DEPARTMENT OF COMPUTER SCIENCE
SERIES OF PUBLICATIONS C
REPORT C-1999-36

Bassist User's Guide

For Version 0.8.3

Hannu Toivonen, Heikki Mannila, Jouni Seppänen, and Kari Vasko

UNIVERSITY OF HELSINKI
FINLAND

Bassist User's Guide

Hannu Toivonen, Heikki Mannila, Jouni Seppänen, and Kari Vasko

University of Helsinki,
Department of Computer Science and Rolf Nevanlinna Institute
PO Box 26, FIN-00014 University of Helsinki, Finland
Email: Hannu.Toivonen@cs.helsinki.fi
Report C-1999-36, June 1999, ii+21 pages

Abstract

Bassist is a tool for Bayesian statistical analysis. Given a high-level specification of a full probability model, Bassist can be used to obtain an approximate sample from the joint posterior distribution of the model parameters and missing data. Bassist is based on Markov chain Monte Carlo (MCMC) techniques, especially the Metropolis-Hastings-Green method.

The model specification language of Bassist is declarative. The design of the language emphasizes clarity in the model description by separating between the model specification, the simulation details, and the particulars of the data. Within the Bassist system, the model specifications are first compiled into a C++ program, which then reads the data and performs the actual simulation.

Bassist is publicly available on the World Wide Web at `<http://www.rni.helsinki.fi/cs/bassist/>`.

Computing Reviews Categories and Subject Descriptors:
D.1.2 Automatic programming
G.3 Probability and Statistics
I.6.7 Simulation Support Systems
I.6.8 Types of Simulation: Monte Carlo

General Terms:
Algorithms, Experimentation

Additional Key Words and Phrases:
Bayesian probability models, MCMC simulation, statistical modeling

Contents

1 Frequently answered questions

What is Bassist? Bassist is a system for Bayesian statistical analysis. It compiles a high-level specification of a full probability model to an executable program: given data, the program generates an approximate sample from the posterior distribution of the model parameters and missing data.

Who is Bassist meant for? Bassist is suited for statisticians who work with Bayesian models, possibly complex ones, and who value a system that automates the implementation of the models.

Bassist can be seen as a high-level programming tool for writing and implementing Bayesian models. Bassist simply does what it is told to do, and it is the modeler's responsibility to tell Bassist do the right things. An understanding of Bayesian modeling and inference is therefore essential for the use of Bassist.

How are models specified to Bassist? Models are specified using the Bassist language. The language is declarative, and it consists of definitions of different quantities and their statistical relationships. At the basic level, a model consists of a number of quantities, such as model and hyper parameters, covariates, and the data, all called *variables* in Bassist and handled in a uniform manner. The joint density of a model is specified as a product of (conditional) densities of all variables in the model.

More formally, a model is defined by a directed acyclic dependency graph where the set of nodes is the set V of variables. The model specification is completed by defining the conditional probability $\mathsf{P}(v \mid \mathrm{pred}(v))$ for all variables $v \in V$, where $\mathrm{pred}(v)$ is the set of immediate predecessors of variable v. The joint probability distribution of the model is then $\Pi_{v \in V} \mathsf{P}(v \mid \mathrm{pred}(v))$.

Section 3 explains how models are specified using the Bassist language.

What does Bassist compute from a model and data? In Bayesian analysis some variables are given as the observed data, and the task is to compute the (posterior) distribution of the remaining variables. For instance, let's assume we want to predict future data $\mathcal{D}_u$ from some stochastic system, given observed data $\mathcal{D}_o$. That is, we are interested in estimating the distribution $f(D_u \mid D_o)$. Denote by $\mathcal{M}$ a statistical model of the system, and by $\Theta_\mathcal{M}$ the set of parameters (i.e., the set of variables is $V = \Theta_\mathcal{M} \cup D_u \cup D_o$). In the Bayesian approach we integrate over all possible parameter values to get

$$\begin{aligned} f(D_u \mid D_o) &= \frac{f(D_u, D_o)}{f(D_o)} && (1)\\ &\propto f(D_u, D_o) && (2)\\ &= \int_{\Theta_\mathcal{M}} f(D_u, D_o, \Theta_\mathcal{M}). && (3) \end{aligned}$$

The function to be integrated is the joint density specified by the model.

How does Bassist compute the distribution? Computing integral (3) exactly is often impossible and, consequently, we have to approximate it. This is what Bassist does. Given a model and data, Bassist constructs a Markov chain whose stationary distribution is (proportional to) integral (3). By simulating this Markov chain Bassist generates a sample that approximates the distribution. Bassist is based on the Markov chain Monte Carlo (MCMC) algorithm, in particular the Metropolis-Hastings method.

Where can Bassist be applied? Bassist is domain independent, and it aims at being a general purpose tool for Bayesian analysis. Thanks to the flexibility of the Metropolis-Hastings method, models with arbitrary probability distributions can be simulated, at least in principle, as long as the user is able to specify the conditional density functions for all variables. Preliminary versions of Bassist have been used in modeling tasks, e.g., in epidemiology, paleoecology, and ecology.

What system requirements etc. are there? Section 7 outlines the technical requirements.

Where can I obtain Bassist? See either section 7.5 of this document for help with obtaining and installing Bassist.

I didn't understand a word. Where can I read about Bayesian modeling and MCMC? For expository literature on these themes see, e.g., [3, 1] (Bayesian modeling) and [4, 2] (MCMC).

You didn't answer my question! How do I find out more? Read the rest of this document, then see the WWW page `<http://www.rni.helsinki.fi/cs/bassist/>`. If you want to know about some usage details, check out the manual page in the Bassist distribution. If your question is still unanswered, write to the authors. If your question is technical in nature (e.g. installation problems), write to `<Jouni.Seppanen@rni.helsinki.fi>`, otherwise to `<Hannu.Toivonen@cs.helsinki.fi>`.

2 Quick Start to Using Bassist

This section shows a typical use of Bassist using a simple example model. The example is the same as the `pump` example in the BUGS examples document [5].

2.1 The Model

Briefly, the model concerns the failures of 10 pumps. Each pump has been operated for some known amount of time, and the number of failures during the operating time is known. The number of failures is assumed to follow a Poisson distribution with respect to the operation time multiplied by some pump-specific failure rate. The failure rates are assumed to be gamma distributed, and the prior distributions of the gamma hyperparameters are Exponential(1.0) and Gamma(0.1, 0.1).

The user first creates a model description in some file, e.g. `pumps.model`. The model description determines the hierarchical structure of the model and its joint distribution.

```
model pumps;

var alpha ~ exp(1.0);
var beta ~ gamma(0.1, 1.0);

record pump {
        var theta ~ gamma(alpha, beta);
        var t;
        var lambda = theta * t;
        var x ~ poisson(lambda);
};

implementation pumps;

init alpha ~ prior; propose alpha ~ prior;
init beta  ~ prior; propose beta  ~ norm(beta, 1);

init pump.theta ~ gamma(1.0, 1.0);
propose pump.theta ~ prior;
init pump.x = 1.0;
propose pump.x ~ prior;
```

2.1.1 Variables

Let us first look at the part of the model description between the "`model pumps;`" and "`implementation pumps;`" lines. This part specifies the *variables* of the model (all numerical quantities are called variables).

In this model, there are two "global" variables called `alpha` and `beta`. For each pump, there is a "record" containing four "local" variables. The variable `theta` corresponds to the failure rate of the individual pump, `t` to the operating time of the pump, and `x` is the observed number of failures. `lambda` is an intermediate variable that is not really necessary in the model but is used to illustrate deterministic dependence.

The `~` character is used to specify probabilistic dependencies between the model variables. All variables except `t` and `lambda` are given such dependencies. On the right side of the `~` character, a distribution is given, with either constant parameters or referring to other variables.

Deterministic dependencies are introduced by the `=` character. The value of `lambda` is completely determined by the values of `theta` and `t`, without any simulation taking place. You could also specify a constant value, e.g. `var pi = 3.14;`

The variable `t` has no distribution or deterministic rule, and it is therefore a constant whose value is supplied in the data.

2.1.2 The Implementation Section

The latter part of the model description specifies some of the implementation details of the model. There is a default for everything that can be specified here, but it is not always a good one, and therefore Bassist warns if you leave something out.

All simulated variables need to be initialized when the simulation starts. The initial value can be given in the data, or it can be drawn from some distribution. By default, it is drawn from the prior distribution, but any distribution could be specified.

There must also be some proposal distribution from which to draw candidates for new values of the variable. The default is to draw from the prior, but this leads often to slow mixing. In the example model, the global variable `beta` showed poor mixing when proposing from its prior, and therefore we use a "random-walk" proposer. On the right-hand side of `~`, `beta` refers to the old value of the variable.

2.1.3 Notes on the Modeling Language

Note the emphasis on declarativity in the example: the model does *not* fix the number of pumps or the constant values. It does not even determine which variables are given in the data and which ones are simulated, except that `t` must of course be given in the data. This facilitates missing-data simulations on the same simulator.

Also note that all numerical quantities such as parameters, hyper parameters, constants, covariates, data, and functionally determined quantities are handled in a uniform manner. In the model, it does not usually matter whether a quantity is an estimated parameter, given data, or missing data.

2.2 Running the Bassist Compiler

Once the model description has been written, the Bassist compiler can be run on it to produce an executable simulator. The compiler generates C++code that is compiled and linked with the Bassist library. Here is an example run, with the C++ compilation details removed for brevity:

```
% bassist pumps.model
bassist: Parsing input files
bassist: Compiling model to C++ files
bassist: Compiling C++ files to simulator program
...
bassist: Done
```

2.3 The Data File

Once the model has been compiled into a simulator, the simulator can be run with some data. Here is the example data file, `pump.data`:

```
pump p1  { t=  94.30, x=  5, theta=i1.0 }
pump p2  { t=  15.70, x=  1, theta=i1.0 }
pump p3  { t=  62.90, x=  5, theta=i1.0 }
pump p4  { t= 126.00, x= 14, theta=i1.0 }
pump p5  { t=   5.24, x=  3, theta=i1.0 }
pump p6  { t=  31.40, x= 19, theta=i1.0 }
pump p7  { t=   1.05, x=  1, theta=i1.0 }
pump p8  { t=   1.05, x=  1, theta=i1.0 }
pump p9  { t=   2.10, x=  4, theta=i1.0 }
pump p10 { t=  10.50, x= 22, theta=i1.0 }

alpha = i1.0
beta = i1.0
```

The data file format is very simple, and is likely to change in the future. For now, "`=i`" is used to set an initial value, and "`=`" to set a constant value.

Note how the model description only has information on the model (and some implementation issues).

2.4 Running the Simulator

When running the simulator, the necessary command-line parameters are the number of iterations and the data file:

```
% ./pumps.sim 10000 pumps.data > /tmp/pumps.out
```

It is advisable to direct the output to a local file system such that it is not a big problem if it should fill up.

The output file contains some header lines that describe the data and thereafter a line for each iteration, with the value of each simulatable variable.

3 Model specification: Bassist language

The model specification determines the structure of a hierarchical model and its joint distribution. The model is constructed from concepts such as variables, records, and references.

Most of the examples in this section refer to section 2, which contains a brief but complete example of a model definition.

3.1 Starting a model definition

Each model specification begins with a declaration

```
model pumps;
```

which gives a name to the model. The name `pumps` will be used to name the output of Bassist.

3.2 Variables

All numerical quantities in models are introduced with the reserved word `var`. There are three basic kinds of variables: stochastic and deterministic variables and constants.

If a variable is given a distribution, as `alpha` in

```
var alpha ~ exp(1.0);
```

then the variable is considered *stochastic*. It *may* be assigned a fixed value in the data, but usually it is automatically estimated in the simulation.

Variables with an assignment instead of a distribution, such as `lambda` in

```
var lambda = theta * t;
```

are called *deterministic*. The value of a deterministic variable is always computed from the given expression.

A special case of a deterministic variable, a *constant* is assigned a value that does not depend on any other variable. The assignment can also be left out:

```
var pi = 3.14;
var t;
```

The latter kind of variables are also constants, but they must be given a value in the data.

3.2.1 Stochastic variables

A stochastic variable follows a priori a given probability distribution. They are declared as follows.

```
var alpha ~ exp(1.0);
```

According to this definition the variable `alpha` is a priori exponentially distributed with parameter 1.0.

Distributions can have as arguments also parameters, or, in more general, numerical expressions. For instance:

```
var theta ~ gamma(alpha, beta);
var x ~ poisson(theta * t);
```

Bassist currently supports the following probability distributions: normal, uniform, gamma, beta, poisson, exponential, geometric and binomial. Appendix A lists these distributions together with the Bassist syntax for them and some of their basic properties. Support for user-defined distributions will be available in future versions of Bassist.

Note that stochastic variables are used to represent parameters, hyper parameters, observed data, and missing data. Whether the value of a stochastic variable is to be estimated, i.e., whether it is a (hyper) parameter of missing data, of whether it is given data, depends on the data provided to a simulator. All stochastic variables that are not given a value in a data file will be simulated.

3.2.2 Deterministic variables

The value of a deterministic variable is an arithmetic expression, possibly a function of other variables. For instance, a deterministic variable `lambda` is declared to be the product of two other variables:

```
var lambda = theta * t;
```

Deterministic variables cannot be assigned values in data files.

3.2.3 Constants

Consider a situation where the user wants to give `alpha` a constant value that does not depend stochastically nor deterministically from other variables. One can define `alpha` as a constant variable:

```
var alpha = 0.9;
```

There are cases where the value of a constant is not known in advance. For instance, the values of covariates `t` in the pumps model are part of the data, and it makes no sense to try to write those values somehow into the model specification. Bassist allows constants to be defined without values:

```
var t;
```

Each constant has to have a value in a simulation run. If the value is not stated in the model definition, then it has to be provided in a data file.

Actually, if the value of a constant is fixed in the model, its value can still be changed in data. In this case the value that is declared in the data takes precedence.

3.3 Arithmetic Expressions and Including C++ code

3.3.1 Arithmetic expressions

Bassist understands all the usual arithmetic expressions. They can be used wherever a numerical value is needed: as parameters to distribution, and as values for deterministic variables.

As an example, it is not necessary to use variable `lambda` in the pumps example. Instead of

```
var lambda = theta * t;
var x ~ poisson(lambda);
```

one could write

```
var x ~ poisson(theta * t);
```

Bassist expressions can contain the operators $+$, $-$, $*$, $/$, and functions from the C++ math library (see the next section).

3.3.2 C and C++ functions

C or C++ functions can be easily used in Bassist models. It possible to call an external function anywhere in an arithmetic expression. No advance declarations are needed, one merely needs to provide the implementation for the function for Bassist at compile time.

For instance, assume `lambda` is a complex function of `theta` and `t`, and we rather provide the function in C++ than write it with the expressions of Bassist. We can then write

```
var lambda = intensity(theta, t);
var x ~ poisson(lambda);
```

or

```
var x ~ poisson(intensity(theta, t));
```

where `intensity` is a C/C++ function taking two arguments of type double and returning a value of type double.

Also, all the functions from the C++ mathematical library that take `double` arguments and return `double` numbers, such as `sin` and `log`, are usable.

When compiling a model, you can use a command-line switch to add your own C++ files. See the manual page for information on command-line switches.

3.4 Records

Logical units of a model can be organized into records. A record consists of variables, or fields. They can be stochastic variables, deterministic variables, or constants, just like global variables (i.e., ones outside all records).

Only record types are defined in the model specification. Actual instances of records are specified in data files. Also the number of instances is determined first by the data, not by the model.

Record types are defined as follows.

```
record pump {
  var theta ~ gamma(alpha, beta);
  var t;
  var lambda = theta * t;
  var x ~ poisson(lambda);
};
```

This code defines the record type `pump`. All instances (objects) of type `pump` will contain the fields listed in the definition: stochastic variables `theta` and `x`, a constant `t`, and a deterministic variable `lambda`.

When necessary, a field of a particular record instance, say the field `x` of the record instance `pump1`, can be referred to by the "dot" notation `pump1.x`.

References to variables are by default within the record instance. For instance, the definition above means that for each pump $\texttt{pump}_j$,

```
pumpn.lambda = pump_j.theta * pump_j.t;
pumpn.x = ~ poisson(pump_j.lambda);
```

and so on.

3.4.1 Record references

It is possible to declare references (like C/C++ pointers) to record instances. The general syntax of reference is

```
ref record_name as reference_name;
```

The actual instance of the record that is referred to is fixed in the data. In the example below two record types are declared.

```
record A {
    var a ~ unif(0,1);
    var b = 10*a;
    var y ~ beta(b,3);
};
record B {
```

```
    ref A as my_reference;
    var x ~ norm(my_reference.y, 1);
};
```

According to the above declarations there are two record types A and B. A record of type B has a pointer to a record of type A and it uses the value of stochastic variable y of record A through reference my_reference as a hyperparameter for prior distribution of its stochastic variable x.

3.5 Aggregate functions

Bassist provides some aggregate functions that evaluate operations on every instance of some specific record type. For instance, we may have a model that consists of record types A and B. Let us assume, that in record B we need to know sum of all variables A.x. This can be done by using the aggregate function aggr_sum(A.x) as presented in the examples below.

```
record A {
  var x;
};
record B {
  var y ∼ norm(aggr_sum(A.x), 2);
};
```

The general semantics of aggregate function (except function count) is

$$\mathtt{aggr\_}\bigoplus(\mathtt{A.x}) \equiv \bigoplus_{\mathtt{d}\in\mathtt{A}} \mathtt{d.x}.$$

Relation d∈A means, that d is instance of record A. At the moment six aggregate functions are available: count, minimum, maximum, sum, product and average. Function count calculates the number of instances of a given record type

```
var count = aggr_count(A);
```

As mentioned, aggr_sum calculates the sum of given record and variable

```
var sum = aggr_sum(A.x);
```

The product can be evaluated with function aggr_product

```
var product = aggr_product(A.x);
```

The arithmetic average of A.x's is

```
var average = aggr_avg(A.x);
```

Minimum and maximum are

```
var minimum = aggr_min(A.x);
var maximum = aggr_min(A.x);
```

respectively.

3.6 Implementation section

To distinguish the structure of a model from its implementation, the implementation details are grouped in another section. Currently, the implementation section may contain proposal distributions and initial values for the stochastic variables of the model. The implementation section begins with the declaration

```
implementation model_name;
```

The name of the model is repeated to facilitate keeping the implementation section in a separate file. Therefore, you can experiment with different implementations of the same model by changing the command line of the Bassist compiler.

3.6.1 Proposal distributions

In order to quicken burn-in it is reasonable, that user is able to define proposal distribution. Proposal distribution for record `A`'s stochastic variable `x` is defined within `implementation` environment. Prior distribution will be used as a default, if you do not define proposal distribution for a stochastic variable. Proposal distribution for a stochastic variable is defined as follows

```
propose record_name.my_stochastic_variable ~
distribution_name(h);
```

In the example above `distribution_name` is some built-in probability distribution (see section 2.1). You may use current value of the stochastic variable as a hyperparameter for the proposal distribution (e.g. if you want to use random walk). For instance random walk with a normal distributed proposer (fixed variance)could be declared in the following way

```
propose A.x ~ norm(x, 1)
```

Remark that declaration of record `A` fixes variable scope such that the mean (the first hyperparameter) is current value of field `x` of record `A`'s instance. If you want to be explicit you can declare, that you want to use prior distribution as a proposal distribution with a declaration

```
propose record_name.my_stochastic_variable ~ prior;
```

3.6.2 Initializations

You may fix initial value for your stochastic variables within implementation specification as follows

```
init record_name.my_stochastic_variable = 42;
```

It is also possible to initialize a value for stochastic variable according to some built-in distribution

```
init record_name.my_stochastic_variable ~ gamma(5,1);
```

4 Structure of the Bassist Data File

In this section the general syntax of valid the Bassist data file will be described. General grammar is as follows (ϵ is empty string).

data ::= *(entity)**
entity ::= `record_name instance_name` *references setvalues predict*
references ::= **[** *(reference*,*)** *reference* **]** | ϵ
reference ::= `reference_name`
setvalues ::= **{** *(setvalue*,*)** *setvalue* **}** | ϵ
setvalue ::= `variable_name` *equals* `numerical_value`
equals ::= = | =**i**
predict ::= **predicted** | ϵ

Let us construct a small example.

```
model demo;

record A {
  var a ~ norm(0,1);
  var x ~ bin(10,2);
};

record B {
  ref A as foo;
  ref A as bar;
  var y ~ norm(foo.x, 1);
  var value ~ poisson(my_function(y)*bar.a);
};
```

A valid datafile for the model `demo` could now be

```
A instanceA1 { a = 10.1 } predicted
A instanceA2 { a = 12.41, x =i 1.2 }
B instanceB1 [instanceA1, instanceA2] { value = 42 }
```

Remark that `=i` sets initial value for the variable (i.e. value of the variable will not be fixed, but initial value for the first iteration is given). This is useful in order to quicken burn-in (assuming, that we know reasonable initial values).

5 Compilation

The usual way of running the Bassist compiler is

```
bassist pumps.model
```

You can also give several file names on the command line, e.g. in the case that you keep the implementation section in a different file. The files will be concatenated by Bassist.

```
bassist pumps.model pumps.impl
```

By default, the name of the simulator is the model name with the "`.sim`" suffix.

If you want to see the code generated by Bassist, use the `-s` option. It creates a directory and places the code files there. By default, the name of the directory is the name of the model with the "`.dir`" suffix, but the `-s` option takes an optional argument that must be typed without a space:

```
bassist -smy_directory pumps.model
```

By default, the `-s` option will not overwrite any files, but you can force it to do so with the `-f` option:

```
bassist -f -smy_directory pumps.model
```

For other options, see the manual page in the Bassist distribution.

6 Post-Processing: Useful Tools

6.1 Processing the Output

The output of a Bassist simulator is a set $\mathcal{S}$ of sampled vectors from the posterior distribution $f(x)$. Due to the law of large numbers, the formula

$$\frac{1}{|\mathcal{S}|}\sum_{s\in\mathcal{S}} s \approx E(f(x)),$$

holds when $|\mathcal{S}|$ is large enough. Therefore, we often want to compute the mean of components in order to approximate the expectation of the posterior distribution. The output format follows the syntax of the `vibra` scripts. `vibra` is a perl script, that provides customary operations for manipulating ASCII-arrays. The use of `vibra` is essential in processing Bassist's output, so its use is shortly described here.

At the moment available operations in `vibra` are projection, sorting, selection and counting. The author of `vibra` is Mika Klemettinen.

The input file of `vibra` consists of a header part and the actual data part. The header defines the number of attributes, the names of the attributes, the record separator and the field separator. Necessary information about the structure of the data is declared between markers "header", "attributes", "field separator" and "record separator". The header syntax is remotely HTML-like. The Header starts with the tag

```
#<header>
```

and ends at the tag

```
#</header>
```

The attribute names are listed between "attributes" tags

```
#<attributes> attr_1_name attr_2_name ...  attr_n_name
</attributes>
```

The record separator is declared between "record separator" tags

```
#<record separator> '\n' </record separator>
```

and the field separator is declared in a similar way

```
#<field separator> ' ' </field separator>
```

According to above declarations it would be assumed that records are separated by new-line character and fields are separated by one space-bar character. Any line that begins with the # character is assumed to be a comment, if it is not followed by any marker and it is within "header" tags. Here is an example of a complete input file:

```
#<header>
#<attributes> C1 C2 C3 C4 C5 </attributes>
#<record separator> '\n' </record separator>
#<field separator> ' ' </field separator>
# My comment line
#</header>
10 11 12 13 14
20 21 22 23 24
30 31 32 33 34
40 41 42 43 44
50 51 52 53 54
60 61 62 63 64
70 71 72 73 74
```

The command line syntax of vibra is described by the help message

```
vibra [-C | -O | -S | -P] [-h] <parameter> < input > output

Arguments:  -C - count mode
-O - sort mode
-S - select mode
-P - project mode
-h - usage help and script information

Help for different modes:  vibra [-C | -O | -S | -P] -h
```

You can find detailed instructions and some use case examples on the WWW at <http://www.RNI.Helsinki.FI/cs/bassist/v0.8/vibra.html>.

7 Technical Issues

7.1 System Environment

At our setting, we run Bassist on Pentium computers with the Debian GNU/Linux 1.3 operating system, but we expect that it should either run or be easily portable to any Unix-like system. One requirement is a reasonably new C++ compiler: we are using `g++` from the EGCS 1.1.1 release, and at present no pre-EGCS versions of `g++` work. (Bassist does *not* use C++ exceptions, but it does require a working STL implementation.)

Bassist simulators are very computing-intensive. At RNI, we use the Condor high-throughput computing environment to distribute the load. In order to run Bassist simulators under Condor, you only need to make a minor modification to the generated `Makefile`. See the Condor WWW site `<http://www.cs.wisc.edu/condor/>` for more information on Condor.

Bassist simulators also produce a lot of data that can easily fill up your disk space if you aren't careful. There are command-line options to reduce the amount of data output. In many cases, you will only be interested in the values of some parameters, and you can discard the rest of the information. If that is not enough, you can opt to discard some initial iterations as "burn-in", and "thin" the rest of the iterations by only outputting, say, every 10th iteration.

7.2 User Requirements

In order to use Bassist, you absolutely need to understand Bayesian modeling and MCMC. You should also be at least somewhat familiar with the command-line shell environment of your operating system, since Bassist does not presently have any graphical user interface.

7.3 Licensing

Presently, Bassist is available under the GNU General Public License (see `<http://www.gnu.org/copyleft/gpl.html>`), but that may very well change in the future. Therefore, if you intend to make any significant contributions to Bassist, please contact us in advance to discuss licensing. Also, if the GNU GPL does not suit your needs as a Bassist user, please contact us.

7.4 Availability

Bassist is being developed in the University of Helsinki both at the Department of Computer Science and at Rolf Nevanlinna Institute. More information and a publicly available version can be found in the net at `<http://www.rni.helsinki.fi/cs/bassist/>`.

7.5 Obtaining and installing Bassist

To obtain the Bassist distribution, use your favorite WWW browser to go to the Bassist home page at `<http://www.rni.helsinki.fi/cs/bassist/>`. Follow the link to the latest major version (v0.8 at this writing), and find the link to the compressed distribution file (e.g. `bassist-0.8.2.tar.gz`) under the heading "Installing Bassist".

Once downloaded, uncompress the file and extract its contents:

```
gunzip bassist-0.8.2.tar.gz
tar xvf bassist-0.8.2.tar
```

With GNU tar, you can simply type

```
tar xzvf bassist-0.8.2.tar.gz
```

In the created directory, type

```
./configure
```

If the script doesn't find the right C and C++ compilers, set the environment variables `CC` and `CXX` accordingly, and rerun the script. Also, if the path `/usr/local` is not where you want to install Bassist, you can change it with a command-line option. The environment variables don't have any effect unless you destroy the prior settings with `make distclean`:

```
make distclean
CC=egcs
CXX=egcs++
export CC CXX
./configure --prefix=/another/path
```

After configuring Bassist, you can type

```
make install
```

to compile and install the program. The program is installed with the version number in the name; if you wish to call it with the name "`bassist`", see the last few lines of the installation output for instructions.

In case of any installation problems, please report a bug—see the Bassist WWW page for contact information.

A Distributions Supported by Bassist

This table lists the library distributions supported by Bassist. Currently, there is no easy way to add support for your own distribution, but one is being planned.

Distribution & notation	Parameters & mean E and variance V
Beta	'prior sample sizes', $\alpha, \beta > 0$
$x \sim \mathrm{beta}(\alpha, \beta)$	$\mathsf{E}(x) = \dfrac{\alpha}{\alpha+\beta}, \quad \mathsf{V}(x) = \dfrac{\alpha\beta}{(\alpha+\beta)^2(\alpha+\beta+1)}$
Binomial*	'sample size' $n \in \{1, 2, \dots\}$, 'probability' $p \in [0, 1]$
$x \sim \mathrm{bin}(n, p)$	$\mathsf{E}(x) = np, \quad \mathsf{V}(x) = np(1-p)$
Exponential	scale $\lambda > 0$
$x \sim \exp(\lambda)$	$\mathsf{E}(x) = \lambda, \quad \mathsf{V}(x) = \lambda^2$
Gamma	shape $\alpha > 0$, inverse scale $\beta > 0$
$x \sim \mathrm{gamma}(\alpha, \beta)$	$\mathsf{E}(x) = \dfrac{\alpha}{\beta}, \quad \mathsf{V}(x) = \dfrac{\alpha^2}{\beta}$
Geometric	'probability' p
$x \sim \mathrm{geom}(p)$	$\mathsf{E}(x) = \dfrac{1-p}{p}, \quad \mathsf{V}(x) = \dfrac{1-p}{p^2}$
Normal	location μ, scale $\sigma^2 > 0$
$x \sim \mathrm{norm}(\mu, \sigma^2)$	$\mathsf{E}(x) = \mu, \quad \mathsf{V}(x) = \sigma^2$
Poisson*	'rate' λ
$x \sim \mathrm{poisson}(\lambda)$	$\mathsf{E}(x) = \lambda, \quad \mathsf{V}(x) = \lambda$
Uniform	boundaries $a < b$
$x \sim \mathrm{unif}(a, b)$	$\mathsf{E}(x) = \dfrac{a+b}{2}, \quad \mathsf{V}(x) = \dfrac{(b-a)^2}{12}$

*discrete distribution

References

[1] D. Draper. *Bayesian Hierarchical Modeling*. 1998. http://www.bath.ac.uk/∼masdd.

[2] D. Gamerman. *Markov Chain Monte Carlo Stochastic simulation for Bayesian inference*. Chapman & Hall, 1997.

[3] A. Gelman, J. B. Carlin, H. S. Stern, and D. B. Rubin. *Bayesian Data Analysis*. Chapman & Hall, New York, 1995.

[4] W. R. Gilks, S. Richardson, and D. J. Spiegelhalter. *Markov Chain Monte Carlo in Practice*. Chapman & Hall, London, 1996.

[5] D. Spiegelhalter, A. Thomas, N. Best, and W. Gilks. Bugs 0.5 examples, volume 1 (version *i*). ftp.mrc-bsu.cam.ac.uk, 1996.