MCSim:

A Monte Carlo Simulation Program

by Frédéric Y. Bois and Don R. Maszle

User's Manual, software version 5.0.0

contact:

Frederic Bois
INERIS
Parc Technologique ALATA
F-60550, Verneuil en Halatte

frederic.bois@ineris.fr
fredomatic@free.fr

1 Software License

GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

1.1 PREAMBLE

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software–to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

1.2 TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

- a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

- b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

- c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

• a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

• b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

• c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program 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.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public 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.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

2 Overview

MCSim is a simulation and statistical inference tool for algebraic or differential equation systems. Other programs have been created to the same end, the Matlab family of graphical interactive programs being some of the more general and easy to use. Still, many available tools are not optimal for performing computer intensive and sophisticated Monte Carlo analyses. *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 are linked to your model to produce executable code. After linking, you will be able to run simulations of your structural model under a variety of conditions, specify an associated statistical model, and perform Monte Carlo 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 [Setting-up Structural Models], page 17). 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). *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 29). 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 33).
- 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 33). 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 42).
- A "*SetPoints*" simulation solves the model for a series of specified parameter sets, listed in a separate ASCII file (see [SetPoints() specification], page 35). 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 36).

2.3 Major changes introduced with version 5.0.0

- An `autoconf` script simplifies installation under Unix/Linux operating systems.
- Starting with this version *MCSim* uses some routines of the GNU scientific libray (`gsl`), which is now required.
- Random variate generating functions "BinomialRandom", "BetaRandom", "BinomialBetaRandom", "Chi2Random", "ExpRandom", "InvGGammaRandom", "GammaRandom, "GGammaRandom", "LogNormalRandom", "LogUniformRandom", "NormalRandom", "PiecewiseRandom", "PoissonRandom", "TruncInvGGammaRandom", "TruncLogNormalRandom", and "TruncNormalRandom", are available for structural model building.
- For MCMC simulations, the parameters of a `Distrib()` specification can include `Data()` qualifiers, in addition to `Prediction()` qualifiers. At any position (except in the second, reserved for the distribution name) of a `Distrib()` specification, `Data()`

can be used to designate data about an input, state or output variable. For any distribution shape parameter, `Prediction()` can also specify a model input, state or output. This give much more flexibility to `Distrib()` specifications. For example:

```
Distrib (Data(R), Binomial, Prediction(P), Data(N));
```

is now valid. In addition, the new keywords, `Likelihood()` and `Density()`, have been defined; they are equivalent to `Distrib()` and have the same syntax. They can help specify clearer statistical models. Likelihoods can now be different at different levels or sub-levels.

- A vector syntax, using square brackets is available for specifying the structural model (note: this syntax is not yet available for input file and is for now of limited usefulness).
- The "OptimDesign()" specification (see [OptimalDesign() specification], page 36) allows you to optimize some aspects of the design of planned experiments.
- The `xmcsim` graphical user's interface (see Chapter 8 [XMCSim], page 49) allows you to run *MCSim* under XWindows and allows graphic outputs.

For a detailed list of changes, you should consult the '`MCSim-changelog`' file distributed with the source code.

3 Installation

3.1 System requirements

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 *MCSim* is using routines from the GNU Scientific Library (`gsl`). Version 1.5 (or higher) of the shared gsl library, gslcblas library, and gsl include files should be installed on your system.

On a Unix or Linux 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 environment (for example by generating only the documentation formats that you can read).

For other operating systems (MacOS, Windows...) you will need a C language development environment or at least a compiler, and some familiarity with it. Here also you might considerer installing a freeware version of `gcc`, such as `djgpp`.

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

3.2 Distribution

MCSim source code is available on Internet through:

'`ftp://prep.ai.mit.edu/pub/gnu/`',

'`http://www.gnu.ai.mit.edu/home.html`',

'`http://fredomatic.free.fr`',

'`http://toxi.ineris.fr`'.

and mirror sites of the GNU project.

Two mailing lists are available for *MCSim* users:

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

- "help-mcsim@prep.ai.mit.edu".

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

- "bug-mcsim@prep.ai.mit.edu".

You can also subscribe to those lists if you want to automatically receive bug reports and help messages from others:

To add yourself, send to (list name)-request@prep.ai.mit.edu the word subscribe (as subject or content). So, for example, for help-mcsim@prep.ai.mit.edu the address would be help-mcsim-request@prep.ai.mit.edu.

To remove yourself, send to (list name)-request@prep.ai.mit.edu the word unsubscribe (as subject or content).

3.3 Machine-specific installation

3.3.1 Unix/Linux operating systems

To install on a Unix/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.so` is placed in '`/usr/lib`'. A copy of the '`mcsim`' source directory (with the '`mod`', '`sim`', '`doc`', '`samples`', and '`xmcsim`' subdirectories) is placed in '`/usr/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/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 or just want to install *MCSim* in your own directory, type:

```
make install-here
```

This will copy or move '`mod`', '`makemcsim`', and '`xmcsim`' in a '`/bin`' directory in your home directory, creating it if necessary. The library `libmcsim.so` will be moved to a '`/lib`' directory in your home directory. The other files in the '`mcsim`' installation directory (sources, manuals, samples) are left untouched. You should move the entire '`mcsim`' directory in your home directory, otherwise you will have to edit the '`bin/xmcsim`' and the '`bin/makemcsim`' scripts to indicate the location of the '`mcsim`' directory

3.3.2 Other operating systems

Under other operating systems (MacOS, Windows, etc.) you should be able to both uncompress and untar the archive with widely distributed archiving tools. Refer to the

documentation of your compiler to create an executable '`mod`' file from the source code files (mod.c, lex.c, lexerr.c, lexfn.c, modd.c, modo.c, modi.c, strutil.c) provided in the '`mod`' directory. Place the executable '`mod`' on your command path. The '`sim`' directory contains all the source files (sim.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, 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.

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

4 Working Through an Example

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 51).

Go to the '`mcsim/samples/perc`' directory (installed either locally or by default in '`usr/share`' under Unix/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 57). 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 *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 17). 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 11). 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 *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 62, 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.*

Several other models and simulation specification files are provided with the package as examples (they are in the '`samples`' directory. Try them 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 Setting-up 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/samples`' directory. Some of them are reproduced in Appendix (see Appendix B [Examples], page 55).

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 18) 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/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 *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 18).

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`. 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 11) or to the authors of this manual.

5.2 Using `makemcsim` to fully process 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.

Two variants of `makemcsim` are also available: `makemcsims`, which creates a standalone version (no dynamic libraries needed), and `makemcsimd`, which creates a standalone version with debugging symbols included (so that you can use gdb, for example, to check what the code 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 55):

```
# Model description file (this is a comment)
<Global variable specifications>
Initialize {
  <Equations for initializing or scaling model parameters>
}
Dynamics {
  <Equations for computing derivatives of the state variables>
}
CalcOutputs {
  <Equations for computing output variables>
}
```

`Initialize`, `Dynamics` and `CalcOutputs` are reserved keywords and, if used, must appear as shown, followed by the curly braces which delimit each section (see Section 5.3.6 [Model initialization], page 25; Section 5.3.7 [Dynamics section], page 26; Section 5.3.8 [Output calculations], page 27). 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.

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 [Global variable declarations], page 21). 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 name *IFN*, in capital letters, is reserved by the program and should not be used as 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, and *MCSim*'s "*special functions*" (see Section 5.3.4 [Special functions], page 23) or "*input functions*" (see Section 5.3.5 [Input functions], page 24). Special functions can take already defined variables, constant 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 `>=`.

- Vectors: (Warning: for now, the square bracket notation can be only used in the model definition file. It is not recognized in simulation specification files; for those, the underscore ('_') unrolling syntax can be used to address vector components. This limits the usefulness of this feature).

  Vectors' declaration: To declare a variable 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 will correspond to valid indices) can be given as (long) 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 [Global variable declarations], page 21). 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 23). Here also, input functions (see Section 5.3.5 [Input functions], page 24) 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/samples/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. Note also!: This syntax is the only one that can be used in simulation specification files. If you declare a parameter '`beta[1-10]`' in your model definition file, the only way to refer to it in the simulation input file will be through its individual components '`beta_1`', '`beta_2`', ... *etc.* This limitation will be removed in a future release of the software.

5.3.2 Global variable declarations

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 5.3.7 [Dynamics section], page 26) (higher orders or partial differential equations are not allowed).
- Inputs are variables independent of the others variables, but eventually varying 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) that do not have dynamics. They can receive assignments in the `Dynamics` or `CalcOutputs` sections.

The format for declaring each of these variables is the same, and consists of the keyword `States`, `Inputs` or `Outputs` followed by 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 24). 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 of an `Experiment` (see [Simulation sections], page 40).

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, but 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 42).

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 40) 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 *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/samples/pde1`' and '`mcsim/samples/pde2`')...

You can use *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 32) 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 sample files in the '`mcsim/samples/discrete`' directory provided with the source code for *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;
- `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*;
- `GetSeed()`: returns the current value of the random generator seed;
- `GammaRandom(`*alpha*`)`: returns a gamma distributed random variate with shape parameter *alpha* and inverse scale equal to 1;
- `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*];
- `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*. That *seed* 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);
- `GGammaRandom(`*alpha*`, `*beta*`, `*a*`, `*b*`)`: returns a truncated gamma distributed random variate with shape parameter *alpha* and inverse scale, 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. The algorithm used is that of Park and Miller (Barry, 1996; Park and Miller, 1988; Vattulainen et al., 1994) (see [Bibliographic References], page 51). A default random generator seed (314159265.3589793) is used.

Note: for all the above random number generating functions, a default random generator seed is used. It can be changed 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. *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 32).

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 at <*initial-time*> in the <*period*> and lasts for <*exposure-time*> time units. Syntax:

```
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*>. The input is turned off once the magnitude reaches a negligible fraction (10^{-17}) of its original value. Note that the input does not accumulate across periods, it resets at each period start. Syntax:

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

- `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:

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

  Note that the list 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. 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:

  ```
  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 37). Variable dependencies are resolved before each simulation specified by an `Experiment` (see [Simulation sections], page 40).

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. Magnitudes change exactly at the times given.

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 5.3.7 [Dynamics section], page 26) and `CalcOutputs` (see Section 5.3.8 [Output calculations], page 27) sections.

5.3.6 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 *MCSim* after the assignments specified in each `Experiment` are done (see [Simulation sections], page 40). They are the last initializations performed. The model file in '`mcsim/samples/perc`' gives an example of the use of `Initialize` (see Section B.3 [perc.model], page 57, 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.

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 5.3.7 [Dynamics section], page 26) cannot be used in this section, since derivatives have not yet been computed when the scaling function is called.

5.3.7 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`' sample 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 23). 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.
```

5.3.8 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 41, and see [PrintStep() specification], page 42). 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. 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). The `dt()` operator can appear in the right-hand side of equations, and refers to current values of the derivatives (see Section 5.3.7 [Dynamics section], page 26). Like in the `Dynamics` section, the integration variable can be accessed if referred to as `t`, as in:

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

5.3.9 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 57). 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...

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/samples`' directory. An sample 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 62).

6.1 Using the compiled program

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 17) to generate an executable '`mcsim_my`'. 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 [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 30). 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 32). 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.

When the program starts up, 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. *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 11) 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 40), eventually enclosed in `Level` sections. (The keyword `Experiment` is now obsolete but can still be used as a synonym for `Simulation`.)

Each `Simulation` section defines simulation conditions, from an initial time (or whatever the dependent variable represents, see Section 5.3.3 [Model types], page 22) 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 sample file in Section B.4 [perc.lsodes.in], page 62):

```
# 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 # Optional End statement. Everything after this line is ignored
```

For Markov chain Monte Carlo simulations (see [MCMC() specification], page 33), 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 42). 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 # Optional statement.
```

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 18) 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 values; mathematical expressions are not allowed.
- Input variables' assignments can use any input function (including the `NDoses()` and `Spikes()` functions) or constant values.
- Output variables cannot receive assignments.

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 its end is reached, or until an `End` command 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 24). Briefly, these are:

- `PerDose()`:

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

- `PerExp()`:

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

- `NDoses()`:

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

- `Spikes()`:

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

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. Two integration routines are provided: `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 51) and `Euler` (Press et al., 1989).

The syntax for `Lsodes` is:

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

where <*rtol*> is a scalar specifying the relative error tolerance for each integration step. The scalar <*atol*> specifies the absolute error tolerance parameter. They 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 for stiff systems. You should try both 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 `Euler` is:

```
Integrate(Euler, <time-step>, 0, 0);
```

where <*time-step*> is a scalar specifying the constant time increment for each integration step. The next two scalars are reserved for future use and should be set to zero.

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 51) 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. The seed <*RandomSeed*> of the pseudo-random number generator 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). Here is an example of use:

```
MonteCarlo("percsimmc.out", 50000, 9386.630);
```

The parameters' sampling distributions are specified by a list of `Distrib()` specifications, as explained in the following (see [Distrib() specification], page 37). The format of the output file of Monte Carlo simulations is discussed later (see Section 6.3 [Analyzing simulation output], page 46).

`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 *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 51).

MCMC simulation chains (which in *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). *MCSim* does not deal (yet) with convergence issues.

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

- a *structural model* (see Chapter 5 [Setting-up Structural Models], page 17),
- a *statistical model* (see Section 6.2.5 [Setting-up statistical models], page 42),

– *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 37).

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 45) 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, including the runs eventually read in the restart file. The next field, <*simTypeFlag*> should be either 0, 1, or 2. 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 the value of <*simTypeFlag*> is set to 1 or 2, a restart file must also be specified. In the case of 1, 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, the entire restart 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).

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, the seed <*RandomSeed*> of the pseudo-random number generator can be any positive real number. Seeds between 1.0 and 2147483646.0 are used as is, others are rescaled silently within those bounds.

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 46).

`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 *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. It has the following syntax:

```
SetPoints("<OutputFilename>", "<SetPointsFilename>", <nRuns>,
          <identifier>, <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 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 46).

Following the `SetPoints()` specification, `Distrib()` statements can be given for parameters not already in the list (see [Distrib() specification], page 37). 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.

`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 [Setting-up Structural Models], page 17), a statistical model in the form of a `likelihood()` function (see Section 6.2.5 [Setting-up statistical models], page 42), 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 51). 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>,
              <identifier>, <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. The seed <*RandomSeed*> of the pseudo-random number generator can be any positive real number. Seeds between 1.0 and 2147483646.0 are used as is, 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/samples`'.

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 37).

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 there is somewhat more complex. We start with simple things; the MCMC case will be dealt with later in this section.

In the context of `MonteCarlo()` or `SetPoints()` simulations (see [MonteCarlo() specification], page 33, and [SetPoints() specification], page 35), 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(<identifier>, <distribution-name>, [<shape parameters>]);
```

The <*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 51). 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.

- `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.
- `HalfNormal`, takes one reals number as parameter: the standard deviation, strictly positive. The mean is automatically set to zero.
- `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 42).
- `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.
- `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 42).
- `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.
- `TruncInvGamma` (truncated inverse gamma distribution), needs four strictly positive real parameters: the shape, the scale, the minimum and the maximum.
- `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 42).
- `TruncNormal` (truncated normal distribution), takes four real parameters: the mean, the standard deviation (strictly positive), the minimum and the maximum.
- `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).

The shape parameters of the above distributions 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, *MCSim* provides extensions of the above `Distrib()` specification syntax.

First, when `Distrib()` is used to specify the distribution of a model parameter, 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 42).

Next, in MCMC simulations, you usually assign a probability distribution (or a likelihood) to the data you are trying to analyze. Typically, your model's state and/or output variables will attempt to predict some aspect of the observed data distributions (mean, variance, *etc.*). *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*".

`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 45) or in a separate datafile (see [MCMC() specification], page 33).

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 41 and [PrintStep() specification], page 42).

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

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

Distrib (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);

Distrib (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);
```

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

Last, for more readable input files, two keywords, `Density()` and `Likelihood()`, can be used instead of `Distrib()`. They are equivalent to `Distrib()` and have the same syntax.

`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`), `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 values (see Section 6.2.1 [General input file syntax], page 31). 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 31) or constant 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()`,
- `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 42).

`StartTime()` specification

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

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

If this specification is not given, a value of zero is used by default. The final time is automatically computed to match the largest output time specified in the `Print()` or `PrintStep()` statements.

`Print()` specification

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

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

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 `PrintStep()` 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. The arguments are the name of only one variable, the first output time, the last one, and a time increment:

```
PrintStep(<identifier>, <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()`, 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 *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.

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 *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 *MCSim* source code (see Section B.1 [linear.model], page 55):

```
# ------------------------------------------
# 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, *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 33]). 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 *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 42), since they are equally spaced. More generally, `Print()` can also be used (see [Print() specification], page 41). 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 *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 correesponding 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 40). 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. This limit of 10 can be increased (up to 255) by changing MAX_LEVELS in the header file '`sim.h`' and re-installing *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
```

A `Level` can also make simple assignments to any model parameter (see Section 6.2.1 [General input file syntax], page 31). 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, *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). *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*, *MCSim* creates two clones to form a total of three instances of *A* (the original and its two clones).

In the sample 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). *** "*cloned*" as many times as there are sub-levels or simulations enclosed its defining level (hence, it will be cloned once here). This mean that **two** *A* parameters will

(hence, it will be cloned twice in the example above, once for each `Simulation` section defined). In that way, the parameters distributions defined at one level in fact apply to the next lower sub-level, or at the `Simulation` level. This convention saves a lot writing and effort in the long run. For example, the uniform distribution assigned to *A*, at the top level, applies to the sub-level 1. There is only one "clone" of *A* at sub-level 1 since only one sub-level is included in the top level. In contrast, two normally-distributed "clones" of *A* will be defined and sampled. The first one will apply to simulation 1, and will be conditioned by the data of that simulation only, and the other will apply to simulation 2. 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 $A(1)$.

`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 33; see Section 6.2.5 [Setting-up statistical models], page 42). 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(<identifier>, <value1>, <value2>, ...);
```

This specification can only be used with a matching `Print()` or `PrintStep()` for the same variable (see [Print() specification], page 41; see [PrintStep() specification], page 42). 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.

The tab-delimited file 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 *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 *MCSim* will do symbolic computations, wash dishes, clothes and cars, and write poems, but for now, that's all folks!

Bibliographic References

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.

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.

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.

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 avoid using the following reserved keywords when building your models:

Beta
BetaRandom
Binomial
BinomialBetaRandom
BinomialRandom
CDFNormal
CalcOutputs
Chi2
Chi2Random
Data
DefaultSim
Density
Distrib
dt
Dynamics
End
Euler
ExpRandom
Experiment
Exponential
GGammaRandom
Gamma
GammaRandom
GetSeed
Gibbs
IFN
Initialize
Inputs
Integrate
InvGGammaRandom
InvGamma
Level
Likelihood
lnDFNormal
lnGamma
LogNormal
LogNormalRandom
LogNormal_v
LogUniform
LogUniformRandom
Lsodes
MCMC
MonteCarlo
NDoses
Normal
NormalRandom
Normal_v
OptimalDesign
OutputFile
Outputs
PerDose
PerExp
Piecewise
PiecewiseRandom
Poisson
PoissonRandom
Prediction
Print
PrintStep
Scale
SetPoints
SetSeed
SimType
Simulation
Spikes
StartTime
States
t
TruncLogNormal
TruncLogNormalRandom
TruncLogNormal_v
TruncNormal
TruncNormalRandom
TruncNormal_v
Uniform
UniformRandom

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 sample 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
```

B.3 'perc.model': A sample model description file

```
#-----------------------------------------------------------
# perc.model
# A four compartment model of Tetrachloroethylene (PERC)
# and total metabolites.
# Copyright (c) 1993.  Don Maszle, Frederic Bois.  All rights reserved.
#-----------------------------------------------------------
# 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, will be 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
```

B.4 'perc.lsodes.in'

```
#-------------------------------------------------------
# perc.lsodes.in
#
# Copyright (c) 1993.  Don Maszle, Frederic Bois.  All rights reserved.
#
#-------------------------------------------------------

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

G

H

I

K

L

M

N

O

P

Q

R

S

X

T

U

V

W

Table of Contents