Risø–R–1288(EN)

RISØ

User and Programmers Guide to the Neutron Ray-Tracing Package McStas, Version 1.5

Per-Olof Åstrand, Kim Lefmann, Emmanuel Farhi and Kristian Nielsen

Risø National Laboratory, Roskilde, Denmark
October 2001

Abstract

The software package McStas is a tool for carrying out Monte Carlo ray-tracing simulations of neutron scattering instruments with high complexity and precision. The simulations can compute all aspects of the performance of instruments and can thus be used to optimize the use of existing equipment, design new instrumentation, and carry out virtual experiments. McStas is based on a unique design where an automatic compilation process translates high-level textual instrument descriptions into efficient ANSI-C code. This design makes it simple to set up typical simulations and also gives essentially unlimited freedom to handle more unusual cases.

This report constitutes the reference manual for McStas, and, together with the manual for the McStas components, it contains full documentation of all aspects of the program. It covers the various ways to compile and run simulations, a description of the meta-language used to define simulations, and some example simulations performed with the program.

This report documents McStas version 1.5, released October 1, 2001.

The authors are:

Per-Olof Åstrand
Materials Research Department, Risø National Laboratory, Roskilde, Denmark
email: `per-olof.aastrand@risoe.dk`

Kim Lefmann
Materials Research Department, Risø National Laboratory, Roskilde, Denmark
email: `kim.lefmann@risoe.dk`

Emmanuel Farhi
Institut Laue-Langevin, Grenoble, France
email: `farhi@ill.fr`

Kristian Nielsen
Materials Research Department, Risø National Laboratory, Roskilde, Denmark
Present address: Sifira A/S, Copenhagen, Denmark

ISBN 87–550–2929–9
ISBN 87–550–2930–2 (Internet)
ISSN 0106–2840
Pitney Bowes Management Services Denmark A/S · Risø National Laboratory · 2001

Contents

Preface and acknowledgements **5**

1 Introduction to McStas **6**
1.1 Background . . . 6
1.1.1 The goals of McStas . . . 6
1.2 The design of McStas . . . 7
1.3 Overview . . . 9

2 Running McStas **10**
2.1 Brief introduction to the graphical user interface . . . 10
2.2 Obtaining McStas . . . 11
2.2.1 New releases of McStas . . . 11
2.3 Compiling McStas from source . . . 12
2.4 Running the instrument compiler . . . 13
2.4.1 Code generation options . . . 14
2.4.2 Specifying the location of files . . . 14
2.4.3 Embedding the generated simulations in other programs . . . 15
2.4.4 Running the C compiler . . . 15
2.5 Running the simulations . . . 16
2.6 Using simulation front-ends . . . 17
2.6.1 The graphical user interface . . . 17
2.6.2 Running simulations with automatic compilation . . . 20
2.6.3 The `gscan` front-end . . . 22
2.6.4 Graphical display of simulations . . . 22
2.6.5 Plotting the results of a simulation . . . 23
2.6.6 Plotting resolution functions . . . 23
2.7 Analyzing and visualizing the simulation results . . . 24

3 The McStas kernel and meta-language **25**
3.1 Notational conventions . . . 25
3.2 Syntactical conventions . . . 26
3.3 Writing instrument definitions . . . 27
3.3.1 The instrument definition head . . . 27
3.3.2 The `DECLARE` section . . . 28
3.3.3 The `INITIALIZE` section . . . 28
3.3.4 The `TRACE` section . . . 28

3.3.5 The FINALLY section . . . 29
3.3.6 The end of the instrument definition . . . 29
3.3.7 Code for the instrument vanadium_example.instr . . . 29
3.4 Writing component definitions . . . 30
3.4.1 The component definition header . . . 30
3.4.2 The DECLARE section . . . 32
3.4.3 The INITIALIZE section . . . 33
3.4.4 The TRACE section . . . 33
3.4.5 The FINALLY section . . . 33
3.4.6 The MCDISPLAY section . . . 35
3.4.7 The end of the component definition . . . 36
3.4.8 A component example: Slit . . . 36
3.4.9 McDoc, the McStas component documentation tool . . . 37

4 Monte Carlo Techniques and simulation strategy 39
4.1 The neutron weight, p . . . 39
4.1.1 Statistical errors of non-integer counts . . . 40
4.2 Weight factor transformations during a Monte Carlo choice . . . 41
4.2.1 Focusing components . . . 41
4.3 Transformation of random numbers . . . 42

5 The component library 44

6 Instrument examples 45
6.1 A test instrument for the component V_sample . . . 45
6.1.1 Scattering from the V-sample test instrument . . . 45
6.2 The triple axis spectrometer TAS1 . . . 45
6.2.1 Simulated and measured resolution of TAS1 . . . 47
6.3 The time-of-flight spectrometer PRISMA . . . 49
6.3.1 Simple spectra from the PRISMA instrument . . . 51

A Kernel calls and conversion constants 53
A.1 Kernel calls and functions . . . 53
A.1.1 Neutron propagation . . . 53
A.1.2 Meta-language extensions . . . 53
A.1.3 Mathematical routines . . . 54
A.1.4 Output from detectors . . . 54
A.1.5 Ray-geometry intersections . . . 55
A.1.6 Random numbers . . . 55
A.2 Constants for unit conversion etc. . . . 55

B The McStas terminology 57

Bibliography 58

Preface and acknowledgements

This document contains information on the Monte Carlo neutron ray-tracing program McStas version 1.5, an update to the initial release in October 1998 of version 1.0 as presented in Ref. [1]. The reader of this document is supposed to have some knowledge of neutron scattering, whereas only little knowledge about simulation techniques is required. In a few places, we also assume familiarity with the use of C, UNIX and of the world wide web (WWW).

It is a pleasure to thank Prof. Kurt N. Clausen for his continuous support to this project and for having initiated the work in the first place. We have also benefited from discussions with many other people in the neutron scattering community, too numerous to mention here.

This project has been supported by the European Union through the XENNI program and the RTD "Cool Neutrons" and "SCANS" programs.

In case of any errors, questions, suggestions, contact the authors at `mcstas@risoe.dk` or consult the McStas WWW home page [2].

New features in McStas version 1.5 as compared to version 1.4 include

- A three-component representaton, (s_x, s_y, s_z), of the neutron spin has been added, which includes the possibility for simulations of polarisation instruments.
- A signal handling system has been added, which gives the possibility for interacting with a running simulation
- The list of available kernel calls have been extended. They are normally used by users coding new components. The new kernel calls include the name, position and orientation of components.
- The McStas manual is separated into two documents: this manual and a new manual containing component descriptions. The first McStas component manual will be released in October/November 2001.
- The release contains many new components and updates of existing components. They will be documented in the new McStas component manual, but information on all McStas components may be found at the McStas web-page [2].

Chapter 1

Introduction to McStas

Efficient design and optimization of neutron spectrometers are formidable challenges. Monte Carlo techniques are well matched to meet these challenges. When McStas version 1.0 was released in October 1998, no existing package offered a general framework for the neutron scattering community to tackle the problems currently faced at reactor and spallation sources. The McStas project was designed to provide such a framework.

McStas is a fast and versatile software tool for neutron ray-tracing simulations. It is based on a meta-language specially designed for neutron simulation. Specifications are written in this language by users and automatically translated into efficient simulation codes in ANSI-C. The present version supports both continuous and pulsed source instruments as well as polarised neutrons, and includes a library of standard components with in total around 60 components.

The McStas package is written in ANSI-C and is freely available for down-load from the McStas web-page [2]. The package is actively being developed and supported by Risø National Laboratory and ILL. The system is well tested and is supplied with several examples and with an extensive documentation.

1.1 Background

The McStas project is the main part of a major effort in Monte Carlo simulations for neutron scattering at Risø National Laboratory. Simulation tools were urgently needed, not only to better utilize existing instruments (*e.g.* RITA-1 and RITA-2 [3–5]), but also to plan completely new instruments for new sources (*e.g.* European Spallation Source, ESS [6]). Writing programs in C or Fortran for each of the different cases involves a huge effort, with debugging presenting particularly difficult problems. A higher level tool specially designed for the needs of simulating neutron instruments is needed. As there was no existing simulation software that would fulfill our needs, the McStas project was initiated.

1.1.1 The goals of McStas

The McStas project had four main objectives that determined the design of the McStas software.

Correctness. It is essential to minimize the potential for bugs in computer simulations. If a word processing program contains bugs, it will produce bad-looking output or may even crash. This is a nuisance, but at least you know that something is wrong. However, if a simulation contains bugs it produces wrong results, and unless the results are far off, you may not know about it! Complex simulations involve hundreds or even thousands of lines of formulae, and "to err is human". Thus the system should be designed from the start to help minimize the potential for bugs to be introduced in the first place, and provide good tools for testing to maximize the chances of finding existing bugs.

Flexibility. When you commit yourself to using a tool for an important project, you need to know if the tool will satisfy not only your present, but also your future requirements. The tool must not have fundamental limitations that restrict its potential usage. Thus the McStas systems needs to be flexible enough to simulate different kinds of instruments (triple-axis, time-of-flight and possible hybrids) as well as many different kind of optical components, and it must also be extensible so that future, as yet unforeseen, needs can be satisfied.

Power. "*Simple things should be simple; complex things should be possible*". New ideas should be easy to try out, and the time from thought to action should be as short as possible. If you are faced with the prospect of programming for two weeks before getting any results on a new idea, you will most likely drop it. Ideally, if you have a good idea at lunch time, the simulation should be running in the afternoon.

Efficiency. Monte Carlo simulations are computationally intensive, hardware capacities are finite (albeit impressive), and humans are impatient. Thus the system must assist in producing simulations that run as fast as possible, without placing unreasonable burdens on the user in order to achieve this.

1.2 The design of McStas

In order to meet these ambitious goals, it was decided that McStas should be based on its own meta-language, specially designed for simulating neutron scattering instruments. Simulations are written in this meta-language by the user, and the McStas compiler automatically translates them into efficient simulation programs written in ANSI-C.

In realizing the design of McStas, the task was separated into four conceptual layers:

1. Modeling the physical processes of neutron scattering, *i.e.* the calculation of the fate of a neutron that passes through the individual components of the instrument (absorption, scattering at a particular angle, etc.)

2. Modeling of the overall instrument geometry, mainly consisting of the type and position of the individual components.

3. Accurate calculation, using Monte Carlo techniques, of instrument properties such as resolution function from the result of ray-tracing of a large number of neutrons. This includes estimating the accuracy of the calculation.

4. Presentation of the calculations, graphical or otherwise.

Though obviously interrelated, these four layers can be treated independently, and this is reflected in the overall system architecture of McStas. The user will in many situations be interested in knowing the details only in some of the layers. For example, one user may merely look at some results prepared by others, without worrying about the details of the calculation. Another user may simulate a new instrument without having to reinvent the code for simulating the individual components in the instrument. A third user may write an intricate simulation of a complex analyser such as the one of the RITA spectrometers, and expect other users to easily benefit from his/her work, and so on. McStas attempts to make it possible to work at any combination of layers in isolation by separating the layers as much as possible in the design of the system and in the meta-language in which simulations are written.

The usage of a special meta-language and an automatic compiler has several advantages over writing a big monolithic program or a set of library functions in C, Fortran, or another general-purpose programming language. The meta-language is more *powerful*; specifications are much simpler to write and easier to read when the syntax of the specification language reflects the problem domain. For example, the geometry of instruments would be much more complex if it were specified in C code with static arrays and pointers. The compiler can also take care of the low-level details of interfacing the various parts of the specification with the underlying C implementation language and each other. This way, users do not need to know about McStas internals to write new component or instrument definitions, and even if those internals change in later versions of McStas, existing definitions can be used without modification.

The McStas system also utilizes the meta-language to let the McStas compiler generate as much code as possible automatically, letting the compiler handle some of the things that would otherwise be the task of the user/programmer. *Correctness* is improved by having a well-tested compiler generate code that would otherwise need to be specially written and debugged by the user for every instrument or component. *Efficiency* is also improved by letting the compiler optimize the generated code in ways that would be time-consuming or difficult for humans to do. Furthermore, the compiler can generate several different simulations from the same specification, for example to optimize the simulations in different ways, to generate a simulation that graphically displays neutron trajectories, and possibly other things in the future that were not even considered when the original instrument specification was written.

The design of McStas makes it well suited for doing "what if..." types of simulations. Once an instrument has been defined, questions such as "what if a slit was inserted", "what if a focusing monochromator was used instead of a flat one", "what if the sample was offset 2 mm from the center of the axis" and so on are easy to answer. Within minutes the instrument definition can be modified and a new simulation program generated. It also makes it simple to debug new components. A test instrument definition may be written containing a neutron source, the component to be tested, and whatever detectors are useful, and the component can be thoroughly tested before being used in a complex simulation with many different components.

The McStas system is based on ANSI-C, making it both efficient and portable. The meta-language allows the user to embed arbitrary C code in the specifications. *Flexibility*

is thus ensured since the full power of the C language is available if needed.

1.3 Overview

The McStas system consists of the following major parts:

- A brief introduction to using McStas is given in Section 2.1.
- How to obtain the McStas software is described in Section 2.2.
- Section 2.3 explains how to compile and install the McStas compiler and associated files, while section 2.4 explains how to run the compiler to produce simulations. Section 2.5 explains how to run the generated simulations.
- The McStas meta-language is described in chapter 3. This chapter also describes a library of kernel functions and definitions that aid in the writing of simulations.
- The McStas component library contains a collection of well-tested beam components that can be used in simulations. The McStas component library is documented in a separate manual and on the McStas web-page [2].
- A collection of example instrument definitions is described in chapter 6.
- A number of front-end programs are used to run the simulations and to aid in the data collection and analysis of the results. These user interfaces are described in section 2.6.

An explanation of McStas terminology can be found in appendix B, and a list of library calls that may be used in component definitions appears in appendix A. Plans for future extensions are presented on the McStas web-page [2].

Chapter 2

Running McStas

This chapter describes the installation and usage of the McStas software. The software should compile without problems on most Unix-like systems with an ANSI-compliant C compiler. It has also been successfully compiled on Windows and MacIntosh systems, but these platforms are not actively supported. In case of problems, the McStas mailing list [2] or the authors should be contacted.

To use McStas, an instrument definition file is written which describes the instrument to be simulated (or an instrument file is obtained from the `examples/` directory in the distribution or from another source). The structure of McStas may be illustrated by Figure 2.1. The input files (instrument and component files) are written in the McStas meta-language and are edited either by a standard editor or by using a graphical user interface. These files are then compiled with the McStas compiler using the FLEX and Bison facilities to produce a C program. The C program can then be compiled with a C compiler and run in combination with various front-end programs to for example present the intensity at the detector as a motor position is varied. The output data may be analyzed in the same way as regular experiments are analyzed by using Matlab or IDL or by using the Perl routines included in McStas.

2.1 Brief introduction to the graphical user interface

This section gives an ultra-brief overview of how to use McStas once it has been properly installed. It is intended for those who do not read manuals if they can avoid it. For details on the different steps, see the following sections. This section uses the `vanadium_example.instr` file supplied in the `examples/` directory of the McStas distribution.

To start the graphical user interface of McStas, run the command `mcgui`. This will open a window with some menus etc., see figure 2.2.

To load an instrument, select "Open instrument" from the "File" menu. Open the file `vanadium_example.instr` in the McStas distribution. Select "Run simulation" from the "Simulation" menu. McStas will translate the definition into an executable program and pop up a dialog window. Type a value for the "ROT" parameter (*e.g.* 90), check the "Plot results" option, and select "Start". The simulation will run, and when it finishes after a while the results will be plotted in a window.

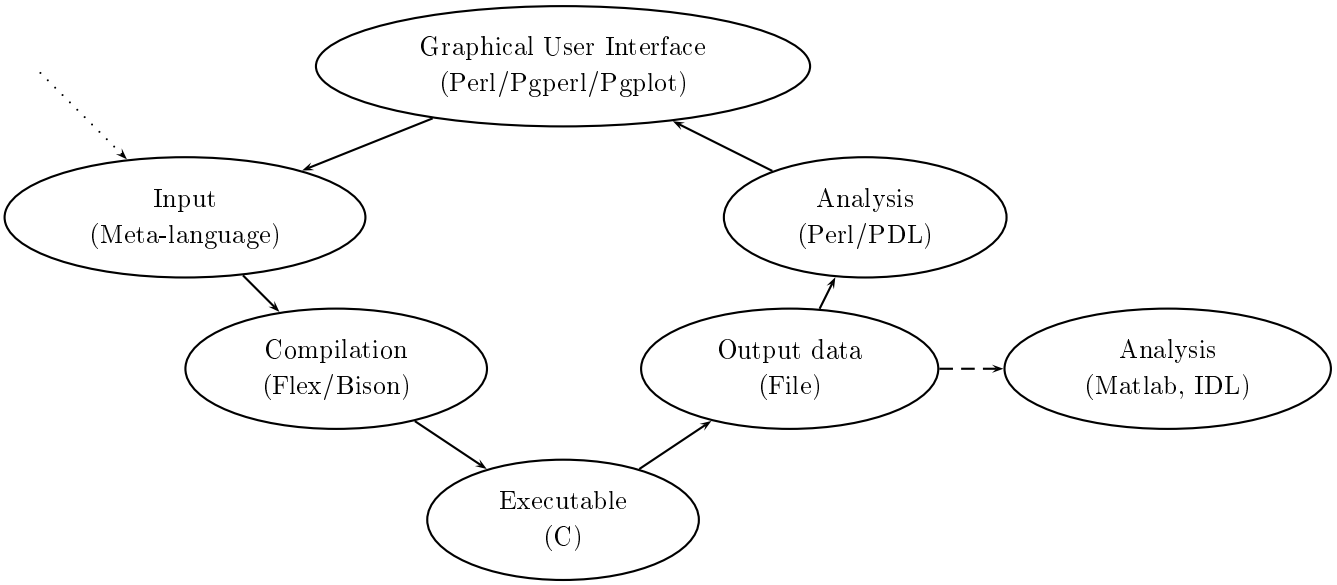

Figure 2.1: An illustration of the structure of McStas.

To debug the simulation graphically, repeat the steps but check the "Trace" option instead of the "Simulate" option. A window will pop up showing a sketch of the instrument. The left mouse button starts a new neutron, the middle button zooms, and the right button resets the zoom. The Q key quits the program.

For a slightly longer gentle introduction to McStas, see the McStas tutorial (available from [2]).

2.2 Obtaining McStas

The source code for McStas may be downloaded from the McStas web-page [2], and should be available in a file named `mcstas-1.5.tar.gz`.

The conditions on the use of McStas can be read in the files `LICENSE` and `LICENSE.LIB` in the distribution. Essentially, McStas may be used and modified freely, but copies of the McStas source code may not be distributed to others. New or modified component and instrument files may, however, be shared by the user community.

2.2.1 New releases of McStas

Releases of new versions of a software can today be carried out more or less continuously. However, users do not update their software on a daily basis, and as a compromise we have adopted the following policy of McStas.

- A version 1.5.x will contain bug fixes and new functionality. A new manual will, however, not be released and the modifications are documented on the McStas web-page. The extensions of the forthcoming version 1.5.x are also listed on the web,

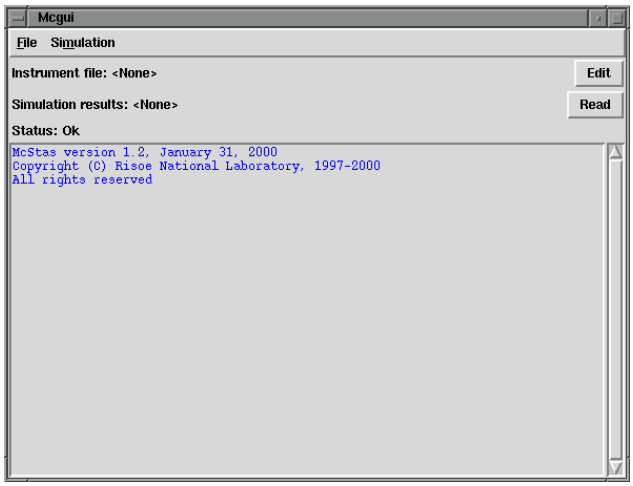

Figure 2.2: The graphical user interface `mcgui`.

and new versions may be released quite frequently when it is requested by the user community.

- A version 1.6 will contain an updated manual. It will typically be released once or twice a year in connection to for example a McStas workshop.
- A version 2.0 will hopefully never be released. It would mean that the code has been rewritten and no backward compability can be expected.

2.3 Compiling McStas from source

Compilation and installation of McStas contain three steps. First, the source files must be unpacked:

```
gunzip -c mcstas-1.5.tar.gz | tar xf -
cd mcstas/
```

Next, the `configure` script is run to configure McStas for the particular machine and operating system, and the software is compiled:

```
./configure
make
```

Finally, McStas is installed:

```
make install
```

By default, McStas will be installed in the `usr/local/` directory (which typically requires superuser privileges). To install in another directory, the `--prefix` option of the `configure` script can be used. For example, to install in `/home/joe` instead:

```
./configure --prefix=/home/joe
make
make install
```

Depending on which directory McStas is installed in, it may be necessary to add the `bin/` subdirectory of the installation directory to the default path, or to run McStas with the full pathname of the program (`/usr/local/bin/mcstas` by default).

The `configure` command will guess some reasonable defaults for the C compiler to use. These will be used to compile McStas itself as well as the simulations produced by McStas. To override[1] the defaults, the environment variables `CC` and `CFLAGS` can be set to the file name of the compiler to use and any special compiler options needed (for example to enable optimization), respectively. After installation, each user may override the settings using the environment variables `MCSTAS_CC` and `MCSTAS_CFLAGS`.

McStas has been tested on x86 Linux, Digital Unix, and HPUX. It should run on most other Unix-like systems without problems. The main thing to ensure is that an ANSI-C compliant compiler is available (GCC works well). If any difficulties arise, the authors should be contacted so that the problems may be fixed in a later release of McStas.

To use the McStas front-end programs (see section 2.6), certain auxiliary packages must be installed, as described in the README file in the distribution. These packages are all freely available, and some of them may be supplied with the operating system (for example, all required packages are included in Debian/GNU Linux [7]).

Note that the core parts of McStas, including the McStas compiler and any generated simulations, work with only an ANSI-C compiler.

2.4 Running the instrument compiler

This section describes how to run the McStas compiler manually. Often, it will be more convenient to use the front-end program `mcgui` (section 2.6.1) or `mcrun` (section 2.6.2). These front-ends will compile and run the simulations automatically.

The compiler for the McStas instrument definition is invoked by typing a command of the form

```
mcstas name.instr
```

This will read the instrument definition `name.instr` which is written in the McStas meta-language. The compiler will translate the instrument definition into a Monte Carlo simulation program provided in ANSI-C. The output is by default written to a file in the current directory with the same name as the instrument file, but with extension `.c` rather than `.instr`. This can be overridden using the `-o` option as follows:

```
mcstas -o code.c name.instr
```

which gives the output in the file `code.c`. A single dash '`-`' may be used for both input and output filename to represent standard input and standard output, respectively.

[1] It may be necessary to remove the file `config.cache` before re-installing McStas to have the new settings take effect

2.4.1 Code generation options

By default, the output files from the McStas compiler are in ANSI-C with some extensions (currently the only extension is the creation of new directories, which is not possible in pure ANSI-C). The use of extensions may be disabled with the `-p` or `--portable` option. With this option, the output is strictly ANSI-C compliant, at the cost of some slight reduction in capabilities.

The `-t` or `--trace` option puts special "trace" code in the output. This code makes it possible to get a complete trace of the path of every neutron through the instrument, as well as the position and orientation of every component. This option is mainly used with the `mcdisplay` front-end as described in section 2.6.4.

The code generation options can also be controlled by using preprocessor macros in the C compiler, without the need to re-run the McStas compiler. If the preprocessor macro `MC_PORTABLE` is defined, the same result is obtained as with the `--portable` option of the McStas compiler. The effect of the `--trace` option may be obtained by defining the `MC_TRACE_ENABLED` macro. Most Unix-like C compilers allow preprocessor macros to be defined using the `-D` option, eg.

```
cc -DMC_TRACE_ENABLED -DMC_PORTABLE ...
```

2.4.2 Specifying the location of files

The McStas compiler needs to be able to find various files during compilation, some explicitly requested by the user (such as component definitions and files referenced by `%include`), and some used internally to generate the simulation executable. McStas looks for these files in three places: first in the current directory, then in a list of directories given by the user, and finally in a special McStas directory. Usually, the user will not need to worry about this as McStas will automatically find the required files. But if users build their own component library in a separate directory or if McStas is installed in an unusual way, it will be necessary to tell the compiler where to look for the files.

The location of the special McStas directory is set when McStas is compiled. It defaults to `/usr/local/lib/mcstas`, but it can be changed to something else, see section 2.3 for details. The location can be overridden by setting the environment variable `MCSTAS`:

```
setenv MCSTAS /home/joe/mcstas
```

for csh/tcsh users, or

```
export MCSTAS=/home/joe/mcstas
```

for bash/Bourne shell users.

To make McStas search additional directories for component definitions and include files, use the `-I` switch for the McStas compiler:

```
mcstas -I/home/joe/components -I/home/joe/neutron/include name.instr
```

Multiple `-I` options can be given, as shown.

2.4.3 Embedding the generated simulations in other programs

By default, McStas will generate a stand-alone C program, which is what is needed in most cases. However, for advanced usage, such as embedding the generated simulation in another program or even including two or more simulations in the same program, a stand-alone program is not appropriate. For such usage, the McStas compiler provides the following options:

- `--no-main` This option makes McStas omit the `main()` function in the generated simulation program. The user must then arrange for the function `mcstas_main()` to be called in some way.

- `--no-runtime` Normally, the generated simulation program contains all the run-time C code necessary for declaring functions, variables, etc. used during the simulation. This option makes McStas omit the run-time code from the generated simulation program, and the user must then explicitly link with the file `mcstas-r.c` from the McStas distribution.

Users that need these options are encouraged to contact the authors for further help.

2.4.4 Running the C compiler

After the source code for the simulation program has been generated with the McStas compiler, it must be compiled with the C compiler to produce an executable. The generated C code obeys the ANSI-C standard, so it should be easy to compile it using any ANSI-C (or C++) compiler. *E.g.* a typical Unix-style command would be

```
cc -O -o name.out name.c -lm
```

The `-O` option typically enables the optimization phase of the compiler, which can make quite a difference in speed of McStas generated simulations. The `-o name.out` sets the name of the generated executable. The `-lm` options is needed on many systems to link in the math runtime library (like the cos() and sin() functions).

Monte Carlo simulations are computationally intensive, and it is often desirable to have them run as fast as possible. Some success can be obtained by adjusting the compiler optimization options. Here are some example platform and compiler combinations that have been found to perform well (up-to-date information will be available on the McStas WWW home page [2]):

- Intel x86 ("PC") with Linux and GCC, using options `gcc -O3`.

- Intel x86 with Linux and EGCS (GCC derivate) using options `egcc -O6`.

- Intel x86 with Linux and PGCC (pentium-optimized GCC derivate), using options `gcc -O6 -mstack-align-double`.

- HPPA machines running HPUX with the optional ANSI-C compiler, using the options `-Aa +Oall -Wl,-a,archive` (the `-Aa` option is necessary to enable the ANSI-C standard).

A warning is in place here: it is tempting to spend far more time fiddling with compiler options and benchmarking than is actually saved in computation times. Even worse, compiler optimizations are notoriously buggy; the options given above for PGCC on Linux and the ANSI-C compiler for HPUX have been known to generate *incorrect code* in some compiler versions. McStas actually puts an effort into making the task of the C compiler easier, by in-lining code and using variables in an efficient way. As a result, McStas simulations generally run quite fast, often fast enough that further optimizations are not worthwhile.

2.5 Running the simulations

Once the simulation program has been generated by the McStas compiler and an executable has been obtained with the C compiler, the simulation can be run in various ways. The simplest way is to run it directly from the command line or shell:

```
./name.out
```

Note the leading dot, which is needed if the current directory is not in the path searched by the shell. When used in this way, the simulation will prompt for the values of any instrument parameters such as motor positions, and then run the simulation. This way of running McStas will only give data for one spectrometer setting which is normally sufficient *e.g.* for a time-of-flight spectrometer, but not for a triple-axis spectrometer where a scan over various spectrometer settings is required. Often the simulation will be run using one of several available front-ends, as described in the next section. These front-ends help manage output from the potentially many detectors in the instruments, as well as running the simulation for each data point in a scan.

The generated simulations accept a number of options and arguments. The full list can be obtained using the `--help` option:

```
./name.out --help
```

The values of instrument parameters may be specified as arguments using the syntax *name*=*val*. For example

```
./vanadium_example.out ROT=90
```

The number of neutron histories to simulate may be set using the `--ncount` or `-n` option, for example `--ncount=2e5`. The initial seed for the random number generator is by default chosen based on the current time so that it is different for each run. However, for debugging purposes it is sometimes convenient to use the same seed for several runs, so that the same sequence of random numbers is used each time. To achieve this, the random seed may be set using the `--seed` or `-s` option.

By default, McStas simulations write their results into several data files in the current directory, overwriting any previous files stored there. The `--dir=`*dir* or `-d`*dir* option causes the files to be placed instead in a newly created directory *dir* (to prevent overwriting previous results an error message is given if the directory already exists). Alternatively, all output may be written to a single file *file* using the `--file=`*file* or `-f`*file* option.

`-s` *seed* `--seed=`*seed*	Set the initial seed for the random number generator. This may be useful for testing to make each run use the same random number sequence.
`-n` *count* `--ncount=`*count*	Set the number of neutron histories to simulate. The default is 1,000,000.
`-d` *dir* `--dir=`*dir*	Create a new directory *dir* and put all data files in that directory.
`-f` *file* `--file=`*file*	Write all data into a single file *file*
`-a` `--ascii-only`	Do not put any headers in the data files.
`-h` `--help`	Show a short help message with the options accepted, including the names of the parameters of the instrument.
`-i` `--info`	Show extensive information on the simulation and the instrument definition it was generated from.
`-t` `--trace`	This option makes the simulation output the state of every neutron as it passes through every component. Requires that the `-t` (or `--trace`) option is also given to the McStas compiler when the simulation is generated.
`--no-output-files`	This option disables the writing of data files (output to the terminal, such as detector intensities, will still be written).
param=*value*	Set the value of an instrument parameter, rather than having to prompt for each one.

Table 2.1: Options accepted by McStas simulations

By default, data files contain header lines with information about the simulation from which they originate. In case the data must be analyzed with programs that cannot read files with such headers, they may be turned off using the `--ascii-only` or `-a` option.

The format of the output files from McStas simulations is described in more detail in section 2.7. The complete list of options and arguments accepted by McStas simulations appears in table 2.1.

2.6 Using simulation front-ends

McStas includes a number of front-end programs that extend the functionality of the simulations. The front-end programs is an interface between the user and the simulations, running the simulations and presenting the output in various ways to the user. An extended set of front-end programs is planned for future versions of McStas, including a NeXus data format option [8].

2.6.1 The graphical user interface

The front-end `mcgui` provides a graphical user interface that interfaces the various parts of the McStas package. It is started using simply the command

```
mcgui
```

The mcgui program may optionally be given the name of an instrument file.

When the front-end is started, a main window is opened. This window displays the output from compiling and running simulations, and also contains a few menus and buttons. The main purpose of the front-end is to edit and compile instrument definitions, run the simulations, and visualize the results.

The menus

The "File" menu has the following features:

Open instrument selects the name of an instrument file to be used.

Edit current opens a simple editor window for editing the current instrument definition. This function is also available from the "Edit" button to the right of the name of the instrument definition in the main window.

Compile instrument forces a recompile of the instrument definition, regardless of file dates. This is for example useful to pick up changes in component definitions, which the front-end will not notice automatically. See section 2.3 for how to override default C compiler options.

Clear output erases all text in the window showing output of compilations and simulations.

Quit exits the graphical user interface front-end.

The "Simulation" menu has the following features:

Read old simulation prompts for the name of a file from a previous run of a McStas simulation (usually called `mcstas.sim`). The file will be read and any detector data plotted using the `mcplot` front-end. The parameters used in the simulation will also be made the defaults for the next simulation run. This function is also available using the "Read" button to the right of the name of the current simulation data.

Run simulation opens the run dialog window, explained further below.

Plot results plots (using `mcplot`) the results of the last simulation run or loaded.

The "Help" menu has a single menu point, "McStas web-page", which attempts to open a Netscape window with the McStas web-page. This obviously requires that Netscape is properly installed on the computer.

The run dialog

The run dialog is used to run simulations. It allows the entry of instrument parameters as well as the specifications of options for running the simulation (see section 2.5 for details). It also allows to run the `mcdisplay` (section 2.6.4) and `mcplot` (section 2.6.5) front-ends together with the simulation.

The meaning of the different fields is as follows:

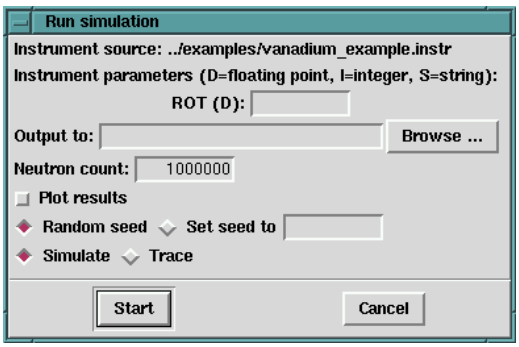

Figure 2.3: The run dialog in `mcgui`.

Instrument parameters allows the setting of the values for the input parameters of the instrument. The type of each instrument parameter is given in parenthesis after each name. Floating point numbers are denoted by (D) (for the C type "`double`"), (I) denotes integer parameters, and (S) denotes strings.

Output to allows the entry of a directory to store the resulting data files in (like the `--dir` option). If no name is given, the results are put in the current directory, to be overwritten by the next simulation.

Neutron count sets the number of neutron histories to simulate (the `--ncount` option).

Plot results – if checked, the `mcplot` front-end will be run after the simulation has finished, and the plot dialog will pop up (see below).

Random seed/Set seed to selects between using a random seed (different in each simulation) for the random number generator, or using a fixed seed (to reproduce results for debugging).

Simulate/Trace selects between running the simulation normally, or using the `mcdisplay` front-end.

Start runs the simulation.

Cancel aborts the dialog.

Before running the simulation, the instrument definition is automatically compiled if it is newer than the generated C file (or if the C file is newer than the executable). The executable is assumed to have a `.out` suffix in the filename.

The plot dialog

Monitors and detectors lists all the one- and two-dimensional detectors in the instrument. By double-clicking, one plots the data in the plot window.

Plot plots the selected detector in the plot window, just like double-clicking its name.

Overview plot plots all the detectors together in the plot window.

B&W postscript prompts for a file name and saves the current plot as a black and white postscript file. This can subsequently be printed on a postscript printer.

Colour postscript creates a colour postscript file of the current plot.

Close ends the dialog.

The editor window

The editor window provides a simple editor for creating and modifying instrument definitions. Apart from the usual editor functions, the "Insert" menu provides some functions that aid in the construction of the instrument definitions:

Instrument template inserts the text for a simple instrument skeleton in the editor window.

Component... opens up a dialog window with a list of all the components available for use in McStas. Selecting a component will display a description. Double-clicking will open up a dialog window allowing the entry of the values of all the parameters for the component (figure 2.4). See section 3.3 for details of the meaning of the different fields.

The dialog will also pick up those of the users own components that are present in the current directory when `mcgui` is started. See section 3.4.9 for how to write components to integrate well with this facility.

Type These menu entries give quick access to the entry dialog for the various components available.

To use the `mcgui` front-end, the programs Perl, Perl/Tk, PGPLOT, PgPerl, and PDL must all be properly installed on the system. It may be necessary to set the `PGPLOT_DIR` environment variable; consult the documentation for PGPLOT on the local system in case of difficulty.

2.6.2 Running simulations with automatic compilation

The `mcrun` front-end provides a convenient command-line interface for running simulations with the same automatic compilation features available in the `mcgui` front-end. It also provides a facility for running a series of simulations while varying an input parameter, thereby replacing the old `gscan` front-end.

The command

`mcrun` *sim args* ...

will compile the instrument definition *sim*.`instr` (if necessary) into an executable simulation *sim*.`out`. It will then run *sim*.`out`, passing the argument list *args*

The possible arguments are the same as those accepted by the simulations themselves as described in section 2.5, with the following extensions:

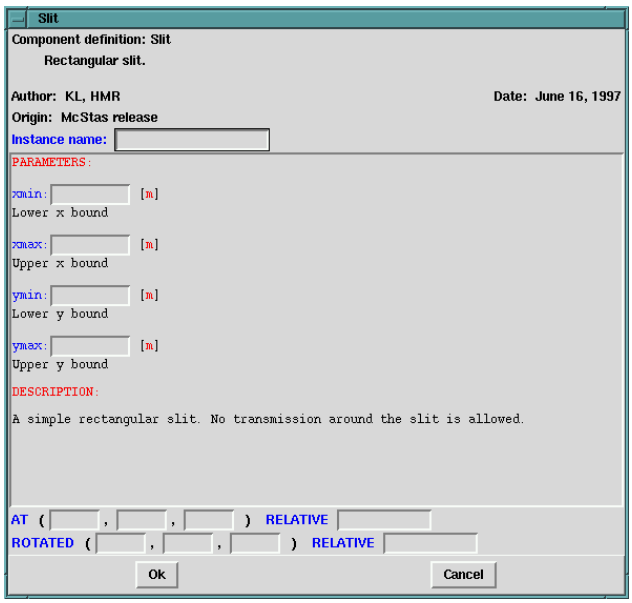

Figure 2.4: Component parameter entry dialog.

- The `-c` or `--force-compile` option may be used to for the recompilation of the instrument definition, regardless of file dates. This may be needed in case any component definitions are changed (in which case `mcrun` does not automatically recompile), or if a new version of McStas has been installed.

- The `-p` *file* or `--param=`*file* option may be used to specify a file containing assignment of values to the input parameters of the instrument definition. The file should consist of specifications of the form *name*`=`*value* separated by spaces or line breaks. Multiple `-p` options may be given together with direct parameter specifications on the command line. If a parameter is assigned multiple times, later assignments override previous ones.

- The `-N` *count* or `--numpoints=`*count* option may be used to perform a series of *count* simulations while varying one or more parameters within specified intervals. Such a series of simulations is called a *scan*. To specify an interval for a parameter X, it should be assigned two values separated with a comma. For example, the command

  ```
  mcrun sim.instr -N4 X=2,8 Y=1
  ```

  would run the simulation defined in `sim.instr` four times, with X having the values 2, 4, 6, and 8, respectively.

  After running the simulation, the results will be written to the file `mcstas.dat` by default. This file contains one line for each simulation run giving the values of the scanned input variables along with the intensity and estimated error in all detectors. Additionally, a file `mcstas.sim` is written that can be read by the `mcplot` front-end to plot the results on the screen or in a Postscript file, see section 2.6.5.

- When doing a scan, the `-f` *file* and `--file=`*file* options make `mcrun` write the output to the files *file*`.dat` and *file*`.sim` instead of the default names.
- When doing a scan, the `-d` *dir* and `--dir=`*dir* options make `mcrun` put all output in a newly created directory *dir*. Additionally, the directory will have subdirectories `1`, `2`, `3`,... containing all data files output from the different simulations. When the `-d` option is not used, no data files are written from the individual simulations (in order to save disk space).

The `mcrun` front-end requires a working installation of Perl to run.

2.6.3 The `gscan` front-end

The front-end `gscan` is obsolete from version 1.3 of McStas, and is included only for backwards compatibility. The front-end `mcrun` (section 2.6.2) includes all the functionality of the old `gscan` front-end and should be used instead.

2.6.4 Graphical display of simulations

The front-end `mcdisplay` is a graphical debugging tool. It presents a schematic drawing of the instrument definition, showing the position of the components and the paths of the simulated neutrons through the instrument. It is thus very useful for debugging a simulation, for example to spot components in the wrong position or to find out where neutrons are getting lost. The graphics is shown on an X Windows display.

To use the `mcdisplay` front-end with a simulation, run it as follows:

```
mcdisplay sim.out args ...
```

where `sim` is the name of the simulation program generated with McStas and *args ...* are the normal command line arguments for the simulation, as explained above. This will view the instrument from above. A multi-display that shows the instrument from three directions simultaneously can be shown using the `--multi` option:

```
mcdisplay --multi sim.out args ...
```

The `mcdisplay` front-end can also be run from the `mcgui` front-end.

Click the left mouse button in the graphics window or hit the space key to see the display of successive neutron trajectories. The 'P' key saves a postscript file containing the current display that can be sent to the printer to obtain a hardcopy; the 'C' key produces color postscript. To stop the simulation prematurely, type 'Q' or use control-C as normal in the window in which `mcdisplay` was started.

To see details in the instrument, it is possible to zoom in on a part of the instrument using the middle mouse button (or the 'Z' key on systems with a one- or two-button mouse). The right mouse button (or the 'X' key) resets the zoom. Note that after zooming, the units on the different axes may no longer be equal, and thus the angles as seen on the display may not match the actual angles.

Another way to see details while maintaining an overview of the instrument is to use the `--zoom=`*factor* option. This magnifies the display of each component along the selected axis only, *e.g.* a Soller collimator is magnified perpendicular to the neutron beam but

not along it. This option may produce rather strange visual effects as the neutron passes between components with different coordinate magnifications, but it is occasionally useful.

When debugging, it is often the case that one is interested only in neutrons that reach a particular component in the instrument. For example, if there is a problem with the sample one may prefer not to see the neutrons that are absorbed in the monochromator shielding. For these cases, the `--inspect=`*comp* option is useful. With this option, only neutrons that reach the component named *comp* are shown in the graphics display.

See section 3.4.6 for how to make new components work with the `mcdisplay` front-end. The `mcdisplay` front-end requires the Perl, the PGPLOT, and the PGPerl packages to be installed.

2.6.5 Plotting the results of a simulation

The front-end `mcplot` is a program that produces plots of all the detectors in a simulation, and it is thus useful to get a quick overview of the simulation results.

In the simplest case, the front-end is run simply by typing

```
mcplot
```

This will plot any simulation data stored in the current directory, which is where simulations put their results by default. If the `--dir` or `--file` options have been used (see section 2.5), the name of the file or directory should be passed to mcplot, *e.g.* "`mcplot` *dir*" or "`mcplot` *file*".

The initial display shows plots for each detector in the simulation. Clicking the left mouse button on a plot produces a full-window version of that plot. The 'P' key saves a postscript file containing the current plot that can be sent to the printer to obtain a hardcopy; the 'C' key produces color postscript. The 'Q' key quits the program (or CTRL-C in the controlling terminal may be used as normal).

To use the `mcplot` front-end, the programs Perl, PGPLOT, PgPerl, and PDL must all be properly installed on the system.

2.6.6 Plotting resolution functions

The `mcresplot` front-end is used to plot the resolution function of a triple-axis spectrometer, as calculated by the Res_sample component. This front-end has been included in the release since it may be useful despite its somewhat rough user interface.

The `mcresplot` front-end is run with the command

```
mcresplot file
```

Here, *file* is the name of a file output from a simulation using the Res_monitor component. The front-end will open two windows. One shows a three-dimensional visualization of the resolution function using the two components of $\boldsymbol{Q}$ in the scattering plane and ω. The plot may be rotated using the mouse while pressing the left button, and zoomed while pressing the right button.

The other window displays the covariance matrix of the resolution function and the resulting resolution matrix. This is mainly useful for triple-axis spectrometers. The four bottom plots visualize the covariance matrix using four different projections. The top left

corner shows histograms of the resolution function along the three axes of $\boldsymbol{Q}$ and along the ω axis.

Pressing the "Q" key while the three-dimensional window is active switches to a combined plot where the yellow dots show the resolution function and the red dots show the covariance matrix. A second press of the "Q" key ends the front-end program.

To use the `mcresplot` front-end, the programs Perl, PGPLOT, PgPerl, and PDL must all be properly installed on the system.

2.7 Analyzing and visualizing the simulation results

To analyze simulation results, one uses the same tools as for analyzing experimental data, *i.e.* programs such as the MATLAB packages Mview and Mfit used at Risø. The output files from simulations are simply columns of ASCII text that most programs should be able to read. A future version of McStas will support output in the NeXus format [8].

One-dimensional histogram detectors (time-of-flight, energy-sensitive) write one line for each histogram bin. Each line contains a number identifying the bin (*i.e.* the time-of-flight) followed by three numbers: the simulated intensity, an estimate of the statistical error as explained in section 4.1.1, and the number of neutron events for this bin.

Two-dimensional histogram detectors (position sensitive detectors) output M lines of N numbers representing neutron intensities, where M and N are the number of bins in the two dimensions. The two-dimentional detectors do not store any error estimates since this is seldom useful, however if needed it can be obtained using `MC_GETPAR` in the `FINALLY` section of the instrument definition, see section 3.4.2.

Single-point detectors output the neutron intensity, the estimated error, and the neutron event count as numbers on the terminal. (The results from a series of simulations may be combined in a data file using the `mcrun` front-end as explained in section 2.6.2).

Both one- and two-dimentional detector output by default start with a header of comment lines, all beginning with the '`#`' character. This header gives such information as the name of the instrument used in the simulation, the values of any instrument parameters, the name of the detector component for this data file, *etc.* The headers may be disabled using the `--ascii-only` option in case the file must be read by a program that cannot handle the headers.

In addition to the files written for each one- and two-dimensional detector component, another file (by default named `mcstas.sim`) is also created. This file is in a special McStas ASCII format. It contains all available information about the instrument definition used for the simulation, the parameters and options used to run the simulation, and the detector components present in the instrument. It is read by the `mcplot` front-end (see section 2.6.5). This file stores the results from single detectors, but by default contains only pointers (in the form of file names) to data for one- and two-dimensional detectors. By storing data in separate files, reading the data with programs that do not know the special McStas file format is simplified. The `--file` option may be used to store all data inside the `mcstas.sim` file instead of in separate files.

Note that the neutron event counts in detectors is typically not very meaningful except as a way to measure the performance of the simulation. Use the simulated intensity instead whenever analysing simulation data.

Chapter 3

The McStas kernel and meta-language

Instrument definitions are written in a special McStas meta-language which is translated automatically by the McStas compiler into a C program which is in turn compiled to an executable that performs the simulation. The meta-language is custom-designed for neutron scattering and serves two main purposes: (i) to specify the interaction of a single neutron with a single optical component, and (ii) to build a simulation by constructing a complete instrument from individual components.

For maximum flexibility and efficiency, the meta-language is based on C. Instrument geometry, propagation of neutrons between the different components, parameters, data input/output etc. is handled in the meta-language and by the McStas compiler. Complex calculations are written in C embedded in the meta-language description of the components. It is possible to set up an instrument from existing components and run a simulation without writing a single line of C code, working entirely in the meta-language. On the other hand, the full power of the C language is available for special-purpose setups in advanced simulations, and for computing neutron trajectories in the components.

Apart from the meta-language, McStas also includes a number of C library functions and definitions that are useful for neutron ray-tracing simulations. The definitions available for users coding components are listed in appendix A. The list includes functions for computing the intersection between a flight-path and various objects (such as cylinders and spheres), functions for generating random numbers with various distributions, convenient conversion factors between relevant units, etc.

The McStas meta-language was designed to be readable, with a verbose syntax and explicit mentioning of otherwise implicit information. The recommended way to get started with the meta-language is to start by looking at the examples supplied with McStas, modifying them as necessary for the application at hand.

3.1 Notational conventions

Simulations generated by McStas use a semi-classical description of the neutron to compute the neutron trajectory through the instrument and its interaction with the different components.

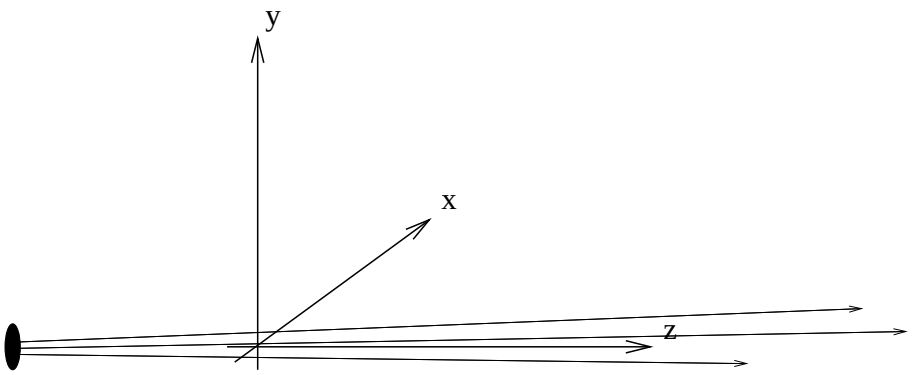

Figure 3.1: conventions for the orientations of the axis in simulations.

An instrument consists of a list of components through which the neutron passes one after the other. The order of components is thus significant since McStas does not automatically check which component is the next to interact with the neutron at a given point in the simulation.

The instrument is given a global, absolute coordinate system. In addition, every component in the instrument has its own local coordinate system that can be given any desired position and orientation (though the position and orientation must remain fixed for the duration of a single simulation). By convention, the z axis points in the direction of the beam, the x axis is perpendicular to the beam in the horizontal plane pointing left as seen from the source, and the y axis points upwards (see figure 3.1). Nothing in McStas enforces this convention, but if every component used different conventions the user would be faced with a severe headache! It is therefore recommended that this convention is followed by users implementing new components.

In the instrument definitions, units of length (*e.g.* component positions) are given in meters and units of angles (*e.g.* rotations) are given in degrees. The state of the neutron is given by its position (x, y, z) in meters, its velocity (v_x, v_y, v_z) in meters per second, the time t in seconds, and the two parameters s_1 and s_2 that are obsolete and cannot be used. A three-component representation of the spin, (s_x, s_y, s_z), normalized to one, is used. In addition, the outgoing neutron has an associated weight p which is used to model fractional neutrons in the Monte Carlo simulation. $p = 0.2$ means that a single neutron following this path has a 20% chance of reaching the present position without being absorbed or scattered away from the instrument. Alternatively, one may regard a ray of neutrons and p is the fraction of neutrons following the considered path.

3.2 Syntactical conventions

Comments follow the normal C syntax "`/* ... */`". C++ style comments "`// ...`" may also be used.

Keywords are not case-sensitive, for example "`DEFINE`", "`define`", and "`dEfInE`" are all equivalent. However, by convention we always write keywords in uppercase to distinguish them from identifiers and C language keywords. In contrast, McStas identifiers, like C

identifiers and keywords, *are* case sensitive, another good reason to use a consistent case convention for keywords.

It is possible, and usual, to split the input instrument definition across several different files. For example, if a component is not explicitly defined in the instrument, McStas will search for a file containing the component definition in the standard component library (as well as in the current directory and any user-specified search directories, see section 2.4.2). It is also possible to explicitly include another file using a line of the form

```
%include "file"
```

Beware of possible confusion with the C language "`#include`" statement, especially when it is used in C code embedded within the McStas meta-language. Files referenced with "`%include`" are read when the instrument is translated into C by the McStas compiler, and must contain valid McStas meta-language input. Files referenced with "`#include`" are read when the C compiler generates an executable from the generated C code, and must contain valid C.

Embedded C code is used in several instances in the McStas meta-language. Such code is copied by the McStas compiler into the generated simulation C program. Embedded C code is written by putting it between the special symbols `%{` and `%}`, as follows:

```
%{
        ... Embedded C code ...
%}
```

The "`%{`" and "`%}`" must appear on a line by themselves.

3.3 Writing instrument definitions

The purpose of the instrument definition is to specify a sequence of components, along with their position and parameters, which together make up an instrument. Each component is given its own local coordinate system, the position and orientation of which may be specified by its translation and rotation relative to another component. An example is given in section 3.3.7 and some additional examples of instrument definitions can be found on the McStas web-page [2] and in the `example` directory.

3.3.1 The instrument definition head

`DEFINE INSTRUMENT` *name* $(a_1, a_2, \ldots)$

This marks the beginning of the definition. It also gives the name of the instrument and the list of instrument parameters. Instrument parameters describe the configuration of the instrument, and usually correspond to setting parameters of the components. A motor position is a typical example of an instrument parameter. The input parameters of the instrument constitute the input that the user (or possibly a front-end program) must supply when the generated simulation is started.

By default, the parameters will be floating point numbers, and will have the C type `double` (double precision floating point). The type of each parameter may optionally be declared to be `int` for the C integer type or `char *` for the C string type. The name `string`

may be used as a synonym for `char *`, and floating point parameters may be explicitly declared using the name `double`. The following example illustrates all possibilities:

```
DEFINE INSTRUMENT tst(d1, double d2, int i, char *s1, string s2)
```

Here `d1` and `d2` will be floating point parameters of C type `double`, `i` will be an integer parameter of C type `int`, and `s1` and `s2` will be string parameters of C type `char *`.

3.3.2 The `DECLARE` section

```
DECLARE
%{
      ... C declarations of global variables etc. ...
%}
```

This gives C declarations that may be referred to in the rest of the instrument definition. A typical use is to declare global variables or small functions that are used elsewhere in the instrument. This section is optional.

3.3.3 The `INITIALIZE` section

```
INITIALIZE
%{
      ... C initializations. ...
%}
```

This gives code that is executed when the simulation starts. This section is optional.

3.3.4 The `TRACE` section

The `TRACE` keyword starts a section giving the list of components that constitute the instrument. Components are declared like this:

`COMPONENT` $name = comp(p_1 = e_1, p_2 = e_2, \ldots)$

This declares a component named *name* that is an instance of the component definition named *comp*. The parameter list gives the setting and definition parameters for the component. The expressions $e_1, e_2, \ldots$ define the values of the parameters. For setting parameters arbitrary ANSI-C expressions may be used, while for definition parameters only constant numbers, strings, names of instrument parameters, or names of C identifiers are allowed (see section 3.4.1 for details of the difference between definition and setting parameters). To assign the value of a general expression to a definition parameter, it is necessary to declare a variable in the `DECLARE` section, assign the value to the variable in the `INITIALIZE` section, and use the variable as the value for the parameter.

The McStas program takes care to rename parameters appropriately in the output so that no conflicts occur between different component definitions or between component and instrument definitions. It is thus possible (and usual) to use a component definition multiple times in an instrument description.

The McStas compiler will automatically search for a file containing a definition of the component if it has not been declared previously. The definition is searched for in a file called "*name*`.comp`", "*name*`.cmp`", or "*name*`.com`". See section 2.4.2 for details on which directories are searched. This facility is often used to refer to existing component definitions in standard component libraries. It is also possible to write component definitions in the main file before the instrument definitions, or to explicitly read definitions from other files using `%include`.

The position of a component is specified using an `AT` modifier following the component declaration:

`AT` (x, y, z) `RELATIVE` *name*

This places the component at position (x, y, z) in the coordinate system of the previously declared component *name*. Placement may also be absolute (not relative to any component) by writing

`AT` (x, y, z) `ABSOLUTE`

Any C expression may be used for x, y, and z. The `AT` modifier is required. Rotation is achieved similarly by writing

`ROTATED` (ϕ_x, ϕ_y, ϕ_z) `RELATIVE` *name*

This will result in a coordinate system that is rotated first the angle ϕ_x (in degrees) around the x axis, then ϕ_y around the y axis, and finally ϕ_z around the z axis. Rotation may also be specified using `ABSOLUTE` rather than `RELATIVE`. If no rotation is specified, the default is $(0, 0, 0)$ using the same relative or absolute specification used in the `AT` modifier.

3.3.5 The `FINALLY` section

```
FINALLY
%{
        ... C code to execute at end of simulation ...
%}
```

This gives code that will be executed when the simulation has ended.

3.3.6 The end of the instrument definition

The end of the instrument definition is marked using the keyword

`END`

3.3.7 Code for the instrument `vanadium_example.instr`

An instrument definition taken from the `examples` directory is given as an example.

```
DEFINE INSTRUMENT test_v_sample(ROT)

DECLARE
%{
```

```
  double coll_div = 60;
%}

TRACE
COMPONENT arm = Arm() AT (0,0,0) ABSOLUTE

COMPONENT source = Source_flat(radius = 0.015, dist = 1,
  xw=0.024, yh=0.015, E0=5, dE=0.2)
 AT (0,0,0) RELATIVE arm

COMPONENT collimator = Soller(len = 0.2, divergence = coll_div,
  xmin = -0.02, xmax = 0.02, ymin = -0.03, ymax = 0.03)
  AT (0, 0, 0.4) RELATIVE arm

COMPONENT target = V_sample(radius_i = 0.008, radius_o = 0.012,
  h = 0.015, focus_r = 0, pack = 1,
  target_x = 0, target_y = 0, target_z = 1)
  AT (0,0,1) RELATIVE arm

COMPONENT PSD_4pi = PSD_monitor_4PI(radius=1e6, nx=101, ny=51,
  filename="vanadium.psd")
  AT (0,0,0) RELATIVE target ROTATED (ROT,0,0) RELATIVE arm
END
```

3.4 Writing component definitions

The purpose of a component definition is to model the interaction of a neutron with the component. Given the state of the incoming neutron, the component definition calculates the state of the neutron when it leaves the component. The calculation of the effect of the component on the neutron is performed by a block of embedded C code. One example of a component definition is given in section 3.4.8, and all component definitions can be found on the McStas web-page [2].

3.4.1 The component definition header

`DEFINE COMPONENT` *name*

This marks the beginning of the definition, and defines the name of the component.

`DEFINITION PARAMETERS` $(d_1, d_2, \ldots)$
`SETTING PARAMETERS` $(s_1, s_2, \ldots)$

This declares the definition and setting parameters of the component. The parameters define the characteristics of the component, and can be accessed from the `DECLARE`, `INITIALIZE`, `TRACE`, `FINALLY`, and `MCDISPLAY` sections (see below).

Setting parameters are translated into C variables of type `double` in the generated simulation program, so they can only be numbers. Definition parameters are translated into `#define` macro definitions, and so can have any type, including strings, arrays, and function pointers.

However, because of the use of `#define`, definition parameters suffer from the usual problems with C macro definitions. Also, it is not possible to use a general C expression

for the value of a definition parameter in the instrument definition, only constants and variable names may be used. For this reason, setting parameters should be used whenever possible.

There are a few cases where the use of definition parameters instead of setting parameters makes sense. If the parameter is not numeric (*i.e.* a string for a file name, for example), a setting parameter cannot be used. Also, because of the use of `#define`, the C compiler can treat definition parameters as constants when the simulation is compiled. For example, if the array sizes of a multidetector are definition parameters, the arrays can be statically allocated in the component `DECLARE` section. If setting parameters were used, it would be necessary to allocate the arrays dynamically using *e.g.* `malloc()`.

`OUTPUT PARAMETERS` $(s_1, s_2, \ldots)$

This declares a list of C identifiers that are output parameters for the component. Output parameters are used to hold values that are computed by the component itself, rather than being passed as input. This could for example be a count of neutrons in a detector or a constant that is precomputed to speed up computation. Output parameters will typically be declared as C variables in the `DECLARE` section, see section 3.4.2 below for an example.

The `OUTPUT PARAMETERS` section is optional.

`STATE PARAMETERS` $(x, y, z, v_x, v_y, v_z, t, s_1, s_2, p)$

This declares the parameters that define the state of the incoming neutron. The task of the component code is to assign new values to these parameters based on the old values and the values of the definition and setting parameters. Note that s_1 and s_2 are obsolete and cannot be used.

`POLARISATION PARAMETERS` (s_x, s_y, s_z)

This line is necessary only if the component handles polarisation of neutrons and thus modifies the spin vector. For an instrument to handle polarisation correctly, it is only required that *one* of the components contains this line.

Optional component parameters

The definition and setting parameters of a component may be given a default value. Parameters with default values are called *optional parameters*, and need not be given an explicit value when the component is used in an instrument definition. A parameter is given a default value using the syntax "*param* `=` *value*". For example

```
SETTING PARAMETERS (radius, height, pack = 1)
```

Here `pack` is an optional parameter and if no value is given explicitly, "1" will be used[1]. In the present version of McStas, only numbers can be given as default values.

Optional parameters can greatly increase the convenience for users of components with many parameters that have natural default values which are seldom changed. Optional

[1] In contrast, if no value is given for `radius` or `height`, an error message will result.

parameters are also useful to preserve backwards compatibility with old instrument definitions when a component is updated. New parameters can be added with default values that correspond to the old behavior, and existing instrument definitions can be used with the new component without changes.

However, optional parameters should not be used in cases where no general natural default value exists. For example, the length of a guide or the size of a slit should not be given default values. This would prevent the error messages that should be given in the common case of a user forgetting to set an important parameter.

3.4.2 The `DECLARE` section

```
DECLARE
%{
      ... C code declarations ...
%}
```

This gives C declarations of global variables etc. that are used by the component code. This may for instance be used to declare a neutron counter for a detector component. This section is optional.

Note that any variables declared in a `DECLARE` section are *global*. Thus a name conflict may occur if two instances of a component are used in the same instrument. To avoid this, variables declared in the `DECLARE` section should be output parameters of the component because McStas will then rename variables to avoid conflicts. For example, a simple detector might be defined as follows:

```
DEFINE COMPONENT Detector
OUTPUT PARAMETERS (counts)
DECLARE
%{
  int counts;
%}
...
```

The idea is that the `counts` variable counts the number of neutrons detected. In the instrument definition, the `counts` parameter may be referenced using the `MC_GETPAR` C macro, as in the following example instrument fragment:

```
COMPONENT d1 = Detector()
...
COMPONENT d2 = Detector()
...
FINALLY
%{
  printf("Detector counts: d1 = %d, d2 = %d\n",
         MC_GETPAR(d1,counts), MC_GETPAR(d2,counts));
%}
```

3.4.3 The INITIALIZE section

```
INITIALIZE
%{
      ... C code initialization ...
%}
```

This gives C code that will be executed once at the start of the simulation, usually to initialize any variables declared in the DECLARE section. This section is optional.

3.4.4 The TRACE section

```
TRACE
%{
      ... C code to compute neutron interaction with component ...
%}
```

This performs the actual computation of the interaction between the neutron and the component. The C code should perform the appropriate calculations and assign the resulting new neutron state to the state parameters.

The C code may also execute the special macro ABSORB to indicate that the neutron has been absorbed in the component and the simulation of that neutron will be aborted. If the component simulates multiple events (for example multiple reflections in a guide, or multiple scattering in a powder sample), the special macro SCATTER should be called. This does not affect the results of the simulation in any way, but it allows the front-end programs to visualize the scattering events properly. The SCATTER macro should be called with the state parameters set to the proper values for the scattering event. For an example of SCATTER, see the Channeled_guide component.

3.4.5 The FINALLY section

```
FINALLY
%{
      ... C code to execute at end of simulation ...
%}
```

This gives code that will be executed when the simulation has ended. This might be used to print out results from components, *e.g.* the simulated intensity in a detector.

In order to work properly with the common output file format used in McStas, all monitor/detector components should use standard macros for outputting data in the FINALLY section, as explained below. In the following, we use $N = \sum_i p_i^0$ to denote the count of detected neutron events, $p = \sum_i p_i$ to denote the sum of the weights of detected neutrons, and $p2 = \sum_i p_i^2$ to denote the sum of the squares of the weights, as explained in section 4.1.1.

Single detectors/monitors The results of a single detector/monitor are written using the following macro:

`DETECTOR_OUT_0D(`t, N, p, $p2$`)`

Here, t is a string giving a short descriptive title for the results, *e.g.* "Single monitor".

One-dimensional detectors/monitors The results of a one-dimensional detector/ monitor are written using the following macro:

`DETECTOR_OUT_1D(`t, *xlabel*, *ylabel*, *xvar*, $x_{\min}$, $x_{\max}$, m, &N[0], &p[0], &$p2$[0], *filename*`)`

Here,

- t is a string giving a descriptive title (*e.g.* "Energy monitor"),
- *xlabel* is a string giving a descriptive label for the X axis in a plot (*e.g.* "Energy [meV]"),
- *ylabel* is a string giving a descriptive label for the Y axis of a plot (*e.g.* "Intensity"),
- *xvar* is a string giving the name of the variable on the X axis (*e.g.* "E"),
- $x_{\min}$ is the lower limit for the X axis,
- $x_{\max}$ is the upper limit for the X axis,
- m is the number of elements in the detector arrays,
- &N[0] is a pointer to the first element in the array of N values for the detector component (or NULL, in which case no error bars will be computed),
- &p[0] is a pointer to the first element in the array of p values for the detector component,
- &$p2$[0] is a pointer to the first element in the array of $p2$ values for the detector component (or NULL, in which case no error bars will be computed),
- *filename* is a string giving the name of the file in which to store the data.

Two-dimensional detectors/monitors The results of a two-dimensional detector/ monitor are written to a file using the following macro:

`DETECTOR_OUT_2D(`t, *xlabel*, *ylabel*, $x_{\min}$, $x_{\max}$, $y_{\min}$, $y_{\max}$, m, n, &N[0][0], &p[0][0], &$p2$[0][0], *filename*`)`

Here,

- t is a string giving a descriptive title (*e.g.* "PSD monitor"),
- *xlabel* is a string giving a descriptive label for the X axis in a plot (*e.g.* "X position [cm]"),
- *ylabel* is a string giving a descriptive label for the Y axis of a plot (*e.g.* "Y position [cm]"),

- $x_{\min}$ is the lower limit for the X axis,
- $x_{\max}$ is the upper limit for the X axis,
- $y_{\min}$ is the lower limit for the Y axis,
- $y_{\max}$ is the upper limit for the Y axis,
- m is the number of elements in the detector arrays along the X axis,
- n is the number of elements in the detector arrays along the Y axis,
- $\&N[0][0]$ is a pointer to the first element in the array of N values for the detector component,
- $\&p[0][0]$ is a pointer to the first element in the array of p values for the detector component,
- $\&p2[0][0]$ is a pointer to the first element in the array of *p2* values for the detector component,
- *filename* is a string giving the name of the file in which to store the data.

Note that for a two-dimensional detector array, the first dimension is along the X axis and the second dimension is along the Y axis. This means that element (i_x, i_y) can be obtained as $p[i_x * n + i_y]$ if p is a pointer to the first element.

3.4.6 The `MCDISPLAY` section

```
MCDISPLAY
%{
      ... C code to draw a sketch of the component ...
%}
```

This gives C code that draws a sketch of the component in the plots produced by the `mcdisplay` front-end (see section 2.6.4). The section can contain arbitrary C code and may refer to the parameters of the component, but usually it will consist of a short sequence of the special commands described below that are available only in the MCDISPLAY section. When drawing components, all distances and positions are in meters and specified in the local coordinate system of the component.

The MCDISPLAY section is optional. If it is omitted, `mcdisplay` will use a default symbol (a small circle) for drawing the component.

The `magnify` command This command, if present, must be the first in the section. It takes a single argument: a string containing zero or more of the letters "x", "y" and "z". It causes the drawing to be enlarged along the specified axis in case `mcdisplay` is called with the `--zoom` option. For example:

```
magnify("xy");
```

The `line` command The `line` command takes the following form:

`line(`x_1`,` y_1`,` z_1`,` x_2`,` y_2`,` z_2`)`

It draws a line between the points (x_1, y_1, z_1) and (x_2, y_2, z_2).

The `multiline` command The `multiline` command takes the following form:

`multiline(`n`,` x_1`,` y_1`,` z_1`, ...,` x_n`,` y_n`,` z_n`)`

It draws a series of lines through the n points (x_1, y_1, z_1), (x_2, y_2, z_2), ..., (x_n, y_n, z_n). It thus accepts a variable number of arguments depending on the value of n. This exposes one of the nasty quirks of C since *no* type checking is performed by the C compiler. It is thus very important that all arguments to `multiline` (except n) are valid numbers of type `double`. A common mistake is to write

```
multiline(3, x, y, 0, ...)
```

which will silently produce garbage output. This must instead be written as

```
multiline(3, (double)x, (double)y, 0.0, ...)
```

The `circle` command The `circle` command takes the following form:

`circle(`*plane*`,` x`,` y`,` z`,` r`)`

Here *plane* should be either `"xy"`, `"xz"`, or `"yz"`. The command draws a circle in the specified plane with the center at (x, y, z) and the radius r.

3.4.7 The end of the component definition

```
END
```

This marks the end of the component definition.

3.4.8 A component example: Slit

A simple example of the component `Slit` is given.

```
/*******************************************************************************
*
* McStas, the neutron ray-tracing package: Slit.comp
*         Copyright 1997-2001 Risoe National Laboratory, Roskilde, Denmark
*
* %I
* Written by: Kim Lefmann and Henrik M. Roennow
* Date: June 16, 1997
* Version: $Revision: 1.7 $
* Origin: McStas 1.5
*
* Rectangular slit.
*
* %D
```

```
* A simple rectangular slit. No transmission around the slit is allowed.
*
* %P
* INPUT PARAMETERS
*
* xmin: Lower x bound (m)
* xmax: Upper x bound (m)
* ymin: Lower y bound (m)
* ymax: Upper y bound (m)
*
* %E
*******************************************************************************/


DEFINE COMPONENT Slit
DEFINITION PARAMETERS ()
SETTING PARAMETERS (xmin, xmax, ymin, ymax)
STATE PARAMETERS (x,y,z,vx,vy,vz,t,s1,s2,p)
TRACE
%{
    PROP_Z0;
    if (x<xmin || x>xmax || y<ymin || y>ymax)
      ABSORB;
%}

MCDISPLAY
%{
  double xw, yh;
  magnify("xy");
  xw = (xmax - xmin)/2.0;
  yh = (ymax - ymin)/2.0;
  multiline(3, xmin-xw, (double)ymax, 0.0,
            (double)xmin, (double)ymax, 0.0,
            (double)xmin, ymax+yh, 0.0);
  multiline(3, xmax+xw, (double)ymax, 0.0,
            (double)xmax, (double)ymax, 0.0,
            (double)xmax, ymax+yh, 0.0);
  multiline(3, xmin-xw, (double)ymin, 0.0,
            (double)xmin, (double)ymin, 0.0,
            (double)xmin, ymin-yh, 0.0);
  multiline(3, xmax+xw, (double)ymin, 0.0,
            (double)xmax, (double)ymin, 0.0,
            (double)xmax, ymin-yh, 0.0);
%}

END
```

3.4.9 McDoc, the McStas component documentation tool

McStas includes a facility called McDoc to help maintain good documentation of components. In the component source code, comments may be written that follow a particular format understood by McDoc. The McDoc facility will read these comments and automatically produce output documentation in various forms. By using the source code itself as the source of documentation, the documentation is much more likely to be a faithful and up-to-date description of how the component actually works.

Two forms of documentation can be generated. One is the component entry dialog in the `mcgui` front-end, see section 2.6.1. The other is a collection of web pages documenting the component, and the complete documentation for all available McStas components may be found at the McStas webpage [2].

Note that McDoc-compliant comments in the source code are no substitute for a good reference manual entry. The mathematical equations describing the physics and algorithms of the component should still be written up carefully for inclusion in the component manual. The McDoc comments are useful for describing the general behaviour of the component, the meaning and units of the input parameters, etc.

The format of the comments in the component source code

The format of the comments understood by McDoc is mostly straight-forward, and is designed to be easily readable both by humans and by automatic tools. McDoc has been written to be quite tolerant in terms of how the comments may be formatted and broken across lines. A good way to get a feeling for the format is to study some of the examples in the existing components. Below, a few notes are listed on the requirements for the comment headers:

The comment syntax uses `%IDENTIFICATION`, `%DESCRIPTION`, `%PARAMETERS`, `%LINKS`, and `%END` keywords to mark different sections of the documentation. Keywords may be abbreviated, *e.g.* as `%IDENT` or `%I`.

- In the `%IDENTIFICATION` section, `author:` (or `written by:` for backwards compatibility with old comments) denote author; `date:`, `version:`, and `origin:` are also supported. Any number of `Modified by:` entries may be used to give the revision history. The `author:`, `date:`, etc. entries must all appear on a single line of their own. Everything else in the identification section is part of a "short description" of the component.

- In the `%PARAMETERS` section, descriptions have the form "*name*: `[`*unit*`]` *text*" or "*name*: *text* `[`*unit*`]`". These may span multiple lines, but subsequent lines must be indented by at least four spaces. Note that square brackets `[]` should be used for units. Normal parenthesis are also supported for backwards compatibility, but nested parenthesis do not work well.

- The `%DESCRIPTION` section contains text in free format. The text may contain HTML tags like `<IMG>` (to include pictures) and `<A>...</A>` (for links to other web pages, but see also the `%LINK` section). In the generated web documentation pages, the text is set in `<PRE>...</PRE>`, so that the line breaks in the source will be obeyed.

- Any number of `%LINK` sections may be given; each one contains HTML code that will be put in a list item in the link section of the description web page. This usually consists of an `<A HREF="..."> ... </A>` pointer to some other source of information.

- After `%END`, no more comment text is read by McDoc.

Chapter 4

Monte Carlo Techniques and simulation strategy

This chapter explains the simulation strategy and the Monte Carlo techniques used in McStas. We first explain the concept of the neutron weight factor, and discuss the statistical errors in dealing with sums of neutron weights. Secondly, we give an expression for how the weight factor should transform under a Monte Carlo choice and specialize this to the concept of focusing components. Finally, we present a way of generating random numbers with arbitrary distributions.

4.1 The neutron weight, p

A totally realistic semi-classical simulation will require that each neutron is at any time either present or not (it might be ABSORB'ed or lost in another way). In many setups, *e.g.* triple axis spectrometers, only a small fraction of the initial neutrons will ever be detected, and simulations of this kind will therefore waste much time in dealing with neutrons that get lost.

An important way of speeding up calculations is to introduce a neutron weight for each simulated neutron and to adjust this weight according to the path of the neutron. If *e.g.* the reflectivity of a certain optical component is 10%, and only reflected neutrons are considered in the simulations, the neutron weight will be multiplied by 0.10 when passing this component, but every neutron is allowed to reflect in the component. In contrast, the totally realistic simulation of the component would require in average ten incoming neutrons for each reflected one.

Let the initial neutron weight be p_0 and let us denote the weight multiplication factor in the j'th component by π_j. The resulting weight factor for the neutron after passage of the whole instrument is equal to the product of all the contributions

$$p = p_0 \prod_{j=1}^{n} \pi_j. \tag{4.1}$$

For convenience, the value of p is updated within each component.

Simulation by weight adjustment is performed whenever possible. This includes

- Transmission through filter.
- Transmission through Soller blade collimator (in the approximation which does not take each blade into account).
- Reflection from monochromator (and analyser) crystals with finite reflectivity and mosaicity.
- Scattering from samples.

4.1.1 Statistical errors of non-integer counts

In a typical simulation, the result will consist of a count of neutrons with different weights.[1] One may write the counting result as

$$I = \sum_i p_i = N\overline{p}, \tag{4.2}$$

where N is the number of neutrons in the detector and the vertical bar denote averaging. By performing the weight transformations, the (statistical) mean value of I is unchanged. However, N will in general be enhanced, and this will improve the statistics of the simulation.

To give some estimate of the statistical error, we proceed as follows: Let us first for simplicity assume that all the counted neutron weights are almost equal, $p_i \approx \overline{p}$, and that we observe a large number of neutrons, $N \geq 10$. Then N almost follows a normal distribution with the uncertainty $\sigma(N) = \sqrt{N}$ [2]. Hence, the statistical uncertainty of the observed intensity becomes

$$\sigma(I) = \sqrt{N}\overline{p} = I/\sqrt{N}, \tag{4.3}$$

as is used in real neutron experiments (where $\overline{p} \equiv 1$). For a better approximation we return to Eq. (4.2). Allowing variations in both N and $\overline{p}$, we calculate the variance of the resulting intensity, assuming that the two variables are independent and both follow a Gaussian distribution.

$$\sigma^2(I) = \sigma^2(N)\overline{p}^2 + N^2\sigma^2(\overline{p}) = N\overline{p}^2 + N^2\sigma^2(\overline{p}). \tag{4.4}$$

Assuming that the individual weights, p_i, follow a Gaussian distribution (which in many cases is far from the truth) we have $N^2\sigma^2(\overline{p}) = \sigma^2(\sum_i p_i) = N\sigma^2(p_i)$ and reach

$$\sigma^2(I) = N\left(\overline{p}^2 + \sigma^2(p_i)\right). \tag{4.5}$$

The statistical variance of the p_i's is estimated by $\sigma^2(p_i) \approx (N-1)^{-1}(\sum_i p_i^2 - N\overline{p}^2)$. The resulting variance then reads

$$\sigma^2(I) = \frac{N}{N-1}\left(\sum_i p_i^2 - \overline{p}^2\right). \tag{4.6}$$

[1] The sum of these weights is an estimate of the mean number of neutrons hitting the monitor (or detector) in a "real" experiment where the number of neutrons emitted from the source is the same as the number of simulated neutrons.

[2] This is not correct in a situation where the detector counts a large fraction of the neutrons in the simulation, but we will neglect that for now.

For large values of N, this is very well approximated by the simple expression

$$\sigma^2(I) \approx \sum_i p_i^2. \tag{4.7}$$

In order to compute the intensities and uncertainties, the detector components in McStas thus must keep track of $N = \sum_i p_i^0$, $I = \sum_i p_i^1$, and $M_2 = \sum_i p_i^2$.

4.2 Weight factor transformations during a Monte Carlo choice

When a Monte Carlo choice must be performed, *e.g.* when the initial energy and direction of the neutron is decided at the source, it is important to adjust the neutron weight so that the combined effect of neutron weight change and Monte Carlo probability equals the actual physical properties of the component.

Let us follow up on the example of a source. In the "real" semi-classical world, there is a distribution (probability density) for the neutrons in the six dimensional (energy, direction, position) space of $\Pi(E, \mathbf{\Omega}, \mathbf{r}) = dP/(dEd\mathbf{\Omega}d^3\mathbf{r})$ depending upon the source temperature, geometry *etc.* In the Monte Carlo simulations, the six coordinates are for efficiency reasons in general picked from another distribution: $f_{\mathrm{MC}}(E, \mathbf{\Omega}, \mathbf{r}) \neq \Pi(E, \mathbf{\Omega}, \mathbf{r})$, since one would *e.g.* often generate only neutrons within a certain parameter interval. However, we must then require that the weights are adjusted by a factor π_j (in this case: $j = 1$) so that

$$f_{\mathrm{MC}}(E, \mathbf{\Omega}, \mathbf{r})\pi_j(E, \mathbf{\Omega}, \mathbf{r}) = \Pi(E, \mathbf{\Omega}, \mathbf{r}). \tag{4.8}$$

For the sources present in version 1.4, only the $(\mathbf{\Omega}, \mathbf{r})$ dependence of the correction factors are taken into account.

The weight factor transformation rule Eq. (4.8) is of course also valid for other types of Monte Carlo choices, although the probability distributions may depend upon different parameters. An important example is elastic scattering from a powder sample, where the Monte-Carlo choices are the scattering position and the final neutron direction.

It should be noted that the π_j's found in the weight factor transformation are multiplied by the π_j's found by the weight adjustments described in subsection 4.1 to yield the final neutron weight given by Eq. (4.1).

4.2.1 Focusing components

An important application of weight transformation is focusing. Assume that the sample scatters the neutrons in many directions. In general, only neutrons flying in some of these directions will stand any chance of being detected. These directions we call the *interesting directions*. The idea in focusing is to avoid wasting computation time on neutrons scattered in the uninteresting directions. This trick is an instance of what in Monte Carlo terminology is known as *importance sampling*.

If *e.g.* a sample scatters isotropically over the whole 4π solid angle, and all interesting directions are known to be contained within a certain solid angle interval $\Delta\mathbf{\Omega}$, only these solid angles are used for the Monte Carlo choice of scattering direction. According to Eq. (4.8), the weight factor will then have to be changed by the (fixed) amount $\pi_j =$

$|\Delta\mathbf{\Omega}|/(4\pi)$. One thus ensures that the mean simulated intensity is unchanged during a "correct" focusing, while a too narrow focusing will result in a lower (*i.e.* wrong) intensity, since one cuts away neutrons that would otherwise have counted.

One could also think of using adaptive importance sampling, so that McStas during the simulations will determine the most interesting directions and gradually change the focusing according to that. A first implementation of this idea is found in the Source_adapt component.

4.3 Transformation of random numbers

In order to perform the Monte Carlo choices, one needs to be able to pick a random number from a given distribution. However, most random number generators only give uniform distributions over a certain interval. We thus need to be able to transform between probability distributions, and we here give a short explanation on how to do this.

Assume that we pick a random number, x, from a distribution $\phi(x)$. We are now interested in the shape of the distribution of the transformed $y = f(x)$, assuming $f(x)$ is monotonous. All random numbers lying in the interval $[x; x + dx]$ are transformed to lie within the interval $[y; y+f'(x)dx]$, so the resulting distribution must be $\phi(y) = \phi(x)/f'(x)$.

If the random number generator selects numbers uniformly in the interval $[0; 1]$, we have $\phi(x) = 1$, and one may evaluate the above expression further

$$\phi(y) = \frac{1}{f'(x)} = \frac{d}{dy} f^{-1}(y). \tag{4.9}$$

By indefinite integration we reach

$$\int \phi(y)dy = f^{-1}(y) = x, \tag{4.10}$$

which is the essential formula for finding the right transformation of the initial random numbers. Let us illustrate with a few examples of transformations used within the McStas components.

The circle For finding a random point within the circle of radius R, one would like to choose the polar angle from a uniform distribution in $[0; 2\pi]$ and the radius from the normalised distribution $\phi(r) = 2r/R^2$. The polar angle is found simply by multiplying a random number with 2π. For the radius, we like to find $r = f(x)$, where again x is the generated random number. Left side of Eq. (4.10) gives $\int \phi(r)dr = \int 2r/R^2 dr = r^2/R^2$, which should equal x. Hence $r = R\sqrt{x}$.

Exponential decay In a simple time-of-flight source, the neutron flux decays exponentially after the initial activation at $t = 0$. We thus want to pick an initial neutron emission time from the normalised distribution $\phi(t) = \exp(-t/\tau)/\tau$. Use of Eq. (4.10) gives $x = -\exp(-t/\tau)$, which is a number in the interval $[-1; 0]$. If we want to pick a positive random number instead, we will have to change sign by $x_1 = -x$ and thus reach $t = -\tau \ln(x_1)$.

The sphere For finding a random point on the surface of the unit sphere, one needs to determine the two angles, (θ, ψ). As for a circle, ψ is chosen from a uniform distribution in $[0; 2\pi]$. The probability distribution of θ should be $\phi(\theta) = \sin(\theta)$ (for $\theta \in [0; \pi/2]$), whence $\theta = \cos^{-1}(x)$.

Chapter 5

The component library

This chapter has been removed from the manual and will instead be published in a separate manual describing the McStas components. The McStas component manual will be edited by the McStas authors and it will include contributions from users writing components. Until the McStas component manual is published we refer to the McStas web-page [2] where all components are documented using the McDoc system or to previous manual for release 1.4.

Chapter 6

Instrument examples

Here, we give a short description of three selected instruments. We present the McStas versions of the Risø standard triple axis spectrometer TAS1 (6.2) and the ISIS time-of-flight spectrometer PRISMA (6.3). Before that, however, we present one example of a component test instrument: the instrument to test the component **V_sample** (6.1). These instrument files are included in the McStas distribution in the `examples/` directory. It is also our intention to extend the list of instrument examples extensively and perhaps publish them in a separate report.

6.1 A test instrument for the component V_sample

This instrument is one of many test instruments written with the purpose of testing the individual components. We have picked this instrument both because we would like to present an example test instrument and because it despite its simplicity has produced quite non-trivial results, also giving rise to the McStas logo.

The instrument consists of a narrow source, a 60' collimator, a V-sample shaped as a hollow cylinder with height 15 mm, inner diameter 16 mm, and outer diameter 24 mm at a distance of 1 m from the source. The sample is in turn surrounded by an unphysical 4π-PSD monitor with 50×100 pixels and a radius of 10^6 m. The set-up is shown in figure 6.1.

6.1.1 Scattering from the V-sample test instrument

In figure 6.2, we present the radial distribution of the scatting from an evenly illuminated V-sample, as seen by a spherical PSD. It is interesting to note that the variation in the scattering intensity is as large as 10%. This is an effect of attenuation of the beam in the cylindrical sample.

6.2 The triple axis spectrometer TAS1

With this instrument definition, we have tried to create a very detailed model of the conventional cold-source triple-axis spectrometer TAS1 at Risø National Laboratory. Unfortunately, no neutron scattering is performed at Risø anymore, but it still serves as a

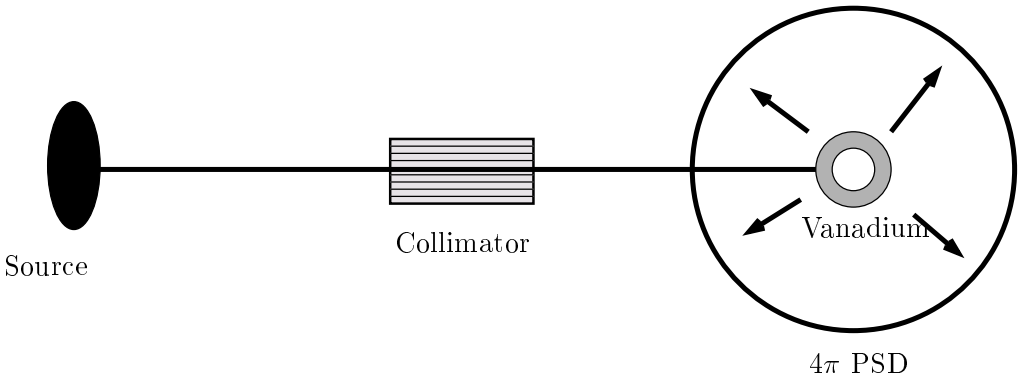

Figure 6.1: A sketch of the test instrument for the component V_sample.

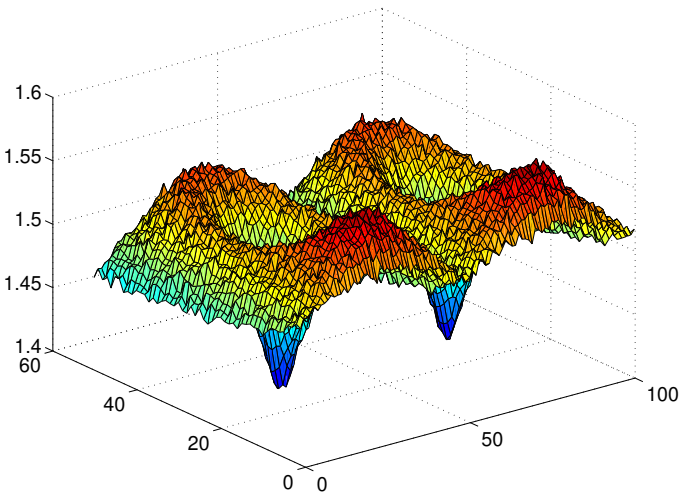

Figure 6.2: Scattering from a V-sample, measured by a spherical PSD. The sphere has been transformed onto a plane and the intensity is plotted as the third dimension. A colour version of this picture is found on the title page of this manual.

good example. Except for the cold source itself, all components used have quite realistic properties. Furthermore, the overall geometry of the instrument has been adapted from the detailed technical drawings of the real spectrometer. The TAS 1 simulation is the first detailed work performed with the McStas package.For further details see reference [9].

At the spectrometer, the channel from the cold source to the monochromator is asymmetric, since the first part of the channel is shared with other instruments. In the instrument definition, this is represented by three slits. For the cold source, we use a flat energy distribution (component **Source_flat**) focusing on the third slit.

The real monochromator consist of seven blades, vertically focusing on the sample. The angle of curvature is constant so that the focusing is perfect at 5.0 meV (20.0 meV for 2nd order reflections) for a 1×1 cm^2 sample. This is modeled directly in the instrument definition using seven **Monochromator** components. The mosaicity of the pyrolytic graphite crystals is nominally 30' (FWHM) in both directions. However, the simulations indicated that the horisontal mosaicities of both monochromator and analyser were more likely 45'. This was used for all mosaicities in the final instrument definition.

The monochromator scattering angle, in effect determining the incoming neutron energy, is for the real spectrometer fixed by four holes in the shielding, corresponding to the energies 3.6, 5.0, 7.2, and 13.7 meV for first order neutrons. In the instrument definition, we have adapted the angle corresponding to 5.0 meV in order to test the simulations against measurements performed on the spectrometer.

The exit channel from the monochromator may on the spectrometer be narrowed down from initially 40 mm to 20 mm by an insert piece. In the simulations, we have chosen the 20 mm option and modeled the channel with two slits to match the experimental set-up.

In the test experiments, we used two standard samples: An Al_2O_3 powder sample and a vanadium sample. The instrument definitions use either of these samples of the correct size. Both samples are chosen to focus on the opening aperture of collimator 2 (the one between the sample and the analyser). Two slits, one before and one after the sample, are in the instrument definition set to the opening values which were used in the experiments.

The analyser of the spectrometer is flat and made from pyrolytic graphite. It is placed between an entry and an exit channel, the latter leading to a single detector. All this has been copied into the instrument definition, where the graphite mosaicity has been set to 45'.

On the spectrometer, Soller collimators may be inserted at three positions: Between monochromator and sample, between sample and analyser, and between analyser and detector. In our instrument definition, we have used 30', 28', and 67' collimators on these three positions, respectively.

An illustration of the TAS1 instrument is shown in figure 6.3. Test results and data from the real spectrometer are shown in Appendix 6.2.1.

6.2.1 Simulated and measured resolution of TAS1

In order to test the McStas package on a qualitative level, we have performed a very detailed simulation of the conventional triple axis spectrometer TAS1, Risø. The measurement series constitutes a complete alignment of the spectrometer, using the direct beam and scattering from V and Al_2O_3 samples at an incoming energy of 20.0 meV, using the second order scattering from the monochromator. In the instrument definitions, we

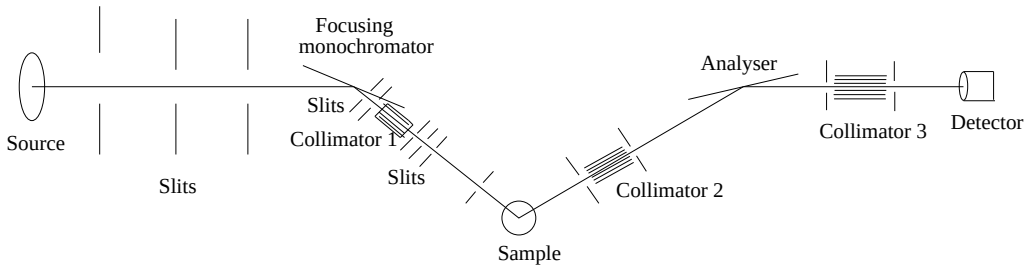

Figure 6.3: A sketch of the TAS1 instrument.

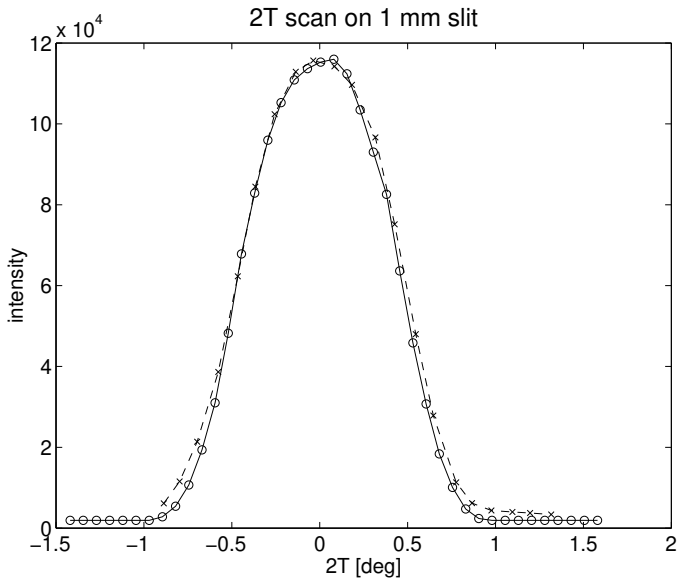

Figure 6.4: Scans of $2\theta_s$ in the direct beam with 1 mm slit on the sample position. "$\times$": measurements, "o": simulations Collimations: open-30'-open-open.

have used all available information about the spectrometer. However, the mosaicities of the monochromator and analyser are set to 45' in stead of the quoted 30', since we from our analysis believe this to be much closer to the truth.

In these simulations, we have tried to reproduce every alignment scan with respect to position and width of the peaks, whereas we have not tried to compare absolute intensities. Below, we show a few comparisons of the simulations and the measurements.

Figure 6.4 shows a scan of $2\theta_s$ on the collimated direct beam in two-axis mode. A 1 mm slit is placed on the sample position. Both the measured width and non-Gaussian peak shape are well reproduced by the McStas simulations.

In contrast, a simulated $2\theta_a$ scan in triple-axis mode on a V-sample showed a surprising offset from zero, see Figure 6.5. However, a simulation with a PSD on the sample position showed that the beam center was 1.5 mm off from the center of the sample, and this was important since the beam was no wider than the sample itself. A subsequent centering

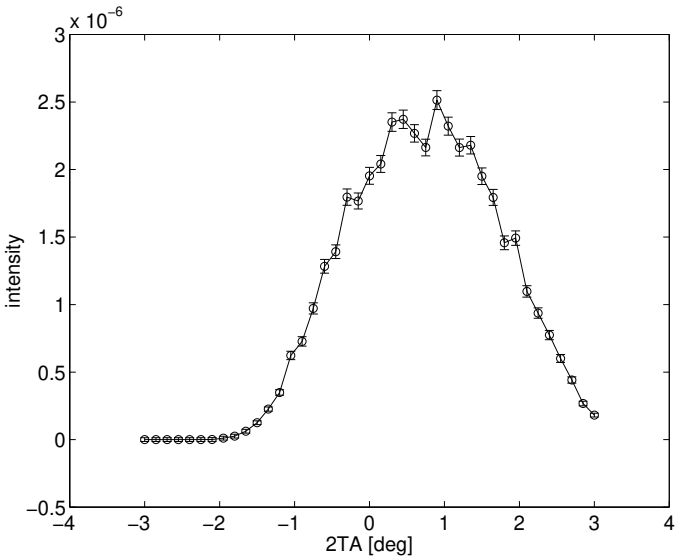

Figure 6.5: First simulated $2\theta_a$ scan on a vanadium sample. Collimations: open-30'-28'-open.

of the beam resulted in a nice agreement between simulation and measurements. For a comparison on a slightly different instrument (analyser-detector collimator inserted), see Figure 6.6.

The result of a $2\theta_s$ scan on an Al_2O_3 powder sample in two-axis mode is shown in Figure 6.7. Both for the scan in focusing mode $(+\ -\ +)$ and for the one in defocusing mode $(+\ +\ +)$ (not shown), the agreement between simulation and experiment is excellent.

As a final result, we present a scan of the energy transfer $E_a = \hbar\omega$ on a V-sample. The data are shown in Figure 6.8.

6.3 The time-of-flight spectrometer PRISMA

In order to test the time-of-flight aspect of McStas, we have in collaboration with Mark Hagen, ISIS, written a simple simulation of a time-of-flight instrument loosely based on the ISIS spectrometer PRISMA. The simulation was used to investigate the effect of using a RITA-style analyser instead of the normal PRISMA backend.

We have used the simple time-of-flight source **Tof_source**. The neutrons pass through a beam channel and scatter off from a vanadium sample, pass through a collimator on to the analyser. The RITA-style analyser consists of seven analyser crystals that can be rotated independently around a vertical axis. After the analysers we have placed a PSD and a time-of-flight detector.

To illustrate some of the things that can be done in a simulation as opposed to a real-life experiment, this example instrument further discriminates between the scattering off each individual analyser crystal when the neutron hits the detector. The analyser component is modified so that a global variable `neu_color` keeps track of which crystal scatters the neutron. The detector component is then modified to construct seven different time-of-

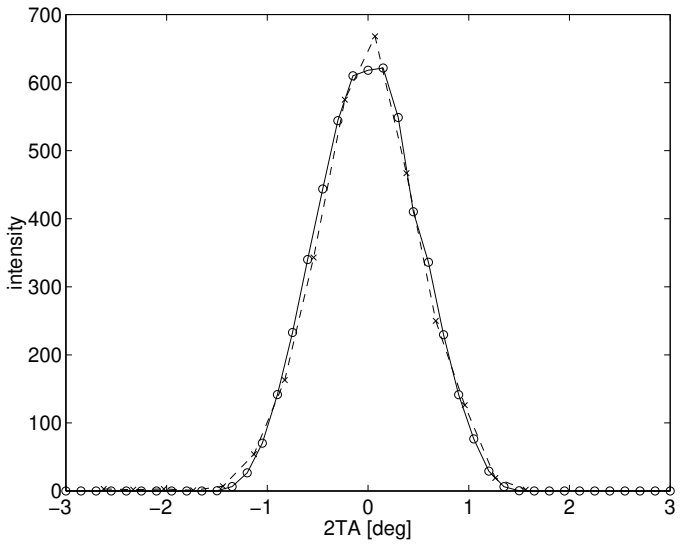

Figure 6.6: Corrected $2\theta_a$ scan on a V-sample. Collimations: open-30'-28'-67'. "×": measurements, "o": simulations.

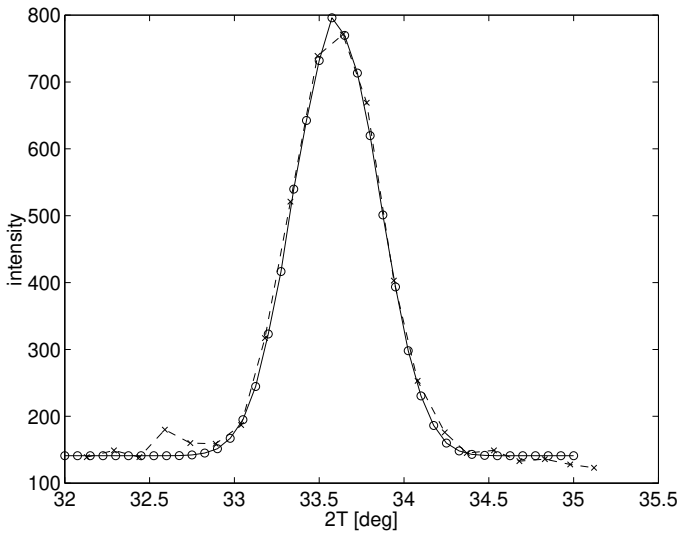

Figure 6.7: $2\theta_s$ scans on Al_2O_3 in two-axis, focusing mode. Collimations: open-30'-28'-67'. "×": measurements, "o": simulations. A constant background is added to the simulated data.

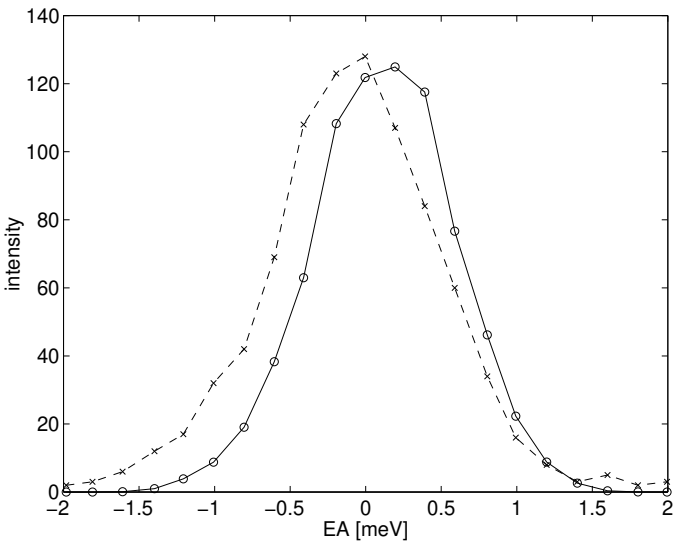

Figure 6.8: Scans of the analyser energy on a V-sample. Collimations: open-30'-28'-67'. "×": measurements, "o": simulations.

flight histograms, one for each crystal (see the source code for the instrument for details). One way to think of this is that the analyser blades paint a color on each neutron which is then observed in the detector. An illustration of the instrument is shown in figure 6.9. Test results are shown in Appendix 6.3.1.

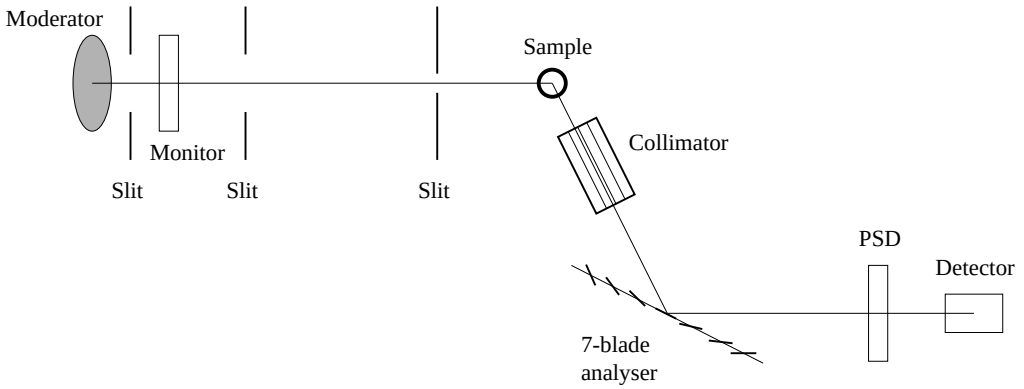

Figure 6.9: A sketch of the PRISMA instrument.

6.3.1 Simple spectra from the PRISMA instrument

A plot from the detector in the PRISMA simulation is shown in Figure 6.10. These results were obtained with each analyser blade rotated one degree relative to the previous one. The separation of the spectra of the different analyser blades is caused by different energy of scattered neutrons and different flight path length from source to detector. We have

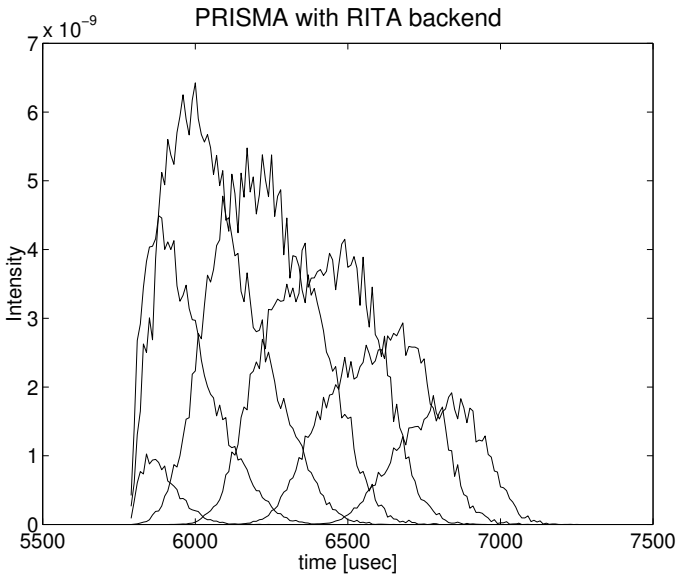

Figure 6.10: Test result from PRISMA instrument using "coloured neutrons". Each graph shows the neutrons scattered from one analyser blade.

not performed any quantitative analysis of the data at this time.

Appendix A

Kernel calls and conversion constants

The McStas kernel contains a number of built-in functions and conversion constants which are useful when constructing components. Here, we present a short list of these features.

A.1 Kernel calls and functions

Here we list a number of preprogrammed macros which may ease the task of writing component and instrument definitions.

A.1.1 Neutron propagation

- **ABSORB**. This macro issues an order to the overall McStas simulator to interrupt the simulation of the current neutron history and to start a new one.
- **PROP_Z0**. Propagates the neutron to the $z = 0$ plane, by adjusting (x, y, z) and t. If the neutron velocity points away from the $z = 0$ plane, the neutron is absorbed.
- **PROP_DT**(dt). Propagates the neutron through the time interval dt, adjusting (x, y, z) and t.
- **PROP_GRAV_DT**(dt, Ax, Ay, Az). Like **PROP_DT**, but it also includes gravity using the acceleration (Ax, Ay, Az). In addition, to adjusting (x, y, z) and t also (vx, vy, vz) is modified.
- **SCATTER**. This macro is used to denote a scattering event inside a component, see section 3.4.4.

A.1.2 Meta-language extensions

- **MC_GETPAR**(). This may be used in the finally section of an instrument definition to reference the output parameters of a component. See page 32 for details.
- **NAME_CURRENT_COMP** gives the name of the current component as a string.

- **POS_A_CURRENT_COMP** gives the absolute position of the current component. A component of the vector is referred to as **POS_A_CURRENT_COMP.i** where i is x, y or z.

- **ROT_A_CURRENT_COMP** and **ROT_R_CURRENT_COMP** give the orientation of the current component as rotation matrices (absolute orientation and the orientation relative to the previous component, respectively). A component of a rotation matrice is referred to as **ROT_A_CURRENT_COMP[m][n]**, where m and n are 0, 1, or 2.

- **POS_A_COMP(comp)** gives the absolute position of the component with the name *comp*. Note that *comp* is not given as a string. A component of the vector is referred to as **POS_A_COMP(comp).i** where i is x, y or z.

- **ROT_A_COMP(comp)** and **ROT_R_COMP(comp)** give the orientation of the component *comp* as rotation matrices (absolute orientation and the orientation relative to its previous component, respectively). Note that *comp* is not given as a string. A component of a rotation matrice is referred to as **ROT_A_COMP(comp)[m][n]**, where m and n are 0, 1, or 2.

- **extend_list**(n, &*arr*, &*len*, *elemsize*). Given an array *arr* with *len* elements each of size *elemsize*, make sure that the array is big enough to hold at least n elements, by extending *arr* and *len* if necessary. Typically used when reading a list of numbers from a data file when the length of the file is not known in advance.

- **mcget_ncount**(). Returns the number of neutron histories to simulate.

A.1.3 Mathematical routines

- **NORM**(x, y, z). Normalizes the vector (x, y, z) to have length 1.

- **scalar_prod**$(a_x, a_y, a_z, b_x, b_y, b_z)$. Returns the scalar product of the two vectors (a_x, a_y, a_z) and (b_x, b_y, b_z).

- **vecprod**$(a_x, a_y, a_z, b_x, b_y, b_z, c_x, c_y, c_z)$. Sets (a_x, a_y, a_z) equal to the vector product $(b_x, b_y, b_z) \times (c_x, c_y, c_z)$.

- **rotate**$(x, y, z, v_x, v_y, v_z, \varphi, a_x, a_y, a_z)$. Set (x, y, z) to the result of rotating the vector (v_x, v_y, v_z) the angle φ (in radians) around the vector (a_x, a_y, a_z).

- **normal_vec**(&n_x, &n_y, &n_z, x, y, z). Computes a unit vector (n_x, n_y, n_z) normal to the vector (x, y, z).

A.1.4 Output from detectors

- **DETECTOR_OUT_0D**(). Used to output the results from a single detector. The name of the detector is output together with the simulated intensity and estimated statistical error. The output is produced in a format that can be read by McStas front-end programs. See section 3.4.5 for details.

- **DETECTOR_OUT_1D**(). Used to output the results from a one-dimentional detector. See section 3.4.5 for details.

- **DETECTOR_OUT_2D**(). Used to output the results from a two-dimentional detector. See section 3.4.5 for details.

A.1.5 Ray-geometry intersections

- **box_intersect**(&t_1, &t_2, x, y, z, v_x, v_y, v_z, d_x, d_y, d_z). Calculates the (0, 1, or 2) intersections between the neutron path and a box of dimensions d_x, d_y, and d_z, centered at the origin for a neutron with the parameters (x, y, z, v_x, v_y, v_z). The times of intersection are returned in the variables t_1 and t_2, with $t_1 < t_2$. In the case of less than two intersections, t_1 (and possibly t_2) are set to zero. The function returns true if the neutron intersects the box, false otherwise.

- **cylinder_intersect**(&t_1, &t_2, x, y, z, v_x, v_y, v_z, r, h). Similar to **box_intersect**, but using a cylinder of height h and radius r, centered at the origin.

- **sphere_intersect**(&t_1, &t_2, x, y, z, v_x, v_y, v_z, r). Similar to **box_intersect**, but using a sphere of radius r.

A.1.6 Random numbers

- **rand01**(). Returns a random number distributed uniformly between 0 and 1.

- **randnorm**(). Returns a random number from a normal distribution centered around 0 and with $\sigma = 1$. The algorithm used to get the normal distribution is explained in [10], chapter 7.

- **randpm1**(). Returns a random number distributed uniformly between -1 and 1.

- **randvec_target_sphere**(&v_x, &v_y, &v_z, &$d\Omega$, aim_x, aim_y, aim_z, r_f). Generates a random vector (v_x, v_y, v_z), of the same length as (aim_x, aim_y, aim_z), which is targeted at a sphere centered at (aim_x, aim_y, aim_z) with radius r_f. All directions that intersect the sphere are chosen with equal probability. The solid angle of the sphere as seen from the position of the neutron is returned in $d\Omega$.

A.2 Constants for unit conversion etc.

The following predefined constants are useful for conversion between units

Name	Value	Conversion from	Conversion to
DEG2RAD	$2\pi/360$	Degrees	radians
RAD2DEG	$360/(2\pi)$	Radians	degrees
MIN2RAD	$2\pi/(360 \cdot 60)$	Minutes of arc	radians
RAD2MIN	$(360 \cdot 60)/(2\pi)$	Radians	minutes of arc
V2K	$10^{10} \cdot m_{\mathrm{N}}/\hbar$	Velocity (m/s)	**k**-vector ($\mathrm{\AA}^{-1}$)
K2V	$10^{-10} \cdot \hbar/m_{\mathrm{N}}$	**k**-vector ($\mathrm{\AA}^{-1}$)	Velocity (m/s)
VS2E	$m_{\mathrm{N}}/(2e)$	Velocity squared ($\mathrm{m}^2\ \mathrm{s}^{-2}$)	Neutron energy (meV)
SE2V	$\sqrt{2e/m_{\mathrm{N}}}$	Square root of neutron energy ($\mathrm{meV}^{1/2}$)	Velocity (m/s)
FWHM2RMS	$1/\sqrt{8\log(2)}$	Full width half maximum	Root mean square (standard deviation)
RMS2FWHM	$\sqrt{8\log(2)}$	Root mean square (standard deviation)	Full width half maximum

Further, we have defined the constants **PI**$= \pi$ and **HBAR**$= \hbar$.

Appendix B

The McStas terminology

This is a short explanation of phrases and terms which have a specific meaning within McStas. We have tried to keep the list as short as possible with the risk that the reader may occasionally miss an explanation. In this case, you are more than welcome to contact the authors.

- **Arm** A generic McStas component which defines a frame of reference for other components.
- **Component** One unit (*e.g.* optical element) in a neutron spectrometer.
- **Definition parameter** An input parameter for a component. For example the radius of a sample component or the divergence of a collimator.
- **Input parameter** For a component, either a definition parameter or a setting parameter. These parameters are supplied by the user to define the characteristics of the particular instance of the component definition. For an instrument, a parameter that can be changed at simulation run-time.
- **Instrument** An assembly of McStas components defining a neutron spectrometer.
- **McStas** Monte Carlo Simulation of Triple Axis Spectrometers (the name of this project).
- **Output parameter** An output parameter for a component. For example the counts in a monitor. An output parameter may be accessed from the instrument in which the component is used using `MC_GETPAR`.
- **Run-time** C code, contained in the files `mcstas-r.c` and `mcstas-r.h` included in the McStas distribution, that declare functions and variables used by the generated simulations.
- **Setting parameter** Similar to a definition parameter, but with the restriction that the value of the parameter must be a number.

Bibliography

[1] K. Lefmann and K. Nielsen. *Neutron News*, **10**, 20–23, 1999.

[2] See http://neutron.risoe.dk/mcstas/.

[3] T. E. Mason, K. N. Clausen, G. Aeppli, D. F. McMorrow, and J. K. Kjems. *Can. J. Phys.*, **73**, 697–702, 1995.

[4] K. N. Clausen, D. F. McMorrow, K. Lefmann, G. Aeppli, T. E. Mason, A. Schröder, M. Issikii, M. Nohara, and H. Takagi. *Physica B*, **241-243**, 50–55, 1998.

[5] K. Lefmann, D. F. McMorrow, H. M. Rønnow, K. Nielsen, K. N. Clausen, B. Lake, and G. Aeppli. *Physica B*, **283**, 343–354, 2000.

[6] See http://www.ess-europe.de.

[7] See http://www.debian.org/.

[8] See http://www.neutron.anl.gov/nexus/.

[9] A. Abrahamsen, N. B. Christensen, and E. Lauridsen. McStas simulations of the TAS1 spectrometer. Student's report, Niels Bohr Institute, University of Copenhagen, 1998.

[10] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C*. Cambridge University Press, 1986.

Bibliographic Data Sheet **Risø–R–1288(EN)**

Title and author(s)

User and Programmers Guide to the Neutron Ray-Tracing Package McStas, Version 1.5

Per-Olof Åstrand, Kim Lefmann, Emmanuel Farhi, Kristian Nielsen

ISBN	ISSN
87–550–2929–9; 87–550–2930–2 (Internet)	0106–2840

Dept. or group	Date
Materials Research Department	October 2001

Groups own reg. number(s)	Project/contract No.
—	—

Pages	Tables	Illustrations	References
60	2	15	10

Abstract (Max. 2000 char.)

The software package McStas is a tool for carrying out Monte Carlo ray-tracing simulations of neutron scattering instruments with high complexity and precision. The simulations can compute all aspects of the performance of instruments and can thus be used to optimize the use of existing equipment, design new instrumentation, and carry out virtual experiments. McStas is based on a unique design where an automatic compilation process translates high-level textual instrument descriptions into efficient ANSI-C code. This design makes it simple to set up typical simulations and also gives essentially unlimited freedom to handle more unusual cases.

This report constitutes the reference manual for McStas, and, together with the manual for the McStas components, it contains full documentation of all aspects of the program. It covers the various ways to compile and run simulations, a description of the meta-language used to define simulations, and some example simulations performed with the program.

Descriptors

Neutron Instrumentation; Monte Carlo Simulation; Software

Available on request from:
Information Service Department, Risø National Laboratory
(Afdelingen for Informationsservice, Forskningscenter Risø)
P.O. Box 49, DK–4000 Roskilde, Denmark
Phone +45 4677 4004, Telefax +45 4677 4013