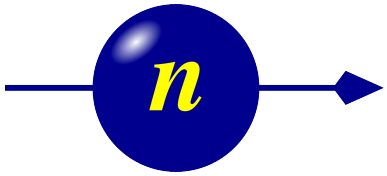
Physics Physics Department,
Technical University of Denmark
2800 Kongens Lyngby, Denmark

User and Programmers Guide to the Neutron Ray-Tracing Package McStas, version 2.1



P. Willendrup, E. Farhi, E. Knudsen, U. Filges, K. Lefmann, J. Stein

February, 2014

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 for e.g. training, experimental planning or data analysis. 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 documentation of most 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 2.1, released February, 2014

The authors are:

Peter Kjær Willendrup <pkwi@fysik.dtu.dk>
Physics Department, Technical University of Denmark, Kongens Lyngby, Denmark

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

Erik Knudsen <erkn@fysik.dtu.dk>
Physics Department, Technical University of Denmark, Kongens Lyngby, Denmark

Kim Lefmann <lefmann@fys.ku.dk>
Niels Bohr Institute, University of Copenhagen, Denmark

Jonas Stein <Jonas.Stein@uni-koeln.de>
Institute of Physics II, University of Cologne, Germany

as well as authors who left the project:

Peter Christiansen <pchristi@hep.lu.se>
Materials Research Department, Risø National Laboratory, Roskilde, Denmark
Present address: University of Lund, Lund, Sweden
Klaus Lieutenant <klaus.lieutenant@helmholtz-berlin.de>
Institut Laue-Langevin, Grenoble, France
Present address: Helmotlz Zentrum Berlin, Germany
Kristian Nielsen <kristian-nielsen@mail.tele.dk>
Materials Research Department, Risø National Laboratory, Roskilde, Denmark
Presently associated with: MySQL AB, Sweden

ISBN 978–87–550–3679–6
ISSN 0106–2840

Information Service Department · Risø DTU · 2014

Contents

Preface and acknowledgements

This document contains information on the Monte Carlo neutron ray-tracing program McStas version 2.1, building on the initial release in October 1998 of version 1.0 as presented in Ref. [LN99]. 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.

If you don't want to read this manual in full, go directly to the brief introduction in chapter 4.1.

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. 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-support@mcstas.org` or consult the McStas home page []. A special bug/request reporting service is available [].

If you **appreciate** this software, please subscribe to the `mcstas-users@mcstas.org` 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.
- P. Willendrup, E. Farhi, E. Knudsen, U. Filges and K. Lefmann, Journal of Neutron Research, **17** (expected 2013)

McStas 2.1 contributors

Several people outside the core developer team have been contributing to McStas 2.1:

- UPDATE THIS LIST!!

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

Third party software included in McStas are:

- UPDATE THIS LIST!!!
- perl Math::Amoeba from John A.R. Williams `J.A.R.Williams@aston.ac.uk`.

- perl Tk::Codetext from Hans Jeuken `haje@toneel.demon.nl`.
- scilab Plotlib from Stéphane Mottelet `mottelet@utc.fr`.
- and optionally PGPLOT from Tim Pearson `tjp@astro.caltech.edu`.

The McStas project has been supported by the European Union through"XENNI / Cool Neutrons" (FP4), "SCANS" (FP5), "nmi3/MCNSI" (FP6). McStas was supported directly from the construction project for the ISIS second target station (TS2/EU), see []. Currently McStas is supported through Danish *in-kind* work packages toward the European Spallation Source (ESS), see [] and the European Union through "nmi3/E-learning" and "nmi3/MCNSI7" (FP7) - see the home pages [;].

1. Introduction to McStas

Efficient design and optimization of neutron spectrometers are formidable challenges, which are efficiently treated by Monte Carlo simulation techniques. When McStas version 1.0 was released in October 1998, except for the NISP/MCLib program [], 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 130 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 core McStas package is written in ISO-C, with various tools based on Perl and Python and is freely available for download from the McStas website []. The package is actively being developed and supported by DTU Physics, Institut Laue Langevin (ILL), Paul Scherrer Institute and the Niels Bohr Institute (NBI). The system is well tested and is supplied with several examples and with an extensive documentation. Besides this manual, a separate component manual exists.

The release at hand McStas 2.1 is a major upgrade from the last 1.12c release, meaning partial loss of backward compatibility - especially in terms of uniform naming of component input parameters. Porting your existing personal instrument files and components should be trivial, but if you experience problems feel free to contact `mcstas-users@mcstas.org` or the authors.

1.1. Development of Monte Carlo neutron simulation

The very 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 [See+00], ResTrax [SK97], McStas , Vitess [Wec+00;], IDEAS [LW02] and IB (Instrument Builder) []. Supplementing the Monte Carlo based methods, various analytical phase-space simulation methods exist, including Neutron Acceptance Diagram Shading (NADS) []. 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 [Cop93; Far+02; Sch+04], instrument optimization and design [ZLa04; Lie05]. 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 asa major assett 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.

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 [Mas+95; Cla+98; Lef+00]), but also to plan completely new instruments for new sources (*e.g.* the Spallation Neutron Source, SNS [] and the planned European Spallation Source, ESS []). 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 fulfil 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. Similarly, the PSI has applied McStas extensively for instrument design and upgrades, provided important component additions and contributed several systematic comparative studies of the European instrument Monte Carlo codes. Hence, PSI has also become a part of the core McStas team. Since year 2001 Risø was no longer a neutron source, and the authors from that site have moved on to positions at University of Copenhagen (NBI) and Technical University of Denmark (DTU Physics), hence these two partners have joined the core McStas team. It is further envisioned that the future ESS Data Management and Software Centre (ESS DMSC) will contribute to the project in the future.

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.

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, for simple neutron beam configurations, may be modelled with analytical methods. This is valid for individual elements such as guides [MS63; Mil90], choppers [Low60; Cop03], Fermi choppers [FMM47; Pet05], velocity selectors [Cla+66], monochromators [Fre83; Sea97; SST02; Ali04], and detectors

[Rad74; Pes+89; Man+04]. In the case of a limited number of optical elements, the so-called acceptance diagram theory [Mil90; Cop93; Cus03] may be used, within which the neutron beam distributions are considered to be homogeneous, triangular or Gaussian. However, real neutron instruments are constituted of a large number of optical elements, and this brings additional complexity by introducing strong correlations between neutron beam parameters like divergence and position - which is the basis of the acceptance diagram method - but also wavelength and time. The usual analytical methods, such as phase-space theory, then reach their limit of validity in the description of the resulting effects.

In order to cope with this difficulty, Monte Carlo (MC) methods (for a general review, see Ref. [Jam80]) may be applied to the simulation of neutron instruments. The use of probability is common place in the description of microscopic physical processes. Integrating these events (absorption, scattering, reflection, ...) over the neutron trajectories results in an estimation of measurable quantities characterizing the neutron instrument. Moreover, using variance reduction (importance sampling) where possible, reduces the computation time and gives better accuracy.

Early implementations of the MC method for neutron instruments used *home-made* computer programs (see [Cop+86; MPC77]) but, more recently, general packages have been designed, providing models for most optical components of neutron spectrometers. The most widely-used packages are NISP [See+00], ResTrax [SK97], McStas [LN99;], Vitess [Wec+00], and IDEAS [LW02], which allow a wide range of neutron scattering instruments to be simulated.

The neutron ray-tracing Monte Carlo method has been used widely for guide studies [Cop93; Far+02; Sch+04], instrument optimisation and design [ZLa04; Lie05]. 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, resolution and generally the optimum parameter set, benefit considerably from MC methods, see 3.

Neutron instrument resolution (in q and E) and flux are often limitations in the experiments. This then motivates instrument responsibles to improve the resolution, flux and 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 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 [Cop+86]. Monte-Carlo techniques also provide means to evaluate some of these quantities.

Technicalities of Monte-Carlo simulation techniques are explained in detail in Chapter 3.

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* (also known as *domain-specific language*), specially designed for simulating neutron scattering instruments. Simulations are written in this language by the user, and the McStas compiler automatically translates them into efficient simulation programs written in ISO-C.

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

1. Graphical user interface and scripting layer, presentation of the calculations, graphical or otherwise. (aka. the tool layer).

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

3. 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.)

4. 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.

If you don't want to read this manual in full, go directly to the brief introduction in chapter 4.1.

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 monitors 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 concerns Monte Carlo techniques and simulation strategies in general
- Chapter 4 includes a brief introduction to the McStas system (section 4.1) as well a section (4.2) on running the compiler to produce simulations. Section 4.3 explains how to run the generated simulations. Running McStas on parallel computers require special attention and is discussed in section 4.6. 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 4.4.
- The McStas meta-language is described in chapter **??**. This chapter also describes a set of library functions and definitions that aid in the writing of simulations. See appendix B 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 [], but a short overview of these components is given in chapter 6 of the Manual.

As of this release of McStas support for simulating neutron polarisation is strongly improved, e.g. by allowing nested magnetic fields, tabulated magnetic fields in numerical inputfiles and by close to "full" support of polarisation in all components. As this is the first stable release with these new features, functionality is likely to change. To reflect this, the documentation is still only available in the appendix of the Component manual. A list of library calls that may be used in component definitions appears in appendix B, and an explanation of the McStas terminology can be found in appendix C

of the Manual.. Plans for future extensions are presented on the McStas web-page [] as well as in section 2.6.

2. New features in McStas 2.1

This version of McStas implements both new features, as well as many bug corrections. Bugs are reported and traced using the McStas Trac Ticket system []. 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 guarantee that the software is bullet proof, but we do our best to correct bugs, when they are reported.

2.1. Kernel

The following changes concern the 'Kernel' (i.e. the McStas meta-language and program). See the dedicated chapter for more details 5.

- The `STATE PARAMETERS` keywords has been removed from components code. We now assume that the neutron state parameters are `x,y,z,vx,vy,vz,sx,sy,sz,t`. McStas 1.X components will need to be adapted by commenting this line.
- The `PREVIOUS` and `MYSELF` keywords can be used in component instance parameters and positioning (AT/ROTATED) so that one can make use of `MC_GETPAR(PREVIOUS, parameter)` directly in the TRACE section.

2.2. Run-time

- The number of neutron counts can now exceed 2.10^9, being stored as a `long long`.
- The writing of files has been improved. The `DETECTOR_OUT` functions now return a `mcdetector` structure which holds all written information. Simulations also write a 'content.sim' file aside the 'mcstas.sim' one, which holds the integrated counts for each monitor.
- A bug has been fixed in the rectangular focusing routine, for large solid angles. Reported by M. Skoulatos, L. Udby and J. Jacobsen.

2.3. Components and Library

We here list the new and updated components (found in the McStas `lib` directory) which are detailed in the *Component manual*, also mentioned in the *Component Overview* of the *User Manual*. Generally, most component parameter names have been uniformized.

2.3.1. Components

- `Single_crystal.comp` now provides a lattice curvature option (M. Schulz, FRM2).
- `PowderN.comp` now handles strain (E. Farhi, R. Rogge).
- `Monitor_nD` was wrong in the determination of the flux per cm2 and steradians.
- `Guide_anyshape` allows to model any reflecting geometry from a set of vertices and polygons.
- `Lens` models refractive lenses and prisms (E. Farhi/C. Monzat, ILL).
- `Lens_simple` for refractive lenses with a simple analytical model (H. Frielinghaus).
- `PSD_Detector` can now work in event mode (E. Farhi).
- `Multilayer_Sample` to model a set of refractive layers for e.g. reflectometry. Requires GNU Scientific Library to be previously installed (R. Dalgliesh, ISIS).

2.3.2. New example instruments and Data files

- Updated ILL instrument models: H16 guide, IN22, IN12, D16
- New ILL instrument models: IN1, IN8, IN20, D2B, D4, IN5
- Added HZB NEAT ToF spectrometer model (E. Farhi, R. Lechner)
- Adde ISIS_CRISP which uses new multilayer sample (R. Dalgliesh, ISIS)
- Sort test instruments in various categories
- Update of many data files.

2.4. Documentation

- Manual and component manual updated

2.5. Tools, installation

2.5.1. New tool features

- Support for per-user mcstas_config.perl file, located in `$HOME/.mcstas/` . This folder is also the default location of the 'host list' for use with MPI or gridding, simply name the file 'hosts'.
- mcgui Save Configuration for saving chosen settings on the 'Configuration options' and 'Run dialogue'.

- Possibilty to run MPI or grid simulations by default from mcgui.
- When scanning parameters, mcrun now terminates with a relevant error message if one or more scan steps failed (intensities explicitly set to 0 in those cases).
- When running parameter optimisations, a logfile (default name is "mcoptim_XXXX.dat" where XXXX is a pseudo-random string) is created during the optimisation, updated at each optim step.
- We now provide syntax-highlighting setup files for vim and gedit editors.
- Rudimentary support for GNUPLOT when plotting with mcplot. Data file format is standard McStas/PGPLOT.

2.5.2. Platform support

- Mac OS X 10.3 Panther (ppc), 10.4 Tiger (pcc/intel), 10.5 Leopard (ppc/intel)
- Windows XP, Windows Vista (Now with a recent perl version; 5.10 plus various fixes). New feature on Windows: Simulations *always* run in the background, freeing mcgui for other work.
- "Any" Linux - reference platforms are Ubuntu 8.04 (and earlier) and Debian 4.0 (and earlier). We have also tested Fedora 8, OpenSuSE 10.3 and CentOS 4 releases recently.
- FreeBSD (FreeBSD release 6.3 and its cousin DesktopBSD 1.6 recently tested)
- SUN Solaris 10 (Intel tested, Sparc probably OK)
- Plus probably any UNIX/POSIX type environment with a bit of effort...

2.5.3. Various

- A number of minor bugs ironed out, both in components, runtime code and tools.
- From release 1.12, McStas is GPL 2 only. The debate on the internet about the GPL 3 license suggests that this license might have implications on the 'derived work', hence have implications on what and how our users use their McStas simulations for. To protect user freedom, we will stick with GPL 2.

2.5.4. Warnings

WARNING: The 'dash' shell which is used as /bin/sh on some Linux system (including Ubuntu 7.04) makes the 'Cancel' and 'Update' buttons fail in mcgui. Solutions are:

a) If your system is a Debian or Ubuntu, please dpkg-reconfigure dash and say 'no' to install dash as /bin/sh

b) If you run another Linux with /bin/sh beeing dash, please install bash and manually change the /bin/sh link to point at bash.

2.6. Future extensions

The following features are planned for the oncoming releases of McStas (not an ordered list):

- Increased validation and testing.
- Extend test cases to all (most) components. One instrument pr. component. (Probably not in `examples/`.
- Updates to mcresplot to support the Matlab backend.
- Global changes of components relating to polarisation visualisation.
- Visualisation of neutron spins in magnetic fields for all graphical backends.
- *Array* `AT` specifiers for components, i.e.
  `COMPONENT MyComp=Comp(...)`
  `AT([Xarray],[Yarray],[Zarray])` and
  `AT Positions('filename')`
- Gui support for array AT specifiers.
- More complete polarisation support including numerically defined magnetic fields and advanced sample components.
- Perl or python plotting alternative to PGPLOT.
- Larger variety of sample components.

3. 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. More details are available in the Appendix concerning random numbers.

3.1. Neutron spectrometer simulations

3.1.1. Monte Carlo ray tracing simulations

The behaviour of a neutron scattering instrument can in principle be described by a complex integral over all relevant parameters, like initial neutron energy and divergence, scattering vector and position in the sample, etc. However, in most relevant cases, these integrals are not solvable analytically, and we hence turn to Monte Carlo methods. The neutron ray-tracing Monte Carlo method has been used widely for guide studies [Cop93; Far+02; Sch+04], instrument optimisation and design [ZLa04; Lie05]. 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, resolution and generally the optimum parameter set, benefit considerably from MC methods. Mathematically, the Monte-Carlo method is an application of the law of large numbers [Jam80; GRR92]. 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.

$$\lim_{n\to\infty} \frac{1}{n} \sum_{i=1, a\leq u_i \leq b}^{n} f(u_i) = \frac{1}{b-a} \int_a^b f(u)du \tag{3.1}$$

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 uniformly) 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 consult James [Jam80] for a detailed review on the Monte-Carlo method.

3.2. 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 relevant detector or monitor.

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 later 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 n components in the instrument becomes the product of all contributions

$$p = p_n = p_0 \prod_{j=1}^{n} \pi_j. \tag{3.2}$$

Each adjustement factor should be $0 < \pi_j < 1$, except in special circumstances, so that total flux can only decrease through the simulation, see section 3.3. 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 and windows.
- Transmission through Soller blade collimators and velocity selectors (in the approximation which does not take each blade into account).
- Reflection from monochromator (and analyser) crystals with finite reflectivity and mosaicity.
- Reflection from guide walls.
- Passage of a continuous beam through a chopper.
- Scattering from all types of samples.

3.2.1. 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

$$I = \sum_i p_i = N\overline{p}, \tag{3.3}$$

where N is the number of rays hitting the detector and the horizontal bar denotes 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

$$\sigma(I) = \sqrt{N}\overline{p} = I/\sqrt{N}, \tag{3.4}$$

as is used in real neutron experiments (where $\overline{p} \equiv 1$). For a better approximation we return to Eq. (3.3). Allowing variations in both N and $\overline{p}$, we calculate the variance of the resulting intensity, assuming that the two variables are statistically independent:

$$\sigma^2(I) = \sigma^2(N)\overline{p}^2 + N^2\sigma^2(\overline{p}). \tag{3.5}$$

Assuming as before 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 some 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

$$\sigma^2(I) = N\left(\overline{p}^2 + \sigma^2(p_i)\right). \tag{3.6}$$

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

$$\sigma^2(I) = \frac{N}{N-1}\left(\sum_i p_i^2 - \overline{p}^2\right). \tag{3.7}$$

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

$$\sigma^2(I) \approx \sum_i p_i^2. \tag{3.8}$$

As a consistency check, we note that for all p_i equal, this reduces to eq. (3.4)

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

3.3. 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

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

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 P, but we make the Monte Carlo choice of transmitting the neutron ray each time: $f_{\mathrm{MC}} = 1$. This must be reflected on the choice of weight multiplier $\pi_j = P$. Of course, one could simulate without weight factor transformation, in our notation written as $f_{\mathrm{MC}} = P, \pi_j = 1$. To generalize, weight factor transformations are given by the master equation

$$f_{\mathrm{MC}}\pi_j = P. \tag{3.9}$$

This probability rule is general, and holds also if, e.g., it is decided to transmit only half of the rays ($f_{\mathrm{MC}} = 0.5$). 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. This weight transformation is much more complex than described above, but still boils down to obeying the master transformation rule 3.9.

3.3.1. 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. This implies $f_{\mathrm{MC}}(\Delta\Omega) = 1$. However, if the physical events are distributed uniformly over the unit sphere, we would have $P(\Delta\Omega) = \Delta\Omega/(4\pi)$, according to Eq. (3.9). 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 reached the final detector.

3.4. Adaptive and Stratified sampling

Another strategy to improve sampling in simulations is *adaptive importance sampling* (also called variance reduction technique), 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.

An other class of efficiency improvement technique is the so-called *stratified sampling*. It consists in partitioning the event distributions in representative sub-spaces, which are then all sampled individualy. The advantage is that we are then sure that each sub-space

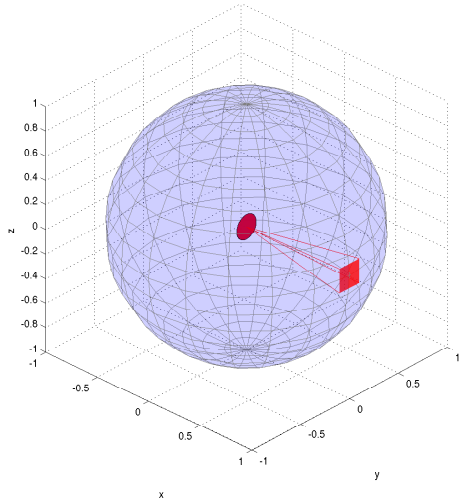


Figure 3.1.: Illustration of the effect of direction focusing in McStas. Weights of neutrons emitted into a certain solid angle are scaled down by the full unit sphere area.

is well represented in the final integrals. This means that instead of shooting N events, we define D partitions and shoot $r = N/D$ events in each partition. In conjunction with adaptive sampling, we may define partitions so that they represent 'interesting' distributions, e.g. from events scattered on a monochromator or a sample. The sum of partitions should equal the total space integrated by the Monte Carlo method, and each partition must be sampled randomly.

In the case of McStas, an ad-hoc implementation of adaptive stratified is used when repeating events, such as in the Virtual sources (Virtual_input, Vitess_input, Virtual_mcnp_input, Virtual_tripoli4_input) and when using the SPLIT keyword in the TRACE section on instrument descriptions. We emphasize here that the number of repetitions r should not exceed the dimensionality of the Monte Carlo integration space (which is $d = 10$ for neutron events) and the dimensionality of the partition spaces, i.e. the number of random generators following the stratified sampling location in the instrument.

3.5. Accuracy of Monte Carlo simulations

When running a Monte Carlo, the meaningfull quantities are obtained by integrating random events into a single value (e.g. flux), or onto an histogram grid. The theory [Jam80] shows that the accuracy of these estimates is a function of the space dimension d and the number of events N. For large numbers N, the central limit theorem provides an estimate of the relative error as $1/\sqrt{N}$. However, the exact expression depends on the random distributions.

Records	Accurarcy
10^3	10 %
10^4	2.5 %
10^5	1 %
10^6	0.25 %
10^7	0.05 %

Table 3.1.: Accuracy estimate as a function of the number of statistical events used to estimate an integral with McStas.

McStas uses a space with $d = 10$ parameters to describe neutrons (position, velocity, spin, time). We show in Table 3.1 a rough estimate of the accurarcy on integrals as a function of the number of records reaching the integration point. This stands both for integrated flux, as well as for histogram bins - for which the number of events per bin should be used for N.

4. Running McStas

This chapter describes usage of the McStas simulation package. In case of problems regarding installation or usage, the McStas mailing list [] 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 a further release. We recommend to use ActiveState **Perl 5.10**. (Note that as of McStas 1.10, all needed support tools for Windows are bundled with McStas in a single installer file.)

Performing a simulation using McStas can be divided into the following steps/elements

- The structure of McStas is illustrated in Figure 4.1.
- 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 input files (instrument and component files) are written in the McStas metalanguage 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 handling and visualisation tools in McStas based on Perl/Python in combination with `chaco`, `matplotlib`, Matlab, GNUPlot or PGPLOT. Further data output formats including NeXus are available, see section 4.5.

4.1. 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

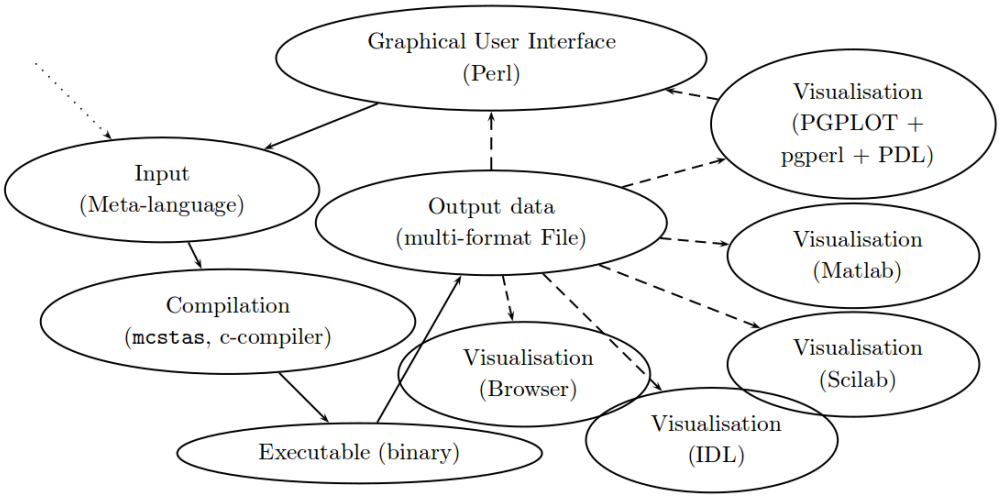


Figure 4.1.: An illustration of the structure of McStas.

it. For details on the different steps, see the following sections. This section uses the `Samples_vanadium.instr` file supplied in the `examples/` directory of the McStas distribution.

To start the graphical user interface of McStas, run the `mcgui` command which will open a window with a number of menus, see figure 4.2. To load an instrument,

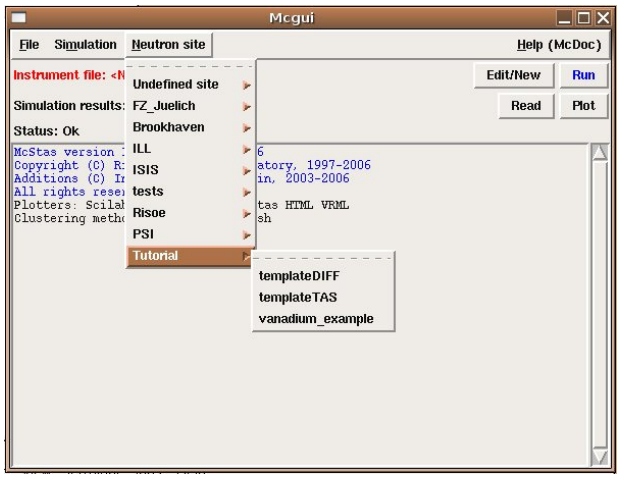


Figure 4.2.: The graphical user interface `mcgui`.

select “Tutorial” from the “Neutron site” menu and open the file `Samples_vanadium`.

Next, check that the current plotting backend setting (select "Choose backend" from the "Simulation" menu) corresponds to your system setup.

- 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 4.3. When using the Matlab backend, full 3D

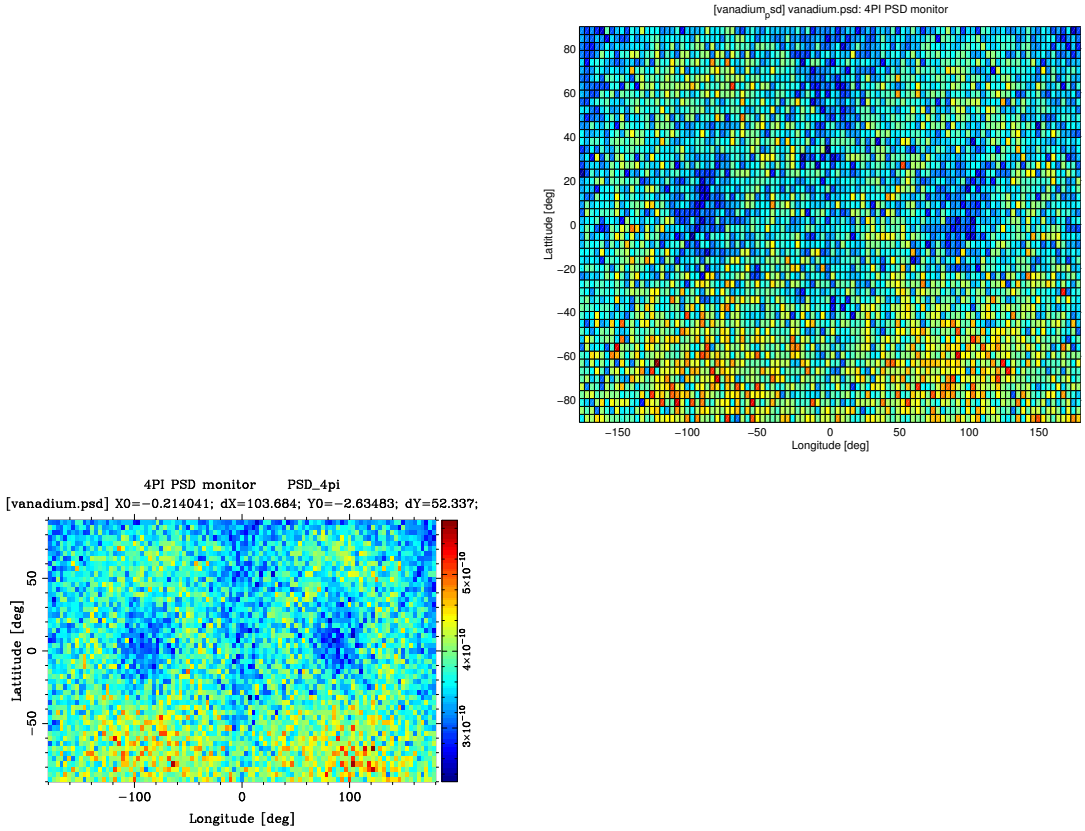


Figure 4.3.: Output from `mcplot` with PGPLOT and Matlab backends

view 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 4.4-4.5.

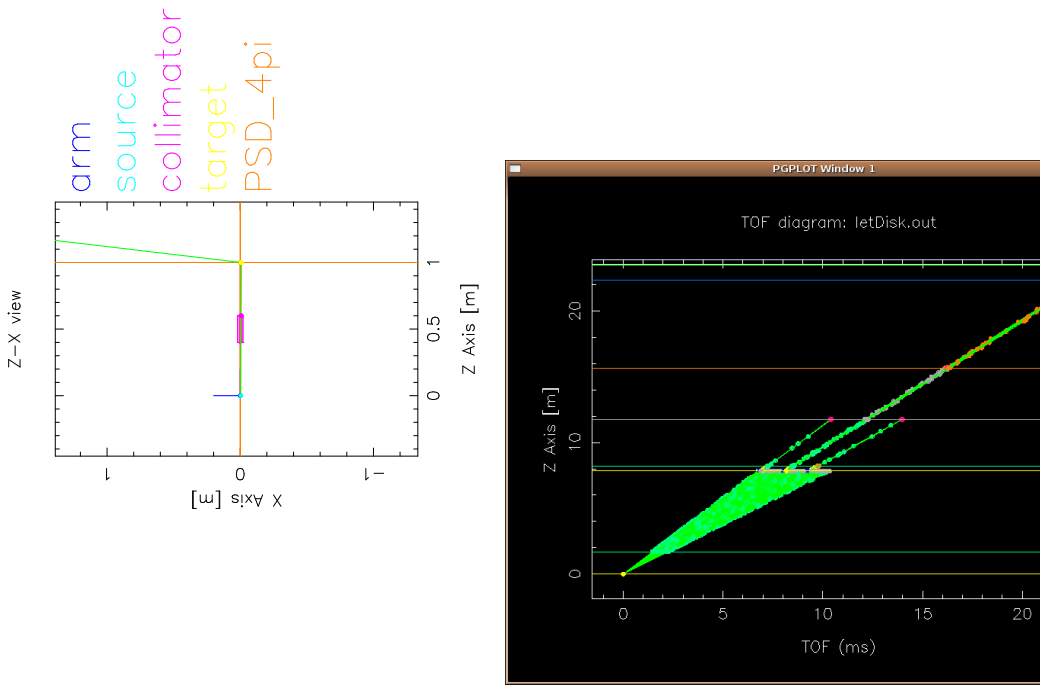


Figure 4.4.: Left: Output from `mcdisplay` with PGPLOT backend. The left mouse button starts a new neutron ray, the middle button zooms, and the right button resets the zoom. The Q key quits the program. Right: The new PGPLOT time-of-flight option. See section 4.4.3 for details.

For a slightly longer gentle introduction to McStas, see the McStas tutorial (available from []), and as of version 2.1 built into the `mcgui` help menu. For more technical details, read on from section 4.2

4.1.1. 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.

- The versions 2.1.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 2.1.x are also

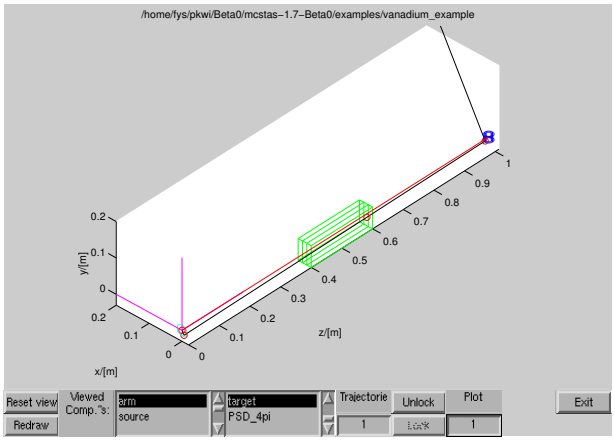


Figure 4.5.: Output from `mcdisplay` with Matlab backend. Display can be adjusted using the window buttons.

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

4.2. 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 4.4.1) or `mcrun` (section 4.4.2). 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

```
1    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 ISO-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:

```
1    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.

4.2.1. Code generation options

By default, the output files from the McStas compiler are in ISO-C with some extensions (currently the only extension is the creation of new directories, which is not possible in

pure ISO-C). The use of extensions may be disabled with the `-p` or `--portable` option. With this option, the output is strictly ISO-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 4.4.3.

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, e.g.

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

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

4.2.2. 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 **??** for details. The location can be overridden by setting the environment variable `MCSTAS`:

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

for csh/tcsh users, or

```
1    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:

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

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

4.2.3. 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.

4.2.4. 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 ISO-C standard, so it should be easy to compile it using any ISO-C (or C++) compiler. *E.g.* a typical Unix-style command would be

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

The McStas team recommends these compiler alternatives for the Intel (and AMD) hardware architectures:

A `gcc` which is a very portable, open source, ISO-C compatible c compiler, available for most platforms. For Linux it is usually part of your distribution, for Windows the McStas distribution package includes a version of `gcc` (in the Dev-CPP sub-package), and for Mac OS X `gcc` is part of the Xcode tools package available on the installation medium.

B `icc` or the Intel c compiler is available for Linux, Mac OS and Windows systems and is a commercial software product. Generally, simulations run with the Intel compiler are **a factor of 2 faster** than the identical simulation run using `gcc`. To use `icc` with McStas on Linux or Mac OS X, set the environment variables

- `MCSTAS_CC=icc`
- `MCSTAS_CFLAGS="-g -O2 -wd177,266,1011,181"`

To use `icc` with MPI on Unix system (see Section 4.6) installations, it seems that *editing* the mpicc shell script and setting the CC variable to "`icc`" is the only requirement! On Windows, the Intel c compiler is 'icl', not 'icc' and has a dependency for Microsoft Visual C++. If you have both these softwares available, running McStas with the Intel compiler should be possible (currently untested by the McStas developer team).

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 []):

- 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 ISO-C compiler, using the options `-Aa +Oall -Wl,-a,archive` (the `-Aa` option is necessary to enable the ISO-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 ISO-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 highly 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 **??**). Refer to section 4.3.4 for other optimization methods.

4.3. 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.

Simple McStas options

In this section, the most common simulation parameters are discussed. For a full list, please consult tables **??**

The simplest way is to run it directly from the command line or shell:

```
1    ./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 angular settings, and then run the simulation. Default instrument parameter values (see section 5.3), if any, will be indicated and entered when hitting the `Return` key. This way of running McStas will only give data for one instrument setting which is normally sufficient for *e.g.*, time-of-flight, SANS or powder instruments, but not for *e.g.* continuous-beam reflectometers or triple-axis spectrometers where a scan over various instrument 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:

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

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

```
1    ./Samples_vanadium.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). If

the `file` is given as `NULL`, the file name is automatically built from the instrument name and a time stamp. The default file name is `mcstas` followed by appropriate extension.

The complete list of options and arguments accepted by McStas simulations appears in Tables 4.1 and 4.2.

4.3.1. 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 4.5. 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. McStas can presently generate the McStas/PGPLOT and the NeXus format.

It is also possible to create and read *Vitess*, *MCNP/PTRAC* and *Tripoli4/batch* neutron event files using components

- `Vitess_input` and `Vitess_output`
- `Virtual_tripoli4_input` and `Virtual_tripoli4_output`
- `Virtual_mcnp_input` and `Virtual_mcnp_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 3.2.1). The former representation is fully additive, and thus enables to add results from separate simulations (e.g. when using a computer Grid - which is automated in the `mcformat` tool). Other acceptable format modifiers are `transpose` to transpose data matrices and `append` to concatenate data to existing files.

4.3.2. Basic import and plot of results

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

```
1    matlab> s=mcstas;
2    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 IDL

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

See section 4.4.4 for another way of plotting simulation results using the `mcplot` frontend.

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.

`-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.
`-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`	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`	Disables the writing of data files (output to the terminal, such as detector intensities, will still be written).
`-g` `--gravitation`	Toggles the gravitation (approximation) handling for the whole neutron propagation within the instrument. May produce wrong results if the used components do no comply with this option.
`--format=`*FORMAT*	Sets the file format for result simulation and data files.
`-N` *STEPS*	Divide simulation into STEPS, varying parameters within given ranges 'min,max'.
param=value *min,max*	Set the value of an instrument parameter, rather than having to prompt for each one. Scans ranges are specified as 'min,max'.

Table 4.1.: Options accepted by McStas simulations. For options specific to MPI and parallel computing, see section 4.6.

`-f` *file* `--file=`*file*	Write all data into a single file *file*. Avoid when using binary formats.
`--format_data=`*FORMAT*	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).
`--mpi=`*NB_CPU*	Distributes the simulation over NB_CPU node (requires MPI to be installed). Speedup has been demonstrated to be linear in number of nodes when the simulation task is `--ncount` is sufficiently large.
`--multi=`*NB_CPU* `--grid=`*NB_CPU*	Distributes the simulation over NB_CPU node (requires SSH to be installed). Speedup has been demonstrated to be linear in number of nodes when the simulation task is `--ncount` is sufficiently large.
`--machines=`*MACHINES*	Specify a list of distant machines/nodes to be used for MPI and grid clustering. Default is to use local SMP cluster.
`--optim`	Run in optimization mode to find best parameters in order to maximize all monitor integral values. Parameters to be varied are given just like scans (min,max).
`--optim=`*COMP*	Same as `--optim` but for specified monitors. This option may be used more than once.
`--optim-prec=`*ACCURACY*	Sets accuracy criteria to end parameter optimization (default is 10^{-3}).
`--test`	Run McStas self test.
`-c` `--force-compile`	Force to recompile the instrument.

Table 4.2.: Additional options accepted by McStas simulations.

4.3.3. 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. This feature is not available when using MPI parallelization.

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 resembling

```
1 <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 4.3 using the 'kill' unix command (or the functionalities of your process manager), e.g.

```
1     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.

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

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 interrupt 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 interrupted), 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 by the user.

Additionally, any system error will result in similar messages, giving indication about the occurrence 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.

4.3.4. Optimizing simulation speed

There are various ways to speed up simulations

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

Table 4.3.: Signals supported by McStas simulations.

- Optimize the compilation of the instrument, as explained in section 4.2.4.
- Execute the simulation in parallel on a computer grid or a cluster (with MPI or ssh grid) as explained in section 4.6.
- Divide simulation into parts using a file for saving or generating neutron events. In this way, a guide may be simulated only once, saving the neutron events at the guide exit 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 preferred whenever possible, at least in the beginning of a simulation project.
- The SPLIT keyword may artificially repeat events reaching specified positions in the instrument. This is *very* efficient, but requires to cast random numbers in the course of the remaining propagation (e.g. at samples, crystals, ...). See section 5.4.6 for details.

A general comment about optimization is that it should be used cautiously, checking that the results are not significantly affected.

4.3.5. Optimizing instrument parameters

Often, the user may wish to optimize the parameters of a simulation, i.e. the best geometry of a given component, for example the optimal curvature of a monochromator.

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. The user is advised to be cautious when interpreting the optimization results.

Using iFit for optimization

One of the authors of McStas has developed a very flexible and general data analysis and fitting package called iFit**??** based on Matlab. Matlab itself is not required, as a stand-alone distributable binary of iFit exists.

iFit contains wrapper functionality for compiling and running McStas simulations as object functions, and allows to select many different optimizers, including swarms and other non-gradient methods. Please see the iFit documentation for more information.

Our experience is that iFit together with McStas is a more robust optimization solution than the McStas built-in Simplex solution.

Using the Simplex method

The McStas package comes with a Simplex optimization method to find best instrument parameters in order to maximize all or some specified monitor integrated values. It uses the Downhill Simplex Method in Multidimensions [NM65; Pre+02] which is a geometric optimization method somewhat similar to genetic algorithms. It is not as fast as the gradient method, but is much more robust. It is well suited for problems with up to about 10-20 parameters to optimize. Higher dimensionalities are not guarantied to converge to a meaningful solution.

When using `mcrun` (section 4.4.2), the optimization mode is set by using the `--optim` option or a list of monitors to maximize with as many `--optim=COMP` as required. The optimization accuracy criterion may be changed with the `--optim-prec=accuracy` option.

From `mcgui` (section 4.4.1), one should choose the 'Optimization' execution mode (instead of the Simulation or Trace mode). Then specify the instrument parameters to optimize by indicating their variation range `param=min,max` (e.g. Lambda=1,4) just like parameter scans. Optionally, the starting guess value might be given with the syntax `param=min,guess,max`. The optimization accuracy criterion is controlled using the 'Precision' entry box in the configuration options (See Figure 4.6). Finally, run the simulation. The optimum set of parameters is then printed at the end of the simulation process. You may ask to maximize only given monitors (instead of all) by selecting their component names in the lower lists in the Run Dialog (up to 3).

If you would like to maximize the flux at a given monitor, with some divergence constrains, you should for instance simply add a divergence collimator before the monitor. Alternatively, write a new component that produce the required 'figure-of-merit'.

The optimization search interval constrains the evolution of parameters. It should be chosen carefully. In particular it is safer for it to indeed contain a high signal domain, and be preferably symmetric with respect to that maximum.

Using custom optimization routines

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.
- executes 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 4.5 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, 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.

4.4. 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 build 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
```

```
9         mcstas2vitess McStas to Vitess component translation utility
10        mcformat      Conversion tool for text files and MPI/grids
11        mcformatgui   GUI for mcformat
12        mcdaemon      Instrument results on-line plotting
13    When used with the -h flag, all tools display a specific help.
14    SEE ALSO: mcstas, mcdoc, mcplot, mcrun, mcgui, mcresplot, mcstas2vitess
15    DOC:      Please visit http://www.mcstas.org
```

4.4.1. The graphical user interface (mcgui)

The front-end `mcgui` provides a graphical user interface that interfaces the various parts of the McStas package. It may be started with the single command

```
1    mcgui
```

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

Dependencies: To run the `mcgui` front-end, the programs Perl and Perl/Tk must be properly installed on the system. Additionally, to use the McStas/PGPLOT back-end the software packages PGPLOT, PgPerl, and PDL are 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.

The menus

When the front-end is started the main window is opened (see figure 4.2). This window displays the output from compiling and running simulations, and contains a few menus and buttons for easy navigation. The main purpose of the front-end is to edit and compile instrument definitions, run the simulations, and visualize the results.

The **File** menu has the following features:

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

File/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.

File/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.

File/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. This might also be required when choosing MPI and NeXus options .

File/Save log file saves the text in the window showing output of compilations and simulations into a file.

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

File/Preferences Opens the choose backend dialog shown in figure 4.6. Several settings can be chosen here:

- Selection of the desired (PGPLOT—Matlab—HTML/VRML) output format and possibility to save 'binary files' when applicable (improved disk I/O).
- One- or three-pane view of your instrument in trace mode when using PGPLOT.
- Clustering option (None—MPI—ssh)
- Choice of editor to use when editing instrument files.
- Automatic quotation of strings when inserting in the built-in editor.
- Possibility to *not* optimize when compiling the generated c-code. This is very handy when setting up an instrument model, which requires regular compilations.
- Adjustment of final precision when doing parameter optimization.

To save the chosen settings for your next McStas run, use Save Configuration in the File menu.

File/Save configuration saves user settings from Configuration options and Run dialogue to disk.

File/Quit exits the graphical user interface front-end.

The **Simulation** menu has the following features:

Simulation/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.

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

Simulation/Plot results plots (using `mcplot`) the results of the last simulation run or spawns a load dialogue to load a set of results.

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 transferred to the active directory (so that users have

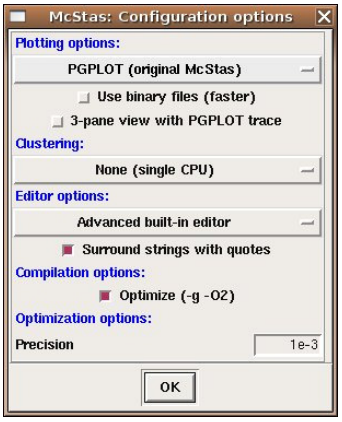


Figure 4.6.: The "configuration options" dialog in `mcgui`.

modification rights) and loaded. One may then view its source (Edit) and use it directly for simulations/trace (3D View).

The **Tools** menu gathers minor tools.

Tools/Plot current/other results Plot current simulation results and other results.

Tools/Online plotting of results installs a DSA key to be used for ssh clustering and MPI (see Section 4.6).

Tools/Dataset convert/merge Opens a GUI to the `mcformat` tool, in order to convert datasets to other formats, merge scattered dataset (e.g. from successive or grid simulations), and assemble scan sets. This tool does not handle raw event files.

Tools/Shortcut keys displays the shortcut keys used for running and editing instruments.

Tools/Install DSA key installs a DSA key to be used for ssh clustering and MPI (see Section 4.6).

The Histogrammer In addition to these tools, the `Neutron site/Tools/Histogrammer.instr` example instrument may read McStas, Vitess, MCNP and Tripoli event files in order to generate histograms of any type.

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/mozilla/firefox` on Unix/Linux and the default browser on Windows.

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

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

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

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

Help/Tutorial opens the McStas tutorial for a quick start.

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

Help/Test McStas installation 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.

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

The run dialog

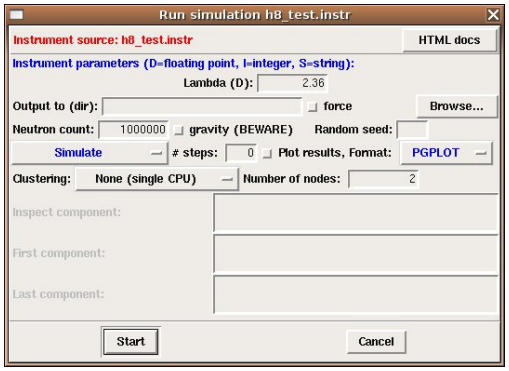


Figure 4.7.: The run dialog in `mcgui`.

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 4.3 for details). It also allows to run the `mcdisplay` (section 4.4.3) and `mcplot` (section 4.4.4) front-ends together with the simulation.

The meaning of the different fields is as follows:

Run: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 and optimizations, enter the minimum and maximum values to scan/optimize, separated by a comma, e.g. `1,10` and do not forget to set the **# Scanpoints** to more than 1.

Run: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.

Run:Force Forces McStas to overwrite existing data files

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

Run:Gravity Activates gravitation handling. Not all components full support the use of gravitation, but all transport in "free space" using the PROP_DT, PROP_Z0 etc. macros will include propagation with gravity. Only local, internal component propagation without the PROP routines will be gravity-less. As a conclusion it is considered safe and to high precision correct to apply the gravitation setting if one takes care to use the Guide_gravity component and other gravity-supporting guide types in combination with non-gravity components that are "small" in size, i.e. samples, lenses, etc.

Run: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).

Run:Simulate/Trace (3D)/Optimize selects between several modes of running the simulation:

- Simulate: perform a normal simulation or a scan when #steps is set to non-zero value
- Trace (3D view): View the instrument in 3D tracing individual neutrons through the instrument
- Optimize: find the optimum value of the simulation parameters in the given ranges (see section 4.3.5).
- Backgrounding (bg): Simulate or Optimize in the background.

Run:# steps / # optim sets the number of simulation to run when performing a parameter scan or the number of iterations to perform in optimization mode.

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

Run:Format quick selection of output format. Binary mode may be checked from the "Simulation/Configuration options" dialog box.

Run:Clustering method selects the mechanism to be used for running on grids and clusters. See section 4.6 on parallel computing for more informations.

Run:Number of nodes sets the number of nodes to use for MPI/ssh clustering.

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

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

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

Run:Maximize monitor (Optimization mode) seletcs up to three monitors which integral value should be maximized, varying instrument parameters. If no monitor is selected, the sum of all monitors is optimized.

Run:Start runs the simulation.

Run:Cancel aborts the dialog.

Most of the settings on the run dialog can be saved for your next McStas run using 'Save configuration' in the File menu.

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. NB: If components are changed, automatic compilation is *not* performed. Instead, use the File/Compile menu item in mcgui.

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:

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

Editor Insert/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 4.8). See section 5.3 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 5.7 for how to write components to integrate well with this facility.

Editor Insert/Type These menu entries give quick access to the entry dialog for the various component types available, i.e. Sources, Optics, Samples, Monitors, Misc, Contrib and Obsolete.

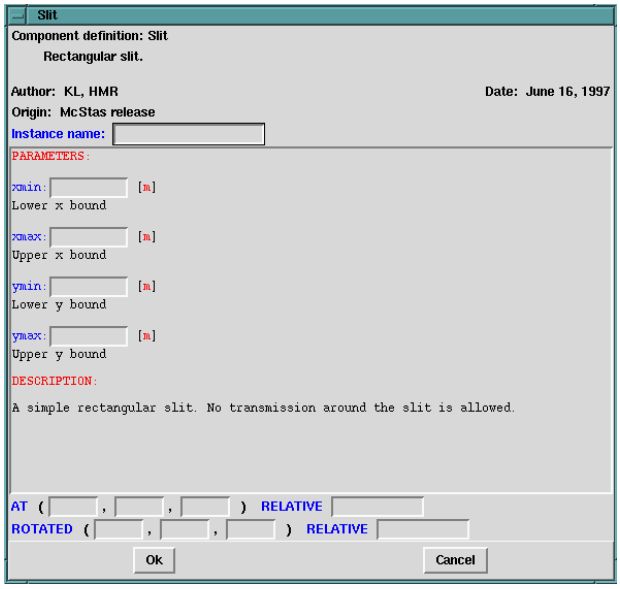


Figure 4.8.: Component parameter entry dialog.

4.4.2. Running simulations on the commandline (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

```
1 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 4.3, 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

```
1 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.m` (when using Matlab 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 4.4.4.

- When performing 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 performing 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.

4.4.3. Graphical display of simulations (mcdisplay)

The front-end `mcdisplay` (mcdisplay.pl on Windows) is a graphical visualization tool, very useful for debugging. 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. (See figures 4.4-4.5.)

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

```
1 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, VRML, and Matlab using the -p*PLOTTER* option. For instance, calling

```
mcdisplay --pMatlab ./Samples_vanadium.out ROT=90+
```

will output graphics using Matlab. The `mcdisplay` front-end can also be run from the `mcgui` front-end. Examples of plotter appearence for `mcdisplay` is shown in figures 4.4-4.5.

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 aspect ratio of the plot may have changed, 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.

As of McStas 1.10, the PGPLOT version has a special mode for time of flight applications. Using the new commandline options `--TOF/-T` and `--tmax=TMAX`, chopper acceptance diagrams can be generated from the statistical information from the simulated neutron rays. As the use in non-interactive, please use with a limited number of neutron rays (`-n/--ncount`). For export of graphics, combine with e.g. `--gif`.

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 back-end 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.

The `mcdisplay` front-end will then require the Perl, and Matlab to be installed.

VRML/OpenGL back-ends When using the `-pVRML` option, the instrument is shown in Virtual Reality (using OpenGL). You may then walk aside instrument, or go inside elements following neutron trajectories. As all neutron trajectories are stored into a VRML file, you better limit the number of stored trajectories below 1000, otherwise file size and processing time becomes significant. The `--inspect` option is not available in VRML format display.

4.4.4. Plotting the results of a simulation (mcplot)

The front-end `mcplot` (mcplot.pl on Windows) is a program that produces plots of all the monitors 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

```
1    mcplot
```

This will plot any simulation data stored in the current directory, which is where simulations store their results by default. If the `--dir` or `--file` options have been used (see section 4.3), 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 Gnuplot using either the -p*PLOTTER* option (e.g. `mcplot -pMatlab file`) or using the current `MCSTAS_FORMAT` environment variable. .

Except for the NeXus format, all other plotters read the legacy McStas/PGPLOT text based data format.

The `mcformat` utility will convert any McStas result into an other data format (see section 4.4.8), but restricting to text data sets. In this case, we recommend to generate data sets using PGPLOT/McStas format, and translate into any other format using `mcformat`.

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 4.4-4.3.

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 back-end 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 function may be called from these language prompt with the same method as in section 4.3, e.g:

```
matlab> s=mcplot;
matlab> help mcplot
matlab> s=mcplot('mcstas.m');
matlab> 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 window will appear (instead of being launched in the current terminal). This will then enable Java support under Matlab, resulting in additional menus and tools. On the other hand, the `-nw` option will force Matlab to run in the current terminal, which is usually faster.

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

4.4.5. Plotting resolution functions (mcresplot)

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 outfile
```

Here, *outfile* 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 port for Matlab may be written in the future.

The front-end will open a window displaying projections of the 4-dimensional resolution function $R(\boldsymbol{Q}, \omega)$, measured at a particular choice of $\boldsymbol{Q}$ and ω, see the component manual. 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.

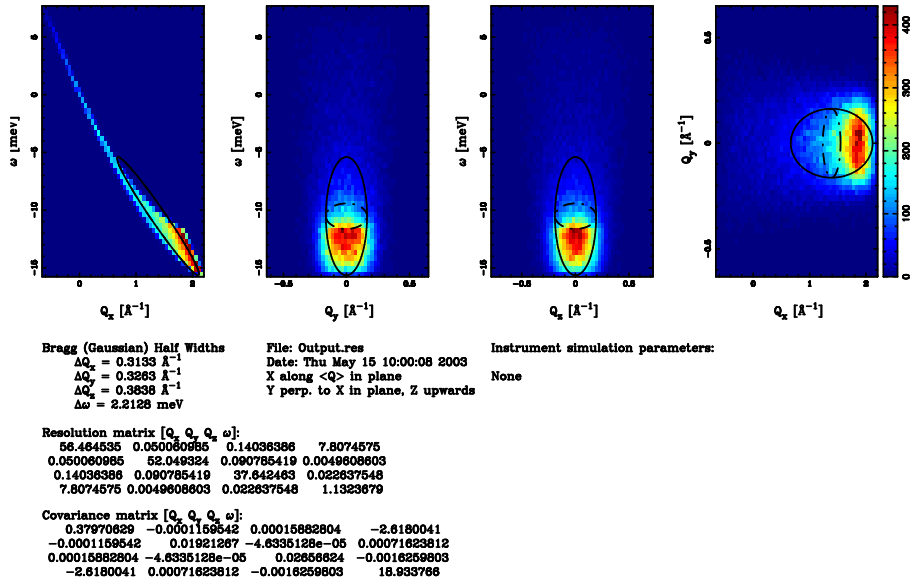


Figure 4.9.: Output from `mcresplot` with PGPLOT backend. Use P, C and G keys to write hardcopy files.

4.4.6. Creating and viewing the library, component/instrument help and Manuals (mcdoc)

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

```
1 mcdoc
2 mcdoc {comp|instr}
3 mcdoc --tools
```

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

Alternatively, if a component or instrument *comp* is specified as in the second example, it will be searched within the library, and an HTML help will be created for all available components matching *comp*.

The last example will list the name and description of all McStas tools.

Additionally, the options `--web`, `--manual` and `--comp` 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 5.7 for more details about the McDoc usage and header format. To use the mcdoc front-end, the program Perl should be available.

4.4.7. 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

```
1 mcstas2vitess Compo.comp
```

This will create a Vitess module of the given component.

Let us assume the component Compo shall be translated. 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 Vitess module 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 labelling.

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).

It is important to notice that, as VITESS uses the standard output stream (stdout) to send neutron events, all information printed to screen must use the *error* stream stderr, so that *all* printf(...) and fprintf(stdout, ...) occurrences should be changed manually into fprintf(stderr, ...).

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

4.4.8. Translating and merging McStas results files (all text formats)

If you have been running a McStas simulation with a given text format output, but finally plan to look at the results with an other plotter (e.g. you ran a simulation with PGPLOT output and want to view it using Matlab), you may use

```
mcformat {file|dir} -d target_dir --format=TARGET_FORMAT
```

to translate files into format TARGET_FORMAT (e.g. NeXus). When given a directory, the translation works recursively. The conversion works only for text files.

The `--merge` option may be used to merge similar files, e.g. obtained from grid systems, just as if a longer run was achieved.

The `--scan` option may be used to reconstruct the scan data from a set of directories which vary by instrument parameters. For instance, you ran a scan, but finally realised you should have prolonged it. Then simply simulate the missing bits, and apply `mcformat -d scan_data --format=PGPLOT --scan step0 .. stepN`. The resulting scan data is compatible with `mcplot` only when generating PGPLOT/McStas format. You may conjugate this option with the `--merge` in order to add/merge similar data sets before re-building the scan.

The data files are analyzed by searching keywords inside data files (e.g. 'Source' for the source instrument description file). If some filenames or component names match these keywords (e.g. using a file 'Source.psd'), the extracted metadata information may be wrong, even though the data itself will be correct.

4.5. Data formats - 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 Matlab, NumPy, IDL. The output files from simulations are usually simple text files containing headers and data blocks. If data blocks are empty they may be accessed referring to an external file indicated in the header.

Each data file contains informations about the simulation, the instrument, the parameters used, and of course the signal, the estimated error on the signal, and the number of events used in each bin. Additionally, all data files indicate their first moment (mean value) and second moment (half width) in the 'statistics' field.

The available data formats are the legacy McStas (McCode) format, and the HDF/NeXus binary when the needed libraries are installed.

In order for the user to choose the data format, we recommend to set it using the `--format=FORMAT` or alternatively *via* the MCSTAS_FORMAT environment variable, which will also make the front-end programs able to import and plot data and instrument consistently (see Section 4.3).

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

4.5.1. 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) 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 3.2.1, 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-dimensional 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 4.4.2).

When using one- and two-dimensional monitors, the integrated intensities are written to terminal as for the single-point monitor type, supplementing file output of the full one- or two-dimensional intensity distribution. Both one- and two-dimensional 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 4.4.4). 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.

4.5.2. NeXus format

The NeXus format [] is a platform independent HDF binary data file. To have McStas use it

1. the HDF and NeXus libraries must have been installed (libNeXus and headers)
2. the compilation of instruments must be done with the `-DUSE_NEXUS -lNeXus` flag (see Section 5.3.4). This is automated with the `mcrun` tool (Section 4.4.2).

All results are saved in a single file, containing 'groups' of data. To view such files, install and use HDFView (or alternatively HDFExplorer). This Java viewer can show

content of all detectors, including metadata (attributes). Basic detector images may also be generated.

4.6. Using computer Grids and Clusters

Parallelizing a computation is in general 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 computers.

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 3.2.1).

- event lists: the merge of events is done by concatenation

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

- when using a set of nodes (grid, cluster or multi-cores), it is possible to distribute simulations on a list of computers and multi-core machines (see section 4.6.1). Results are automatically merged after completion. This method is very efficient, and only requires SSH server to be installed/configured on slave machines. In order to use an heterogeneous system, a C compiler should be optionally installed on slave machines.

- when using an homogeneous computer cluster, each simulation (including scan steps) may be computed in parallel using MPI. We recommend this method on clusters (see section 4.6.2).

Last but not least, you may run simulations manually on a number of machines. Once the distributed simulation have been completed, you may merge their results using `mcformat` (see section 4.4.8) in order to obtain a set of files just as if it had been executed on a single machine.

All of these methods can be used, when available, from `mcgui`.

4.6.1. Distribute mcrun simulations on grids, multi-cores and clusters (SSH grid)

This method distributes simulations on a set of machines using *ssh* connections, using a command such as `mcrun --grid=4 ...`. Each of the scan steps is split and executed

on distant slave machines, sending the executable with *scp*, executing single simulations, and then retrieving individual results on the master machine. These are then merged using `mcformat`.

The `mcrun` script has been adapted to use transparently SSH grids. The syntax is:

- `--grid=<number>`: tells `mcrun` to use the grid 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`. When used on a single SMP machine (multi-core/cpu), this option may be omitted.
- `--force-compile`: this option is required on heterogeneous systems. The C code is sent to all slaves and simulation is compiled on each node before starting computation. The default is to send directly the executatble from the master node, which only works on homogeneous systems. computation.

This method shows similar efficiency as MPI, but without MPI installation. It is especially suited on multi-core machines, but may also be used on any set of distant machines (grids), as well as clusters. For Windows master machines, we recommend the installation of the PuTTY SSH client. The overhead is proportional to the number of nodes and the amount of data files to transfer per simulation. It is usually larger than the pure MPI method. We thus recommend to launch long runs on fewer nodes rather than many short runs on many nodes.

Requirements and limitation (SSH grids)

1. A master machine with an SSH client, and McStas installation.
2. A set of machines (homogeneous or heterogeneous) with SSH servers.
3. On heterogeneous grids, a C compiler must also be installed on all slave nodes.
4. `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. The key generation and registering mechanism may be done automatically for the local machine from the *Help menu/Install DSA key* item of `mcgui`.
5. 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. If it does not exist, only the current machine will be used (for multi-processor/core machines).

6. Without ssh keys, passwords will be prompted many times. To avoid this, we recommend to use only the local machine (for multi-cores/cpu), i.e. do not use a machine hosts file.

7. If your simulation/instrument requires *data files* (Powders, Sqw, source description, ...), these must be copied at the same level as the instrument definition. They are sent to all slave nodes before starting each computation. Take care to limit as much as possible the required data file volume in order to avoid large data transfers.

8. Interrupting or sending Signals may fail during computations. However, simulation scans can be interrupted as soon as the on-going computation step ends.

9. With heterogeneous systems, we recommend to use the `mcrun --force-compile` command rather than McGUI, which may skip the required simulation compilation on slaves.

4.6.2. Parallel computing (MPI)

The MPI support requires that MPICH (recommended), or alternatively LAM-MPI or OpenMPI, is installed on a set of nodes. This usually also requires properly setup `ssh` connections and keys as indicated in the ssh grid system (Section 4.6.1). Some OpenMPI implementations do not recognise multi-cores as separate nodes. In this case, you should use MPICH, or the SSH grid.

There are 3 methods for using MPI

- Basic usage requires to compile and run the simulation by hand (mpicc, mpirun). This should be used when running LAM-MPI.

- A much simpler way is to use `mcrun -c --mpi=NB_CPU ...` which will recompile and run MPICH.

- The McGUI interface supports MPICH from within the Run Dialog.

The MPI support is especially suited on clusters. As an alternative, the SSH grid presented above (section 4.6.1) is very flexible and requires lighter configuration.

Requirements and limitation (MPI)

To use MPI you will need

1. A master machine with an SSH client/server, and McStas installation.

2. A set of unix machines of the same architecture (binary compatible) with SSH servers.

3. `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. The key generation and registering mechanism may be done automatically for the local machine from the *Help menu/Install DSA key* item of `mcgui`.

4. 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. If it does not exist, only the current machine will be used (for multi-processor/core machines).

5. Without ssh keys, passwords will be prompted many times. To avoid this, we recommend to use only the local machine (for multi-cores/cpu), i.e. do not use a machine hosts file.

6. Signals are *not* supported while simulating with MPI (since asynchronous events cannot be easily transmitted to all nodes). This means it is not possible to cancel an on-going computation. However, simulation scans can be interrupted as soon as the on-going computation step ends.

7. 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. On non-local accounts, this procedure may fail and ssh always require passwords.

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>
```

4.6.3. McRun script with MPI support (mpich)

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`. When used on a single SMP machine (multi-core/cpu), this option may be omitted.

When available, the MPI option will show up in the `mcgui` Run dialog. Specify the number of nodes required.

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.

4.6.4. McStas/MPI Performance

Theoretically, 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

The efficiency of McStas using MPI has been tested on large clusters, up to 500 nodes. The computation time decreases in the same proportion as the number of nodes, showing an ideal efficiency. However, a small overhead may appear depending on the cluster internal network load, which may be estimated at most of about 10-20 s. This overhead comes from the spread and the fusion of the computations. For instance, spreading a computation implies often an `rsh` or and `ssh` session to be opened on every node. To reach the best efficiency, the computation time should not be lower than 30 seconds, or the overhead time may become significant compared to total time.

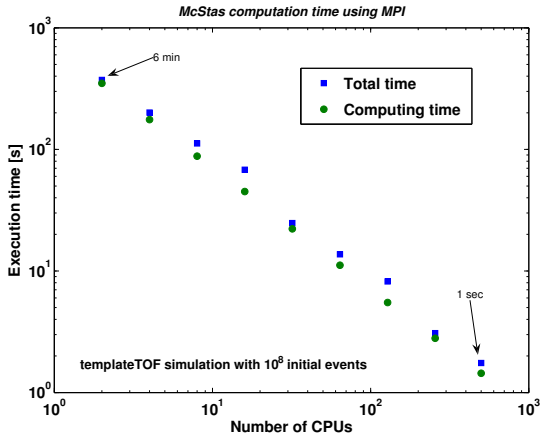


Figure 4.10.: McStas/MPI execution time as a function of computing nodes, with *templateTOF* instrument and 1e8 initial neutron events. Tests performed on Lonestar@TACC (US Teragrid, 2008).

4.6.5. MPI and Grid Bugs and limitations

- 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.
- Interacting with a running simulation (USR1 and USR2 signals) is disabled with MPI.

5. 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 B. The list includes functions for

- Computing the intersection between a flight-path and various objects (such as planes, cylinders, boxes and spheres)
- Functions for generating random numbers with various distributions
- Functions for reading or writing information 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.

5.1. 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 generally when setting an execution flag (`-g`) to perform gravitation computation. This latter setting is only an approximation and may produce wrong results with some components.

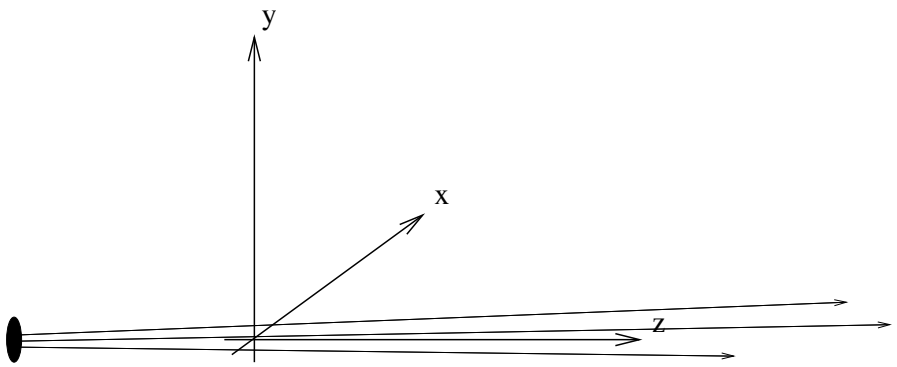


Figure 5.1.: conventions for the orientations of the axes in simulations.

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. Note that in case of a negative propagation time from one component to the next, the neutron ray is by default *absorbed* as this is often an indication of unphysical conditions.

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 5.1). Nothing in the McStas metalanguage enforces this convention, but if every component used different conventions the user would be faced with a severe headache! It is therefore necessary 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), and finally the neutron weight p described in 3.

5.2. Syntactical conventions

Comments follow the C and C++ syntax

```
1 /* C style comment */
2 // C++ style comment
```

Keywords are not case-sensitive, for example "`DEFINE`", "`define`", and "`dEfInE`" are all equivalent. However, by convention we always write keywords in uppercase to dis-

tinguish 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 5.1.

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 4.2.2). It is also possible to explicitly include another file using a line of the form

```
1    %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

```
1 %{
2 // Embedded C code...
3 %}
```

The "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. See also section 5.4 for insertion of full instruments in instruments (instrument concatenation).

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 macro), and there are no name conflicts between instrument and component instances variables.

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.
COPY	I,C	copy/duplicate an instance or a component definition.
DECLARE	I,C	Declares C internal variables.
DEFINE	I,C	Starts an INSTRUMENT or COMPONENT definition.
DEFINITION	C	Defines component parameters that are constants (#define).
END	I,C	Ends the instrument or component definition.
SPLIT	I	Enhance incoming statistics by event repetition.
EXTEND	I	Extends a component TRACE section (plug-in).
FINALLY	I,C	Embeds C code to execute when simulation ends.
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).
JUMP	I	Iterative (loops) and conditional jumps.
INITIALIZE	I,C	Embeds C code to be executed when starting.
ITERATE	I	Defines iteration counter for JUMP.
MCDISPLAY	C	Embeds C code to display component geometry.
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 (DEFINITION, SETTING).
PREVIOUS	C	Refers to a previous component position/orientation.
RELATIVE	I	Indicates that the AT and ROTATED keywords are relative to an other component.
REMOVABLE	I	Indicates that this component will be removed when the instrument is inserted into an other one using the %include keyword.
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.
SHARE	C	Declares global functions and variables to be shared.
TRACE	I,C	Defines the instrument as a the component sequence.
WHEN	I	Condition for component activation and JUMP.

Table 5.1.: Reserved McStas keywords. Scope is 'I' for instrument and 'C' for component definitions.

5.3. 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 5.3.9 and some additional examples of instrument definitions can be found on the McStas web-page [] and in the `example` directory.

As a summary, the usual grammar for instrument descriptions is

```
1 DEFINE INSTRUMENT name(parameters)
2 DECLARE C_code
3 INITIALIZE C_code
4 TRACE components
5 {FINALLY C_code}
6 END
```

5.3.1. The instrument definition head

```
1 DEFINE INSTRUMENT name (a_1, a_2, ...)
```

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 5.5. 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:

```
1 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 4.3), 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 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.

5.3.2. 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.

5.3.3. The `INITIALIZE` section

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

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

5.3.4. The `NEXUS` extension

The NeXus format [] requires to link the simulation to additional libraries (HDF and NeXus) which must have been pre-installed. Preferably, McStas should have been installed with the `./configure --with-nexus` on Unix/Linux systems. To activate the NeXus output, the compilation of the instrument must be done with flag `-DUSE_NEXUS -lNeXus`. The resulting executable is no longer portable.

The default NeXus format is NeXus 5 with compression.

You may choose the name of the output file with the `-f filename` option from the instrument executable or `mcrun` (see Sections 4.3, 4.4.2 and Table 4.2).

Then, the output format is chosen as usual with the `--format=NeXus` option when launching the simulation. All output files are stored in the output *filename*, as well as the instrument description itself. Other formats are still available. When run on a distributed system (e.g. MPI), detectors are gathered, but list of events (see e.g. component Virtual_output) are stored as one data set per node.

5.3.5. The `TRACE` section

As a summary, the usual grammar for component instances within the instrument TRACE section is

```
COMPONENT name = comp(parameters)
  AT (...) [RELATIVE [reference|PREVIOUS] | ABSOLUTE]
 {ROTATED {RELATIVE [reference|PREVIOUS] | ABSOLUTE} }
```

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, ...)
```

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 5.5.1 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.

Beware about variable type conversion when setting numerical parameter values, as in `p1=12/1000`. In this example, the parameter `p1` will be set to 0 as the division of the two integers is indeed 0. To avoid that, use explicitely floating type numbers as in `p1=12.0/1000`.

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 4.2.2 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:

```
1 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

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

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

```
1 ROTATED (phi_x,phi_y,phi_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. We *strongly* recommend to apply all rotations of an instrument description on Arm class components only, acting as goniometers, and position the optics on top of these. This usually makes it much easier to orient pieces of the instrument, and avoid positioning errors.

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.

```
1 AT (x,y,z) RELATIVE PREVIOUS
2 ROTATED (phi_x, phi_y, phi_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 beginning 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.

For experienced users, we recommend as well the usage of the `WHEN` and `EXTEND` keywords, as well as other syntax extensions presented in section 5.4 below.

5.3.6. 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. It is also executed when the simulation ends. This section is optional.

5.3.7. The `FINALLY` section

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

This gives code that will be executed when the simulation has ended. 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 receiving a TERM or INT signal (on Unix systems), or with Control-C, causing code in `FINALLY` to be evaluated.

5.3.8. The end of the instrument definition

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

```
END
```

5.3.9. 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.

5.4. Writing instrument definitions - complex arrangements and syntax

In this section, we describe some additional ways to build instruments using groups, code extension, conditions, loops and duplication of components.

As a summary, the nearly complete grammar definition for component instances within the instrument TRACE section is:

```
{SPLIT} COMPONENT name = comp(parameters) {WHEN condition}
  AT (...) [RELATIVE [reference|PREVIOUS] | ABSOLUTE]
  {ROTATED {RELATIVE [reference|PREVIOUS] | ABSOLUTE} }
  {GROUP group_name}
  {EXTEND C_code}
  {JUMP [reference|PREVIOUS|MYSELF|NEXT] [ITERATE number_of_times | WHEN
      condition] }
```

5.4.1. Embedding instruments in instruments TRACE

The `%include` insertion mechanism may be used within the TRACE section, in order to concatenate instruments together. This way, each DECLARE, INITIALIZE, SAVE, and FINALLY C blocks, as well as instrument parameters from each part are concatenated. The TRACE section is made of inserted COMPONENTS from each part. In principle, it is then possible to write an instrument as:

```
DEFINE concatenated()
TRACE

%include "part1.instr"
%include "part2.instr"

END
```

where each inserted instrument is a valid full instrument. In order to avoid some components to be duplicated - e.g. Sources from each part - a special syntax in the TRACE section

```
REMOVABLE COMPONENT a=...
```

marks the component *a* as removable when inserted. In principle, inserted instruments may themselves use `%include`.

5.4.2. Groups and component extensions - GROUP - EXTEND

It is sometimes desirable to slightly 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 embedded C code is appended to the component `TRACE` section, and all its internal variables (as well as all the `DECLARE` instrument variables, *except* instrument parameters)

may be used. To use instrument parameters, you should copy them into global variables in the DECLARE instrument section, and refer to these latter. This component declaration modifier is of course optional. You will find numerous usage examples, and in particular in the Sources section of the Component manual.

In some peculiar configurations it is necessary 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 (after the AT-ROTATED positioning):

```
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).

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 (
                    p_1 = e_1,
                    p_2 = e_2,
                    ...)
AT (0,0,0) ABSOLUTE

COMPONENT name1 = comp...
AT (...) ROTATED (...)

GROUP GroupName EXTEND
%{
   if (SCATTERED) printf("I scatter"); else printf("I do not scatter");
%}

COMPONENT} name2 = comp ...
AT (...) ROTATED (...)
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 recommend to make use of the SCATTER keyword each time a component 'uses' the neutron (scatters, detects, ...) within component definitions (see section 5.5). Also, the components to be grouped should be consecutive in the TRACE section of the instrument, and the GROUPed section should not contain components which are not part of the group.

A usage example of the GROUP keyword can be found in the `Neutron site/ILL/ILL_H15_IN6` instrument from the `mcgui`, to model 3 monochromators.

Combining EXTEND, GROUP and WHEN can result in unexpected behaviour. Please read the related warning at the end of section 5.4.4.

5.4.3. Duplication of component instances - COPY

Often, one has a set of similar component instances in an instrument. These could be e.g. a set of identical monochromator blades, or a set of detectors or guide elements. Together with JUMPs (see below), there is a way to copy a component instance, duplicating parameter set, as well as any EXTEND, GROUP, JUMP and WHEN keyword. Position (AT) and rotation (ROTATED) specification must be explicitely entered in order to avoid component overlapping.

The syntax for instance copy is

```
COMPONENT name = COPY (instance_name)
```

where *instance_name* is the name of a preceeding component instance in the instrument. It may be 'PREVIOUS' as well.

If you would like to change only some of the parameters in the instance copy, you may write, e.g.:

```
COMPONENT name = COPY (instance_name)(par1=0, par2=1)
```

which will override the original instance parameter values. This possibility to override parameters is very useful in case of describing e.g. sample environments using the Isotropic_Sqw and PowderN components, which allow *concentric* geometry (first instance must have `concentric = 1` and the second `concentric = 0`). In case EXTEND, GROUP, JUMP and WHEN keywords are defined for the copied instance, these will override the settings from the copied instance.

In the case where there are many duplicated components all originating from the same instance, there is a mechanism for automating copied instance names:

```
COMPONENT COPY(root_name) = COPY(instance_name)
```

will concatenate a unique number to *root_name*, avoiding name conflicts. As a side effect, referring to this component instance (for e.g. further positioning) is not straight forward as the name is determined by McStas and does not depend completely on the user's choice, even though the PREVIOUS keyword may still be used. We thus recommend to use this naming mechanism only for components which should not be referred to in the instrument.

This automatic naming may be used anywhere in the TRACE section of the instrument, so that all components which do not need further referring may be labelled as COPY(Origin).

As an example, we show how to build a guide made of equivalent elements. Only the first instance of the Guide component is defined, whereas following instances are copies of that definition. The instance name of Guide components is set automatically.

```
COMPONENT CG_In = Arm() AT (...)

COMPONENT CG_1  = Guide_gravity(l=L/n, m=1, ...)
  AT (0,0,0) RELATIVE PREVIOUS

COMPONENT COPY(CG_1)  = COPY(CG_1)
  AT (0,0,L/n+d) RELATIVE PREVIOUS
  ROTATED (0, (L/n+d)/R*180/PI, 0) RELATIVE PREVIOUS

COMPONENT COPY(CG_1)  = COPY(CG_1)
  AT (0,0,L/n+d) RELATIVE PREVIOUS
  ROTATED (0, (L/n+d)/R*180/PI, 0) RELATIVE PREVIOUS
...
COMPONENT CG_Out = Arm() AT (0,0,L/n) RELATIVE PREVIOUS
```

5.4.4. Conditional components - WHEN

One of the most useful features of the extended McStas syntax is the conditional `WHEN` modifier. This optional keyword comes before the AT-ROTATED positioning. It basically enables the component only when a given condition is true (non null).

```
  COMPONENT name = comp (p_1 = e_1, p_2 = e_2, ...)
  WHEN condition
```

The condition has the same scope as the EXTEND modifier, i.e. may use component internal variables as well as all the `DECLARE` instrument variables, *except* instrument parameters. To use instrument parameters, you should copy them into global variables in the DECLARE instrument section, and refer to these latter. It is evaluated before the component TRACE section.

Usage examples could be to have specific monitors only sensitive to selected processes, or to have components which are only present under given circumstances (e.g. removable guide or radial collimator), or to select a sample among a set of choices.

In the following example, an EXTEND block sets a condition when a scattering event is encountered, and the following monitor is then activated.

```
COMPONENT Sample = V_sample(...) AT ...
  EXTEND
  %{
    if (SCATTERED) flag=1; else flag=0;
  %}

COMPONENT MyMon = Monitor(...) WHEN (flag==1)
  AT ...
```

The WHEN keyword only applies to the TRACE section and related EXTEND blocks of instruments/components. Other sections (INITIALIZE, SAVE, MCDISPLAY, FINALLY) are executed independently of the condition. As a side effect, the 3D view of the instrument (mcdisplay) will show all components as if all conditions were true.

Also, the `WHEN` keyword is a condition for `GROUP`. This means that when the `WHEN` is false, the component instance is not active in the `GROUP` it belongs to.

A usage example of the WHEN keyword can be found in the
`Neutron site/ILL/ILL_TOF_Env` instrument from the `mcgui`, to monitor neutrons depending on their fate.

WARNING: Combining WHEN, EXTEND and GROUP can result in unexpected behaviour, please use with caution! Let for instance a GROUP of components all have the same WHEN condition, i.e. if the WHEN condition is false, none of the elements SCATTER, meaning that all neutrons will be ABSORBed. As a solution to this problem, we propose to include an EXTENDed Arm component in the GROUP, but with the opposite WHEN condition and a SCATTER keyword in the EXTEND section. This means that when none of the other GROUP elements are present, the Arm will be present and SCATTER.

5.4.5. Component loops and non sequential propagation - JUMP

There are situations for which one would like to repeat a given component many times, or under a given condition. The JUMP modifier is meant for that and should be placed after the positioning, GROUP and EXTEND. This breaks the sequential propagation along components in the instrument description. There may be more than one JUMP per component instance.

The jump may depend on a condition:

```
COMPONENT name = comp(p_1 = e_1, p_2 = e_2, ...)
  AT (...)
  JUMP reference WHEN condition
```

in which case the instrument TRACE will jump to the *reference* when *condition* is true.

The *reference* may be an instance name, as well as PREVIOUS, PREVIOUS(n), MYSELF, NEXT, and NEXT(n), where n is the index gap to the target either backward (PREVIOUS) or forward (NEXT), so that PREVIOUS(1) is PREVIOUS and NEXT(1) is NEXT. MYSELF means that the component will be iterated as long as the condition is true. This may be a way to handle multiple scattering, if the component has been designed for that.

The jump arrives directly inside the target component, in the local coordinate system (i.e. without applying the AT and ROTATED keywords). In order to control better the target positions, it is *required* that, except for looping MYSELF, the target component type should be an *Arm*.

There is a more general way to iterate components, which consists in repeating the loop for a given number of times.

```
JUMP reference ITERATE number_of_times
```

This method is specially suited for very long curved guides of similar components, but in order to take into account rotation and translation between guide sections, the iterations are performed between Arm's.

In the following example for a curved guide made on $n = 500$ elements of length L on a curvature radius R, with gaps d between elements, we simply write:

```
COMPONENT CG_In = Arm() AT (...)

COMPONENT CG_1  = Guide_gravity(l=L/n, m=1, ...)
  AT (0,0,0) RELATIVE PREVIOUS

COMPONENT CG_2_Position = Arm()
  AT (0,0,L/n+d) RELATIVE PREVIOUS
  ROTATED (0, (L/n+d)/R*180/PI, 0) RELATIVE PREVIOUS

COMPONENT CG_2  = Guide_gravity(l=L/n, m=1, ...)
  AT (0,0,0) RELATIVE PREVIOUS
  ROTATED (0, (L/n+d)/R*180/PI, 0) RELATIVE PREVIOUS
  JUMP CG_2_Position ITERATE n
...
COMPONENT CG_Out = Arm() AT (0,0,L/n) RELATIVE PREVIOUS
```

Similarly to the `WHEN` modifier (see section 5.4.4), `JUMP` only applies within the TRACE section of the instrument definition. Other sections (INITIALIZE, SAVE, MCDISPLAY, FINALLY) are executed independently of the jump. As a side effect, the 3D view of the instrument (mcdisplay) will show components as if there was no jump. This means that in the following example, the very long guide 3D view only shows a single guide element.

It is *not* recommended to use the `JUMP` inside `GROUP`s, as the JUMP condition/counter applies to the component instance within its group.

We would like to emphasize the potential errors originating from such jumps. Indeed, imbricating many jumps may lead to situations were it is difficult to understand the flow of the simulation. We thus recommend the usage of JUMPs only for experienced and cautious users.

5.4.6. Enhancing statistics reaching components - SPLIT

The following method applies when the incoming neutron event distribution is considered to be representative of the real beam, but neutrons are lost in the course of propagation (with low efficiency processes, absorption, etc). Then, one may think that it's a pity to have so few events reaching the 'interesting' part of the instrument (usually close to the end of the instrument description). If some components make extensive use of random numbers (MC choices), they shuffle this way the distributions, so that identical incoming events will not produce the same outgoing event. In this case, you may use the `SPLIT` keyword with the syntax

```
SPLIT r COMPONENT name = comp(...) $
```

where the optional number r specifies the number of repetitions for each event. Default is $r = 10$. Each neutron event reaching component *name* will be repeated r times with a weight divided by r, so that in practice the number of events for the remaining part of the simulation (down to the END), will potentially have more statistics. This is only true if following components (and preferably component *name*) use random numbers. You may use this method as many times as you wish in the same instrument, e.g. at the monochromator and sample position. This keyword can also be used within a GROUP. The efficiency is roughly r raised to the number of occurrences in the instrument, so that enhancing two components with the default $r = 10$ will produce at the end an enhancement effect of 100 in the number of events. The execution time will usually get slightly longer. This technique is known as the *stratified sampling* (see Appendix 3). If the instrument makes use of global variables - e.g. in conjunction with a WHEN or User Variable monitoring (see Monitor_nD) - you should take care that these variables are set properly for each SPLIT loop, which usually means that they must be reset inside the SPLITed section and assigned/used further on.

A usage example of the SPLIT keyword can be found in the
`Neutron site/ILL/ILL_H15_IN6` instrument from the `mcgui`, to enhance statistics for neutrons scattering on monochromators and sample.

5.5. 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 5.5.10, and all component definitions can be found on the McStas web-page [] and are described in the McStas component manual.

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

5.5.1. The component definition header

```
1 DEFINE COMPONENT name
```

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

```
1 DEFINITION PARAMETERS (d_1, d_2, ...)
2 SETTING PARAMETERS (s_1, s_2, ...)
```

This declares the definition and setting parameters of the component. These parameters can be accessed from all sections of the component (see below), as well as in `EXTEND` sections of the instrument definition (see section 5.3).

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 5.3).

```
1 OUTPUT PARAMETERS (s_1, s_2, ...)
```

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 *strongly recommended* 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 5.5.2 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.

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 behaviour, 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.

5.5.2. 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.

5.5.3. 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 occurrence 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 duplicate 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.

5.5.4. 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.

5.5.5. The TRACE section

```
1 TRACE
2 %{
3 // C code to compute neutron interaction with component
4 %}
```

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 B.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 precede 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 GROUPs in an instrument definition (see section 5.3.5). It basically increments the SCATTERED counter. The SCATTER macro should be called with the state parameters set to the proper values for the scattering event, so that neutron events are displayed correctly. For an example of SCATTER, see the optics/Guide component.

5.5.6. The SAVE section

```
1 SAVE
2 %{
3 // C code to execute in order to save data
4 %}
```

This gives code that will be executed when the simulation ends, or is requested to save data, for instance when receiving a USR2 signal (on Unix systems, see section 4.3), 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 table ?? and section 4.3) 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 3.2.1.

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=sigma CompNameN=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,
          xvar, x_min, x_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,
          xvar, x_min, x_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.

Customizing detectors/monitors Users may want to have additional information than the default one written by the DETECTOR_OUT macros. A mechanism has been implemented for monitor components to output customized meta data. The macro:

```
DETECTOR_CUSTOM_HEADER(t)
```

defines a string to be written during the next `DETECTOR\_OUT*` call, as a field *custom*. This string may additionally use the symbol `%PRE` which is replaced by the format specific comment character, e.g. '#' for McStas/PGPLOT, '%' for Octave/Matlab, '//' for Scilab and ';' for IDL, and ' ' for other formats. The argument t to the macro may be a static string, e.g. *"My own additional information"*, or the name of a *character array variable* containing the meta data. After the detector/monitor file being written, the custom meta data output is deactivated. This way, each monitor file may have its own meta data definition by repeating the DETECTOR_CUSTOM_HEADER call. You may either do that inside the component SAVE section, or within an instrument description in an EXTEND code preceeding the monitor (e.g. following an Arm component).

5.5.7. 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. This section also triggers the SAVE section to be executed.

5.5.8. 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 4.4.3). 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 `dashed_line` command The `dashed_line` command takes the following form:

```
dashed_line(x_1, y_1, z_1, x_2, y_2, z_2, n)
```

It draws a dashed line between the points (x_1, y_1, z_1) and (x_2, y_2, z_2) with n equidistant spaces.

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

```
  \texttt{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 `rectangle` command The `rectangle` command takes the following form:

```
rectangle(plane, x, y, z, width, height)
```

Here *plane* should be either `"xy"`, `"xz"`, or `"yz"`. The command draws a rectangle in the specified plane with the center at (x, y, z) and the size *width* $\times$ *height*. Depending on *plane* the width and height are defined as:

plane	*width*	*height*
xy	x	y
xz	x	z
yz	y	z

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

```
box(x, y, z, xwidth, yheight, zlength)
```

The command draws a box with the center at (x, y, z) and the size *xwidth* $\times$ *yheight* $\times$ *zlength*.

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.

5.5.9. The end of the component definition

```
END
```

This marks the end of the component definition.

5.5.10. A component example: Slit

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

5.6. Extending component definitions

Suppose you are interested by one component of the McStas library, but you would like to customize it a little. There are different ways to extend an existing component.

5.6.1. Extending from the instrument definition

If you only want to *add* something on top of the component existing behaviour, the simplest is to work from the instrument definition `TRACE` section, using the `EXTEND` modifier (see section 5.4.2). You do not need to write a new component definition, but only add a piece of code to execute.

5.6.2. Component heritage and duplication

There is a heritage mechanism to create children of existing components. These are exact duplicates of the parent component, but one may override/extend original definitions of any section.

The syntax for a full component child is

```
DEFINE COMPONENT child_name COPY parent_name
```

This single line will copy all parts of the *parent* into the *child*, except for the documentation header.

As for normal component definitions, you may add other parameters, `DECLARE`, `TRACE`, ... sections. Each of them will replace or extend (be concatenated to, with the COPY-/EXTEND keywords, see example below) the corresponding *parent* definition. In practice, you could copy a component and only rewrite some of it, as in the following example:

```
DEFINE COMPONENT child_name COPY parent_name

  SETTING PARAMETERS (newpar1, newpar2)
  INITIALIZE COPY parent_name EXTEND
  %{
// C code to be concatenated to the parent_name INITIALIZE
  %}
  SAVE
  %{
// C code to replace the parent_name SAVE
  %}
```

where two additional parameters have been defined, and should be handled in the extension of the original INITIALIZE section.

On the other hand, if you do not derive a component as a whole from a parent, you may still use specific parts from any component:

```
DEFINE COMPONENT name ...
  DECLARE COPY parent1
  INITIALIZE COPY parent2 EXTEND
%{
// C code to be concatenated to the parent2 INITIALIZE
%}
TRACE COPY parent3
```

This mechanism may lighten the component code, but a special care should be taken in mixing bits from different sources, specially concerning variables. This may result in difficulties to compile components.

5.7. McDoc, the McStas library documentation tool

McStas includes a facility called McDoc to help maintain good documentation of components and instruments. In the 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 4.4.1. The other is a collection of web pages documenting the components and instruments, handled via the `mcdoc` front-end (see section 4.4.6), and the complete documentation for all available McStas components and instruments may be found at the McStas webpage [], as well as in the McStas library (see 6.1). 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`, `%EXAMPLE:`, `%LINKS`, and `%END` keywords to mark different sections of the documentation. Keywords may be abbreviated (except for `%EXAMPLE:`), *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 parentheses are also supported for backwards compatibility, but nested parentheses 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.

- The %EXAMPLE: lines in instrument headers indicate an example parameter set or command that may be run to test the instrument. A following Detector: <name>_I=<value> indicates what value should be obtained for a given monitor. More than one example line may be specified in instruments.

- 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.

6. 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 [] where all components are documented using the McDoc system.

6.1. 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. The location of this library is detailed in section 4.2.2. All of them are believed to be reliable, and some amount of systematic tests have been carried out. However, no absolute guaranty can be given concerning their accuracy.

The `contrib` directory of the library contains components that were submitted 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 4.4.6) enables to display both the catalog of the McStas library, e.g using:

```
1 mcdoc
```

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

```
1 mcdoc --text name
2 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	Continuous 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	Continuous neutron source with adaptive importance sampling.
Source_div	Continuous neutron source with a specified Gaussian divergence.
Source_simple	A simple, continuous circular neutron source with flat energy/wavelength spectrum.
Source_gen	General, continuous neutron source with tunable shape, spectrum, and divergence.
Virtual_input	Source-like component that reads neutron events from an asci-i/binary 'virtual source' file (recorded by Virtual_output).
Virtual_output	Detector-like component that writes neutron state (for use in Virtual_input).

Table 6.1.: 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`).
Collimator_linear	A simple analytical Soller collimator.
Collimator_radial	A radial Soller collimator.
DiskChopper	Disk chopper.
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.
Pol_bender	Polarising bender.
Pol_guide_vmirror	Guide with semi-transparent, polarising mirror.
Pol_mirror	Polarising mirror
Pol_simpleBfield	Numerical precession in analytical B-fields.
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 (longer execution time than FermiChopper). From the Vitess package.

Table 6.2.: Optics components of the McStas library.

MCSTAS/samples	Description
Inelastic_Incoherent	Inelastic incoherent sample with quasielastic and elastic contributions.
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).
Powder1	General powder sample with a single scattering vector.
PowderN	General powder sample with N scattering vectors, using a data file. Can assume *concentric* shape, i.e. can be used to model sample enviroment.
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. Optional quasielastic broadening.
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

Table 6.3.: Sample components of the McStas library.

MCSTAS/monitors	Description
DivPos_monitor	Divergence/position monitor (acceptance diagram).
Divergence_monitor	Horizontal+vertical divergence monitor (2D).
E_monitor	Energy-sensitive monitor.
L_monitor	Wavelength-sensitive monitor.
Monitor	Simple single detector/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.
PSD_monitor_4PI	Spherical position-sensitive detector.
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 vs. wavelength monitor.
TOF_monitor	Rectangular Time-of-flight monitor.

Table 6.4.: Selected Monitor components of the McStas library.

MCSTAS/misc	Description
Progress_bar	Displays status of a running simulation. May also trigger intermediate SAVE.
Beam_spy	A monitor that displays mean statistics (no output file).
Set_pol	sets polarisation vector.
Vitess_input	Read neutron state parameters from a VITESS neutron file.
Vitess_output	Write neutron state parameters to a VITESS neutron file.

Table 6.5.: Miscellaneous components of the McStas library.

MCSTAS/contrib	Description
Al_window	Aluminium transmission window.
Collimator_ROC	Radial Oscillating Collimator (ROC).
Exact_radial_coll	Radial collimator.
FermiChopper_ILL	Fermi Chopper with rotating frame and SM coating (straight).
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.
multi_pipe	a multi pipe slit (for SANS).
PSD_Detector	Realistic detector model, with gas effects.
PSD_monitor_rad	A banana PSD monitor.
SiC	SiC multilayer sample for reflectivity simulations.
SNS_source	SNS moderator models based on MCNP calculations.
Source_multi_surfaces	An array of sources described from spectrum tables.
Virtual_mcnp_input	Reads a PTRAC neutron events file from MCNP.
Virtual_mcnp_output	Writes a PTRAC neutron events file for MCNP.
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 6.6.: 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 6.7.: Shared libraries of the McStas library. See Appendix B 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 6.8.: Data files of the McStas library.

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

Table 6.9.: Instrument example files of the McStas library.

7. Instrument examples

In this section, we present a few typical instruments. We then give a longer description of three selected instruments. We present the McStas versions of the Risø standard triple axis spectrometer TAS1 (7.3) and the ISIS time-of-flight spectrometer PRISMA (7.4).

But first we present an example of a component test instrument: the instrument to test the component **V_sample** (7.2). These instrument files are included in the McStas distribution in the `examples/` directory. Most of the instrument examples therein may be executed automatically throught the McStas self-test procedure. The list of instrument examples has been extended considerably, see the "Neutron Site" menu in mcgui (See page 4.1).

7.1. A quick tour of instrument examples

7.1.1. Neutron site: Brookhaven

The former Brookhaven reactor hosted the H8 triple-axis spectrometer. This latter was modelled in order to cross-check the NISP, Vitess, Restrax, McStas and IDEAS neutron propagation Monte Carlo codes. Results were published in *Neutron News* **13** (No. 4), 24-29 (2002).

7.1.2. Neutron site: Tools

This category currently contains a special `Histogrammer` that can read any event file from Vitess, MCNP, tripoli and McStas in order to produce histograms of any type. This is a very powerful tool to analyse sparse huge events data sets.

7.1.3. Neutron site: ILL

Cold and thermal guide models are given in the ILL neutron site. Descriptions are very accurate, based on actual drawings, including all elements with curvature and gaps. Simulated capture fluxes were found in very good agreement with measurements.

Additionally, the `IN12` triple-axis instrument has been detailed, in its year 2005 configuration. It is located at the end of the H15 cold guide. The sample is a Vanadium rod.

The `IN6` instrument is a hybrid time-of-flight spectrometer with 3 monochromators and a Fermi chopper. The sample is an isotropic scatterer (liquid). It makes use of the SPLIT keyword to enhance statistics.

The `ILL_TOF_Env` example is a simple neutron pulse (e.g. from a chopper system coming out of IN4, IN5 or IN6) illuminating a sample, with a container around and a cylinder shaped sample environment.

7.1.4. Neutron site: tests

This large set of examples shows simple instruments using particular components (samples, polarized beams, detectors). They may be used as starting point for further more complex models.

7.1.5. Neutron site: ISIS

You will find here examples of ISIS instruments, some of them using the ISIS moderator component (obtained from MCNP computations).

7.1.6. Neutron site: Risø

This section contains former Risø instruments, which constitue the default test series for the McStas distribution.

7.1.7. Neutron site: PSI

The Focus time-of-flight instrument is, with the ILL IN6 and IN12 models, the most advanced example. It as been built from drawings, including the guide and all optics elements. Simulated capture fluxes were found in good agreement with measurements.

7.1.8. Neutron site: Tutorial

A typical diffractometer including a powder sample and a Rescal-type triple-axis models are in this category. Instrument parameters enable models to cope with most existing instruments of these classes. The TAS model includes a basic in-plane UB-matrix transformation. It may be used to estimate the resolution functions of a TAS configuration.

7.1.9. Neutron site: ESS

This section contains design studies for the future, long-pulsed European Spallation Source (ESS). The only contents is currently the `ESS_IN5_reprate` instrument for simulating an IN5-TYPE (cold chopper) multi-frame spectrometer for the long-pulsed ESS. (Also serves as example instrument for `Tunneling_sample.comp`.)

7.2. 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.

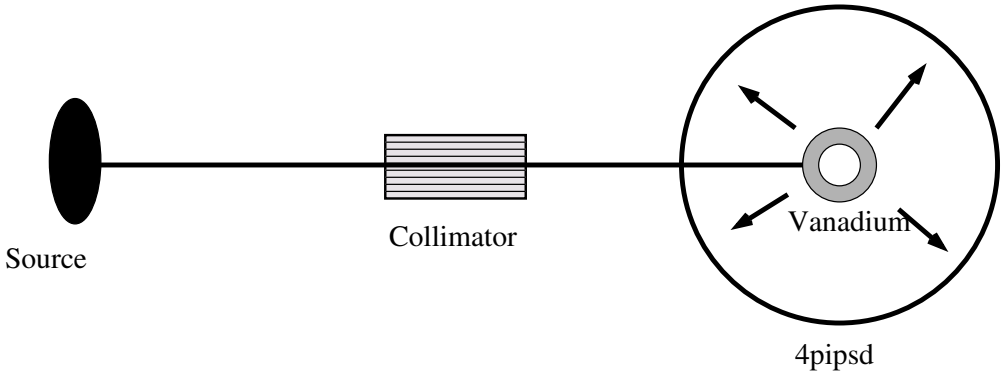


Figure 7.1.: A sketch of the test instrument for the component V_sample.

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 7.1.

7.2.1. Scattering from the V-sample test instrument

In figure 7.2, we present the radial distribution of the scattering 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 anisotropic attenuation of the beam in the cylindrical sample.

7.3. 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 neutron source DR3 of Risø National Laboratory. Except for the cold source itself, all components used have quite realistic properties. Furthermore, the overall geometry of the instrument has been adapted from the detailed technical drawings of the real spectrometer. The TAS 1 simulation was the first detailed work performed with the McStas package. For further details see reference [ACL98].

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

The real monochromator consist of seven blades, vertically focusing on the sample. The angle of curvature is constant so that the focusing is perfect at 5.0 meV (20.0 meV for 2nd order reflections) for a 1×1 cm^2 sample. This is modeled directly in the instrument

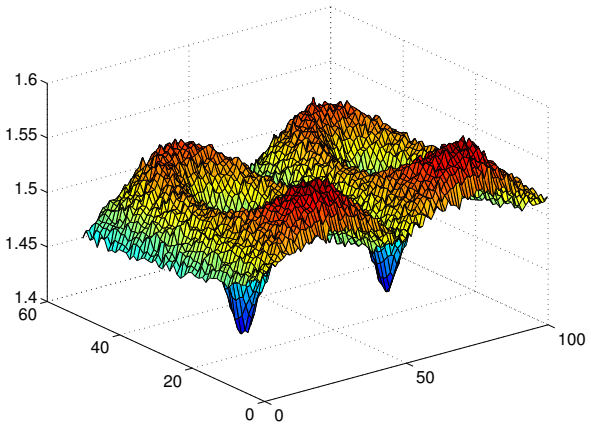


Figure 7.2.: Scattering from a V-sample, measured by a spherical PSD. The sphere has been transformed onto a plane and the intensity is plotted as the third dimension.

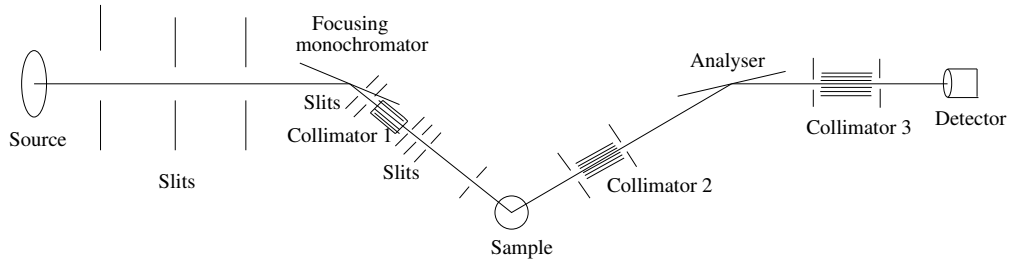


Figure 7.3.: A sketch of the TAS1 instrument.

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 width of 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 7.3. Test results and data from the real spectrometer are shown in Appendix 7.3.1.

7.3.1. 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. The mea-

surement 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 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 7.4 shows a scan of $2\theta_m$ 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 7.5.

The result of a $2\theta_s$ scan on an Al_2O_3 powder sample in two-axis mode is shown in Figure 7.6. 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 7.7.

7.4. 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 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 7.8. Test results are shown in Appendix 7.4.1.

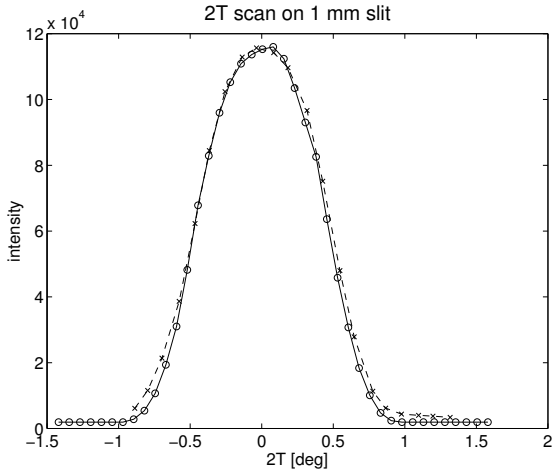


Figure 7.4.: TAS1: Scans of $2\theta_s$ in the direct beam with 1 mm slit on the sample position. "×": measurements, "o": simulations, scaled to the same intensity Collimations: open-30'-open-open.

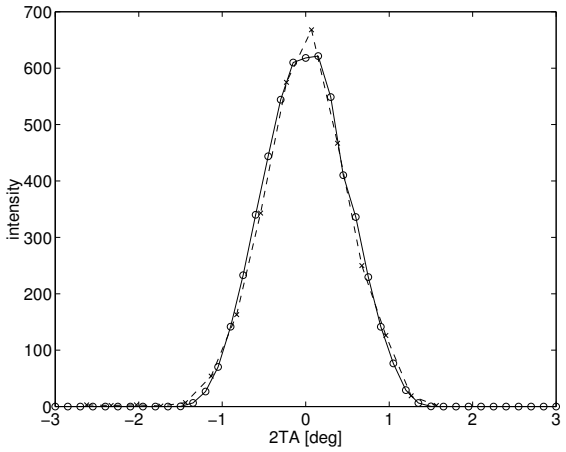


Figure 7.5.: TAS1: Corrected $2\theta_a$ scan on a V-sample. Collimations: open-30'-28'-67'. "×": measurements, "o": simulations.

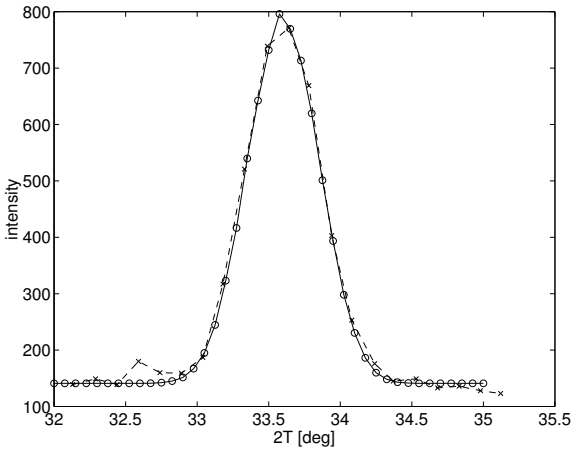


Figure 7.6.: TAS1: $2\theta_s$ scans on Al_2O_3 in two-axis, focusing mode. Collimations: open-30'-28'-67'. "×": measurements, "o": simulations. A constant background is added to the simulated data.

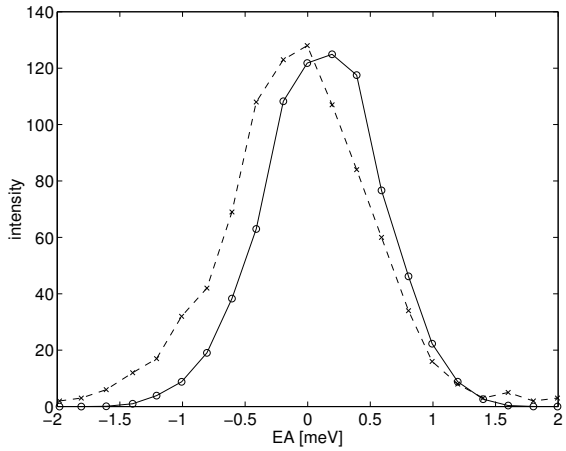


Figure 7.7.: TAS1: Scans of the analyser energy on a V-sample. Collimations: open-30'-28'-67'. "×": measurements, "o": simulations.

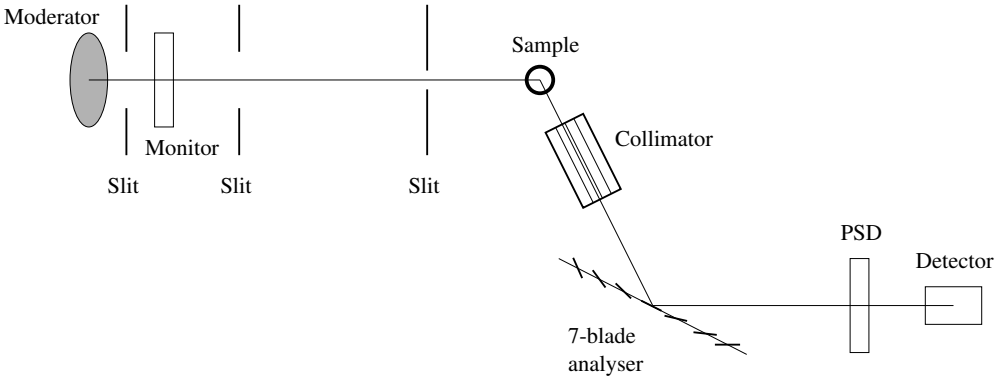


Figure 7.8.: A sketch of the PRISMA instrument.

7.4.1. Simple spectra from the PRISMA instrument

A plot from the detector in the PRISMA simulation is shown in Figure 7.9. 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.

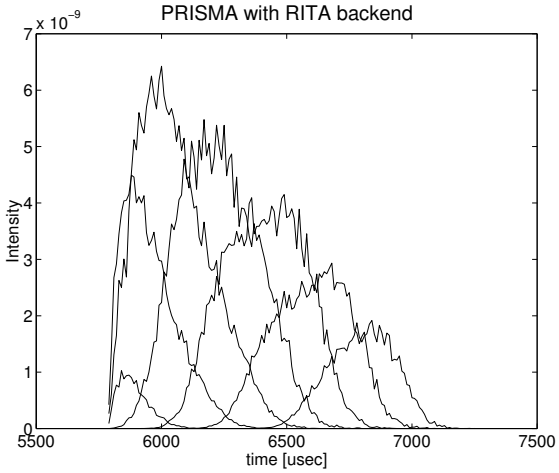


Figure 7.9.: Test result from PRISMA instrument using “colored neutrons”. Each graph shows the neutrons scattered from one analyser blade.

A. Random numbers in McStas

A.1. Transformation of random numbers

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

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

If the random number generator selects numbers uniformly in the interval $[0; 1]$, we have $\phi(x) = 1$ (inside the interval; zero outside), and we reach

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

By indefinite integration we reach

$$\int \Psi(y)dy = f^{-1}(y) = x, \tag{A.2}$$

which is the essential formula for random number transformation, since we in general know $\Psi(y)$ and like to determine the relation $y = f(x)$. Let us illustrate with a few examples of transformations relevant for the McStas components.

The circle For finding a random point within the circle of radius R, one would like to choose the polar angle, ϕ, from a uniform distribution in $[0; 2\pi]$, giving $\Psi_\phi = 1/(2\pi)$. and the radius from the (normalised) distribution $\Psi_r = 2r/R^2$.

For the radial part, eg. (A.2) becomes $y/(2\pi) = x$, whence ϕ is found simply by multiplying a random number (x) with 2π.

For the radial part, the left side of eq. (A.2), gives $\int \Psi(r)dr = \int 2r/R^2 dr = r^2/R^2$, which from (A.2) should equal x. Hence we reach the wanted transformation $r = R\sqrt{x}$.

The sphere For finding a random point on the surface of the unit sphere, we need to determine the two angles, (θ, ϕ).

Ψ_ϕ is chosen from a uniform distribution in $[0; 2\pi]$, giving $\phi = 2\pi x$ as for the circle.

The probability distribution of θ should be $\Psi_\theta = \sin(\theta)$ (for $\theta \in [0; \pi]$), whence by eq. (A.2) $\theta = \cos^{-1}(x)$.

Exponential decay In a simple time-of-flight source, the neutron flux decays exponentially after the initial activation at $t = 0$. We thus want to pick an initial neutron emission time from the normalised distribution $\Psi(t) = \exp(-t/\tau)/\tau$. Use of Eq. (A.2) gives $x = 1 - \exp(-t/\tau)$. For convenience we now use the random variable $x_1 = 1 - x$ (with the same distributions as x), giving the simple expression $t = -\tau \ln(x_1)$.

Normal distributions The important normal distribution can not be reached as a simple transformation of a uniform distribution. In stead, we rely on a specific algorithm for selecting random numbers with this distribution.

A.2. Random generators

Even though there is the possibility to use the system random generator, as well as the initial McStas version 1.1 random generator, the default algorithm is the so-called "Mersenne Twister", by Makoto Matsumoto and Takuji Nishimura. See `http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html` for original source.

It is considered today to be by far the best random generator, which means that both its period is extremely large $2^{19937} - 1$, and cross-correlations are negligible, i.e distributions are homogeneous and independent up to 623 dimensions. It is also extremely fast.

B. Libraries and 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):

```
1  %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,

```
1  %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.

B.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.

B.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, except if a `ALLOW_BACKPROP;` preceeds it.

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. This macro automatically calls PROP_GRAV_DT when the `--gravitation` option has been set for the whole simulation.

- **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).

B.1.2. Coordinate and component variable retrieval

- **MC_GETPAR**($comp, outpar$). This may be used in e.g. 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.

B.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 $t.a$

B.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,φ,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.

B.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-dimensional detector. Integrated intensities error etc. is also reported as for DETECTOR_OUT_0D.

- **DETECTOR_OUT_2D**(...). Used to output the results from a two-dimentional detector. Integrated intensities error etc. is also reported as for DETECTOR_OUT_0D.

- **DETECTOR_OUT_3D**(...). Used to output the results from a three-dimentional detector. Arguments are the same as in DETECTOR_OUT_2D, but with an additional z axis. Resulting data files are treated as 2D data, but the 3rd dimension is specified in the *type* field. Integrated intensities error etc. is also reported as for DETECTOR_OUT_0D.

- **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.

B.1.6. Ray-geometry intersections

- **inside_rectangle**(&x, &y, x, xw, yh). Return 1 if $-xw/2 \leq x \leq xw/2$ AND $-yh/2 \leq y \leq yh/2$. Else return 0.

- **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.

B.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 sample the normal distribution is explained in Ref. [Pre+86, ch.7].

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

- **randtriangle**(). Returns a random number from a triangular distribution 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.

B.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 get element [i,j] */
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 and return an empty table. 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 concatenate 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.

- **Table_Free**(&$Table$) free allocated memory blocks.

- **Table_Value2d**($Table$, X, Y) Uses 2D linear interpolation on a Table, from (X,Y) coordinates and returns the corresponding value.

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 is 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 6.8).

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 concatenated. 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)

```

```
8   y = Table_Index(Table[0], 2,5);  // read in 1st block the 3rd row, 6th
        column element
9                                    // ONLY use Table[i] with i < n !
10  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.

B.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.

B.4. Adaptive importance sampling Library

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

B.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.

B.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 ($\mathrm{\AA}^{-1}$)
K2V	$10^{-10} \cdot \hbar/m_{\mathrm{N}}$	**k**-vector ($\mathrm{\AA}^{-1}$)	Velocity (m/s)
VS2E	$m_{\mathrm{N}}/(2e)$	Velocity squared ($\mathrm{m}^2\ \mathrm{s}^{-2}$)	Neutron energy (meV)
SE2V	$\sqrt{2e/m_{\mathrm{N}}}$	Square root of neutron energy ($\mathrm{meV}^{1/2}$)	Velocity (m/s)
FWHM2RMS	$1/\sqrt{8\log(2)}$	Full width half maximum	Root mean square (standard deviation)
RMS2FWHM	$\sqrt{8\log(2)}$	Root mean square (standard deviation)	Full width half maximum
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	

C. 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. These are considered as Types of elements to be instantiated in an Instrument description.
- **Component Instance** A named Component (of a given Type) inserted in an Instrument description.
- **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.

Index

Bibliography

[] See http://www.mcstas.org (cit. on pp. 9, 11, 13, 16, 17, 28, 31, 35, 69, 80, 91, 93).

[] See http://trac.mccode.org/report (cit. on pp. 9, 18).

[] See http://ts-2.isis.rl.ac.uk/ (cit. on p. 10).

[] See http://www.ess-europe.de (cit. on pp. 10, 12).

[] See http://neutron-eu.net/en/ (cit. on p. 10).

[] See http://mcnsi.risoe.dk/ (cit. on p. 10).

[] See http://strider.lansce.lanl.gov/NISP/Welcome.html (cit. on p. 11).

[] See http://www.hmi.de/projects/ess/vitess/ (cit. on p. 11).

[] See http://neutrons-dev.ornl.gov/eqsans/software/ib/ (cit. on p. 11).

[] http://code.google.com/p/jnads/ (cit. on p. 11).

[] See http://www.sns.gov/ (cit. on p. 12).

[] See http://www.neutron.anl.gov/nexus/ (cit. on pp. 58, 70).

[ACL98] A. Abrahamsen, N. B. Christensen, and E. Lauridsen. *McStas simulations of the TAS1 spectrometer*. Student's report. Niels Bohr Institute, University of Copenhagen, 1998 (cit. on p. 102).

[Ali04] L. Alianelli. In: *J. Appl. Cryst.* 37 (2004), p. 732 (cit. on p. 12).

[Cla+66] C.D. Clark et al. In: *J. Sci. Instrum.* 43 (1966), p. 1 (cit. on p. 12).

[Cla+98] K. N. Clausen et al. "The RITA spectrometer at Risø - design considerations and recent results". In: *Physica B* 241-243 (1998), pp. 50–55 (cit. on p. 12).

[Cop93] J.R.D. Copley. In: *J. Neut. Research* 1 (1993), p. 21 (cit. on pp. 12, 13, 22).

[Cop03] J.R.D. Copley. In: *Nucl. Instr. Meth.* A510 (2003), p. 318 (cit. on p. 12).

[Cop+86] J. R. D. Copley et al. "Improved Monte Carlo calculation of multiple scattering effects in thermal neutron scattering experiments". In: *Comput. Phys. Commun.* 40 (1986), p. 337. ISSN: 0010-4655. DOI: `http://dx.doi.org/10.1016/0010-4655(86)90118-9` (cit. on p. 13).

[Cus03] L. D. Cussen. In: *J. Appl. Cryst.* 36 (2003), p. 1204 (cit. on p. 13).

[Far+02] E. Farhi et al. In: *Appl. Phys.* A 74 (2002), S1471 (cit. on pp. 12, 13, 22).

[FMM47] E. Fermi, J. Marshall, and L. Marshall. In: *Phys. Rev.* 72 (1947), p. 193 (cit. on p. 12).

[Fre83] A.K. Freund. In: *Nucl. Instr. Meth.* 213 (1983), p. 495 (cit. on p. 12).

[GRR92] Grimmett, G. R., and Stirzakerand D. R. *Probability and Random Processes, 2nd Edition*. Clarendon Press, Oxford, 1992 (cit. on p. 22).

[Jam80] F. James. In: *Rep. Prog. Phys.* 43 (1980), p. 1145 (cit. on pp. 13, 22, 26).

[LW02] W.-T. Lee and X.-L. Wang. In: *Neutron News* 13 (2002). See website http://www.sns.gov/ p. 30 (cit. on pp. 11, 13).

[LN99] K. Lefmann and K. Nielsen. "McStas, a general software package for neutron ray-tracing simulations". In: *Neutron News* 10 (1999), pp. 20–23. DOI: 10.1080/10448639908233684 (cit. on pp. 9, 13).

[Lef+00] K. Lefmann et al. "Added flexibility in triple axis spectrometers: the two RITAs at Risø". In: *Physica B* 283 (2000), pp. 343–354 (cit. on p. 12).

[Lie05] K. Lieutenant. In: *J. Phys.: Condens. Matter* 17 (2005), S167 (cit. on pp. 12, 13, 22).

[Low60] R.A. Lowde. In: *J. Nucl. Energy, Part A: Reactor Sci.* 11 (1960), p. 69 (cit. on p. 12).

[MS63] H. Maier-Leibnitz and T. Springer. In: *Reactor Sci. Technol.* 17 (1963), p. 217 (cit. on p. 12).

[Man+04] G. Manzin et al. In: *Nucl. Instr. Meth.* A535 (2004), p. 102 (cit. on p. 13).

[Mas+95] T. E. Mason et al. "RITA: The reinvented triple axis spectrometer". In: *Can. J. Phys.* 73 (1995), pp. 697–702 (cit. on p. 12).

[Mil90] D.F.R. Mildner. In: *Nucl. Instr. Meth.* A290 (1990), p. 189 (cit. on pp. 12, 13).

[MPC77] D.F.R. Mildner, C.A. Pellizari, and J.M. Carpenter. In: *Acta. Cryst. A* 33 (1977), p. 954 (cit. on p. 13).

[NM65] J.A. Nelder and R. Mead. In: *Computer Journal* 7 (1965), pp. 308–313 (cit. on p. 42).

[Pes+89] V. Peskov et al. In: *Nucl. Instr. and Meth.* A277 (1989), p. 547 (cit. on p. 13).

[Pet05] J. Peters. In: *Nucl. Instr. Meth.* A540 (2005), p. 419 (cit. on p. 12).

[Pre+86] W. H. Press et al. *Numerical Recipes in C*. Cambridge University Press, 1986 (cit. on p. 118).

[Pre+02] W.H. Press et al. *Numerical Recipes (2nd Edition)*. Cambridge University Press, 2002 (cit. on p. 42).

[Rad74] V. Radeka. In: *IEEE Trans. Nucl. Sci.* NS-21 (1974), p. 51 (cit. on p. 13).

[SK97] J. Saroun and J. Kulda. In: *Physica B* 234 (1997). See website http://omega.ujf.cas.cz/res p. 1102 (cit. on pp. 11, 13).

[Sch+04] C. Schanzer et al. In: *Nucl. Instr. Meth.* A 529 (2004), p. 63 (cit. on pp. 12, 13, 22).

[Sea97] V.F. Sears. In: *Acta Cryst.* A53 (1997), p. 35 (cit. on p. 12).

[See+00] P. A. Seeger et al. In: *Physica B* 283 (2000), p. 433 (cit. on pp. 11, 13).

[SST02] G. Shirane, S.M. Shapiro, and J.M. Tranquada. *Neutron Scattering with a Triple-Axis Spectrometer*. Cambridge University Press, 2002 (cit. on p. 12).

[Wec+00] D. Wechsler et al. In: *Neutron News* 25 (2000), p. 11 (cit. on pp. 11, 13).

[ZLa04] G. Zsigmond, K. Lieutenant, and S. Manoshin et al. In: *Nucl. Instr. Meth.* A 529 (2004), p. 218 (cit. on pp. 12, 13, 22).