GNU MCSim:

A Monte Carlo Simulation Program

---

by Frédéric Y. Bois and Don R. Maszle

User's Manual, software version 6.2.0

Copyright © 1997-2020 Free Software Foundation, Inc.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

contact:
Frederic Bois
fbois@member.fsf.org

Table of Contents

1 Software and Documentation Licenses 1

- 1.1 Software license 1
- 1.2 Documentation license 1

2 Overview 9

- 2.1 General procedure 9
- 2.2 Types of simulations 10
- 2.3 Major changes introduced with version 5.4.0 10
- 2.4 Major changes introduced with version 5.5.0 10
- 2.5 Major changes introduced with version 5.6.0 11
- 2.6 Major changes introduced with version 6.0.0 11
- 2.7 Major changes introduced with version 6.1.0 12
- 2.8 Major changes introduced with version 6.2.0 12

3 Installation 13

- 3.1 System requirements 13
- 3.2 Distribution 13
- 3.3 Machine-specific installation 14
  - 3.3.1 Unix and GNU/Linux operating systems 14
  - 3.3.2 Other operating systems 15

4 Working Through an Example 17

5 Writing and Compiling Structural Models 19

- 5.1 Using `mod` to preprocess model description files 19
- 5.2 Using `makemcsim` to preprocess and compile model files 20
- 5.3 Syntax of the model description file 20
  - 5.3.1 General syntax 21
  - 5.3.2 Declarations of global variables 23
  - 5.3.3 Model types 25
  - 5.3.4 Special functions 25
  - 5.3.5 Input functions 27
  - 5.3.6 In line functions 29
  - 5.3.7 Model initialization 29
  - 5.3.8 Dynamics section 30
  - 5.3.9 Delay differential equations 31
  - 5.3.10 Output calculations 31
  - 5.3.11 Comments on style 32
- 5.4 Reading SBML models and applying a template 32
- 5.5 Working with the *R* package *deSolve* 35

6 Running Simulations **37**
6.1 Using the compiled program 37
6.2 Syntax of the simulation definition file 38
6.2.1 General input file syntax 39
6.2.2 Input functions (revisited) 40
6.2.3 Global specifications 40
`OutputFile()` specification 41
`Integrate()` specification 41
`MonteCarlo()` specification 42
`MCMC()` specification 43
`SetPoints()` specification 45
`OptimalDesign()` specification 46
`Distrib()` specification 48
`SimType()` specification 52
6.2.4 Specifying basic conditions to simulate 52
`Simulation` sections 52
`Events()` specification for state discontinuities 53
`StartTime()` specification 54
`Print()` specification 54
`PrintStep()` specification 54
6.2.5 Setting-up statistical models 55
`Level` sections 57
`Data()` specification 58
6.3 Analyzing simulation output 58
6.4 Error handling 59

7 Common Pitfalls **61**

8 XMCSim **63**

Bibliographic References **65**

Appendix A Keywords List **67**

Appendix B Examples **71**
B.1 `linear.model` 71
B.2 `1cpt.model`: A example model description file 71
B.3 `perc.model`: A example model description file 73
B.4 `perc.lsodes.in` 78

Concept Index **79**

1 Software and Documentation Licenses

1.1 Software license

GNU MCSim is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 3 of the License, or (at your option) any later version. This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

1.2 Documentation license

The GNU Free Documentation License

Version 1.2, November 2002

> Copyright © 2000,2001,2002 Free Software Foundation, Inc.
> 51 Franklin St, Fifth Floor, Boston, MA 02110-1301, USA
>
> Everyone is permitted to copy and distribute verbatim copies
> of this license document, but changing it is not allowed.

0. PREAMBLE

   The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

   This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

   We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

   This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as “Acknowledgements”, “Dedications”, “Endorsements”, or “History”.) To “Preserve the Title” of such a section when you modify the Document means that it remains a section “Entitled XYZ” according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

   You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

   You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

   If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document’s license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

   If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

   If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

   You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

   The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

   In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

   You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

   If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

   If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See `http://www.gnu.org/copyleft/`.

    Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

2 Overview

GNU MCSim is a simulation and statistical inference tool for algebraic or differential equation systems. Other programs, such as *GNU Octave*, have been created to the same end. Still, most available tools are not optimal for performing computer intensive and sophisticated Monte Carlo analyses. *GNU MCSim* was created specifically to this end: to perform Monte Carlo analyses in an optimized, and easy to maintain environment. The software consists in two pieces, a model generator and a simulation engine:

- The model generator, "*mod*", was created to facilitate structural model definition and maintenance, while keeping execution time short. You code your model using a simplified syntax and `mod` translates it in C.

- The simulation engine is a set of routines which are linked to your model during compilation to produce executable code. After that you can run simulations of your model under a variety of conditions, specify an associated statistical model, and perform simulations.

2.1 General procedure

Model building and simulation proceeds in four stages:

1. You create with any text editor (*e.g.*, `emacs`) a model description file. The reference section on `mod`, later in this manual gives you the syntax to use (see Chapter 5 [Writing and Compiling Structural Models], page 19). This syntax allows you to describe the model variables, parameters, equations, inputs and outputs in a C-like fashion without having you to actually know how to write a C program.
2. You instruct the model generator, `mod`, to preprocess your structural model description file. `Mod` creates a C file, called `model.c`.
3. You compile and link the newly created `model.c` file together with a library containing the other C routines (or with the other C files of the `mcsim/sim` directory). *GNU MCSim* C code is standard, so you should be able to compile it with any standard C compiler, for example GNU `gcc`. After compiling and linking, an executable simulation program is created, specific of your particular model. These preprocessing and compilation steps can be performed in Unix with a single shell command `makemcsim` (in which case, the `model.c` is created only temporarily and erased afterward). This produces the most efficient code for your particular machine.
4. You then write any number of simulation specification files and run them with the compiled `mcsim` program. These simulation files describe the kind of simulation to run (simple simulations, Monte Carlo etc.), various settings for the integration algorithm if needed, and a description of one or several simulation conditions (eventually with a statistical model and data to fit) (see Chapter 6 [Running Simulations], page 37). The simulation output is written to standard ASCII files.

Little or no knowledge of computer programming is required, unless you want to tailor the program to special needs, beyond what is described in this manual (in which case you may want to contact us).

Under Unix, a graphical user interface written in Tcl/Tk, *XMCSim* (called by the command `xmcsim`), is also provided. This menu-driven interface automatizes the compilation and running tasks. It also offers a convenient interface to 2-D and 3-D plotting of the simulation results.

2.2 Types of simulations

Five types of simulations are available:

- A simple simulation will solve (eventually integrate) the equations you specified, using the default parameter values eventually overridden in the simulation specification file. User-requested outputs are sent to an output file of your choice.
- "*Monte Carlo*" simulations will perform repeated (stochastic) simulations across a randomly sampled region of the model parameter space (see [MonteCarlo() specification], page 42).
- A Markov-chain Monte Carlo (*MCMC*) simulation performs a series of simulations along a Markov chain in the model parameter space (see [MCMC() specification], page 43). In MCMC simulations the random choice of a new parameter value is influenced by the current value. They can be used to obtain the Bayesian posterior distribution of the model parameters, given a statistical model, prior parameter distributions (that you need to specify) and data for which a likelihood function can be computed. The program handles hierarchical (*e.g.*, random effects and mixed effects) statistical models (see Section 6.2.5 [Setting-up statistical models], page 55).
- A "*SetPoints*" simulation solves the model for a series of specified parameter sets, listed in a separate ASCII file (see [SetPoints() specification], page 45). You can create these parameter sets yourself (on a regular grid, for example) or use the output of a previous Monte Carlo or MCMC simulation.
- An "*OptimalDesign*" procedure optimizes the number and location of observation times for experimental conditions, in order to minimize the variance of a parameter or an output you specify, given a structural model, a statistical model, and prior distributions for their parameters (see [OptimalDesign() specification], page 46).

2.3 Major changes introduced with version 5.4.0

- GNU MCSim is now distributed under version 3 of the GNU General Public License.
- The installation scripts have been rewritten using GNU `autoconf`, `automake` and `libtool`. This *should* make *GNU MCSim* easier to install and more portable.
- Systems Biology Markup Language (SBML) models are read by libSBML if it is installed.
- Tempered MCMC (useful for hard, multimodal posterior densities, for rapid and guaranteed convergence, and for model choice) and stochastic optimizations are offered as options of the `MCMC()` specification.

2.4 Major changes introduced with version 5.5.0

- The installation scripts have been regenerated using GNU `autoconf` version 2.69 which fixes a potential security problem in the installation.
- The `mod` utility can now generate C model files suitable for use with the R package `deSolve`. Use `mod -R` for that.

2.5 Major changes introduced with version 5.6.0

- The keyword `End` is now mandatory at the end of every model. This is not backward compatible (you will need to modify your older models accordingly).
- The `StartTime()` specification can now accept a symbolic parameter. That allows you to treat the initial time as a random variable in error-in-variable problems (when the initial time is an unknown).
- The `PrintStep()` specification can read a list of variables to print (as `Print()` does).
- In MCMC simulation, the jump kernel is now output in a file with the *.kernel* extension. If the simulations are restarted in a continuation mode and if a kernel file with the same name as the restart file (with an added *.kernel* extension) is present, the jump kernel will restored to its saved value.
- Delay differential equations can now be coded and solved using the `CalDelay` function.
- Discontinuities in state variables can now be forced to happen at predefined times through the `Events()` specification.

2.6 Major changes introduced with version 6.0.0

In `mod`:

- You can now specify the Jacobian matrix of the model's derivatives with respect to state variables in a `Jacobian` section. It will then be used by the `Lsodes` integrator instead of numerical differentiation, see Section 5.3 [Syntax of mod files], page 20.
- The standard C function `fmax` and `fmin` can be used in your models, see [Standard functions], page 21.

In `sim`:

- The vector notation (see [Vectors], page 22) can now be used in input simulation files.
- If you use the GNU Scientific Library (`libgsl`), the very long-period "Mersenne twister" random number generator is now used, see [Random Generator], page 27. Otherwise the Park and Miller generator is used, as before.
- A new input function, `PerTransit`, is available to simulate delayed gut absorption, for example (see [PerTransit], page 27).
- The Sundials library `Cvodes` integrator `https://computation.llnl.gov/projects/sundials` can be called in a `Integrate` specification, see [Integrate() specification], page 41.
- Specifications for tempered MCMC simulations have been extended to include thermodynamic integration and allow infinite temperature (*i.e.*, perk 0), see [Tempered MCMC], page 44.
- Two new distributions, `Normal_cv` and `TruncNormal_cv`, can be used to draw normal random variates with specified mean and coefficient of variation, see [Normal_cv], page 49.
- Symbols (denoting previously defined variables) can be used to specify times and magnitudes in `Events` (see [Events() specification], page 53).
- The `Prediction()` specification can be used as a synonym for `Print()`, see [Print() specification], page 54.

- Minor bugs were fixed (see the `http://savannah.gnu.org/projects/mcsim` repository for details).

2.7 Major changes introduced with version 6.1.0

In `sim`:

- In the case of posterior tempered MCMC (*simTypeFlag* equal to 3) or thermodynamic integration (*simTypeFlag* equal to 4), the inverse temperature (perk) scale is determined automatically (unless you specific your own scale). The scale optimization is rather efficient and often reaches perk 0, which offers garanteed convergence with only one chain and estimation of the target posterior's normalization constant (hence Bayes factors for model choice).
- `UserSpecifiedLL` can be used to specify an arbitrary data likelihood.

2.8 Major changes introduced with version 6.2.0

In `mod`:

- You can use C preprocessor directives (*i.e.*, `#include`) in `Inline()` statements. That you allows you, for example, to use GSL code in such statements or import large C code sections in your models.
- The function `NegativeBinomialRandom()` is available to generate corresponding random variates in your models.

In `sim`:

- GNU MCSim can now run Monte Carlo, SetPoints and MCMC simulations (monitoring their convergence in real time) in parallel on multiprocessor architectures if a MPI library (Open MPI or MPICH, for example) is installed.
- The negative binomial distribution is available for use in statistical models.
- The efficiency of the truncated normal and log-normal samplers has been improved by using C. Robert 1995 algorithm (see [Bibliographic References], page 65).
- New options for the Cvodes integrator are available.

3 Installation

3.1 System requirements

GNU MCSim is written in ANSI-standard C language. We are distributing the source code and you should be able to compile it for any system, provided you have an ANSI C compliant compiler.

Starting with version 5.0.0 *GNU MCSim* is using a few routines from the GNU Scientific Library (`libgsl`). We recommend that you install version 1.5 (or higher) of the shared GSL library, gslcblas library, and GSL include files on your system. Otherwise, some features (the `TruncInvGamma` density, and the Mersenne twister random number generator, see [Random Generator], page 27) will not be available (you'll get a error message if you are trying to use them.)

Version 5.4.0 and higher of *GNU MCSim* can take advantage of (`libSBML`) to read SBML models. If you choose to install libSBML on your system, we recommend that you use version 3.3.2 (or higher) of libSBML. LibSBML needs an XML parser library (either Expat, Xerces, or libxml2). The Expat library has worked well for us under Linux.

Version 6.2.0 and higher of *GNU MCSim* can take advantage of (`MPI`), the message passing protocol, to parallelize Monte Carlo, Setpoints, or MCMC simulations on multiprocessor architectures. You will need to have a MPI library installed on your system.

On any system we recommend the GNU `gcc` compiler (freeware). The automated installation script checks for the availability on your system of the tools needed for compilation and proper running of the software. It should warn you of missing component and eventually adapt the installation to your needs (for example by installing the package locally if you do not have superuser's priviledges).

To run the graphical user interface *XMCsim*, you need a GNU/Linux or Unix system with "XWindows", "Tcl/Tk" and "wish" installed.

3.2 Distribution

GNU MCSim source code is available on Internet through:

- `http://savannah.gnu.org/projects/mcsim`.

Packaged distributions are available at:

- `http://ftp.gnu.org/gnu/mcsim`,
- `http://www.gnu.org/software/mcsim`,

and mirror sites of the GNU project.

Three mailing lists are available for *GNU MCSim* users:

General *info* on *GNU MCSim* is broadcasted through:

- `http://lists.gnu.org/archive/html/info-mcsim`

You can subscribe to the info list by going to:

- `http://lists.gnu.org/mailman/listinfo/info-mcsim`.

You can request *help* from us, and from other *GNU MCSim* users, by sending email to:

- help-mcsim@gnu.org

(see `http://lists.gnu.org/mailman/listinfo/help-mcsim` for subscribing).

Help archives are found at:

- `http://news.gmane.org/gmane.comp.gnu.mcsim`,
- `http://lists.gnu.org/archive/html/help-mcsim`.

You can report *bugs* to us, by sending email to:

- bug-mcsim@gnu.org

(`http://lists.gnu.org/mailman/listinfo/bug-mcsim` for subscribing).

Bugs archives are located at:

- `http://news.gmane.org/gmane.comp.gnu.mcsim.bugs`,
- `http://lists.gnu.org/archive/html/bug-mcsim`.

3.3 Machine-specific installation

3.3.1 Unix and GNU/Linux operating systems

To install on a Unix or GNU/Linux machine, download (in binary mode) the distributed archive file to your machine. Place it in a directory where there is no existing `mcsim` subdirectory that could be erased (make sure you check that). Decompress the archive with GNU gunzip (`gunzip <archive-name>.tar.gz`). Untar the decompressed archive with tar (`tar xf <archive-name>.tar`) (do `man tar` for further help). Move to the `mcsim` directory just created and issue the following commands:

```
./configure
make
make check
```

The first command above checks for the availability of the tools needed for installation and proper running of the software. The second compiles the `mod` program and the dynamic `libmcsim.so` library and eventually compiles this manual in various formats. The third checks whether the software is running and producing meaningful results in test cases. In case of error messages, don't panic: check the actual differences between the culprit output file and the file `sim.out` produced by the checking. Small differences may occur from different machine precision. This can happen for random numbers, in which case the Markov chain simulations (MCMC) can diverge greatly after a while.

If you are logged in as "root" or have sufficient access rights, you can then install the software in common directories in `/usr` by typing at the shell prompt:

```
make install
```

If this system-wide installation is successful the executable files `mod`, `makemcsim`, `xmcsim` are installed in `/usr/local/bin`. The library `libmcsim` is placed in `/usr/local/lib`. A copy of the `mcsim` source directory (with the `mod`, `sim`, `doc`, `examples`, and `xmcsim` subdirectories) is placed in `/usr/local/share`. If you have the GNU `info` system available, an `mcsim` node is added to the main `info` menu, so that `info mcsim` will show you this manual. Finally, a symbolic link to `/usr/local/share/mcsim/doc`, which contains the documentation files and this manual (if it was generated), is created as `/usr/share/doc/mcsim`.

If you do not have the necessary access rights and want to install *GNU MCSim* in a directory such as `/home/me`, type:

```
./configure prefix=/home/me
```

This will copy or move `mod`, `makemcsim`, and `xmcsim` in a `/bin` directory in the `/home/me` directory, creating it if necessary. The library `libmcsim.so` will be moved to the `/home/me/lib` directory, *etc.*

If MPI is available on your system, parallelization of Monte Carlo runs will be enabled by default. To force the `configure` script to ignore MPI, you should type:

```
./configure --with-mpi=no
```

See also the `README` and `INSTALL` text files located in the top directory of the distribution.

On certain platforms (Linux...), you will also need to do one of the following:
1) run 'ldconfig' (see the man page if this is unfamiliar)
2) set the LD_LIBRARY_PATH (or equivalent) environment variable to contain the path "/usr/local/lib" or whatever you set so that programs can find the libSBML library at run-time.

3.3.2 Other operating systems

Under other operating systems (Windows, etc.) or if everything else fails you should be able to both uncompress and untar the archive with widely distributed archiving tools. Refer to the documentation of your C compiler to create an executable `mod` file from the source code files (getopt.c, lex.c, lexerr.c, lexfn.c, mod.c, modd.c, modi.c, modiSBML.c, modiSBML2.c, modo.c, strutil.c) provided in the `mod` directory. If you want to process SBML models it is best to install the libSBML library first. You would then compile mod with the HAVE_LIBSBML flag defined (option `-DHAVE_LIBSBML`) and link with the library (using the `-lsbml` directive). Place then the executable `mod` on your command path.

The `sim` directory contains all the source files (delays.c, getopt.c, lex.c, lexerr.c, lexfn.c, list.c, lsodes1.c, lsodes2.c, matutil.c, matutilo.c, mh.c, modelu.c, optdsign.c, random.c, sim.c, simi.c, siminit.c, simmonte.c, simo.c, strutil.c, yourcode.c) to create a dynamic library or a set of objects to link with the `model.c` files generated by `mod` after processing your models. Compilation also requires reference to the `config.h` file sitting in the main folder (one level above the `sim` directory). The -I.. option should make the compiler aware of the correct location of `config.h`. Alternatively, `config.h` can be copied into the `sim` directory to make the package complete (apart of model.c).

The final product should be an executable able to run your model. Linking with the GNU Scientific Library (`gsl`) is recommended (but not mandatory. In that case, define the `HAVE_LIBGSL` flag and link with the `-lgsl` and `-lgslcblas` (in that order).

You are now ready to use *GNU MCSim*. We recommend that you go through the next section of this manual, which walks you through an example of model building and running.

4 Working Through an Example

Several models and simulation specification files are provided with the package as examples (they are in the `examples` directory. You can try any of them. The linear regression model is particularly simple, but to be more complete we will try here a nonlinear implicit model, specified through differential equations.

Pharmacokinetics models describe the transport and transformation of chemical compounds in the body. These models often include nonlinear first-order differential equations. The following example is taken from our own work on the kinetics of tetrachloroethylene (a solvent) in the human body (Bois et al., 1996; Bois et al., 1990) (see [Bibliographic References], page 65).

Go to the `mcsim/examples/perc` directory (installed either locally or by default in `usr/share` under Unix or GNU/Linux). Open the file `perc.model` with any text editor (*e.g.*, `emacs` or `vi` under Unix). This file is an example of a model definition file. It is also printed at in Appendix the end of this manual (see Section B.3 [perc.model], page 73). You can use it as a template for your own model, but you should leave it unchanged for now. In that file, the pound signs (`#`) indicate the start of comments. Notice that the file defines:

- state variables for the model (for which differentials are defined), for example:

```
States = {Q_fat,   # Quantity of PERC in the fat (mg)
          Q_wp,    #   ...   in the well-perfused compartment (mg)
          Q_pp,    #   ...   in the poorly-perfused compartment (mg)
          Q_liv,   #   ...   in the liver (mg)
          Q_exh,   #   ...   exhaled (mg)
          Q_met}   # Quantity of metabolite formed (mg)
```

- output variables (obtainable at any time as analytical functions of the states, inputs and parameters), for example:

```
Outputs = {C_liv,           # mg/l in the liver
           C_alv,           # ... in the alveolar air
           C_exh,           # ... in the exhaled air
           C_ven,           # ... in the venous blood
           Pct_metabolized, # % of the dose metabolized
           C_exh_ug}        # ug/l in the exhaled air
```

- input variables (independent of the others variables, and eventually varying with time), for example:

```
Inputs = {C_inh,   # Concentration inhaled (ppm)
          Q_ing};  # Quantity ingested (mg)
```

- model parameters (independent of time), such as:

```
LeanBodyWt = 55; # lean body weight (kg)
```

- model initialization and parameters' scaling (the parameters used in the dynamic equations can be made functions of other parameters: for example volumes can be computed from masses and densities, etc.),
- system's dynamics (differential or algebraic equations defining the model *per se*),
- equations to compute the output variables.

This model definition file as a simple syntax, easy to master. It needs to be turned into a C program file before compilation and linking to the other routines (integration, file management etc.) of *GNU MCSim*. You will use `mod` for that. First, quit the editor and return to the operating system.

To start `mod` under Unix just type `mod perc.model`. After a few seconds, with no error messages if the model definition is syntactically correct, `mod` announces that the `model.c` file has been created. It should operate similarly under other operating systems.

The next step is to compile and link together the various C files that will constitute the simulation program for your particular model. Note that each time you want to change an equation in your model you will have to change the model definition file and repeat the steps above. However, changing just parameter values or state initial values does not require recompilation since that can be done through simulation specification files.

- Under Unix, the simplest is to use the `makemcsim` script. Just type `makemcsim` and compilation will be done automatically (see Section 5.2 [Using makemcsim], page 20). An executable `mcsim.perc` is created. You can rename it if you wish.
- Under other operating systems, you should use the command `make` or its equivalent to compile and link together the `model.c` file created by `mod` and the other C files of the `sim` directory (see Chapter 3 [Installation], page 13). That should create an application (you should give it a name specific to the model you are developing, *e.g.*, `mcsim.perc`). Refer to your compiler manual for details on how to use your programming environment. Your executable `mcsim.perc` program is now ready to perform simulations.

To start your *GNU MCSim* program just type `mcsim.perc` (if you gave it that name) under Unix. After an introductory banner (telling in particular which model file the program has been compiled with), you are prompted for an input file name: type in *`perc.lsodes.in`* (see Section B.4 [perc.lsodes.in], page 78, to see this file in Appendix), then a space, and then type in the output file name: *`perc.lsodes.out`*. After a few seconds or less (depending on your machine) the program announces that it has finished and that the output file is `perc.lsodes.out`. You can open the output file with any text editor or word processor, you can edit it for input in graphic programs *etc.*

You can try the various demonstration models provided in the `examples` directory and observe the output you obtain. You can then start programming you own models and doing simulations. The next sections of this manual reference the syntax for model definition and simulation specifications.

5 Writing and Compiling Structural Models

The model generator, "*mod*", was created to facilitate structural model definition and maintenance, while keeping short execution time through compilation. This chapter explains how to use `mod`, and how to code your models using a simplified syntax that `mod` can translate in C (creating thereby a `model.c` file).

After compiling and linking of the newly created `model.c` file together with the other C files of the `mcsim/sim` directory (or after linking with a dynamic library `libmcsim.so`), an executable simulation program is created, specific of your particular model. These preprocessing and compilation steps can be performed in Unix with a single shell command `makemcsim` (in which case, the `model.c` is created only temporarily and erased after that).

Several examples of model simulation files are included in the `mcsim/examples` directory. Some of them are reproduced in Appendix (see Appendix B [Examples], page 71).

5.1 Using `mod` to preprocess model description files

The `mod` program is a stand-alone facility. It takes a model description file in the "user-friendly" format described below (see Section 5.3 [Syntax of mod files], page 20) and creates a C language file `model.c` which you will compile and link to produce the simulation program. `Mod` allows the user to define equations for the model, assign default values to parameters or default initial values to model variables, and to initialize them using additional algebraic equations. `Mod` lets the user create and modify models without having to maintain C code. Under Unix or GNU/Linux, the command line syntax for the mod program is:

```
mod [input-file [output-file]]
```

where the brackets indicate that the input and output filenames are optional. If the input filename is not specified, the program will prompt for both. If only the input filename is specified, the output is written by default to the file `model.c`. Unless you feel like doing some makefile programming, we recommend using this default since the makefile for *GNU MCSim* assumes the C language model file to have this name. You have to have prepared a text file containing a description of the model following the syntax described in the following (see Section 5.3 [Syntax of mod files], page 20).

The following options are available:

- -h, -H gives a short online help.
- -R generate a C file of the format requested for use by the `deSolve` package of the `R` software for statistical analysis; `deSolve` implements differential equations solvers with interesting capabilities.

Most error messages given by `mod` are self-explanatory. Where appropriate, they also give the line number in the model file where the error occurred. Beware, however, of cascades of errors generated as a consequence of a first one; so don't panic: start by fixing the first one and rerun `mod`. Note that when using the -R option, care has to be taken to adopt the `deSolve` code conventions (see the `deSolve` manual on R CRAN). If you get really stuck you can send a message to the help mailing list (see Chapter 3 [Installation], page 13) or to the authors of this manual.

5.2 Using `makemcsim` to preprocess and compile model files

`makemcsim` is a Unix `sh` shell script that further facilitates preprocessing and compilation. You run `makemcsim` by entering it at the command prompt:

```
makemcsim [model-file]
```

where the brackets indicate that the model filename is optional. If a model filename is not specified, the first file having extension `.model` (by alphabetical order) is used. Makemcsim calls `mod` if the model file has changed since last compilation, compiles the `model.c` generated, links it to the shared `libmcsim.so` library to create an executable `mcsim.<root-model-name>`. The extension `root-model-name` corresponds to your model filename (without its last extension if it has one; *i.e.,*typically, without the `.model` extension). The `model.c` file is deleted afterward; if you want to inspect it (for example, if you got error messages from `mod`), run `mod` on your model definition file.

Three variants of `makemcsim` are also available:

- `makemcsimp`, which creates a parallelized standalone version to run with a MPI wrapper, such as `mpirun` (no dynamic `libmcsim.so` library loaded).
- `makemcsims`, which creates a standalone version of your model (no dynamic `libmcsim.so` library loaded, no parallelization).
- `makemcsimd`, which creates a non-parallelized standalone version with debugging symbols included (so that you can use `gdb`, for example, to check what the code actually does).

5.3 Syntax of the model description file

The model description file is a text (ASCII) file that consists of several sections, including global declarations, dynamics specifications (with derivative calculations), model initialization ("scaling"), and output computations. Here is a template for such a file (for further examples see Appendix B [Examples], page 71):

```
# Model description file (this is a comment)
<Declarations of global variables>
Initialize {
  <Equations for initializing or scaling model parameters>
}
Dynamics {
  <Equations for computing derivatives of the state variables>
}
Jacobian {
  <Equation for the Jacobian of the state derivatives>
}
CalcOutputs {
  <Equations for computing output variables>
}
End. # mandatory ending keyword
```

`Initialize`, `Dynamics`, `Jacobian` and `CalcOutputs` are reserved keywords and, if used, must appear as shown, followed by the curly braces which delimit each section (see Section 5.3.7 [Model initialization], page 29; Section 5.3.8 [Dynamics section], page 30; Section 5.3.10 [Output calculations], page 31). Please note that at least one of the sections `Dynamics` or `CalcOutputs` should be defined, and that `Dynamics` must be used if the model includes differential equations. Finally the model definition file must have the `End` keyword at the beggining of a line, eventually preceeded by white spaces or tabs. Text after the `End` keyword is ignored.

5.3.1 General syntax

The general syntax of the model description file is as follows:

- Comments begin with a pound sign (`#`) and continue to the end of the line.
- Blank lines are allowed and ignored.
- All commands can span several lines and are terminated by a semi-colon (`;`).
- Four types of variables are used: state variables, output variables, input variables, and parameters (see Section 5.3.2 [Declarations of global variables], page 23). The name of a variable should be a valid C identifier, starting with a letter or underscore (`_`) and followed by any number of alpha-numeric characters or underscores, up to a maximum of 80. Variable names are case sensitive. Note that the names *`IFN`* and *`IGS`*, in capital letters, are reserved by the program and should not be used as a parameter or variable name.
- Variable assignments have the following syntax:

  ```
  <variable-name> '=' <constant-value-or-expression> ';'
  ```

  The equal sign is needed. The right-hand side expression can be a valid C mathematical expression including numerical constants, already defined variables, standard ANSI C mathematical functions (`acos`, `asin`, `atan`, `atan2`, `ceil`, `cos`, `cosh`, `exp`, `fabs`, `floor`, `fmax`, `fmin`, `fmod`, `log`, `log10`, `pow`, `sin`, `sinh`, `sqrt`, `tan`, `tanh`), and *GNU MCSim*'s "*special functions*" (see Section 5.3.4 [Special functions], page 25) or "*input functions*" (see Section 5.3.5 [Input functions], page 27). Special functions can take already defined variables, constant numerical values or expressions as parameters. Input functions can only be used on the right hand side of assignments to input variables.

  Colon conditional assignments have the following syntax:

  ```
  <variable-name> = (<test> ? <value-if-true> : <value-if-false>);
  ```

  For example:

  ```
  Adjusted_Param = (Input_Var > 0.0 ? Param * 1.1 : Param);
  ```

  In this example, if '`Input_Var`' is greater than 0, the parameter '`Adjusted_Param`' is computed as the product of '`Param`' by '`1.1`'; otherwise '`Adjusted_Param`' is equal to '`Param`'. Note that conditional assignments can be nested (*i.e.*, <value-if-true> or <value-if-false> can themselves be a conditional expression). The comparison operators allowed are the equality operator `==`, and non-equality operators `!=`, `<`, `>`, `<>`, `<=` and `>=`.

  More complex conditions can also be specified, but the Boolean AND, OR and NOT operations have not yet been implemented. You can use:

  `(('A'*'B')>0)` for AND `(('A'+'B')>0)` for OR `('A'==0)` for NOT

- Vectors: You can use vectors (arrays) in your model code or simulation definition files to simplify it.

  Declaring vectors: To declare a state variable, an input, an output, or a parameter as a vector, use the one of the two following syntaxes when you first define it:

  ```
  <variable-name> '[' <integer> ']'
  <variable-name> '[' <integer> '-' <integer> ']'
  ```

  The variable name is immediately followed by an opening square bracket ('['). The array index bounds (which define the valid indices) can be given as (long) positive or null integers separated by an hyphen ('-') (spaces are allowed). In this case the second integer must be higher the first. They are followed by a closing bracket (']'). The hyphen and second integer are optional. If only one bound (integer) is given, only the component with corresponding index is declared. Both syntaxes can be mixed. For example:

  ```
  States = {y[0-9]};
  alpha[0-2] = 1;
  beta[0] = 1;
  beta[1] = 2;
  beta[2-4];
  ```

  The previous lines define a state variable '`y`' as a vector of length 10, with valid indices ranging between 0 and 9, included. The parameter vector '`alpha`' is defined with range 0 to 2, each component being initialized to value 1. For parameter '`beta`', components 0, 1 and 2 to 4 are initialized separately (components 2 to 4 are initialized with default value 0).

  Accessing vectors' components: After declaration, vector's components can be accessed individually using the square bracket syntax:

  ```
  <variable-name> '[' <integer> ']'
  ```

  For example:

  ```
  Outputs = {x[0-1]};
  beta[0] = 0;
  beta[1] = beta[0] + 1;
  CalcOutputs {
    x[0] = beta[0] * t;
    x[1] = beta[1] * t;
  }
  ```

  In the above example, '`beta[0]`', '`beta[1]`', '`x[0]`', and '`x[1]`' are accessed individually. The variable '`t`' refers to the implicit variable 'time'.

  Vectorization of equations: The equations specifying the model, which consist in assignments, can be vectorized in the `Initialize`, `Dynamics` and `CalcOutputs` sections (but not in the global section) (see Section 5.3.2 [Declarations of global variables], page 23). Vectorization allows you to specify an operation for an entire vector or parts of it. The following syntax should be used:

  ```
  <var-name>'['<integer>'-'<integer>']' = <vectorized-expression>;
  ```

  On the right-hand side, the vectorized expression should be a valid C mathematical expression including numerical constants, already defined state, input, output, other (parameter) variables or vectors, and standard ANSI C mathematical functions or special functions (see Section 5.3.4 [Special functions], page 25). Here also, input functions (see Section 5.3.5 [Input functions], page 27) can only be used on the right hand side of assignments to input variables. Vector indices on the right-hand side can take the special form of "*bracketed expressions*". Bracketed expressions can be composed of integers, the 4 basic arithmetic operators ('+', '-', '*', '/'), parentheses and the index letter 'i'. The running index 'i' points in turn to each component in the range specified on the left-hand side (imagine that the range given on the left-hand side corresponds to a 'for' loop with index 'i' running from the lower bound to the upper bound). This is best understood by looking at some code. In the previous example, the assignments to x[0] and x[1] obviously deserve vectorization. This is achieved by the following statements:

```
CalcOutputs {
  x[0-1] = beta[i] * t;
}
```

Here, the index 'i' refers to the values 0 and 1. Here is another example:

```
Outputs{x[1-10]};
CalcOutputs {
  x[1] = 0;
  x[2-10] = x[i-1] + 1;
}
```

This is equivalent to:

```
Outputs{x[1-10]};
CalcOutputs {
  x[1] = 0;
  x[2] = x[1] + 1;
  ...
  x[10] = x[9] + 1;
}
```

and will assign value 1 to '`x[2]`', 2 to '`x[3]`', *etc.* On the right-hand side, more complicated bracketed expressions like '`[(2*i-1)/(i+3)]`' can be used. Another, working, example of vector use is given in the `mcsim/examples/pde2` directory.

Alternative 'underscore' ('_') syntax: Individual vector components can be declared and used (everywhere in the model file) with the following syntax:

```
<variable-name>'_'<integer>
```

The integer indicates which component of the vector is referred to. For example '`x_1`' is strictly equivalent to '`x[1]`'. Note!: No space are allowed between the variable name, the underscore and the integer.

- The `End` keyword must used to indicate model termination.

5.3.2 Declarations of global variables

Commands not specified within the delimiting braces of another section are considered to be global declarations. In the global section, you first declare the state, input, and output variables. There should be at least one state or output variable in your model.

- States are variables for which a first-order differential equation is defined in the `Dynamics` section (see Section 5.3.8 [Dynamics section], page 30) (higher orders or partial differential equations are not allowed).
- Inputs are variables independent of the others variables, and may change with time (for example an exposure concentration to a chemical).
- Outputs are dependent model variables, obtainable at any time as analytical functions of the states, inputs or parameters. They must receive assignments in either the `Dynamics` or `CalcOutputs` sections. You should not assume that their past values are accessible and correct: they have to be calculated in the same section where they are used and before being used.

The format for declaring each of these variables is the same, and consists of the keyword `States`, `Inputs` or `Outputs` followed by an equal sign and a list of the variable names enclosed in curly braces as shown here:

```
States =   {Qb_fat,  # Benzene in the fat
            Qb_bm,   # ...     in the bone marrow
            Qb_liv}; # ...     in the liver and others

Inputs =   {Q_gav,   # Gavage dose
            C_inh};  # Inhalation concentration

Outputs =  {Cb_exp,  # Concentration in expired air
            Cb_ven}; # ...           in venous blood
```

After being defined, states, inputs and outputs can then be given initial values (constants or expressions). Inputs can also be assigned input functions, described below (see Section 5.3.5 [Input functions], page 27). Some examples of initialization are shown here:

```
Qb = 0.1; # Default initial value for state variable Qb

# Input variable assigned a periodic exponential input function
Q = PerExp(1, 60, 0, 1); # Magnitude of 1.0,
                         # period of 60 time units,
                         # T0 in period is 0,
                         # Rate constant is 1.0
```

If a state, input, or output variable is not explicitly given an initial value, that value will be set to zero by default. Initial values are reset to their specified value by the simulation program at the start of each `Simulation` (see [Simulation sections], page 52).

All the other variables are "*parameters*". Model parameters you want to be able to change in simulation input files should be declared in the global section. For example:

```
Wind_speed; # (m/s) Local wind speed
```

Parameters are by default assigned a value of zero. To assign a different nominal values, use the assignment rules given above. For example:

```
BodyWt = 65.0 + sqrt(15.0); # Weight of the subject (in kg)
```

All parameters and variables are computed in double precision floating-point format. Initial values should not be such as to cause computation errors in the model equations; this is likely to lead to crashing of the program (so, for example, do not assign a default value of zero to a parameter appearing alone in a denominator). Note that the order of global declarations matters within the global section itself (*i.e.*, parameters and variables should be defined and initialized before being used in assignments of others), but not with respect to other blocks. A parameter defined at the end of the description file can be used in the `Dynamics` section which may appear at the beginning of the file. Still, such an inverse order should be avoided. For this reason, the format above, where global declarations come first, is strongly suggested to avoid confusing results. Note again that the name `IFN`, in capital letters, is reserved by the program and should not be used as parameter or variable name. Finally, if a parameter is defined several times, only the first definition is taken into account (a warning is issued, beware of it).

5.3.3 Model types

This section deals with structural models. Statistical models that you setup for model calibration and data analysis are defined in the simulation input files, through statistical distribution functions. They are dealt with later in this manual (see Section 6.2.5 [Setting-up statistical models], page 55).

GNU MCSim can easily deal with purely algebraic structural models. You do not need to define state variables or a `Dynamics` section for such models. Simply use input and output variables and parameters and specify the model in the `CalcOutputs` section. You can use the time variable `t` if that is natural for your model. If your model does not use `t`, you will still need to specify "output times" in `Print()` or `PrintStep()` statements to obtain outputs: you can use arbitrary times. If you do not use `t` as "independent" model variable, you will also need do define a `Simulation` section (see [Simulation sections], page 52) for each combination of values for the independent variables of your model. This may be clumsy if many values are to be used. In that case, you may want to use the variable `t` to represent something else than time.

Ordinary differential models, with algebraic components, can be easily setup with *GNU MCSim*. Use state variables and specify a `Dynamics` section. Time, `t` is the integration variable, but here again, `t` can be used to represent anything you want. For partial differential equations some problems might be solved by implementing line methods (see examples in `mcsim/examples/pde1` and `mcsim/examples/pde2`)...

You can use *GNU MCSim* for discrete-time dynamic models (or difference models). That is a bit tricky. Assignments in the `CalcOutputs` section are volatile (not memorized), so the model equations have to be given in a `Dynamics` section. But the model variables should still be declared as outputs, because they should not be updated by integration. However, you need at least one true differential equation in the `Dynamics` section, so you should declare a dummy state variable (and assign to its derivative a constant value of zero). You also want the calls to `Dynamics` to be precisely scheduled, so it is best to use the `Euler` integration routine (see [Integrate() specification], page 41) which uses a constant step. Since `Euler` may call repeatedly `Dynamics` at any given time, you want to guard against untimely updating... Altogether, we recommend that you examine the example files in the `mcsim/examples/discrete` directory provided with the source code for *GNU MCSim*.

5.3.4 Special functions

The following special functions (whose name is case-sensitive) are available to the user for assignment of parameters and variables in the model definition file:

- `BetaRandom(`*`alpha`*`, `*`beta`*`, a, `*`b`*`)`: returns a Beta distributed variate on the interval [a,*b*] with shape parameters *alpha* and *beta*;
- `BinomialBetaRandom(`*`E`*`, `*`alpha`*`, `*`beta`*`)`: return random variate, of mathematical expectation *E*, drawn from a binomial distribution with probability *p*, *p* being Beta distributed with parameters *alpha* and *beta*;
- `BinomialRandom(`*`p`*`, `*`N`*`)`: returns a binomially distributed random variate;
- `CauchyRandom(`*`s`*`)`: returns a Cauchy distributed random variate with scale *s*;
- `CDFNormal(`*`x`*`)`: the normal cumulative density function;
- `Chi2Random(`*`dof`*`)`: returns a Chi-squared random variate with *dof* degrees of freedom;
- `erfc(`*`x`*`)`: the complementary error function;
- `ExpRandom(`*`beta`*`)`: returns an exponential variate with inverse scale *beta*;
- `GammaRandom(`*`alpha`*`)`: returns a gamma distributed random variate with shape parameter *alpha* and inverse scale equal to 1;
- `GetSeed()`: returns the current value of the random generator seed (this function is not available if the GNU Scientific Library is used; if called in that case, it exits with an error message);
- `GGammaRandom(`*`alpha`*`, `*`beta`*`)`: returns a gamma distributed random variate with shape parameter *alpha* and inverse scale *beta*;
- `InvGGammaRandom(`*`alpha`*`, `*`beta`*`)`: returns an inverse gamma distributed random variate with shape parameter *alpha* and scale parameter *beta*;
- `lnDFNormal(`*`x`*`, `*`mean`*`, `*`sd`*`)`: the natural logarithm of the normal density function;
- `lnGamma(`*`x`*`)`: the natural logarithm of the gamma function;
- `LogNormalRandom(`*`mean`*`, `*`sd`*`)`: returns a lognormally distributed variate with geometric mean *mean* and geometric standard deviation *sd* (*i.e.*, the log of the returned variate is normally distributed with mean ln(*mean*) standard deviation ln(*sd*);
- `LogUniformRandom(a, `*`b`*`)`: returns variate log-uniformly distributed on the interval [a,*b*];
- `NegativeBinomialRandom(`*`r`*`, `*`p`*`)`: returns a negative binomial random variable with prescribed number of failures, *r*, until trials are stopped, and probability of failure, *p*, for each trial The probability mass function of this distribution for the number k of successes is equal to:

  $$C(k+r-1,k)p^r(1-p)^k$$

  where C() is the binomial coefficient;
- `NormalRandom(`*`mean`*`, `*`sd`*`)`: returns a normally distributed random variable with prescribed mean and standard deviation;
- `PiecewiseRandom(`*`min`*`, a, `*`b`*`, `*`max`*`)`: the distribution of the returned variate has the form of a truncated triangle, with base from *min* to *max* and a plateau between a and *b*. If $a = b$, the distribution is the triangular distribution;
- `PoissonRandom(`*`mu`*`)`: returns a Poisson-distributed random variate, of rate *mu*;
- `SetSeed(`*`seed`*`)`: sets the current value of the pseudo-random generator seed to the specified *seed*. When using the GNU Scientific Library, that *seed* can be any positive integer (including zero). When using the native code, it can be any positive real number

(seeds between 1.0 and 2147483646.0 are used as is, the others are rescaled within those bounds, and a warning is issued);

- `StudentTRandom(`*`dof`*`, `*`mean`*`, `*`sd`*`)`: returns a Student *t* distributed random variate with *dof* degrees of freedom and given *mean* and standard deviation;
- `TruncInvGGammaRandom(`*`alpha`*`, `*`beta`*`, a, b)`: returns a truncated inverse gamma distributed random variate with shape parameter *alpha* and scale *beta*, in the range [*a*,*b*]. Explicit specification of *a*,*b* is required;
- `TruncLogNormalRandom(`*`mean`*`, `*`sd`*`, a, b)`: returns a truncated lognormal variate with geometric mean *mean* and geometric standard deviation *sd*, in the range [*a*,*b*]. Explicit specification of *a*,*b* is required;
- `TruncNormalRandom(`*`mean`*`, `*`sd`*`, a, b)`: returns a truncated normal variate with prescribed mean and standard deviation, in the range [*a*,*b*]. Explicit specification of *a*,*b* is required;
- `UniformRandom(`*`min`*`, `*`max`*`)`: returns a uniformly distributed random variable, sampled between min and max.

Note: If you have linked Graph_Sampler with GNU Scientific Library (gls), all the above random number generating functions use an extremely long period random number generator: the Mersenne twister generator (gsl_rng_mt19937). Otherwise, the random generator used is that of Park and Miller (Barry, 1996; Park and Miller, 1988; Vattulainen et al., 1994) (see [Bibliographic References], page 65). When using the GNU Scientific Library, the default random generator seed value is 0. When using the native code, it is 314159265.3589793. Those default values can be overridden with the function `SetSeed`.

Note also that assignment of a random number generating function to a state variable derivative will define a form of stochastic differential equation. *GNU MCSim*'s integration routines are not particularly suited to the resolution of such equations. If you wish to try it anyway, you may want to consider using the "robust" Euler method (see [Integrate() specification], page 41).

5.3.5 Input functions

These functions can be used in special assignments, valid only for input variables. Inputs can be initialized to a constant or to a standard mathematical expression, or assigned one of the following input functions:

- `PerDose()` specifies a periodic input of constant <*magnitude*>. The input begins exactly at <*initial-time*> in the <*period*> and lasts for <*exposure-time*> time units. Syntax:

  ```
  <input variable> = PerDose(<magnitude>, <period>, <initial-time>,
                             <exposure-time>);
  ```

- `PerExp()` specifies a periodic exponential input. At time <*initial-time*> in the <*period*> the input rises instantaneously to <*magnitude*> and begins to decay exponentially with the constant <*decay-constant*>. Note that the input does not accumulate across periods, it resets at each period start. Syntax:

  ```
  <input variable> = PerExp(<magnitude>, <period>, <initial-time>,
                            <decay-constant>);
  ```

- `PerTransit()` models a delayed input mechanism in which a substance has to go through a chain of (linear) transfer compartments before being actually input (see Savic et al., 2007) (see [Bibliographic References], page 65). The actual input is computed as:

$$\frac{(K_{tr}t)^n}{n!}e^{-K_{tr}t}$$

where K_{tr} is the transit rate constant from compartment to compartment, and n is the number of "apparent" transit compartments (a positive real number). Factorial n is computed using Stirling's formula. This is a popular absorption model in pharmacokinetics, where the number of n and the transfer rate constant are estimated from data.

The function `PerTransit()` has 5 arguments: a <*magnitude*>; a <*period*> (for multiple dosing) at the beginning of which input is reset. Note that the input does not accumulate across periods, it resets at each period start; an <*initial-time-in-period*> in <*period*> at which dosing actually starts; a <*transfer-rate-constant*>, as defined above; a <*number-of-input-compartments*>. Its syntax is:

```
<input variable> = PerTransit(<magnitude>, <period>,
                              <initial-time-in-period>,
                              <transfer-rate-constant>,
                              <number-of-input-compartments>);
```

An demonstration of its use is given in `mcsim/examples/test_transit_input`.

- `NDoses()` specifies a number of stepwise inputs of variable magnitude and their starting times. The first argument, <*n*>, is the number of input steps and start times. Next come a list of magnitudes and a list of corresponding initial times. Each list is comma-separated. The duration of each input step is computed automatically by difference between the listed times. Currently this function can only be used in the simulation description file, and not in the model description file (which simply implies that you cannot use it as a default). Syntax:

  ```
  <input variable> = NDoses(<n>, <list-of-magnitudes>,
                            <list-of-initial-times>);
  ```

  Instead of lists of magnitudes and times, you can use vectors specifying them, as in:

  ```
  My_input = NDoses(10, My_magnitudes[1-10], My_initial_times[1-10]);
  ```

  Note that the list or vector of times *must* begin at the starting time of the simulation (typically time zero), even if the magnitude at that first time is zero.

- `Spikes()` specifies a number of instantaneous inputs of variable magnitude and their exact times of occurrence. However, a spike occuring exactly at the final simulation time will be ignored (see below the Note on discontinuity). The first argument, <*n*>, is the number of inputs and input times. Next come a list of magnitudes and a list of times. Each list is comma-separated. Currently this function can only be used in the simulation description file, and not in the model description file (which simply implies that you cannot use it as a default). Syntax:

  ```
  <input variable> = Spikes(<n>, <list-of-magnitudes>,
                            <list-of-times>);
  ```

The arguments of input functions can either be constants or variables. For example, if '`Mag`' and '`RateConst`' are defined model parameters, then the input variable '`Q_in`' can be defined as:

```
Q_in = PerExp(Mag, 60, 0, RateConst);
```

In this way the parameters of input functions can, for example, be assigned statistical distributions in Monte Carlo simulations (see [Distrib() specification], page 48). Variable dependencies are resolved before each simulation specified by a `Simulation` section (equivalently `Experiment`) (see [Simulation sections], page 52).

For each of the periodic functions, a single exposure beginning at time *initial-time* can be specified by giving an effectively infinite period, *e.g.* 10^{10}. The first period starts at the initial time of the simulation.

Note on discontinuities: Magnitudes change exactly at the times given. At a discontinuity, the time point belongs to the NEW time period UNLESS we are at the final time.

Input variables assigned input functions can be combined to give a lot of flexibility (*e.g.*, an input variable can be declared as the sum of others). Separate inputs can also be declared in the global section of the model definition file and combined in the `Dynamics` (see Section 5.3.8 [Dynamics section], page 30) and `CalcOutputs` (see Section 5.3.10 [Output calculations], page 31) sections.

5.3.6 In line functions

`Inline()` functions can be placed in the various sections of a model file to introduce standard C code (or whatever) in your models. Text placed between the parentheses of an `Inline` function will be passed *as is* to the C compiler. That text can span several lines but its size should not exceed MAX_EQN (defined in `lex.h`); In case it does, you can increase MAX_EQN (and recompile `mod`...) or you can split you text between any number of `Inline()` in a row. It is your responsibility to make sure that the code passed can be compiled without errors!

Example:

```
Inline( printf("hello/n"); );
```

You can use C compiler (in fact preprocessor) directives, such `#include` directives, inside `Inline` statements, provided that you use the `\#` escape sequence instead of simply the `#` character (otherwise it is confused with a *GNU MCSim* comment and Hell breaks loose).

Example:

```
Inline( \#include <gsl/gsl_interp.h> );
```

This can be useful to include large amounts of C code. Note also that the inlined code is likely to be dependent on whether or not you are using the `-R` option of `mod`.

5.3.7 Model initialization

The model initialization section begins with the keyword `Initialize` (the keyword `Scale` is obsolete but is still understood) and is enclosed in curly braces. The equations given in this section will define a function (subroutine) that will be called by *GNU MCSim* after the assignments specified in each `Simulation` section are done (see [Simulation sections], page 52). They are the last initializations performed. The model file in `mcsim/examples/perc` gives an example of the use of `Initialize` (see Section B.3 [perc.model], page 73, in Appendix).

All model variables and parameters, except inputs, can be changed in this section. Modifications to state variables affect initial values only. In this section, state variables, outputs and parameters (but not input variables) can also appear at the the right-hand side of equations. The integration variable can be accessed if referred to as `t`

Warning: Assignments to state variables in the `Initialize` section override the same assignments made in input files.

Additional parameters (to those declared in the global section) may be used within the section. They will be declared as local temporary variables and their scope will be limited to the `Initialize` section (*i.e.*, their value and existence will be forgotten outside the section).

The `dt()` operator (see Section 5.3.8 [Dynamics section], page 30) cannot be used in this section, since derivatives have not yet been computed when the scaling function is called.

5.3.8 Dynamics section

The dynamics specification section begins with the keyword `Dynamics` and is enclosed in curly braces. The equations given in this section will be called by the integration routines at each integration step. `Dynamics` must be used if the model includes differential equations.

Additional parameters (to those declared in the global section) may be used for any calculations within the section. They will be declared as local temporary variables. (Note, for example, the use of '`Cout_fat`' and '`Cout_wp`' in the `perc.model` example file). Local variables are not accessible from the simulation program, or from other sections of the model definition file, so don't try to output them.

Each state variable declared in the global section must have one corresponding differential equation in the `Dynamics` section. If a differential equation is missing, `mod` issues an error message such as:

```
Error: State variable 'Q_foo' has no dynamics.
```

and no `model.c` file or executable program will be created.

The derivative of a state variable is defined using the `dt()` operator, as shown here:

```
dt(state-variable) '=' constant-value-or-expression ';'
```

The right-hand side can be any valid C expression, including standard math library calls and the special functions mentioned above (see Section 5.3.4 [Special functions], page 25). Note that no syntactic check is performed on the library function calls. Their correctness is your responsibility.

The `dt()` operator can also be used in the right-hand side of equations in the dynamics section to refer to the value of a derivative at that point in the calculations. For example:

```
dt(Qm_in) = Qmetabolized - dt(Qm_out);
```

The integration variable (*e.g.*, time) can be accessed if referred to as `t`, as in:

```
dt(Qm_in) = Qmetabolized - t;
```

Output variables can also be made a function of `t` in the `Dynamics` section.

Note that while state variables, input variables and model parameters can be used on the right-hand side of equations, they cannot be assigned values in the `Dynamics` section. If you need a parameter to change with time, you can declare it as an output variable in the global section. Assignments to states, inputs or parameters in this section causes an error message like the following to be issued:

```
Error: line 48: 'YourParm' used in invalid context.
Parameters cannot be defined in Dynamics{} section.
```

Note also that, in `Dynamics`, output variables should be assigned a value or equation before being used. You cannot assume that the value computed during a previous call to the function is valid, because the integrator may have taken too long a time step, for example. The integrator keeps track of state variables only.

5.3.9 Delay differential equations

GNU MCSim can solve delay differential equations.

Delay differential equations are equations that depend on *past* values of the state variables, say at time *t-tau* instead of at time *t*.

This is done very easily in *GNU MCSim* models with the `CalcDelay` funtion. Its syntax is:

```
CalcDelay (<variable>, <delay>);
```

The `variable` must be a declared state or output variable. `CalcDelay()` will return its past value (at time *delay*). The *delay* specified must be either a declared parameter or a constant floating point or integer value. For example:

```
tau = 100;
dt (Q1) = k * CalcDelay(Q3, tau);
dt (Q2) = k * CalcDelay(Q3, 10);
```

An example of a model using `CalcDelay()` and input file is given in the `example/delay_diff_eqns` folder. Note that currently, the `CalcDelay()` function cannot be used with the `-R` option for `mod` (*i.e.*, for use with the deSolve R package).

GNU MCSIm stores required variables past values in a arrays of size MAX_DELAY (equal by default to 1000). If the needed past recall exceeds that capacity you will need to increase the value of MAX_DELAY in the C file `delays.c` and recompile the library and your model.

5.3.10 Output calculations

The output calculation section begins with the keyword `CalcOutputs` and is enclosed in curly braces. The equations given in this section will be called by the simulation program at each output time specified by a `Print()` or `PrintStep()` statement (see [Print() specification], page 54, and see [PrintStep() specification], page 54). In this way, output computations are done efficiently, only when values are to be saved.

Only variables that have been declared with the keyword `Outputs`, or local temporary variables, can receive assignments in this section. As in the `Dynamics` section, output variables should be assigned a value or equation before being used. Assignments to other types of variables cause an error message like the following to be issued:

```
Error: line 56: 'Qb_fat' used in invalid context.
Only output and local variables can be defined in CalcOutputs section.
```

Any reference to an input or state variable will use the current value (at the time of output). Note that `CalcOutputs` code is called only on the correct state trajectory, but you cannot reuse past values of outputs in your code, for example forming cumulative sums such as:

```
a = a + b;
```

The `dt()` operator can appear in the right-hand side of equations, and refers to current values of the derivatives (see Section 5.3.8 [Dynamics section], page 30). Like in the `Dynamics` section, the integration variable can be accessed if referred to as `t`, as in:

```
Qx_out = DQx * t;
```

5.3.11 Comments on style

For your model file to be readable and understandable, it is useful to use a consistent notation style. The example file `perc.model` tries to follow such a style (see Section B.3 [perc.model], page 73). For example we suggest that:

- All variable names begin with a capital letter followed by meaningful lower case subscripts.
- Where two subscripts are necessary, they can be separated by an underscore, such as in '`Qb_fat`'.
- Where there is only one subscript an underscore can still be used to increase readability as in '`Q_fat`'.
- Where two words are used in combination to name one item, they can be separated visually by capitalizing each word, as in '`BodyWt`'.

These conventions are suggestions only. The key to have a consistent notation that makes sense to you. Consistency is one of the best ways to:

1. Increase readability, both for others and for yourself. If you have to suspend work for a month or two and then come back to it, the last thing you want is to have to decipher your own file.
2. Decrease the likelihood of mistakes. If all of the equations are coded with a consistent, logical convention, mistakes stand out more readily.

Last, but not least, do use comments to annotate your code! Also: make sure your comments are accurate and update them when you change your code. In our experience, an enormous number of hours has been wasted in trying to figure out inconsistencies that existed only because of inaccurate comments (*e.g.*, erroneous comments about the reasons for choice of default parameter values). That does not decrease the value of good comments, however...

5.4 Reading SBML models and applying a template

To read models written in SBML, you simply need to create a text file of the same format as an *MCsim* model definition file (comments starting with `#`, file ending with `End`, *etc.*) In that file, you specify a list of one or more SBML model files with the keyword `SBMLModels` followed by an equal sign and a list of SBML model file names (of maximum size MAX_FILENAMESIZE, 80 characters, as defined in mod.h), enclosed in curly braces, as shown here:

```
#--------------------------------------------------------------
# SBML_List.in
# Use it as "mod SBML_List.in"
#--------------------------------------------------------------
```

```
SBMLModels = {useID, "default", "C_central.xml", "C_periph.xml"};

End.
```

The first two tokens, `useID` and `"default"`, at the very beginning of the list, are optional (but if the appear they must both appear). The first indicates that species are recognized by "IDs" (at the moment this the only possibility, hardly an option indeed, but in the future we would like to be able also to use Names"). The second one gives the name of the default (external) compartment in your SBML files. Here the default is named "default", which works for CellDesigner, but JDesigner, for example, uses "compartment"... The list of files comes after, with the filenames being enclosed with double quotes. There is no restriction about their extension (".xml" is just an example). They just have to be valid filenames.

Give that file as input to `mod`, by typing in `mod SBML_List.in` on your shell command line. A `model.c` file is produced, suitable for further compilation by `makemcsim`. SBML model files are typically ASCII text files with a *xml* extension (for examples see `mcsim/sim/examples/SBML`). SBML Level 1 and Level 2 are recognized (but some features of Level 2 are not yet understood by `mod`, such as `functionDefinition`, `unit`, `rateRule` and a few others.) Omitted reaction stoichiometries are set to value 1 by default. Compartments are ignored unless a model template for circulating species is given (see below).

If two or more of the SBML files define a same chemical species, `mod` merges the models: namely, the rate equation for that species will be the sum of the rate equations implied by each model. In fact, `mod` constructs the rate equations from the reaction descriptions given by the SBML format. State variables and parameters keep the same name after merging, so they should be unique from the start to each SBML model, to avoid confusion.

To automatically extend SBML (level 2) models (*e.g.*, with transport component terms, an example which will be used throughout this part of the manual), a template model can be applied to all the species defined in SBML which are placed in the default compartment (conventionally called `compartment`), and outside of other compartments whose names are specified by the template itself. Specifically: the template model should define compartments and differential equations terms for the transport of an unspecified species between those compartments (the syntax for that will be described below). For each species placed in SBML *outside* one of the defined compartments, a differential equations is created for each compartment using the template transport terms. The differentials for the compartments found in both the template and the SBML models will contain both transport and kinetic terms. With the exception of the generic `compartment`, the SBML models merged can only use the compartments defined by the template (compartment name recognition is case-sensitive). Species placed in SBML *inside* one of the defined compartments are not transported and stay local to that compartment. Their differentials will contain only the reaction kinetics terms defined by the SBML models. For simplicity, the default initial values of the model state variables (species) specified by the user are ignored and set to zero.

Warning: At the moment, libSBML does not guarantee the type of operands or their format for a division. While `5.4/2.3` will appear correctly as a float division, `5.0/2.0` is likely to be translated in `5/2` which is an integer division in C! It is probably best to use the multiplication by the inverse in that case, but that forces you to check SBML models manually for all divisions...

There might be many applications of the template mechanism, and they are left to the reader to imagine.

The template model to use is specified with `PKTemplate` keyword, followed by an equal sign and the file name of a template model file (of maximum size MAX_FILENAMESIZE, 80 characters, as defined in mod.h), enclosed in curly braces, like in:

```
#---------------------------------------------------------
# SBML_List.in
# Use it as "mod SBML_List.in"
#---------------------------------------------------------

PKTemplate = {"input_f/2cpt_PBPK.model"};

SBMLModels = {"C_central.xml", "C_periph.xml"};

End.
```

The template structure is similar to that of other *GNU MCSim* models, with two exceptions: First, the variables and parameters which are supposed to apply to several chemical species defined in SBML are preceeded with an underscore (`_`). Second, the compartments allowed in SBML and for which the template is defined are declared using the keyword `Compartment`, followed by an equal sign and a list of the compartment names enclosed in curly braces. Here is an example of template file:

```
#---------------------------------------------------------
# Template for pharmacokinetic modeling
#---------------------------------------------------------

States = {_central, _periph};
Inputs = {_Dose_rate};
Outputs = {_Q_total};
Compartments = {central, peripheral};

# Species-dependent parameters
_PC = 2;
_K_urine;

# Species-independent parameters
V_central;
V_periph;

Initialize { _K_urine = _K_urine * 2; }

Dynamics {
  _Q_rate_c_p = 5 * (_central - _periph / _PC);

  # Central compartment quantity
  _pk_central  = (_Dose_rate - _Q_rate_c_p -
                  _K_urine * _central) / V_central;
```

```
    dt(_central) = _pk_central;

    # Peripheral compartment quantity
    _pk_periph  = _Q_rate_c_p / V_periph;
    dt(_periph) = _pk_periph;
   }

  CalcOutputs { _Q_total = _central / V_central + _periph / V_periph; }

  End.
```

With that template, each species, say *S1*, defined in SBML to be outside of the `central` or `peripheral` compartments will be cloned to form the state variables *S1_central* and *S1_peripheral*. Those will have associated differential equations using specific parameters *S1_PC* or unspecific ones like V_central. The chemical reactions defined in SBML to take place inside the central or peripheral compartments will be translated into specific terms added to the `dt(_central)` and `dt(_periph)` equations.

Warning: the automatic creation of a model by merging SBML files with a template may shuffle the order of the differential equation declarations. Therefore you should not use an already defined `dt()` term in a subsequent differential equation in the template model.

5.5 Working with the *R* package *deSolve*

GNU MCSim `mod` model generator can be passed the `-R` option. For example:

```
mod -R perc.model
```

In that case, the C code produced can be used by the `deSolve` package of the *R* statistical software (see http://www.r-project.org/) to perform simulations of your models. The numerical integators provided by deSolve are improved implementations of the lsode family of integrators used by *GNU MCSim*), and `deSolve` provides a few more options than *GNU MCSim* (see the `deSolve` user manuals). However, if you need raw speed (say, for Markov chain Monte Carlo simulations) *GNU MCSim* is probably the fastest option.

In addition to producing a `model.c` file in C language, `mod` called with the `-R` option also generates `model_inits.R` file. That file can be loaded in *R* and provides the *R* two functions `initParms`, `initStates` and the variable *Outputs*, which can be handy in *R* scripts:

- `initParms()` without parameters reset the model parameters to their default values. The *newparms* parameter takes a vector of named parameters and values, assign the given values to the corresponding model parameters and reset the others to their default values.
- `initStates()` with just the *parms* parameter reset the model initial states to their default values. If the *newparms* parameter is used it takes a vector of named states and values, assign the given values as initial values to the corresponding model state variables and reset the others to their default values.
- *Outputs* is simply an array of output variable names.

An example of R script using `GNU MCsim` to generate and run a model is given in the `example/R` folder.

6 Running Simulations

After having your model processed by `mod` or `makemcsim`, and obtained an executable `mcsim_...` file, you are ready to run simulations. For this you need to write simulation files. This chapter explains how to write such files with the proper syntax and how to run the executable program.

You may want to first give a look at the examples given in the `mcsim/examples` directory. An example file `perc.lsodes.in`, which works with the perchloroethylene model `perc.model`, is also given in an Appendix to this manual (see Section B.4 [perc.lsodes.in], page 78).

6.1 Using the compiled program

GNU MCSim provides several types of simulations for the models you create. Simulations are specified in a text file of format similar to that of the model description file.

Assume that your model `a.model` has been preprocessed and compiled by `makemcsim` (see Section 5.2 [Using makemcsim], page 20) to generate an executable `mcsim_a` (if you have renamed the executable file, substitute `mcsim_my` by the name of your executable in the following). In Unix the command-line syntax to run that executable is simply:

```
mcsim_a [run-time options] [input-file [output-file]]
```

where the brackets indicate optional arguments. If no input and output file names are specified, the program will prompt you for them. You must provide an input file name. That file should describe the simulations to perform and specify which outputs should be printed out (see Section 6.2 [Syntax of simulation files], page 38). If you just hit the return key when prompted for the output name, the program will use the name you have specified in the input file, if any, or a default name (see [OutputFile() specification], page 41). If just one file name is given on the command-line, the program will assume that it specifies the input file. For the output filename, the program will then use the name you have specified in the input file, if any, or a default name.

The following run-time options are available:

- `-c` displays the Gelman-Rubin's MCMC convergence diagnostic as simulated chains progress in parallel on different processors (if MPI is used).
- `-D=print-hierarchy` or `-D print-hierarchy` prints out the details of a multilevel statistical model for debugging (see [Level sections], page 57).
- `-h` or `-H` display a short help on the program use.
- `-i=<arg>` or `-i <arg>`, where `arg` is an integer, prints out the number of simulations done, every `<arg>` iteration.

Note on parallelized code: If your model has been preprocessed and compiled by `makemcsimp` (see Section 5.2 [Using makemcsim], page 20) to generate a parallelization-enabled executable `mcsim_a`, you should run it with a specific wrapper like `mpirun`:

```
mpirun -np 50 mcsim_a [run-time options] [input-file [output-file]]
```

The particular wrapper to use depends on the MPI library installed on your system. In the case of Open MP `mpirun`, the number of processors to use is specified by the `-np` option (50 processors in the example above).

When the program starts, it announces which model description file was used to create it. While the input file is read or while simulations are running, some informations will be printed on your computer screen. They can help you check that the input file is correctly interpreted and that the program runs as it should. *GNU MCSim* can also post error messages, which should be self-explanatory. Where appropriate, they show the line number in the input file where the error occurred. Beware, however, of cascades of errors generated as a consequence of a first one; also errors may be detected after the line in which they really occur and the line number shown will be unhelpful; don't panic: start by fixing the first error in the input file and rerun your executable. You should not need to recompile your executable, unless you have changed the model itself. If you get really stuck you can send a message to the mailing list "help-mcsim@prep.ai.mit.edu" (see Chapter 3 [Installation], page 13) or to the authors of this manual.

The program ends (if everything is fine) by giving you the name of the output file generated. If you want to run the program in batch mode (in the background), you may want to redirect the screen output and error messages; refer for this to the `man` pages for your shell.

6.2 Syntax of the simulation definition file

A simulation specification file is a text (ASCII) file that consists of several sections, starting with global specifications and assignments (valid throughout the file), followed by a number of `Simulation` sections (see [Simulation sections], page 52), eventually enclosed in `Level` sections. (The keyword `Experiment` is now obsolete but can still be used as a synonym for `Simulation`.)

It is important to note that all simulations are fundamentally dynamic, and that time is the explicit independent variable. There can be time-discontuities in your simulations if a input changes instantaneously (see Section 5.3.5 [Input functions], page 27). At a discontinuity, the time point belongs to the new time period unless we are at the final time. Current simulation time can be accessed in your model using variable `t`. It is possible to ignore time and write models which do not consider it, but inputs and outputs of such models will still refer to time. For example, you are required to specify at least one output time to print results from your model, even though time is inconsequent. In that case you can use an arbitrary time or sets of times.

Each `Simulation` section defines simulation conditions, from an initial time (or whatever the dependent variable represents, see Section 5.3.3 [Model types], page 25) to a final time. Initial values of the model state variables, parameter values, input variables time-course, and which outputs are to be printed at which times, can all be changed in a given `Simulation` section.

In simple cases, the general layout of the file is therefore (see also the example file in Section B.4 [perc.lsodes.in], page 78):

```
# Input file (text after # are comments)
<Global assignments and specifications>
Simulation {
  <Specifications for first simulation>
}
Simulation {
  <Specifications for second simulation>
}
# Unlimited number of simulation specifications
End. # Mandatory End keyword. Everything after this line is ignored
```

For Markov chain Monte Carlo simulations (see [MCMC() specification], page 43), the general layout of the file must include `Level` sections. `Level` sections are used to define a hierarchy of statistical dependencies (see Section 6.2.5 [Setting-up statistical models], page 55). In that case, the general layout of the file is:

```
# Input file
<Global assignments and specifications>
Level {
  # Up to 10 levels of hierarchy
  Simulation {
    <Specifications and data for first simulation>
  }
  Simulation {
    <Specifications and data for second simulation>
  }
  # Unlimited number of simulation specifications
} # end Level
End. # Mandatory keyword.
```

6.2.1 General input file syntax

The general syntax of the file is the same as that of structural model definition files (see Section 5.3.1 [General syntax], page 21) except that:

- No new variable can be created (all variables must have been defined in the model definition file). Assignments can only modify the value of already defined model variables. This implies that parameters needed to set up a statistical model must be declared in the model definition file, even if the structural model does not use them.
- Assignments to state variables or parameters can only use constant numerical values; mathematical expressions are not allowed.
- Input variables' assignments can use any input function (including the `NDoses()` and `Spikes()` functions), an `Events()` specification (see [Events() specification], page 53) or a constant numerical values.
- Output variables cannot receive assignments.

Note that vectors (see [Vectors], page 22) can be used, as in:

```
Simulation {
  y[1-10] = 1.00;
  PrintStep (y[1-5],  0, 10, 0.5);
  Print     (y[6-10], 0, 5, 10);
}
```

At the program start, all model parameters are initialized to the nominal values specified in the model description file. Next, after the input file is read, modifications given in its global section (including random sampling) are applied, then those specified at each `Level`, and finally any modifications specified by the `Simulation` sections. Computations specified in the `Initialize` section of the model definition file are the last initialization statements executed.

Structural changes to the model (*e.g.*, addition of a state, input, output or parameter) cannot be done here and must be done in the model description file. The simulation specification file is read until a mandatory `End` keyword is reached.

6.2.2 Input functions (revisited)

Input variables can be assigned all the input functions defined previously (see Section 5.3.5 [Input functions], page 27). Briefly, these are:

- `PerDose()`:

  ```
  PerDose(<magnitude>, <period>, <initial-time>, <exposure-time>);
  ```

- `PerExp()`:

  ```
  PerExp(<magnitude>, <period>, <initial-time>, <decay-constant>);
  ```

- `PerTransit()`:

  ```
  PerTransit(<magnitude>, <period>, <initial-time-in-period>,
             <decay-constant>, <number-of-input-compartments>);
  ```

- `NDoses()`:

  ```
  NDoses(<n>, <list-of-magnitudes>, <list-of-initial-times>);
  ```

- `Spikes()`:

  ```
  Spikes(<n>, <list-of-magnitudes>, <list-of-times>);
  ```

In addition, they can be assigned a `Events()` specification (see [Events() specification], page 53). Again, at a discontinuity, the time point belongs to the new time period unless we are at the final time.

6.2.3 Global specifications

In the global section you can modify, by assignment, the value of already defined state or input model variables or parameters (you cannot assign a value to an output variable). These assignments will be in effect throughout the input file, unless they are overridden later in the file. Here is an exemple of assignment (assuming that `x` and `Pi` have been properly defined in the model definition file):

```
x = 10; # set the initial value if x is a state variable
Pi = 3; # to stop worrying about little decimals...
```

In the global section, you can also give specifications relevant to all `Simulation` or `Level` sections. These specifications are not needed if you just want to perform simple simulations. They should also not appear inside `Simulation` or `Level` sections (with the notable exception of `Distrib()` specifications which can appear inside `Level` sections). They are used to call for and define the parameters of special computations (*e.g.*, the number of Monte Carlo simulations to run, which sampling distributions to use for a given parameter, the data likelihood, *etc.*) These specifications are the following:

`OutputFile()` specification

The `OutputFile()` specification allows you to specify a name for the output file of basic simulations. If this specification is not given the name `sim.out` is used if none has been supplied on the command-line or during the initial dialog. The corresponding syntax is:

```
OutputFile("<OutputFilename>");
```

where the character string <*OutputFilename*>, enclosed in double quotes, should be a valid file name for your operating system.

`Integrate()` specification

The integrator settings can be changed with the `Integrate` specification. Three integration routines can be used: `Lsodes` (which originates from the SLAC Fortran library and is originally based on Gear's routine) (Gear, 1971b; Gear, 1971a; Press et al., 1989) (see [Bibliographic References], page 65), `Cvodes` (from the Sundials library `https://computation.llnl.gov/projects/sundials`) which may be more stable than `Lsodes` in difficult cases, and `Euler` (Press et al., 1989).

The syntax for `Lsodes` is:

```
Integrate(Lsodes, <rtol>, <atol>, <method>);
```

where <*rtol*> is a floating point scalar specifying the relative error tolerance for each integration step. The floating point scalar <*atol*> specifies the absolute error tolerance parameter. Those tolerances are used for all state variables. The estimated local error for a state variable y is controlled so as to be roughly less (in magnitude) than $rtol \times |y| + atol$. Thus the local error test passes if, for each state variable, either the absolute error is less than <*atol*>, or the relative error is less than <*rtol*>. Set <*rtol*> to zero for pure absolute error control, and set <*atol*> to zero for pure relative error control. Caution: actual (global) errors may exceed these local tolerances, so choose them conservatively.

The <*method*> flag should be 0 (zero) for non-stiff differential systems and 1 or 2 for stiff systems. If you specify <*method*> 2 you should provide the Jacobian of your differential system (see the "Perc" model in the example folder); otherwise the Jacobian will be computed by numerical differentiation (which is about as fast if the Jacobian is dense). You should try flag 0 or 1 and select the fastest for equal accuracy of output, unless insight from your system leads you to choose one of them *a priori*. In our experience, a good starting point for <*atol*> and <*rtol*> is about 10^{-6}.

The syntax for `Cvodes` is:

```
Integrate(Cvodes, <rtol>, <atol>, <maxsteps>, <maxnef>, <maxcor>,
          <maxncf>, <nlscoef>);
```

<*atol*> and <*rtol*> have the same meaning as for `Lsodes`, above.

The <*maxsteps*> is an integer secifying the maximum number of internal steps taken before exiting a Cvodes call; the default value in Sundials is 500. *maxnef* is an integer specifying the maximum number of error test failures accepted; the default value in Sundials is 7. *maxcor* is an integer giving the maximum number of nonlinear iterations attempted; the default value in Sundials is 3. *maxncf* is again an integer specifying the maximum number of convergence failures accepted; the default value in Sundials is 10. Finally, *nlscoef* is a floating point scaler corresponding to the coefficient of the nonlinear convergence test; the default value in Sundials is 0.1:

Example of use:

```
Integrate(Cvodes, 1E-6, 1E-6, 500, 7, 3, 10, 0.1);
```

The syntax for `Euler` is:

```
Integrate(Euler, <time-step>);
```

where <*time-step*> is a scalar specifying the constant time increment for each integration step.

Note: if the `Integrate()` specification is not used, the default integration method is `Lsodes` with parameters 10^{-5}, 10^{-7} and 1. We recommend using `Lsodes`, since is it highly accurate and efficient. `Euler` can be used for special applications (*e.g.*, in system dynamics) where a constant time step and a simple algorithm are needed.

`MonteCarlo()` specification

Monte Carlo simulations (Hammersley and Handscomb, 1964; Manteufel, 1996) (see [Bibliographic References], page 65) randomly sample parameter values and run the model for each parameter set so generated. The statistical distribution of the model outputs can be studied for uncertainty analysis, sensitivity analysis *etc.* Such simulations require the use of two specifications, `MonteCarlo()` and `Distrib()`, which must appear in the global section of the file, before the `Simulation` sections. They are ignored if they appear inside a `Simulation` section.

The `MonteCarlo` specification gives general information required for the runs: the output file name, the number of runs to perform, and a starting seed for the random number generator. Its syntax is:

```
MonteCarlo("<OutputFilename>", <nRuns>, <RandomSeed>);
```

The character string <*OutputFilename*>, enclosed in double quotes, should be a valid filename for your operating system. If a null-string `""` is given, the default name `simmc.out` will be used. The number of runs <*nRuns*> should be an integer, and is only limited by either your storage space for the output file or the largest (long) integer available on your machine. When using the GNU Scientific Library, the seed <*RandomSeed*> of the pseudo-random number generator can be any positive integer (including zero). When using the native code, it can be any positive real number (seeds between 1.0 and 2147483646.0 are used as is, the others are rescaled silently within those bounds). Here is an example of use:

```
MonteCarlo("percsimmc.out", 50000, 9386);
```

The parameters' sampling distributions are specified by a list of `Distrib()` specifications, as explained in the following (see [Distrib() specification], page 48). The format of the output file of Monte Carlo simulations is discussed later (see Section 6.3 [Analyzing simulation output], page 58).

Note on parallelized code: If your model has been preprocessed and compiled by `makemcsimp` (see Section 5.2 [Using makemcsim], page 20) to generate a parallelization-enabled executable, the number of Monte Carlo simulations requested will be split evenlly between the recruited processors and computed in parallel. The number of processors to use should specified on the calling command-line with the `-np` option (see Section 6.1 [Using the compiled program], page 37).

`MCMC()` specification

Markov chain Monte Carlo (*MCMC*) can be defined as stochastic simulations following a Markov chain in a given parameter space. In MCMC simulations, the random choice of a new parameter value is influenced by the current value. They can be used to obtain a sample of parameter values from complex distribution functions, eventually intractable analytically. Such complex distribution functions are typically encountered during Bayesian data analysis, under the guise of posterior distributions of a model's parameters. The reader wishing to use the MCMC capabilities of *GNU MCSim* is referred to the published literature (for example, Bernardo and Smith, 1994; Gelman, 1992; Gelman et al., 1995; Gelman et al., 1996; Gilks et al., 1996; Smith, 1991; Smith and Roberts, 1993) (see [Bibliographic References], page 65).

MCMC simulation chains (which in *GNU MCSim* start from a sample from the specified prior) need to reach "*equilibrium*". Checking that equilibrium is obtained is best achieved, in our opinion, by running multiple independent chains (*cf.* Gelman and Rubin, 1992, and other relevant statistical literature). *GNU MCSim* does not deal (yet) with convergence issues.

The Bayesian analysis of data with *GNU MCSim* requires you to setup:

- a *structural model* (see Chapter 5 [Writing and Compiling Structural Models], page 19),
- a *statistical model* (see Section 6.2.5 [Setting-up statistical models], page 55),
- *prior distributions* for the model parameters you want to sample and "*data likelihoods*" (defining the probability of some observed realizations of the modeled process, conditionally to the model) (see [Distrib() specification], page 48).

Setting-up a statistical model requires `Level` sections and `Data()` specifications. Assigning priors and likelihoods is achieved through the `Distrib()` statements (or its equivalents `Density()` and `Likelihood()`). Please refer to the corresponding sections of this manual, if you are not familiar with them. The `MCMC()` statement, gives general directives for MCMC simulations and has the following syntax:

```
MCMC("<OutputFilename>", "<RestartFilename>", "<DataFilename>",
     <nRuns>, <simTypeFlag>, <printFrequency>, <itersToPrint>,
     <RandomSeed>);
```

The character strings <*OutputFilename*>, <*RestartFilename*>, and <*DataFilename*>, enclosed in double quotes, should be valid file names for your operating system. If a null-string `""` is given instead of the output file name, the default name `MCMC.default.out` will be used.

If a restart file name is given, the first simulations will be read from that file (which must be a text file). This allows you to continue a simulated Markov chain where you left it, since an MCMC output file can be used as a restart file with no change. Note that the first line of the file (which typically contains column headers) is skipped. Also, the number of lines in the file must be less than or equal to <*nRuns*>. The first column of the file should be integers, and the following columns (tab- or space-separated) should give the various parameters, in the same order as specified in the list of `Distrib()` specifications in the input file.

If a data file name is given, the observed (data) values for the simulated outputs will be read from that file (in ASCII format); otherwise, `Data()` specifications (see [Data() specification], page 58) should be provided. We recommend that you use `Data()` specifications rather that the data file, which is much more error prone. The first line of the data file is skipped and can be used for comments. The total number of data points should equal the total number of outputs requested. The data values should be given on the second and following lines, separated by white spaces or tabs. A data value of `"-1"` will be treated as `"missing data"` and ignored in likelihood calculations. The convention `"-1"` can be changed by changing INPUT_MISSING_VALUE in the header file `mc.h` and recompiling the entire library.

The integer <*nRuns*> gives the total number of runs to be performed (*i.e.*, the number of posterior samples to draw), including the runs eventually read in the restart file.

Note on parallelized code: If your model has been preprocessed and compiled by `makemcsimp` (see Section 5.2 [Using makemcsim], page 20) to generate a parallelization-enabled executable, *GNU MCSim* will run, in parallel, as many simulated Markov chains (of length <*nRuns*>) as there are available processors. The number of processors to use (hence, of chains to run) should specified on the calling command-line with the `-np` option (see Section 6.1 [Using the compiled program], page 37). You can assess the chains' convergence in real time with the Gelman-Rubin's MCMC convergence diagnostic with the run-time command line option `-c` (see Section 6.1 [Using the compiled program], page 37)

The next field, <*simTypeFlag*> should be between 0 and 4 (included):

- It should be set at zero to start a chain of MCMC simulations. In that case, parameters are updated by Metropolis steps, one at a time.
- If <*simTypeFlag*> = 1, a restart file must also be specified. The output file will contain codes for the level sequence, simulation numbers, printing times, data values and the corresponding model predictions, computed using the last parameter vector of the restart file. This is useful to quickly check the model fit to the data.
- If <*simTypeFlag*> is equal to 2, a restart file must also be specified and that entire file is used to compute the parameters' covariance matrix. All parameters are then updated at once using a multivariate normal kernel as proposal distribution of the Metropolis steps. This may result in large improvement in speed. However, we recommend that this option be used only when convergence is approximately obtained (therefore, you should run MCMC simulations with <*simTypeFlag*> set to 0 first, up to approximate convergence, and then restart the chain with the flag at 2).
- With <*simTypeFlag*> equal to 3 or 4, component by component simulated tempering is performed (Geyer and Thompson, 1995) (see [Bibliographic References], page 65). With option 3, the whole posterior is tempered (as in Geyer and Thompson) (see [Bibliographic References], page 65). With option 4, only the likelihood is tempered (thermodynamic integration, as in Calderhead and Girolami (2009). In those cases, a grid of inverse temperatures (perks) is first determined automatically for you (diagnos-

tic are printed out to the file `...out.perks`), unless you define your own perks scale with the `InvTemperature()` specification:

```
InvTemperature(<nValues>, <value 1>, <...>, <value n>);
```

The first number of the specification gives the number of perks to use and is followed by a list of them. Those should be numbers in the interval [0,infinity[. Values above 1 lead in fact to simulated annealing (sharpening of the posterior distribution), suitable for optimization (see Amzal et al. 2006) (see [Bibliographic References], page 65). For simulated tempering, including thermodynamic integration, you should have perks lower or equal to 1. You usually want to include the value 1 (since that perk corresponds to your target distribution). At perk zero, the posterior distribution is uniform for all parameters in the case of posterior tempering, or equal to the prior in the case of likelihood tempering. Each time the simulated Markov chain reaches perk zero is a regeneration time (Geyer and Thompson, 1995). Samples obtained at perk 1 between regeneration times are guaranteed to be from the posterior distribution, so that only one chains needs to be run and convergence need not to be checked (a significant advantage of simulated tempering). Simulated tempering is also adapted to problems with multiple maxima of the posterior distribution, in which standard samplers or Hamiltonian MCMC usually get stuck in a local mode. See Section 6.3 [Analyzing simulation output], page 58 for the output format of tempered MCMC simulations.

- Finally with <*simTypeFlag*> set to 5, GNU MCSim does stochastic optimization (an MCMC in which only, but all, jumps leading posterior density improvement are accepted).

The integer <*printFrequency*> should be set to 1 if you want an output at each iteration, to 2 if you want an output at every other iteration etc. The parameter <*itersToPrint*> is the number of final iterations for which output is required (*e.g.*, 1000 will request output for the last 1000 iterations; to print all iterations just set this parameter to the value of <*nRuns*>). Note that if no restart file is used, the first iteration is always printed, regardless of the value of <*itersToPrint*>. Finally, when using the GNU Scientific Library, the seed <*RandomSeed*> of the pseudo-random number generator can be any positive integer (including zero). When using the native code, it can be any positive real number (seeds between 1.0 and 2147483646.0 are used as is, the others are rescaled silently within those bounds).

In the case of component by component jumps (<*simTypeFlag*> set to 0), tempered or stochastic optimization (<*simTypeFlag*> set to 3 or higher), the jump kernel is saved with the same name as the output file, with a *.kernel* extension. If the simulations are restarted in a continuation mode and if a kernel file with the same name as the restart file (with an added *.kernel* extension) is present, the jump kernel is restored.

Finally, the format of the output file of MCMC simulations is quite similar to that of straight Monte Carlo simulations and will discussed in a later section (see Section 6.3 [Analyzing simulation output], page 58).

`SetPoints()` specification

To impose a series of set points (*i.e.*, already tabulated values for the parameters), the global section can include a `SetPoints()` specification. It allows you to perform additional simulations with previously Monte Carlo sampled parameter values, eventually filtered. You can also generate parameters values in a systematic fashion, over a grid for example, with another program, and use them as input to *GNU MCSim*. Importance sampling, Latin hypercube sampling, grid sampling, can be accommodated in this way.

This command specifies an output filename, the name of a text file containing the chosen parameter values, the number of simulations to perform and a list of model parameters to read in. Parameters can mix scalar and vector notations. It has the following syntax:

```
SetPoints("<OutputFilename>", "<SetPointsFilename>", <nRuns>,
          <parameter identifier>, <parameter identifier>, ...);
```

If a null string is given for the output filename, the set points output will be written to the same default output file used for Monte Carlo analyses, `simmc.out`.

The *SetPointsFilename* is required and must refer to an existing file containing the parameter values to use. The first line of the set points file is skipped and can contain column headers, for example. Each of the other lines should contain an integer (*e.g.*, the line number) followed by values of the various parameters in the order indicated in the `SetPoints()` specification. If extra fields are at the end of each line they are skipped. The first integer field is needed but not used (this allows you to directly use Monte Carlo output files for additional `SetPoints` simulations).

The variable <*nRuns*> should be less or equal to the number of lines (minus one) in the set points file. If a zero is given, all lines of the file are read. Finally, a comma-separated list of the parameters or vectors to be read in the *SetPointsFilename* is given. The format of the output file of set points simulations is discussed below (see Section 6.3 [Analyzing simulation output], page 58).

Following the `SetPoints()` specification, `Distrib()` statements can be given for parameters not already in the list (see [Distrib() specification], page 48). These parameters will be sampled accordingly before to performing each simulation. The shape parameters of the distribution specifications can reference other parameters, including those of the list.

Note on parallelized code: If your model has been preprocessed and compiled by `makemcsimp` (see Section 5.2 [Using makemcsim], page 20) to generate a parallelization-enabled executable, the number of `SetPoints` simulations requested will be split evenlly between the recruited processors and computed in parallel. The number of processors to use should specified on the calling command-line with the `-np` option (see Section 6.1 [Using the compiled program], page 37).

`OptimalDesign()` specification

The "*OptimalDesign*" procedure optimizes the number and location of observation times for experimental conditions you specify, in order to minimize the variance of a parameter or an output you designate. It requires a structural model (see Chapter 5 [Writing and Compiling Structural Models], page 19), a statistical model in the form of a `likelihood()` function (see Section 6.2.5 [Setting-up statistical models], page 55), and a random set of parameter vectors sampled from a prior distribution (using Monte Carlo or MCMC simulations) (for example and details, see Bois et al., 1999) (see [Bibliographic References], page 65). The statistical model used should be quite simple and cannot not use `Level` sections (and hence cannot be hierarchical).

The *OptimalDesign* command has the following syntax:

```
OptimalDesign("<OutputFilename>", "<ParameterSampleFilename>",
              <nSamples>, <RandomSeed>, <Style>,
              <parameter identifier>, <parameter identifier>, ...);
```

The character strings <*OutputFilename*>, and <*ParameterSampleFilename*>, enclosed in double quotes, should be valid file names for your operating system. If a null-string `""` is given instead of the output file name, the default name `simopt.default.out` will be used.

A parameter sample file name must be given (that file must be a text file). The first line of the file (which typically contains column headers) is skipped. The number of lines in the file must be less than or equal to <*nSamples*>. The first column of the file should be integers (typically row numbers), and the following columns (tab- or space-separated) should be values of the various parameters in the order indicated in the list at the end of the `OptimalDesign()` specification. If extra fields are at the end of each line they are skipped. The first integer field is needed but not used (this allows you to directly use Monte Carlo output files for `OptimalDesign` simulations).

The integer <*nSamples*> indicates the number of lines to read from the <*ParameterSampleFilename*> file. When using the GNU Scientific Library, the seed <*RandomSeed*> of the pseudo-random number generator can be any positive integer (including zero). When using the native code, it can be any positive real number (seeds between 1.0 and 2147483646.0 are used as is, the others are rescaled silently within those bounds). The directive *Style* should be either the keyword `Forward` or the keyword `Backward`. Forward optimization will start from no new data and will add, sequentially, optimal observation times. Backward optimization starts with the full set of observation times you propose and delete the least informative ones, sequentially. We recommand that you try both options. Finally, a comma-separated list of the parameters to be read in the *ParameterSamplFilename* should be given.

The input file must then contain two sets of `Simulation` definitions. You should look at the sample optimal design files provided in `mcsim/examples`.

The first set specifies all experimental conditions and the set of observation times to optimize, for one or several output variables given in `Print` statements. The output times you specify for each output variable define an array of observation time values that the optimization algorithm will rank by order of the estimated variance reduction they permit for variables or parameters you will specify in the second set of `Simulation` definitions. Data will be simulated for each of the required output. There must be one Data statement per output specified (the data values are arbitrary). An error model must be specified for those data, using a `Likelihood` statement (see [Distrib() specification], page 48).

The second set of `Simulation` specifies optimization target parameters or outputs. The algorithm will select time-points (in the first section's `Simulation` specifications) that minimize the estimation variance of those parameters or outputs. When a parameter is targeted no inputs are needed. If you optimize for an output variable variance (*i.e.*, for the variance of a model prediction), the experimental conditions can be very different from those of the experiment whose conditions you optimize. The link is afforded solely by the parameters (in the first set you are trying to determine the conditions that will optimally identify the parameter values conditioning the predictions – or trivially, the parameters – of the second set)

The format of the output file of design optimization simulations is quite specific. The first column is an iteration number. At each iteration one observation point is added (`Forward` mode) or removed (`Backward` mode). Each step is therefore conditioned by the selection of an observation time-point made by the previous step. The following columns give, for each observation time point you specify, the average variance of the target outputs or parameters achieved if this point is added (`Forward` mode) or removed (`Backward` mode). Next the chosen time point at this step is given (the one minimizing average variance), followed by the variance it leads to (in expectation) and the corresponding standard deviation. The last column `"Utility"` is zero, unless you uncomment the function `Compute_utility` and modify its code in `optdesign.c` to compute a utility of your own.

`Distrib()` specification

The specification of distributions for simple Monte Carlo simulations is quite straighforward. MCMC simulations require the definition of a full statistical model and the use of distributions is somewhat more complex in that case, but the use of `Distrib()` is basically the same.

In the context of `MonteCarlo()` or `SetPoints()` simulations (see [MonteCarlo() specification], page 42, and [SetPoints() specification], page 45), one (and only one) `Distrib()` specification must be included for each model parameter to randomly sample. State, input or output variables cannot be randomly sampled by `Distrib()` in this context. A simulation specification file can include any number of `Distrib()` commands at the global level.

`Distrib()` specifies the name of the parameter to sample, and its sampling distribution. Its syntax is:

```
Distrib(<parameter identifier>, <distribution-name>,
        [<shape parameters>]);
```

The <*parameter identifier*> gives the name of the parameter to sample. The <*distribution-name*> and the corresponding <*shape parameters*> indicate the sampling distribution to use (Bernardo and Smith, 1994; Gelman et al., 1995) (see [Bibliographic References], page 65). They are specified as follow:

- `Beta`, takes at least two strictly positive real shape parameters: A and B. By default the Beta distribution is defined over the interval [0;1]. If a range is given for the beta distribution, the [0;1] interval is mapped onto the specified range.
- `Binomial`, needs two strictly positive numbers: the probability p (a real in the interval [0;1]), and the sample size N, an integer. If N is not given as an integer it will be rounded down during the computations.
- `Cauchy`, takes one strictly positive real number as parameter: its scale s.
- `Chi2`, takes one strictly positive real number as parameter: n. This distribution is the same as Gamma($n/2$, $1/2$).
- `Exponential`, uses one strictly positive real number: the inverse-scale b. The density of this distribution is equal to be^{-bx}.
- `Gamma`, uses two strictly positive real parameter: the shape and the inverse scale.
- `HalfCauchy`, takes one strictly positive real number as parameter: the scale s. The mode is at zero, on the lower boundary. The random variates returned are strictly positive.

- `HalfNormal`, takes one real number as parameter: the standard deviation, strictly positive. The mode is at zero, on the lower boundary. The random variates returned are strictly positive.
- `InvGamma` (inverse gamma distribution), needs two strictly positive real parameters: the shape and the scale.
- `LogNormal`, takes two reals numbers as parameters: the geometric mean (exponential of the mean in log-space) and the geometric standard deviation (exponential, strictly superior to 1, of the standard deviation in log-space).
- `LogNormal_v`, is the lognormal distribution with the variance (in log space!) instead of the standard deviation as second parameter. You can use it to specify a hierarchical model with a conjugate prior on the variance (see Section 6.2.5 [Setting-up statistical models], page 55).
- `LogUniform`, with two shape parameters: the minimum and the maximum of the sampling range (real numbers) in natural space.
- `Normal`, takes two reals numbers as parameters: the mean and the standard deviation, the latter being strictly positive.
- `NegativeBinomial`, needs two positive or null numbers: the number of failures r until trials are stopped and probability of success (p, a real in the interval [0;1]) for each trial. The probability mass function of this distribution for a k of number of successes is equal to:

  $$C(k+r-1,k)p^r(1-p)^k$$

  where C() is the binomial coefficient.
- `Normal_cv`, is the normal distribution with the coefficient of variation instead of the standard deviation as second parameter.
- `Normal_v`, is also the normal distribution with the variance instead of the standard deviation as second parameter. You can use it to specify a hierarchical model with a conjugate prior on the variance (see Section 6.2.5 [Setting-up statistical models], page 55).
- `Piecewise`, uses four reals as parameters: the *minimum*, A, B, and the *maximum*. The distribution has the form of a truncated triangle, with a plateau between A and B. If $A = B$, the distribution is the triangular distribution.
- `Poisson`, needs a strictly positive real: the rate A.
- `StudentT`, requires three parameters: its number of degrees of freedom (an integer), its mean, and its standard deviation.
- `TruncInvGamma` (truncated inverse gamma distribution), needs four strictly positive real parameters: the shape, the scale, the minimum and the maximum. To use it you need to have the GNU Scientific Library (`gsl`) to be installed.
- `TruncLogNormal` (truncated lognormal distribution), uses four real numbers: the geometric mean and geometric standard deviation (strictly superior to 1), the minimum and the maximum in natural space. For example:

  ```
  Distrib(Var, TruncLogNormal, 1, 2.718, 0.01, 10);
  ```

  samples '`Var`' such that $\ln(Var)$ is a standardized normal variate of mean ln(1) and standard deviation ln(2.718) — while '`Var`' is truncated to fall between 0.01 to 10.

- `TruncLogNormal_v`, is like the truncated lognormal, except that it takes the variance (in log space!) instead of the standard deviation as second parameter. You can use it to specify a hierarchical model with a conjugate prior on the variance (see Section 6.2.5 [Setting-up statistical models], page 55).
- `TruncNormal` (truncated normal distribution), takes four real parameters: the mean, the standard deviation (strictly positive), the minimum and the maximum.
- `TruncNormal_cv`, is like the truncated normal distribution with the coefficient of variation instead of the standard deviation as second parameter.
- `TruncNormal_v`, is like the truncated normal distribution with the variance instead of the standard deviation as second parameter.
- `Uniform`, with two shape parameters: the minimum and the maximum of the sampling range (real numbers).
- `UserSpecifiedLL` can be used to specify a data likelihood. It allows the user to compute the log-likelihood of the data inside the model definition file (in the `Dynamics` or `CalcOutputs` sections). It uses only one parameters: the calculated log-likelihood. For example:

  ```
  Likelihood(LL, UserSpecifiedLL, Prediction(LL));
  ...
  Print (LL, 1);
  Data  (LL, 1);
  ```

  In this case, the `Data()` statements are not actually used by the model. They must still be provided, with arbitrary data values (except that you should not use the missing data code, -1 by default). The actual data values must be passed as parameter(s) to the model.

The shape parameters of the above distributions, if they are defined, can symbolically reference other model parameters, even if distributions for these have already been defined. For example:

```
Distrib(A, Normal, 0, 1);
Distrib(B, Normal, A, 2);
```

In the context of MCMC sampling, *GNU MCSim* provides extensions of the above `Distrib()` specification syntax.

First, when `Distrib()` is used to specify the prior or conditional distribution of a model parameter (in a Bayesian framework), that parameter can also appear as a shape parameter, if a distribution has already been specified for the parameter at an upper `Level` of the file.

For example:

```
Level {                        # upper level
  Distrib(A, Normal,   0, 1);
  Distrib(B, InvGamma, 2, 2);
  Level {                      # sub-level
    Distrib(A, Normal_v, A, B);
    ...
  }                            # end sub-level
}                              # end upper level
```

In that case, the parameter A, used for shape specification (as the mean of a Normal distribution) in the sub-level, refers to the "*parent*" A parameter, for which a standard Normal distribution is defined at the upper `Level`. The *sampled* values of the parent parameters A and B will be used as mean and variance for their "*child*" parameter, A, when it will be its turn to be randomly sampled. This forms the basis of the specification of multilevel (hierarchical) models (see Section 6.2.5 [Setting-up statistical models], page 55).

Note that the keyword `Density()` can be used instead of `Distrib()` to make more understandable MCMC input files. `Density()` and `Distrib()` are equivalent and have the same syntax. We recommend its use.

Next, in MCMC simulations, you usually assign a probability distribution (*i.e.*, a likelihood) to the data you analyze. Typically, your model's state and/or output variables will be used to predict some aspect of the observed data distributions (mean, variance, *etc.*). *GNU MCSim* gives you the possibility to specify a distribution for your data, using model parameters, input, state, or output model variables, or even other data, to define the distribution shape. This is achieved through the use of the `Data()` and `Prediction()` "*qualifiers*". In that case, the keyword `Likelihood()` should be used instead of `Distrib()` (the two keywords are equivalent and have the same syntax, but we recommend that you use `Likelihood()` to make your MCMC input files more understandable and make it clear that the data are *not* sampled, but only assigned a likelihood).

`Data()` can be used at the first position of a `Distrib()` statement, or as a distribution shape parameter. It uses the following syntax:

```
Data(<identifier>)
```

where <*identifier*> corresponds to a valid input, state or output model variable for which data are available. Model parameters cannot be used (but you can assign a simple parameter value to an output variable in your model definition file and use that output here). The actual data values need to be given later in the simulation input file through `Data()` specifications (which, in addition to a variable identifier, give a list of numerical data values, see [Data() specification], page 58) or in a separate datafile (see [MCMC() specification], page 43).

Working hand in hand with `Data()`, and using the same syntax, the `Prediction()` qualifier can be used to designate actual model inputs, states and outputs for any shape parameter of a specified distribution (therefore `Prediction()` must appear after the distribution name). The actual predicted values, matching exactly the corresponding data, need to be given later in the simulation input file through `Print()` or `PrintStep()` specifications [see [Print() specification], page 54 and [PrintStep() specification], page 54).

Here are some example of use of `Data()` and `Prediction()` in the extended syntax of a `Likelihood()` specification:

```
Likelihood (Data(y), Normal,   Prediction(y), 0.01);
...
Data (y, 0.1, 2,  5,  3,  9.2);
Print(y, 10,  20, 40, 60, 100);

Likelihood (Data(y), Normal,   Prediction(y), Prediction(sigma));
...
Data (y, 1.01, 1.20, 0.97, 0.80, 1.02);
```

```
PrintStep(y,     10, 50, 10);
PrintStep(sigma, 10, 50, 10);

Likelihood (Data(R), Binomial, Prediction(P), Data(N));
...
Data (R, 0,  2,  5, 5,  8, 9, 10, 10);
Data (N, 10, 10, 9, 10, 9, 9, 11, 10);
Print(P, 10, 20, 30,40, 50,60,70, 80);
```

(all these statements could not appear as such in an input file, they would need to be embedded in `Level` and `Simulation` sections.)

`SimType()` specification

This specification is now obsolete and should not be used. It is left for compatibility with old input files. It specifies the type of analysis to perform. Syntax:

```
SimType(<keyword>);
```

The following keywords can be used: `DefaultSim` (the list of specified simulations is simulated), `MonteCarlo`, `MCMC` (previously `Gibbs`, which is now deprecated), `SetPoints`. If `MonteCarlo`, `MCMC`, or `SetPoints` analyses are requested, additional specifications are needed (see below).

6.2.4 Specifying basic conditions to simulate

Any simulation file must define at least one `Simulation` section. `Simulation` sections include particular specifications, which are presented in the following.

`Simulation` sections

After global specifications, if any, `Simulation` sections must be included in the input file. Expectedly, these sections start with the keyword `Simulation` and are enclosed in curly braces.

A `Simulation` section can make assignments to any state variable, input variable or parameter defined in the global section of the model description file. Output variables cannot receive assignments in simulation input files.

State variables and parameters can only take constant numerical values (see Section 6.2.1 [General input file syntax], page 39). For state variables, this sets the initial value only. So, for example, in a `Simulation` section the parameter `Speed`, if properly defined, can be set using:

```
Speed = 83.2;
```

This overrides any previously assigned values, even if randomly sampled, for the specified parameter.

Inputs can be redefined with input functions (see Section 6.2.2 [Input functions revisited], page 40) or constant numerical values. Input functions can reference other variables (eventually randomly sampled), as in:

```
Q_in = PerExp(InMag, 60, 0, RateConst);
```

The maximum number of `Simulation` sections allowed in an input file is 200. This can be changed by changing MAX_INSTANCES and MAX_EXPERIMENTS in the header file `sim.h` and recompiling the program (this requires re-installation).

Within a `Simulation` section, several additional specifications can be used:

- `StartTime()`,
- `Print()` (or its synonym, `Prediction()`)
- `PrintStep()`,
- `Data()`.

The `Data()` specification is used only when a statistical model is set up and will be covered in the corresponding section of this manual (Section 6.2.5 [Setting-up statistical models], page 55).

`Events()` specification for state discontinuities

You can impose state variables discontinuities at a given set of times with the `Events` specification. `Events` has a syntax similar to the `NDoses` input function (see Section 5.3.5 [Input functions], page 27). It is in fact a special type of input function that resets a state variable, hence the need to assign it to a (dummy) input variable. Its syntax is:

```
<input variable> = Events(<state variable>, <n>, <list-of-times>,
                          <list-of-operation>, <list-of-scalars>);
```

The first argument, <*state variable*>, is the state whose value you want to reset at given times. The integer <*n*> is the number of reset times you want to specify; <*list-of-times*> is a comma separated list of those reset times; <*list-of-operations*> is a comma separated list of operations that will affect the given state variable at the specified times (see next paragraph), and <*list-of-scalar*> is the list of floating point values used for by the operations specified, at the corresponding times.

The three keywords operations are `Add`, `Multiply` and `Replace`. `Add` adds the corresponding scalar to the target state variable at the specified time; `Multiply` multiplies the state variable by the specified scalar; `Replace` simply replaces the value of the state variable by the given scalar.

The assigned input value takes the value 1 at the specified times and is zero otherwise. If you don't have a use for such an input, simply define a dummy input variable. For example:

In the model definition file define:

```
Inputs = {events_v1, events_v2};
```

and in the simulation specification file you can request, within a `Simulation` specification:

```
events_v1 = Events (v1, 2, 1,   9,
                           Add, Add,
                           1,   4);

events_v2 = Events (v2, 2, 1,        5,
                           Multiply, Replace,
                           2,        6);
```

At time 1, a value of 1 will be added to the state variable *v1* and *v2* will be multiplied by 2; At time 5 *v2* will be reset to 6, and at time 9 the value 4 will be added to *v1*.

See (and run) the example model and input file provided in the example/events folder.

`StartTime()` specification

The origin of time for a simulation, if it needs to be defined, can be set with the `StartTime()` specification, whose syntax is:

```
StartTime(<initial-time>);
```

It just shifts the time scale. If this specification is not given, a value of zero is used by default. A parameter can be used as initital time value, so that initial time can be sampled in error-in-variable models, for example:

```
T0 = 20;
StartTime(T0);
```

The final time is automatically computed to match the largest output time specified in the `Print()` or `PrintStep()` statements. Output times cannot be inferior to the initial time.

`Print()` specification

The value of any model variable or parameter can be requested for output with `Print()` or `Prediction()` specifications. Their arguments are a comma-separated list of variable names (at least one and up to MAX_PRINT_VARS, which is set to 10 by default), and a comma-separated list of increasing times at which to output their values:

```
Print(<identifier1>, <identifier2>, ..., <time1>, <time2>, ...);
Prediction(<identifier3>, <identifier4>, ..., <time1>, <time2>, ...);
```

where <*identifier1*>, <*identifier2*> etc. correspond to valid input, state or output model variables, or parameter.

The same output times are used for all the variables specified. The size of the time list is only limited by the available memory at run time. The limit of 10 variables names can be increased by changing MAX_PRINT_VARS in the header file `sim.h` and re-installing the whole software. The number of `Print()` statements you can used in a given `Simulation` section is only limited by the available memory at run time. The same variable or parameter can appear in more than one `Print()` specification in a given `Simulation` section.

`PrintStep()` specification

The value of any model variable or parameter can be also output with `PrintStep()` specifications. They allow dense printing, suitable for smooth plots, for example. Their arguments are: a comma-separated list of variable names (at least one and up to MAX_PRINT_VARS, which is set to 10 by default), the first output time, the last one, and a time increment:

```
PrintStep(<identifier1>, <identifier2>, ..., <start-time>, <end-time>,
          <time-step>);
```

The final time has to be superior to the initial time and the time step has to be less than the time span between end and start. If the time step is not an exact divider of the time span the last printing step is shorter and the last output time is still the end-time specified. The number of outputs produced is only limited by the memory available at run time. You can use several `PrintStep()` specification, and the same variable or parameter can appear in more than one `PrintStep()`, in a given `Simulation` section.

6.2.5 Setting-up statistical models

With *GNU MCSim*, you must define a statistical model to use the `MCMC()` specification. `MCMC` simulations will give you a sample from the joint posterior distribution of the parameters that you designate as randomly sampled through `Distrib()` specifications. You do not need to specify explicitly that joint posterior distribution (in fact, in most case, this is impossible). The posterior distribution is implicitly defined by a statistical model, that is simply a set of conditional relationship between the parameters and some data.

GNU MCSim handles multilevel (hierarchical) random effects and mixed effects statistical models in a Bayesian framework. These models need to be defined in the simulation specification file, rather than in the structural model definition file. Yet, due to compilation constraints, if you need special parameters for your statistical model (*e.g.*, variances) you have to declare them in the structural model file, even if they are not used by the structural model itself.

So, how do we go about specifying a statistical model with *GNU MCSim*? Take for example the following simple linear regression model:

$$y_i = N(\mu_i, \sigma^2) \tag{1}$$

$$\mu_i = \alpha + \beta(x_i - \overline{x}) \tag{2}$$

where the observed (x, y) pairs are $(1, 1)$, $(2, 3)$, $(3, 3)$, $(4, 3)$ and $(5, 5)$. Assume that the parameters α and β are given $N(0, 10000)$ priors, and that $1/\sigma^2$ is given a $Gamma(10^{-2}, 10^{-2})$ prior. We want the posterior distributions of α, β, and σ^2.

The first thing to do is to define a structural (or link) model to compute y as a function of x. Here is such a model (quite similar to the one distributed with *GNU MCSim* source code (see Section B.1 [linear.model], page 71):

```
# ----------------------------------------------
# Model definition file for a linear model
# ----------------------------------------------
Outputs = {y};

# Structural model parameters
Alpha = 0;
Beta = 0;
x_bar = 0;

# Statistical parameter
Sigma2 = 1;

CalcOutputs { y = Alpha + Beta * (t - x_bar); }
# ----------------------------------------------
```

The parameters' default values are arbitrary, and could be anything reasonable. They will be changed or sampled through the input file. Note thatσ^2 is not used in the model equations, but still needs to be defined here in order to be part of the statistical model. On the other hand, μ is not defined, since we do not really need it. Finally x is replaced by the time, `t`, for convenience. An alternative would be to define an input ‘`x`’ and use it instead of `t`.

We now need to write an input file specifying the distribution of y (*i.e.*, the likelihood), and the prior distributions of the various parameters. Technically, *GNU MCSim* uses Metropolis sampling and you do not need to worry about issues of conjugacy or log-concavity of your prior or posterior distributions. Here is what a simulation file with a statistical model looks like:

```
# ---------------------------------------------------------------
# Simulation input file for a linear regression
# ---------------------------------------------------------------
MCMC ("linear.MCMC.out", "", "", 50000, 0, 5, 40000, 63453.1961);
Level {
  Distrib(Alpha,  Normal_v, 0, 10000);
  Distrib(Beta,   Normal_v, 0, 10000);
  Distrib(Sigma2, InvGamma, 0.01, 0.01);
  Likelihood(Data(y), Normal_v, Prediction(y), Sigma2);
  Simulation {
    x_bar = 3.0;
    PrintStep (y, 1, 5, 1);
    Data (y, 1, 3, 3, 3, 5);
  }
} # end Level
End.
# ---------------------------------------------------------------
```

The file begins with `MCMC()` (see [MCMC() specification], page 43). The keyword `Level` comes next. `Level` is used to specify hierarchical dependences between model parameters. There should be at least one `Level` in every MCMC input file, even for a non-hierarchical model like the one above. See below for further discussion of the `Level` keyword. You can also look at the MCMC input files provided as examples with *GNU MCSim* source code. The `Distrib()` statements define the parameter priors. `Normal_v` specifications are used since we use variances instead of standard deviations. The inverse-Gamma distribution is used for the variance component, since the precision is supposed to be Gamma-distributed. The likelihood is the distribution of the data, given the model: it is specified by a `Likelihood()` specification, valid for every y data point. Again, note that the μ variable is not used. Instead, the `Prediction(y)` specification designates the linear model output. The distributions and likelihoods specified are in effect for every sub-level or every `Simulation` section included in the current `Level`.

The "simulations" to perform, and the corresponding data values, are specified by a `Simulation` section. Only one `Simulation` section is needed here, but several could be specified. In this section, the value of $\overline{x}$ is provided. The different values of x (time in our formulation of the model) can be specified via `PrintStep()` (see [PrintStep() specification], page 54), since they are equally spaced. More generally, `Print()` can also be used (see [Print() specification], page 54). The data values are given in a `Data()` statement (see below).

The following paragraphs deal with `Level` sections and `Data()` specifications.

`Level` sections

Markov chain Monte Carlo simulations require the definition of a statistical model structured with "*levels*". Think for example of the definition of a prior distribution as a top level in a hierarchy, with the data likelihood being at the lowest level. The hierarchy levels are defined in *GNU MCSim* with the help of `Level` sections. At least one `Level` section must be defined in the simulation input file (you cannot use `Level` in a structural model definition file). A `Level` section starts with the corresponding keyword and is enclosed in curly braces ('{}'). It can include any number of sub-levels or `Simulations` sections. `Simulations` (where the data are specified) form the lowest level of the hierarchy (see [Simulation sections], page 52). In terms of structure, `Simulation` sections behave like `Level` sections (in particular with regard to "*cloning*" of random variables, see below) except that they cannot include further levels. There must be one and only one top `Level` and at most 10 nested sub-levels in the hierarchy. No parameter assignement or distribution specification should occur outside a `Level`, otherwise it will be ignored (which is probably not what you want). This limit of 10 can be increased (up to 255) by changing MAX_LEVELS in the header file `sim.h` and re-installing *GNU MCSim*.

A `Level` can specify or change the sampling distribution of any model parameter properly defined in the global section of the structural model description file. These distribution specifications apply to all sub-levels of the `Level` where they take place. For example:

```
MCMC("samp.out", "", "", 1, 1, 1, 1, 1); # we are in an MCMC context
Level { # this is the top level
  Distrib(A, Uniform, 0, 1);
  Likelihood(Data(y), Normal, Prediction(y), 1);
  Level { # sub-level 1
    Distrib(A, Normal, A, 1);
    Simulation { ... } # simulation 1
    Simulation { ... } # simulation 2
  } # End sub-level 1
} # End top, end file
End.
```

A `Level` can also make simple assignments to any model parameter (see Section 6.2.1 [General input file syntax], page 39). So, for example, in an simulation, the parameter A could be modified with:

```
A = 2.0;
```

This overrides any previously assigned values for the specified parameter, even if randomly sampled, and applies to the sub-levels of the `Level` where it take place.

An important concept to grasp here is that of parameter "*cloning*". Cloning automatically creates, using templates, as many new parameters as you need in your multilevel model. One of the characteristic feature of multilevel models is the same parameters appear at several levels. For example, in a random effect model, a parameter (*e.g.*, size) will be assumed to be randomly distributed in a population of individuals. If you have 100 individuals in your database, your model will have to deal with 100 individual size parameters and an average size. To spare you the tedium of defining the same distribution for many parameters, *GNU MCSim* creates an appropriate number of parameters for your model on the basis of its level structure. Assume that you have specified a distribution for a parameter A at a given level (that we label *L1* for clarity). *GNU MCSim* will automatically create new parameters ("*clones*") with the same distribution as A to match the number of immediate sub-levels in *L1*. For example, if there are three sub-levels included in *L1*, *GNU MCSim* creates two clones to form a total of three instances of A (the original and its two clones). This convention saves a lot writing and effort in the long run.

In the example of code given above, the parameter A, defined at the top level, will be simply moved to sub-level 1 (cloning is not necessary since there is only on sub-level directly included in the top level). Within sub-level 1, the normally-distributed A will be cloned once in order to create another normal variate with the same distribution. Each one of those two will be moved to a lower `Simulation`, where they will be conditioned by the data of that simulation only. A total of three variables of "type" A will be sampled and will be printed in the output file (coded so that the position in the hierarchy is apparent): the "parent" $A(1)$, a priori uniformly distributed, and two "dependents" $A(1.1)$ and $A(1.2)$, *a priori* normally distributed around $A(1)$.

Note that you can check on the console screen the detail of the actual multilevel structure of your statistical model at run-time by using option `-D=print-hierarchy` (see Section 6.1 [Using the compiled program], page 37).

`Data()` specification

Experimental observations of model variables, inputs, outputs, or parameters, can be specified with the `Data()` command. Markov chain Monte Carlo sampling requires that you specify `Data()` statements (see [MCMC() specification], page 43; see Section 6.2.5 [Setting-up statistical models], page 55). The data are then used internally to evaluate the likelihood function for the model. The arguments are the name of the variable for which observations exist, and a comma-separated list of data values:

```
Data(<variable identifier>, <value1>, <value2>, ...);
```

This specification can only be used with a matching `Print()` or `PrintStep()` for the same variable (see [Print() specification], page 54; see [PrintStep() specification], page 54). You must make sure that there are as many data values in the `Data()` specification as output time requested in the corresponding `Print()` or `PrintStep()`. A data value of "-1" is treated as "missing data" and ignored in likelihood calculations. The convention "-1" can be changed by changing INPUT_MISSING_VALUE in the header file `mc.h` and recompiling.

6.3 Analyzing simulation output

The output from Monte Carlo or `SetPoints` simulations is a tab-delimited text file with one row for each run (i.e., parameter set) and one column for each parameter and output in the order specified. Thus each line of the output file is in the following order:

```
<# of run> <parameters> <outputs for Exp 1> <outputs for Exp2> ...
```

The parameters are printed in the order they were sampled or set.

The first line gives the column headers. A variable called *name* requested for output in an simulation *i* at a time *j* is labeled *name_i.j*.

The output of Markov chain Monte Carlo simulations is also a text file with one row for each run. It displays a column of iteration labels, and one column for each parameter sampled. The last three columns contain respectively, the sum of the logarithms of each parameter's density given its parents' values ('`LnPrior`'), the logarithm of the data likelihood ('`LnData`'), and the sum of the previous two values ('`LnPosterior`'). The first line gives the column headers. On this line, parameters names are tagged with a code identifying their position in the hierarchy defined by the Level sections. For example, the second instance of a parameter called *name* placed at the fist level of the hierarchy is labeled *name*(*2*); the first instance of the same parameter placed at the second instance of the second level of the hierarchy is labeled *name*(*2.1*), etc.

If tempered Markov chain Monte Carlo simulations have been requested, the overall output format is the same as above, with the addition of a column of perk index (the lowest perk having index zero), and columns of log-pseudoprior values (one column per perk, starting with the log-pseudoprior of the lowest perk). Those extra colums are labelled '`IndexT`', '`LnPseudoPrior(1)`', '`LnPseudoPrior(2)`', etc. and come just after the parameters. The samples at the highest perk (typically, perk 1) can be found by selecting the highest value of '`IndexT`'.

In addition, in the case of tempered MCMC, an history of the perks values computed is saved with the same name as the output file, with a *.perks* extension. It lists perk values, counts of samples and log-pseudoprior at each perk, attempted jumps and successful jumps between perks. This can help check the perk schedule optimisation (in particular whether perk zero has been reached) and the behavior of the chain over perks values. If the perks are correctly spaced, there should be about equal number of samples at each perk, and jumps between perks should be frequent.

Those tab-delimited files can easily be imported into your favorite spreadsheet, graphic or statistical package for further analysis.

6.4 Error handling

If integration fails for a `imulation` in `DefaultSim` simulations no output is generated for that simulation, and the user is warned by an error message on the screen. In `MonteCarlo` or `SetPoints` simulations, the corresponding simulation line is not printed, but the iteration number is incremented. Finally, in MCMC simulations, the parameter for which the data likelihood was computed is simply not updated (which implicitly forbids the uncomputable region of the parameter space). In all cases an error message is given on the screen, or wherever the screen output has been redirected.

7 Common Pitfalls

The following mistakes are particularly easy to make, and sometimes hard to notice, or understand at first.

- Forgetting about type-related arithmetics in C: ‘`1000/882`’ gives ‘`1`’ since it is interpreted as an integer division by the compiler. To get a floating-point (usual) division use ‘`1000./882.`’.
- Forgetting a semi-colon (’;’) at the end of statements: the error is usually detected at the following line(s) where in fact nothing may be wrong.

8 XMCSim

XMCSim is a menu-driven interface which automatizes the compilation and running tasks of *GNU MCSim*. It also offers a convenient interface to 2-D and 3-D plotting of the simulation results. Note that you need `XWindows`, `Tcl/Tk` and `wish` installed to run *XMCSim*. `xemacs` is also recommended.

Just type `xmcsim` at the command promt. A windows appear, with a menu bar. Menu items are:

- `File`, which allows you to choose an existing model file or to exit the program. Once you have chosen a model file, its file name appears as a reminder at the bottom of the window.
- `Edit`, which calls `xemacs` for you to create a new model file or edit any file of your choice (for example an input or output file). Note: if you do not have `xemacs` installed you can change the file `xmcsim` to replace the call to `xemacs` by a call to your editor.
- `Compile` has two items: `Compile model` will compile the current model file or prompt you for one and will call `mod` to generate a `model.c` file from it; `Compile mcsim` will first call `mod` and will then go on to create an executable mcsim filevia a call to `makemcsim` create an executable program.
- `Run` with three items: `Run` which will prompt you for an executable mcsim file, an input file and an output file (the latter is optional) and will then launch the executable; `Stop` will just stop a running executable; `Debug` will produce a standalone executable with a name starting with `debugmcsim` and will launch `xemacs` for you (you will then need to call `gdb` or another debugger by yourself; if you find a way to start gdb on an executable *via* xemacs on the command line please tell me...).
- `Plot` will start an Xgnuplot-based interface to `gnuplot` An `Help` menu available there to guide you further in the arcanes of `gnuplot`, but we recommend that you also browse `gnuplot` documentation.

At some point *GNU MCSim* may do symbolic computations, wash dishes, clothes and cars, and write poems, but for now, that's all, folks!

Bibliographic References

Amzal B., Bois F., Parent E. and Robert C.P. (2006). Bayesian optimal design via interacting MCMC, Journal of the American Statistical Association, **101**:773-785.

Barry T.M. (1996). Recommendations on the testing and use of pseudo-random number generators used in Monte Carlo analysis for risk assessment. Risk Analysis **16**:93-105.

Bernardo J.M. and Smith A.F.M. (1994). Bayesian Theory. Wiley, New York.

Bois F.Y., Gelman A., Jiang J., Maszle D., Zeise L. and Alexeef G. (1996). Population toxicokinetics of tetrachloroethylene. Archives of Toxicology **70**:347-355.

Bois F.Y., Smith T.J., Gelman, A., Chang H.Y., Smith A.E. (1999). Optimal design for a study of butadiene toxicokinetics in humans. Toxicological Sciences **49**:213-224.

Bois F.Y., Zeise L. and Tozer T.N. (1990). Precision and sensitivity analysis of pharmacokinetic models for cancer risk assessment: tetrachloroethylene in mice, rats and humans. Toxicology and Applied Pharmacology **102**:300-315.

Calderhead B. and Girolami M. (2009). Estimating Bayes factors via thermodynamic integration and population MCMC. Computational Statistics and Data Analysis **53**:4028-4045.

Gear C.W. (1971a). Algorithm 407 - DIFSUB for solution of ordinary differential equations [D2]. Communications of the ACM **14**:185-190.

Gear C.W. (1971b). The automatic integration of ordinary differential equations. Communications of the ACM **14**:176-179.

Gelman A. (1992). Iterative and non-iterative simulation algorithms. Computing Science and Statistics **24**:433-438.

Gelman A., Bois F.Y. and Jiang J. (1996). Physiological pharmacokinetic analysis using population modeling and informative prior distributions. Journal of the American Statistical Association **91**:1400-1412.

Gelman A., Carlin J.B., Stern H.S. and Rubin D.B. (1995). Bayesian Data Analysis. Chapman & Hall, London.

Gelman A. and Rubin D.B. (1992). Inference from iterative simulation using multiple sequences (with discussion). Statistical Science **7**:457-511.

Geyer C.J. and Thompson E.A. (1995). Annealing Markov chain Monte Carlo with applications to ancestral inference. Journal of the American Statistical Association**90**: 909-920.

Gilks W.R., Richardson S. and Spiegelhalter D.J. (1996). Markov Chain Monte Carlo In Practice. Chapman & Hall, London.

Hammersley J.M. and Handscomb D.C. (1964). Monte Carlo Methods. Chapman and Hall, London.

Manteufel R.D. (1996). Variance-based importance analysis applied to a complex probabilistic performance assessment. Risk Analysis **16**:587-598.

Park S.K. and Miller K.W. (1988). Random number generators: good ones are hard to find. Communications of the ACM **31**:1192-1201.

Press W.H., Flannery B.P., Teukolsky S.A. and Vetterling W.T. (1989). Numerical Recipes (2nd ed.). Cambridge University Press, Cambridge.

Robert C.P. (1995). Simulation of truncated normal variables. Statistics and Computing. **5**(2):121-125. doi:10.1007/BF00143942.

Savic R.M., Jonker D.M., Kerbusch T., Karlsson M.O. (2007). Implementation of a transit compartment model for describing drug absorption in pharmacokinetic studies. Journal of Pharmacokinetics and Pharmacodynamics **34**:711726. doi:10.1007/s10928-007-9066-0.

Smith A.F.M. (1991). Bayesian computational methods. Philosophical Transactions of the Royal Society of London, Series A **337**:369-386.

Smith A.F.M. and Roberts G.O. (1993). Bayesian computation via the Gibbs sampler and related Markov chain Monte Carlo methods. Journal of the Royal Statistical Society Series B **55**:3-23.

Vattulainen I., Ala-Nissila T. and Kankaala K. (1994). Physical tests for random numbers in simulations. Physical Review Letters **73**:2513-2516.

Appendix A Keywords List

You should use the following reserved keywords as prescribed when building your models and input files:

- Add
- Beta
- BetaRandom
- Binomial
- BinomialBetaRandom
- BinomialRandom
- CDFNormal
- CalcDelay
- CalcOutputs
- Cauchy
- CauchyRandom
- Chi2
- Chi2Random
- Compartment
- Cvodes
- Data
- DefaultSim
- Density
- Distrib
- dt
- Dynamics
- End
- erfc
- Euler
- Events
- ExpRandom
- Experiment
- Exponential
- GGammaRandom
- Gamma
- GammaRandom
- GetSeed
- HalfCauchy
- HalfNormal
- IFN

- Initialize
- Inline
- Inputs
- Integrate
- InvGGammaRandom
- InvGamma
- InvTemperature
- Jacobian
- Level
- Likelihood
- lnDFNormal
- lnGamma
- LogNormal
- LogNormalRandom
- LogNormal_v
- LogUniform
- LogUniformRandom
- Lsodes
- MCMC
- MonteCarlo
- Multiply
- NegativeBinomial
- NegativeBinomialRandom
- NDoses
- Normal
- NormalRandom
- Normal_cv
- Normal_v
- OptimalDesign
- OutputFile
- Outputs
- PerDose
- PerExp
- Piecewise
- PiecewiseRandom
- PKTemplate
- Poisson
- PoissonRandom
- Prediction

- Print
- PrintStep
- Replace
- SBMLModels
- Scale
- SetPoints
- SetSeed
- SimType
- Simulation
- Spikes
- StartTime
- States
- StudentT
- StudentTRandom
- t
- TruncInvGamma
- TruncInvGGammaRandom
- TruncLogNormal
- TruncLogNormalRandom
- TruncLogNormal_v
- TruncNormal
- TruncNormalRandom
- TruncNormal_cv
- TruncNormal_v
- Uniform
- UniformRandom
- useID

Appendix B Examples

You will find here some examples of model description files and simulation input files.

B.1 `linear.model`

```
# Linear Model with a random component
# y = A + B * time + N(0,SD_true)
# Setting SD_true to zero gives the deterministic version
#---------------------------------------------------------

# Outputs
Outputs = {y};

# Model Parameters
A = 0;
B = 1;
SD_true = 0;
SD_esti = 0;

CalcOutputs { y = A + B * t + NormalRandom(0,SD_true); }
```

B.2 `1cpt.model`: A example model description file

```
# One Compartment Model
# First order input and output
#---------------------------------------------------------

# Inputs
Inputs = {Dose};

# Outputs
Outputs = {C_central, AUC, ln_C_central, ln_AUC,
           SD_C_computed, SD_A_computed};

# Model Parameters
ka = 1;
ke = 0.5;
F  = 1;
V  = 2;

# Statistical Parameters
SDb_ka = 0;
SDw_ka = 0;
SDb_ke = 0;
SDw_ke = 0;
SDb_V  = 0;
min_F  = 0;
```

```
max_F  = 0;
SD_C_central = 0;
SD_AUC       = 0;
CV_C_cen     = 0;
CV_AUC       = 0;
CV_C_cen_true = 0;
CV_AUC_true   = 0;


# Calculate Outputs
CalcOutputs {

  # algebraic equation for C_central
  C_central = (ka != ke ?
               (exp(-ke * t) - exp(-ka * t)) *
               F * ka * Dose / (V * (ka - ke))):
               exp(-ka * t) * ka * t * F * Dose / V);

  # algebraic equation for AUC
  AUC = (ka != ke ?
         ((1 - exp(-ke * t)) / ke -
          (1 - exp(-ka * t)) / ka) * F * ka * Dose / (V * (ka - ke))) :
         F * Dose * (1 - (1 + ka * t) * exp(-ka * t)) / (V * ke));

  C_central = C_central + NormalRandom(0, C_central * CV_C_cen_true);
  AUC       = AUC + NormalRandom(0, AUC * CV_AUC_true);

  ln_C_central = (C_central > 0 ? log (C_central) : -100);
  ln_AUC = (AUC > 0 ? log (AUC) : -100);

  SD_C_computed = (C_central > 0 ? C_central * CV_C_cen : 1e-10);
  SD_A_computed = (AUC > 0 ? AUC * CV_AUC : 1e-10);

} # End of output calculations

End.
```

B.3 `perc.model`: A example model description file

```
#----------------------------------------------------------
# perc.model
# A four compartment model of Tetrachloroethylene (PERC)
# and total metabolites.
#----------------------------------------------------------
# States are quantities of PERC and metabolite formed, they can be
# output

States = {Q_fat,        # Quantity of PERC in the fat
          Q_wp,         #   ...   in the well-perfused compartment
          Q_pp,         #   ...   in the poorly-perfused compartment
          Q_liv,        #   ...   in the liver
          Q_exh,        #   ...   exhaled
          Qmet};        # Quantity of metabolite formed


# Extra outputs are concentrations at various points

Outputs = {C_liv,               # mg/l in the liver
           C_alv,               # ... in the alveolar air
           C_exh,               # ... in the exhaled air
           C_ven,               # ... in the venous blood
           Pct_metabolized,     # % of the dose metabolized
           C_exh_ug};           # ug/l in the exhaled air

Inputs = {C_inh}                # Concentration inhaled

# Constants
# Conversions from/to ppm: 72 ppm = .488 mg/l

PPM_per_mg_per_l = 72.0 / 0.488;
mg_per_l_per_PPM = 1/PPM_per_mg_per_l;

#----------------------------------------------------------
# Nominal values for parameters
# Units:
# Volumes: liter
# Vmax:    mg / minute
# Weights: kg
# Km:      mg / minute
# Time:    minute
# Flows:   liter / minute
#----------------------------------------------------------

InhMag = 0.0;
```

```
Period = 0.0;
Exposure = 0.0;

C_inh = PerDose (InhMag, Period, 0.0, Exposure);

LeanBodyWt = 55;    # lean body weight

# Percent mass of tissues with ranges shown

Pct_M_fat  = .16;   # % total body mass
Pct_LM_liv = .03;   # liver, % of lean mass
Pct_LM_wp  = .17;   # well perfused tissue, % of lean mass
Pct_LM_pp  = .70;   # poorly perfused tissue, recomputed in scale

# Percent blood flows to tissues

Pct_Flow_fat = .09;
Pct_Flow_liv = .34;
Pct_Flow_wp  = .50; # will be recomputed in scale
Pct_Flow_pp  = .07;

# Tissue/blood partition coeficients

PC_fat = 144;
PC_liv = 4.6;
PC_wp  = 8.7;
PC_pp  = 1.4;
PC_art = 12.0;

Flow_pul   = 8.0;    # Pulmonary ventilation rate (minute volume)
Vent_Perf = 1.14;    # ventilation over perfusion ratio

sc_Vmax = .0026;     # scaling coeficient of body weight for Vmax

Km = 1.0;

# The following parameters are calculated from the above values in
# the Scale section before the start of each simulation.
# They are left uninitialized here.

BodyWt = 0;

V_fat = 0;           # Actual volume of tissues
V_liv = 0;
V_wp  = 0;
V_pp  = 0;
```

```
Flow_fat = 0;        # Actual blood flows through tissues
Flow_liv = 0;
Flow_wp  = 0;
Flow_pp  = 0;

Flow_tot = 0;        # Total blood flow
Flow_alv = 0;        # Alveolar ventilation rate

Vmax = 0;            # kg/minute


#---------------------------------------------------------
# Dynamics
# Define the dynamics of the simulation. This section is
# calculated with each integration step. It includes
# specification of differential equations.
#---------------------------------------------------------

Dynamics {

# Venous blood concentrations at the organ exit

Cout_fat = Q_fat / (V_fat * PC_fat);
Cout_wp  = Q_wp  / (V_wp  * PC_wp);
Cout_pp  = Q_pp  / (V_pp  * PC_pp);
Cout_liv = Q_liv / (V_liv * PC_liv);

# Sum of Flow * Concentration for all compartments

dQ_ven = Flow_fat * Cout_fat + Flow_wp * Cout_wp
         + Flow_pp * Cout_pp + Flow_liv * Cout_liv;

# Venous blood concentration

C_ven =  dQ_ven / Flow_tot;

# Arterial blood concentration
# Convert input given in ppm to mg/l to match other units

C_art = (Flow_alv * C_inh / PPM_per_mg_per_l +  dQ_ven) /
        (Flow_tot + Flow_alv / PC_art);

# Alveolar air concentration

C_alv = C_art / PC_art;

# Exhaled air concentration
```

```
C_exh = 0.7 * C_alv + 0.3 * C_inh / PPM_per_mg_per_l;

# Differentials

dt (Q_exh) = Flow_alv * C_alv;
dt (Q_fat) = Flow_fat * (C_art - Cout_fat);
dt (Q_wp)  = Flow_wp  * (C_art - Cout_wp);
dt (Q_pp)  = Flow_pp  * (C_art - Cout_pp);

# Quantity metabolized in liver

dQmet_liv = Vmax * Q_liv / (Km + Q_liv);
dt (Q_liv) = Flow_liv * (C_art - Cout_liv) - dQmet_liv;

# Metabolite formation

dt (Qmet)  = dQmet_liv;

} # End of Dynamics


#---------------------------------------------------------
# Scale
# Scale certain model parameters and resolve dependencies
# between parameters. Generally the scaling involves a
# change of units, or conversion from percentage to actual
# units.
#---------------------------------------------------------

Scale {

# Volumes scaled to actual volumes

BodyWt = LeanBodyWt/(1 - Pct_M_fat);

V_fat = Pct_M_fat  * BodyWt/0.92;        # density of fat = 0.92 g/ml
V_liv = Pct_LM_liv * LeanBodyWt;
V_wp  = Pct_LM_wp  * LeanBodyWt;
V_pp  = 0.9 * LeanBodyWt - V_liv - V_wp; # 10% bones

# Calculate Flow_alv from total pulmonary flow

Flow_alv = Flow_pul * 0.7;

# Calculate total blood flow from the alveolar ventilation rate and
# the V/P ratio.
```

```
Flow_tot = Flow_alv / Vent_Perf;

# Calculate actual blood flows from total flow and percent flows

Flow_fat = Pct_Flow_fat * Flow_tot;
Flow_liv = Pct_Flow_liv * Flow_tot;
Flow_pp  = Pct_Flow_pp  * Flow_tot;
Flow_wp  = Flow_tot - Flow_fat - Flow_liv - Flow_pp;

# Vmax (mass/time) for Michaelis-Menten metabolism is scaled
# by multiplication of bdw^0.7

Vmax = sc_Vmax * exp (0.7 * log (LeanBodyWt));

} # End of model scaling


#---------------------------------------------------------
# CalcOutputs
# The following outputs are only calculated just before values
# are saved.  They are not calculated with each integration step.
#---------------------------------------------------------

CalcOutputs {

# Fraction of TCE metabolized per day

Pct_metabolized = (InhMag ?
                   Qmet / (1440 * Flow_alv * InhMag * mg_per_l_per_PPM):
                   0);

C_exh_ug  = C_exh * 1000; # milli to micrograms

} # End of output calculation

End.
```

B.4 perc.lsodes.in

```
#---------------------------------------------------------
# perc.lsodes.in
#
#---------------------------------------------------------

Integrate (Lsodes, 1e-4, 1e-6, 1);


#---------------------------------------------------------
# The following is a simulation of one of Dr. Monster's
# exposure experiments described in "Kinetics of Tetracholoroethylene
# in Volunteers; Influence of Exposure Concentration and Work Load,"
# A.C. Monster, G. Boersma, and H. Steenweg,
# Int. Arch. Occup. Environ. Health, v42, 1989, pp303-309
#
# The paper documents measurements of levels of TCE in blood and
# exhaled air for a group of 6 subjects exposed to
# different concentrations of PERC in air.
#
# Inhalation is specified as a dose of magnitude InhMag for the
# given Exposure time.
#
# Inhalation is given in ppm
#---------------------------------------------------------

Simulation {

InhMag = 72;            # ppm
Period = 1e10;          # Only one dose
Exposure = 240;         # 4 hour exposure

# measurements before end of exposure
# and at [5' 30'] 2hr 18 42 67 91 139 163

Print (C_exh_ug, 239.9 245 270 360 1320 2760 4260 5700 8580 10020 );
Print (C_ven, 239.9 360 1320 2760 4260 5700 8580 10020 );

}

END.
```

Concept Index

'

'!=' operator 21
'#' sign 21
'-' (hyphen) sign 22
':' sign 21
';' sign 21
'<' operator 21
'<=' operator 21
'<>' operator 21
'=' sign 21
'==' operator 21
'>' operator 21
'>=' operator 21
'?' sign 21
'[]' (square brackets) notation 22
'_' (underscore) in templates 34
'_' (underscore) vector notation 23

A

Algebraic models 25
Analyzing simulation output 58
Arrays 22
Assignment 21

B

`Beta` distribution 48
`BetaRandom()` function 26
Bibliographic references 65
`Binomial` distribution 48
`BinomialBetaRandom()` function 26
`BinomialRandom()` function 26
Blank lines 21

C

`CalcOutputs`, output section 31
`Cauchy` distribution 48
`CauchyRandom()` function 26
`CDFNormal()` function 26
Chi-square distribution 48
`Chi2` distribution 48
`Chi2Random()` function 26
Colon conditional assignment 21
Comments 21
Common pitfalls 61
Comparison operators 21
`Compartment` keyword 34
Complementary error function 26
Conditional assignment 21
Cumulative density function, Normal 26
`Cvodes` integrator 41

D

`Data()` qualifier, for use in `Distrib()` 50
`Data()` specification 58
Default 10
Defining models 19
Delay differential equations 31
Density function, Normal 26
`Density()` specification 50
Derivative specification 30
Differential models 25
Discrete-time models 25
`Distrib()` specification 48
Distribution, `Beta` 48
Distribution, `Binomial` 48
Distribution, `Cauchy` 48
Distribution, Chi-square 48
Distribution, `Chi2` 48
Distribution, `Exponential` 48
Distribution, `Gamma` 48
Distribution, `HalfCauchy` 48
Distribution, `HalfNormal` 49
Distribution, inverse-gamma 49
Distribution, `InvGamma` 49
Distribution, `LogNormal` 49
Distribution, `LogNormal_v` 49
Distribution, `Loguniform` 49
Distribution, `NegativeBinomial` 49
Distribution, `Normal` 49
Distribution, `Normal_cv` 49
Distribution, `Normal_v` 49
Distribution, `Piecewise` 49
Distribution, `Poisson` 49
Distribution, `StudentT` 49
Distribution, triangular 49
Distribution, truncated inverse-gamma 49
Distribution, truncated lognormal 49
Distribution, truncated normal 50
Distribution, `TruncInvGamma` 49
Distribution, `TruncLogNormal` 49
Distribution, `TrunclogNormal_v` 50
Distribution, `TruncNormal` 50
Distribution, `Truncnormal_cv` 50
Distribution, `Truncnormal_v` 50
Distribution, `Uniform` 50
Distribution, `UserSpecifiedLL` 50
Documentation license 1
`Dt()` operator 30
Dynamics section 30

E

End keyword 23
`erfc()` function 26
Error function, complementary 26

Error handling 59
`Euler` integrator 41
`Events` specification 53
Examples 19, 71
`Experiment` sections 52
Experimental design optimization 10
`Exponential` distribution 48
`ExpRandom()` function 26

F

Function, `BetaRandom()` 26
Function, `BinomialBetaRandom()` 26
Function, `BinomialRandom()` 26
Function, `CauchyRandom()` 26
Function, `CDFNormal()` 26
Function, `Chi2Random()` 26
Function, `erfc()` 26
Function, `ExpRandom()` 26
Function, `GammaRandom()` 26
Function, `GetSeed()` 26
Function, `GGammaRandom()` 26
Function, Inline, in line 29
Function, input 21
Function, `InvGGammaRandom()` 26
Function, `lnDFNormal()` 26
Function, `lnGamma()` 26
Function, `LogNormalRandom()` 26
Function, `LogUniformRandom()` 26
Function, `NDoses()` 28, 40
Function, `NegativeBinomialRandom()` 26
Function, `NormalRandom()` 26
Function, `PerDose()` 27, 40
Function, `PerExp()` 27, 40
Function, `PerTransit()` 27, 40
Function, `PiecewiseRandom()` 26
Function, `PoissonRandom()` 26
Function, `SetSeed()` 26
Function, special 21
Function, `Spikes()` 28, 40
Function, `StudentTRandom()` 27
Function, `TruncInvGGammaRandom()` 27
Function, `TruncLogNormalRandom()` 27
Function, `TruncNormalRandom()` 27
Function, `UniformRandom()` 27
Functions, input 27, 40
Functions, special 25

G

`Gamma` distribution 48
Gamma function 26
`GammaRandom()` function 26
General input file syntax 39
`GetSeed()` function 26
`GGammaRandom()` function 26
Global specifications 40

H

`HalfCauchy` distribution 48
`HalfNormal` distribution 49

I

Initialize, initialization section, ... 29
Inline, in line functions 29
Input functions 21, 27, 40
Input variables 24
Installation 13
`Integrate()` specification 41
Integration routine, `Cvodes` 41
Integration routine, `Euler` 41
Integration routine, `Lsodes` 41
Integration variable 30
Inverse-gamma distribution 49
`InvGamma` distribution 49
`InvGGammaRandom()` function 26
`InvTemperature()` specification 44

K

Keyword, `Compartment` 34
Keywords list 67

L

`Level` sections 57
Licenses 1
`Likelihood()` specification 50
`lnDFNormal()` function 26
`lnGamma()` function 26
Logical tests 21
`LogNormal` distribution 49
`LogNormal_v` distribution 49
`LogNormalRandom()` function 26
`LogUniform` distribution 49
`LogUniformRandom()` function 26
`Lsodes` integrator 41

M

Major changes in version 5.4.0 10
Major changes in version 5.5.0 10
Major changes in version 5.6.0 11
Major changes in version 6.0.0 11
Major changes in version 6.1.0 12
Major changes in version 6.2.0 12
`makemcsim` script 20
`makemcsimd` script 20
`makemcsimp` script 20
`makemcsims` script 20
Markov-chain Monte Carlo simulations 10, 43
`MCMC` simulations 10, 43
`MCMC()` specification 43
`mod` syntax 20

`mod` usage 19
Model definition files 19
Model types 25
Models, algebraic 25
Models, differential 25
Models, discrete-time 25
Models, statistical 55
Monte Carlo simulations 10, 42
`MonteCarlo()` specification 42
MPI 20

N

`NDoses()` function 28, 40
`NegativeBinomial` distribution 49
`NegativeBinomialRandom()` function 26
Normal cumulative density function 26
Normal density function 26
`Normal` distribution 49
`Normal_cv` distribution 49
`Normal_v` distribution 49
`NormalRandom()` function 26

O

`OptimalDesign()` specification 46
Output specification 31
Output variables 24
`OutputFile()` specification 41
Overview 9

P

Parallelization 14
Parallelized code 20
Parameter declaration 23
Parameter scaling 29
`PerDose()` function 27, 40
`PerExp()` function 27, 40
`PerTransit()` function 27, 40
`Piecewise` distribution 49
`PiecewiseRandom()` function 26
Pitfalls 61
`Poisson` distribution 49
`PoissonRandom()` function 26
`Prediction()` qualifier, for use in `Distrib()` . . . 50
`Prediction()` specification (in `Simulation` context) 54
`Print()` specification 54
`PrintStep()` specification 54

Q

Qualifier, `Data()` 50
Qualifier, `Prediction()` 50

R

R deSolve 35
Random number, beta 26
Random number, binomial 26
Random number, binomial-beta 26
Random number, Cauchy 26
Random number, Chi-squared 26
Random number, exponential 26
Random number, gamma 26
Random number, general-gamma 26
Random number, inverse-gamma 26
Random number, lognormal 26
Random number, loguniform 26
Random number, negative binomial 26
Random number, normal 26
Random number, piecewise 26
Random number, Poisson 26
Random number, Student t 27
Random number, truncated inverse-gamma 27
Random number, truncated lognormal 27
Random number, truncated normal 27
Random number, uniform 27
Random seed, reading its value 26
Random seed, setting its value 26
Reserved keywords 67
Running simulations 37

S

SBML models 32
Scale, scaling section 29
Semi-colon 21
`SetPoints` simulations 10
`SetPoints()` specification 45
`SetSeed()` function 26
Setting-up statistical models 55
`SimType()` specification 52
Simulation definition files 37
Simulation file, syntax 38
`Simulation` sections 52
Software license 1
Special functions 21, 25
Specification, `Data()` 58
Specification, `Density()` 50
Specification, `Distrib()` 48
Specification, `Integrate()` 41
Specification, `InvTemperature()` 44
Specification, `Likelihood()` 50
Specification, `MCMC()` 43
Specification, `MonteCarlo()` 42
Specification, `OptimalDesign()` 46
Specification, `OutputFile()` 41
Specification, `Print()` 54
Specification, `PrintStep()` 54
Specification, `SetPoints()` 45
Specification, `SimType()` 52
Specification, `StartTime()` 54
Specifications, global 40

Specifying simulations 37

`Spikes()` function 28, 40

Square brackets 22

`StartTime()` specification 54

State discontinuities 53

State variables 24

Statistical models 55

Structural models 19, 25

`StudentT` distribution 49

`StudentTRandom()` function 27

Style 32

Syntax for `mod` 20

Syntax of simulation files 38

T

Tempered MCMC 44

Template models 32

Tests, logical 21

Triangular distribution 49

Truncated Inverse-gamma distribution 49

Truncated lognormal distribution 49

Truncated normal distribution 50

`TruncInvGamma` distribution 49

`TruncInvGGammaRandom()` function 27

`TruncLogNormal` distribution 49

`TruncLogNormal_v` distribution 50

`TruncLogNormalRandom()` function 27

`TruncNormal` distribution 50

`TruncNormal_cv` distribution 50

`TruncNormal_v` distribution 50

`TruncNormalRandom()` function 27

U

Underscore syntax for vectors 23

Underscore use for template models 34

`Uniform` distribution 50

`UniformRandom()` function 27

`UserSpecifiedLL` distribution 50

Using deSolve 35

V

Variable names 21

Vectorization 22

Vectors 22

W

Working Through an Example 17

Writing and Compiling structural models 19

X

`xmcsim` 63