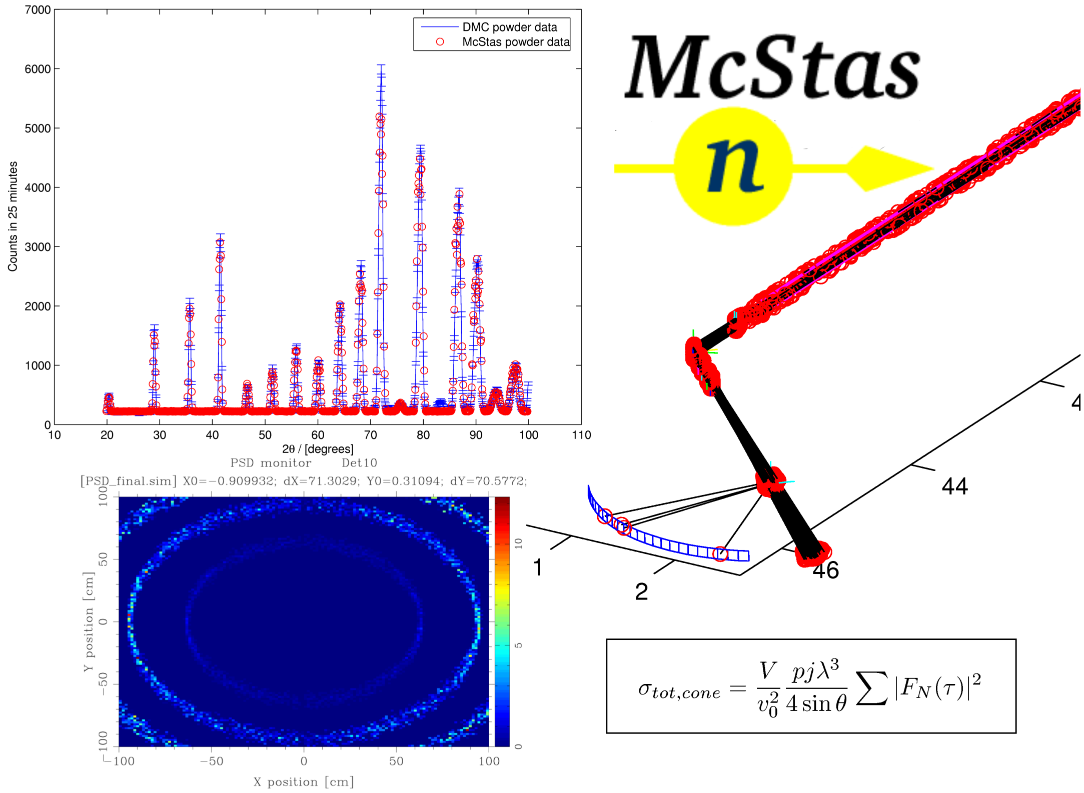
Risø–R–1416(EN)

RISØ

User and Programmers Guide to the Neutron Ray-Tracing Package McStas, Version 1.9

Peter Willendrup, Emmanuel Farhi, Kim Lefmann, Klaus Lieutenant and Kristian Nielsen



Risø National Laboratory, Roskilde, Denmark
November 2005

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 1.9, released September, 2005

The authors are:

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

Peter Kjær Willendrup `<peter.willendrup@risoe.dk>`
Materials Research Department, Risø National Laboratory, Roskilde, Denmark

Emmanuel Farhi `<farhi@ill.fr>`
Institut Laue-Langevin, Grenoble, France

Klaus Lieutenant `<lieutena@ill.fr>`
Institut Laue-Langevin, Grenoble, France

as well as authors who left the project:

Kristian Nielsen `<kn@sifira.dk>`
Materials Research Department, Risø National Laboratory, Roskilde, Denmark
Present address: Sifira A/S, Copenhagen, Denmark,

ISBN 87–550–3481–0 (Internet)
ISSN 0106–2840
Pitney Bowes Management Services Denmark A/S · Risø National Laboratory · 2005

Contents

Preface and acknowledgements

This document contains information on the Monte Carlo neutron ray-tracing program McStas version 1.9, building on 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 the C programming language and UNIX/Linux.

It is a pleasure to thank Prof. Kurt N. Clausen, PSI, for his continuous support to this project and for having initiated McStas in the first place. Essential support has also been given by Prof. Robert McGreevy, ISIS. Apart from the authors of this manual, also Per-Olof Åstrand, NTNU Trondheim, has contrubuted to the development of the McStas system. We have also benefited from discussions with many other people in the neutron scattering community, too numerous to mention here.

In case of errors, questions, or suggestions, do not hesitate to contact the authors at `mcstas@risoe.dk` or consult the McStas WWW home page [2]. A special bug/request reporting service is available [3].

If you **appreciate** this software, please subscribe to the `neutron-mc@risoe.dk` email list, send us a smiley message, and contribute to the package. We also encourage you to refer to this software when publishing results, with the following citations:

- K. Lefmann and K. Nielsen, Neutron News 10/3, 20, (1999).
- P. Willendrup, E. Farhi and K. Lefmann, Physica B, 350 (2004) 735.

McStas 1.9 contributors

Several people outside the core developer team have been contributing to McStas 1.9:

- Dickon Champion and Stuart Ansell, ISIS: `ISIS_moderator`. Dickon also contributed his model of the ISIS HET instrument plus a test instrument for the moderator component and assisted in various debugging.
- Uwe Filges, PSI: `Guide_tapering` and a model of the PSI FOCUS instrument
- Ross Stewart, ILL: New `Guide_curved`
- Chris Ling, ILL: ILL D9 instrument
- Lise Arleth, KVL Denmark: Co-author of the `Sans_spheres` component

- Geza Zsigmond helped a lot in porting his Fermi Chopper component from Vitess [4, 5] to McStas.

Thank you guys! This is what McStas is all about!

The McStas project has been supported by the European Union, initially through the XENNI program and the RTD "Cool Neutrons" program in FP4, In FP5, McStas was supported strongly through the "SCANS" program. Currently, in FP6, McStas is supported through the Joint Research Activity "MCNSI" under the Integrated Infrastructure Initiative "NMI3", see the WWW home pages [6, 7].

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, except for the NISP/MCLib program [8], 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, and includes a library of standard components with in total around 100 components. These enable to simulate all kinds of neutron scattering instruments (diffractometers, spectrometers, reflectometers, small-angle, back-scattering,...) for both continuous and pulsed sources.

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 the Institut Laue Langevin (ILL). The system is well tested and is supplied with several examples and with an extensive documentation. Besides this manual, a separate component manual exists.

1.1 Historical background

In the late 90'ies at Risø National Laboratory, simulation tools were urgently needed, not only to better utilize existing instruments (*e.g.* RITA-1 and RITA-2 [9–11]), but also to plan completely new instruments for new sources (*e.g.* the Spallation Neutron Source, SNS [12] and the planned European Spallation Source, ESS [13]). 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 simulating neutron instruments was needed. As there was no existing simulation software that would fulfill our needs, the McStas project was initiated. In addition, the ILL required an efficient and general simulation package in order to achieve renewal of its instruments and guides. A significant contribution to both the component library and the McStas kernel itself was early performed at the ILL and included in the package. ILL later became a part of the core McStas team.

1.2 Scientific background

What makes scientists happy? Probably collect good quality data, pushing the instruments to their limits, and fit that data to physical models. Among available measurement techniques, neutron scattering provides a large variety of spectrometers to probe structure and dynamics of all kinds of materials.

Unfortunately the neutron flux is often a limitation in the experiments. This then motivates instrument responsibles to improve the flux and the overall efficiency at the spectrometer positions, and even to design new machines. Using both analytical and numerical methods, optimal configurations may be found.

But achieving a satisfactory experiment on the best neutron spectrometer is not all. Once collected, the data analysis process raises some questions concerning the signal: what is the background signal ? What proportion of coherent and incoherent scattering has been measured ? Is is possible to identify clearly the purely elastic (structure) contribution from the quasi-elastic and inelastic one (dynamics) ? What are the contributions from the sample geometry, the container, the sample environment, and generally the instrument itself ? And last but not least, how does multiple scattering affect the signal ? Most of the time, the physicist will elude these questions using rough approximations, or applying analytical corrections [14]. Monte-Carlo techniques provide a mean to evaluate some of these quantities. The technicalities of Monte-Carlo simulation techniques are explained in detail in Chapter 4.

1.2.1 The goals of McStas

Initially, the McStas project had four main objectives that determined its design.

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, making debugging a major issue. 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 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.

1.3 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 component, e.g. a detailed description of a rotating velocity selector, 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.

1.4 Overview

The McStas system documentation consists of the following major parts:

- A short list of new features introduced in this McStas release appears in chapter 2
- Chapter 3 explains how to obtain, compile and install the McStas compiler, associated files and supportive software
- Chapter 4 concerns Monte Carlo techniques and simulation strategies in general
- Chapter 5 includes a brief introduction to the McStas system (section 5.1) as well a section (5.2) on running the compiler to produce simulations. Section 5.3 explains how to run the generated simulations. Running McStas on parallel computers require special attention and is discussed in section 5.6. 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 5.4.

- The McStas meta-language is described in chapter 6. This chapter also describes a set of library functions and definitions that aid in the writing of simulations. See appendix A for more details.
- The McStas component library contains a collection of well-tested, as well as user contributed, 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], but a short overview of these components is given in chapter 7 of the Manual.
- A collection of example instrument definitions is described in chapter 8 of the Manual.

A list of library calls that may be used in component definitions appears in appendix A, and an explanation of the McStas terminology can be found in appendix B of the Manual.. Plans for future extensions are presented on the McStas web-page [2] as well as in section 2.7.

Chapter 2

New features in McStas 1.9

This version of McStas implements both new features, as well as many bug corrections, especially for the Windows plateform. Bugs are reported and traced using the McStas Bugzilla system [3]. We will not present here an extensive list of corrections, and we let the reader refer to this bug reporting service for details. Only important changes are indicated here.

Of course, we can not garranty that the sofware is bullet proof, but we do our best to correct bugs, when they are reported.

2.1 General

The license covering the McStas package is the GNU General Pubclic License. See `http://www.gnu.org` for details.

2.2 Kernel

The following changes concern the 'Kernel' (i.e. the McStas meta-language and program). See the dedicated chapter in the *User manual* for more details.

- Instrument description files exchanged between Unix and Windows platforms could not always be compiled. This has been solved (bug #28).

2.3 Run-time

- Gravitation support has been validated. There was a bug revealed with long wavelength neutrons. Some components (see e.g. some guides) still do not support this option (bug #1).
- Monitor binning was not plotted correctly (not centered). This has been fixed for PGPLOT/McStas, Matlab, Scilab and Octave plotters (bugs #11, 12, 39).
- McStas can use MPI parallel processing, if it is properly installed.

- Alternatively, a basic 'grid' computing may be used to split scan steps onto a list of machines.
- Plenty of warning messages to guide the user in interpreting possibly wrong results have been added in the simulation output messages (bug #29, 43).
- Support for sample environments is now possible using the Isotropic_Sqw in Concentric mode. Refer to the `mcdoc` help page of the component.

2.4 Components and Library

We here list some of the new components (found in the McStas `lib` directory) which are detailed in the *Component manual*, also mentioned in the *Component Overview* of the *User Manual*.

- the Chopper_Fermi component has been **removed** has it produced notoriously wrong results. Rather use the new FermiChopper and Vitess_ChopperFermi components.
- Many minor bug corrections in: Arm (bug #10), Vitess_input/output (bug #11), Progress_bar (bug #14), Virtual_output (bug #18), Powder1, Powder2 (bugs #20,21), Monitor_nD (bug #17, 18), V_sample (bug #31)
- New components: Isotropic_Sqw, Vitess_ChopperFermi, PowderN

2.5 Documentation

As of McStas 1.9 an improved and updated tutorial has been integrated into to the graphical user interface of McStas.

The long awaited Component Manual has finally been written.

2.6 Tools, installation

A renewal of most McStas Tools, used to edit and display instrument or results, has been undertaken, aiming at proposing alternatives to the Perl+PerlTk+PGPLOT+PDL libraries.

Quite a lot of work was achieved in order to solve the installation problems that have been encoutered so far. A fully working McStas distribution now only requires a C compiler, perl, perl-Tk and one of Matlab, Scilab and (PGPLOT, perl-DL). The Plotlib Scilab library has been included in the package, and does not need to be installed separately anymore.

This has improved significantly the portability of McStas and thus simplified the installation of the package. Details about the installation and the available tools are given in chapter 3.

- Windows support improved (bugs #3, 4, 51)
- **Installation on Windows** systems *requires* to use exclusively **Perl 5.6**. Earlier versions will fail (bug #3).

- The most recent Scilab plotter (3.1) has some compatibility issues. McStas currently requires Scilab ¡= 3.0 (Unix and Windows)
- Many small bug corrections related to Matlab/Scilab capabilities (bugs #6, 7, 15, 16, 22, 24).

2.7 Future extensions

The following features are planned for the oncoming releases of McStas:

- Language extension 'JUMP' for enabeling loops, 'teleporting' etc. in instrument descriptions.
- Polarised components and magnetic field computation components.
- Optimize a set of parameters for a better flux and/or resolution on a given monitor.
- Better support for heterogeneous grid computing.
- Tools to optimize a set of parameters for a better flux and/or resolution on a given monitor.
- Concentric components (on the meta language level, currently the Isotropic_Sqw component has support for concentricness).
- Improvements to the NeXus data format (initial support is already included).
- Updates to mcresplot to support the Matlab and Scilab backends.

Chapter 3

Installing McStas

The information in this chapter is also available as a separate html/ps/pdf document in the `install_docs/` folder of your McStas installation package.

3.1 Licensing

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, and copies of the McStas source code may be distributed to others. New or modified component and instrument files may be shared by the user community, and the core team will be glad to include user contributions in the package.

3.2 Installing support Applications

To get a fully functional McStas installation, a few support applications are required. Essentially, you will need a C compiler, Perl and Perl-Tk, as well as a plotter such as Matlab, Scilab or PGPLOT. These should better be installed before McStas.

Under Windows, we recommand to avoid using spaces when specifying the location of packages during their installation.

3.2.1 C compiler

The C compiler is used for building the instrument simulation executable, which performs the actual simulation. It is called transparently by the mcgui and mcrun McStas tools.

- Win32: Bloodshed Dev-C++ (Win32). To install Bloodshed Dev-C++, download the installer package from

  http://www.bloodshed.net/dev/devcpp.html.

  When installed, add the `C:\Dev-Cpp\bin` directory to your PATH using

  'Start/Settings/Control Panel/System/Advanced/Environment Variables'.

- Unix/Linux: standard C compiler. Most Unix/Linux systems come with a system C compiler, usually named 'cc' or 'gcc'. In case this would not be the case, install such a compiler from Linux packages (RedHat, SuSe or Debian), or compile it from scratch.

3.2.2 Gui tools (Perl + Tk)

The McStas tools are written using the Perl language, with calls the perl-Tk library.

- Win32:
  - Get and install the ActivePerl package from http://downloads.activestate.com/ActivePerl/Windows/5.6/ActivePerl-5.6.1.638-MSWin32-x86.msi This package provides Perl and Perl-Tk.
  - You may need to update/install the Microsoft Windows Installer. Get it from Windows Support Downloads at http://www.microsoft.com/.
  - Make sure that during installation, you have requested to attach the .pl and .tcl extensions to Perl and Tcl, and the `C:\Perl\bin` is in the PATH.
- Unix/Linux:
  - Install Perl, Tcl/Tk and perl-Tk. Prebuilt packages exist for most Linux distributions and also most other Unix-like operating systems. You will find these packages at http://freshmeat.net/, http://www.rpmfind.net/linux/RPM/ (for SuSe and RedHat) and using Fink Commander (for Mac OS X). Some Linux like system do not provide perl-Tk as an RPM/Debian file. You will then have to compile it from the source code (with ./configure; make; make install).
  - Consult the McStas webpage at http://www.mcstas.org for updated links to the source code distributions.

3.2.3 Plotting backends (All platforms)

For plotting with McStas, different support packages can be used:

- PGPLOT/PDL (perl-PDL)/pgperl (perl-PGPLOT) (Unix only) - Binary builds of the packages exist for various Linux distributions (for instance Debian comes with prebuilt versions). Prebuilt versions also exist for some commercial Unix'es. Refer to distributor/vendor for documentation. The packages can also be built from source using some (in many cases much) effort. See the PGPLOT documentation for further details. You may need to define the `PGPLOT_DIR` variable to the location of the Pgplot library.

- Matlab (Some Unix/Win32) - refer to http://www.mathworks.com. Matlab licenses are rather costly, but discount programmes for university and research departments exist.

- Scilab (Unix/Win32/Mac...) - a free 'Matlab-like' package, available from http://www.scilab.org/download/ind
  IMPORTANT: McStas plotting only works properly with Scilab ver.$\leq$3.0

On Windows systems, Scilab should be installed in a directory such as `C:\Scilab` (do NOT use spaces in names) and you should then add the `C:\Scilab\bin` directory to your PATH using

'Start/Settings/Control Panel/System/Advanced/Environment Variables'.

3.3 Getting McStas

The McStas package is available in three different distribution packages, from the project website at http://www.mcstas.org, e.g.

- `mcstas-1.9-src.tar.gz`
  Source code package for building McStas on (at least) Linux and Windows 2000. This package should compile on most Unix platforms with an `ANSI-c` compiler. - Refer to section 3.4

- `mcstas-1.9-i686-unknown-Linux.tar.gz`
  Binary package for Linux systems, currently built on Debian GNU/Linux 3.0 'woody'. Should work on most Linux setups. - Refer to section 3.5

- `mcstas-1.9-i686-unknown-Win32.zip`
  Binary package for Win32 systems, currently built on Microsoft Windows 2000 professional, using the `gcc` 2.95 compiler from Bloodshed Dev-C++ 5 Beta 7 - Refer to section 3.6

3.4 Source code build

The McStas package is beeing co-developed for mainly Linux and Windows systems. However, the Linux build instructions below will work on most Unix systems, including Mac OS X). For an updated list of platforms on which McStas has been built, refer to the project website.

3.4.1 Windows build

- Start by unpacking the `mcstas-1.9-src.tar.gz` package using e.g. Winzip.
- Compile the McStas package using the `build.bat` script of the `mcstas-1.9` directory you just unpacked. Follow the on screen instructions.
- When the build has been done (e.g. `mcstas.exe` has been produced), proceed to install (Section 3.6).

3.4.2 Unix build

McStas uses autoconf to detect the system configuration and creates the proper Makefiles needed for compilation. On Unix-like systems, you should be able to compile and install McStas using the following steps:

1. Unpack the sources to somewhere convenient and change to the source directory:
   gunzip -c mcstas-1.9-src.tar.gz — tar xf -
   cd mcstas-1.9/

2. Configure and compile McStas:
   ./configure
   make

3. Install McStas (as superuser):
   make install

The installation of McStas in step 3 by default installs in the /usr/local/ directory, which on most systems requires superuser (root) privileges. To install in another directory, use the –prefix= option to configure in step 2. For example,

./configure –prefix=/home/joe

will install the McStas programs in /home/joe/bin/ and the library files needed by McStas in /home/joe/lib/mcstas/.

In case ./configure makes an incorrect guess, some environment variables can be set to override the defaults:

- The CC environment variable may be set to the name of the C compiler to use (this must be an ANSI C compiler). This will also be used for the automatic compilation of McStas simulations in mcgui and mcrun.
- CFLAGS may be set to any options needed by the compiler (eg. for optimization or ANSI C conformance). Also used by mcgui/mcrun.
- PERL may be set to the path of the Perl interpreter to use.

To use these options, set the variables before running ./configure. Eg.

setenv PERL /pub/bin/perl5
./configure

It may be necessary to remove configure's cache of old choices first:

rm -f config.cache

If you experience any problems, or have some questions or ideas concerning McStas, please contact peter.willendup@risoe.dk or the McStas mailing list at neutron-mc@risoe.dk.

You should try to make sure that the directory containing the McStas binaries (mcstas, gscan, mcdisplay, etc.) is contained in the PATH environment variable. The default directory is /usr/local/bin, which is usually, but not always, included in PATH. Alternatively, you can reference the McStas programs using the full path name, ie.

```
/usr/local/bin/mcstas my.instr
perl /usr/local/bin/mcrun -N10 -n1e5 mysim -f output ARG=42
```

```
perl /usr/local/bin/mcdisplay --multi mysim ARG=42
```

This may also be necessary for the front-end programs if the install procedure could not determine the location of the perl interpreter on your system.

If McStas is installed properly, it should be able to find the files it needs automatically. If not, you should set the MCSTAS environment variable to the directory containing the runtime files "mcstas-r.c" and "mcstas-r.h" and the standard components (*.comp). Use one of

```
MCSTAS=/usr/local/lib/mcstas; export MCSTAS # sh, bash
setenv MCSTAS /usr/local/lib/mcstas         # csh, tcsh
```

3.5 Binary install, Linux

Should be very easy, simply start from 'make install' in Section 3.4.2.

3.6 Binary install, Windows

- Start by unpacking the `mcstas-1.9-i686-unknown-Win32.zip` package using e.g. Winzip.
- Execute the `install.bat` installation script. Follow the on screen instructions.
- Set the required (see output of `install.bat`) environment variables using

  'Start/Settings/Control Panel/System/Advanced/Environment Variables'

  - `PATH` - Append e.g. `C:\mcstas\bin`
  - `MCSTAS` - Create it as e.g. `C:\mcstas\lib`

It is important that Perl is correctly installed to execute all the McStas tools (e.g. mcdoc.pl, mcrun.pl, mcgui.pl, ...). Create a shortcut of the `C:\mcstas\bin\mcgui.pl` on your Desktop (the icon should be a yellow dot). Whenever launched from the Windows Command window (cmd), you must specify the .pl extension to all McStas Perl script commands (e.g. 'mcrun.pl' and 'mcgui.pl', not 'mcrun' or 'mcgui') except for `mcstas` itself.

Actually, the modifications to your environnement variables should be

- `PATH` - Append e.g.
  `C:\mcstas\bin;c:\Program Files\Scilab-3.0\bin;C:\Perl\bin\;C:\Dev-Cpp\bin;`
- `MCSTAS` - Create it as e.g. `C:\mcstas\lib`

using menu item 'Start/Settings/Control Panel/System/Advanced/Environment Variables'. On some Windows systems, it may be necessary to logout/login (no need to restart the computer) to make these changes active for the whole session.

3.7 Finishing and Testing the McStas distribution

Once installed, you may check and tune the guessed configuration stored within file

- `MCSTAS\tools\perl\mcstas_config.perl` on Windows systems
- `MCSTAS/tools/perl/mcstas_config.perl` on Unix/Linux systems

where `MCSTAS` is the location for the McStas library.

The `examples` directory of the distribution contains a set of instrument examples. These are used for the McStas self test procedure, which is executed with

```
mcrun --test # mcrun.pl on Windows
```

This test takes a few minutes to complete, and ends with a short report on the installation itself, the simulation accuracy and the plotter check.

You should now be able to use McStas. For some examples to try, see the examples/ directory. Start 'mcgui' (mcgui.pl on Windows), and select one of the examples in the 'Neutron Sites' menu.

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 transforms under a Monte Carlo choice and specialize this to the concept of direction focusing. Finally, we present a way of generating random numbers with arbitrary distributions.

4.1 Neutron spectrometer simulations

Neutron scattering instruments are built as a series of neutron optics elements. Each of these elements modifies the beam characteristics (e.g. divergence, wavelength spread, spatial and time distributions) in a way which may be modeled through analytical methods, for simplified neutron beam configurations. This stands for individual elements such as guides [15, 16], choppers [17, 18], Fermi choppers [19, 20], velocity selectors [21], monochromators [22–25], and detectors [26–28]. In the case of selected neutron instrument parts, one may use efficiently the so-called acceptance diagram theory [16, 29, 30] within which the neutron beam distributions are considered to be homogeneous or gaussian. However, the concatenation of a high number of neutron optical elements, which indeed constitute real instruments, brings additional complexity by introducing strong correlations between neutron beam parameters: divergence and position - which is the basis of the acceptance diagram method - but also wavelength and time. The usual analytical methods (phase-space theory...) then reach their limit of validity in the description of the resulting fine effects.

In principle, computing individual neutron event propagation at each instrument part, using analytical and numerical models, is not such a hard task. The use of probablilities is common to describe microscopic physical processes. Integrating all these events over the propagation path will result in an estimation of measurable quantities characterizing the neutron instrument. Moreover, using variance reduction (*e.g.* importance sampling), whenever possible, will both speed-up the computation and achieve a better accuracy. What we just sketched is nothing else than the basis of the Monte-Carlo (MC) method

[31], applied to neutron ray-tracing instrumentation.

4.1.1 Monte Carlo ray tracing simulations

Mathematically, the Monte-Carlo method is an application of the law of large numbers [31, 32]. Let $f(u)$ be a finite continuous integrable function of parameter u for which an integral estimate is desirable. The discrete statistical mean value of f (computed as a series) in the uniformly sampled interval $a < u < b$ converges to the mathematical mean value of f over the same interval.

$$\lim_{n\to\infty} \frac{1}{n} \sum_{i=1, a\le u_i \le b}^{n} f(u_i) = \frac{1}{b-a} \int_a^b f(u)du \tag{4.1}$$

In the case were the u_i values are regularly sampled, we come to the well known midpoint integration rule. In the case were the u_i values are randomly (but regularly) sampled, this is the Monte-Carlo integration technique. As random generators are not perfect, we rather talk about *quasi*-Monte-Carlo technique. We encourage the reader to refer to James [31] for a detailed review on the Monte-Carlo method.

Although early implementations of the method for neutron instruments used *home-made* computer programs (see e.g. papers by J.R.D. Copley, D.F.R. Mildner, J.M. Carpenter, J. Cook), more general packages have been designed, providing models for most parts of the simulations. These present existing packages are: NISP [33], ResTrax [34], McStas [1, 2, 35], Vitess [4, 5], and IDEAS [36]. Their usage usually covers all types of neutron spectrometers, most of the time through a user-friendly graphical interface, without requiring programming skills.

The neutron ray-tracing Monte-Carlo method has been used widely for *e.g.* guide studies [29, 37, 38], instrument optimization and design [39, 40]. Most of the time, the conclusions and general behaviour of such studies may be obtained using the classical analytical approaches, but accurate estimates for the flux, the resolutions, and generally the optimum parameter set, benefit advantageously from MC methods.

Recently, the concept of virtual experiments, *i.e.* full simulations of a complete neutron experiment, has been suggested as the main goal for neutron ray-tracing simulations. The goal is that simulations should be of benefit to not only instrument builders, but also to users for training, experiment planning, diagnostics, and data analysis.

4.2 The neutron weight

A totally realistic semi-classical simulation will require that each neutron is at any time either present or lost. In many instruments, only a very 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 never hit the detector.

An important way of speeding up calculations is to introduce a neutron "weight factor" for each simulated neutron ray and to adjust this weight according to the path of the ray. If *e.g.* the reflectivity of a certain optical component is 10%, and only reflected neutrons ray 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 ray after passage of the whole instrument becomes the product of all contributions

$$p = p_n = p_0 \prod_{j=1}^{n} \pi_j. \tag{4.2}$$

For convenience, the value of p is updated (within each component) during the simulation.

Simulation by weight adjustment is performed whenever possible. This includes

- Transmission through filters.
- 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.
- Passage of a continuous beam through a chopper.
- Scattering from samples.

4.2.1 Statistical errors of non-integer counts

In a typical simulation, the result will consist of a count of neutrons histories ("rays") with different weights. The sum of these weights is an estimate of the mean number of neutrons hitting the monitor (or detector) per second in a "real" experiment. One may write the counting result as

$$I = \sum_i p_i = N\overline{p}, \tag{4.3}$$

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 accuracy of the simulation.

To give an 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 $\sigma(N) = \sqrt{N}$ [1]. Hence, the statistical uncertainty of the observed intensity becomes

$$\sigma(I) = \sqrt{N}\overline{p} = I/\sqrt{N}, \tag{4.4}$$

as is used in real neutron experiments (where $\overline{p} \equiv 1$). For a better approximation we return to Eq. (4.3). Allowing variations in both N and $\overline{p}$, we calculate the variance of the resulting intensity, assuming that the two variables are independent:

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

[1] 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.

Assuming that N follows a normal distribution, we reach $\sigma^2(N)\overline{p}^2 = N\overline{p}^2$. Further, 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

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

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

$$\sigma^2(I) = \frac{N}{N-1}\left(\sum_i p_i^2 - \overline{p}^2\right). \tag{4.7}$$

For almost any positive value of N, this is very well approximated by the simple expression

$$\sigma^2(I) \approx \sum_i p_i^2. \tag{4.8}$$

As a consistency check, we note that for all p_i equal, this reduces to eq. (4.4)

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 $M_2 = \sum_i p_i^2$.

4.3 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 ray 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 of making this particular choice equals the actual physical properties we like to model.

Let us follow up on the simple example of transmission. The probability of transmitting the real neutron is T, but we make the Monte Carlo choice of transmitting the neutron ray each time: $f_{\rm MC} = 1$. This must be reflected on the choice of weight multiplier π_j gven by the master equation

$$f_{\rm MC}\pi_j = P. \tag{4.9}$$

This probability rule is general, and holds also if, e.g., it is decided to transmit only half of the rays ($f_{\rm MC} = 0.5$). An important different example is elastic scattering from a powder sample, where the Monte-Carlo choices are the particular powder line to scatter from, the scattering position within the sample and the final neutron direction within the Debye-Scherrer cone.

4.3.1 Direction focusing

An important application of weight transformation is direction focusing. Assume that the sample scatters the neutron rays in many directions. In general, only neutron rays 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 other 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 $\mathbf{\Delta\Omega}$, only these solid angles are used for the Monte Carlo choice of scattering direction. According to Eq. (4.9), the weight factor will then have to be changed by the amount $\pi_j = |\mathbf{\Delta\Omega}|/(4\pi)$. One thus ensures that the mean simulated intensity is unchanged during a "correct" direction focusing, while a too narrow focusing will result in a lower (*i.e.* wrong) intensity, since we cut neutrons rays that should have counted.

4.4 Adaptive sampling

Another strategy to improve sampling in simulations is adaptive importance sampling, where McStas during the simulations will determine the most interesting directions and gradually change the focusing according to that. Implementation of this idea is found in the **Source_adapt** and **Source_Optimizer** components.

Chapter 5

Running McStas

This chapter describes usage of the McStas simulation package. Refer to Chapter 3 for installation instructions. In case of problems regarding installation or usage, the McStas mailing list [2] or the authors should be contacted.

Important note for Windows users: It is a known problem that some of the McStas tools do not support filenames / directories with spaces. We are working on a more general approach to this problem, which will hopefully be solved in the next release. Also, there are known issues for McStas with recent Perl versions for Windows. We recommand to use ActiveState **Perl 5.6**.

To use McStas, an instrument definition file describing the instrument to be simulated must be written. Alternatively, an example instrument file can be obtained from the `examples/` directory in the distribution or from another source.

The structure of McStas is illustrated in Figure 5.1.

The input files (instrument and component files) are written in the McStas meta-language and are edited either by using your favourite editor or by using the built in editor of the graphical user interface (`mcgui`).

Next, the instrument and component files are compiled using the McStas compiler, relying on built in features from the FLEX and Bison facilities to produce a C program.

The resulting C program can then be compiled with a C compiler and run in combination with various front-end programs for example to present the intensity at the detector as a motor position is varied.

The output data may be analyzed and visualized in the same way as regular experiments by using the data handeling and visualisation tools in McStasbased on Perl and Matlab, Scilab [41] or PGPLOT. Further data output formats including IDL and XML are available, see section 5.5.

5.1 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.

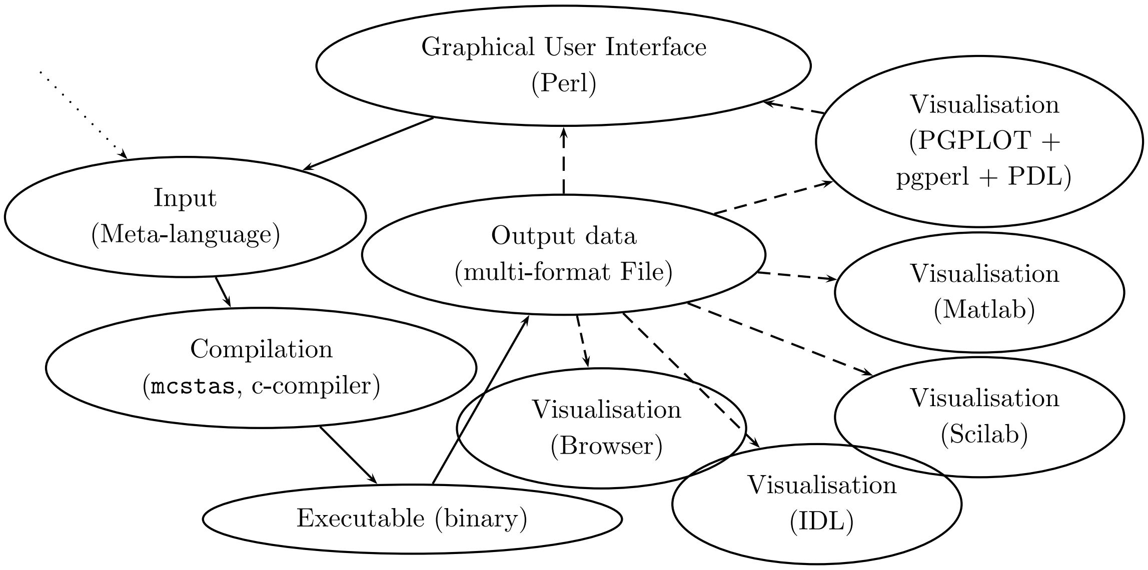


Figure 5.1: An illustration of the structure of McStas .

To start the graphical user interface of McStas, run the command `mcgui` (`mcgui.pl` on Windows). This will open a window with a number of menus, see figure 5.2.

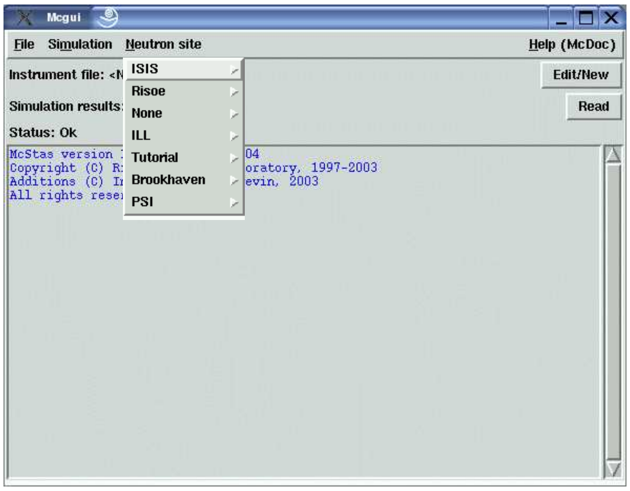


Figure 5.2: The graphical user interface `mcgui`.

To load an instrument, select “Tutorial” from the “Neutron site” menu and open the file `vanadium_example`. Next, check that the current plotting backend setting (select “Choose backend” from the “Simulation” menu) corresponds to your system setup. The default setting can be adjusted as explained in Chapter 3

- by editing the `tools/perl/mcstas_config.perl` setup file of your installation
- by setting the `MCSTAS_FORMAT` environment variable.

Next, 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. Depending on your chosen plotting backend, the presented graphics will resemble one of those shown in figure 5.3. When using the Scilab or Matlab backends, full 3D view

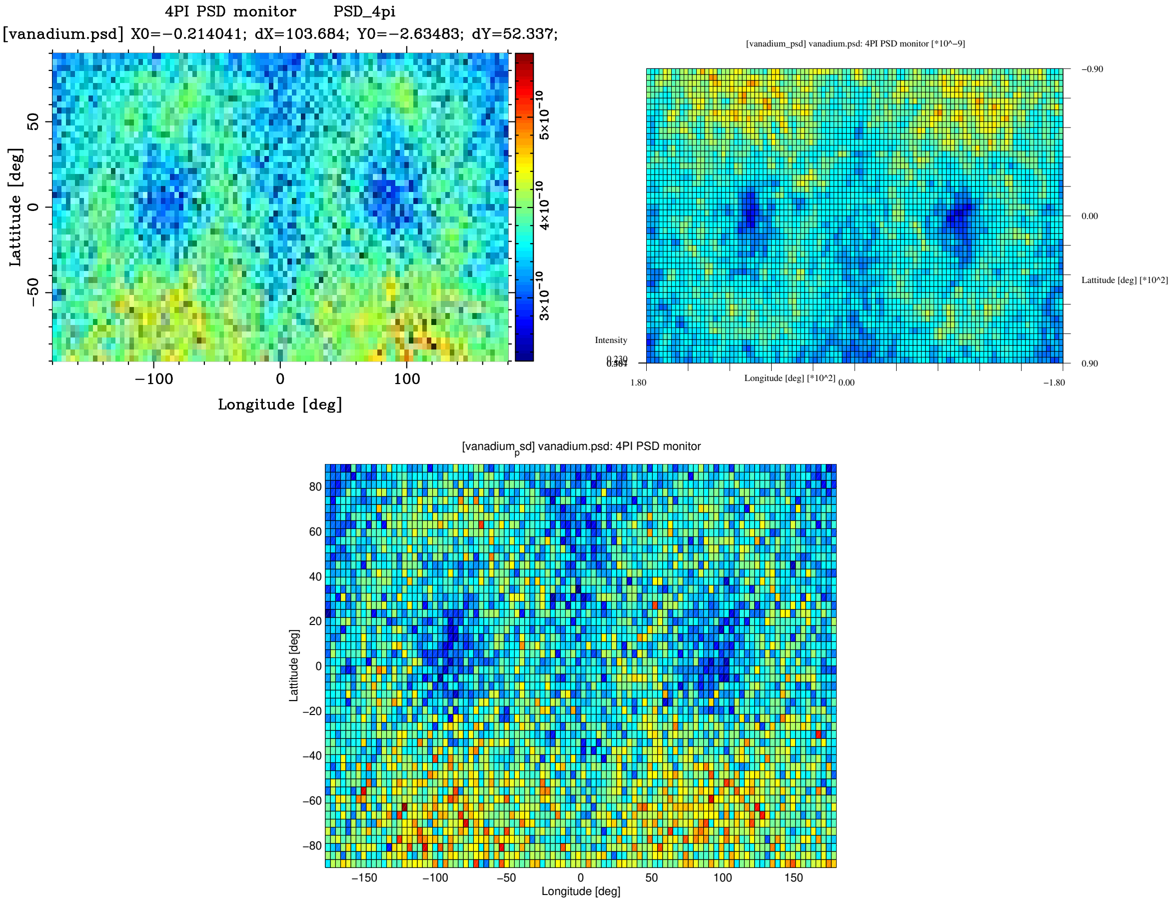


Figure 5.3: Output from `mcplot` with PGPLOT, Scilab and Matlab backends

of plots and different display possibilities are available. Use the attached McStas window menus to control these. Features are quite self explanatory. For other options, execute `mcplot --help` (`mcplot.pl --help` on windows) to get help.

To visualize or 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. Depending on your chosen plotting backend, the presented graphics will resemble one of those shown in figures 5.4-5.6.

For a slightly longer gentle introduction to McStas, see the McStas tutorial (available from [2]), and as of version 1.9 built into the `mcgui` help menu. For more technical details, read on from section 5.2

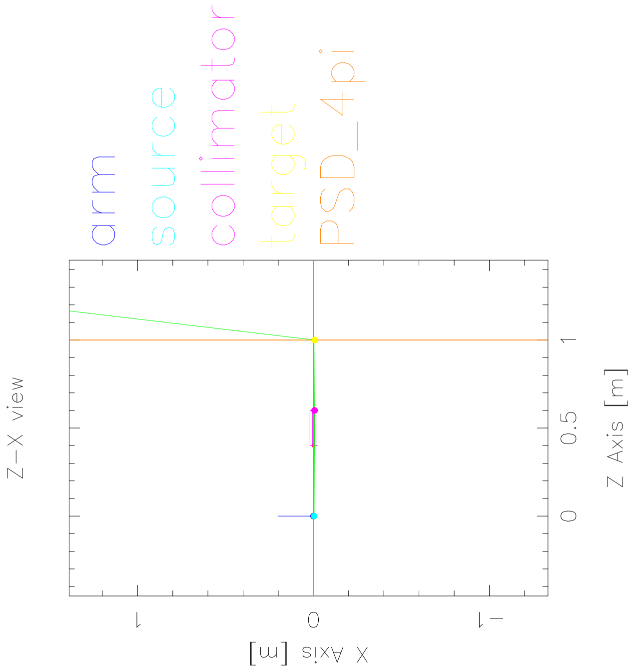


Figure 5.4: Output from `mcdisplay` with PGPLOT backend. The left mouse button starts a new neutron ray, the middle button zooms, and the right button resets the zoom. The Q key quits the program.

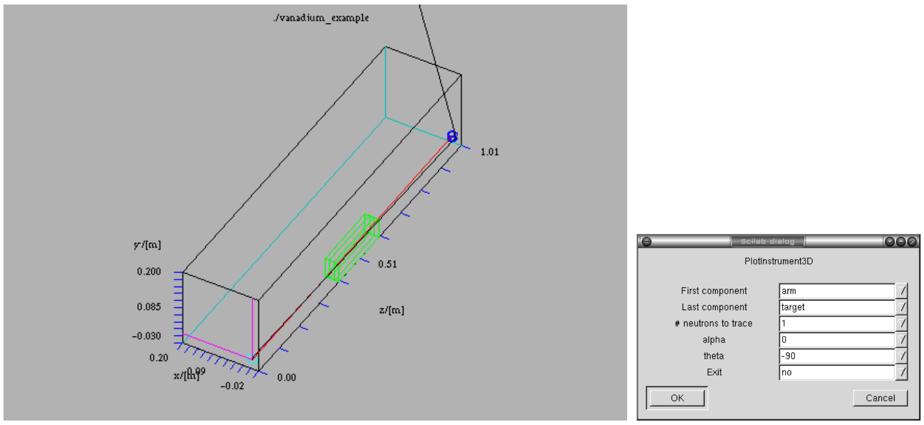


Figure 5.5: Output from `mcdisplay` with Scilab backend. Display can be adjusted using the dialogbox (right).

5.1.1 New releases of McStas

Releases of new versions of a software package 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 .

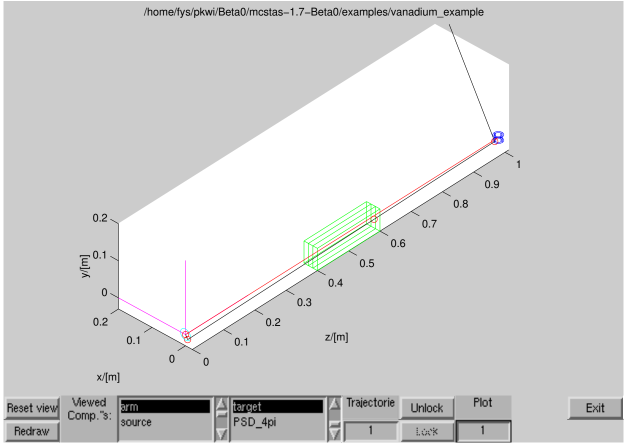


Figure 5.6: Output from `mcdisplay` with Matlab backend. Display can be adjusted using the window buttons.

- The versions 1.9.x will possibly contain bug fixes and minor 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 1.9.x are also listed on the web, and new versions may be released quite frequently when it is requested by the user community.

- The version 1.10 will contain important new features and an updated manual. It will be released in 2006.

5.2 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 5.4.1) or `mcrun` (section 5.4.2). 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.

5.2.1 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 ray 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 5.4.3.

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 ...
```

Finally, the `--verbose` option will list the components and libraries beeing included in the instrument.

5.2.2 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` on Unix-like systems and `C:\mcstas\lib` on Windows systems, but it can be changed to something else, see section 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. For Windows Users, you should define the `MCSTAS` from the menu 'Start/Settings/Control Panel/System/Advanced/Environment Variables' by creating `MCSTAS` with the value `C:\mcstas\lib`

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.

5.2.3 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` as well as other shared libraries from the McStas distribution.

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

5.2.4 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).
- SGI machines running Irix with the options `-Ofast -o32 -w`

Optimization flags will typically result in a speed improvement by a factor about 3, but the compilation of the instrument may be 5 times slower.

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. Also, optimizations are higly time and memory consuming during compilation, and thus may fail when dealing with large instrument descriptions (e.g. more that 100 elements). The compilation process is simplified when using components of the library making use of shared libraries (see `SHARE` keyword in chapter 6). Refer to section 5.3.4 for other optimization methods.

5.3 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 ".", 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. Default instrument parameter values (see section 6.3), if any, will be indicated and entered when hitting the `Return` key. This way of running McStas will only give data for one spectrometer setting which is normally sufficient for *e.g.*, time-of-flight, SANS or powder instruments, but not for *e.g.* reflectometers or triple-axis spectrometers 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 (which should probably be avoided when saving in binary format, see below).

The complete list of options and arguments accepted by McStas simulations appears in Table 5.1.

5.3.1 Choosing an output data file format

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 `--data-only` or `-a` option.

The format of the output files from McStas simulations is described in more detail in section 5.5. It may be chosen either with `--format=FORMAT` for each simulation or globally by setting the MCSTAS_FORMAT environment variable. The available format list is obtained using the `name.out --help` option, and shown in Table 5.2. McStas can presently generate many formats, including the original McStas/PGPLOT and the new Scilab and Matlab formats. All formats, except the McStas/PGPLOT, may eventually support binary files, which are much smaller and faster to import, but are platform dependent. The simulation data file extensions are appended automatically, depending on the format, if the file names do not contain any. Binary files are particularly recommanded for the IDL format (e.g. `--format=IDL_binary`), and the Matlab and Scilab format when handling large detectors (e.g. more than 50x50 bins). For example:

```
./vanadium_example.out ROT=90 --format="Scilab_binary"
```

or more generally (for bash/Bourne shell users)

```
export MCSTAS_FORMAT="Matlab"
./vanadium_example.out ROT=90
```

It is also possible to create and read *Vitess* and *Tripoli4* neutron event files using components

- `Vitess_input` and `Vitess_output`
- `Tripoli_input` and `Tripoli_output`

Additionally, adding the `raw` keyword to the FORMAT will produce raw $[N, p, p^2]$ data sets instead of $[N, p, \sigma]$ (see Section 4.2.1). The former representation is fully additive, and thus enables to add results from separate simulations (e.g. when using a computer Grid).

5.3.2 Basic import and plot of results

The previous example will result in a `mcstas.m` file, that may be read directly from Matlab (using the *sim file* function)

```
matlab> s=mcstas;
matlab> s=mcstas('plot')
```

The first line returns the simulation data as a single structure variable, whereas the second one will additionally plot each detector separately. This also equivalently stands for Scilab (using the `get_`*sim file* function, the 'exec' call is required in order to compile the code)

```
scilab> exec('mcstas.sci', -1); s=get_mcstas();
scilab> exec('mcstas.sci', -1); s=get_mcstas('plot')
```

and for IDL

```
idl> s=mcstas()
idl> s=mcstas(/plot)
```

See section 5.4.4 for an other way of plotting simulation results using the `mcplot` front-end.

When choosing the HTML format, the simulation results are saved as a web page, whereas the monitor data files are saved as VRML files, displayed within the web page.

5.3.3 Interacting with a running simulation

Once the simulation has started, it is possible, under Unix, Linux and Mac OS X systems, to interact with the on-going simulation.

McStas attaches a signal handler to the simulation process. In order to send a signal to the process, the process-id *pid* must be known. Users may look at their running processes with the Unix 'ps' command, or alternatively process managers like 'top' and 'gtop'. If a *file.out* simulation obtained from McStas is running, the process status command should output a line ressembling

`<user>` *13277* `7140 99 23:52 pts/2    00:00:13` *file.out*

where `user` is your Unix login. The *pid* is there '13277'.

Once known, it is possible to send one of the signals listed in Table 5.3 using the 'kill' unix command (or the functionalities of your process manager), e.g.

```
kill -USR2 13277
```

This will result in a message showing status (here 33 % achieved), as well as the position in the instrument of the current neutron.

```
# McStas: [pid 13277] Signal 12 detected SIGUSR2 (Save simulation)
# Simulation: file (file.instr)
# Breakpoint: MyDetector (Trace) 33.37 % (  333654.0/ 1000000.0)
# Date      : Wed May  7 00:00:52 2003
# McStas: Saving data and resume simulation (continue)
```

`-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. (1e6)
`-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*. Avoid when using binary formats.
`-a` `--data-only`	Do not put any headers in the data files.
`-h` `--help`	Show a short help message with the options accepted, available formats and 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).
`-g` `--gravitation`	This option toggles the gravitation (approximation) handling for the whole neutron propagation within the instrument. May produce wrong results depending on the components.
`--format=`*FORMAT*	This option sets the file format for result simulation and data files.
`--format_data=` *FORMAT*	This option sets the file format for result data files from monitors. This enables to have simulation files in one format (e.g. HTML), and monitor files in an other format (e.g. VRML).
param=*value*	Set the value of an instrument parameter, rather than having to prompt for each one.

Table 5.1: Options accepted by McStas simulations. For options specific to MPI and parallel computing, see section 5.6.

`McStas PGPLOT`	.sim	Original format for PGPLOT plotter (may be used with -f and -d options)
`Scilab`	.sci	Scilab format (may be used with -f and -d options)
`Scilab_binary`		Scilab format with external binary files (may be used with -d option). (-a option implicitly set)
`Matlab`	.m	Matlab format (may be used with -f and -d options)
`Matlab_binary`		Matlab format with external binary files (may be used with -d option). (-a option implicitly set)
`Octave`	.m	Octave format (may be used with -f and -d options)
`Octave_binary`		Octave format with external binary files (may be used with -d option). (-a option implicitly set)
`IDL`	.pro	IDL format. *Must* be used with -f option.
`IDL_binary`		IDL format with external binary files (may be used with -d option). (-a option implicitly set)
`XML`	.xml	XML/NeXus format (may be used with -f and -d options).
`HTML`	.html	HTML format (generates a web page, may be used with -f and -d options). Data files are saved as VRML objects.
`VRML`	.wrl	Virtual Reality file format for data files. Simulation files are not saved properly, and the HTML format should be used preferably.
Tripoli		Tripoli 4 neutron events file format. Use `Virtual_tripoli4_input`/`Virtual_tripoli4_output` components.
Vitess		Vitess neutron events file format. Use `Vitess_input`/`Vitess_output` components.
McStas events		McStas event files in text or binary format. Use `Virtual_input`/`Virtual_output` components.

Table 5.2: Available formats supported by McStas simulations.

`USR1`	Request informations (status)
`USR2, HUP`	Request informations and performs an intermediate saving of all monitors (status and save). This triggers the execution of all `SAVE` sections (see chapter 6).
`INT, TERM`	Save and exit before end (status)

Table 5.3: Signals supported by McStas simulations.

followed by the list of detector outputs (integrated counts and files). Finally, sending a `kill 13277` (which is equivalent to `kill -TERM 13277`) will end the simulation before the initial 'ncount' preset.

A typical usage example would be, for instance, to save data during a simulation, plot or analyze it, and decide to interupt the simulation earlier if the desired statistics has been achieved. This may be done automatically using the `Progress_bar` component.

Whenever simulation data is generated before end (or the simulation is interupted), the 'ratio' field of the monitored data will provide the level of achievement of the computation (for instance '3.33e+05/1e+06'). Intensities are then usually to be scaled accordingly.

Additionally, any system error will result in similar messages, giving indication about the occurence of the error (component and section). Whenever possible, the simulation will *try* to save the data before ending. Most errors appear when using a newly written component, in the `INITIALIZE`, `TRACE` or `FINALLY` sections. Memory errors usually show up when C pointers have not been allocated/unallocated before usage, whereas mathematical errors are found when, for instance, dividing by zero.

5.3.4 Optimizing simulations

There are various ways to speedup simulations

- Optimize the compilation of the instrument, as explained in section 5.2.4.
- Execute the simulation in parallel on a computer grid or a cluster (with MPI) as explained in section 5.6.
- Divide simulation into parts using a file for saving or generating neutron events. This way, a guide may be simulated only once, saving the neutron events getting out from it as a file, which is being read quickly by the second simulation part. Use the Virtual_input and Virtual_output components for this technique.
- Use source optimizers like the components Source_adapt or Source_Optimizer. Such component may sometimes not be very efficient, when no neutron importance sampling can be achieved, or may even sometimes alter the simulation results. Be careful and always check results with a (shorter) non-optimized computation.
- Complex components usually take into account additional small effects in a simulation, but are much longer to execute. Thus, simple components should be prefered whenever possible, at least to start with.

Additionally, the user may wish to optimize the parameters of a simulation (e.g. find the optimal curvature of a monochromator, or the best geometry of a given component). The user should write a function script or a program that

- inputs the simulation parameters, which are usually numerical values such as TT in the `prisma2` instrument from the `examples` directory of the package.
- builds a command line from these parameters.
- execute that command, and waits until the end of the computation.
- reads the relevant data from the monitors.
- outputs a simulation quality measurement from this data, usually the integrated counts or some peak width.

For instance, for the `prisma2` instrument we could write a function for Matlab (see section 5.5 for details about the Matlab data format) in order to study the effects of the TT parameter:

```
function y = instr_value(p)
  TT = p(1);     % p may be a vector/matrix containing many parameters
  syscmd = [ 'mcrun prisma2.instr -n1e5 TT=' num2str(TT) ...
             ' PHA=22 PHA1=-3 PHA2=-2 PHA3=-1 PHA4=0 PHA5=1' ...
             ' PHA6=2 PHA7=3 TTA=44 --format="Matlab binary"' ];
  system(syscmd); path(path) % execute simulation, and rehash files
  s = mcstas;      % get the simulation data, and the monitor data
  s = s.prisma2.m_mcstas.detector.prisma2_tof.signal;
  eval(s);         % we could also use the 'statistics' field
  y = -Mean;       % 'value' of the simulation
```

Then a numerical optimization should be available, such as those provided with Matlab, Scilab, IDL, and Perl-PDL high level languages. In this example, we may wish to maximize the `instr_value` function value. The `fminsearch` function of Matlab is a minimization method (that's why we have a minus sign for y value), and:

```
matlab> TT = fminsearch('instr_value', -25)
```

will determine the best value of TT, starting from -25 estimate, in order to minimize function `instr_value`, and thus maximize the mean detector counts.

The choice of the optimization routine, of the simulation quality value to optimize, the initial parameter guess and the simulation length all have a large influence on the results. Be cautious and wise when interpreting the optimal guess.

5.4 Using simulation front-ends

McStas includes a number of front-end programs that extend the functionality of the simulations. A front-end program is an interface between the user and the simulations, running the simulations and presenting the output in various ways to the user.

The list of available McStas front-end programs may be obtained from the `mcdoc --tools` command:

```
McStas Tools
   mcstas        Main instrument compiler
   mcrun         Instrument maker and execution utility
   mcgui         Graphical User Interface instrument builder
   mcdoc         Component library documentation generator/viewer
   mcplot        Simulation result viewer
   mcdisplay     Instrument geometry viewer
   mcresplot     Instrument resolution function viewer
   mcstas2vitess McStas to Vitess component translation utility
   mcconvert     Matlab <-> Scilab script conversion tool
When used with the -h flag, all tools display a specific help.
SEE ALSO: mcstas, mcdoc, mcplot, mcrun, mcgui, mcresplot, mcstas2vitess
DOC:      Please visit http://www.mcstas.org
```

An extended set of front-end programs is planned for future versions of McStas, including a NeXus data format option [42].

5.4.1 The graphical user interface (mcgui)

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 (mcgui.pl on Windows) program may optionally be given the name of an instrument file.

When the front-end is started, a main window is opened (se figure 5.2). 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 with McStas syntax highlighting 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.

Spawn editor This starts the editor defined in the environment variable `VISUAL` or `EDITOR` on the current instrument file. It is also possible to start an external editor manually; in any case `mcgui` will recompile instrument definitions as necessary based on the modification dates of the files on the disk.

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 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 spawns a load dialogue to load a set of results.

Configuration options selection of plotting backend (PGPLOT/Matlab/Scilab). Opens the choose backend dialog shown in figure 5.7. All formats are chosen as full text, but a 'Use binary files' option is possible. Additionally, the editor to use to list instrument descriptions can be chosen. The HTML/VRML data files save results as a web-page document requiring a VRML viewer to display the data files as a 3D surface.

The **Neutron Site** menu contains a list of template/example instruments as found in the McStas library, sorted by neutron site. When selecting one of these, a local copy of the instrument description is transfered to the active directory (so that users have modification rights) and loaded. One may then view its source (Edit) and use it directly for simulations/trace (3D View).

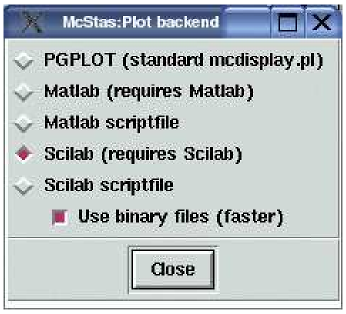


Figure 5.7: The choose backend dialog in `mcgui`.

The **Help** menu has the following features, through use of `mcdoc` and a web browser. To customize the used web browser, set the `BROWSER` environment variable. If `BROWSER` is not set, `mcgui` uses `netscape` on Unix and the default browser on Windows.

McStas User manual calls `mcdoc --manual`, brings up the local pdf version of this manual, using a web browser.

McStas Component manual calls `mcdoc --comp`, brings up the local pdf version of the component manual, using a web browser.

Component library index displays the component documentation using the component `index.html` index file.

McStas web page calls `mcdoc --web`, brings up the McStas website in a web browser.

Tutorial opens the McStas tutorial for a quick start.

Current instrument info generates a description web-page of the current edited instrument.

Test McStas installtion launches a self test procedure to check that the McStas package is installed properly, generates accurate results, and may use the plotter to display the results.

Generate component index (re-)generates locally the component `index.html`.

The run dialog

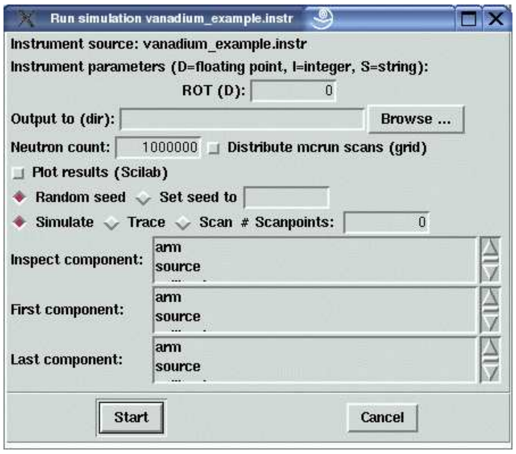


Figure 5.8: The run dialog in `mcgui`.

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 5.3 for details). It also allows to run the `mcdisplay` (section 5.4.3) and `mcplot` (section 5.4.4) front-ends together with the simulation.

The meaning of the different fields is as follows:

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. For parameter scans, enter the minimum and maximum values to scan, separated by a comma, e.g. `1,10` and do not forget to set the **# Scanpoints** to more than 1.

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

Force Forces to overwrite existing data files

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

Gravity Activates gravitation handling. Some components may not handle it properly, and produce wrong results.

Distribute mcrun scans (grid) activates the distribution of the scan steps over a computer grid, using an `ssh` mechanism. See section 5.6 on parallel computing for more informations about how to define and access the grid. This options is only visible if MPI/ssh is available.

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

Format The data file format can be changed the same way as in the "Simulation/Configuration options".

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 (3D) selects between running the simulation normally, or using the `mcdisplay` front-end to view the instrument in 3D and trace the neutrons individually inside components.

 # Scanpoints sets the number of times the simulation has to repeated to cover a parameter scan. If no parameter scan is specified, the simulation is just repeated.

Inspect component will trace only neutron trajectories that reach a given component (e.g. sample or detector).

First component seletcs the first component to plot (default is first) in order to define a region of interest.

Last component seletcs the last component to plot (default is first) in order to define a region of interest.

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

This dialog only shows-up when using the McStas/PGPLOT plotter. Other plotters have attached menus to control graphics and file exports.

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. Has the same effect as 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.

Colour GIF creates a colour GIF 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 5.9). See section 6.3 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 6.4.11 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 and Perl/Tk must be properly installed on the system. Additionally, if the McStas/PGPLOT back-end is used for data format, PGPLOT, PgPerl, and PDL will be required. It may be necessary to set the `PGPLOT_DIR` and `PGPLOT_DEV` environment variable; consult the documentation for PGPLOT on the local system in case of difficulty.

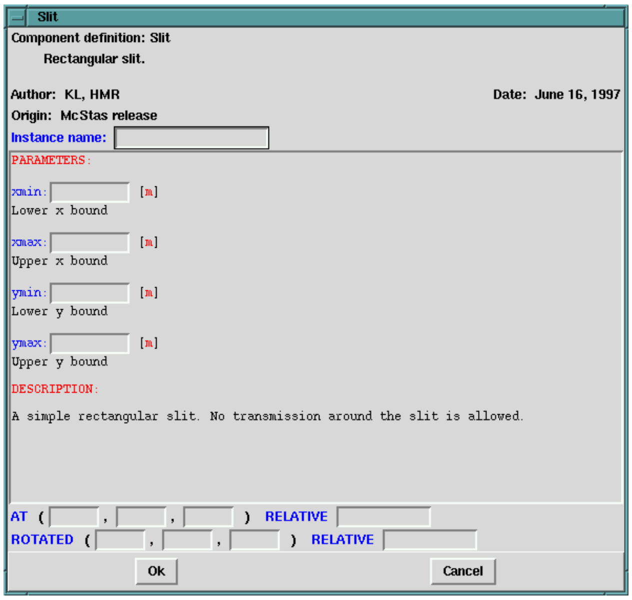


Figure 5.9: Component parameter entry dialog.

5.4.2 Running simulations with automatic compilation (mcrun)

The `mcrun` front-end (mcrun.pl on Windows) 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.

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 5.3, with the following extensions:

- The `-c` or `--force-compile` option may be used to force 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 by 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 integrated intensity and estimated error in all monitors. Additionally, a file `mcstas.sci` (when using Scialb format) 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 5.4.4.

- 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 --test` command will test your McStas installation, accuracy and plotter.

The `-h` option will list valid options. The `mcrun` front-end requires a working installation of Perl to run.

5.4.3 Graphical display of simulations (mcdisplay)

The front-end `mcdisplay` (mcdisplay.pl on Windows) 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.

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

`mcdisplay sim` *args ...*

where `sim` is the name of either the instrument source *sim*`.instr` or the simulation program *sim*`.out` generated with McStas, and *args ...* are the normal command line arguments for the simulation, as explained above. The `-h` option will list valid options.

The drawing back-end program may be selected among PGPLOT, Matlab and Scilab using either the -p*PLOTTER* option or using the current `MCSTAS_FORMAT` environment variable. For instance, calling

```
mcdisplay -pScilab ./vanadium_example.out ROT=90
```

or (`csh`/`tcsh` syntax)

```
setenv MCSTAS_FORMAT Scilab
mcdisplay ./vanadium_example.out ROT=90
```

will output graphics using Scilab. The `mcdisplay` front-end can also be run from the `mcgui` front-end. Examples of plotter appearence for `mcdisplay` is shown in figures 5.4-5.6.

McStas/PGPLOT back-end 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* ...

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.

The `mcdisplay` front-end will then require the Perl, the PGPLOT, and the PGPerl packages to be installed. It may be necessary to set the `PGPLOT_DIR` and `PGPLOT_DEV` environment variable; consult the documentation for PGPLOT on the local system in case of difficulty.

Matlab and Scilab back-ends A 3D view of the instrument, and various operations (zoom, export, print, trace neutrons, ...) is available from dedicated Graphical User Interfaces. The `--inspect` option may be used (see previous paragraph), as well as the `--first` and `--last` options to specify a region of interest. **Note:** The Scilab plotter for *Windows* does not allow to rotate interactively the 3D view, and you are to use the *alpha* and *theta* choices in the pop-up Dialog.

The `mcdisplay` front-end will then require the Perl+PGPLOT, and either Scilab or Matlab to be installed.

5.4.4 Plotting the results of a simulation (mcplot)

The front-end `mcplot` (mcplot.pl on Windows) 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 5.3), the name of the file or directory should be passed to mcplot, *e.g.* "`mcplot` *dir*" or "`mcplot` *file*". It is also possible to plot one single text (not binary) data file from a given monitor, passing its name to mcplot.

The drawing back-end program may be selected among PGPLOT, Matlab and Scilab using either the -p*PLOTTER* option (e.g. `mcplot -pScilab file`) or using the current `MCSTAS_FORMAT` environment variable. Moreover, the drawing back-end program will also be set depending on the *file* extension (see Table 5.2).

It should be emphasized that `mcplot` may *only* display simulation results with the format that was chosen during the computation. Indeed, if you request data in a given format from a simulation, you will only be able to display results using that same drawing back-end. Anyway, the `mcconvert` utility may convert a McStas data file between Matlab and Scilab formats (see section 5.4.8).

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

The initial display shows plots for each detector in the simulation. Examples of plotter appearence for `mcplot` is shown in figures 5.4-5.3.

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

Matlab and Scilab back-ends A dedicated McStas/Mcplot Dialog or menu attached to the plotting window is available, and provides many operations (duplication, export, colormaps, . . .). The corresponding 'mcplot' Matlab and Scilab functions may be called from these language prompt with the same method as in section 5.3, e.g:

```
matlab> s=mcplot;
matlab> help mcplot
scilab> s=mcplot();
matlab or scilab> s=mcplot('mcstas.m');
matlab or scilab> mcplot(s);
```

A full parameter scan simulation result, or simply one of its scan steps may be displayed using the 'Scan step' menu item. When the `+nw` option is specified, a separate Matlab or Scilab window will appear (instead of being launched in the current terminal). This will then enable Java support under Matlab and Tk support under Scilab, resulting in additional menus and tools.

To use the `mcplot` front-end, the programs Perl, and either Scilab or Matlab are required.

5.4.5 Plotting resolution functions (mcresplot)

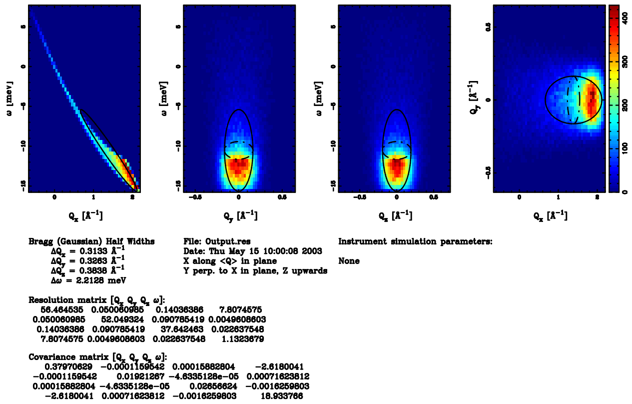


Figure 5.10: Output from `mcresplot` with PGPLOT backend. Use P, C and G keys to write hadrcopy files.

The `mcresplot` front-end is used to plot the resolution function, particularly for triple-axis spectrometers, as calculated by the Res_sample component or TOF_res_sample for time-of-flight instruments. It requires to have a Res_monitor component further in the instrument description (at the detector position). 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 launched with the command

`mcresplot` *file*

Here, *file* is the name of a file output from a simulation using the Res_monitor component.

This front-end currently only works with the PGPLOT plotter, but ports for Matlab and Scilab may be written in the future.

The front-end will open a window displaying projections of the 4-dimensional resolution function $R(\boldsymbol{Q}, \omega)$. The covariance matrix of the resolution function, the resolution along each projection axis and the resulting resolution matrix are also shown, as well as the instrument name and parameters used for the simulation.

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

5.4.6 Creating and viewing the library and component/instrument help (mcdoc)

McStas provides an easy way to generate automatically an HTML help page about a given component or instrument, or the whole McStas library.

```
mcdoc
mcdoc comp—instr
mcdoc -l
```

The first example generates an *index.html* catalog file using the available components and instruments (both locally, and in the McStas library. When called with the `--show` or `-s` option, the library catalog of components is opened using the `BROWSER` environment variable (e.g. netscape, konqueror, nautilus, MSIE, mozilla, . . .).

Alternatively, if a component or instrument *comp* is specified, it will be searched within the library, and an HTML help will be created for all available components matching *comp*. When using the `-s`, the help will be opened. If the `BROWSER` is not defined, the help is displayed as text in the current terminal. This latter output may be forced with the `-t` or `--text` option.

The last example will list the name and action of all McStas tools (same as `--tools` option).

Additionally, the `--web`, `--manual` and `--comp` options will open the McStas web site page, the User Manual (this document) and the Component Manual, all requiring `BROWSER` to be defined. Finally, the `--help` option will display the command help, as usual.

See section 6.4.11 for more details about the McDoc usage and header format. To use the `mcdoc` front-end, the program Perl should be available.

5.4.7 Translating McStas components for Vitess (mcstas2vitess)

Any McStas component may be translated for usage with Vitess (starting from Vitess version 2.3). The syntax is simply

```
mcstas2vitess Compo.comp
```

This will create a Vitess module of the given component.

Let us assume the component `Compo` shall be transferred. The tool first creates a small instrument called `McStas_Compo.instr` consisting of

1. component Vitess_input
2. component `Compo`
3. component Vitess_output

This file is parsed to generate the C file `McStas_Compo.c`. The last step is to compile the C-file and build the executable `McStas_compo`. For both steps McStas is used as for any other instrument. `McStas_compo` has to be moved to the directory 'MODULES' that contains all VITESS executables.

Additionally, a file `McStas_compo.tcl` is created that contains (most of) what is needed to get a GUI window in VITESS. To obtain that, the content of this file has to added into 'vitess.tcl'. To make it accessible within the given GUI structure of VITESS, it is necessary to add the name 'compo' - NO capital letters ! - to one of the folders in 'proc makeModuleSets' (beginning of 'vitess.tcl').

The component Virtual_input transfers all neutron parameters to the McStas definition of the co-ordinate system and the units used. (Of course Virtual_output transfers it back.) This means that 'Compo' works with the McStas definition of co-ordinate system and units, while it is used as a VITESS module. Be careful with axis labeling.

The original parameters of the component and its position have to be given. The origin of this shift is the centre of the end of the previous module, (as it is always the case in VITESS).

To use the `mcstas2vitess` front-end, the program Perl should be available.

5.4.8 Translating McStas results files between Matlab and Scilab formats

If you have been running a McStas simulation with Scilab output, but finally plan to look at the results with Matlab, or the contrary, you may use

`mcconvert` *file.m—sci*

to simply translate one file format to the other. This works only for text files of course. The binary files need not be translated.

5.5 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 IDL, Matlab and Scilab. The output files from simulations are usually simple text files containing headers and data blocks. If data blocks are empty they may be accessed refering to an external file indicated in the header. This file may also be a binary file (except with the original McStas/PGPLOT format), which does not contain any header (except if simulation is launched with `+a` option), but are in turn smaller in size and faster to import.

In order for the user to choose the data format, we recommand to set it *via* the MCSTAS_FORMAT environment variable, which will also make the front-end programs able to import and plot data and instrument consistently. The available format list is shown in Table 5.2.

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.

McStas and PGPLOT format The McStas original format, which is equivalent to the PGPLOT format, is simply columns of ASCII text that most programs should be able to read.

One-dimensional histogram monitors (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 4.2.1, and the number of neutron events for this bin.

Two-dimensional histogram monitors (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 monitors also store the error estimates and event counts as additional matrices.

Single-point monitors 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 5.4.2).

Both one- and two-dimentional monitor 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 monitor component for this data file, *etc.* The headers may be disabled using the `--data-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 monitor 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 monitor components present in the instrument. It is read by the `mcplot` front-end (see section 5.4.4). This file stores the results from single monitors, but by default contains only pointers (in the form of file names) to data for one- and two-dimensional monitors. 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.

Matlab, Scilab and IDL formats With these formats McStas automatically writes scripts containing the data as a structure, as well as in-line import and plot functions for the selected language. Usage examples are given in section 5.3. Thus, it is not necessary to write a load routine for each format, as the script is itself a program that knows how to handle the data. Alternatively, using `mcplot` with Matlab and Scilab plotters provide additional functionalities from menus and dialogs (see section 5.4.4).

When imported through the data generated script (see section 5.3), or using `mcplot` (see section 5.4.4), a single variable may be created into the Matlab, Scilab or IDL base workspace. This variable is a *structure* constituting a data tree containing many fields, some being themselves structures. Field names are the initial names from the instrument (components, files, ...), transformed into valid variable names, e.g containing only letters, digits and the '_' character, except for the first character which may only be a letter[1]. In this tree, you will find the monitor names, which fields contain the monitored data. The usual structure is

`s`.*instrument*.*simulation*.*comp_name*.*file_name*

For instance, reading the data from a 'test' instrument using Matlab format will look like

```
matlab> s=mcstas; % or mcplot mcstas.m from the terminal
```

[1]For instance in most case, the simulation location is './mcstas.m' which turns into field 'm_mcstas'.

```
matlab> s
s =
     Format: 'Matlab with text headers'
        URL: 'http://neutron.risoe.dk'
     Editor: 'farhi on pcfarhi'
    Creator: 'test (test.instr) McStas 1.9 - Nov. 15, 2005 simulation'
       Date: 1.0529e+09
       File: './mcstas'
       test: [1x1 struct]
    EndDate: 1.0529e+09
      class: 'root'
matlab> s.test.m_mcstas.monitor1.monitor1_y_kz
ans =
          ...
          Date:       1.0529e+09
          File:       'monitor1.y_kz'
          type:       'array_2d(20, 10)'
          ...
          ratio:      '1e+06/1e+06'
          signal:     'Min=0; Max=5.54051e-10; Mean= 6.73026e-13;'
          statistics: 'X0=0.438302; dX=0.0201232; Y0=51019.6; dY=20557.1;'
          ...
matlab> eval(s.test.m_mcstas.monitor1.monitor1_y_kz);
matlab> dX
ans =
          0.0201232
```

The latter example accesses the data from the 'monitor1.y_kz' file written by the 'monitor1' component in the 'test' instrument during the './mcstas' simulation. You may evaluate directly the 'signal' and 'statistics' fields of the structure to obtain useful informations.

HTML, XML/NeXus and NeXus formats Both HTML and XML/NeXus formats are available. The former may be viewed using any web browser (Netscape, Internet Explorer, Nautilus), while the latter may be browsed for instance using Internet Explorer (Windows and Mac OS) or GXMLViewer and KXMLEditor (under Linux).

The support for NeXus in McStas is prelimminary as NeXus is not fully laid out. A future version of McStas will fully support output in the final NeXus format [42].

5.6 Using computer Grids and Clusters

Parallelizing a computation is possible when dependencies between each computation are not too strong. The situation of McStas is ideal since each neutron ray can be simulated without interfering with other simulated neutron rays. Therefore each neutron ray can be simulated independently on a set of computer.

McStas provides two methods in order to distribute computations on many computers.

- when scanning instrument parameters (using `mcrun --multi -N`), it is possible to distribute each scan step on a list of computers.
- when using an homogeneous computer cluster, each simulation (including scan steps) may be computed in parallel using MPI (using `mcrun --mpi=Nb_Nodes`).

5.6.1 Distribute mcrun scans (grid) - Unix only

This is an experimental option to distribute scan steps on a set of machines using *ssh* connections. To use this option you will need

1. A set of unix machines of the same architecture (binary compatible). All machines should have the same McStas distribution installed.
2. `ssh` access from the master node (where McStas is installed) to the slaves through e.g. DSA keys *without* a password. These keys should be generated using the command `ssh-keygen`. Run *e.g.* `ssh-keygen -t dsa` on master node, enter no passphrase and add resulting `.ssh/id_dsa.pub` to `.ssh/authorized_keys` on all the slave nodes.
3. The machine names listed in the file `.mcstas-hosts` in your home directory or in the `MCSTAS/tools/perl/mcstas-hosts` on the master node, one node per line. The `--machines=<file>` option enables to specify the hosts file to use.
4. Execute `mcrun` as normally, appending `--multi` or `-M` will divide the N scanpoints across the machines in `.mcstas-hosts` + `localhost`.

Each of the scan steps is executed on distant machines, sending the executable with *scp*, executing single simulations, and then retrieving individual scan steps on the master machine. **Warning:** This option is considered *experimental* - i.e., you are welcome to use it, but be ready for trouble. Note also that this is a unix only option in `mcgui` and `mcrun`, it **does not** work on windows.

5.6.2 Parallel computing (MPI)

When computing N neutron rays with p computers, each computer will simulate $\frac{N}{p}$ neutrons. As a result there will be $p \cdot \frac{N}{p} = N$ neutrons simulated. As a result, McStas generates two kinds of data sets:

- intensity measurements, internally represented by three values (p_0, p_1, p_2) where p_0, p_1, p_2 are additive. Therefore the final value of p_0 is the sum of all local value of p_0 computed on each node. The same rule applies for p_1 and p_2. The evaluation of the intensity errors σ is performed using the final p_0, p_1, and p_2 arrays (see Section 4.2.1).
- event lists: the merge of events is done by concatenation

The MPI support requires to have MPICH, or alternatively LAM-MPI, installed on a set of nodes. This usually also requires to setup properly the `ssh` connections and keys.

MPI Basic usage

To enable parallel computation, compile McStas output C-source file with `mpicc` with the flag `-DUSE_MPI` and run it using the wrapper of your MPI implementation (`mpirun` for mpich or lammpi) :

```
# generate a C-source file [sim.c]
mcstas sim.instr

# generate an executable with MPI support [sim.mpi]
mpicc -DUSE_MPI -o sim.mpi sim.c

# execute with parallel processing over <N> computers
# here you have to list the computers you want to use
# in a file [machines.list] (using mpich implementation)
# (refer to MPI documentation for a complete description)
mpirun -machinefile machines.list -n <N> \
       ./sim.mpi <instrument parameters>
...
```

If you don't want to spread the simulation, run it as usual :

```
./sim.mpi <instrument parameters>
```

If you want to disable MPI support (for example if you do not have MPI installed), just compile the C-source file generated by `mcstas` without `mpicc` and without the `-DUSE_MPI` flag :

```
# generate a C-source file [sim.c]
mcstas sim.instr

# generate an executable without MPI support [sim.no_mpi]
cc -o sim.no_mpi sim.c

# execute on a single computer
./sim.no_mpi <instrument parameters>
...
```

5.6.3 McRun script with MPI support

The `mcrun` script has been adapted to use MPICH implementation of MPI. Two new options have been added:

- `--mpi=<number>`: tells `mcrun` to use MPI, and to spread the simulation over `<number>` nodes
- `--machines=<file>`: defines a text file where the nodes which are to be used for parallel computation are listed; by default, `mcrun` will look at `$HOME/.mcstas-hosts` and `MCSTAS/tools/perl/mcstas-hosts`.

Suppose you have four machines named `node1` to `node4`. A typical machine list file, `machines.list` looks like :

```
node1
node2
node3
node4
```

You can then spread a simulation `sim.instr` using `mcrun` :

```
  mcrun -c --mpi=4 --machines=machines.list \
        sim.instr <instrument parameters>
```

Warning: when using `mcrun` with MPI, be sure to recompile your simulation with MPI support (see `-c` flag of `mcrun`): a simulation compiled without MPI support cannot be used with MPI, whereas a simulation compiled with MPI support can be used without MPI.

5.6.4 McStas/MPI Performance

Theorically, a computation which lasts T seconds on a single computer, should lasts at least $\frac{T}{p}$ seconds when it is distributed over p computers. In practice, there will be overhead time due to the split and merge operations.

- the split is immediate: constant time cost $\mathcal{O}(1)$
- the merge is at worst linear against the number of computers:
  - linear time cost : $\mathcal{O}(p)$ when saving an event list
  - logarithmic time cost: $\mathcal{O}(\log p)$ when not saving an event list

Here is a table showing some comparison between MPI and no-MPI simulations. The cluster is built with 2 bi-xeon 2GHz machines, resulting in 4 processors.

# processors p	duration T (seconds)	notes
1	622	without MPI support
1	630	with MPI support, run without MPI
1	620	with MPI support, run with MPI
2	316	Overhead 6 seconds, $T_{p=1}/2 = 310$
3	210	Overhead 3 seconds, $T_{p=1}/3 = 207$
4	158	Overhead 3 seconds, $T_{p=1}/4 = 155$

The results consistently show that there is a very small overhead when computing using MPI. This overhead comes from the spread and the fusion of the computations. For instance, spreading a computation implies either an `rsh` or and `ssh` session to be opened on every node.

5.6.5 McStas/MPI Bugs and limitations

- signals are *not* supported while simulating with MPI (since asynchroneous events cannot be easily transmitted to all nodes). Anyway, depending on the MPI implementation, you may force all nodes to synchronize and save data using the HUP signal (`kill -HUP mcstasPID`).
- some header of output files might contain minor errors
- the computation split does not take into account the speed or the load of nodes: the overall time of a distributed computation is forced by the slowest node; for optimal performance, the "cluster" should be homogeneous
- forcing the random seed while spreading the computation doesn't work since every node will use identical seeds and will then generate identical pseudo-random values
- MPI must be correctly configured: if using `ssh`, you have to set ssh keys to avoid use of passwords; if using `rsh`, you have to set a `.rhosts` file

Chapter 6

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 ray 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. However, 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.

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 component developers are listed in appendix A. The list includes functions for computing the intersection between a flight-path and various objects (such as cylinders, boxes and spheres), functions for generating random numbers with various distributions, functions for reading or writing informations from/to data files, 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.

6.1 Notational conventions

Simulations generated by McStas use a semi-classical description of the neutron rays to compute the neutron trajectory through the instrument and its interaction with the different components. The effect of gravity is taken into account either in particular components (e.g. `Guide_gravity`), or more generaly when setting an execution flag (`-g`) to perform

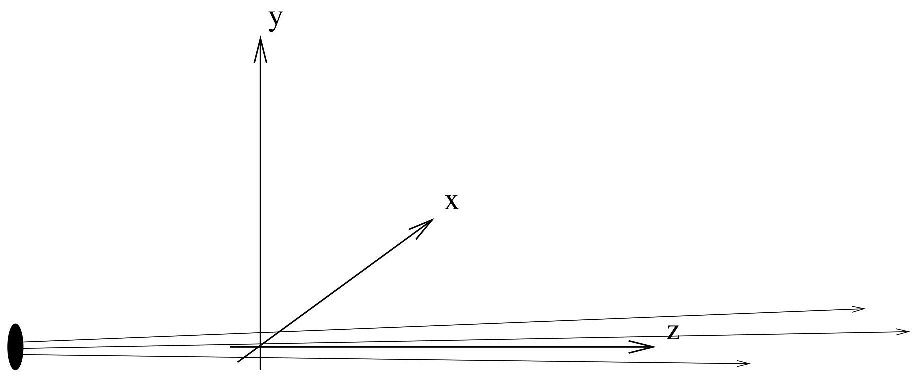


Figure 6.1: conventions for the orientations of the axis in simulations.

gravitation computation. This latter setting is only an approximation and may produce wrong results with some components.

An instrument consists of a list of components through which the neutron ray 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 ray 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 6.1). 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 strongly 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 three spin parameters (s_x, s_y, s_z), normalized to unity and finally the neutron weight p described in 4.

6.2 Syntaxical 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 (names), like C identifiers and keywords, *are* case sensitive, another good reason to use a consistent case convention for keywords. All McStas keywords are reserved, and thus should not be used as C variable names. The list of these reserved keywords is shown in table 6.1.

Keyword	Scope	Meaning
ABSOLUTE	I	Indicates that the AT and ROTATED keywords are in the absolute coordinate system.
AT	I	Indicates the position of a component in an instrument definition.
DECLARE	I,C	Declares C internal instrument or component variables.
DEFINE - INSTRUMENT - COMPONENT	I,C	Starts an instrument or component definition. (each associated parameter may have default values)
DEFINITION	C	Defines component parameters that are constants (like C #define).
END	I,C	Ends the instrument or component definition.
EXTEND	I	Extends the TRACE section of a component in an instrument definition.
FINALLY	I,C	Embeds C code to execute when simulation ends (also executes the SAVE section).
GROUP	I	Defines an exclusive group of components.
%include	I,C	Imports an instrument part, a component or a piece of C code (when within embedded C).
INITIALIZE	I,C	Embeds C code to be executed when starting.
MCDISPLAY	C	Embeds C code to display a geometric representation of a component.
OUTPUT	C	Defines internal variables to be public and protected symbols (usually all global variables and functions of DECLARE).
PARAMETERS	C	Defines a class of component parameter.
POLARISATION	C	Defines neutron polarisation coordinates.
PREVIOUS	C	Refers to a previous component position/orientation.
RELATIVE	I	Indicates that the AT and ROTATED keywords are relative to an other component.
ROTATED	I	Indicates the orientation of a component in an instrument definition.
SAVE	I,C	Embedded C code to execute when saving data.
SETTING	C	Defines component parameters that are variables (double, int, char*).
SHARE	C	Declares global functions and variables to be used by all components.
STATE	C	Defines neutron state coordinates.
TRACE	I,C	Lists the components or embedded C code to execute during simulation.

Table 6.1: Reserved McStas keywords. Scope is 'I' for instrument and 'C' for component definitions.

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 5.2.2). 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 (and possibly C code). 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 (do not add comments after). Additionally, if a "`%include`" statement is found *within* an embedded C code block, the specified file will be included from the 'share' directory of the standard component library (or from the current directory and any user-specified search directories) as a C library, just like the usual "`#include`" *but only once*. For instance, if many components require to read data from a file, they may all ask for "`%include "read_table-lib"`" without duplicating the code of this library. If the file has no extension, both `.h` and `.c` files will be searched and included, otherwise, only the specified file will be imported. The McStas 'run-time' shared library is included by default (equivalent to "`%include "mcstas-r"`" in the `DECLARE` section). For an example of `%include`, see the monitors/Monitor_nD component.

If the instrument description compilation fails, check that the keywords syntax is correct, that no semi-colon `;` sign is missing (e.g. in C blocks and after an ABSORB macros), and there are no name conflicts between instrument and component instances variables.

6.3 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 6.3.8 and some additional examples of instrument definitions can be found on the McStas web-page [2] and in the `example` directory.

6.3.1 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, see section . 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 test(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 *`. The parameters of an instrument may be given default values. Parameters with default values are called *optional parameters*, and need not be given an explicit value when the instrument simulation is executed. When executed without any parameter value in the command line (see section 5.3), the instrument asks for all parameter values, but pressing the `Return` key selects the default value (if any). When used with at least one parameter value in the command line, all non specified parameters will have their value set to the default one (if any). A parameter is given a default value using the syntax "*param*= *value*". For example

```
DEFINE INSTRUMENT test(d1= 1, string s2="hello")
```

Here `d1` and `d2` are an optional parameters and if no value are given explicitly, "1" and "hello" will be used.

Optional parameters can greatly increase the convenience for users of instruments for which some parameters are seldom changed or of unclear signification to the user. Also, if all instrument parameters have default values, then the simple command `mcdisplay test.instr` will show the instrument view without requesting any other input, which is usually a good starting point to study the instrument design.

6.3.2 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. The `%include ''file''` keyword may be used to import a specific component definition or a part of an instrument. Variables defined here are global, and may conflict with internal McStas variables, specially symbols like `x,y,z,sx,sy,sz,vx,vy,vz,t` and generally all names starting with `mc` should be avoided. If you can not compile the instrument, this may be the reason. The `DECLARE` section is optional.

6.3.3 The `INITIALIZE` section

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

This gives code that is executed when the simulation starts. This section is optional. Instrument setting parameters (e.g. doing tests or automatic settings) may be modified in this section.

6.3.4 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 6.4.1 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`". See section 5.2.2 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` (not within embedded C blocks).

The physical 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. The *position* of a component is actually the origin of its local coordinate system. Usually, this is used as the input window position (e.g. for guide-like components), or the center position for cylindrical/spherical components.

The PREVIOUS keyword is a generic name to refer to the previous component in the simulation. Moreover, the PREVIOUS(n) keyword will refer to the n-th previous component, starting from the current component, so that PREVIOUS is equivalent to PREVIOUS(1). This keyword should be used after the RELATIVE keyword, but not for the first component instance of the instrument description.

AT (x, y, z) RELATIVE PREVIOUS ROTATED (ϕ_x, ϕ_y, ϕ_z) RELATIVE PREVIOUS(2)

Invalid PREVIOUS references will be assumed to be absolute placement.

The order and position of components in the TRACE section does not allow components to overlap, except for particular cases (see the GROUP keyword below). Indeed, many components of the McStas library start by propagating the neutron event to the begining of the component itself. Anyway, when the corresponding propagation time is found to be negative (*i.e.* the neutron ray is already *after* or *aside* the component, and has thus passed the 'active' position), the neutron event is ABSORBed, resulting in a zero intensity and event counts after a given position. The number of such removed neutrons is indicated at the end of the simulation. Getting such warning messages is an indication that either some components overlap, or some neutrons are getting outside of the simulation, for instance this usually happens after a monochromator, as the non-reflected beam is indeed lost. A special warning appears when no neutron ray has reached some part of the simulation. This is usually the sign of either overlapping components or a very low intensity.

In some experimental configurations it is neccesary to position one or more groups of components nested, in parallel, or overlapping. One example is a multiple crystal monochromator. One would then like the neutron ray to interact with *one of* the components of the group and then continue.

In order to handle such arrangements without removing neutrons, groups are defined by the GROUP modifier

GROUP *name*

to all involved component declarations. All components of the same named group are tested one after the other, until one of them interacts (uses the SCATTER macro). The selected component acts on the neutron ray, and the rest of the group is skipped. Such groups are thus exclusive (only one of the elements is active). If no component of the group could intercept the neutron ray, it is ABSORBed. If you wish not to absorb these neutrons, you may end each group by a large monitor or another dummy component, which will eventually catch neutrons that were not caught by previous components of the group.

It is sometimes desirable to modify an existing component of the McStas library. One would usually make a copy of the component, and extend the code of its `TRACE` section. McStas provides an easy way to change the behaviour of existing components in an instrument definition without duplicating files, using the `EXTEND` modifier

```
EXTEND
%{
        ... C code executed after the component TRACE section ...
%}
```

The embeded C code is appended at the end of the component `TRACE` section, and all its internal variables (as well as all the `DECLARE` instrument variables, *except* instrument parameters) may be used. This component declaration modifier is of course optional. In the following, two examples of usage of the `EXTEND` keyword are given.

Sources are special components that set the neutron STATE parameters (e.g. position, time, velocity and initial weight) and are usually positioned at the very beginning of the TRACE section. As a default, neutrons are defined at the origin, with zero time and speed. As a consequence, the time spread is zero. Most sources components only set the position and velocity, but do not set the time of the created neutrons. You may manually set the neutron time spreading (for e.g. Time of Flight instruments) in an `EXTEND` code section such as:

```
COMPONENT MySource = source(...) AT(...) EXTEND
  %{
      t = randpm1()*10e-6;
  %}
```

This will set the time randomly between -10 and 10 μ-seconds.

Within a `GROUP`, *all* `EXTEND` sections of the group are executed. In order to discriminate components that are active from those that are skipped, one may use the SCATTERED flag, which is set to zero when entering each component or group, and incremented when the neutron is SCATTERed, as in the following example

```
COMPONENT name0 = comp(p1 = e1, p2 = e2, ...)
  AT (0, 0, 0) ABSOLUTE
COMPONENT name1 = comp(...)
  AT (x, y, z) RELATIVE name0
  ROTATED (φx, φy, φz) RELATIVE name0
  GROUP GroupName EXTEND
  %{
```

```
    if (SCATTERED) printf("I scatter"); else printf("I do not scatter");
  %}
COMPONENT name2 = comp(...)
  AT (x, y, z) RELATIVE name0
  ROTATED (φx, φy, φz) RELATIVE name0
  GROUP GroupName
```

Components *name1* and *name2* are at the same position. If the first one intercepts the neutron (and has a SCATTER within its `TRACE` section), the SCATTERED variable becomes true, the code extension will result in printing "I scatter", and the second component will be skipped. Thus, we recommand to make use of the SCATTER keyword each time a component 'uses' the neutron (scatters, detects, ...).

6.3.5 The `SAVE` section

```
SAVE
%{
    ... C code to execute each time a temporary save is required ...
%}
```

This gives code that will be executed when the simulation is requested to save data, for instance when receiving a USR2 signal (on Unix systems), or using the `Progress_bar` component with intermediate savings. This section is optional.

6.3.6 The `FINALLY` section

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

This gives code that will be executed when the simulation has ended. When existing, the `SAVE` section is first executed. The `FINALLY` section is optional. A simulation may be requested to end before all neutrons have been traced when recieving a TERM or INT signal (on Unix systems), or with Control-C, causing code in `FINALLY` to be evaluated.

6.3.7 The end of the instrument definition

The end of the instrument definition must be explicitly marked using the keyword

```
END
```

6.3.8 Code for the instrument `vanadium_example.instr`

A commented instrument definition taken from the `examples` directory is here shown as an example of the use of McStas.

```
/*******************************************************************************
*
* McStas, neutron ray-tracing package
*         Copyright 1997-2002, All rights reserved
*         Risoe National Laboratory, Roskilde, Denmark
*         Institut Laue Langevin, Grenoble, France
*
* Instrument: vanadium_example
*
* %Identification
* Written by: Kristian Nielsen and Kim Lefmann
* Date: 1998
* Origin: Risoe
* Release: McStas 1.1
* Version: $Revision: 1.6 $
* %INSTRUMENT_SITE: Tutorial
*
* A test instrument using a vanadium cylinder
*
* %Description
* This instrument shows the vanadium sample scattering anisotropy.
* This is an effect of attenuation of the beam in the cylindrical sample.
*
* Example: mcrun vanadium_example.instr ROT=0
*
* %Parameters
* INPUT PARAMETERS:
* ROT: (deg) Rotation angle of the PSD monitor
*
* %Link
* The McStas User manual
* The McStas tutorial
*
* %End
*******************************************************************************/
/* The line below defines the 'name' of our instrument */
/* Here, we have a single input parameter, ROT         */
DEFINE INSTRUMENT vanadium_example(ROT=0)

/* The DECLARE section allows us to declare variables  */
/* in c syntax. Here, coll_div (collimator divergence) */
/* is set to 60 degrees...                             */
DECLARE
%{
  double coll_div = 60;
%}

/* Here comes the TRACE section, where the actual      */
/* instrument is defined....                           */
TRACE

/* The Arm() class component defines reference points  */
/* in 3D space. Every component instance must have a   */
/* unique name. Here, arm is used. This Arm() component*/
/* is set to define the origin of our global coordinate*/
/* system (AT (0,0,0) ABSOLUTE)                        */
```

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

/* Next, we need some neutrons. Let's place a neutron  */
/* source. Refer to documentation of Source_flat to    */
/* understand the different input parameters.          */
/* The source component is placed RELATIVE to the arm  */
/* component, meaning that modifying the position or   */
/* orientation of the arm will also affect the source  */
/* component (and other components after that one...   */
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

/* Here we have a soller - placed to improve beam      */
/* divergence.                                         */
/* The component is placed at a distance RELATIVE to   */
/* a previous component...                             */
COMPONENT collimator = Collimator_linear(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

/* We also need something to 'shoot at' - here a sample*/
/* made from vanadium, an isotropic scatterer. Options */
/* are available to restrict the solid angle in which  */
/* neutrons are emitted (no need to simulate anything  */
/* that we know for sure will not gain us more insight)*/
/* Other options for smart targeting are available -   */
/* refer to component documentation for info.          */
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

/* Here, a secondary arm - or reference point, placed  */
/* on the sample position. The ROT parameter above     */
/* defines rotation of this arm (and components        */
/* relative to the arm)                                */
COMPONENT arm2 = Arm()
  AT (0,0,0) RELATIVE target
  ROTATED (0,ROT,0) relative arm

/* For data output, let us place a detector. This       */
/* detector is not very realistic, since it has a spher-*/
/* ical shape with a 10 m radius, but has the advantage */
/* that EVERYTHING emitted from the sample will be      */
/* picked up. Notice that this component changes        */
/* orientation with the ROT input parameter of the      */
/* instrument.                                          */
COMPONENT PSD_4pi = PSD_monitor_4PI(radius=10, nx=101, ny=51,
  filename="vanadium.psd")
  AT (0,0,0) RELATIVE target ROTATED (ROT,0,0) RELATIVE arm2
END
```

6.4 Writing component definitions

The purpose of a McStas component is to model the interaction of a neutron with a physical component of a real instrument. Given the state of the incoming neutron ray, the component definition calculates the state of the neutron ray 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 6.4.10, and all component definitions can be found on the McStas web-page [2] and described in the McStas component manual.

There exists a large number of functions and constants available in order to write efficient components. See appendix A for

- neutron propagation functions
- geometric intersection time computations
- mathematical functions
- random number generation
- physical constants
- coordinate retrieval and operations
- file generation routines (for monitors),
- data file reading
- ...

6.4.1 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 can be accessed from all sections of the component (see below), as well as in `EXTEND` sections of the instrument definition (see section 6.3).

Setting parameters are translated into C variables usually of type `double` in the generated simulation program, so they are usually 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.

Outside the `INITIALIZE` section of components, changing setting parameter values only affects the current section.

There are a few cases where the use of definition parameters instead of setting parameters makes sense. If the parameter is not numeric, nor a character string (*i.e.* an array, 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()`.

Setting parameters may optionally be declared to be of type `int`, `char *` and `string`, just as in the instrument definition (see section 6.3).

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

This declares a list of C identifiers (variables, functions) that are output parameters (*i.e.* global) 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.

Using `OUTPUT PARAMETERS` is *highly recommanded* for `DECLARE` and internal/global component variables and functions in order to prevent that instances of the same component use the same variable names. Moreover (see section 6.4.2 below), these may be accessed from any other instrument part (e.g. using the `MC_GETPAR` C macro). On the other hand, the variables from the SHARE sections should *not* be defined as OUTPUT parameters.

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 stand for s_x and s_y members of the polarisation vector. To access s_z, use:

`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

Just as for instrument 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. In contrast, if no value is given for radius or height, an error message will result.

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

6.4.2 The DECLARE section

```
DECLARE
%{
        ...C code declarations (variables, definitions, functions)...
        ...These are usually OUTPUT parameters to avoid name conflicts ...
%}
```

This gives C declarations of global variables, functions, 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));
%}
```

This way, McStas takes care to rename transparently the two 'counts' `OUTPUT` parameters so that they are distinct, and can be accessed from elsewhere in the instrument (EXTEND, FINALLY, SAVE, ...) or from other components. This particular example is outdated since McStas monitors will themselves output their contents.

6.4.3 The `SHARE` section

```
SHARE
%{
        ... C code shared declarations (variables, definitions, functions)...
        ... These should not be OUTPUT parameters ...
%}
```

The `SHARE` section has the same role as `DECLARE` except that when using more than one instance of the component, it is inserted *only once* in the simulation code. No occurence of the items to be shared should be in the `OUTPUT` parameter list (not to have McStas rename the identifiers). This is particularly useful when using many instances of the same component (for instance guide elements). If the declarations were in the `DECLARE` section, McStas would duplicates it for each instance (making the simulation code longer). A typical example is to have shared variables, functions, type and structure definitions that may be used from the component `TRACE` section. For an example of `SHARE`, see the samples/Single_crystal component. The `%include ''file''` keyword may be used to import a shared library. The `SHARE` section is optional.

6.4.4 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. Component setting parameters may be modified in this section, affecting the rest of the component.

6.4.5 The `TRACE` section

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

This performs the actual computation of the interaction between the neutron ray and the component. The C code should perform the appropriate calculations and assign the resulting new neutron state to the state parameters. Most components will require propagation routines to reach the component entrance/area. Special macros `PROP_Z0;` and `PROP_DT();` are provided to automate this process (see section A.1).

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. On the other hand, if the neutron event should be *allowed* be backpropagated, the special macro `ALLOW_BACKPROP;` should preceed the call to the `PROP_` call inside the component. When the neutron state is changed or detected, for instance if the component simulates multiple events as multiple reflections in a guide, 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, and to handle component `GROUP`s in an instrument definition (see section 6.3.4). 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 optics/Guide component.

6.4.6 The `SAVE` section

```
SAVE
%{
        ... C code to execute in order to save data ...
%}
```

This gives code that will be executed when the simulation is requested to save data, for instance when receiving a USR2 signal (on Unix systems, see section 5.3), or when triggered by the `Progress_bar(flag_save=1)` component. This might be used by monitors and detectors in order to write results. An extension depending on the selected output format (see 5.2 and section 5.3) is automatically appended to file names, if these latter do not contain extension.

In order to work properly with the common output file format used in McStas, all monitor/detector components should use standard macros for writing data in the SAVE or 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 4.2.1.

As a default, all monitors using the standard macros will display the integral p of the monitor bins, as well as the 2^{nd} moment σ and the number of statistical events N. This will result in a line such as:

`Detector:` *CompName*_I=p *CompName*_ERR=σ *CompName*_N=N "*filename*"

For 1D, 2D and 3D monitors/detectors, the data histogram store in the files is given *p*er bin when the signal is the neutron intensity (*i.e.* most of the cases). Most monitors define binning for an x_n axis value as the sum of events falling into the $[x_n x_{n+1}]$ range, *i.e* the bins are *n*ot centered, but left aligned. Using the Monitor_nD component, it is possible to monitor other signals using the '`signal=`*variable_name*' in the 'options' parameter (refer to that component documentation).

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.

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

DETECTOR_OUT_3D(t, *xlabel*, *ylabel*, *zlabel*, *xvar*, *yvar*, *zvar*, $x_{\min}$, $x_{\max}$, $y_{\min}$, $y_{\max}$, $z_{\min}$, $z_{\max}$, m, n, j
$\&N[0][0][0]$, $\&p[0][0][0]$, $\&p2[0][0][0]$, *filename*)

The meaning of parameters is the same as those used in the 1D and 2D versions of DETECTOR_OUT. The available data format currently saves the 3D arrays as 2D, with the 3rd dimension specified in the *type* field of the data header.

6.4.7 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 free memory and print out final results from components, *e.g.* the simulated intensity in a detector.

6.4.8 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 5.4.3). 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.

6.4.9 The end of the component definition

```
END
```

This marks the end of the component definition.

6.4.10 A component example: Slit

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

```
/*******************************************************************************
*
* McStas, neutron ray-tracing package
*         Copyright 1997-2002, All rights reserved
*         Risoe National Laboratory, Roskilde, Denmark
*         Institut Laue Langevin, Grenoble, France
*
* Component: Slit
*
* %I
* Written by: Kim Lefmann and Henrik M. Roennow
* Date: June 16, 1997
* Version: $Revision: 1.21 $
* Origin: Risoe
* Release: McStas 1.6
*
* Rectangular/circular slit with optional insignificance cut
*
* %D
* A simple rectangular or circular slit. You may either
* specify the radius (circular shape), or the rectangular bounds.
* No transmission around the slit is allowed.
* If cutting option is used, low-weight neutron rays are ABSORBED
*
* Example: Slit(xmin=-0.01, xmax=0.01, ymin=-0.01, ymax=0.01)
*          Slit(radius=0.01, cut=1e-10)
*
* %P
* INPUT PARAMETERS
*
* radius: Radius of slit in the z=0 plane, centered at Origo (m)
* xmin: Lower x bound (m)
* xmax: Upper x bound (m)
* ymin: Lower y bound (m)
* ymax: Upper y bound (m)
*
* Optional parameters:
* cut:  Lower limit for allowed weight (1)
*
* %E
*******************************************************************************/


DEFINE COMPONENT Slit
DEFINITION PARAMETERS ()
SETTING PARAMETERS (xmin=0, xmax=0, ymin=0, ymax=0, radius=0, cut=0)
```

```
STATE PARAMETERS (x,y,z,vx,vy,vz,t,s1,s2,p)

INITIALIZE
%{
  if (xmin == 0 && xmax == 0 && ymin == 0 && ymax == 0 && radius == 0)
    { fprintf(stderr,"Slit: %s: Error: give geometry\n", NAME_CURRENT_COMP); exit(-1); }

%}

TRACE
%{
    PROP_Z0;
    if (((radius == 0) && (x<xmin || x>xmax || y<ymin || y>ymax))
    || ((radius != 0) && (x*x + y*y > radius*radius)))
      ABSORB;
    else
      if (p < cut)
        ABSORB;
      else
        SCATTER;
%}

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
```

6.4.11 McDoc, the McStas library documentation tool

McStas includes a facility called McDoc to help maintain good documentation of components and instruments. 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/instrument actually works.

Two forms of documentation can be generated. One is the component entry dialog in the `mcgui` front-end, see section 5.4.1. The other is a collection of web pages documenting the components and instruments, handled via the `mcdoc` front-end (see section 5.4.6), and the complete documentation for all available McStas components and instruments may be found at the McStas webpage [2], as well as in the McStas library (see 7.1). All available McStas documentation is accessible from the `mcgui` 'Help' menu.

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 library 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 and instruments. 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`.

Additionally, optional keys `%VALIDATION` and `%BUGS` may be found to list validation status and possible bugs in the component.

- 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.

- Optionally, an `%INSTRUMENT_SITE` section followed by a single word is used to sort *instruments* by origin/location in the 'Neutron Site' menu in `mcgui`.
- After `%END`, no more comment text is read by McDoc.

Chapter 7

The component library: Abstract

This chapter presents an abstract of existing components. As a complement to this chapter and the detailed description in the McStas component manual, you may use the `mcdoc -s` command to obtain the on-line component documentation and refer to the McStas web-page [2] where all components are documented using the McDoc system.

7.1 A short overview of the McStas component library

The table in this section gives a quick overview of available McStas components provided with the distribution, in the `MCSTAS` library. The location of this library is detailed in section 5.2.2. All of them are believed to be reliable, and some amount of systematic tests have been carried out. However, no absolute guaranty may be given concerning their accuracy.

The `contrib` directory of the library contains components that were given by McStas users, but where responsibility has not (yet) been taken by the McStas core team.

Additionally the `obsolete` directory of the library gathers components that were renamed, or considered to be outdated. These component are kept for backwards compatibility and they still all work as before.

The `mcdoc` front-end (section 5.4.6) enables to display both the catalog of the McStas library, e.g using:

```
mcdoc
```

as well as the documentation of specific components, e.g with:

`mcdoc --text` *name*
`mcdoc` *file.comp*

The first line will search for all components matching the *name*, and display their help section as text, where as the second example will display the help corresponding to the *file.comp* component, using your BROWSER setting, or as text if unset. The `--help` option will display the command help, as usual.

MCSTAS/sources	Description
Adapt_check	Optimization specifier for the Source_adapt component.
ESS_moderator_long	A parametrised pulsed source for modelling ESS long pulses.
ESS_moderator_short	A parametrised pulsed source for modelling ESS short pulses.
Moderator	A simple pulsed source for time-of-flight.
Monitor_Optimizer	To be used after the Source_Optimizer component.
Source_Maxwell_3	Source with up to three Maxwellian distributions
Source_Optimizer	Optimizes the neutron flux passing through the Source_Optimizer in order to have the maximum flux at the Monitor_Optimizer position.
Source_adapt	Neutron source with adaptive importance sampling.
Source_div	Neutron source with a specified Gaussian divergence.
Source_simple	A simple circular neutron source with flat energy/wavelength spectrum.
Source_gen	General neutron source with tunable shape, spectrum, and divergence.

Table 7.1: Source and source-related components of the McStas library.

MCSTAS/optics	Description
Arm	Arm/optical bench.
Beamstop	Rectangular/circular beam stop.
Bender	A curved neutron guide (shown straight in `mcdisplay`).
Chopper	Disk chopper.
Collimator_linear	A simple analytical Soller collimator.
Collimator_radial	A radial Soller collimator.
FermiChopper	Fermi Chopper with rotating-frame calculations.
Filter_gen	This components may either set the flux or change it (filter-like), using an external data file.
Guide	Straight neutron guide.
Guide_channeled	Straight neutron guide with channels (bender section).
Guide_gravity	Straight neutron guide with gravity. Can be channeled and focusing.
Guide_wavy	Straight neutron guide with gaussian waviness.
Mirror	Single mirror plate.
Monochromator_curved	Doubly bent multiple crystal slabs with anisotropic Gaussian mosaic.
Monochromator_flat	Flat Monochromator crystal with anisotropic Gaussian mosaic.
Selector	A velocity selector (helical lamella type) such as V_selector component.
Slit	Rectangular/circular slit.
V_selector	Velocity selector.
Vitess_ChopperFermi	Curved Fermi chopper (slower than FermiChopper). From the Vitess package.

Table 7.2: Optics components of the McStas library.

MCSTAS/samples	Description
Isotropic_Sqw	A general S(q,ω) scatterer with multiple scattering, for liquids, powders, glasses, polymers. May be concentrically arranged. Coherent/incoherent, elastic/inelastic scattering.
Phonon_simple	Single-crystal sample with acoustic isotropic phonons (simple).
PowderN	General powder sample with N scattering vectors, using a data file.
Sans_spheres	Simple sample for Small Angle Neutron Scattering - hard spheres
Single_crystal	Mosaic single crystal with multiple scattering vectors using a data file.
V_sample	Vanadium sample, or other incoherent scatterer.

Table 7.3: Sample components of the McStas library.

MCSTAS/monitors	Description
DivPos_monitor	Divergence/position monitor (acceptance diagram).
Divergence_monitor	Horizontal+vertical divergence monitor.
E_monitor	Energy-sensitive monitor.
L_monitor	Wavelength-sensitive monitor.
Monitor_nD	General monitor that can output 0/1/2D signals (Intensity or signal vs. [something] and vs. [something] ...).
PSD_monitor	Position-sensitive monitor.
PreMonitor_nD	This component is a PreMonitor that is to be used with one Monitor_nD, in order to record some neutron parameter correlations.
TOFLambda_monitor	Time-of-flight/wavelength monitor.
TOF_monitor	Rectangular Time-of-flight monitor.

Table 7.4: Monitor components of the McStas library.

MCSTAS/misc	Description
Progress_bar	Displays status of a running simulation. May also trigger intermediate SAVE.
Res_sample	Sample-like component for resolution function calculation.
TOF_Res_sample	Sample-like component for resolution function calculation in TOF instruments.
Res_monitor	Monitor for resolution function calculations
TOF_Res_monitor	Monitor for resolution function calculations for TOF instruments
Virtual_input	Source-like component that reads neutron events from an ascii/binary 'virtual source' file (recorded by Virtual_output).
Virtual_output	Detector-like component that writes neutron state (for use in Virtual_input).
Vitess_input	Read neutron state parameters from a VITESS neutron file.
Vitess_output	Write neutron state parameters to a VITESS neutron file.

Table 7.5: Miscellaneous components of the McStas library.

MCSTAS/contrib	Description
Al_window	Aluminium transmission window.
Collimator_ROC	Radial Oscillating Collimator (ROC).
Filter_graphite	Pyrolytic graphite filter (analytical model).
Filter_powder	Box-shaped powder filter based on Single_crystal (unstable).
Guide_curved	Non focusing continuous curved guide (shown curved).
Guide_honeycomb	Neutron guide with gravity and honeycomb geometry. Can be channeled and/or focusing.
Guide_tapering	Rectangular tapered guide (parabolic, elliptic, sections ...).
He3_cell	Polarised 3He cell.
ISIS_moderator	ISIS Target 1 and 2 moderator models based on MCNP calculations.
Monochromator_2foc	Doubly bent monochromator with multiple slabs.
PSD_monitor_rad	A banana PSD monitor.
SiC	SiC multilayer sample for reflectivity simulations.
SNS_source	SNS moderator models based on MCNP calculations.
Virtual_tripoli4_input	Reads a 'Batch' neutron events file from Tripoli 4.
Virtual_tripoli4_output	Writes a 'Batch' neutron events file for Tripoli 4.

Table 7.6: Contributed components of the McStas library.

MCSTAS/share	Description
adapt_tree-lib	Handles a simulation optimisation space for adatative importance sampling. Used by the Source_adapt component.
mcstas-r	Main Run-time library (always included).
monitor_nd-lib	Handles multiple monitor types. Used by Monitor_nD, Res_monitor, ...
read_table-lib	Enables to read a data table (text/binary) to be used within an instrument or a component.
vitess-lib	Enables to read/write Vitess event binary files. Used by Vitess_input and Vitess_output

Table 7.7: Shared libraries of the McStas library. See Appendix A for details.

MCSTAS/data	Description
*.lau	Laue pattern file, as issued from Crystallographica. For use with Single_crystal, PowderN, and Isotropic_Sqw. Data: [h k l Mult. d-space 2Theta F-squared]
*.laz	Powder pattern file, as obtained from Lazy/ICSD. For use with PowderN, Isotropic_Sqw and possibly Single_crystal.
*.trm	transmission file, typically for monochromator crystals and filters. Data: [k (Angs-1) , Transmission (0-1)]
*.rfl	reflectivity file, typically for mirrors and monochromator crystals. Data: [k (Angs-1) , Reflectivity (0-1)]
*.sqw	$S(q,\omega)$ files for Isotropic_Sqw component. Data: [q] [ω] [$S(q,\omega)$]

Table 7.8: Data files of the McStas library.

MCSTAS/examples	Description
*.instr	This directory contains example instruments, accessible throught the `mcgui` "Neutron site" menu.

Table 7.9: Instrument example files of the McStas library.

Chapter 8

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 (8.2) and the ISIS time-of-flight spectrometer PRISMA (8.3). Before that, however, we present one example of a component test instrument: the instrument to test the component **V_sample** (8.1). These instrument files are included in the McStas distribution in the `examples/` directory. All the instrument examples there-in may be executed automatically throught the McStas self-test procedure (see section 3.7). It is our intention to extend the list of instrument examples extensively and perhaps publish them in a separate report.

8.1 A test instrument for the component V_sample

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

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 8.1.

8.1.1 Scattering from the V-sample test instrument

In figure 8.2, 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.

8.2 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 the now closed source of Risø National Laboratory. Except for the cold source itself, all components used have quite

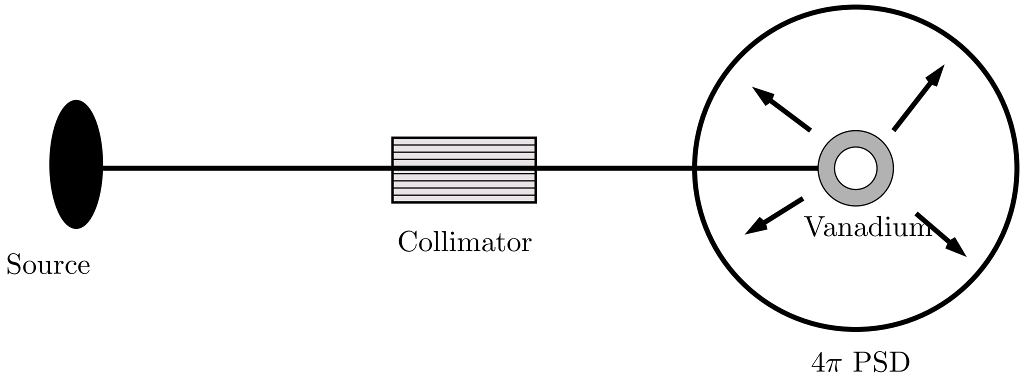


Figure 8.1: A sketch of the test instrument for the component V_sample.

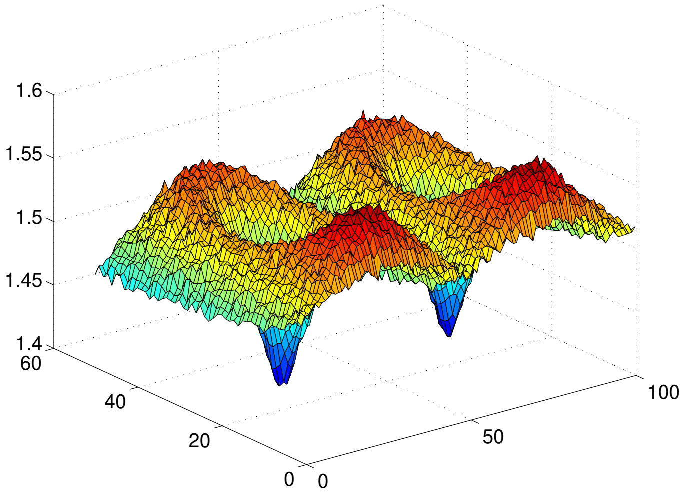


Figure 8.2: Scattering from a V-sample, measured by a spherical PSD. The sphere has been transformed onto a plane and the intensity is plotted as the third dimension.

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 was the first detailed work performed with the McStas package. For further details see reference [43].

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 $1{\times}1$ 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 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.

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 8.3. Test results and data from the real spectrometer are shown in Appendix 8.2.1.

8.2.1 Simulated and measured resolution of TAS1

In order to test the McStas package on a qualitative level, we have performed a very detailed comparison of a simulation with a standard experiment from 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 these simulations, we have tried to reproduce every alignment scan with respect to

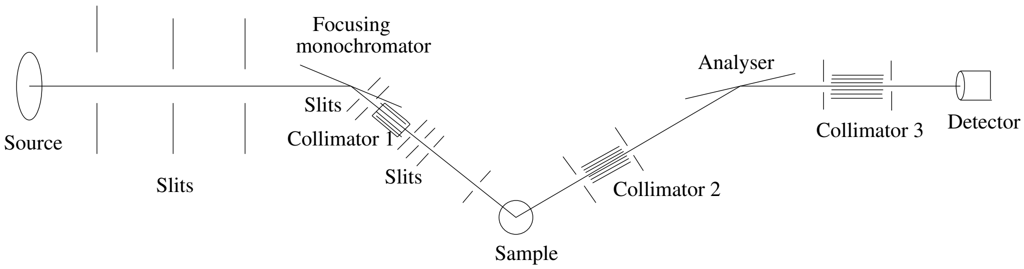


Figure 8.3: A sketch of the TAS1 instrument.

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 8.4 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 0 degrees. 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 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 8.5.

The result of a $2\theta_s$ scan on an Al_2O_3 powder sample in two-axis mode is shown in Figure 8.6. 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 8.7.

8.3 The time-of-flight spectrometer PRISMA

In order to test the time-of-flight aspect of McStas, we have in collaboration with Mark Hagen, now at SNS, 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` registers which crystal scatters the neutron ray. The detector component is then modified to construct seven different time-of-flight histograms, one for each crystal (see the source code for the instrument for details). One

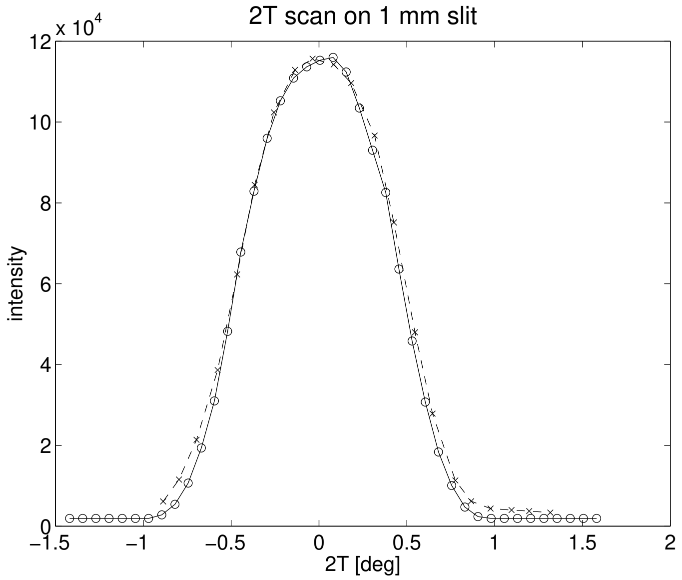


Figure 8.4: TAS1: Scans of $2\theta_s$ in the direct beam with 1 mm slit on the sample position. "$\times$": measurements, "o": simulations, scaled to the same intensity Collimations: open-30'-open-open.

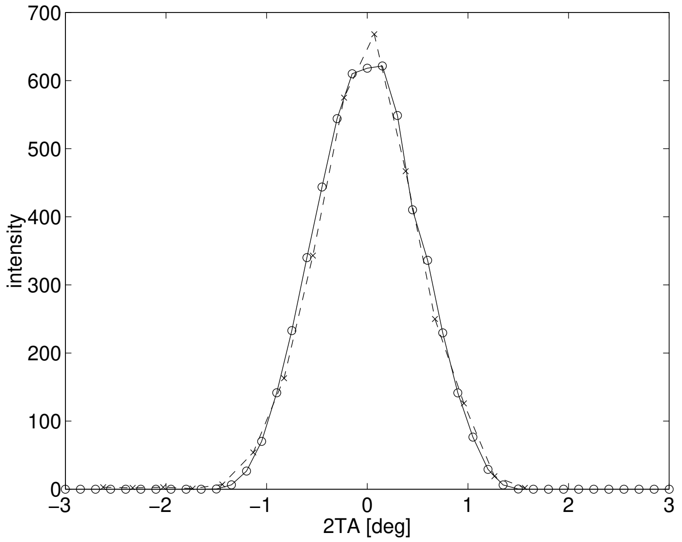


Figure 8.5: TAS1: Corrected $2\theta_a$ scan on a V-sample. Collimations: open-30'-28'-67'. "$\times$": measurements, "o": simulations.

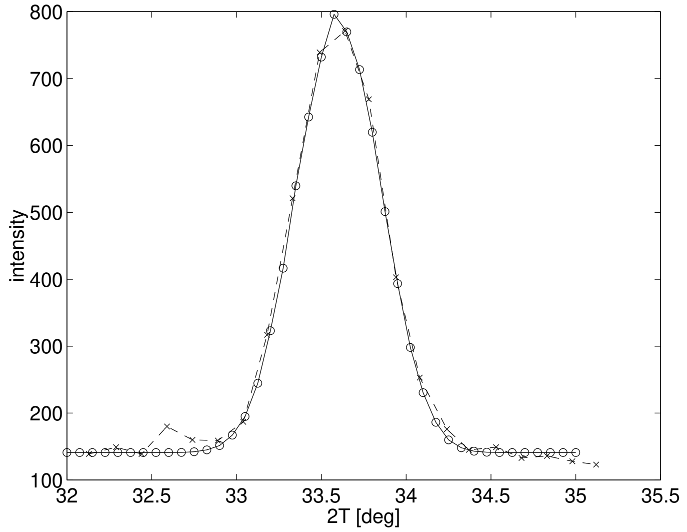


Figure 8.6: TAS1: $2\theta_s$ scans on Al_2O_3 in two-axis, focusing mode. Collimations: open-30'-28'-67'. "$\times$": measurements, "o": simulations. A constant background is added to the simulated data.

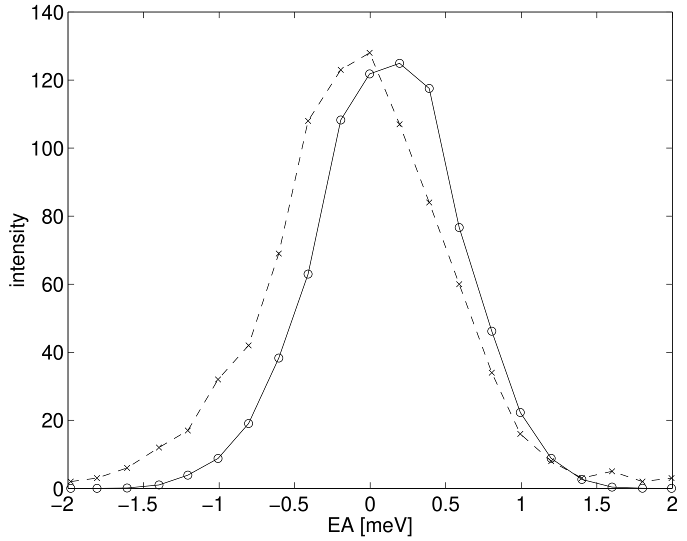


Figure 8.7: TAS1: Scans of the analyser energy on a V-sample. Collimations: open-30'-28'-67'. "$\times$": measurements, "o": simulations.

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 8.8. Test results are shown in Appendix 8.3.1.

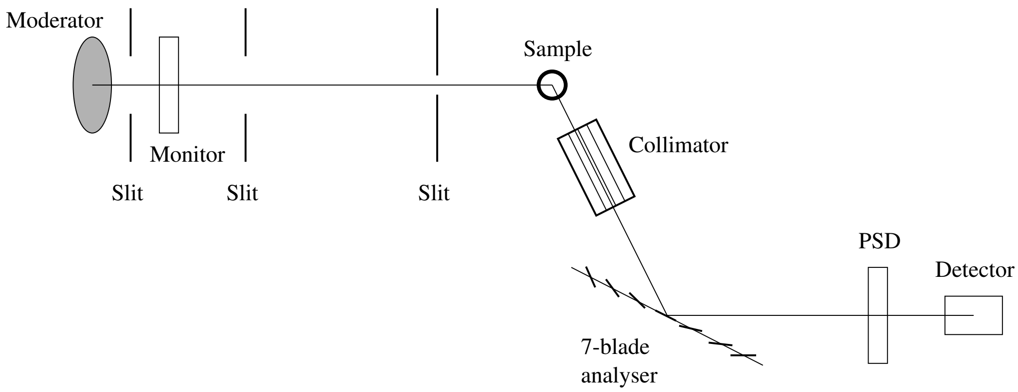


Figure 8.8: A sketch of the PRISMA instrument.

8.3.1 Simple spectra from the PRISMA instrument

A plot from the detector in the PRISMA simulation is shown in Figure 8.9. 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 not performed any quantitative analysis of the data.

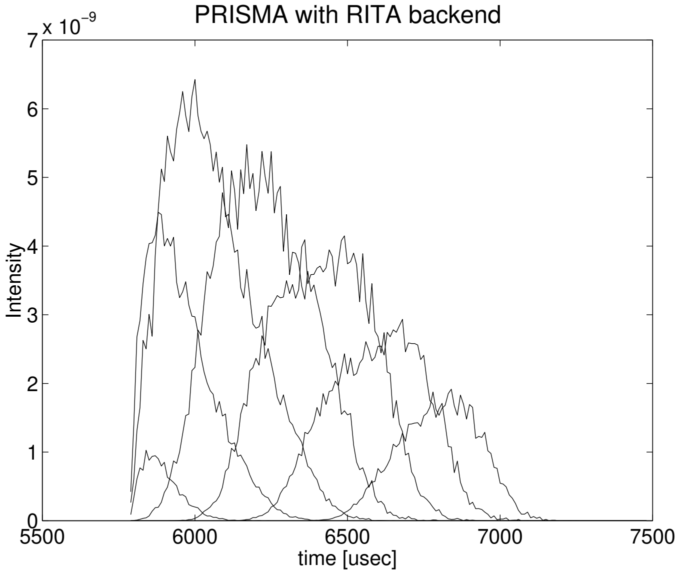


Figure 8.9: Test result from PRISMA instrument using “colored neutrons”. Each graph shows the neutrons scattered from one analyser blade.

Appendix A

Libraries and conversion constants

The McStas Library contains a number of built-in functions and conversion constants which are useful when constructing components. These are stored in the `share` directory of the `MCSTAS` library.

Within these functions, the 'Run-time' part is available for all component/instrument descriptions. The other parts are dynamic, that is they are not pre-loaded, but only imported once when a component requests it using the `%include` McStas keyword. For instance, within a component C code block, (usually SHARE or DECLARE):

```
%include "read_table-lib"
```

will include the 'read_table-lib.h' file, and the 'read_table-lib.c' (unless the `--no-runtime` option is used with `mcstas`). Similarly,

```
%include "read_table-lib.h"
```

will *only* include the 'read_table-lib.h'. The library embedding is done only once for all components (like the SHARE section). For an example of implementation, see **Res_monitor**.

In this Appendix, we present a short list of both each of the library contents and the run-time features.

A.1 Run-time calls and functions (`mcstas-r`)

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

A.1.1 Neutron propagation

Propagation routines perform all necessary operations to transport neutron rays from one point to an other. Except when using the special `ALLOW_BACKPROP;` call prior to exectuting any `PROP_*` propagation, the neutron rays which have negative propagation times are removed automatically.

- **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 accordingly from knowledge of the neutron velocity (vx, vy, vz). If the propagation time is negative, the neutron ray is absorbed.

  For components that are centered along the z-axis, use the `_intersect` functions to determine intersection time(s), and then a `PROP_DT` call.

- **PROP_DT**(dt). Propagates the neutron through the time interval dt, adjusting (x, y, z) and t accordingly from knowledge of the neutron velocity.

- **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.

- **ALLOW_BACKPROP**. Indicates that the next propagation routine will not remove the neutron ray, even if negative propagation times are found. Further propagations are not affected.

- **SCATTER**. This macro is used to denote a scattering event inside a component. It should be used e.g to indicate that a component has interacted with the neutron ray (e.g. scattered or detected). This does not affect the simulation (see, however, **Beamstop**), and it is mainly used by the `MCDISPLAY` section and the `GROUP` modifier See also the SCATTERED variable (below).

A.1.2 Coordinate and component variable retrieval

- **MC_GETPAR**(). This may be used in the FINALLY section of an instrument definition to reference the output parameters of a component.

- **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 matrix is referred to as ROT_A_CURRENT_COMP$[m][n]$, where m and n are 0, 1, or 2 standing for x, y and z coordinates respectively.

- **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.

- **INDEX_CURRENT_COMP** is the number (index) of the current component (starting from 1).
- **POS_A_COMP_INDEX**(*index*) is the absolute position of component *index*. POS_A_COMP_INDEX (INDEX_CURRENT_COMP) is the same as POS_A_CURRENT_COMP. You may use POS_A_COMP_INDEX (INDEX_CURRENT_COMP+1) to make, for instance, your component access the position of the next component (this is usefull for automatic targeting). A component of the vector is referred to as POS_A_COMP_INDEX(*index*).i where i is x, y or z.
- **POS_R_COMP_INDEX** works the same as above, but with relative coordinates.
- **STORE_NEUTRON**($index, x, y, z, vx, vy, vz, t,\ sx, sy, sz, p$) stores the current neutron state in the trace-history table, in local coordinate system. *index* is usually INDEX_CURRENT_COMP. This is automatically done when entering each component of an instrument.
- **RESTORE_NEUTRON**($index, x, y, z, vx, vy, vz, t, sx, sy, sz, p$) restores the neutron state to the one at the input of the component *index*. To ignore a component effect, use RESTORE_NEUTRON (INDEX_CURRENT_COMP, $x, y, z, vx, vy, vz, t, sx, sy, sz, p$) at the end of its TRACE section, or in its EXTEND section. These neutron states are in the local component coordinate systems.
- **SCATTERED** is a variable set to 0 when entering a component, which is incremented each time a SCATTER event occurs. This may be used in the `EXTEND` sections to determine whether the component interacted with the current neutron ray.
- **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.
- **mcset_ncount**(n). Sets the number of neutron histories to simulate to n.
- **mcget_ncount**(). Returns the number of neutron histories to simulate (usually set by option `-n`).
- **mcget_run_num**(). Returns the number of neutron histories that have been simulated until now.

A.1.3 Coordinate transformations

- **coords_set**(x, y, z) returns a Coord structure (like POS_A_CURRENT_COMP) with x, y and z members.
- **coords_get**(P, &x, &y, &z) copies the x, y and z members of the Coord structure P into x, y, z variables.

- **coords_add**(a, b), **coords_sub**(a, b), **coords_neg**(a) enable to operate on coordinates, and return the resulting Coord structure.

- **rot_set_rotation**(*Rotation t*, ϕ_x, ϕ_y, ϕ_z) Get transformation matrix for rotation first ϕ_x around x axis, then ϕ_y around y, and last ϕ_z around z. t should be a 'Rotation' ([3][3] 'double' matrix).

- **rot_mul***(Rotation t1, Rotation t2, Rotation t3)* performs $t3 = t1.t2$.

- **rot_copy***(Rotation dest, Rotation src)* performs $dest = src$ for Rotation arrays.

- **rot_transpose***(Rotation src, Rotation dest)* performs $dest = src^t$.

- **rot_apply***(Rotation t, Coords a)* returns a Coord structure which is $t.a$

A.1.4 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).

- **vec_prod**$(a_x, a_y, a_z, b_x, b_y, b_z, c_x, c_y, c_z)$. 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).

- **solve_2nd_order**(*t, A, B, C). Solves the 2^{nd} order equation $At^2 + Bt + C = 0$ and returns the smallest positive solution into pointer *t.

A.1.5 Output from detectors

Details about using these functions are given in the McStas User Manual.

- **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.

- **DETECTOR_OUT_1D**(...). Used to output the results from a one-dimentional detector.

- **DETECTOR_OUT_2D**(...). Used to output the results from a two-dimentional detector.

- **DETECTOR_OUT_3D**(...). Used to output the results from a three-dimentional detector. Arguments are the same as in DETECTOR_OUT_2D, but with the additional z axis (the signal). Resulting data files are treated as 2D data, but the 3rd dimension is specified in the *type* field.

- **mcinfo_simulation***(FILE *f, mcformat, char *pre, char *name)* is used to append the simulation parameters into file f (see for instance **Res_monitor**). Internal variable $mcformat$ should be used as specified. Please contact the authors for further information.

A.1.6 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.1.7 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 Ref. [44, ch.7].

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

- **randvec_target_circle**(&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 *disk* centered at (aim_x, aim_y, aim_z) with radius r_f (in meters), and perpendicular to the *aim* vector.. All directions that intersect the circle are chosen with equal probability. The solid angle of the circle as seen from the position of the neutron is returned in $d\Omega$. This routine was previously called **randvec_target_sphere** (which still works).

- **randvec_target_rect_angular**(&v_x, &v_y, &v_z, &$d\Omega$, aim_x, aim_y, aim_z,h, w, Rot) does the same as randvec_target_circle but targetting at a rectangle with angular dimensions h and w (in **radians**, not in degrees as other angles). The rotation matrix *Rot* is the coordinate system orientation in the absolute frame, usually ROT_A_CURRENT_COMP.

- **randvec_target_rect**(&v_x, &v_y, &v_z, &$d\Omega$, aim_x, aim_y, aim_z,$height, width, Rot$) is the same as randvec_target_rect_angular but *height* and *width* dimensions are given in meters. This function is useful to e.g. target at a guide entry window or analyzer blade.

A.2 Reading a data file into a vector/matrix (Table input, read_table-lib)

The `read_table-lib` provides functionalities for reading text (and binary) data files. To use this library, add a `%include "read_table-lib"` in your component definition DECLARE or SHARE section. Tables are structures of type t_{Table} (see `read_table-lib.h` file for details):

```
/* t_Table structure (most important members) */
double *data;     /* Use Table_Index(Table, i j) to extract [i,j] element */
long    rows;     /* number of rows */
long    columns;  /* number of columns */
char   *header;   /* the header with comments */
char   *filename; /* file name or title */
double  min_x;    /* minimum value of 1st column/vector */
double  max_x;    /* maximum value of 1st column/vector */
```

Available functions to read *a single* vector/matrix are:

- **Table_Init**(&$Table$, $rows$, *columns*) returns an allocated Table structure. Use $rows$ = *columns* = 0 not to allocate memory. Calls to Table_Init are optional, since initialization is being performed by other functions already.

- **Table_Read**(&$Table$, $filename$, *block*) reads numerical block number *block* (0 to catenate all) data from *text* file $filename$ into $Table$, which is as well initialized in the process. The block number changes when the numerical data changes its size, or a comment is encoutered (lines starting by '`# ; % /`'). If the data could not be read, then $Table$.*data* is NULL and $Table$.$rows$ = 0. You may then try to read it using Table_Read_Offset_Binary. Return value is the number of elements read.

- **Table_Read_Offset**(&$Table$, $filename$, *block*, &*offset*, n_{rows}) does the same as Table_Read except that it starts at offset *offset* (0 means begining of file) and reads n_{rows} lines (0 for all). The *offset* is returned as the final offset reached after reading the n_{rows} lines.

- **Table_Read_Offset_Binary**(&$Table$, $filename$, *type*, *block*, &*offset*, n_{rows}, $n_{columns}$) does the same as Table_Read_Offset, but also specifies the *type* of the file (may be "float" or "double"), the number n_{rows} of rows to read, each of them having $n_{columns}$ elements. No text header should be present in the file.

- **Table_Rebin**(&$Table$) rebins all $Table$ rows with increasing, evenly spaced first column (index 0), e.g. before using Table_Value. Linear interpolation is performed for all other columns. The number of bins for the rebinned table is determined from the smallest first column step.

- **Table_Info**($Table$) print information about the table $Table$.

- **Table_Index**($Table, m, n$) reads the $Table[m][n]$ element.

- **Table_Value**($Table, x, n$) looks for the closest x value in the first column (index 0), and extracts in this row the n-th element (starting from 0). The first column is thus the 'x' axis for the data. It does not contain any interpolation method.
- **Table_Free**(&$Table$) free allocated memory blocks.

Available functions to read *an array* of vectors/matrices in a *text* file are:

- **Table_Read_Array**($File$, &n) read and split $file$ into as many blocks as necessary and return a t_{Table} array. Each block contains a single vector/matrix. This only works for text files. The number of blocks in put into n.
- **Table_Free_Array**(&$Table$) free the $Table$ array.
- **Table_Info_Array**(&$Table$) display information about all data blocks.

The format of text files is free. Lines starting by '`# ; % /`' characters are considered to be comments, and stored in $Table.header$. Data blocks are vectors and matrices. Block numbers are counted starting from 1, and changing when a comment is found, or the column number changes. For instance, the file 'MCSTAS/data/BeO.trm' (Transmission of a Berylium filter) looks like:

```
# BeO transmission, as measured on IN12
# Thickness: 0.05 [m]
# [ k(Angs-1) Transmission (0-1) ]
# wavevector multiply
1.0500  0.74441
1.0750  0.76727
1.1000  0.80680
...
```

Binary files should be of type "float" (i.e. REAL*32) and "double" (i.e. REAL*64), and should *not* contain text header lines. These files are platform dependent (little or big endian).

The $filename$ is first searched into the current directory (and all user additional locations specified using the `-I` option, see the 'Running McStas ' chapter in the User Manual), and if not found, in the `data` sub-directory of the `MCSTAS` library location. This way, you do not need to have local copies of the McStas Library Data files (see table 7.8).

A usage example for this library part may be:

```
t_Table Table;       // declare a t_Table structure
char file[]="BeO.trm";  // a file name
double x,y;

Table_Read(&Table, file, 1);  // initialize and read the first numerical block
Table_Info(Table);            // display table informations
...
x = Table_Index(Table, 2,5);  // read the 3rd row, 6th column element
                              // of the table. Indexes start at zero in C.
```

```
y = Table_Value(Table, 1.45,1);  // look for value 1.45 in 1st column (x axis)
                                // and extract 2nd column value of that row
Table_Free(&Table);             // free allocated memory for table
```

Additionally, if the block number (3rd) argument of **Table_Read** is 0, all blocks will be catenated. The **Table_Value** function assumes that the 'x' axis is the first column (index 0). Other functions are used the same way with a few additional parameters, e.g. specifying an offset for reading files, or reading binary data.

This other example for text files shows how to read many data blocks:

```
t_Table *Table;       // declare a t_Table structure array
long     n;
double y;

Table = Table_Read_Array("file.dat", &n); // initialize and read the all numerical block
n = Table_Info_Array(Table);     // display informations for all blocks (also returns n)

y = Table_Index(Table[0], 2,5);  // read in 1st block the 3rd row, 6th column element
                                 // ONLY use Table[i] with i < n !
Table_Free_Array(Table);         // free allocated memory for Table
```

You may look into, for instance, the source files for **Monochromator_curved** or **Virtual_input** for other implementation examples.

A.3 Monitor_nD Library

This library gathers a few functions used by a set of monitors e.g. Monitor_nD, Res_monitor, Virtual_output, etc. It may monitor any kind of data, create the data files, and may display many geometries (for `mcdisplay`). Refer to these components for implementation examples, and ask the authors for more details.

A.4 Adaptative importance sampling Library

This library is currently only used by the components **Source_adapt** and **Adapt_check**. It performs adaptative importance sampling of neutrons for simulation efficiency optimization. Refer to these components for implementation examples, and ask the authors for more details.

A.5 Vitess import/export Library

This library is used by the components **Vitess_input** and **Vitess_output**, as well as the `mcstas2vitess` utility. Refer to these components for implementation examples, and ask the authors for more details.

A.6 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 (Å^{-1})
K2V	$10^{-10} \cdot \hbar/m_{\mathrm{N}}$	**k**-vector (Å^{-1})	Velocity (m/s)
VS2E	$m_{\mathrm{N}}/(2e)$	Velocity squared (m^2 s^{-2})	Neutron energy (meV)
SE2V	$\sqrt{2e/m_{\mathrm{N}}}$	Square root of neutron energy (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
MNEUTRON	$1.67492 \cdot 10^{-27}$ kg	Neutron mass, m_{n}	
HBAR	$1.05459 \cdot 10^{-34}$ Js	Planck constant, $\hbar$	
PI	3.14159265...	π	
FLT_MAX	3.40282347E+38F	a big float value	

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 McStas core team.

- **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.
- **Kernel** The McStas language definition and the associated compiler
- **McStas** Monte Carlo Simulation of Triple Axis Spectrometers (the name of this package). Pronounciation ranges from *mex-tas*, to *mac-stas* and *m-c-stas*.
- **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://www.mcstas.org.

[3] See http://neutron.risoe.dk/McZilla/.

[4] D. Wechsler, G. Zsigmond, F. Streffer, and F. Mezei. *Neutron News*, **25**, 11, 2000.

[5] See http://www.hmi.de/projects/ess/vitess/.

[6] See http://mcnsi.risoe.dk/.

[7] See http://neutron-eu.net/en/.

[8] See http://strider.lansce.lanl.gov/NISP/Welcome.html.

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

[10] 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.

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

[12] See http://www.sns.gov/.

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

[14] J. R. D. Copley, P. Verkerk, A. A. van Well, and H. Fredrikze. *Comput. Phys. Commun.*, **40**, 337, 1986.

[15] H. Maier-Leibnitz and T. Springer. *Reactor Sci. Technol.*, **17**, 217, 1963.

[16] D.F.R. Mildner. *Nucl. Instr. Meth.*, **A290**, 189, 1990.

[17] R.A. Lowde. *J. Nucl. Energy, Part A: Reactor Sci.*, **11**, 69, 1960.

[18] J.R.D. Copley. *Nucl. Instr. Meth.*, **A510**, 318, 2003.

[19] E. Fermi, J. Marshall, and L. Marshall. *Phys. Rev.*, **72**, 193, 1947.

[20] J. Peters. *Nucl. Instr. Meth.*, **A540**, 419, 2005.

[21] C.D. Clark, E.W.J. Mitchell, D.W. Palmer, and I.H. Wilson. *J. Sci. Instrum.*, **43**, 1, 1966.

[22] A.K. Freund. *Nucl. Instr. Meth.*, **213**, 495, 1983.

[23] V.F. Sears. *Acta Cryst.*, **A53**, 35, 1997.

[24] G. Shirane, S.M. Shapiro, and J.M. Tranquada. *Neutron Scattering with a Triple-Axis Spectrometer*. Cambridge University Press, 2002.

[25] L. Alianelli. *J. Appl. Cryst.*, **37**, 732, 2004.

[26] V. Radeka. *IEEE Trans. Nucl. Sci.*, **NS-21**, 51, 1974.

[27] V. Peskov, G. Charpak, W. Dominik, and F. Sauli. *Nucl. Instr. and Meth.*, **A277**, 547, 1989.

[28] G. Manzin, B. Guerard, F.A.F. Fraga, and L.M.S. Margato. *Nucl. Instr. Meth.*, **A535**, 102, 2004.

[29] J.R.D. Copley. *J. Neut. Research*, **1**, 21, 1993.

[30] L. D. Cussen. *J. Appl. Cryst.*, **36**, 1204, 2003.

[31] F. James. *Rep. Prog. Phys.*, **43**, 1145, 1980.

[32] Grimmett, G. R., and Stirzakerand D. R. *Probability and Random Processes, 2nd Edition*. Clarendon Press, Oxford, 1992.

[33] P. A. Seeger, L. L. Daemen, T. G. Thelliez, and R. P. Hjelm. *Physica B*, **283**, 433, 2000.

[34] J. Saroun and J. Kulda. *Physica B*, **234**, 1102, 1997. See website http://omega.ujf.cas.cz/restrax/.

[35] P. Willendrup, E. Farhi, and K. Lefmann. *Physica B*, **350**, 735, 2004.

[36] W.-T. Lee and X.-L. Wang. *Neutron News*, **13**, 30, 2002. See website http://www.sns.gov/ideas/.

[37] E. Farhi, T. Hansen, A. Wildes, R. Ghosh, and K. Lefmann. *Appl. Phys.*, **A 74**, S1471, 2002.

[38] C. Schanzer, P. Boni, U. Filges, and T. Hils. *Nucl. Instr. Meth.*, **A 529**, 63, 2004.

[39] G. Zsigmond, K. Lieutenant, and S. Manoshin et al. *Nucl. Instr. Meth.*, **A 529**, 218, 2004.

[40] K. Lieutenant. *J. Phys.: Condens. Matter*, **17**, S167, 2005.

[41] See http://www-rocq.inria.fr/scilab/.

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

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

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

Index

Bibliographic Data Sheet **Risø–R–1416(EN)**

Title and author(s)

User and Programmers Guide to the Neutron Ray-Tracing Package McStas, Version 1.9

Peter Kjær Willendrup, Emmanuel Farhi, Klaus Lieutenant, Kim Lefmann, and Kristian Nielsen

ISBN	ISSN
ISBN 87–550–3481–0 (Internet)	0106–2840

Dept. or group	Date
Materials Research Department	September, 2005

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

Pages	Tables	Illustrations	References
112	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