergence of the algorithm. Repeat the process of generating candidate points L times, where L is the number of evolution steps taken by each complex before returning to the SCEM-UA algorithm.

3.3.3 SCEM-UA Variables

Name	Type	Value
q	`Integer`	The number of parallel complexes to evolve.
s	`Integer`	The total population size.
ndraw	`Integer`	Maximum number of simulation executions.
gamma	`Real`	The error model of the residuals. The residuals are assumed normally distributed when $\gamma = 0$, double exponential when $\gamma = 1$, and tend to a uniform distribution as $\gamma \to -1$.

Name	Type	Value
Sigma	`Real`	Measurement error (standard deviation) in measured data sets.
option	`Integer`	If the value is 1 then the simulation directly computes posterior density. Otherwise if the value is 2 then we are assuming the residuals are mutually independent, Gaussian distributed, and with constant variance. The likelihood is computed using [1] $$L(\theta^{(t)}\|\mathbf{y}) = \exp\left[-\frac{1}{2}\sum_{i=1}^{N}\left(\frac{e(\theta^{(t)})_i}{\sigma}\right)^2\right]$$ If the value is 3 then (in addition to the assumptions from option 2) we are assuming a noninformative prior of the form $p(\theta) \propto \sigma^{-1}$ and the likelihood can be computed using [1] $$M(\theta^{(t)}) = \sum_{i=1}^{N} e\left(\theta^{(t)}\right)_i^2$$
delConvCrit	`Real`	The maximum delta of Gelman and Rubin convergence criteria from one iteration to the next in order to declare convergence.
min	`Array` of `Reals`	An array of lower bounds for the prior uniform distribution on `uncertain` variables. The order of elements in this array must match the order of `uncertain` variables as they appear in the model.
max	`Array` of `Reals`	An array of upper bounds for the prior uniform distribution on `uncertain` variables. The order of elements in this array must match the order of `uncertain` variables as they appear in the model.

3.4 Sampling Mode

Sampling mode has not yet been implemented. When implemented, there will be support for uncertainty analysis using Monte Carlo sampling of traditional crisp probabilities, imprecise probabilities as probability boxes, and imprecise probabilities as Dempster-Shafer belief structures. Probability boxes will be discretized into a finite number of intervals that can be propagated using belief structures. The implementation of Monte Carlo sampling of crisp probabilities will happen first, as there are no outstanding research issues that must be addressed before completing this task. Expect to see more in the fall of 2008.

Algorithm 1 Sequence Evolution Metropolis (SEM) algorithm. C is the set of the current best samples for a complex, and S is the sequence of the most recent samples in this complex, such that $|C| = |S|$.

procedure SEM(C, S)
  $\beta \leftarrow 0$
  while $\beta < L$ **do**
    $\theta^{(t)} \leftarrow S_0$ ▷ S_0 is the most recent element of S
    $\mu \leftarrow$ the mean μ of C
    $\Sigma \leftarrow$ the covariance matrix of C
    $\alpha \leftarrow$ (the mean posterior density of C)/(the mean posterior density of S)
    $\Gamma \leftarrow$ (the highest posterior density of C)/(the lowest posterior density of C)
    if $\alpha < T$ **then**
      $\theta^{(t+1)} \leftarrow multinorm(\theta^{(t)}, c_n^2 \cdot \Sigma)$ ▷ Sample from a multinormal distribution
    else
      $\theta^{(t+1)} \leftarrow multinorm(\mu, c_n^2 \cdot \Sigma)$
    end if
    $\Omega \leftarrow p(\theta^{(t+1)}|\mathbf{y})/p(\theta^{(t)}|\mathbf{y})$
    $Z \leftarrow uniform(0, 1)$ ▷ Sample from a uniform distribution
    if $Z \leq \Omega$ **then**
      Replace the best member of C with $\theta^{(t+1)}$
    else
      $\theta^{(t+1)} \leftarrow \theta^{(t)}$
      if $T < \Gamma$ and $p(\theta^{low}|\mathbf{y}) < p(\theta^{t+1}|\mathbf{y})$ **then**
        Replace the worst member of C with $\theta^{(t+1)}$
      end if
    end if
    $S_0 \leftarrow \theta^{(t+1)}$ ▷ Insert $\theta^{(t+1)}$ as the most recent element of S
    $\beta \leftarrow \beta + 1$
  end while
  return C ▷ Return to SCEM-UA
end procedure

4 Examples

```
model BiscayneBay
    parameter Integer nSubbasins = 2;
    parameter Integer nFluxes = 4;
    uncertain parameter Real evpercent = 0.0;
    uncertain parameter Real[nFluxes] fluxcoefficients =
        {101.246241771378, 0.01, 111.819874311523, 125.648193380631};
    measurement discrete Real[nSubbasins] salinity;
    discrete Real date;
calibration
    method = "SCEM-UA";
    q = 10;
    s = 350;
    nout = 2000;
    ndraw = 10000;
    gamma = 0.0;
    Sigma = 0.1;
    option = 2;
    delConvCrit = 0.001;
    min = {0.1, 0.01, 0.01, 0.01, 0.01};
    max = {1.0, 300.0, 300.0, 0.01, 0.01};
    timeField = "date";
    executionPath = "/home/ms6ep/mastri/cosby/Biscayne Bay report";
    executableName = "wine dymosim.exe";
    measurements = {"data/salinity[1].txt",
                    "data/salinity[2].txt"};
initial equation
...
...
equation
...
...
end BiscayneBay;
```

References

[1] George E. P. Box and George C. Tiao. *Bayesian Inference in Statistical Analysis*. Wiley Classics Library, 1992. Originally published Addison-Wesley Publishing Company 1973.

[2] David Broman, Peter Fritzson, and Sebastien Furic. Types in the Modelica language. In *Proceedings of the 5th International Modelica Conference*, Vienna, Austria, 2006.

[3] Q. Y. Duan, V. K. Gupta, and Soroosh Sorooshian. Shuffled complex evolution approach for effective and efficient global minimization. *Journal of Optimization Theory and Applications*, 76(3):501–521, March 1993.

[4] Jeffrey S. Foster, Manuel Fahndrich, and Alexander Aiken. A theory of type qualifiers. In *Proceedings of SIGPLAN '99 (PLDI)*, 1999.

[5] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1998.

[6] Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1.* Wiley-IEEE Press, 2004.

[7] Andrew Gelman and Donald B. Rubin. Inference from iterative simulation using multiple sequences. *Statistical Science*, 7(4):457–472, November 1992.

[8] Peter M. Lee. *Bayesian Statistics: An Introduction.* Hodder Arnold, third edition edition, 2004.

[9] Nicholas Metropolis, Arianna W Rosenbluth, Marshall N Rosenbluth, Augusta H Teller, and Edward Teller. Equation of state calculations by fast computing machines. *The Journal of Chemical Physics*, 21(6):1087–1092, 1953.

[10] Jasper A. Vrugt, Hoshin V. Gupta, Willem Bouten, and Soroosh Sorooshian. A shuffled complex evolution metropolis algorithm for optimization and uncertainty assessment of hydrological model parameters. *Water Resources Research*, 39(8):1201–1214, 2003.