Technical University of Denmark

User and programmers guide to the neutron ray-tracing package McStas, version 1.2

Nielsen, Kristian; Lefmann, K.

Publication date:
2000

Document Version
Publisher's PDF, also known as Version of record

Link back to DTU Orbit

Citation (APA):
Nielsen, K., & Lefmann, K. (2000). User and programmers guide to the neutron ray-tracing package McStas, version 1.2. (Denmark. Forskningscenter Risoe. Risoe-R; No. 1175(EN)).

Risø–R–1175(EN)

User and Programmers Guide to the Neutron Ray-Tracing Package McStas, Version 1.2

Kristian Nielsen, Kim Lefmann

Risø National Laboratory, Roskilde, Denmark
January 2000

Abstract

The software package McStas is a tool for writing Monte Carlo ray-tracing simulations of neutron scattering instruments with very 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 as well as the design of new instrumentation. 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 give essentially unlimited freedom to handle more unusual needs.

This report constitutes the reference manual for McStas, and contains full documentation for all ascpects of the program. It covers the various ways to compile and run simulations; a description of the metalanguage used to define simulations; a full description of all algorithms used to calculate the effects of the various optical components in instruments; and some example simulations performed with the program.

This report documents McStas version 1.2, released January 31, 2000.

Front page illustration:

Simulated scattering from a vanadium sample taking into account the secondary extinction. See section D.1.

ISBN 87–550–2683–4
ISBN 87–550–2684–2 (Internet)
ISSN 0106–2840

Information Service Department · Risø · 2000

Contents

Preface and acknowledgements

This document contains information on the Risø Monte Carlo neutron ray-tracing program McStas version 1.2, an update to the initial release 1.0 as presented in Ref. [2]. The reader of this document is supposed to have some knowledge of neutron scattering, whereas only little knowledge about simulation techniques is required. In a few places, we also assume familiarity with the use of C, UNIX and of the world wide web (WWW).

It is a pleasure to thank Prof. Kurt N. Clausen for his continuous support to this project and for having initiated the work in the first place. Both he and our other collaborators, Henrik M. Rønnow and Mark Hagen have made major contributions to the project. Also the contributions from our test users, the students Asger Abrahamsen, Niels Bech Christensen, and Erik Lauridsen, are gratefully acknowledged; they gave us an excellent opportunity to pinpoint a vast amount of serious errors in the test version. Useful comments to this document itself have been given by Bella Lake and Alan Tennant. We have also benefited from discussions with many other people in the neutron scattering community, too numerous to mention here.

Philipp Bernhardt contributed the two chopper components in sections 5.4.2 and 5.4.3, for which we are very grateful. We encourage other users to contribute components with manual entries for inclusion in future versions of McStas.

This project has been supported by the European Union through the XENNI program and the RTD "Cool Neutrons" and "SCANS" programs. This version of the project was released January 31, 2000.

In case any errors, questions, suggestions, or other need for support should arise, do not hesitate to contact the authors

Kristian Nielsen, Condensed Matter Physics and Chemistry Department,
Risø National Laboratory, 4000 Roskilde, Denmark.
phone +45 46 77 55 15, e-mail `kristian.nielsen@risoe.dk`

Kim Lefmann, Condensed Matter Physics and Chemistry Department,
Risø National Laboratory, 4000 Roskilde, Denmark.
phone +45 46 77 47 13, e-mail `kim.lefmann@risoe.dk`

or consult the McStas WWW home page [1].

Chapter 1

Introduction to McStas

Efficient design and optimization of neutron spectrometers is a formidable challenge. Monte Carlo techniques are well matched to meeting this challenge. However, no existing package offers a general framework for tackling the problems currently faced at reactor and spallation sources. The McStas project is designed to provide such a framework.

McStas (*Monte Carlo Simulations of Triple Axis Spectrometers*) 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 simulations written in ANSI-C. The present version supports both continuous and pulsed source type instruments, and includes a library of standard components including single, position-sensitive, and time-of flight detectors, supermirror guides, monochromators/analysers, a velocity selector and a disk chopper, and powder and vanadium samples.

The McStas package is written in ANSI-C and is freely available for down-load from the project home page [1]. The package is actively being developed and supported at Risø. The system is well tested and is supplied with several examples and with documentation in the form of this manual.

1.1 Background

The McStas project is the main part of a major effort in Monte Carlo simulations for neutron scattering at Risø National Laboratory. Simulation tools are urgently needed, not only to better utilize existing instruments (RITA [3, 4]), but also to design instrument upgrades (TAS7), and to plan completely new instruments for new sources (European Spallation Source, ESS [5]). Writing programs in C or FORTRAN for each of the different cases involves a huge effort, with debugging presenting particularly difficult problems. A higher level tool specially designed for the needs of simulating neutron instruments is needed. As there was no existing simulation software that would fulfill our needs, the McStas project was initiated.

1.1.1 The goals of McStas

The McStas project has four main goals:

Correctness It is essential to minimize the potential for bugs in computer simulations. When 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, when a simulation contains bugs it produces wrong results, and unless the results are far off, you may not know about it! Complex simulations involve hundreds or even thousands of lines of formulae, and "to err is human". Thus the system should be designed from the start to help minimize the potential for bugs to be introduced in the first place, and provide good tools for testing to maximize the chances of finding the bugs that do creep in.

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 needs. The tool must not have fundamental limitations that restrict its potential usage. Thus the McStas systems needs to be flexible enough to simulate different kinds of instruments (triple-axis, time-of-flight and possible hybrids) as well as many different kind of optical components, and it must also be extensible so that future, as yet unforeseen, needs can be satisfied.

Power "*Simple things should be simple; complex things should be possible*". New ideas should be easy to try out; 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, then the simulation should be running in the afternoon.

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

1.2 The design of McStas

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

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

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

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

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

4. Presentation of the calculations, graphical or otherwise.

Though obviously interrelated, these four layers can be usefully 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 might want to simulate a new instrument without having to reinvent the code for simulating the individual components in the instrument. A third user may write an intricate simulation of a complex analyser such as the one in the RITA spectrometer, 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. And 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; in a matter of minutes the instrument definition can be modified and a new simulation program generated. It also makes it simple to debug new components in isolation. A test instrument definition is written containing a neutron source, the component to be tested, and whatever detectors are useful, and the component can be thoroughly tested before being used in a complex simulation with many different components.

The McStas system is based on ANSI-C, making it both efficient and portable. The meta-language allows the user to embed arbitrary C code in the specifications. *Flexibility* is thus ensured since the full power of the C language is available if needed.

1.3 Overview

The McStas system consists of the following major parts:

- The McStas compiler. Section 2.3 explains how to compile and install the compiler and associated files, while section 2.4 explains how to run the compiler to produce simulations. Section 2.5 explains how to run the generated simulations.
- The McStas meta-language, described in chapter 3. This chapter also describes a library of kernel functions and definitions that aid in the writing of simulations.
- The McStas component library. A collection of already written, well-tested optical components that can be used in simulations. This library is documented in detail in chapter 5. Code for the components can be found in appendix B.
- A collection of example instrument definitions, described in chapter 6, with source code given in appendix C.
- A number of front-end programs that are used to run the generated simulations and aid in the data collection and analysis from the results. These are described in section 2.6

In addition, some of the results that have been obtained from simulations produced with McStas are described in appendix D. An explanation of McStas terminology can be found in appendix E. Some planned extensions are listed in chapter 7, and a list of library calls that are used in component definitions appears in appendix A.

Chapter 2

Running McStas

This is a tutorial for the first time user in how to make McStas work – either via downloading the ANSI-C source code over the WWW, or on direct connection to the host computer. This release of McStas assumes that the user will compile the software himself on a Unix-like system. Binary installations for Digital Unix, ix86 Linux, and HPUX will be made available on request.

To use McStas one first writes an instrument definition file which describes the instrument to be simulated (or obtains a definition from the `examples/` directory in the distribution or from another source). This is then compiled with the McStas compiler to produce a C program. The C program can then be compiled with a C compiler and run in combination with various front-end programs to for example output the intensity at the detector as a motor position is varied.

2.1 Using the graphical user interface

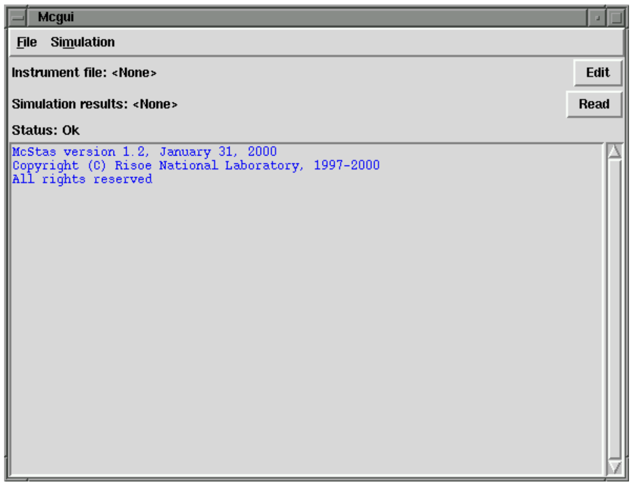
This section gives an ultra-brief overview of how to use McStas once it has been properly installed. It is intended for those who do not read manuals if they can avoid it. For details on the different steps, see the following sections. This section uses the `vanadium_example.instr` file supplied in the `examples/` directory of the McStas distribution, see appendix C.1.

To start the graphical user interface of McStas, run the command `mcgui`. This will open a window with some menus *etc*, see figure 2.1.

To load an instrument, select “Open instrument” from the “File” menu. Open the file `vanadium_example.instr` in the McStas distribution. Select “Run simulation” from the “Simulation” menu. McStas will translate the definition into an executable program and pop up a dialog window. Type a value for the “ROT” parameter (*eg.* 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.

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

For a slightly longer gentle introduction to McStas, see the McStas tutorial (available



Instrument file: <None>

Simulation results: <None>

Status: Ok

```
McStas version 1.2, January 31, 2000
Copyright (C) Risoe National Laboratory, 1997-2000
All rights reserved
```

Figure 2.1: The graphical user interface `mcgui`.

from [1]).

2.2 Obtaining McStas

The source code for McStas may be obtained from Risø on a CD-Rom, or it may be downloaded from the McStas WWW home page [1]. In either case, the source should be available in a file named `mcstas.tar.gz` (the CD-Rom also contains this file under the name `mcstas.tgz` for systems that do not understand long filenames, as well as the unpacked sources in the directory `mcstas/`).

The conditions on the use of McStas can be read in the files `LICENSE` and `LICENSE.LIB` in the distribution. Essentially, McStas may be used freely, but copies of McStas may not be passed on to others. We are considering releasing future versions of McStas under a more liberal license.

2.3 Compiling McStas from source

Compilation and installation of McStas proceeds in three simple steps. First, the sources must be unpacked:

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

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

```
./configure
make
```

Finally, McStas must be installed:

```
make install
```

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

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

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

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

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

To use the McStas front-end programs (see section 2.6), certain auxiliary packages must be installed, as described in the README file in the distribution. These packages are all freely available, and have been included on the McStas CD-Rom and on the WWW home page. Some of these packages may be supplied with the operating system; for example, all needed packages are included with Debian/GNU Linux [6].

Note that the core parts of McStas, including the McStas compiler and any generated simulations, can work with no additional software apart from an ANSI-C compiler.

2.4 Running the instrument compiler

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

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

```
mcstas name.instr
```

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

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

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

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

2.4.1 Code generation options

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

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

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

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

2.4.2 Specifying the location of files

The McStas compiler needs to be able to find various files during compilation, some explicitly requested by the user (such as component definitions and files referenced by `%include`), and some used internally to generate the simulations. 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 files.

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

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

for csh/tcsh users, or

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

for bash/Bourne shell users.

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

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

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

2.4.3 Embedding the generated simulations in other programs

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

- `--no-main` This option makes McStas omit the `main()` function in the generated simulation program. The user must then arrange for the function `mcstas_main()` to be called in some way.
- `--no-runtime` Normally, the McStas compiler copies into the generated simulation program 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; the user must then explicitly link with the file `mcstas-r.c` from the McStas distribution.

Users that need these options are encouraged to contact the authors for further help; see page 7 for contact addresses.

2.4.4 Running the C compiler

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

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

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

Monte Carlo simulations are computationally intensive, and it is often desirable to have them run as fast as possible. Some success can be had in this respect 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 [1]):

- Intel x86 (“PC”) with Linux and GCC, using options `gcc -O3`.
- Intel x86 with Linux and EGCS (GCC derivate) using options `egcc -O6`.
- Intel x86 with Linux and PGCC (pentium-optimized GCC derivate), using options `gcc -O6 -mstack-align-double`.

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

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

2.5 Running the simulations

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

```
./name.out
```

Note the leading dot, which is needed if the current directory is not in the path searched by the shell. When used in this way, the simulation will prompt for the values of any instrument parameters such as motor positions, and then run the simulation and output the results. This will output only a single data point compared to the tens of points usually needed in a scan (such as of motor positions in a triple-axis instrument). Often the simulation will be run using one of several available front-ends, as described in the next section. These front-ends help manage output from the potentially many detectors in the instruments, as well as running the simulation for each data point in a scan.

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

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

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

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

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

By default, McStas simulations write their results into several data files in the current directory, overwriting any previous files stored there. The `--dir=`*dir* or `-d`*dir* option causes the files to be placed instead in a newly created directory *dir*; to prevent overwriting

`-s` *seed* `--seed=`*seed*	Set the initial seed for the random number generator. This may be useful for testing to make each run use the same random number sequence.
`-n` *count* `--ncount=`*count*	Set the number of neutron histories to simulate. The default is 1,000,000.
`-d` *dir* `--dir=`*dir*	Create a new directory *dir* and put all data files in that directory.
`-f` *file* `--file=`*file*	Write all data into a single file *file*
`-a` `--ascii-only`	Do not put any headers in the data files.
`-h` `--help`	Show a short help message with the options accepted, including the names of the parameters of the instrument.
`-i` `--info`	Show extensive information on the simulation and the instrument definition it was generated from.
`-t` `--trace`	This option makes the simulation output the state of every neutron as it passes through every component. Requires that the `-t` (or `--trace`) option is also given to the McStas compiler when the simulation is generated.
param`=`*value*	Set the value of an instrument parameter, rather than having to prompt for each one.

Table 2.1: Options accepted by McStas generated simulations

previous results it is an error if the directory already exists. Alternatively, all output may be written instead to a single file *file* using the `--file=`*file* or `-f`*file* option.

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

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

2.6 Using simulation front-ends

McStas includes a number of front-end programs that extend the functionality of the generated simulations. The front-end programs sit between the user and the simulations, running the simulations and presenting the output in various ways to the user.

An extended set of front-end programs is planned for future versions of McStas, including a NeXus data format option [7].

2.6.1 The graphical user interface

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

```
mcgui
```

The program may optionally be given the name of a simulation definition to load.

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

The menus

The "File" menu has the following features:

Open instrument This selects the name of an instrument file to use for other operations.

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

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

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

Quit Exit the graphical user interface front-end.

The "Simulation" menu has the following features:

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

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

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

The run dialog

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

The meaning of the different fields is as follows:

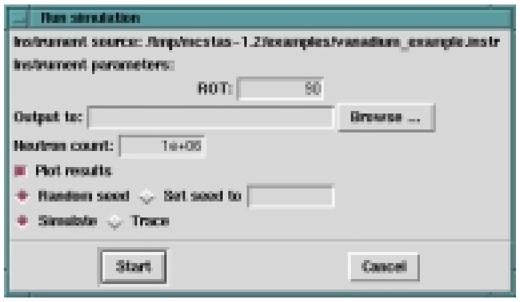

Run simulation
Instrument source: /tmp/mcstas-1.2/examples/vanadium_example.instr
Instrument parameters:
ROT: 90
Output to:
Neutron count: 1e+06
🟥 Plot results
● Random seed ○ Set seed to
● Simulate ○ Trace

Figure 2.2: The run dialog in `mcgui`.

Instrument parameters This allows the setting of the values for the input parameters of the instrument.

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

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

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

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

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

Start Run the simulation.

Cancel Abort the dialog.

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

The plot dialog

Monitors and detectors This lists all the one- and two-dimensional detectors in the instrument. Double-clicking one plots the data in the plot window.

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

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

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

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

Close This ends the dialog.

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

2.6.2 Running simulations with automatic compilation

The `mcrun` front-end is a command-line program that implements the same automatic compilation feature that is used in `mcgui`. The command

```
mcrun sim.instr parms ...
```

will compile the instrument definition *sim*`.instr` (if necessary) into an executable simulation *sim*`.out`. It will then run *sim*`.out`, passing the parameters *parms*. See section 2.5 for details on the format of *parms*.

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

2.6.3 Scans — varying simulation parameters over multiple runs

The front-end `gscan` extends McStas generated simulations with the ability to run a series of simulations, varying one or more parameters with each run, and collecting the results in a data file. We refer to such a series of runs as a *scan*. The `gscan` front-end is typically used to simulate scans on triple-axis instruments.

To run a scan with a simulation *sim* that has been previously generated with McStas and compiled with the C compiler, run a command like

```
gscan M N sim file params ...
```

Here, M is the number of simulations to run, N is the number of neutrons to simulate in each run, and *file* is the name of the file in which to output the results. The *params* argument is a list of assignments of values to instrument parameters. It may take one of two forms: *param=value* to assign a constant *value* to *param*, or *param=low,high* to vary *param* linearly between *low* and *high*. For example

```
gscan 21 1e6 ./instrum.out output.dat TTM=74 TT=-5,5 TTA=74
```

The output file is in ASCII format, with one line for each simulation run. Each line contains the values for the non-constant instrument parameters followed by the simulated intensity, the estimated statistical error, and the neutron event count of each detector in the instrument. *I.e.* in the simple case of an instrument with a single detector where a single instrument parameter is scanned, there will be four numbers on each line: the value of the scanned parameter, the detector intensity, the estimated detector error, and the neutron event count.

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

2.6.4 Graphical display of simulations

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

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

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

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

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

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

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

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

Another way to see detail while maintaining an overview of the instrument is to use the `--zoom=`*factor* option. This magnifies the display of each component along selected axis only, eg. 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 occationally useful.

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

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

2.6.5 Plotting the results of a simulation

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

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

```
mcplot
```

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

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

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

2.6.6 Plotting resolution functions

The `mcresplot` front-end is used to plot the resolution function of a triple-axis or inverse geometry time-of-flight spectrometer, as calculated by the Res_sample component (see section 5.8.1). This front-end is still experimental, however it has been included in the release since it may be useful despite its somewhat rough user interface.

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

```
mcresplot file
```

Here, *file* is the name of a file output from a simulation using the Res_monitor component (section 5.5.12). The front-end will open two windows. One shows a three-dimensional visualization of the resolution function using the two components of $\boldsymbol{Q}$ in the scattering plane and ω. The plot may be rotated using the mouse while pressing the left button, and zoomed while pressing the right button.

The other window displays the covariance matrix of the resolution function and the resulting resolution matrix. This is mainly useful for triple-axis spectrometers. The bottom four plots visualize the covariance matrix using four different projections. The top left corner shows histograms of the resolution function along the three axes of $\boldsymbol{Q}$ and along the ω axis.

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

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

2.7 Analyzing and visualizing the simulation results

To analyze simulation results, one uses the same tools as for analyzing experimental data, *i.e.* programs such as the MATLAB packages Mview and Mfit [8] used at Risø. The output files from simulations are simply columns of ASCII text that most programs should be able to read. A future version of McStas will support output in the NeXus format [7].

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

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

Single-point detectors output the neutron intensity, the estimated error, and the neutron event count as numbers on the terminal. (The results from a series of simulations may be combined in a data file using the `gscan` front-end as explained in section 2.6.3).

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

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

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

Chapter 3

The McStas kernel and meta-language

Instrument definitions are written in a special McStas meta-language which is translated automatically by the McStas compiler into a C program that performs the simulation. The meta-language is custom-designed for neutron scattering and serves two main purposes: to specify the interaction of a single neutron with a single optical component, and 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 done by C code embedded in the meta-language description of the components. It is possible to set up an instrument from existing components and run a simulation without writing a single line of C code, working entirely in the meta-language. On the other hand, the full power of the C language is available for special-purpose setups in advanced simulations, and for computing neutron trajectories in the components.

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

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

3.1 Notational conventions

Simulations generated by McStas use a semi-classical description of the neutron to compute the neutron trajectory through the instrument and its interaction with the different components. In the current version of McStas the effect of gravity is not taken into account by the existing components, though it is perfectly possible to handle gravity in user-written components if so desired.

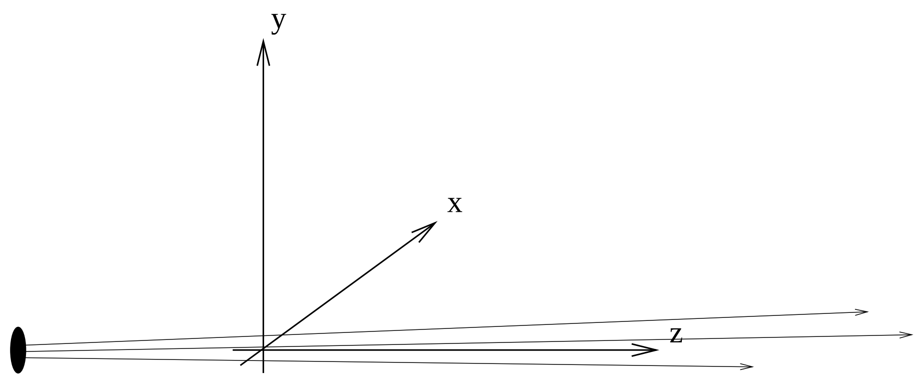


Figure 3.1: conventions for the orientations of the axis in simulations.

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

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

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 spin[1] s_1, s_2 having no dimension. In addition, the outgoing neutron has an associated weight p which is used to model fractional neutrons in the Monte Carlo simulation (so $p = 0.2$ means that a neutron following this path has a 20% chance of reaching the present position without being absorbed or scattered away from the instrument).

3.2 Syntactical conventions

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

Keywords are not case-sensitive, so for example "`DEFINE`", "`define`", and "`dEfInE`"

[1]The spin is ignored in the current version 1.2 of McStas. However, while not documented in this manual, preliminary support for components that handle the neutron spin is implemented using the POLARISATION PARAMETER construct. We are currently working together with Trefor Roberts at the ILL to get a correct handling of the spin.

are all equivalent. However, by convention we always write keywords in uppercase to distinguish them from identifiers and C language keywords. In contrast, McStas identifiers, like C identifiers and keywords, *are* case sensitive, another good reason to use a consistent case convention for keywords.

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

```
%include "file"
```

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

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

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

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

3.3 Writing instrument definitions

The purpose of the instrument definition is to specify a sequence of components, along with their position and parameters, which together make up an instrument. Each component is given its own local coordinate system, the position and orientation of which may be specified by its translation and rotation relative to another component. Some complete examples of instrument definitions can be found in appendix C.

An instrument definition looks as follows:

3.3.1 The instrument definition head

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

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

3.3.2 The `DECLARE` section

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

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

3.3.3 The `INITIALIZE` section

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

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

3.3.4 The `TRACE` section

The `TRACE` keyword starts a section giving the list of components that constitute the instrument

Components are declared like this:

`COMPONENT` $name = comp(p_1 = v_1, p_2 = v_2, \ldots)$

This declares a component named *name* that is an instance of the component definition named *comp*. The parameter list gives the setting and definition parameters for the component. The values $v_1, v_2, \ldots$ may be constant numbers, strings, names of instrument parameters, or names of C identifiers. To assign the value of a general expression to a parameter, it is currently 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. A future version of McStas will make it possible to write the expression directly in the argument list of the component.

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 quite possible (and usual) to use a component definition multiple times in an instrument description.

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

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

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

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

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

The `AT` modifier is required.

Rotation is achieved similarly by writing

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

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

3.3.5 The `FINALLY` section

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

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

3.3.6 The end of the instrument definition

The end of the instrument definition is marked using the keyword

```
END
```

3.4 Writing component definitions

The purpose of a component definition is to model the interaction of a neutron with the component. Given the state of the incoming neutron, the component definition calculates the state of the neutron when it leaves the component. The calculation of the effect of the component on the neutron is performed by a block of embedded C code. Complete examples of component definitions can be found in appendix B.

A component definition looks as follows:

3.4.1 The component definition header

`DEFINE COMPONENT` *name*

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

`DEFINITION PARAMETERS` $(d_1, d_2, \ldots)$

This declares the definition parameters of the component. Definition parameters define properties of the component that cannot change for a given physical incarnation of the component, but which might vary among different components of the same type. Typical examples are physical dimensions, crystal plane distances, etc.

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

This declares the setting parameters of the component. Setting parameters define the configuration of the component and typically changes during an experiment. An example is the position of a motor.

The reason for the distinction between definition and setting parameters is mainly historical. In the current McStas version, there is not much difference between them in practice, and the use of one or the other is often just a question of conventions. There is one important difference, though. The argument for a setting parameter must be of numeric type (*e.g.* a C `double`). If a parameter is of non-numeric type (*e.g.* a string or a macro definition), it must be made a definition parameter.

It is possible that setting and definition parameters will be merged in a future version, though they will both be supported for backward compatibility.

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

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

The `OUTPUT PARAMETERS` section is optional.

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

This declares the parameters that define the state of the incoming neutron. The task of the component code is to assign new values to these parameters based on the old values and the values of the definition and setting parameters.

3.4.2 The `DECLARE` section

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

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

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

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

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

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

3.4.3 The `INITIALIZE` section

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

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

3.4.4 The `TRACE` section

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

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

The C code may also execute the special macro `ABSORB` to indicate that the neutron has been absorbed in the component; the simulation of that neutron will then be aborted. If the component simulates multiple events (for example multiple reflections in a guide, or multiple scattering in a powder sample), the special macro `SCATTER` should be called. This does not affect the results of the simulation in any way, but it allows the front-end programs to visualize the scattering events properly. The `SCATTER` macro should be called with the state parameters set to the proper calues for the scattering event. For an example of `SCATTER`, see the Channeled_guide component (section 5.3.4).

3.4.5 The `FINALLY` section

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

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

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

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

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

Here, t is a string giving a short descriptive title for the results, eg. “Single monitor”.

One-dimensional detectors/monitors The results of a one-dimensional detector/ monitor are output 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 (eg. “Energy monitor”),
- *xlabel* is a string giving a descriptive label for the X axis in a plot (eg. “Energy [meV]”),
- *ylabel* is a string giving a descriptive label for the Y axis of a plot (eg. “Intensity”),
- *xvar* is a string giving the name of the variable on the X axis (eg. “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 output using the following macro:

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

Here,

- t is a string giving a descriptive title (eg. “PSD monitor”),
- *xlabel* is a string giving a descriptive label for the X axis in a plot (eg. “X position [cm]”),
- *ylabel* is a string giving a descriptive label for the Y axis of a plot (eg. “Y position [cm]”),
- $x_{\min}$ is the lower limit for the X axis,
- $x_{\max}$ is the upper limit for the X axis,
- $y_{\min}$ is the lower limit for the Y axis,
- $y_{\max}$ is the upper limit for the Y axis,
- m is the number of elements in the detector arrays along the X axis,
- n is the number of elements in the detector arrays along the Y axis,
- &N[0][0] is a pointer to the first element in the array of N values for the detector component,
- &p[0][0] is a pointer to the first element in the array of p values for the detector component,
- &$p2$[0][0] is a pointer to the first element in the array of $p2$ values for the detector component,
- *filename* is a string giving the name of the file in which to store the data.

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

3.4.6 The `MCDISPLAY` section

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3.4.7 The end of the component definition

```
END
```

This marks the end of the component definition.

Chapter 4

Monte Carlo Techniques and simulation strategy

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

4.1 The neutron weight, p

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

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

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

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

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

Simulation by weight adjustment is performed whenever possible. This includes

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

4.1.1 Statistical errors of non-integer counts

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

4.2 Weight factor transformations during a Monte Carlo choice

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

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

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

For the sources present in version 1.2, only the $(\mathbf{\Omega}, \mathbf{r})$ dependence of the correction factors are taken into account.

The weight factor transformation rule (4.8) is of course also valid for other types of Monte Carlo choices, although the probability distributions may depend upon different parameters. An important example is elastic scattering from a powder sample, where the Monte-Carlo choices are the scattering position and the final neutron direction. See subsection 5.7.

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

4.3 Focusing components

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

If *e.g.* a sample scatters isotropically over the whole 4π solid angle, and all interesting directions are known to be contained within a certain solid angle interval $\Delta\mathbf{\Omega}$, only these solid angles are used for the Monte Carlo choice of scattering direction. According to (4.8), the weight factor will then have to be changed by the (fixed) amount $\pi_j = |\Delta\mathbf{\Omega}|/(4\pi)$. One thus ensures that the mean simulated intensity is unchanged during a "correct" focusing, while a too narrow focusing will result in a lower (*i.e.* wrong) intensity, since one cuts away neutrons that would otherwise have counted..

One could also think of using adaptive importance sampling, so that McStas during the simulations will determine the most interesting directions and gradually change the focusing according to that. A first implementation of this idea is found in the Source_adapt component, described in section 5.1.6.

4.4 Transformation of random numbers

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

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

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

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

By indefinite integration we reach

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

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

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

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

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

Chapter 5

The component library

This section is devoted to a description of components included in the component library. The component library is maintained by the Risø group. All components were written at Risø except the chopper components in sections 5.4.2 and 5.4.3 which have been kindly contributed by Philipp Bernhardt, Lehrstuhl für Kristallographie und Strukturphysik. Users are encouraged to send contributions to us for inclusion in future releases.

In the explanations of the individual components we will use the usual symbols $\mathbf{r}$ for the position (x, y, z) of the particle (unit m), and $\mathbf{v}$ for the particle velocity (v_x, v_y, v_z) (unit m/s). Another frequently used symbol is the wave vector $\mathbf{k} = m_\mathrm{N}\mathbf{v}/\hbar$, where m_N is the neutron mass. $\mathbf{k}$ is usually given in Å^{-1}, while neutron energies are given in meV. In general, vectors are denoted by boldface symbols. Subscripts "i" and "f" denotes "initial" and "final", respectively, and are used in connection with the neutron state before and after a scattering event. Note that all mentioning of component geometry refer to the local coordinate system of the individual component.

The source code for components is listed in Appendix B. The components follow the same numbering in the Appendix as in the main text, *e.g.* component **Arm**, subsection 5.2.1, appears in the Appendix as B.2.1. Source code for many of the more trivial components are not included in this manual. All sources may be found in the `lib/mcstas/` subdirectory of the McStas installation; the default is `/usr/local/lib/mcstas/`.

5.1 Source components

The main function of the source components is to determine a set of initial parameters $(\mathbf{r}, \mathbf{v})$, or equivalent $(\mathbf{r}, v, \mathbf{\Omega})$, for each neutron. This is done by Monte Carlo choices. In the current sources no polarization dependence is implemented, whence we let $\mathbf{s} = (0, 0)$.

The sources to be presented in the following all make their Monte Carlo choices on the basis of simple analytical expressions (*e.g.* the energy distribution). More realistic sources would require that (at least) the Monte-Carlo choice for the initial energy was made on basis of a measured, tabulated energy spectrum. This is planned to be implemented in a later version of McStas.

5.1.1 Source_flat: A circular continuous source with a flat energy spectrum

This component **Source_flat** is a simple continous source with a flat energy distribution. The time-of-flight aspect is not relevant for this component, so we put $t = 0$ for all neutrons.

The initial neutron position is chosen randomly from within a circle of radius r_s in the $z = 0$ plane. This is a fair approximation of a cylindrical cold source with the beam going out along the cylinder axis, like the one at Risø.

The initial neutron velocity direction is focused within a solid angle, defined by a rectangular target of width w, height h, parallel to the xy plane placed at $(0, 0, z_\mathrm{f})$. A small angle approximation is used, assuming that $w, h \ll z_\mathrm{f}$.

The weight multiplier of the created neutron, π_1, is set to the solid angle of the focusing opening divided by 4π, see discussion in 4.3

The input parameters of **Source_flat** are the source radius, r_s, the distance to the target, z_f, the dimensions of the target, w and h, and the centre and spread of the energy distribution, E_0 and ΔE.

5.1.2 Source_flat_lambda: A continous source with flat wavelength spectrum

The component **Source_flat_lambda** is similar to the Source_flat component, except that the spectrum is flat in wavelength, rather than in energy.

The input parameters for Source_flat_lambda are *radius* to set the source radius in meters; *dist*, *xw*, and *yh* to set the focusing as for Source_flat; and *lambda_0* and *d_lambda* to set the range of wavelength emitted (the range will be from *lambda_0* $-$ *d_lambda* to *lambda_0* $+$ *d_lambda*).

5.1.3 Source_flux_lambda: A continuous source with absolute flux

The component **Source_flux_lambda** is a variation on the Source_flat_lambda component. The only difference is the possibility to specify the absolute flux of the source. The specified flux is used to adjust the initial neutron weight so that the intensity in the detectors is directly comparable to a measurement of one second on a real source with the same flux. This also makes the simulated detector intensities independent of the number of neutron histories simulated, easing the comparison between different simulation runs (though of course more neutron histories will give better statistics).

The flux Φ is the number of neutrons emitted per second from a one cm^2 area on the source surface, with direction within a a one steradian solid angle, and with wavelength within a one Ångstrøm interval. The total number of neutrons emitted towards a given diaframe in one second is therefore

$$N_\mathrm{total} = \Phi A \Omega \Delta\lambda$$

where A is the source area, Ω is the solid angle of the diaframe as seen from the source surface, and $\Delta\lambda$ is the width of the wavelength interval in which neutrons are emitted (assuming a uniform wavelength spectrum). If N_sim denotes the number of neutron histories to simulate, the initial neutron weight p_0 must be set to

$$p_0 = \frac{N_{\text{total}}}{N_{\text{sim}}} = \frac{\Phi}{N_{\text{sim}}} A\Omega\Delta\lambda$$

The input parameters for Source_flux_lambda are *radius* to set the source radius in meters; *dist*, *xw*, and *yh* to set the focusing as for Source_flat; *lambda_0* and *d_lambda* to set the range of wavelength emitted (the range will be from *lambda_0* $-$ *d_lambda* to *lambda_0* $+$ *d_lambda*); and *flux* to set the source flux in units of $\text{cm}^{-2}\text{st}^{-1}\AA$.

5.1.4 Source_div: A divergent source

Source_div is a rectangular source which emits a beam of a certain divergence around the main exit direction (the direction of the z axis). The beam intensity and divergence are uniform over the whole of the source, and the energy distribution of the beam is uniform.

This component may be used as a simple model of the beam profile at the end of a guide or at the sample position.

The input parameters for Source_div are the source dimensions w and h (in m), the divergencies δ_h and δ_v (FWHM in degrees), and the mean energy E_0 and the energy spread dE (both in meV). The neutron energy range is $(E_0 - dE; E_0 + dE)$.

5.1.5 Moderator: A time-of-flight source

The simple time-of-flight source component **Moderator** resembles the source component **Source_flat** described in 5.1.1. Like **Source_flat**, **Moderator** is circular and focuses on a rectangular target. Further, the initial velocity is chosen with a linear distribution within an interval, defined by the minimum and maximum energies, E_0 and E_1, respectively.

The initial time of the neutron is determined on basis of a simple heuristical model for the time dependence of the neutron intensity from a time-of-flight source. For all neutron energies, the flux decay is assumed to be exponential,

$$\Psi(E,t) = \exp(-t/\tau(E)), \tag{5.1}$$

where the decay constant is given by

$$\tau(E) = \begin{cases} \tau_0 & ; E < E_c \\ \tau_0/[1+(E-E_c)^2/\gamma^2] & ; E \geq E_c \end{cases} \tag{5.2}$$

The input parameters for **Moderator** are the source radius, r_{s}, the minimum and maximum energies, E_0 and E_1 (in meV), the distance to the target, z_{f}, the dimensions of the target, w and h, and the decay parameters τ_0 (in μs), E_c, and γ (both in meV).

5.1.6 Source_adapt: A neutron source with adaptive importance sampling

The **Source_adapt** component is a neutron source that uses adaptive importance sampling to improve the efficiency of the simulations. It works by changing on-the-fly the probability distributions from which the initial neutron state is sampled so that samples in regions that contribute much to the accuracy of the overall result are preferred over samples that contribute little. The method can achive improvements of a factor of ten or sometimes several hundred in simulations where only a small part of the initial phase space contains useful neutrons.

The physical characteristics of the source are similar to those of Source_flat (see section 5.1.1). The source is a thin rectangle in the X-Y plane with a flat energy spectrum in a user-specified range. The flux per area per steradian per Ångstrøm per second is specified by the user; the total weight of neutrons emitted from the source will then be the same irrespectively of the number of neutron histories simulated, corresponding to one second of measurements.

The initial neutron weight is given by (see section 5.1.3 for details)

$$p_0 = \frac{N_{\mathrm{total}}}{N_{\mathrm{sim}}} = \frac{\Phi}{N_{\mathrm{sim}}} A\Omega\Delta\lambda$$

Here $\Delta\lambda$ is the total wavelength range of the source; since the spectrum is flat in energy (but not in wavelength), the flux will actually be different for different energies. A later version of this component will probably adapt (in a backward-compatible way) a more sensible way to specify the flux. For now, an energy or wavelength monitor (see sections 5.5.7 and 5.5.8) placed just after the source will show the actual energy-dependent flux.

The adaption algorithm

The adaptive importance sampling works by subdividing the initial neutron phase space into a number of equal-sized bins. The division is done on the three dimensions of energy, horizontal position, and horizontal divergence, using N_{eng}, N_{pos}, and N_{div} number of bins in each dimension, respectively. The total number of bins is therefore

$$N_{\mathrm{bin}} = N_{\mathrm{eng}} N_{\mathrm{pos}} N_{\mathrm{div}}$$

Each bin i is assigned a sampling weight w_i; the probability of emitting a neutron within bin i is

$$P(i) = \frac{w_i}{\sum_{j=1}^{N_{\mathrm{bin}}} w_j}$$

In order to avoid false learning, the sampling weight of a bin is kept larger than w_{min}, defined as

$$w_{\mathrm{min}} = \frac{\beta}{N_{\mathrm{bin}}} \sum_{j=1}^{N_{\mathrm{bin}}} w_j, \qquad 0 \leq \beta \leq 1$$

This way a (small) fraction β of the neutrons are sampled uniformly from all bins, while the fraction $(1-\beta)$ are sampled in an adaptive way.

Compared to a uniform sampling of the phase space (where the probability of each bin is $1/N_{\mathrm{bin}}$), the neutron weight must be adjusted by the amount

$$\pi_i = \frac{1/N_{\mathrm{bin}}}{P(i)} = \frac{\sum_{j=1}^{N_{\mathrm{bin}}} w_j}{N_{\mathrm{bin}} w_i}$$

In order to set the criteria for adaption, the Adapt_check component is used (see section 5.5.13). The source attemps to sample only from bins from which neutrons are not absorbed prior to the position in the instrument at which the Adapt_check component is placed. Among those bins, the algorithm attemps to minimize the variance of the neutron weights at the Adapt_check position. Thus bins that would give high weights at the Adapt_check position are sampled more often (lowering the weights), while those with low weights are sampled less often.

Let $\pi = p_1/p_0$ denote the ratio between the neutron weight p_1 at the Adapt_check position and the initial weight p_0 just after the source. For each bin, the component keeps track of the sum ψ of π's as well as of the total number of neutrons n_i from that bin. The average weight at the Adapt_source position of bin i is thus ψ_i/n_i.

We now distribute a total sampling weight of β uniformly among all the bins, and a total weight of $(1-\beta)$ among bins in proportion to their average weight ψ_i/n_i at the Adapt_source position:

$$w_i = \frac{\beta}{N_{\mathrm{bin}}} + (1-\beta)\frac{\psi_i/n_i}{\sum_{j=1}^{N_{\mathrm{bins}}} \psi_j/n_j}$$

After each neutron event originating from bin i, the sampling weight w_i is updated.

This basic idea can be improved with a small modification. The problem is that until the source has had the time to learn the right sampling weights, neutrons may be emitted with high neutron weights (but low probability). These low probability neutrons may account for a large fraction of the total intensity in detectors, causing large variances in the result. To avoid this, the component emits early neutrons with a lower weight, and later neutrons with a higher weight to compensate. This way the neutrons that are emitted with the best adaption contribute the most to the result.

The factor with which the neutron weights are adjusted is given by a logistic curve

$$F(j) = C\frac{y_0}{y_0 + (1-y_0)e^{-r_0 j}} \tag{5.3}$$

where j is the index of the particular neutron history, $1 \le j \le N_{\mathrm{hist}}$. The constants y_0, r_0, and C are given by

$$\begin{aligned} y_0 &= \frac{2}{N_{\mathrm{bin}}} && (5.4)\\ r_0 &= \frac{1}{\alpha}\frac{1}{N_{\mathrm{hist}}}\log\left(\frac{1-y_0}{y_0}\right) && (5.5)\\ C &= 1 + \log\left(y_0 + \frac{1-y_0}{N_{\mathrm{hist}}}e^{-r_0 N_{\mathrm{hist}}}\right) && (5.6) \end{aligned}$$

The number α is given by the user and specifies (as a fraction between zero and one) the point at which the adaption is considered good. The initial fraction α of neutron histories are emitted with low weight; the rest are emitted with high weight:

$$p_0(j) = \frac{\Phi}{N_{\mathrm{sim}}}A\Omega\Delta\lambda\frac{\sum_{j=1}^{N_{\mathrm{bin}}} w_j}{N_{\mathrm{bin}} w_i}F(j)$$

The choice of the constants y_0, r_0, and C ensure that

$$\int_{t=0}^{N_{\mathrm{hist}}} F(j) = 1$$

so that the total intensity over the whole simulation will be correct

Similarly, the adjustment of sampling weights is modified so that the actual formula used is

$$w_i(j) = \frac{\beta}{N_{\text{bin}}} + (1-\beta)\frac{y_0}{y_0 + (1-y_0)e^{-r_0 j}}\frac{\psi_i/n_i}{\sum_{j=1}^{N_{\text{bins}}} \psi_j/n_j}$$

The implementation

The heart of the algorithm is a discrete distribution p. The distribution has N *bins*, $1 \ldots N$. Each bin has a value v_i; the probability of bin i is then $v_i/(\sum_{j=1}^{N} v_j)$.

Two basic operations are possible on the distribution. An *update* adds a number a to a bin, setting $v_i^{\text{new}} = v_i^{\text{old}} + a$. A *search* finds, for given input b, the minimum i such that

$$b \leq \sum_{j=1}^{i} v_j.$$

The search operation is used to sample from the distribution p. If r is a uniformly distributed random number on the interval $[0; \sum_{j=1}^{N} v_j]$ then $i = \text{search}(r)$ is a random number distributed according to p. This is seen from the inequality

$$\sum_{j=1}^{i-1} v_j < r \leq \sum_{j=1}^{i} v_j,$$

from which $r \in [\sum_{j=1}^{i-1} v_j; v_i + \sum_{j=1}^{i-1} v_j]$ which is an interval of length v_i. Hence the probability of i is $v_i/(\sum_{j=1}^{N} v_j)$. The update operation is used to adapt the distribution to the problem at hand during a simulation. Both the update and the add operation can be performed very efficiently; how this is achieved will be described elsewhere.

The input parameters for Source_adapt are *xmin*, *xmax*, *ymin*, and *ymax* in meters to set the source dimensions; *dist*, *xw*, and *yh* to set the focusing as for Source_flat (section 5.1.1); *E0* and *dE* to set the range of energies emitted, in meV (the range will be from *E0*$-$ *dE* to *E0*$+$ *dE*); flux to set the source flux Φ in cm^{-2}st^{-1}Ås^{-1}; N_{eng}, N_{pos}, and N_{div} to set the number of bins in each dimensions; *alpha* and *beta* to set the parameters α and β as described above; and *filename* to give the name of a file in which to output the final sampling destribution.

A good general-purpose value for α and β is $\alpha = \beta = 0.25$. The number of bins to choose will depend on the application. More bins will allow better adaption of the sampling, but will require more neutron histories to be simulated before a good adaption is obtained. The output of the sampling distribution is only meant for debugging, and the units on the axis are not necessarily meaningful. Setting the filename to `NULL` disables the output of the sampling distribution.

5.2 Simple optical components: Arms, slits, collimators, filters

Below we list a number of simple optical components which require only a minimum of explanation.

5.2.1 Arm: The generic component

The component **Arm** is empty; is resembles an optical bench and has no effect on the neutron. The function of this component is only to set up a local frame of reference within the instrument definition. Other components of the same arm/optical bench may then be positioned relative to the arm component using the McStas meta-language. The use of arm components in the instrument definitions is not required but is recommended for clarity.

Arm has no input parameters. For more about the use of this component, see the sample instrument definitions listed in Appendix C.

5.2.2 Slit: The rectangular slit

The component **Slit** is a very simple construction. It sets up a rectangular opening at $z = 0$, and propagates the neutrons onto the plane of this rectangle by the kernel call PROP_Z0.

Neutrons within the slit opening are unaffected, while all other neutrons (no matter how far from the opening their paths intersect the plane) are discarded by the kernel call ABSORB. By this simplification, some neutrons contributing to the background in a real experiment will be neglected. These are the ones that scatter off the inner side of the slit, penetrates the slit material, or that clear one of the outer edges of the slit.

The input parameters of **Slit** are the four coordinates, $(x_{\rm min}, x_{\rm max}, y_{\rm min}, y_{\rm max})$ defining the opening of the rectangle.

5.2.3 Circular_slit: The circular slit

The component **Circular_slit** defines a circle in the $z = 0$ plane, centered in the origin. In analogy with **Slit**, neutrons are propagated to this plane, and those which intersect the plane outside the circle are ABSORB'ed.

The only input parameter of **Circular_slit** is the radius, r, of the circle.

5.2.4 Beamstop_rectangular: The rectangular beam stop

The component **Beamstop_rectangular** models a thin, infinitely absorbing rectangle in the X-Y plane, centered on the origin. The input parameters are *xmin*, *xmax*, *ymin*, and *ymax* defining the edges of the slit in meters.

5.2.5 Beamstop_circular: The circular beam stop

The component **Beamstop_circular** models a thin, infinitely absorbing circular disk in the X-Y plane, centered on the origin. It takes a single input parameter *radius* to define the circle radius in meters.

5.2.6 Soller: The simple Soller blade collimator

The component **Soller** defines two rectangular openings like the one in **Slit**. Neutrons not clearing both these openings are ABSORB'ed, see the discussion in 5.2.2. The collimating effect is taken care of by employing an ideal triangular transmission through the collimator,

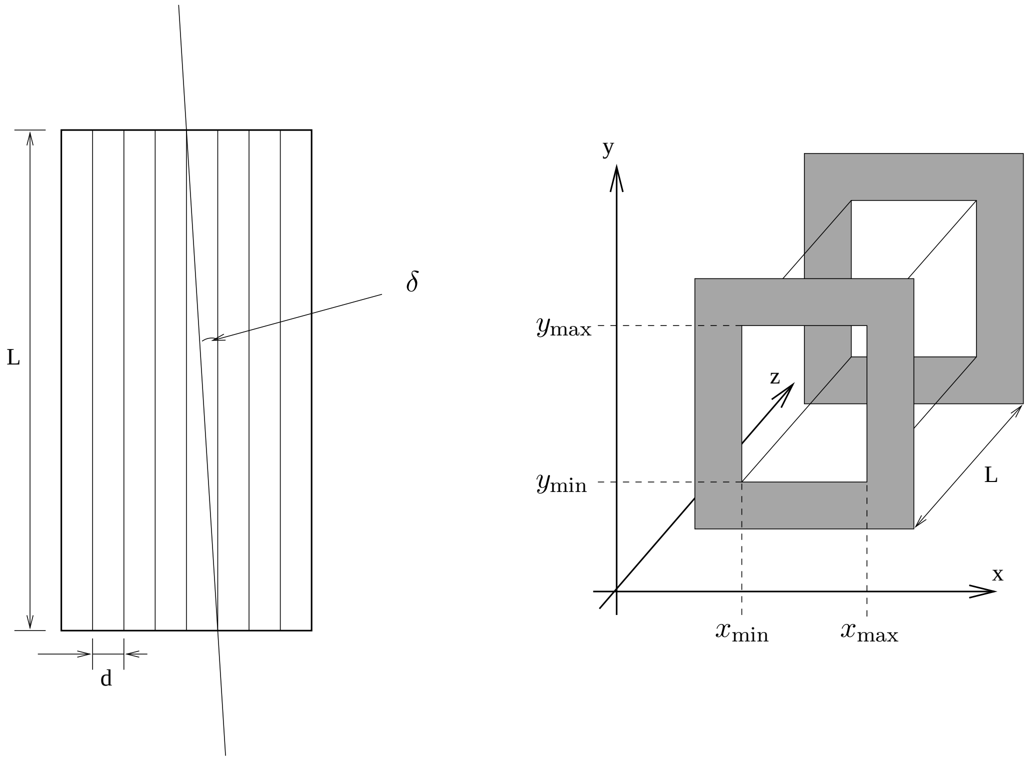


Figure 5.1: The geometry of a simple Soller blade collimators: The real Soller collimator, seen from the top (left), and a sketch of the component **Soller** (right). The symbols are defined in the text.

as explained below. For a more detailed Soller collimator simulation the Channeled_guide component can be employed, see section 5.3.4.

Let the collimation angle be $\delta = \tan^{-1}(d/L)$, where L is the length of the collimator and d is the distance between the blades, and let ϕ be the divergence angle between the neutron path and a vertical plane along the collimator axis, see Fig. 5.1. Neutrons with a large divergence angle $|\phi| \geq \delta$ will always hit at least one collimator blade and will thus be absorbed. For smaller divergence angles, $|\phi| < \delta$, the fate of the neutron depends on its exact entry point. Assuming that a typical collimator has many blades, the absolute position of each blade perpendicular to the collimator axis is somewhat uncertain (and also unimportant). A simple statistical consideration now shows that the transmission probability is $T = 1 - \tan|\phi|/\tan\delta$.

We simulate the collimator by transmitting all neutrons with $|\phi| < \delta$, but adjusting their weight with the amount

$$\pi_i = T = 1 - \tan|\phi|/\tan\delta, \qquad (5.7)$$

while all others are discarded by the kernel call ABSORB.

The input parameters for **Soller** are the coordinates $(x_{\min}, x_{\max}, y_{\min}, y_{\max})$, defining the identical entry and exit apertures, the length, l, between the slits, and the collimator divergence δ. If $\delta = 0$, the collimating effect is disabled, so that $\pi_i = 1$ whenever the neutron clears the two apertures.

5.2.7 Filter: A transmission filter

A neutron transmission filter act in much of the same way as two identical slits, one after the other. The only difference is that the transmission of the filter varies with the neutron energy.

In the simple component **Filter**, we have not tried to simulate the details of the transmission process (which includes absorption, incoherent scattering, and Bragg scattering in a polycrystalline sample, *e.g.* Be). Instead, the transmission is parametrised to be $\pi_i = T_0$ when $E \le E_{\rm min}$, $\pi_i = T_1$ when $E \ge E_{\rm max}$, and linearly interpolated between the two values in the intermediate interval.

$$\pi_i = \begin{cases} T_0 & E \le E_{\rm min} \\ T_1 + (T_0 - T_1)\frac{E_{\rm max}-E}{E_{\rm max}-E_{\rm min}} & E_{\rm min} < E < E_{\rm max} \\ T_1 & E \ge E_{\rm max} \end{cases} \tag{5.8}$$

If $T_1 = 0$, the neutrons with $E > E_{\rm max}$ are ABSORB'ed.

The input parameters are the four slit coordinates, $(x_{\rm min}, x_{\rm max}, y_{\rm min}, y_{\rm max})$, the distance, l, between the slits, and the transmission parameters T_0, T_1, $E_{\rm min}$, and $E_{\rm max}$. The energies are given in meV.

5.3 Advanced optical components: mirrors and guides

This section describes advanced neutron optical components such as supermirrors and guides. The first subsection, however, contains only a description of the reflectivity of a supermirror.

5.3.1 Mirror reflectivity

To compute the reflectivity of the supermirrors, we use an empirical formula derived from experimental data (see figure 5.2). The reflectivity is given by the following formula

$$R = \begin{cases} R_0 & \text{if } Q \le Q_{\rm c} \\ \frac{1}{2}R_0(1 - \tanh[(Q - mQ_{\rm c})/W])(1 - \alpha(Q - Q_{\rm c})) & \text{if } Q > Q_{\rm c} \end{cases} \tag{5.9}$$

Here Q is the length of the scattering vector (in Å^{-1}) defined by

$$Q = |\mathbf{k_i} - \mathbf{k_f}| = \frac{m_{\rm n}}{\hbar}|\mathbf{v_i} - \mathbf{v_f}|, \tag{5.10}$$

$m_{\rm n}$ being the neutron mass. The value m is a parameter determined by the mirror materials, the bilayer sequence, and the number of bilayers. As can be seen, $R = R_0$ for $Q < Q_{\rm c}$, which is the critical scattering wave vector for a single layer of the mirror material. At higher values of Q, the reflectivity starts falling linearly with a slope α until a cut-off at $Q = mQ_{\rm c}$. The width of the cut-off is denoted W. For the curve in figure 5.2, the values are

$$m = 4 \qquad R_0 = 1 \qquad Q_{\rm c} = 0.02\ \text{Å}^{-1} \qquad \alpha = 6.49\ \text{Å} \qquad W = 1/300\ \text{Å}^{-1}$$

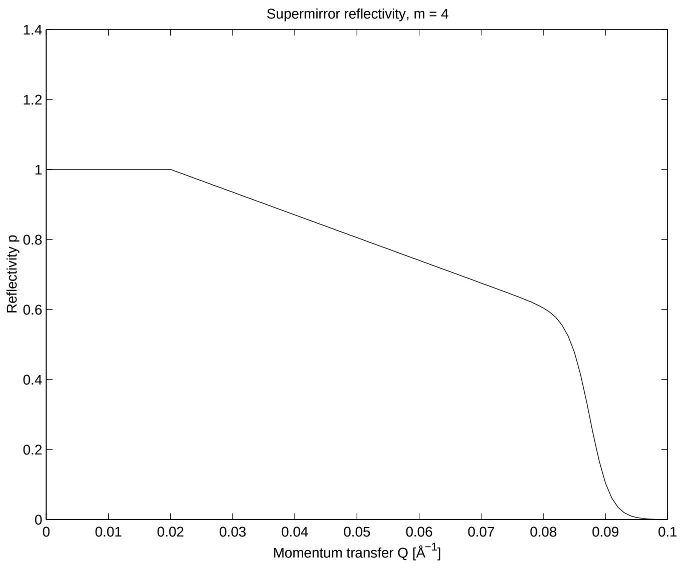


Figure 5.2: A typical reflectivity curve for a supermirror, Eq. (5.10).

As a special case, if $m = 0$ then the reflectivity is zero for all Q, *ie.* the surface is completely absorbing.

In the components, the neutron weight is adjusted with the amount $\pi_i = R$. To avoid spending large amounts of computation time on very low-weight neutrons, neutrons for which the reflectivity is lower than about 10^{-10} are ABSORB'ed.

5.3.2 Mirror: The single mirror

The component **Mirror** models a single rectangular neutron mirror plate. It can be used to *e.g.* assemble a complete neutron guide by putting multiple mirror components at appropriate locations and orientations in the instrument definition, much like a real guide is build from individual mirrors.

The mirror is assumed to lie in the first quadrant of the x-y plane, with one corner at $(0, 0, 0)$. If the neutron trajectory intersects the mirror plate, it is reflected, otherwise it is left untouched. Since the mirror lies in the x-y plane, an incoming neutron with velocity $\mathbf{v}_\mathrm{i} = (v_x, v_y, v_z)$ is reflected with velocity $\mathbf{v}_\mathrm{f} = (v_x, v_y, -v_z)$. The computation of the reflectivity is handled as detailed in section 5.3.1.

The input parameters of this component are the rectangular mirror dimensions (l, h) and the values of R_0, m, Q_c, W, and α for the mirror.

5.3.3 Guide: The guide section

The component **Guide** models a guide tube consisting of four flat mirrors. The guide is centered on the z axis with rectangular entrance and exit openings parallel to the x-y plane. The entrance has the dimensions (w_1, h_1) and placed at $z = 0$. The exit is of dimensions (w_2, h_2) and is placed at $z = l$ where l is the guide length. See figure 5.3.

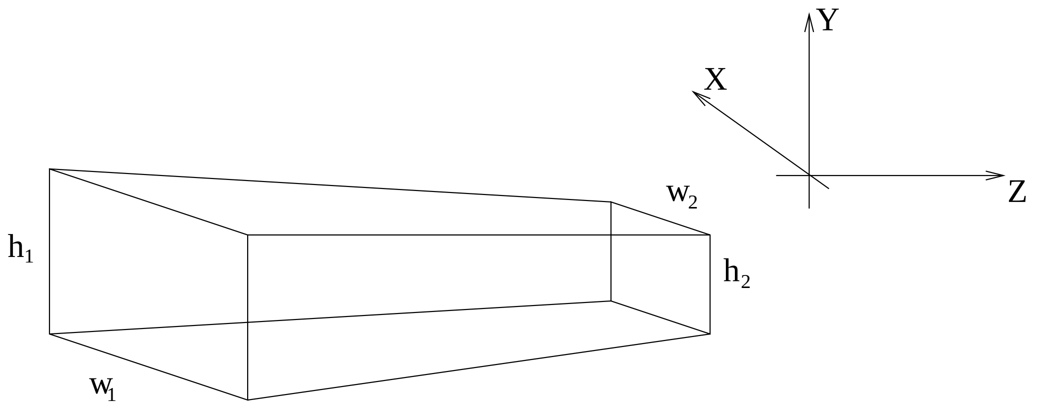


Figure 5.3: The geometry used for the guide component.

Neutrons not clearing the guide entrance are ABSORB'ed. For a more general guide simulation, see the Channeled_guide component in section 5.3.4.

For computations on the guide geometry, we define the planes of the four guide sides by giving their normal vectors (pointing into the guide) and a point lying in the plane:

$$\begin{array}{rclrcl}
\mathbf{n}_1^v &=& (l, 0, (w_2 - w_1)/2) & \mathbf{O}_1^v &=& (-w_1/2, 0, 0) \\
\mathbf{n}_2^v &=& (-l, 0, (w_2 - w_1)/2) & \mathbf{O}_2^v &=& (w_1/2, 0, 0) \\
\mathbf{n}_1^h &=& (0, l, (h_2 - h_1)/2) & \mathbf{O}_1^h &=& (0, -h_1/2, 0) \\
\mathbf{n}_2^h &=& (0, -l, (h_2 - h_1)/2) & \mathbf{O}_2^h &=& (0, h_1/2, 0)
\end{array}$$

In the following, we refer to an arbitrary guide side by its origin **O** and normal **n**.

With these definitions, the time of intersection of the neutron with a guide side can be computed by considering the projection onto the normal:

$$t_1 = \frac{(\mathbf{O} - \mathbf{r}_0) \cdot \mathbf{n}}{\mathbf{v} \cdot \mathbf{n}} \tag{5.11}$$

For a neutron that leaves the guide through the guide exit we have

$$t_2 = \frac{l - z_0}{v_z} \tag{5.12}$$

To compute the interaction of the neutron with the guide, the neutron is initially propagated to the $z = 0$ plane of the guide entrance. If it misses the entrance, it is ABSORB'ed. Otherwise, we repeatedly compute the time of intersection with the four mirror sides and the guide exit. The smallest positive t thus found gives the time of the next intersection with the guide (or in the case of the guide exit, the time when the neutron leaves the guide). The neutron is propagated to this point, the reflection from the side is computed and the process is repeated until the neutron leaves the guide.

The reflected velocity $\mathbf{v}_\mathrm{f}$ of the neutron with incoming velocity $\mathbf{v}_\mathrm{i}$ is computed by the formula

$$\mathbf{v}_\mathrm{f} = \mathbf{v}_\mathrm{i} - 2\frac{\mathbf{n} \cdot \mathbf{v}_\mathrm{i}}{|\mathbf{n}|^2}\mathbf{n} \tag{5.13}$$

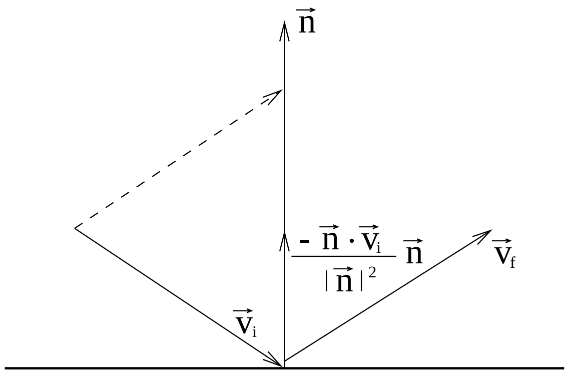


Figure 5.4: Neutron reflecting from mirror. $\mathbf{v}_\mathrm{i}$ and $\mathbf{v}_\mathrm{f}$ are the initial and final velocities, respectively, and $\mathbf{n}$ is a vector normal to the mirror surface.

This expression is arrived at by again considering the projection onto the mirror normal (see figure 5.4). The reflectivity of the mirror is taken into account as explained in section 5.3.1.

There are a few optimizations possible here to avoid redundant computations. Since the neutron is always inside the guide during the computations, we always have $(\mathbf{O}-\mathbf{r}_0)\cdot\mathbf{n} \leq 0$. Thus $t \leq 0$ if $\mathbf{v}\cdot\mathbf{n} \geq 0$, so in this case there is no need to actually compute t. Some redundant computations are also avoided by utilizing symmetry and the fact that many components of $\mathbf{n}$ and $\mathbf{O}$ are zero.

The input parameters of this component are the opening sizes of the entry and exit point of the guide, (w_1, h_1) and (w_2, h_2), respectively, the guide length, l, and the values of R_0, m, Q_c, W, and α for the mirror.

5.3.4 Channeled_guide: A guide section component with multiple channels

The component Channeled_guide is a more flexible variation of the Guide component described in the previous section. It allows the specification of different supermirror parameters for the horizontal and vertical mirrors, and also implements guides with multiple channels as used in neutron bender devices. By setting the m value of the supermirror coatings to zero, nonreflecting walls are simulated; this may be used to simulate a Soller collimator.

The channel walls are assumed to be infinitely absorbing. The implementation is basen on that of the Guide component. Initially, the channel which the neutron will enter is computed. The x coordinate is then shifted so that the channel can be simulated as a single instance of the Guide component. Finally the coordinates are restored when the neutron exits the guide or is absorbed.

The input parameters are *w1*, *h1*, *w2*, *h2*, and l to set the guide dimensions in meters as for the Guide component (entry window, exit window, and length); k to set the number of channels; d to set the thickness of the channel walls, in meters; and *R0*, *W*, *Qcx*, *Qcy*, *alphax*, *alphay*, *mx*, and *my* to set the supermirror parameters as described above (the names with x denote the vertical mirrors, and those with y denote the horizontal ones).

5.4 Chopper-like components

In this section, we will present rotating components such as chopppers, velocity selectors, *etc.*

5.4.1 V_selector: The rotating velocity selector

The component **V_selector** models a rotating velocity selector constructed from a number of collimator blades arranged radially on an axis. Two identical slits at a 12 o'clock position allow neutron passage at the position of the blades. The blades are "twisted" on the axis so that a stationary velocity selector does not transmit any neutrons; the total twist angle is denoted ϕ. By rotating the selector you allow transmittance of neutrons around certain velocities, given by

$$V_0 = \omega L/\phi, \tag{5.14}$$

which means that the selector has turned the twist angle ϕ during the neutron flight time L/V_0.

Neutrons having a velocity slightly smaller or larger than V_0 will either be transmitted or absorbed depending on the exact position of the rotator blades when the neutron enters the selector. Assuming this position to be unknown (and assuming infinitely thin blades), we arrive at

$$T = \begin{cases} 1-(N/2\pi)|\phi-\omega L/V| & \text{if } -1<(N/2\pi)(\phi-\omega L/V)<1 \\ 0 & \text{otherwise} \end{cases} \tag{5.15}$$

where N is the number of collimator blades.

A horisontal divergence changes the above formula because of the angular difference between the entry and exit points of the neutron. The resulting transmittance resembles the one above, only with V replaced by V_z and ϕ replaced by $(\phi+\psi)$, where ψ is the angular difference due to the divergence. An additional vertical divergence does not change this formula, but it may contribute to ψ. (We have here ignored the very small non-linearity of ψ along the neutron path in case of both vertical and horisontal divergence).

Adding the effect of a finite blade thickness, t, reduces the transmission by the overall amount

$$dT = (Nt)/(2\pi r), \tag{5.16}$$

where r is the distance from the rotation axis. We ignore the variation of r along the neutron path and use just the average value.

The input parameters for V_selector are the slit dimensions, *width*, *height* (in m), the distance between apertures, L_0 (in m), the length of the collimator blades, L_1 (in m), the height from rotation axix to the slit centre, r_0 (in m), the rotation speed ω (in rpm) the twist angle ϕ (in degrees), the blade thickness t (in m), and the number of blades, N.

The local coordinate system is centered at the slit centre.

5.4.2 Chopper: The disc chopper

This component was contributed by Philipp Bernhardt, Lehrstuhl für Kristallographie und Strukturphysik.

To cut a continuous neutron beam into short pulses, you can use a disc chopper (figure 5.5). This is a fast rotating disc with the rotating axis parallel to the neutron beam. The disk consists of neutron absorbing materials. To form the pulses there are slits through which the neutrons can pass.

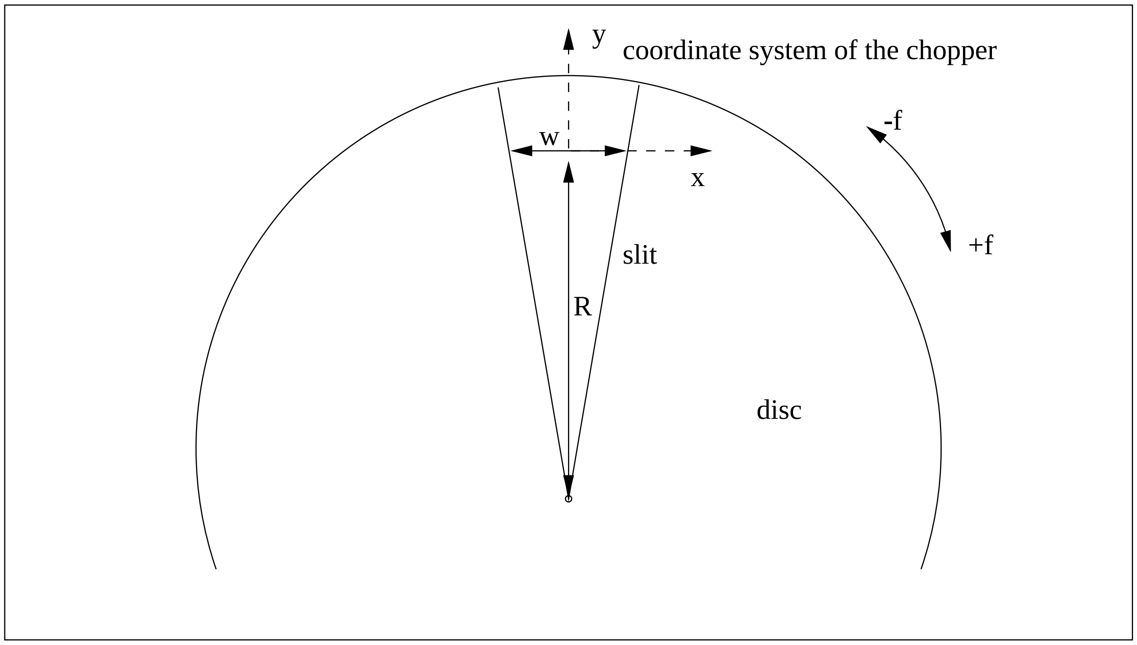


Figure 5.5: disc chopper

This component simulates choppers with more than one slit. The slits are symmetrically disposed on the disc. You can set the direction of rotation, which allows to simulate double choppers. You can also define the phase by setting the time at which one slit is positioned at the top. The sides of the slits are pointing towards the center of the disc. The thickness of the disc is neglected. There is no parameter for the height of the slits, so if you like to limit the neutrons in the y-direction, just use a slit component in front of the chopper.

If you use a rectangular shaped beam and the beam has nearly the same size as the slit, you will get an almost triangular shape of the transmission curve (figure 5.6).

The input parameters for this component are the width w of the slit at the radius R of the disc, the phase pha, the number of slits n and the angular frequency f. The sign of f defines the direction of rotation, as can be seen in figure 5.5.

5.4.3 First_chopper: The first disc chopper

This component was contributed by Philipp Bernhardt, Lehrstuhl für Kristallographie und Strukturphysik.

The disadvantage of the component 'Chopper' is the bad statistic, because most of the neutrons of a continuous beam are absorbed. Furthermore TOF-instruments define the starting time of the neutrons at the position of the first chopper and not at the source. Therefore this component is useful. This 'first disc chopper' has the same geometrical and

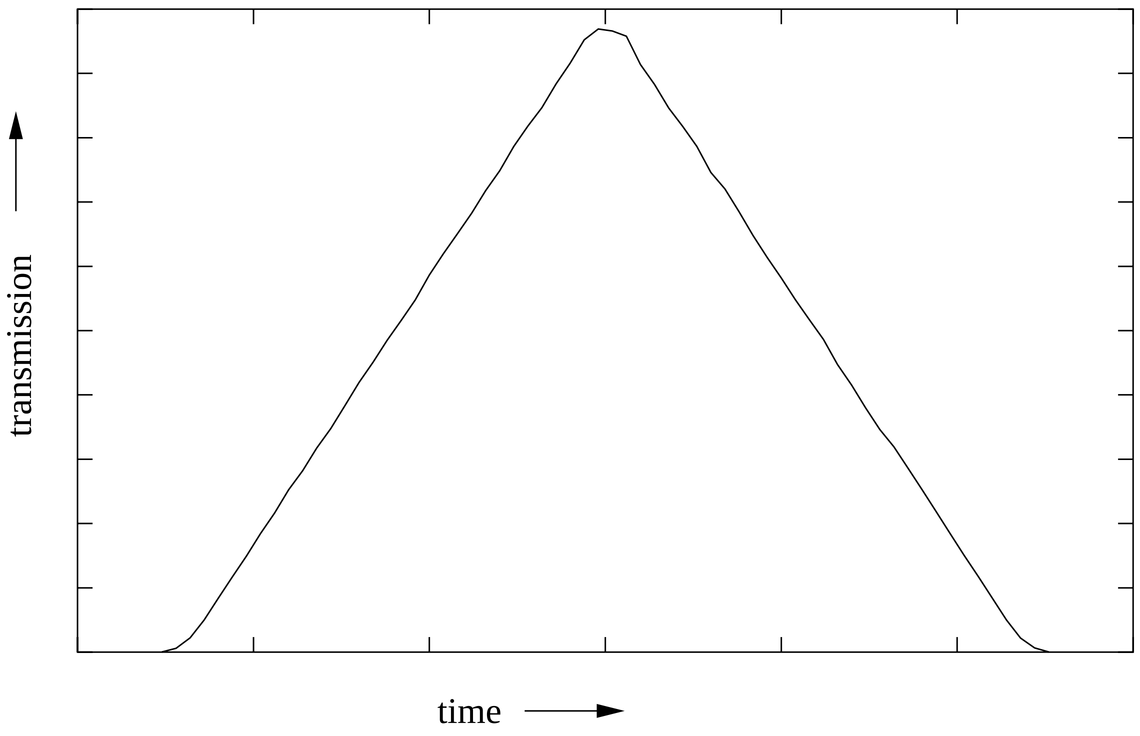


Figure 5.6: example transmission curve for the disc chopper

physical attributes as the normal disc chopper before. But it does not check if the neutron can pass the disc chopper, it instead gives the neutron a time at which it is possible to pass. There is no absorption in this component, all neutrons will be used.

Because the value t of the incoming neutron will be overwritten, this chopper can only be used as a first chopper.

The input parameters are, again, the width w of the slit at the radius R of the disc, the phase pha of the chopper, the number of slits n and the angular frequency f (sign defines direction of rotation). With the additional parameter a you can set the number of pulses. This is useful if you want to investigate frame overlaps.

5.5 Detectors and monitors

In real neutron experiments, detectors and monitors play quite different roles. One wants the detectors to be as efficient as possible, counting all neutrons (and absorbing them in the process), while the monitors measure the intensity of the incoming beam, and must as such be almost transparent, interacting only with (roughly) 0.1-1% of the neutrons passing by. In computer simulations, it is of course possible to detect every neutron without absorbing it or disturbing any of its parameters. Hence, the two components have very similar functions in the simulations, and we do not distinguish between them. For simplicity, they are from here on just called monitors, since they do not absorb the neutron.

Another difference between computer simulations and real experiments is that one may allow the monitor to be sensitive to any neutron property, as *e.g.* direction, energy, and polarization, in addition to what is found in advanced existing monitors (space and time). One may, in fact, let the monitor have several of these properties at the same time, as seen for example in the energy sensitive monitor in section 5.5.7.

5.5.1 Monitor: The single monitor

The component **Monitor** consists of a rectangular opening — like that for **slit**. The neutron is propagated to the plane of the monitor by the kernel call PROP_Z0. Any neutron that passes within the opening is counted — the number counting variable is incremented: $N_i = N_{i-1}+1$, the neutron weight p_i is added to the weight counting variable: $I_i = I_{i-1} + p_i$, and the second moment of the weight is updated: $M_{2,i} = M_{2,i-1} + p_i^2$. The input parameters for **Monitor** are the opening coordinates $x_{\rm min}, x_{\rm max}, y_{\rm min}, y_{\rm max}$, and the output parameters are the three count numbers, N, I, and M_2.

5.5.2 Monitor_4PI: The 4π monitor

The component **Monitor_4PI** does not model any physical monitor but may be thought of as a spherical monitor completely surrounding the previous component. It simply detects all neutrons that have not been absorbed at the position in the instrument in which it is placed. If this monitor is placed in the instrument file after another component, *e.g.* a sample, it will count any neutron scattered from this component. This may be useful during tests.

The output parameters for **Monitor_4PI** are the three count numbers, N, I, and M_2.

5.5.3 PSD_monitor: The PSD monitor

The component **PSD_monitor** closely resembles **Monitor**. In the PSD monitor, though, the rectangular monitor window is divided into $n \times m$ pixels, each of which acts like a single monitor.

The input parameters for **PSDmonitor** are the opening coordinates $x_{\rm min}, x_{\rm max}, y_{\rm min}$, $y_{\rm max}$, the array dimensions (n, m), and a name of a file in which to store $I(x, y)$. The output parameters are three two-dimensional arrays of counts: $N(x, y), I(x, y), M_2(x, y)$.

5.5.4 PSD_monitor_4PI: The 4π PSD monitor

The component **PSD_monitor_4PI** represents a PSD monitor shaped as a sphere, much like **Monitor_4PI**. It subdivides the surface of the sphere into pixels of equal area (using a projection onto a cylinder with an axis which is vertical in the local coordinate system) and distributes the incoming neutron counts into the respective pixels.

The 4π PSD monitor is typically placed around another component. Used in this way, the 4π PSD monitor is very useful for debugging components.

The input parameters for **PSD_monitor_4PI** are the monitor radius, the number of pixels, (n_x, n_y) – where y is the vertical direction, and the name of the file in which to store $I(x, y)$. The output parameters of the component are the three count arrays $N(x, y), I(x, y)$, and $M_2(x, y)$.

5.5.5 PSD_monitor_4PI_log: The 4π PSD monitor with log scale

The component **PSD_monitor_4PI_log** is the same as PSD_monitor_4PI described in the previous section, except that the output histograms contain the base-10 logarithm of the intensities rather than the intensities themselves. Currently, this does not work well together with the McStas mechanism to output detector results (see section 3.4.5), so the total intensity as output by McStas from this detector component will be wrong. However, the component was sufficiently useful in Laue-type diffraction instruments to be included here nevertheless. A future version of McStas may implement a better way to get log-scale in output files.

The input parameters for PSD_monitor_4PI_log are the same as for PSD_monitor_4PI.

5.5.6 TOF_monitor: The time-of-flight monitor

TOF_monitor is a rectangular single monitor which is sensitive to the absolute time, where the neutron is hits the component. Like in a real time-of-flight detector, the time dimension is binned into small time intervals of length *dt*, whence this monitor updates a one-dimensional array of counts.

The input parameters for **TOF_monitor** are the opening coordinates $x_{\rm min}, x_{\rm max}, y_{\rm min}$, $y_{\rm max}$, the number of time bins (beginning from $t = 0$), $n_{\rm chan}$, the time spacing between bins, *dt* (in μs), and the name of the output file. Output parameters of the component are the three count arrays $N(i), I(i)$, and $M_2(i)$, where i is the bin number.

5.5.7 E_monitor: The energy sensitive monitor

The component **E_monitor** resembles **TOF_monitor** to a very large extent. Only this monitor is sensitive to the neutron energy, which in binned in *nchan* bins between $E_{\rm min}$ and $E_{\rm max}$.

The input parameters for **E_monitor** are the opening coordinates $x_{\rm min}, x_{\rm max}, y_{\rm min}$, $y_{\rm max}$, the total energy interval given by $E_{\rm min}$ and $E_{\rm max}$ (in meV), and *nchan* and the name of the output file. Output parameters of the component are the three count arrays $N(i), I(i)$, and $M_2(i)$, i being the bin number.

5.5.8 L_monitor: The wavelength sensitive monitor

The component **L_monitor** is a rectangular monitor with an opening in the *x*-*y* plane which is sensitive to the neutron wavelength. The wavelength spectrum is output in a one-dimensional histogram. Only neutrons with wavelength $\lambda_0 < \lambda < \lambda_1$ are detected.

The input parameters for **L_monitor** are the opening coordinates *xmin*, *xmax*, *ymin*, and *ymax* defining the edges of the slit in meters; the lower and upper wavelength limit *lambda_0* and *lambda_1* in Ångstrøm; the number of histogram bins *nchan*; and *filename*, a string giving the name of the file to store the data in.

5.5.9 Divergence_monitor: The divergence sensitive monitor

The component **Divergence_monitor** is a rectangular monitor with an opening in the *x*-*y* plane, which is sensitive to the neutron divergence, *i.e.* the angle between the neutron path and the monitor surface normal.

The divergence is divided into horisontal and vertical divergencies, which are calculated as $\delta_h = \tan^{-1}(v_x/v_z)$ and $\delta_v = \tan^{-1}(v_y/v_z)$, respectively. Only neutrons within a divergence window of $\delta_h = (-\delta_{\rm h,max}; \delta_{\rm h,max})$, $\delta_v = (-\delta_{\rm v,max}; \delta_{\rm v,max})$ are detected. The counts are binned in an array of $n_h \times n_v$ pixels.

The input parameters for the Divergence_monitor component are the opening coordinates $(x_{\rm min}, x_{\rm max}, y_{\rm min}, y_{\rm max})$, the number of pixels (n_h, n_v), the parameters $(\delta_{\rm h,max}, \delta_{\rm v,max})$ defining the divergence interval, and a name of the file in which to store the detected intensities.

Note that a divergence sensitive monitor with a small opening may be thought of as a non-reversing pinhole camera.

5.5.10 DivPos_monitor: The divergence-position sensitive monitor

The component **DivPos_monitor** is a rectangular monitor with an opening in the *x*-*y* plane, which is sensitive to both the horizontal neutron divergence and the horizontal neutron position. The neutron intensity as a function of position and divergence is output in a two-dimensional histogram. This output may be directly compared to an acceptance diagram, an analytical technique that is sometimes used to calculate neutron guide performances.

The horizontal divergence is calculated as $\delta_h = \tan^{-1}(v_x/v_z)$. Only neutrons within a divergence window of $\delta_h = (-\delta_{\rm h,max}; \delta_{\rm h,max})$ are detected.

The input parameters for the DivPos_monitor component are the opening coordinates *xmin*, *xmax*, *ymin*, and *ymax* in meters; the number of histogram bins *npos* and *ndiv* in position and divergence; the maximum divergence *maxdiv* to detect, in degrees; and *filename*, a string giving the name of the file to store the data in.

5.5.11 DivLambda_monitor: The divergence-wavelength sensitive monitor

The component **DivLambda_monitor** is a rectangular monitor with an opening in the *x*-*y* plane, which is sensitive to both the horizontal neutron divergence and the wavelength. The neutron intensity as a function of wavelength and divergence is output in a two-dimensional histogram.

The horizontal divergence is calculated as $\delta_h = \tan^{-1}(v_x/v_z)$. Only neutrons within a divergence window of $\delta_h = (-\delta_{\rm h,max}; \delta_{\rm h,max})$ and with wavelength $\lambda_0 < \lambda < \lambda_1$ are detected.

The input parameters for the DivLambda_monitor component are the opening coordinates *xmin*, *xmax*, *ymin*, and *ymax* in meters; the number of histogram bins *nlam* and *ndiv* in wavelength and divergence; the maximum divergence *maxdiv* to detect, in degrees; *lambda_0* and *lambda_1* to define the wavelength window, in Ångstrøm; and *filename*, a string giving the name of the file to store the data in.

5.5.12 Res_monitor: The resolution monitor

The component **Res_monitor** is used together with the **Res_sample** component (described in section 5.8.1) and the `mcresplot` front-end (described in section 2.6.6). It works like a normal single detector, but also records all scattering events in the resolution sample and writes them to a file that can later be read by `mcresplot`.

The instrument definition should contain an instance of the **Res_sample** component, the name of which should be passed as an input parameter to **Res_monitor**. For example

```
COMPONENT mysample = Res_sample( ... )
...
COMPONENT det = Res_monitor(res_sample_comp = mysample, ...)
...
```

The output file is in ASCII format, one line per scattering event, with the following columns:

- $\mathbf{k}_\mathrm{i}$, the three components of the initial wave vector.
- $\mathbf{k}_\mathrm{f}$, the three components of the final wave vector.
- $\mathbf{r}$, the three components of the position of the scattering event in the sample.
- p_i, the neutron weight just after the scattering event.
- p_f, the relative neutron weight adjustment from sample to detector (so the total weight in the detector is $p_\mathrm{i}p_\mathrm{f}$).

From $\mathbf{k}_\mathrm{i}$ and $\mathbf{k}_\mathrm{f}$, we may compute $\mathbf{Q} = \mathbf{k}_\mathrm{i} - \mathbf{k}_\mathrm{f}$ and $\omega = (2.072\ \mathrm{meV}\cdot\mathrm{\AA}^2)(\mathbf{k}_\mathrm{i}^2 - \mathbf{k}_\mathrm{f}^2)$.

The vectors are given in the local coordinate system of the resolution sample component. The wave vectors are in units of $\mathrm{\AA}^{-1}$, the scattering position in units of meters.

The input parameters for **Res_monitor** are the opening coordinates $x_\mathrm{min}, x_\mathrm{max}, y_\mathrm{min}, y_\mathrm{max}$ as for the single monitor component, the name of the file to write in *filename*, and *res_sample_comp* which should be set to the name of the resolution sample component used in the instrument. The output parameters are the three count numbers, *Nsum*, *psum*, and *p2sum*, and the handle *file* of the output file.

5.5.13 Adapt_check: The simple adaptive importance sampling monitor

The component **Adapt_check** is used together with the Source_adapt component — see section 5.1.6 for details. When placed somewhere in an instrument using Source_adapt, the source will optimize for neutrons that reach that point without being absorbed (regardless of neutron position, direction, wavelength, *etc*).

The Adapt_check component takes a single input parameter *source_comp*. This should be set to the name given to the Source_adapt component in the instrument, for example

```
...
COMPONENT mysource = Source_adapt( ... )
...
COMPONENT mycheck = Adapt_check(source_comp = mysource)
...
```

5.6 Bragg scattering single crystals, monochromators

In this class of components, we are concerned with elastic Bragg scattering from single crystals. The Mosaic_anisotropic component is a replacement for the Monochromator component from previous releases; it uses a better algorithm that works in some cases where the old component would give wrong results. The Mosaic_simple component is similar, but has an isotropic mosaic and allows a scattering vector that is not perpendicular to the surface. The Single_crystal component is a general single crystal sample that allows the input of an arbitrary unit cell and a list of structure factors, and also allows a $\Delta d/d$ lattice space variation.

Thus in this version of McStas, *either* non-perpendicular scattering vectors (in Mosaic_simple and Single_crystal) *or* anisotropic mosaic (in Mosaic_anisotropic), but not both, is allowed. The reason for this is not the difficulty of simulating both together, but rather the difficulty of finding a good way for the user to specify the mosaic of a crystal that works for scattering vectors in all directions. Suggestions for this are much welcomed by the authors!

5.6.1 Mosaic_simple: An infinitely thin mosaic crystal with a single scattering vector

The component **Mosaic_simple** simulates an infinitely thin single crystal with a single scattering vector and a mosaic spread. A typical use for this component is to simulate a monocromator or an analyzer.

The physical model used in the component is a rectangular piece of material composed of a large number of small micro-crystals. The orientation of the microcrystals deviates from the nominal crystal orientation so that the probability of a given microcrystal orientation is proportional to a gaussian of the angle between the given and the nominal orientation. The width of the gaussian is given by the mosaic spread of the crystal. The mosaic spread is assumed to be large compared to the Bragg width of the scattering vector.

As a further simplification, the crystal is assumed to be infinitely thin. This means that multiple scattering effects are not simulated. It also means that the total reflectivity can be used as a parameter for the model rather than the atomic scattering cross section.

When a neutron trajectory intersects the crystal, the first step in the computation is to determine the probability of scatttering. This probability is then used in a Monte Carlo choice desiding whether to scatter or transmit the neutron. The scattering probability is the sum of the probabilities of first-order scattering, second-order, ..., up to the highest order that permits Bragg scattering at the given neutron wave length. However, in most cases at most one order will have a significant scattering probability, and the computation thus considers only the order that best matches the neutron wavelength. Bragg's law is

$$n\mathbf{Q}_0 = 2\mathbf{k}_i \sin\theta$$

Thus, the scattering order is obtained simply as the integer multiple n of the nominal scattering vector $\mathbf{Q}_0$ which is closest to the projection of $2\mathbf{k}_i$ onto $\mathbf{Q}_0$ (see figure 5.7). Once n has been determined, the Bragg angle θ can be computed. The angle d that the nominal scattering vector $\mathbf{Q}_0$ makes with the closest scattering vector $\mathbf{q}$ that admits Bragg

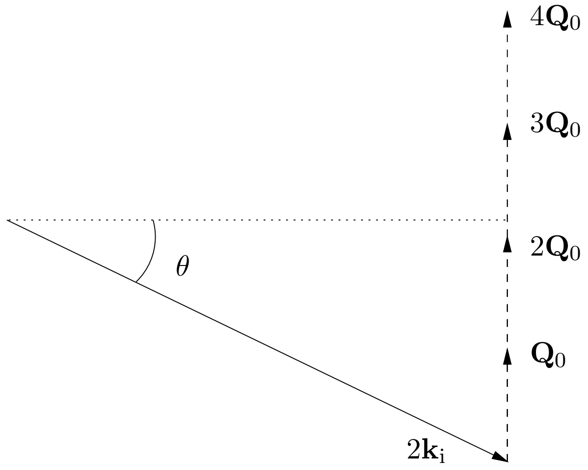


Figure 5.7: Selection of the Bragg order ("2" in this case).

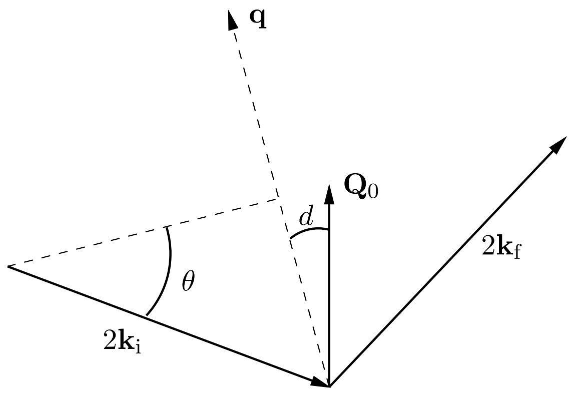


Figure 5.8: Computing the deviation d from the nominal scattering direction.

scattering is then used to compute the probability of reflection from the mosaic

$$p_{\mathrm{reflect}} = R_0 e^{-d^2/2\sigma^2},$$

where R_0 is the reflectivity at the Bragg angle (see figure 5.8). The probability p_{reflect} is used in a Monte Carlo choice to decide whether the neutron is transmitted or reflected.

In the case of reflection, the neutron will be scattered into the Debye-Scherrer cone, with the probability of each point on the cone being determined by the mosaic. The Debye-Scherrer cone can be described by the equation

$$\mathbf{k}_\mathrm{f} = \mathbf{k}_\mathrm{i} \cos 2\theta + \sin 2\theta (\mathbf{c} \cos\varphi + \mathbf{b} \sin\varphi), \qquad \varphi \in [-\pi; \pi], \tag{5.17}$$

where $\mathbf{b}$ is a vector perpendicular to $\mathbf{k}_\mathrm{i}$ and $\mathbf{Q}_0$, $\mathbf{c}$ is perpendicular to $\mathbf{k}_\mathrm{i}$ and $\mathbf{b}$, and both $\mathbf{b}$ and $\mathbf{c}$ have the same length as $\mathbf{k}_\mathrm{i}$ (see figure 5.9). In the component, φ is sampled from a gaussian distribution with twice the mosaic width, since the scattering angle is twice the Bragg angle.

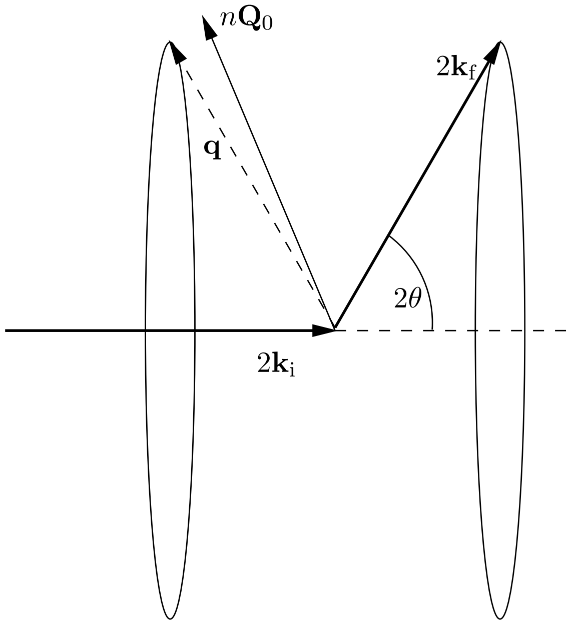


Figure 5.9: Scattering into the part of the Debye-Scherrer cone covered by the mosaic.

What remains is to get the neutron weight right. The distribution from which the scattering event is sampled is a gaussian in φ of width 2σ,

$$f_{\mathrm{MC}}(\varphi) = \frac{1}{\sqrt{2\pi}2\sigma} e^{-\varphi^2/2(2\sigma)^2}$$

In the physical model, the probability of the scattering event is proportional to a gaussian in the angle between the nominal scattering vector $\mathbf{Q}_0$ and the actual scattering vector $\mathbf{q}$. The normalisation condition is that the integral over all φ should be 1. Thus the probability of the scattering event in the physical model is

$$\Pi(\varphi) = e^{\frac{-d(\varphi)^2}{2\sigma^2}} \Big/ \int_{-\pi}^{\pi} e^{\frac{-d(\varphi)^2}{2\sigma^2}} d\varphi \tag{5.18}$$

where $d(\varphi)$ denotes the angle between the nominal scattering vector and the actual scattering vector corresponding to φ. According to equation (4.8), the weight adjustment π_j is then given by

$$\pi_j = \Pi(\varphi)/f_{\mathrm{MC}}(\varphi).$$

In the component, the integral in (5.18) is computed using a 15-order Gaussian quadrature formula, with the integral restricted to an inteval 5 times wider than the mosaic width σ.

The input parameters for Mosaic_simple are *zmin*, *zmax*, *ymin*, and *ymax* to define the surface of the crystal in the Y-Z plane; *mosaic* to give the FWHM of the mosaic spread; *R0* to give the reflectivity at the Bragg angle, and *Qx*, *Qy*, and *Qz* to give the scattering vector.

5.6.2 Mosaic_anisotropic: The crystal with anisotropic mosaic

The component **Mosaic_anisotropic** is a modified version of the Mosaic_simple component, intended to replace the Monocromator component from previous releases. It restricts the scattering vector to be perpendicular to the crystal surface, but extends the Mosaic_simple component by allowing different mosaics in the horizontal and vertical direction.

The code is largely similar to that for Mosaic_simple, and the documentation for the latter should be consulted for details. The differences are mainly due to two reasons:

- Some simplifications have been done since two of the components of the scattering vector are known to be zero.
- The computation of the Gaussian for the mosaic is done done using different mosaics for the two axes.

The input parameters for the component Mosaic_anisotropic are *zmin*, *zmax*, *ymin*, and *ymax* to define the size of the crystal (in meters); *mosaich* and *mosaicv* to define the mosaic (in minutes of arc); *r0* to define the reflectivity (no unit); and Q to set the length of the scattering vector (in Å^{-1}).

5.6.3 Single_crystal: The single crystal component

The physical model

The textbook expression for the scattering cross-section of a crystal is [9]:

$$\left(\frac{d\sigma}{d\Omega}\right)_{\text{coh.el.}} = N\frac{(2\pi)^3}{V_0}\sum_{\boldsymbol{\tau}}\delta(\boldsymbol{\tau}-\boldsymbol{\kappa})|F_{\boldsymbol{\tau}}|^2$$

Here $|F_{\boldsymbol{\tau}}|^2$ is the structure factor, N is the number of unit cells, V_0 is the volume of an individual unit cell, and $\boldsymbol{\kappa} = \mathbf{k}_i - \mathbf{k}_f$ is the scattering vector. $\delta(\boldsymbol{x})$ is a 3-dimensional delta function in reciprocal space, so for given incoming wave vector $\mathbf{k}_i$ and lattice vector $\boldsymbol{\tau}$, only a single final wave vector $\mathbf{k}_f$ is allowed. In a real crystal, however, reflections are not perfectly sharp. Because of imperfection and finite-size effects, there will be a small region around $\boldsymbol{\tau}$ in reciprocal space of possible scattering vectors.

The Single_crystal component simulates a crystal with a mosaic spread η and a lattice plane spacing uncertainty $\Delta d/d$. In such crystals the reflections will not be completely sharp; there will be a small region around each reciprocal lattice point of the crystal that contains valid scattering vectors.

We model the mosaicity and $\Delta d/d$ of the crystal with 3-dimensional Gaussian functions in reciprocal space (see figure 5.10). Two of the axes of the Gaussian are perpendicular to the reciprocal lattice vector $\boldsymbol{\tau}$ and model the mosaicity. The third one is parallel to $\boldsymbol{\tau}$ and models $\Delta d/d$. We use an isotropic mosaicity, so the two axes perpendicular to $\boldsymbol{\tau}$ are of equal length η. We assume that the mosaicity is small so that the possible directions of the scattering vector may be approximated with a Gaussian in rectangular coordinates.

We now derive a quantitative expression for the scattering cross-section of the crystal in the model. For this, we introduce a *local coordinate system* for each reciprocal lattice

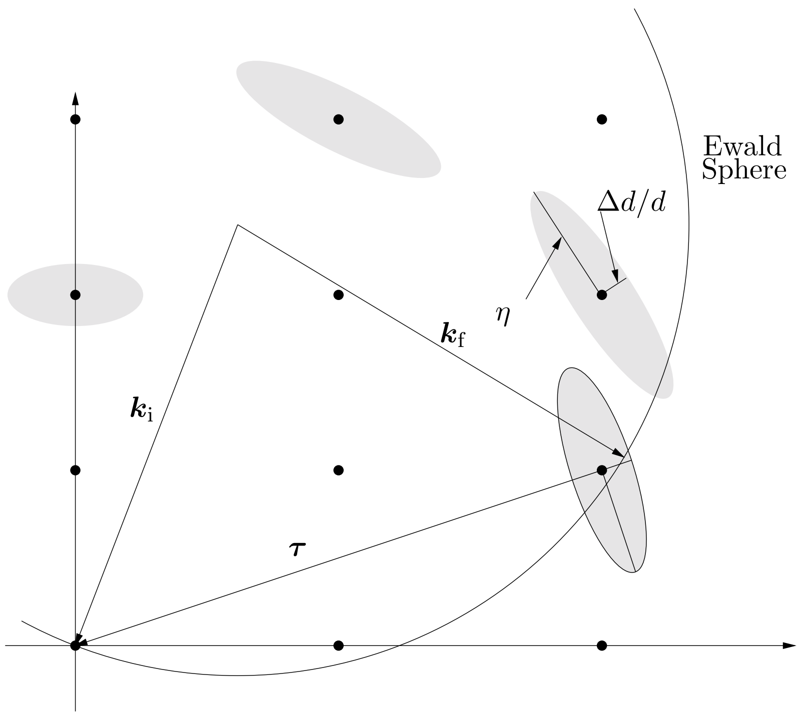


Figure 5.10: Ewald sphere construction for a single neutron showing the Gaussian broadening of reciprocal lattice points in their local coordinate system.

point $\boldsymbol{\tau}$ and use $\boldsymbol{x}$ for vectors written in local coordinates. The origin is $\boldsymbol{\tau}$, the first axis is parallel to $\boldsymbol{\tau}$ and the other two axes are perpendicular to $\boldsymbol{\tau}$. In the local coordinate system, the 3-dimensional Gaussian is given by

$$G(x_1, x_2, x_3) = \frac{1}{(\sqrt{2\pi})^3} \frac{1}{\sigma_1\sigma_2\sigma_3} e^{-\frac{1}{2}(\frac{x_1^2}{\sigma_1^2}+\frac{x_2^2}{\sigma_2^2}+\frac{x_3^2}{\sigma_3^2})} \tag{5.19}$$

The axes of the Gaussian are $\sigma_1 = \tau\Delta d/d$ and $\sigma_2 = \sigma_3 = \eta\tau$. Here we used the assumption that η is small, so that $\tan\eta \approx \eta$ (with η given in radians). By introducing the diagonal matrix

$$D = \begin{pmatrix} \frac{1}{2}\sigma_1^2 & 0 & 0 \\ 0 & \frac{1}{2}\sigma_2^2 & 0 \\ 0 & 0 & \frac{1}{2}\sigma_3^2 \end{pmatrix}$$

equation (5.19) can be written as

$$G(\boldsymbol{x}) = \frac{1}{(\sqrt{2\pi})^3} \frac{1}{\sigma_1\sigma_2\sigma_3} e^{-\boldsymbol{x}^{\rm T} D \boldsymbol{x}} \tag{5.20}$$

again with $\boldsymbol{x} = (x_1, x_2, x_3)$ written in local coordinates.

To get an expression in the coordinates of the reciprocal lattice of the crystal, we introduce a matrix U such that if $\boldsymbol{y} = (y_1, y_2, y_3)$ are the global coordinates of a point in the crystal reciprocal lattice, then $U(\boldsymbol{y}+\boldsymbol{\tau})$ are the coordinates in the local coordinate system for $\boldsymbol{\tau}$. The matrix U is given by

$$U^{\mathrm{T}} = (\hat{u}_1, \hat{u}_2, \hat{u}_3),$$

where $\hat{u}_1$, $\hat{u}_2$, and $\hat{u}_3$ are the axes of the local coordinate system, written in the global coordinates of the reciprocal lattice. Thus $\hat{u}_1 = \boldsymbol{\tau}/\tau$, and $\hat{u}_2$ and $\hat{u}_3$ are unit vectors perpendicular to $\hat{u}_1$ and to each other. The matrix U is unitarian, that is $U^{-1} = U^{\mathrm{T}}$. The translation between global and local coordinates is

$$\boldsymbol{x} = U(\boldsymbol{y}+\boldsymbol{\tau}) \qquad \boldsymbol{y} = U^{\mathrm{T}}\boldsymbol{x} - \boldsymbol{\tau}$$

The expression for the 3-dimensional Gaussian in global coordinates is

$$G(\boldsymbol{y}) = \frac{1}{(\sqrt{2\pi})^3}\frac{1}{\sigma_1\sigma_2\sigma_3}e^{-(U(\boldsymbol{y}+\boldsymbol{\tau}))^{\mathrm{T}}D(U(\boldsymbol{y}+\boldsymbol{\tau}))} \tag{5.21}$$

The elastic coherent cross-section is then given by

$$\left(\frac{d\sigma}{d\Omega}\right)_{\text{coh.el.}} = N\frac{(2\pi)^3}{V_0}\sum_{\boldsymbol{\tau}} G(\boldsymbol{\tau}-\boldsymbol{\kappa})\delta(k_{\mathrm{i}}-k_{\mathrm{f}})|F_{\boldsymbol{\tau}}|^2 \tag{5.22}$$

where the δ-function specifies the condition that the scattering must be elastic.

The user must specify a list of reciprocal lattice vectors $\boldsymbol{\tau}$ to consider along with their structure factors $|F_{\boldsymbol{\tau}}|^2$. The user must also specify the coordinates (in direct space) of the unit cell axes $\boldsymbol{a}$, $\boldsymbol{b}$, and $\boldsymbol{c}$, from which the reciprocal lattice will be computed.

In this version of the Single_crystal component, no account is taken of extinction (the sample is assumed to be so thin that extinction is not important). A future version will include secondary extinction and multiple scattering.

The algorithm

The overview of the algorithm used in the Single_crystal component is as follows:

1. Check if the neutron intersects the crystal, and if so, select at random a point of scattering inside the crystal.
2. Search through a list of reciprocal lattice points of interest, selecting those that are close enough to the Ewald sphere to have a non-vanishing scattering probability.
3. Of the selected reciprocal lattice points, choose one at random for this scattering event.
4. Select an outgoing wave vector $\boldsymbol{k}_{\mathrm{f}}$ from the intersection between the Ewald sphere and the Gaussian ellipsoid.
5. Adjust the neutron weight to get the correct cross-section in (5.22).

For point 1, since no extinction is considered the scattering point is chosen uniformly on the potential flight path through the crystal. For point 2, the distance *dist* between a reciprocal lattice point and the Ewald sphere is considered small enough to allow scattering if it is less than five times the maximum axis of the Gaussian, $\mathit{dist} \leq 5\max(\sigma_1, \sigma_2, \sigma_3)$.

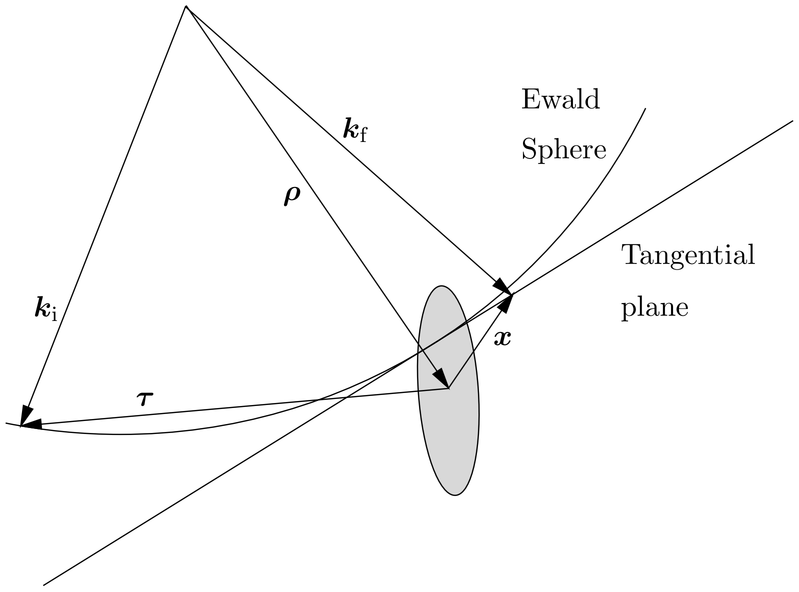


Figure 5.11: The scattering triangle in the single crystal.

Choosing the outgoing wave vector The final wave vector $\boldsymbol{k}_\mathrm{f}$ must lie on the intersection between the Ewald sphere and the Gaussian ellipsoid. Since η and $\Delta d/d$ are assumed small, the intersection can be approximated with a plane tangential to the sphere, see figure 5.11. The tangential point is taken to lie on the line between the center of the Ewald sphere $-\boldsymbol{k}_\mathrm{i}$ and the reciprocal lattice point $\boldsymbol{\tau}$. Since the radius of the Ewald sphere is k_i, this point is

$$\boldsymbol{o} = (1 - k_\mathrm{i}/\rho)\boldsymbol{\rho} - \boldsymbol{\tau}$$

where $\rho = \boldsymbol{k}_\mathrm{i} - \boldsymbol{\tau}$.

The equation for the plane is

$$\boldsymbol{P}(\boldsymbol{t}) = \boldsymbol{o} + B\boldsymbol{t}, \qquad \boldsymbol{t} \in \mathbb{R}^2 \tag{5.23}$$

Here $B = (\boldsymbol{b}_1, \boldsymbol{b}_2)$ is a 3×2 matrix with the two generators for the plane $\boldsymbol{b}_1$ and $\boldsymbol{b}_2$. These are (arbitrary) unit vectors in the plane, being perpendicular to each other and to the plane normal $\boldsymbol{n} = \boldsymbol{\rho}/\rho$.

Each $\boldsymbol{t}$ defines a potential final wave vector $\boldsymbol{k}_\mathrm{f}(\boldsymbol{t}) = \boldsymbol{k}_\mathrm{i} + \boldsymbol{P}(\boldsymbol{t})$. The value of the 3-dimensional Gaussian for this $\boldsymbol{k}_\mathrm{f}$ is

$$G(\boldsymbol{x}(\boldsymbol{t})) = \frac{1}{(\sqrt{2\pi})^3}\frac{1}{\sigma_1\sigma_2\sigma_3}e^{-\boldsymbol{x}(\boldsymbol{t})^\mathrm{T}D\boldsymbol{x}(\boldsymbol{t})} \tag{5.24}$$

where $\boldsymbol{x}(\boldsymbol{t}) = \boldsymbol{\tau} - (\boldsymbol{k}_\mathrm{i} - \boldsymbol{k}_\mathrm{f}(\boldsymbol{t}))$ is given in local coordinates for $\boldsymbol{\tau}$. It can be shown that equation (5.24) can be re-written as

$$G(\boldsymbol{x}(\boldsymbol{t})) = \frac{1}{(\sqrt{2\pi})^3}\frac{1}{\sigma_1\sigma_2\sigma_3}e^{-\alpha}e^{-(\boldsymbol{t}-\boldsymbol{t}_0)^\mathrm{T}N(\boldsymbol{t}-\boldsymbol{t}_0)} \tag{5.25}$$

where $N = B^{\mathrm{T}}DB$ is a 2×2 symmetric and positive definite matrix, $\boldsymbol{t}_0 = -N^{-1}B^{\mathrm{T}}D\boldsymbol{o}$ is a 2-vector, and $\alpha = -\boldsymbol{t}_0^{\mathrm{T}}N\boldsymbol{t}_0 + \boldsymbol{o}^{\mathrm{T}}D\boldsymbol{o}$ is a real number. Note that this is a two-dimensional Gaussian (not necessarily normalized) in $\boldsymbol{t}$ with center $\boldsymbol{t}_0$ and axis defined by N.

To choose $\boldsymbol{k}_\mathrm{f}$ we sample $\boldsymbol{t}$ from the 2-dimensional Gaussian distribution (5.25). To do this, we first construct the Cholesky decomposition of the matrix $(\frac{1}{2}N^{-1})$. This gives a 2×2 matrix L such that $LL^{\mathrm{T}} = \frac{1}{2}N^{-1}$ and is possible since N is symmetric and positive definite. It is given by

$$L = \begin{pmatrix} \sqrt{\nu_{11}} & 0 \\ \frac{\nu_{12}}{\sqrt{\nu_{11}}} & \sqrt{\nu_{22} - \frac{\nu_{12}^2}{\nu_{11}}} \end{pmatrix} \qquad \text{where } \frac{1}{2}N^{-1} = \begin{pmatrix} \nu_{11} & \nu_{12} \\ \nu_{12} & \nu_{22} \end{pmatrix}$$

Now let $\boldsymbol{g} = (g_1, g_2)$ be two random numbers drawn form a Gaussian distribution with mean 0 and standard deviation 1, and let $\boldsymbol{t} = L\boldsymbol{g} + \boldsymbol{t}_0$. The probability of a particular $\boldsymbol{t}$ is then

$$\begin{aligned} P(\boldsymbol{t})d\boldsymbol{t} &= \frac{1}{2\pi}e^{-\frac{1}{2}\boldsymbol{g}^{\mathrm{T}}\boldsymbol{g}}d\boldsymbol{g} && (5.26) \\ &= \frac{1}{2\pi}\frac{1}{\det L}e^{-\frac{1}{2}(L^{-1}(\boldsymbol{t}-\boldsymbol{t}_0))^{\mathrm{T}}(L^{-1}(\boldsymbol{t}-\boldsymbol{t}_0))}d\boldsymbol{t} && (5.27) \\ &= \frac{1}{2\pi}\frac{1}{\det L}e^{-(\boldsymbol{t}-\boldsymbol{t}_0)^{\mathrm{T}}N(\boldsymbol{t}-\boldsymbol{t}_0)}d\boldsymbol{t} && (5.28) \end{aligned}$$

where we used that $\boldsymbol{g} = L^{-1}(\boldsymbol{t} - \boldsymbol{t}_0)$ so that $d\boldsymbol{g} = \frac{1}{\det L}d\boldsymbol{t}$. This is just the normalized form of (5.25). Finally we set $\boldsymbol{k}'_\mathrm{f} = \boldsymbol{k}_\mathrm{i} + \boldsymbol{P}(\boldsymbol{t})$ and $\boldsymbol{k}_\mathrm{f} = (k_\mathrm{i}/k'_f)\boldsymbol{k}'_\mathrm{f}$ to normalize the length of $\boldsymbol{k}_\mathrm{f}$ to correct for the (small) error introduced by approximating the Ewald sphere with a plane.

Adjusting the neutron weight We now calculate the correct neutron weight adjustment. The probability of a neutron with initial wave vector $\boldsymbol{k}_\mathrm{i}$ that hits the crystal within a small area A being scattered with a wave vector $\boldsymbol{k}_\mathrm{f}$ within a small solid angle $d\Omega$ is $n_\mathrm{in}/n_\mathrm{out}$, where n_in and n_out are the number of incident and scattered neutrons, respectively. The definition of the cross-section is

$$\left(\frac{d\sigma}{d\Omega}\right)_{\text{coh.el.}} = n_\mathrm{out}/\phi_\mathrm{in}$$

where $\phi_\mathrm{in} = n_\mathrm{in}/A$ is the incoming flux. We can thus express the scattering probability in terms of the cross-section as follows:

$$\Pi(\boldsymbol{\tau}, \boldsymbol{k}_\mathrm{f}) = \frac{n_\mathrm{out}}{\phi_\mathrm{in}A} = \frac{1}{A}\left(\frac{d\sigma}{d\Omega}\right)_{\text{coh.el.}}$$

The volume of the crystal as seen by a beam with cross-section A is $\ell A = NV_0$ where ℓ is the path length of the beam all the way through the crystal. The probability of scattering in the physical model is thus

$$\begin{aligned} \Pi(\boldsymbol{\tau}, \boldsymbol{k}_\mathrm{f}) &= \frac{\ell}{NV_0}\left(\frac{d\sigma}{d\Omega}\right)_{\text{coh.el.}} && (5.29) \\ &= \frac{\ell(2\pi)^3}{V_0^2}\delta(k_\mathrm{i} - k_\mathrm{f})|F_{\boldsymbol{\tau}}|^2\frac{1}{(\sqrt{2\pi})^3}\frac{1}{\sigma_1\sigma_2\sigma_3}e^{-\boldsymbol{x}^{\mathrm{T}}D\boldsymbol{x}} && (5.30) \end{aligned}$$

where $\boldsymbol{x} = \boldsymbol{\tau} - (\boldsymbol{k}_\mathrm{i} - \boldsymbol{k}_\mathrm{f})$.

The Monte Carlo probability $f(\boldsymbol{\tau}, \boldsymbol{k}_\mathrm{f})$ of the scattering event taking place in the simulation is the product of the probability of selecting the particular reciprocal lattice point $\boldsymbol{\tau}$ and the probability of selecting the particular $\boldsymbol{k}_\mathrm{f}$. Let a be the number of reciprocal lattice vectors closer than *dist* to the Ewald sphere. From (5.28) we then have

$$\begin{aligned} f(\boldsymbol{\tau}, \boldsymbol{k}_\mathrm{f}) d\Omega &= \frac{1}{a}\frac{1}{2\pi}\frac{1}{\det L} e^{-(\boldsymbol{t}-\boldsymbol{t}_0)^\mathrm{T} N (\boldsymbol{t}-\boldsymbol{t}_0)} d\boldsymbol{t} && (5.31) \\ &= \frac{1}{a}\frac{1}{2\pi}\frac{k_\mathrm{i}^2}{\det L}\delta(k_\mathrm{i} - k_\mathrm{f}) e^{\alpha} e^{-\boldsymbol{x}^\mathrm{T} D \boldsymbol{x}} d\Omega && (5.32) \end{aligned}$$

where we used equations (5.24) and (5.25), as well as the fact that $d\boldsymbol{t} = k_\mathrm{i}^2 d\Omega$. We also introduced a δ-function since by construction, $f(\boldsymbol{\tau}, \boldsymbol{k}_\mathrm{f})$ is non-zero only when $k_\mathrm{i} = k_\mathrm{f}$.

We can now use equation (4.8) to get the correct weight adjustment:

$$\begin{aligned} \pi(\boldsymbol{\tau}, \boldsymbol{k}_\mathrm{f}) &= \frac{\Pi(\boldsymbol{\tau}, \boldsymbol{k}_\mathrm{f})}{f(\boldsymbol{\tau}, \boldsymbol{k}_\mathrm{f})} && (5.33) \\ &= \frac{\ell}{V_0^2} |F_{\boldsymbol{\tau}}|^2 (2\pi)^{5/2} \frac{\det L}{k_\mathrm{i}^2} a \frac{e^{-\alpha}}{\sigma_1 \sigma_2 \sigma_3} && (5.34) \end{aligned}$$

The implementation

The equations describing the Single_crystal simulation are quite complex, and consequently the code is fairly sizeable. Most of it is just the expansion of the vector and matrix equations in individual coordinates, and should thus be straightforward to follow.

The implementation pre-computes a lot of the necessary values in the `INITIALIZE` section. It is thus actually very efficient despite the complexity. If the list of reciprocal lattice points is big, however, the search through the list will be slow. The precomputed data is stored in the structures `hkl_info` and in an array of `hkl_data` structures (one for each reciprocal lattice point in the list). In addition, for every neutron event an array of `tau_data` is computed with one element for each reciprocal lattice point close to the Ewald sphere. Except for the search for possible $\boldsymbol{\tau}$ vectors, all computations are done in local coordinates using the matrix U to do the necessary transformations.

The list of reciprocal lattice points is specified in an ASCII data file. Each line contains seven numbers, separated by white space. The first three numbers are the (h, k, l) indices of the reciprocal lattice point, and the last number is the value of the structure factor $|F_{\boldsymbol{\tau}}|^2$, in barns. The middle three numbers are not used; they are nevertheless required since this makes the file format compatible with the output from the Crystallographica program [10].

The input parameters for the components are *xwidth*, *yheight*, and *zthick* to define the dimensions of the crystal in meters; *delta_d_d* and *mosaic* to give the value of $\Delta d/d$ (no unit) and η (in minutes of arc); (*ax*, *ay*, *az*), (*bx*, *by*, *bz*), and (*cx*, *cy*, *cz*) to define the axes of the direct lattice of the crystal (the sides of the unit cell) in units of Ångstrøm; and *reflections*, a string giving the name of the file with the list of structure factors to consider.

5.6.4 Monochromator: The monochromator crystal

The component **Monochromator** is obsolete as from McStas version 1.2. Use the component **Mosaic_anisotropic** instead.

5.7 Powder-like sample components

In this section, we consider elastic coherent and incoherent scattering from polycrystalline samples. We have chosen to simulate the correct physical processes within the powder samples on a quite detailed level.

Within many samples, the incident beam is attenuated by scattering and absorption, so that the illumination varies considerably throughout the sample. For single crystals, this phenomenon is known as *secondary extinction* [11], but the effect is also important in powders. In analytical treatments, attenuation is difficult to deal with, and is thus often ignored, making a *thin sample approximation.* In Monte Carlo simulations, the beam attenuation is easily taken care of, as will be shown below. For simplicity we ignore multiple scattering, which will be implemented in a later version of McStas.

5.7.1 Weight transformation in samples; focusing

Let us look in detail on how to simulate the physics of the scattering process within the powder. The sample has an absorption cross section per unit cell of σ_c^a and a scattering cross section per unit cell of σ_c^s. The neutron path length in the sample before the scattering event is denoted by l_1, and the path length within the sample after the scattering is denoted by l_2, see figure 5.12. We then define the inverse penetration lengths as $\mu^s = \sigma_c^s/V_c$ and $\mu^a = \sigma_c^a/V_c$, where V_c is the volume of a unit cell. Physically, the beam along this path is attenuated according to

$$P(l) = \exp(-l(\mu^s + \mu^a)), \tag{5.35}$$

where the normalization is taken to be $P(0) = 1$.

The probability for a neutron to be scattered from within the interval $[l_1; l_1 + dl]$ will be

$$\Pi(l_1)dl = \mu^s P(l_1)dl, \tag{5.36}$$

while the probability for a neutron to be scattered from within this interval into the solid angle Ω *and* not being scattered further or absorbed on the way out of the sample is

$$\Pi(l_1, \Omega)dld\Omega = \mu^s P(l_1)P(l_2)\gamma(\Omega)d\Omega dl, \tag{5.37}$$

where $\gamma(\Omega)$ is the directional distribution of the scattered neutrons, and l_2 is determined by l_1, Ω, and the sample geometry, see figure 5.12.

In our Monte-Carlo simulations, we will often choose the scattering parameters by making a Monte-Carlo choice of l_1 and Ω from a distribution different from $\Pi(l_1, \Omega)$. By doing this, we must adjust π_i according to the probability transformation rule (4.8). If we *e.g.* choose the scattering depth, l_1, from a flat distribution in $[0; l_{\rm full}]$, and choose the directional dependence from $g(\Omega)$, we have a Monte Carlo probability

$$f(l_1, \Omega) = g(\Omega)/l_{\rm full}, \tag{5.38}$$

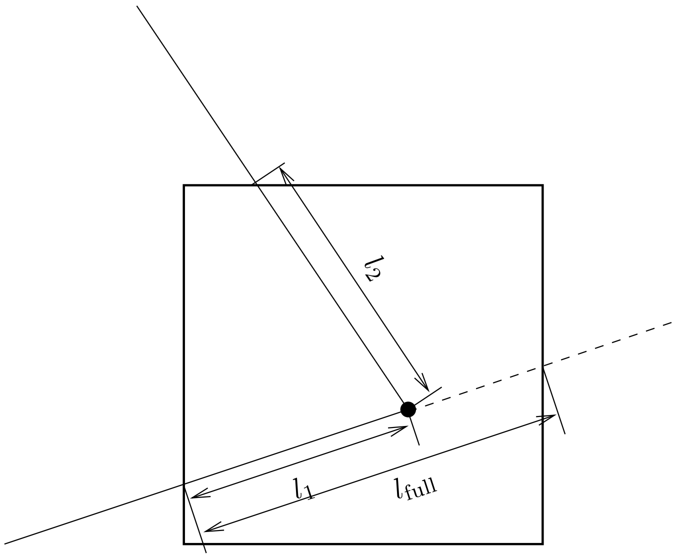


Figure 5.12: The geometry of a scattering event within a powder sample.

l_{full} is here the path length through the sample as taken by a non-scattered neutron (although we here assume that all simulated neutrons are being scattered). According to (4.8), the neutron weight factor is now adjusted by the amount

$$\pi_i(l_1, \Omega) = \mu^s l_{\text{full}} \exp\left[-(l_1 + l_2)(\mu^a + \mu^s)\right] \frac{\gamma(\Omega)}{g(\Omega)}. \tag{5.39}$$

In analogy with the source components, it is possible to define interesting directions for the scattering. One will then try to focus the scattered neutrons, choosing a $g(\Omega)$, which peaks around these directions. To do this, one uses (5.39), where the fraction $\gamma(\Omega)/g(\Omega)$ corrects for the focusing. One must choose a proper distribution so that $g(\Omega) > 0$ in every interesting direction. If this is not the case, the Monte Carlo simulation gives incorrect results.

All samples of the powder type have been constructed with a focusing and a non-focusing option.

5.7.2 V_sample: An incoherent scatterer, the V-sample

A vanadium sample is frequently being used for calibration purposes, as almost all of the scattering from the sample occurs incoherently.

In the component **V_sample**, shown in B.7.2 we assume *only* absorption and incoherent scattering. For the sample geometry, we have assumed the shape of a hollow cylinder (which has the solid cylinder as a limiting case). The sample dimensions are: Inner radius r_i, outer radius r_o, and height h, see figure 5.13.

When calculating the neutron path length within the sample material, the kernel function `CYLINDER_INTERSECT` is used twice, once for the outer radius and once for the inner radius.

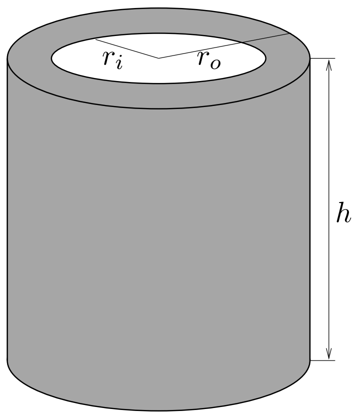


Figure 5.13: The geometry of the hollow-cylinder vanadium sample.

The incoherent scattering gives a completely uniform angular distribution of the scattered neutrons from each V-nucleus: $\gamma(\Omega) = 1/4\pi$. For the focusing we choose to have a uniform distribution on a target sphere of radius r_t, at the position $(x_\mathrm{t}, y_\mathrm{t}, z_\mathrm{t})$ in the local coordinate system. This gives an angular distribution (in a small angle approximation) of

$$g(\Omega) = \frac{1}{4\pi} \frac{x_\mathrm{t}^2 + y_\mathrm{t}^2 + z_\mathrm{t}^2}{(\pi r_\mathrm{t}^2)}. \tag{5.40}$$

The input parameters for the component **V_sample** are the sample dimensions (r_i, r_o, and h), the packing factor for the V-sample (`pack`), and the focusing parameters ($x_\mathrm{t}, y_\mathrm{t}, z_\mathrm{t}$, and r_t) for the target sphere. The relevant material parameters for V (σ_c^s, σ_c^a, and the unit cell volume V_c) are contained within the component.

Note: When simulating a realistic V-sample of this geometry one finds that the resulting direction dependence of the scattered intensity is *not* isotropic. This is explained by the variation of attenuation with scattering angle. One test result is shown in Appendix D.

5.7.3 Powder1: A general powder sample

General considerations

An ideal powder sample consists of many small crystallites, although each crystallite is sufficiently large not to cause size broadening. The orientation of the crystallites is evenly distributed, and there is thus always a certain number of crystallites oriented to fulfill the Bragg condition

$$n\lambda = 2d \sin\theta, \tag{5.41}$$

where n is the order of the scattering (an integer), λ is the neutron wavelength, d is the lattice spacing of the sample, and 2θ is the scattering angle, see figure 5.14. As all crystal orientations are realised in a powder sample, the neutrons are scattered within a *Debye-Scherrer cone* of opening angle 4θ [11].

Equation (5.41) may be cast into the form

$$|\mathbf{Q}| = 2|\mathbf{k}| \sin\theta, \tag{5.42}$$

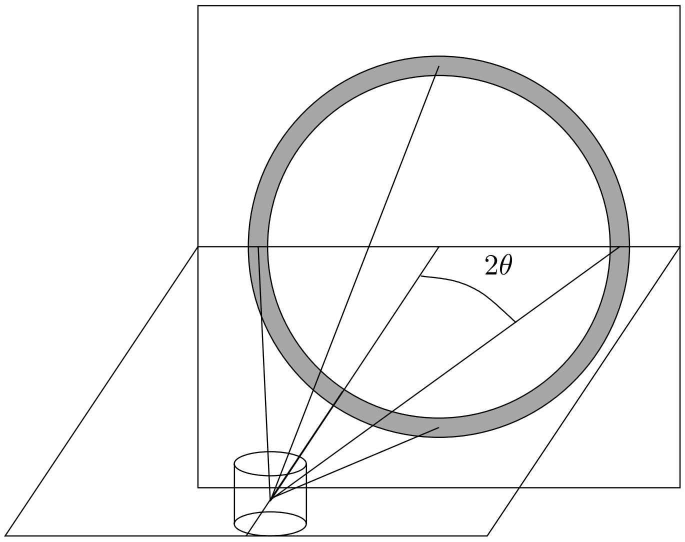


Figure 5.14: The scattering geometry of a powder sample showing the Debye-Scherrer cone and the Debye-Scherrer circle.

where $\mathbf{Q}$ is a vector of the reciprocal lattice, and $\mathbf{k}$ is the wave vector of the neutron. It is seen that only reciprocal vectors fulfilling $|\mathbf{Q}| < 2|\mathbf{k}|$ contribute to the scattering. For a complete treatment of the powder sample, one needs to take into account all these $\mathbf{Q}$-values, since each of them contribute to the attenuation.

The textbook expression for the scattering intensity from one reflection in a slab-shaped powder sample, much larger than the beam cross section, reads [11]

$$\frac{P}{P_0} = \frac{\lambda^3 l_s}{4\pi r}\frac{\rho'}{\rho} tjN_c^2|F(\mathbf{Q})|^2 \exp(-2W)\frac{\exp(-\mu^{\mathrm{a}}t/cos\theta)}{\sin^2(2\theta)} \tag{5.43}$$

$$|F(\mathbf{Q})|^2 = \left|\sum_j b_j \exp(\mathbf{R}_j \cdot \mathbf{Q})\right|^2, \tag{5.44}$$

where the sum in the structure factor runs over all atoms in one unit cell. The meanings and units of the symbols are

P_0	s^{-1}	Incoming intensity of neutrons
P	s^{-1}	Detected intensity of neutrons
l_s	m	Height of detector
r	m	Distance from sample to detector
ρ'/ρ	1	Packing factor of the powder
t	m	Slab thickness
j	1	Multiplicity of the reflection
N_c	m^{-3}	Density of unit cells in bulk material
$\|F(\mathbf{Q})\|^2$	m^2	Structure factor
$\exp(-2W)$	1	Debye-Waller factor
μ^a	m^{-1}	Linear attenuation factor due to absorption.

In analogy with this, the textbook expression for a cylinder shaped powder sample, completely illuminated by the beam, reads [11]

$$\frac{P}{\Psi_0} = \frac{V\rho'}{\rho} N_c^2 |F(\mathbf{Q})|^2 j \exp(-2W) \frac{A_{hkl}}{\sin(\theta)\sin(2\theta)} \frac{l_s}{2\pi r} \frac{\lambda^3}{4}, \tag{5.45}$$

where the new symbols are

Ψ_0	$\mathrm{s}^{-1}\mathrm{m}^{-2}$	Incoming beam flux
V	m^3	Sample volume
A_{hkl}	1	Attenuation factor.

Eq. (5.43) for a slab shaped sample may be cast into the form of the cylinder expression above by using the substitutions

Incoming flux	$P_0/(wh\cos\theta)$	$\rightarrow$	Ψ_0
Sample volume	wht	$\rightarrow$	V
Absorption factor	$\exp(-\mu^a t/\cos\theta)$	$\rightarrow$	A_{hkl},

where h and w are the height and width of the sample, respectively. Often, one defines the *scattering power* as

$$Q \equiv N^2 \frac{|F(\mathbf{Q})|^2 \lambda^3}{V\sin(2\theta)} = N_c^2 V \frac{\rho'}{\rho} \frac{|F(\mathbf{Q})|^2 \lambda^3}{\sin(2\theta)}, \tag{5.46}$$

where N is the number of unit cells.

A cut though the Debye-Scherrer cone perpendicular to its axis is a circle. At the distance r from the sample, the radius of this circle is $r\sin(2\theta)$. Thus, the detector (in a small angle approximation) only counts a fraction $f_d = l_s/(2\pi r \sin(2\theta))$ of the scattered neutrons. One may now calculate the linear attenuation coefficient in the material due to scattering (from one $\mathbf{Q}$-value only):

$$\mu^s \equiv -\frac{1}{P_0}\frac{d(P/f_d)}{dl} = \frac{Q}{V} j \exp(-2W)\cos(\theta). \tag{5.47}$$

A powder sample will in general have several allowed reflections $\mathbf{Q}_j$, which will all contribute to the attenuation. These reflections will have different values of $|F(\mathbf{Q}_j)|^2$ (and hence of Q_j), j_j, $\exp(-2W_j)$, and θ_j. The total attenuation through the sample due to scattering is given by $\mu^s = \mu^s_{\mathrm{inc}} + \sum_j \mu^s_j$, where μ^s_{inc} represents the incoherent scattering.

This implementation

For component **Powder1**, we assume that the sample has the shape of a solid cylinder. Further, the incoherent scattering is only taken into account by the attenuation of the beam, given by (5.47) and σ^a_c. The incoherently scattered neutrons are not propagated through to the detector, but rather not generated at all. Focusing is performed by only scattering into one angular interval, $d\phi$ of the Debye-Scherrer circle. The center of this interval is located at the point where the Debye-Scherrer circle intersects the half-plane defined by the initial velocity, $\mathbf{v}_\mathrm{i}$, and a user-specified vector, $\mathbf{f}$. Multiple scattering is not implemented.

The input parameters for this component are

r	m	Radius of cylinder
h	m	Height of cylinder
σ_c^a	fm^2	Absorption cross section per unit cell (at 2200 m/s)
$\sigma_{i,c}^s$	(fm)^2	Incoherent scattering cross section per unit cell
ρ'/ρ	1	Packing factor
V_c	Å^3	Volume of unit cell
$\mathbf{Q}$	Å^{-1}	The reciprocal lattice vector under consideration
$\|F(\mathbf{Q}_j)\|^2$	(fm)^2	Structure factor
j	1	Multiplicity of reflection
$\exp(-2W)$	1	Debye-Waller factor
$d\phi$	deg	Angular interval of focusing
f_x	m	
f_y	m	Focusing vector
f_z	m	

The source text for the component is shown in Appendix B.7.3.

In a later version, more reciprocal lattice vectors will be allowed. Further, we intent to include the effect of multiple scattering.

5.8 Inelastic scattering kernels

In this section, samples with inelastic scattering are described. Currently, only a single sample is available that scatters uniformly in $(\mathbf{Q}, \omega)$ and is used for computing resolution functions in tripple-axis instruments.

5.8.1 Res_sample: A uniform scatterer for resolution calculation

The component **Res_sample** models an inelastic sample that scatters completely homogeneous in position and energy; regardless of the state of the incoming neutron, all directions and energies for the scattered neutron have the same probability. This clearly does not correspond any physically realizable samples, but the component is very useful for computation of the resolution function and may also be used for test and debugging purposes. The component is designed to be used together with the **Res_monitor** component, described in section 5.5.12.

The shape of the sample is either a hollow cylinder (like the vanadium sample described in section 5.7.2) or a rectangular box. The hollow cylinder shape is specified with inner and outer radius *radius_i* and *radius_o* and height *h*. If *radius_o* is negative, the shape is instead a box of width *radius_i* along the X axis, height *h*, and thickness $-$*radius_o* along the Z axis, centered on the Z axis and with the front face in the X-Y plane. See figure 5.15.

The component only propagates the neutrons that are scattered; neutrons that would pass through or miss the sample are absorbed. There is no modeling of the cross section of the sample, secondary extinction *etc.*; the scattering probability is proportional to the neutron flight path length inside the sample, with the constant of proportionality arbitrarily set to $1/(2|radius\_o|)$. The reason for this is that the component is designed

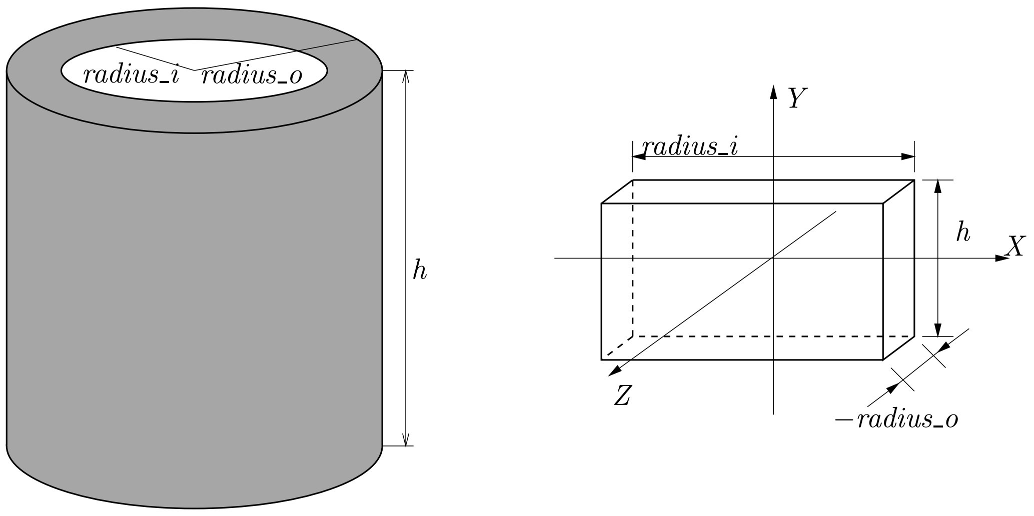


Figure 5.15: The two possible shapes of the **Res_sample** component.

for computing the resolution function of an instrument, including the sample size but independent of any sample properties such as scattering and absorbtion cross sections.

The point of scattering in the sample is chosen at a random position along the neutron flight path inside the sample, and the scattered neutron is given a random energy and direction. The energy is selected in a user-specified interval $[E_0 - \Delta E; E_0 + \Delta E]$ which must be chosen large enough to cover all interesting neutrons, but preferably not excessively large for reasons of efficiency. Similarly, the direction is chosen in a user-specified range; the range is such that a sphere of given center and radius is fully illuminated.

A special feature, used when computing resolution functions, is that the component stores complete information about the scattering event in the output parameter *res_struct*. The information includes initial and final wave vectors, the coordinates of the scattering point, and the neutron weight after the scattering event. From this information the scattering parameters $(\mathbf{Q}_i, \omega_i)$ for every scattering event i may be recorded and used to compute the resolution function of an instrument, as explained below. For an example of how to use the information in the output parameter, see the description of the **Res_monitor** component in section 5.5.12.

The input parameters to the **Res_sample** components are the sample dimensions *radius_i*, *radius_o*, and *h*, all in meters; the center of the scattered energy range *E0* and the energy spread *dE* in meV; and the target sphere position in the local coordinate system *target_x*, *target_y*, *target_z*, and radius *focus_r*, in meters. The only output parameter is *res_struct* containing information about the scattering event, with all vectors given in the local coordinate system of the component in units of meter.

Background

In an experiment, as well as in the simulation, the expected intensity is by definition of the resolution function given by

$$I = \int R(\mathbf{Q},\omega)\sigma(\mathbf{Q},\omega)d\mathbf{Q}d\omega$$

Here $I(\mathbf{Q}_0,\omega_0)$ is the measured or simulated intensity in the detector, R is the resolution function for the instrument in a given setup, σ is the scattering cross section of the sample, and $(\mathbf{Q},\omega)$ denote the scattering vector and energy transfer in the sample. For the uniform scatterer, $\sigma(\mathbf{Q},\omega) = 1/V_0$ is a constant, so we have

$$I = 1/V_0 \int R(\mathbf{Q},\omega)d\mathbf{Q}d\omega$$

If we instead consider only the intensity contributed by scattering with parameters $(\mathbf{Q},\omega)$ that lie within a small part $\Delta\Omega$ of the total phase space and has volume ΔV,

$$I_{\Delta\Omega} = 1/V_0 \int_{\Delta\Omega} R(\mathbf{Q},\omega)d\mathbf{Q}d\omega = \frac{\Delta V}{V_0}R(\Delta\Omega)$$

(where $R(\Delta\Omega)$ denotes the average value of R over $\Delta\Omega$), we get a good approximation of the value of R provided that $\Delta\Omega$ is sufficiently small. This is useful with the output from the simulations, since $I_{\Delta\Omega}$ is approximated by

$$I_{\Delta\Omega} \approx \sum_{(\mathbf{Q_i},\omega_i)\in\Delta\Omega} p_i$$

This can be used to histogram the resolution function or visualize it in different ways. The 3D visualization of the resolution function produced by the `mcresplot` program for example uses this by displaying a cloud of dots, the local density of which is proportional to the resolution function.

The `mcresplot` program also computes the covariance and resolution matrices. Letting $(x_i^1, x_i^2, x_i^3, x_i^4)$ denote the $(\mathbf{Q_i},\omega_i)$ values obtained from the scattering events in the simulation and $\mu^j = (\sum_i p_i x_i^j)/(\sum_i p_i)$ the mean value of x_i^j, the covariance matrix is computed as

$$\mathbf{C}_{jk} = \Big(\sum_i p_i(x_i^j - \mu_j)(x_i^k - \mu_k)\Big)/\Big(\sum_i p_i\Big)$$

This covariance matrix is given in the local coordinate system of the sample component. The `mcresplot` program actually outputs the covariance matrix in another coordinate system which is rotated around the Y axis so that the projection to the X-Z plane of the average scattering vector $\mathbf{Q}_{\text{avg}} = (\sum_i p_i\mathbf{Q}_i)/(\sum_i p_i)$ is parallel to the X axis.

The resolution matrix $\mathbf{M}$ is the inverse of the covariance matrix and is also output in the rotated coordinate system by `mcresplot`. The 4-dimensional gaussian distribution, defined by

$$f(\mathbf{X}) = e^{-\frac{1}{2}\mathbf{X}^T\mathbf{M}\mathbf{X}} \qquad (5.48)$$

where $\mathbf{X} = (\mathbf{Q},\omega)$, has covariance matrix $\mathbf{C}$ and thus defines the gaussian resolution function with the same covariance as the resolution computed by the simulation.

The `mcresplot` program provides for the simultaneous visualization of the computed and the gaussian resolution function by obtaining an appropriate number of random points with the statistical distribution (5.48). Each point $\mathbf{X}$ is obtained as follows: A vector $\mathbf{Y}$ is generated of four individually gaussian distributed random numbers with mean zero and variance one. Using the Cholesky decomposition of $\mathbf{C}$, $\mathbf{C} = \mathbf{L}\mathbf{L}^T$, we have

$$\mathbf{X} = \mathbf{L}\mathbf{Y}.$$

Chapter 6

The instrument library

Here, we give a short description of three selected instruments. We present the McStas versions of the Risø triple axis spectrometer TAS1 (6.2) and the ISIS time-of-flight spectrometer PRISMA (6.3). Before that, however, we present one example of a component test instrument: the instrument to test the component **V_sample** (6.1).

The source text for the three instrument definitions is listed in Appendix C. These files are also included in the McStas distribution in the `examples/` directory.

6.1 A test instrument for the component V_sample

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

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

6.2 The triple axis spectrometer TAS1

With this instrument definition, we have tried to create a very detailed model of the conventional cold source triple axis spectrometer TAS1 at Risø National Laboratory. Except for the cold source itself, all components used have quite realistic properties. Further, the overall geometry of the instrument has been adapted from the detailed technical drawings of the real spectrometer. The TAS 1 simulations are by far the most extensive work yet performed with the McStas package, and a few of the simulation results are shown in Appendix D. For further details see reference [12].

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 one with a flat energy distribution (component **Source_flat**) focusing on the third slit.

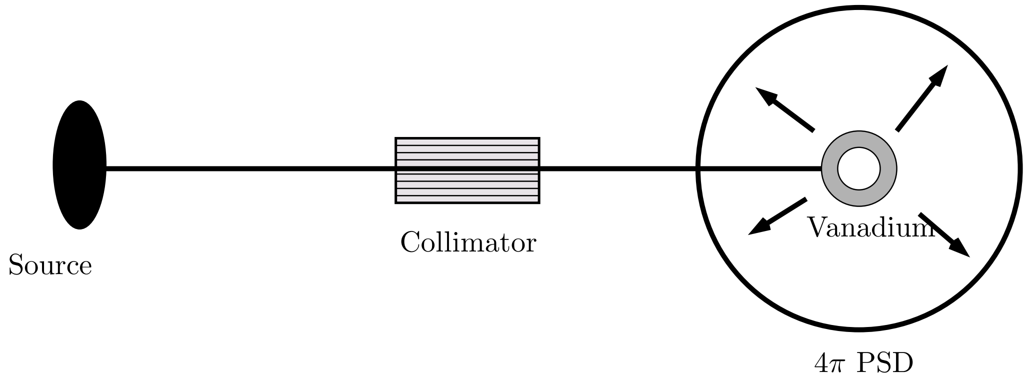


Figure 6.1: A sketch of the test instrument for the component V_sample.

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 cm by 1 cm sample. This is modeled directly in the instrument definition using seven **Monochromator** components. The mosaicity of the pyrolytic graphite crystals is nominally 30' in both directions. However, the simulations indicated that the horisontal mosaicities of both monochromator and analyser were more likely 45'. This was used for all mosaicities in the final instrument definition.

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

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

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

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

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

An illustration of the TAS1 instrument is shown in figure 6.2. Test results and data from the real spectrometer are shown in Appendix D.2.

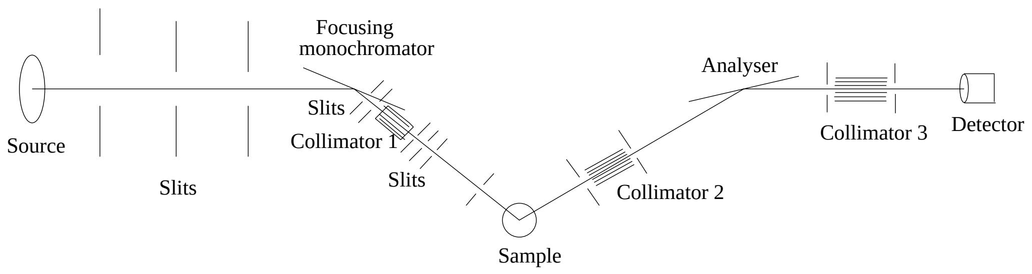


Figure 6.2: A sketch of the TAS1 instrument.

6.3 The time-of-flight spectrometer PRISMA

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

We have used the simple time-of-flight source **Tof_source**, as described under the component library. The neutrons pass through a beam channel and scatter off from a vanadium sample, pass through a collimator on to the analyser.

The RITA-style analyser consists of seven analyser crystals that can be rotated independently around a vertical axis. After the analysers we have placed a PSD and a time-of-flight detector.

To illustrate some of the things that can be done in a simulation as opposed to a real-life experiment, this example instrument further discriminates between the scattering off each individual analyser crystal when the neutron hits the detector. The analyser component is modified so that a global variable `neu_color` keeps track of which crystal scatters the neutron. The detector component is then modified to construct seven different time-of-flight histograms, one for each crystal (see the source code for the instrument in appendix C for details). One way to think of this is that the analyser blades paint a color on each neutron which is then observed in the detector.

An illustration of the instrument is shown in figure 6.3. Test results are shown in Appendix D.3.

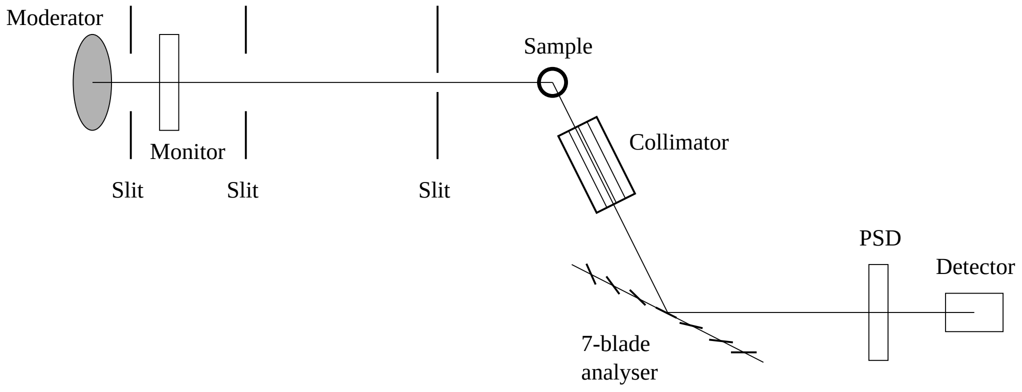


Figure 6.3: A sketch of the PRISMA instrument.

Chapter 7

Planned expansions of McStas in the future

During the work so far on McStas, we have run across a number of points we would like to include in McStas in the future.

For the McStas meta-language itself, these points include:

- Facilities for making a Monte-Carlo choice on the basis of tabulated values. This would be useful in source and filter components.
- Facilities for assembling a number of existing components into one compound component, like a multi-bladed analyser.

We also would like to improve on the component and instrument libraries:

- Output in NeXus format.
- Allow multiple scattering in sample components.
- More samples for inelastic scattering.
- A powder sample with more than one reflection.
- Handle gravitation.
- The RITA spectrometer.
- The new Risø TAS7 spectrometer.
- A detailed version of the ISIS PRISMA spectrometer.

Further, we would like to make improvements on the interface software:

- Interface to existing control software (*e.g.* the Risø program TASCOM)

Appendix A

Kernel calls and conversion constants

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

A.1 Kernel calls and functions

Here we list a number of preprogrammed macros which may ease the task of writing components

- **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.
- **DETECTOR_OUT_0D**(). Used to output the results from a single detector. The name of the detector is output together with the simulated intensity and estimated statistical error. The output is produced in a format that can be read by McStas front-end programs. See section 3.4.5 for details.
- **DETECTOR_OUT_1D**(). Used to output the results from a one-dimentional detector. See section 3.4.5 for details.
- **DETECTOR_OUT_2D**(). Used to output the results from a two-dimentional detector. See section 3.4.5 for details.
- **MC_GETPAR**(). This may be used in the finally section of an instrument definition to reference the output parameters of a component. See page 31 for details.
- **NORM**(x, y, z). Normalizes the vector (x, y, z) to have length 1.
- **PROP_Z0**. Propagates the neutron to the $z = 0$ plane, by adjusting (x, y, z) and t. If the neutron velocity points away from the $z = 0$ plane, the neutron is absorbed.
- **PROP_DT**(dt). Propagates the neutron through the time interval dt, adjusting (x, y, z) and t.

- **SCATTER**. This macro is used to denote a scattering event inside a component, see section 3.4.4.

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

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

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

And here we list a number of preprogrammed C functions.

- **cylinder_intersect**(&t_1, &t_2, x, y, z, v_x, v_y, v_z, r, h). Calculates the (0, 1, or 2) intersections between the neutron path and a cylinder of height h and radius r, 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 returned with a negative value.

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

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

- **randnorm**(). Returns a random number from a normal distribution centered around 0 and with $\sigma = 1$. The algorithm used to get the normal distribution is explained in [13], chapter 7.

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

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

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

A.2 Constants for unit conversion etc.

The following predefined constants are useful for conversion between units

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

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

Appendix B

McStas source code for the component library

List of component input and output parameters

Before listing the source code for the components, we bring a list of the components with their input and output parameters.

Source_flat	**Input:** (radius, dist, xw, yh, E0, dE) **Output:** (hdiv, vdiv, p_in)
Source_flat_lambda	**Input:** (radius, dist, xw, yh, lambda_0, d_lambda) **Output:** (hdiv, vdiv, p_in)
Source_flux_lambda	**Input:** (radius, dist, xw, yh, lambda_0, d_lambda, flux) **Output:** (hdiv, vdiv, p_in)
Source_div	**Input:** (width, height, hdiv, vdiv, E0, dE)
Moderator	**Input:** (radius, E0, E1, dist, xw, yh, t0, Ec, gam)
Source_adapt	**Input:** (xmin, xmax, ymin, ymax, dist, xw, yh, E0, dE, flux) (n_E, N_xpos, N_xdiv, alpha, beta, filename)
Arm	**Input:** ()
Slit	**Input:** (xmin, xmax, ymin, ymax)
Circular_slit	**Input:** (radius)
Beamstop_rectangular	**Input:** (xmin, xmax, ymin, ymax)
Beamstop_circular	**Input:** (radius)
Soller	**Input:** (xmin, xmax, ymin, ymax, len, divergence)
Filter	**Input:** (xmin, xmax, ymin, ymax, len, T0, T1, Emin, Emax)
Mirror	**Input:** (xlength, yheight, R0, Qc, alpha, m, W)
Guide	**Input:** (w1, h1, w2, h2, l, R0, Qc, alpha, m, W)
Channeled_Guide	**Input:** (w1, h1, w2, h2, l, d, k) (R0, Qcx, Qcy, alphax, alphay, mx, my, W)

V_selector	**Input:** (width, height, l0, r0, phi, l1, tb, rot, nb)
Chopper	**Input:** (w, R, f, n, pha) **Output:** (Tg, To)
First_Chopper	**Input:** (w, R, f, n, pha, a) **Output:** (Tg, To)
Monitor	**Input:** (xmin, xmax, ymin, ymax) **Output:** (Nsum, psum, p2sum)
Monitor_4PI	**Output:** (Nsum, psum, p2sum)
PSD_monitor	**Input:** (xmin, xmax, ymin, ymax, nx, ny, filename) **Output:** (PSD_N, PSD_p, PSD_p2)
PSD_monitor_4PI	**Input:** (radius, nx, ny, filename) **Output:** (PSD_N, PSD_p, PSD_p2)
PSD_monitor_4PI_log	**Input:** (radius, nx, ny, filename) **Output:** (PSD_N, PSD_p, PSD_p2)
TOF_monitor	**Input:** (xmin, xmax, ymin, ymax, nchan, dt, filename) **Output:** (TOF_N, TOF_p, TOF_p2)
E_monitor	**Input:** (xmin, xmax, ymin, ymax, Emin, Emax, nchan, filename) **Output:** (E_N, E_p, E_p2)
L_monitor	**Input:** (xmin, xmax, ymin, ymax, Lmin, Lmax, nchan, filename) **Output:** (L_N, L_p, L_p2)
Divergence_monitor	**Input:** (xmin, xmax, ymin, ymax, nh, nv) (h_maxdiv, v_maxdiv, filename) **Output:** (Div_N, Div_p, Div_p2)
DivPos_monitor	**Input:** (xmin, xmax, ymin, ymax, npos, ndiv, maxdiv, filename) **Output:** (Div_N, Div_p, Div_p2)
DivLambda_monitor	**Input:** (xmin, xmax, ymin, ymax, nlam, ndiv, maxdiv) (lambda_0, lambda_1, filename) **Output:** (Div_N, Div_p, Div_p2)
Res_monitor	**Input:** (xmin, xmax, ymin, ymax, filename, res_sample_comp) **Output:** (Nsum, psum, p2sum, file)
Adapt_check	**Input:** (source_comp)
Mosaic_simple	**Input:** (zmin, zmax, ymin, ymax, mosaic, R0, Qx, Qy, Qz)
Mosaic_anisotropic	**Input:** (zmin, zmax, ymin, ymax, mosaich, mosaicv, r0, Q)
Single_crystal	**Input:** (xwidth, yheight, zthick, delta_d_d, mosaic) (ax, ay, az, bx, by, bz, cx, cy, cz, reflections)
V_sample	**Input:** (radius_i,radius_o,h,pack,focus_r) (target_x, target_y, target_z)
Powder1	**Input:** (d_phi0, radius, h, pack, Vc, sigma_a, j, q, F2, DW) (target_x, target_y, target_z) **Output:** (my_s_v2, my_a_v, q_v)
Res_sample	**Input:** (radius_i, radius_o, h, focus_r, E0, dE) (target_x, target_y, target_z) **Output:** (res_struct)

B.1 Source components

B.1.1 Source_flat

```
/*******************************************************************
*
* McStas, version 1.0, released October 26, 1998
*         Maintained by Kristian Nielsen and Kim Lefmann,
*         Risoe National Laboratory, Roskilde, Denmark
*
* Component: Source_flat
*
* Written by: KL, October 30, 1997
* Modified by: KL, KN, October 5, 1998
*
* The routine is a circular neutron source, which aims at a square target
* centered at the beam (in order to improve MC-acceptance rate).  The angular
* divergence is then given by the dimensions of the target. The neutron energy is
* uniformly distrubuted between E0-dE and E0+dE.
*
* ToDo: More flexible specification of E distribution.
*
* radius: (m)   Radius of circle in (x,y,0) plane where neutrons
*               are generated.
* dist:   (m)   Distance to target along z axis.
* xw:     (m)   Width(x) of target
* yh:     (m)   Height(y) of target
* E0:     (meV) Mean energy of neutrons.
* dE:     (meV) Energy spread of neutrons.
*
*******************************************************************/

DEFINE COMPONENT Source_flat
DEFINITION PARAMETERS (radius, dist, xw, yh, E0, dE)
SETTING PARAMETERS ()
OUTPUT PARAMETERS (hdiv, vdiv, p_in)
STATE PARAMETERS (x,y,z,vx,vy,vz,t,s1,s2,p)
DECLARE
%{
 double hdiv,vdiv;
 double p_in;
%}
INITIALIZE
%{
 hdiv = atan(xw/(2.0*dist));
 vdiv = atan(yh/(2.0*dist));
 p_in = (4*hdiv*vdiv)/(4*PI); /* Small angle approx. */
%}
TRACE
%{
 double theta0,phi0,chi,theta,phi,E,v,r;

 p=p_in;
 z=0;

 chi=2*PI*rand01();                          /* Choose point on source */
 r=sqrt(rand01())*radius;                    /* with uniform distribution. */
 x=r*cos(chi);
 y=r*sin(chi);

 theta0= -atan(x/dist);              /* Angles to aim at target centre */
 phi0= -atan(y/dist);

 theta=theta0+hdiv*randpm1();        /* Small angle approx. */
 phi=phi0+vdiv*randpm1();

 E=E0+dE*randpm1();                  /* Assume linear distribution */
 v=sqrt(E)*SE2V;

 vz=v*cos(phi)*cos(theta);
 vy=v*sin(phi);
 vx=v*cos(phi)*sin(theta);
%}

MCDISPLAY
%{
  magnify("xy");
  circle("xy",0,0,0,radius);
%}

END
```

B.1.3 Source_flux_lambda

```
/*******************************************************************************
*
* McStas, the neutron ray-tracing package
*         Maintained by Kristian Nielsen and Kim Lefmann,
*         Copyright 1997-2000 Risoe National Laboratory, Roskilde, Denmark
*
* Component: Source_flux_lambda
*
* Modified by: KN, 1998 from Source_flat.comp
*
* The routine is a circular neutron source, which aims at a square target
* centered at the beam (in order to improve MC-acceptance rate).  The angular
* divergence is then given by the dimensions of the target. The neutron
* wavelength is uniformly distrubuted between lambda_0 - d_lambda and
* lambda_0 + d_lambda. The source flux is specified in neutrons per steradian
* per square cm per AAngstroem.
*
* radius:   (m)              Radius of circle in (x,y,0) plane where neutrons
*                            are generated.
* dist:     (m)              Distance to target along z axis.
* xw:       (m)              Width(x) of target
* yh:       (m)              Height(y) of target
* lambda_0: (AA)             Mean wavelength of neutrons.
* d_lambda: (AA)             Wavelength spread of neutrons.
* flux:     (1/(cm**2*st*AA) Source flux
*
*******************************************************************************/

DEFINE COMPONENT Source_flux_lambda
DEFINITION PARAMETERS (radius, dist, xw, yh, lambda_0, d_lambda, flux)
SETTING PARAMETERS ()
OUTPUT PARAMETERS (hdiv, vdiv, p_in)
STATE PARAMETERS (x,y,z,vx,vy,vz,t,s1,s2,p)
DECLARE
%{
 double hdiv,vdiv;
 double p_in;
%}
INITIALIZE
%{
  double factor, delta_lambda, source_area;

  hdiv = atan(xw/(2.0*dist));
  vdiv = atan(yh/(2.0*dist));

  delta_lambda = 2*d_lambda;
  source_area = radius*radius*PI*1e4; /* cm^2 */
  factor = flux/mcget_ncount()*delta_lambda*source_area;
  p_in = (4*hdiv*vdiv)*factor;  /* Small angle approx. */
%}

TRACE
%{
 double theta0,phi0,chi,theta,phi,lambda,v,r;
 p=p_in;
 z=0;

 chi=2*PI*rand01();                         /* Choose point on source */
 r=sqrt(rand01())*radius;                   /* with uniform distribution. */
 x=r*cos(chi);
 y=r*sin(chi);

 theta0= -atan(x/dist);             /* Angles to aim at target centre */
 phi0= -atan(y/dist);

 theta=theta0+hdiv*randpm1();       /* Small angle approx. */
 phi=phi0+vdiv*randpm1();

 lambda = lambda_0+d_lambda*randpm1();
 v = K2V*(2*PI/lambda);

 vz=v*cos(phi)*cos(theta);
 vy=v*sin(phi);
 vx=v*cos(phi)*sin(theta);
%}

MCDISPLAY
%{
  magnify("xy");
  circle("xy",0,0,0,radius);
%}

END
```

B.1.4 Source_div

```
/*******************************************************************
*
* McStas, version 1.1, released ??
*         Maintained by Kristian Nielsen and Kim Lefmann,
*         Risoe National Laboratory, Roskilde, Denmark
*
* Component: Source_div
*
* Written by: KL, November 20, 1998
*
* The routine is a rectangular neutron source, which has a gaussian
* divergent output in the forward direction.
* The neutron energy is uniformly distrubuted between E0-dE and E0+dE.
*
* ToDo: More flexible specification of E distribution.
*
* width:  (m)    Width of source
* height  (m)    Height of source
* hdiv    (deg)  FWHM of horisontal divergence
* vdiv    (deg)  FWHM of vertical divergence
* E0:     (meV) Mean energy of neutrons.
* dE:     (meV) Energy spread of neutrons.
*
*******************************************************************/

DEFINE COMPONENT Source_div
DEFINITION PARAMETERS (width, height, hdiv, vdiv, E0, dE)
SETTING PARAMETERS ()
OUTPUT PARAMETERS ()
STATE PARAMETERS (x,y,z,vx,vy,vz,t,s1,s2,p)
DECLARE
%{
  double thetah, thetav, sigmah, sigmav, tan_h, tan_v;
%}
INITIALIZE
%{
  sigmah = DEG2RAD*hdiv/(2.0*sqrt(2.0*log(2.0)));
  sigmav = DEG2RAD*vdiv/(2.0*sqrt(2.0*log(2.0)));
%}
TRACE
%{
 double E,v;

 p=1;
 z=0;
 t=0;

 x=randpm1()*width/2.0;
 y=randpm1()*height/2.0;

 E=E0+dE*randpm1();                  /* Assume linear distribution */
 v=sqrt(E)*SE2V;

 thetah = randnorm()*sigmah;
 thetav = randnorm()*sigmav;
 tan_h = tan(thetah);
 tan_v = tan(thetav);

 /* Perform the correct treatment - no small angle approx. here! */
 vz = v / sqrt(1 + tan_v*tan_v + tan_h*tan_h);
 vy = tan_v * vz;
 vx = tan_h * vz;
%}
END
```

B.1.5 Moderator

```
/*******************************************************************************
*
* McStas, version 1.0, released October 26, 1998
*         Maintained by Kristian Nielsen and Kim Lefmann,
*         Risoe National Laboratory, Roskilde, Denmark
*
* Component: Moderator
*
* Written by: KN, M.Hagan, August 1998
*
* Produces a simple time-of-flight spectrum, with a flat energy disstribution
*
* Input parameters:
*
* radius: (m)   Radius of source
* E0:     (meV) Lower edge of energy distribution
* E1:     (meV) Upper edge of energy distribution
* dist:   (m)   Distance from source to the focusing rectangle
* xw:     (m)   Width of focusing rectangle
* yh:     (m)   Height of focusing rectangle
* t0:     (mus) decay constant for low-energy neutrons
* Ec:     (meV) Critical energy, below which the flux decay is constant
* gam:    (meV) energy dependence of decay time
*
*******************************************************************************/

DEFINE COMPONENT Moderator
DEFINITION PARAMETERS (radius, E0, E1, dist, xw, yh, t0, Ec, gam)
SETTING PARAMETERS ()
OUTPUT PARAMETERS (hdiv,vdiv,p_in)
STATE PARAMETERS (x,y,z,vx,vy,vz,t,s1,s2,p)
DECLARE
%{
  double hdiv,vdiv;
  double p_in;
%}
INITIALIZE
%{
  hdiv = atan(xw/(2.0*dist));
  vdiv = atan(yh/(2.0*dist));
  p_in = (4*hdiv*vdiv)/(4*PI);
%}
TRACE
%{
  double theta0,phi0,chi,theta,phi,v,r,tauv,E;

  p=p_in;
  z=0;

  chi = 2*PI*rand01();          /* Choose point on source */
  r = sqrt(rand01())*radius;    /* with uniform distribution. */
  x = r*cos(chi);
  y = r*sin(chi);

  theta0 = -atan(x/dist);       /* Angles to aim at target centre */
  phi0 = -atan(y/dist);

  theta = theta0 + hdiv*randpm1(); /* Small angle approx. */
  phi = phi0 + vdiv*randpm1();

  E = E0+(E1-E0)*rand01();      /* Assume linear distribution */
  v = SE2V*sqrt(E);

  vz = v*cos(phi)*cos(theta);   /* Small angle approx. */
  vy = v*sin(phi);
  vx = v*cos(phi)*sin(theta);

  if(E < Ec)
  {
    tauv = t0;
  }
  else
  {
    double tmp = ((E - Ec) / gam);
    tauv = t0 / (1 + (tmp*tmp));
  }
  t = -tauv*log(rand01())*1E-6;
%}

MCDISPLAY
%{
  magnify("xy");
  circle("xy",0,0,0,radius);
%}

END
```

B.1.6 Source_adapt

```
/*******************************************************************************
*
* McStas, the neutron ray-tracing Monte-Carlo software.
* Copyright(C) 1999 Risoe National Laboratory.
*
* Component: Source_adapt
*
* Written by Kristian Nielsen 1999
*
* Rectangular source with flat energy distribution that uses adaptive
* importance sampling to improve simulation efficiency. Works
* together with the Adapt_check component.
*
* The source divides the three-dimensional phase space of (energy,
* horizontal position, horizontal divergence) into a number of
* rectangular bins. The probability for selecting neutrons from each
* bin is adjusted so that neutrons that reach the Adapt_check
* component with high weights are emitted more frequently than those
* with low weights. The adjustment is made so as to attemt to make
* the weights at the Adapt_check components equal.
*
* Focosing is achieved by only emitting neutrons towards a rectangle
* perpendicular to and placed at a certain distance along the Z axis.
* Focusing is only approximate (for simplicity); neutrons are also
* emitted to pass slightly above and below the focusing rectangle,
* more so for wider focusing.
*
* In order to prevent false learning, a parameter beta sets a
* fraction of the neutrons that are emitted uniformly, without regard
* to the adaptive distribution. The parameter alpha sets an initial
* fraction of neutrons that are emitted with low weights; this is
* done to prevent early neutrons with rare initial parameters but
* high weight to ruin the statistics before the component adapts its
* distribution to the problem at hand. Good general-purpose values
* for these parameters are alpha = beta = 0.25.
*
* INPUT PARAMETERS:
*
* xmin:   (m)          Left edge of rectangular source
* xmax:   (m)          Right edge
* ymin:   (m)          Lower edge
* ymax:   (m)          Upper edge
* dist:   (m)          Distance to target rectangle along z axis
* xw:     (m)          Width(x) of target
* yh:     (m)          Height(y) of target
* E0:     (meV)        Mean energy of neutrons
* dE:     (meV)        Energy spread (energy range is from E0-dE to E0+dE)
* flux:   (1/(cm**2*AA**st) Absolute source flux
* N_E:    (1)          Number of bins in energy dimension
* N_xpos: (1)          Number of bins in horizontal position
* N_xdiv: (1)          Number of bins in horizontal divergence
* alpha:  (1)          Learning cut-off factor (0 < alpha <= 1)
* beta:   (1)          Aggressiveness of adaptive algorithm (0 < beta <= 1)
* filename: (string)   Optional filename for adaptive distribution output
*
```

```
* OUTPUT PARAMETERS:
*
* p_in:  Internal, holds initial neutron weight
* y_0:   Internal
* C:     Internal
* r_0:   Internal
* count: Internal, counts neutrons emitted
* adpt:  Internal structure shared with the Adapt_check component
******************************************************************************/

DEFINE COMPONENT Source_adapt
DEFINITION PARAMETERS (xmin,xmax,ymin,ymax, dist, xw, yh, E0, dE, flux,
                       N_E, N_xpos, N_xdiv, alpha, beta, filename)
SETTING PARAMETERS ()
OUTPUT PARAMETERS (p_in, y_0, C, r_0, count, adpt)
STATE PARAMETERS (x,y,z,vx,vy,vz,t,s1,s2,p)
DECLARE
%{
  struct source_adapt
    {
      struct adapt_tree *atree; /* Adaptive search tree */
      int idx;                  /* Index of current bin */
      double *psi, *n;          /* Arrays of weight sums, neutron counts */
      double psi_tot;           /* Total weight sum */
      double pi, num;           /* Initial p, number of bins in tree */
      double factor;            /* Adaption quality factor */
      double a_beta;            /* Adaption agression factor */
    } adpt;
  double count;                 /* Neutron counter */
  double y_0, C, r_0;
  double p_in;
%}
INITIALIZE
%{
  int i;
  double a, lambda_min, lambda_max, delta_lambda, source_area;

  adpt.num = N_E*N_xpos*N_xdiv;
  adpt.a_beta = beta;
  lambda_min = sqrt(81.81/(E0+dE)); /* AAngstroem */
  lambda_max = sqrt(81.81/(E0-dE));
  delta_lambda = lambda_max - lambda_min;
  source_area = (xmax - xmin)*(ymax - ymin)*1e4; /* cm^2 */
  p_in = flux/mcget_ncount()*delta_lambda*source_area;
  adpt.atree = adapt_tree_init(adpt.num);
  adpt.psi = malloc(adpt.num*sizeof(*adpt.psi));
  adpt.n = malloc(adpt.num*sizeof(*adpt.n));
  if(!(adpt.psi && adpt.n))
  {
    fprintf(stderr, "Fatal error: out of memory.\n");
    exit(1);
  }
  for(i = 0; i < adpt.num; i++)
  {
    adapt_tree_add(adpt.atree, i, 1.0/adpt.num);
    adpt.psi[i] = adpt.n[i] = 0;
  }
  adpt.psi_tot = 0;
  count = 0;
  y_0 = adpt.num > 8 ? 2.0/adpt.num : 0.25;
  r_0 = 1/(double)alpha*log((1 - y_0)/y_0)/(double)mcget_ncount();
  C = 1/(1 + log(y_0 + (1 - y_0)*exp(-r_0*mcget_ncount()))/(r_0*mcget_ncount()));
%}
TRACE
%{
  double thmin,thmax,phmin,phmax,theta,phi,E,v,r;
  double new_v;
  int i_E, i_xpos, i_xdiv;

  /* Randomly select a bin in the current distribution */
  r = rand01();
  adpt.idx = adapt_tree_search(adpt.atree, adpt.atree->total*r);
  if(adpt.idx >= adpt.num)
  {
    fprintf(stderr,
            "Hm, idx is %d, num is %d, r is %g, atree->total is %g\n",
            adpt.idx, (int)adpt.num, r, adpt.atree->total);
    adpt.idx = adpt.num - 1;
  }
  /* Now find the bin coordinates. */
  i_xdiv = adpt.idx % (int)N_xdiv;
  i_xpos = (adpt.idx / (int)N_xdiv) % (int)N_xpos;
  i_E = (adpt.idx / (int)N_xdiv) / (int)N_xpos;
  /* Compute the initial neutron parameters, selecting uniformly randomly
     within each bin dimension. */
  x = xmin + (i_xpos + rand01())*((xmax - xmin)/(double)N_xpos);
  y = ymin + rand01()*(ymax - ymin);
  z=0;
  thmin = atan2(-xw/2.0 - x, dist);
  thmax = atan2( xw/2.0 - x, dist);
  theta = thmin + (i_xdiv + rand01())*((thmax - thmin)/(double)N_xdiv);
  phmin = atan2(-yh/2.0 - y, dist);
  phmax = atan2( yh/2.0 - y, dist);
  phi = phmin + rand01()*(phmax - phmin);
  E = E0 - dE + (i_E + rand01())*(2.0*dE/(double)N_E);
  v = sqrt(E)*SE2V;
  vy = v*sin(phi);
  vx = v*cos(phi)*sin(theta);
  vz = v*cos(phi)*cos(theta);
  t = 0;
  /* Adjust neutron weight. */
  p = p_in;
  adpt.factor = y_0/(y_0 + (1 - y_0)*exp(-r_0*count));
  count++;
  p /= adpt.atree->v[adpt.idx]/(adpt.atree->total/adpt.num);
  p *= C*adpt.factor*(thmax - thmin)*(sin(phmax) - sin(phmin));
  /* Update distribution, assuming absorbtion. */
  if(adpt.n[adpt.idx] > 0)
    adpt.psi_tot -= adpt.psi[adpt.idx]/
      (adpt.n[adpt.idx]*(adpt.n[adpt.idx] + 1));
  adpt.n[adpt.idx]++;
  if(adpt.psi_tot != 0)
  {
    new_v = (1 - adpt.a_beta)*adpt.factor*adpt.psi[adpt.idx]/
                (adpt.n[adpt.idx]*adpt.psi_tot) +
            adpt.a_beta/adpt.num;
    adapt_tree_add(adpt.atree, adpt.idx, new_v - adpt.atree->v[adpt.idx]);
  }
  /* Remember initial neutron weight. */
  adpt.pi = p;
%}

FINALLY
%{
  double *p1 = NULL;
  int i;

  if(filename)
  {
    p1 = malloc(adpt.num*sizeof(double));
    if(!p1)
      fprintf(stderr, "Warning: Source_adapt: "
              "not enough memory to write distribution.\n");
  }
  if(p1)
  {
    for(i = 0; i < adpt.num; i++)
      p1[i] = adpt.atree->v[i]/adpt.atree->total;
    DETECTOR_OUT_1D("Adaptive source energy distribution",
                    "Energy [meV]",
                    "Probability",
                    "E", E0 - dE, E0 + dE, adpt.num,
                    NULL, p1, NULL, filename);
    free(p1);
  }
  adapt_tree_free(adpt.atree);
%}
MCDISPLAY
%{
  magnify("xy");
  multiline(5, (double)xmin, (double)ymin, 0.0,
               (double)xmax, (double)ymin, 0.0,
               (double)xmax, (double)ymax, 0.0,
               (double)xmin, (double)ymax, 0.0,
               (double)xmin, (double)ymin, 0.0);
%}

END
```

B.2 Simple components

B.2.1 Arm

```
/*******************************************************************
*
* McStas, version 1.0, released October 26, 1998
*         Maintained by Kristian Nielsen and Kim Lefmann,
*         Risoe National Laboratory, Roskilde, Denmark
*
* Component: Arm
*
* Written by: KL, KN, September 1997
*
* An arm does not actually do anything, it is just there to set
* up a new coordinate system.
*
* Input parameters:
*
* (none)
*
*******************************************************************/

DEFINE COMPONENT Arm
DEFINITION PARAMETERS ()
SETTING PARAMETERS ()
STATE PARAMETERS ()
TRACE
%{
%}

MCDISPLAY
%{
  /* A bit ugly; hard-coded dimensions. */
  magnify("");
  line(0,0,0,0.2,0,0);
  line(0,0,0,0,0.2,0);
  line(0,0,0,0,0,0.2);
%}

END
```

B.2.2 Slit

```
/*******************************************************************
*
* McStas, version 1.0, released October 26, 1998
*         Maintained by Kristian Nielsen and Kim Lefmann,
*         Risoe National Laboratory, Roskilde, Denmark
*
* Component: Slit
*
* Written by: KL, HMR  June 16, 1997
*
* A simple rectangular slit. No transmission around the slit is allowed.
*
* INPUT PARAMETERS
*
* xmin: Lower x bound (m)
* xmax: Upper x bound (m)
* ymin: Lower y bound (m)
* ymax: Upper y bound (m)
*
*******************************************************************/


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

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

END
```

B.2.6 Soller

```
/*******************************************************************************
*
* McStas, version 1.0, released October 26, 1998
*         Maintained by Kristian Nielsen and Kim Lefmann,
*         Risoe National Laboratory, Roskilde, Denmark
*
* Component: Soller
*
* Written by: KN, August 1998
*
* Soller collimator with rectangular opening and specified length. The
* transmission function is an average and does not utilize knowledge of the
* actual neutron trajectory.
* A zero divergence disables collimation (then the component works as a double slit).
*
* INPUT PARAMETERS:
*
* xmin:       (m)               Lower x bound on slits
* xmax:       (m)               Upper x bound on slits
* ymin:       (m)               Lower y bound on slits
* ymax:       (m)               Upper y bound on slits
* len:        (m)               Distance between slits
* divergence: (minutes of arc) Divergence angle (calculated as atan(d/len),
*                               where d is the blade spacing)
*
*******************************************************************************/


DEFINE COMPONENT Soller
DEFINITION PARAMETERS (xmin, xmax, ymin, ymax, len, divergence)
SETTING PARAMETERS ()
OUTPUT PARAMETERS (slope)
STATE PARAMETERS (x,y,z,vx,vy,vz,t,s1,s2,p)
DECLARE
%{
  double slope;
%}
INITIALIZE
%{
  slope = tan(MIN2RAD*divergence);
%}
TRACE
%{
    double phi, dt;

    PROP_Z0;
    if (x<xmin || x>xmax || y<ymin || y>ymax)
      ABSORB;
    dt = len/vz;
    PROP_DT(dt);
    if (x<xmin || x>xmax || y<ymin || y>ymax)
      ABSORB;

    if(slope > 0.0)
    {
      phi = fabs(vx/vz);
      if (phi > slope)
        ABSORB;
      else
        p *= 1.0 - phi/slope;
    }
%}

MCDISPLAY
%{
  double x;
  int i;

  magnify("xy");
  for(x = xmin, i = 0; i <= 3; i++, x += (xmax - xmin)/3.0)
    multiline(5, x, (double)ymin, 0.0, x, (double)ymax, 0.0,
              x, (double)ymax, (double)len, x, (double)ymin, (double)len,
              x, (double)ymin, 0.0);
  line(xmin, ymin, 0,   xmax, ymin, 0);
  line(xmin, ymax, 0,   xmax, ymax, 0);
  line(xmin, ymin, len, xmax, ymin, len);
  line(xmin, ymax, len, xmax, ymax, len);
%}

END
```

B.2.7 Filter

```
/*******************************************************************************
*
* McStas, version 1.0, released October 26, 1998
*         Maintained by Kristian Nielsen and Kim Lefmann,
*         Risoe National Laboratory, Roskilde, Denmark
*
* Component: Filter
*
* Written by: KL, KN, Sept. 14 1998
*
* Be-type filter defined by two identical rectangular opening apertures.
* The transmission is interpolated liniearly between the high- and low-energy
* transmissions beyond the upper and lower cut-off energies.
*
* INPUT PARAMETERS:
*
* xmin: Lower x bound (m)
* xmax: Upper x bound (m)
* ymin: Lower y bound (m)
* ymax: Upper y bound (m)
* len:  Distance between apertures (m)
* T0:   Transmittance of low energy neutrons (1)
* T1:   Transmittance of high energy neutrons (1)
* Emin: Lower cut-off energy (meV)
* Emax: Upper cut-off energy (meV)
*
*******************************************************************************/


DEFINE COMPONENT Filter
DEFINITION PARAMETERS (xmin, xmax, ymin, ymax, len, T0, T1, Emin, Emax)
SETTING PARAMETERS ()
STATE PARAMETERS (x,y,z,vx,vy,vz,t,s1,s2,p)
TRACE
%{
    double E;
    double dt;

    PROP_Z0;
    E=VS2E*(vx*vx+vy*vy+vz*vz);
    if (x<xmin || x>xmax || y<ymin || y>ymax)
      ABSORB;
    dt = len/vz;
    PROP_DT(dt);
    if (x<xmin || x>xmax || y<ymin || y>ymax)
      ABSORB;

    if(E>=Emax)
      if(T1==0)
        ABSORB;
      else
        p*=T1;
    else if(E<=Emin)
      if(T0==0)
        ABSORB;
      else
        p*=T0;
    else
      p*= T1+(T0-T1)*(Emax-E)/(Emax-Emin);
%}

MCDISPLAY
%{
  magnify("xy");
  multiline(5, (double)xmin, (double)ymin, 0.0,
               (double)xmax, (double)ymin, 0.0,
               (double)xmax, (double)ymax, 0.0,
               (double)xmin, (double)ymax, 0.0,
               (double)xmin, (double)ymin, 0.0);
  multiline(5, (double)xmin, (double)ymin, (double)len,
               (double)xmax, (double)ymin, (double)len,
               (double)xmax, (double)ymax, (double)len,
               (double)xmin, (double)ymax, (double)len,
               (double)xmin, (double)ymin, (double)len);
  line(xmin, ymin, 0.0, xmin, ymin, len);
  line(xmax, ymin, 0.0, xmax, ymin, len);
  line(xmin, ymax, 0.0, xmin, ymax, len);
  line(xmax, ymax, 0.0, xmax, ymax, len);
%}
END
```

B.3 Beam optical components

B.3.3 Guide

```
/*******************************************************************************
*
* McStas, version 1.0, released October 26, 1998
*         Maintained by Kristian Nielsen and Kim Lefmann,
*         Risoe National Laboratory, Roskilde, Denmark
*
* Component: Guide.
*
* Written by: KN, September 2 1998
* Modified by: KL, October 6, 1998
*
* Models a rectangular guide tube centered on the Z axis. The entrance lies
* in the X-Y plane.
* For details on the geometry calculation see the description in the McStas
* reference manual.
*
* INPUT PARAMETERS:
*
* w1:      (m)    Width at the guide entry
* h1:      (m)    Height at the guide entry
* w2:      (m)    Width at the guide exit
* h2:      (m)    Height at the guide exit
* l:       (m)    length of guide
* R0:      (1)    Low-angle reflectivity
* Qc:      (AA-1) Critical scattering vector
* alpha:   (AA)   Slope of reflectivity
* m:       (1)    m-value of material. Zero means completely absorbing.
* W:       (AA-1) Width of supermirror cut-off
*
* Example values: m=4 Qc=0.02 W=1/300 alpha=6.49 R0=1
*******************************************************************************/

DEFINE COMPONENT Guide
DEFINITION PARAMETERS (w1, h1, w2, h2, l, R0, Qc, alpha, m, W)
SETTING PARAMETERS ()
STATE PARAMETERS (x,y,z,vx,vy,vz,t,s1,s2,p)

TRACE
%{
  double t1,t2;                                 /* Intersection times. */
  double av,ah,bv,bh,cv1,cv2,ch1,ch2,d;         /* Intermediate values */
  double vdotn_v1,vdotn_v2,vdotn_h1,vdotn_h2;   /* Dot products. */
  int i;                                        /* Which mirror hit? */
  double q;                                     /* Q [1/AA] of reflection */
  double vlen2,nlen2;                           /* Vector lengths squared */

  /* ToDo: These could be precalculated. */
  double ww = .5*(w2 - w1), hh = .5*(h2 - h1);
  double whalf = .5*w1, hhalf = .5*h1;
  double lwhalf = l*whalf, lhhalf = l*hhalf;

  /* Propagate neutron to guide entrance. */
  PROP_Z0;
```

```
if(x <= -whalf || x >= whalf || y <= -hhalf || y >= hhalf)
  ABSORB;
for(;;)
{
  /* Compute the dot products of v and n for the four mirrors. */
  av = l*vx; bv = ww*vz;
  ah = l*vy; bh = hh*vz;
  vdotn_v1 = bv + av;         /* Left vertical */
  vdotn_v2 = bv - av;         /* Right vertical */
  vdotn_h1 = bh + ah;         /* Lower horizontal */
  vdotn_h2 = bh - ah;         /* Upper horizontal */
  /* Compute the dot products of (O - r) and n as c1+c2 and c1-c2 */
  cv1 = -whalf*l - z*ww; cv2 = x*l;
  ch1 = -hhalf*l - z*hh; ch2 = y*l;
  /* Compute intersection times. */
  t1 = (l - z)/vz;
  i = 0;
  if(vdotn_v1 < 0 && (t2 = (cv1 - cv2)/vdotn_v1) < t1)
  {
    t1 = t2;
    i = 1;
  }
  if(vdotn_v2 < 0 && (t2 = (cv1 + cv2)/vdotn_v2) < t1)
  {
    t1 = t2;
    i = 2;
  }
  if(vdotn_h1 < 0 && (t2 = (ch1 - ch2)/vdotn_h1) < t1)
  {
    t1 = t2;
    i = 3;
  }
  if(vdotn_h2 < 0 && (t2 = (ch1 + ch2)/vdotn_h2) < t1)
  {
    t1 = t2;
    i = 4;
  }
  if(i == 0)
    break;                    /* Neutron left guide. */
  PROP_DT(t1);
  switch(i)
  {
    case 1:                   /* Left vertical mirror */
      nlen2 = l*l + ww*ww;
      q = V2Q*(-2)*vdotn_v1/sqrt(nlen2);
      d = 2*vdotn_v1/nlen2;
      vx = vx - d*l;
      vz = vz - d*ww;
      break;
    case 2:                   /* Right vertical mirror */
      nlen2 = l*l + ww*ww;
      q = V2Q*(-2)*vdotn_v2/sqrt(nlen2);
      d = 2*vdotn_v2/nlen2;
      vx = vx + d*l;
      vz = vz - d*ww;
      break;
      case 3:                   /* Lower horizontal mirror */
        nlen2 = l*l + hh*hh;
        q = V2Q*(-2)*vdotn_h1/sqrt(nlen2);
        d = 2*vdotn_h1/nlen2;
        vy = vy - d*l;
        vz = vz - d*hh;
        break;
      case 4:                   /* Upper horizontal mirror */
        nlen2 = l*l + hh*hh;
        q = V2Q*(-2)*vdotn_h2/sqrt(nlen2);
        d = 2*vdotn_h2/nlen2;
        vy = vy + d*l;
        vz = vz - d*hh;
        break;
    }
    /* Now compute reflectivity. */
    if(m == 0)
      ABSORB;
    if(q > Qc)
    {
      double arg = (q-m*Qc)/W;
      if(arg < 10)
        p *= .5*(1-tanh(arg))*(1-alpha*(q-Qc));
      else
        ABSORB;                              /* Cutoff ~ 1E-10 */
    }
    p *= R0;
  }
%}

MCDISPLAY
%{
  double x;
  int i;

  magnify("xy");
  multiline(5,
            -w1/2.0, -h1/2.0, 0.0,
             w1/2.0, -h1/2.0, 0.0,
             w1/2.0,  h1/2.0, 0.0,
            -w1/2.0,  h1/2.0, 0.0,
            -w1/2.0, -h1/2.0, 0.0);
  multiline(5,
            -w2/2.0, -h2/2.0, (double)l,
             w2/2.0, -h2/2.0, (double)l,
             w2/2.0,  h2/2.0, (double)l,
            -w2/2.0,  h2/2.0, (double)l,
            -w2/2.0, -h2/2.0, (double)l);
  line(-w1/2.0, -h1/2.0, 0, -w2/2.0, -h2/2.0, (double)l);
  line( w1/2.0, -h1/2.0, 0,  w2/2.0, -h2/2.0, (double)l);
  line( w1/2.0,  h1/2.0, 0,  w2/2.0,  h2/2.0, (double)l);
  line(-w1/2.0,  h1/2.0, 0, -w2/2.0,  h2/2.0, (double)l);
%}

END
```

B.3.4 Channeled_Guide

```
/*******************************************************************************
*
* McStas, the neutron ray-tracing package
*         Maintained by Kristian Nielsen and Kim Lefmann,
*         Copyright 2000 Risoe National Laboratory, Roskilde, Denmark
*
* Component: Channeled_guide.
*
* Written by: KN, 1999
*
* Models a rectangular guide tube centered on the Z axis. The entrance lies
* in the X-Y plane.
* The guide may be tapered, and may have vertical subdivisions (used for
* bender devices).
*
* INPUT PARAMETERS:
*
* w1:      (m)    Width at the guide entry
* h1:      (m)    Height at the guide entry
* w2:      (m)    Width at the guide exit
* h2:      (m)    Height at the guide exit
* l:       (m)    Length of guide
* d:       (m)    Thickness of subdividing walls
* k:       (1)    Number of channels in the guide (>= 1)
* R0:      (1)    Low-angle reflectivity
* Qcx:     (AA-1) Critical scattering vector for left and right vertical
*                 mirrors in each channel
* Qcy:     (AA-1) Critical scattering vector for top and bottom mirrors
* alphax:  (AA)   Slope of reflectivity for left and right vertical
*                 mirrors in each channel
* alphay:  (AA)   Slope of reflectivity for top and bottom mirrors
* mx:      (1)    m-value of material for left and right vertical mirrors
*                 in each channel. Zero means completely absorbing.
* my:      (1)    m-value of material for top and bottom mirrors. Zero
*                 means completely absorbing.
* W:       (AA-1) Width of supermirror cut-off for all mirrors
*
* Example values: mx=4 my=2 Qcx=Qcy=0.02 W=1/300 alphax=alphay=6.49 R0=1
*******************************************************************************/

DEFINE COMPONENT Channeled_guide
DEFINITION PARAMETERS (w1, h1, w2, h2, d, k, l,
                       R0, Qcx,  Qcy, alphax, alphay, mx, my, W)
SETTING PARAMETERS ()
OUTPUT PARAMETERS (w1c,w2c,ww,hh,whalf,hhalf,lwhalf,lhhalf)
STATE PARAMETERS (x,y,z,vx,vy,vz,t,s1,s2,p)

DECLARE
%{
  double w1c;
  double w2c;
  double ww, hh;
  double whalf, hhalf;
  double lwhalf, lhhalf;
%}
```

```
INITIALIZE
%{
  w1c = (w1 + d)/(double)k;
  w2c = (w2 + d)/(double)k;
  ww = .5*(w2c - w1c);
  hh = .5*(h2 - h1);
  whalf = .5*(w1c - d);
  hhalf = .5*h1;
  lwhalf = l*whalf;
  lhhalf = l*hhalf;
%}

TRACE
%{
  double t1,t2;                               /* Intersection times. */
  double av,ah,bv,bh,cv1,cv2,ch1,ch2,dd;      /* Intermediate values */
  double vdotn_v1,vdotn_v2,vdotn_h1,vdotn_h2;  /* Dot products. */
  int i;                                      /* Which mirror hit? */
  double q;                                   /* Q [1/AA] of reflection */
  double vlen2,nlen2;                         /* Vector lengths squared */
  double edge;
  double hadj;                                /* Channel displacement */

  /* Propagate neutron to guide entrance. */
  PROP_Z0;
  if(x <= w1/-2.0 || x >= w1/2.0 || y <= -hhalf || y >= hhalf)
    ABSORB;
  /* Shift origin to center of channel hit (absorb if hit dividing walls) */
  x += w1/2.0;
  edge = floor(x/w1c)*w1c;
  if(x - edge > w1c - d)
  {
    x -= w1/2.0; /* Re-adjust origin */
    ABSORB;
  }
  x -= (edge + (w1c - d)/2.0);
  hadj = edge + (w1c - d)/2.0 - w1/2.0;
  for(;;)
  {
    /* Compute the dot products of v and n for the four mirrors. */
    av = l*vx; bv = ww*vz;
    ah = l*vy; bh = hh*vz;
    vdotn_v1 = bv + av;         /* Left vertical */
    vdotn_v2 = bv - av;         /* Right vertical */
    vdotn_h1 = bh + ah;         /* Lower horizontal */
    vdotn_h2 = bh - ah;         /* Upper horizontal */
    /* Compute the dot products of (O - r) and n as c1+c2 and c1-c2 */
    cv1 = -whalf*l - z*ww; cv2 = x*l;
    ch1 = -hhalf*l - z*hh; ch2 = y*l;
    /* Compute intersection times. */
    t1 = (l - z)/vz;
    i = 0;
    if(vdotn_v1 < 0 && (t2 = (cv1 - cv2)/vdotn_v1) < t1)
    {
      t1 = t2;
```

```
        i = 1;
      }
      if(vdotn_v2 < 0 && (t2 = (cv1 + cv2)/vdotn_v2) < t1)
      {
        t1 = t2;
        i = 2;
      }
      if(vdotn_h1 < 0 && (t2 = (ch1 - ch2)/vdotn_h1) < t1)
      {
        t1 = t2;
        i = 3;
      }
      if(vdotn_h2 < 0 && (t2 = (ch1 + ch2)/vdotn_h2) < t1)
      {
        t1 = t2;
        i = 4;
      }
      if(i == 0)
        break;                    /* Neutron left guide. */
      PROP_DT(t1);
      switch(i)
      {
        case 1:                   /* Left vertical mirror */
          nlen2 = l*l + ww*ww;
          q = V2Q*(-2)*vdotn_v1/sqrt(nlen2);
          dd = 2*vdotn_v1/nlen2;
          vx = vx - dd*l;
          vz = vz - dd*ww;
          break;
        case 2:                   /* Right vertical mirror */
          nlen2 = l*l + ww*ww;
          q = V2Q*(-2)*vdotn_v2/sqrt(nlen2);
          dd = 2*vdotn_v2/nlen2;
          vx = vx + dd*l;
          vz = vz - dd*ww;
          break;
        case 3:                   /* Lower horizontal mirror */
          nlen2 = l*l + hh*hh;
          q = V2Q*(-2)*vdotn_h1/sqrt(nlen2);
          dd = 2*vdotn_h1/nlen2;
          vy = vy - dd*l;
          vz = vz - dd*hh;
          break;
        case 4:                   /* Upper horizontal mirror */
          nlen2 = l*l + hh*hh;
          q = V2Q*(-2)*vdotn_h2/sqrt(nlen2);
          dd = 2*vdotn_h2/nlen2;
          vy = vy + dd*l;
          vz = vz - dd*hh;
          break;
      }
      /* Now compute reflectivity. */
      if((i <= 2 && mx == 0) || (i > 2 && my == 0))
      {
        x += hadj; /* Re-adjust origin */
        ABSORB;
```

```
    }
    if((i <= 2 && q > Qcx) || (i > 2 && q > Qcy))
    {
      if (i <= 2)
      {
        double arg = (q - mx*Qcx)/W;
        if(arg < 10)
          p *= .5*(1-tanh(arg))*(1-alphax*(q-Qcx));
        else
        {
          x += hadj; /* Re-adjust origin */
          ABSORB;                              /* Cutoff ~ 1E-10 */
        }
      } else {
        double arg = (q - my*Qcy)/W;
        if(arg < 10)
          p *= .5*(1-tanh(arg))*(1-alphay*(q-Qcy));
        else
        {
          x += hadj; /* Re-adjust origin */
          ABSORB;                              /* Cutoff ~ 1E-10 */
        }
      }
    }
    p *= R0;
    x += hadj; SCATTER; x -= hadj;
  }
  x += hadj; /* Re-adjust origin */
%}

MCDISPLAY
%{
  double x;
  int i;

  magnify("xy");
  for(i = 0; i < k; i++)
  {
    multiline(5,
              i*w1c - w1/2.0, -h1/2.0, 0.0,
              i*w2c - w2/2.0, -h2/2.0, (double)l,
              i*w2c - w2/2.0,  h2/2.0, (double)l,
              i*w1c - w1/2.0,  h1/2.0, 0.0,
              i*w1c - w1/2.0, -h1/2.0, 0.0);
    multiline(5,
              (i+1)*w1c - d - w1/2.0, -h1/2.0, 0.0,
              (i+1)*w2c - d - w2/2.0, -h2/2.0, (double)l,
              (i+1)*w2c - d - w2/2.0,  h2/2.0, (double)l,
              (i+1)*w1c - d - w1/2.0,  h1/2.0, 0.0,
              (i+1)*w1c - d - w1/2.0, -h1/2.0, 0.0);
  }
  line(-w1/2.0, -h1/2.0, 0.0, w1/2.0, -h1/2.0, 0.0);
  line(-w2/2.0, -h2/2.0, (double)l, w2/2.0, -h2/2.0, (double)l);
%}

END
```

B.4 Chopper-like components

B.4.1 V_selector.comp

```
/*******************************************************************************
*
* McStas, version 1.1, released ?
*         Maintained by Kristian Nielsen and Kim Lefmann,
*         Risoe National Laboratory, Roskilde, Denmark
*
* Component: V_selector
*
* Written by:  KL, Nov 25, 1998
* Last change: KL, Jan 22, 1999
*
* Velocity selector consisting of rotating Soller-like blades
* defining a helically twisted passage.
* Geometry defined by two identical, centered apertures at 12 o'clock
* position, Origo is at the centre of the selector.
* Transmission is analytical assuming a continuous source.
*
* INPUT PARAMETERS:
*
* width:     (m)   Width of aperture
* height:    (m)   Height of aperture
* l0:        (m)   Distance between apertures
* r0:        (m)   Height from aperture centre to rotation axis
* phi:       (rad) Twist angle along the cylinder
* l1:        (m)   Length of cylinder (less than l0)
* tb:        (m)   Thickness of blades
* rot:       (rpm) Cylinder rotation speed, counter-clockwise
* nb:        (1)   Number of Soller blades
*
*******************************************************************************/


DEFINE COMPONENT V_selector
DEFINITION PARAMETERS (width, height, l0, r0, phi, l1, tb, rot, nb)
SETTING PARAMETERS ()
STATE PARAMETERS (x,y,z,vx,vy,vz,t,s1,s2,p)
DECLARE
%{
  double RPM2OM, omega, phi_rad, dt0, dt1, r_i, r_f, r_mean, theta_i, theta_f, A, d_s_phi;
%}
INITIALIZE
%{
    RPM2OM = 2*PI/60.0;
    omega=rot*RPM2OM;
    phi_rad = phi*DEG2RAD;
%}
TRACE
%{
    if (vz == 0)
      ABSORB;
    dt1= (-l0/2.0 - z)/vz;
    PROP_DT(dt1); /* Propagate to the entry aperture */
    if (x<(-width/2.0) || x>(width/2.0) || y<(-height/2.0) || y>(height/2.0))
```

```
      ABSORB;

    dt0 = (l0-l1)/(2.0*vz); /* Propagate to the cylinder start */
    PROP_DT(dt0);
    r_i = sqrt(x*x+(y+r0)*(y+r0));
    theta_i = atan2(x,y+r0);

    dt1 = l1/vz; /* Propagate along the cylinder length */
    PROP_DT(dt1);
    r_f = sqrt(x*x+(y+r0)*(y+r0));
    theta_f = atan2(x,y+r0);

    dt0 = (l0-l1)/(2.0*vz); /* Propagate to the exit aperture */
    PROP_DT(dt0);
    if (x<(-width/2.0) || x>(width/2.0) || y<(-height/2.0) || y>(height/2.0))
      ABSORB;

    /* Calculate analytical transmission assuming continuous source */

    r_mean = (r_i + r_f)/2.0;          /* Approximation using mean radius */
    d_s_phi = theta_f-theta_i;
    A = nb/(2*PI)*( tb/r_mean + fabs(phi_rad+d_s_phi-omega*l1/vz) );
    if (A >= 1)
      ABSORB;
    p*= (1-A);
%}
END
```

B.4.2 Chopper.comp

```
/*******************************************************************************************
*
* Component: Chopper
*
* Written by: Philipp Bernhardt, Januar 22 1999
*
* Models a disc chopper with n identical slits, which are symmetrically disposed on the disc.
*
* INPUT PARAMETERS:
*
* w:       (m)      Width of the slits at the bottom side
* R:       (m)      Radius of the disc
* f:       (rad/s)  angular frequency of the Chopper (algebraic sign defines the direction
*                   of rotation
* n:       (1)     Number of slits
* pha:     (s)      Phase
*
* Example values: w=0.05 R=0.5 f=2500 n=3 pha=0
********************************************************************************************/

DEFINE COMPONENT Chopper
DEFINITION PARAMETERS (w, R, f, n, pha)
SETTING PARAMETERS ()
OUTPUT PARAMETERS (Tg, To)
STATE PARAMETERS (x, y, z, vx, vy, vz, t, s1, s2, p)

DECLARE
 %{
      double Tg,To;
 %}

INITIALIZE
 %{
      /* time between two pulses */
      Tg=2*PI/fabs(f)/n;

      /* how long can neutrons pass the Chopper at a single point */
      To=2*atan(w/R/2.0)/fabs(f);
 %}

TRACE
 %{
        double toff;

        PROP_Z0;

        toff=fabs(t-atan2(x,y+R)/f-pha)+To/2.0;

        /* does neutron hit the slit? */
        if (fmod(toff,Tg)>To)
           ABSORB;

 %}

END
```

B.5 Detectors and monitors

B.5.1 Monitor

```
/*******************************************************************
*
* McStas, version 1.0, released October 26, 1998
*         Maintained by Kristian Nielsen and Kim Lefmann,
*         Risoe National Laboratory, Roskilde, Denmark
*
* Component: Monitor
*
* Written by: KL, October 4, 1997
*
* Sums neutrons (0th, 1st, and 2nd moment of p) flying through
* the rectangular monitor opening. May also be used as detector.
*
* INPUT PARAMETERS:
*
* xmin: Lower x bound of opening
* xmax: Upper x bound of opening
* ymin: Lower y bound of opening
* ymax: Upper y bound of opening
*
* OUTPUT PARAMETERS:
*
* Nsum:  Number of neutron hits
* psum:  Sum of neutron weights
* p2sum: 2nd moment of neutron weights
*
*******************************************************************/

DEFINE COMPONENT Monitor
DEFINITION PARAMETERS (xmin, xmax, ymin, ymax)
SETTING PARAMETERS ()
OUTPUT PARAMETERS (Nsum, psum, p2sum)
STATE PARAMETERS (x,y,z,vx,vy,vz,t,s1,s2,p)
DECLARE
  %{
    int Nsum;
    double psum, p2sum;
  %}
INITIALIZE
  %{
    psum = 0;
    p2sum = 0;
    Nsum = 0;
  %}
TRACE
  %{
    PROP_Z0;
    if (x>xmin && x<xmax && y>ymin && y<ymax)
    {
      Nsum++;
      psum += p;
      p2sum += p*p;
    }
  %}
FINALLY
  %{
    DETECTOR_OUT_0D("Single monitor", Nsum, psum, p2sum);
  %}

MCDISPLAY
%{
  magnify("xy");
  multiline(5, (double)xmin, (double)ymin, 0.0,
               (double)xmax, (double)ymin, 0.0,
               (double)xmax, (double)ymax, 0.0,
               (double)xmin, (double)ymax, 0.0,
               (double)xmin, (double)ymin, 0.0);
%}

END
```

B.5.3 PSD_monitor

```
/*******************************************************************************
*
* McStas, version 1.0, released October 26, 1998
*         Maintained by Kristian Nielsen and Kim Lefmann,
*         Risoe National Laboratory, Roskilde, Denmark
*
* Component: PSD_monitor
*
* Written by: KL,  Feb 3, 1998
*
* An (n times m) pixel PSD monitor. This component may also be used as a beam
* detector.
*
* INPUT PARAMETERS:
*
* xmin:     Lower x bound of detector opening (m)
* xmax:     Upper x bound of detector opening (m)
* ymin:     Lower y bound of detector opening (m)
* ymax:     Upper y bound of detector opening (m)
* nx:       Number of pixel columns (1)
* ny:       Number of pixel rows (1)
* filename: Name of file in which to store the detector image (text)
*
* OUTPUT PARAMETERS:
*
* PSD_N:    Array of neutron counts
* PSD_p:    Array of neutron weight counts
* PSD_p2:   Array of second moments
*
*******************************************************************************/


DEFINE COMPONENT PSD_monitor
DEFINITION PARAMETERS (xmin, xmax, ymin, ymax, nx, ny, filename)
SETTING PARAMETERS ()
OUTPUT PARAMETERS (PSD_N, PSD_p, PSD_p2)
STATE PARAMETERS (x,y,z,vx,vy,vz,t,s1,s2,p)
DECLARE
  %{
    int PSD_N[nx][ny];
    double PSD_p[nx][ny];
    double PSD_p2[nx][ny];
  %}
INITIALIZE
  %{
    int i,j;

    for (i=0; i<nx; i++)
     for (j=0; j<ny; j++)
     {
      PSD_N[i][j] = 0;
      PSD_p[i][j] = 0;
      PSD_p2[i][j] = 0;
     }
  %}
```

```
TRACE
  %{
    int i,j;

    PROP_Z0;
    if (x>xmin && x<xmax && y>ymin && y<ymax)
    {
      i = floor((x - xmin)*nx/(xmax - xmin));
      j = floor((y - ymin)*ny/(ymax - ymin));
      PSD_N[i][j]++;
      PSD_p[i][j] += p;
      PSD_p2[i][j] += p*p;
    }
  %}
FINALLY
  %{
    DETECTOR_OUT_2D(
        "PSD monitor",
        "X position [cm]",
        "Y position [cm]",
        xmin*100.0, xmax*100.0, ymin*100.0, ymax*100.0,
        nx, ny,
        &PSD_N[0][0],&PSD_p[0][0],&PSD_p2[0][0],
        filename);
  %}

MCDISPLAY
%{
  magnify("xy");
  multiline(5, (double)xmin, (double)ymin, 0.0,
               (double)xmax, (double)ymin, 0.0,
               (double)xmax, (double)ymax, 0.0,
               (double)xmin, (double)ymax, 0.0,
               (double)xmin, (double)ymin, 0.0);
%}

END
```

B.5.4 PSD_monitor_4PI

```
/*******************************************************************
*
* McStas, version 1.0, released October 26, 1998
*         Maintained by Kristian Nielsen and Kim Lefmann,
*         Risoe National Laboratory, Roskilde, Denmark
*
* Component: PSD_monitor_4PI
*
* Written by: KL and KN, April 17, 1998
*
* An (n times m) pixel spherical PSD monitor using a cylindrical projection.
* Mostly for test and debugging purposes.
* INPUT PARAMETERS:
*
* radius:   Radius of detector (m)
* nx:       Number of pixel columns (1)
* ny:       Number of pixel rows (1)
* filename: Name of file in which to store the detector image (text)
*
* OUTPUT PARAMETERS:
*
* PSD_N:    Array of neutron counts
* PSD_p:    Array of neutron weight counts
* PSD_p2:   Array of second moments
*
*******************************************************************/


DEFINE COMPONENT PSD_monitor_4PI
DEFINITION PARAMETERS (radius, nx, ny, filename)
SETTING PARAMETERS ()
OUTPUT PARAMETERS (PSD_N, PSD_p, PSD_p2)
STATE PARAMETERS (x,y,z,vx,vy,vz,t,s1,s2,p)
DECLARE
%{
  int PSD_N[nx][ny];
  double PSD_p[nx][ny];
  double PSD_p2[nx][ny];
%}
INITIALIZE
%{
  int i,j;

  for (i=0; i<nx; i++)
    for (j=0; j<ny; j++)
    {
      PSD_N[i][j] = 0;
      PSD_p[i][j] = 0;
      PSD_p2[i][j] = 0;
    }
%}
TRACE
%{
  double t0, t1, phi;
  int i,j;
  if(sphere_intersect(&t0, &t1, x, y, z, vx, vy, vz, radius) && t1 > 0)
  {
    if(t0 < 0)
      t0 = t1;
    /* t0 is now time of intersection with the sphere. */
    PROP_DT(t0);
    phi = atan2(z,x);
    i = floor(nx*(phi/(2*PI) + 0.5));
    if(i == nx)
      i--;                      /* Special case for phi = PI. */
    else if(i < 0)
      y = 0;
    j = floor(ny*(y/(2*radius) + 0.5));
    if(j == ny)
      j--;                      /* Special case for y = radius. */
    else if(j < 0)
      j = 0;
    PSD_N[i][j]++;
    PSD_p[i][j] += p;
    PSD_p2[i][j] += p*p;
  }
%}

FINALLY
%{
  DETECTOR_OUT_2D(
    "4PI PSD monitor",
    "Longitude [deg]",
    "Lattitude [deg]",
    -180, 180, -90, 90,
    nx, ny,
    &PSD_N[0][0],&PSD_p[0][0],&PSD_p2[0][0],
    filename);
%}

MCDISPLAY
%{
  magnify("");
  circle("xy",0,0,0,radius);
  circle("xz",0,0,0,radius);
  circle("yz",0,0,0,radius);
%}

END
```

B.5.6 TOF_monitor

```
/*******************************************************************
*
* McStas, version 1.0, released October 26, 1998
*         Maintained by Kristian Nielsen and Kim Lefmann,
*         Risoe National Laboratory, Roskilde, Denmark
*
* Component: TOF-monitor
*
* Written by:  KN, M. Hagan, August 1998
* Modified by: KL, October 7, 1998
*
* Rectangular Time-of-flight monitor.
*
* INPUT PARAMETERS:
*
* xmin:     Lower x bound of detector opening (m)
* xmax:     Upper x bound of detector opening (m)
* ymin:     Lower y bound of detector opening (m)
* ymax:     Upper y bound of detector opening (m)
* nchan:    Number of time bins (1)
* dt:       Length of each time bin (mu-s)
* filename: Name of file in which to store the detector image (text)
*
* OUTPUT PARAMETERS:
*
* TOF_N:    Array of neutron counts
* TOF_p:    Array of neutron weight counts
* TOF_p2:   Array of second moments
*
*******************************************************************/

DEFINE COMPONENT TOF_monitor
DEFINITION PARAMETERS (xmin, xmax, ymin, ymax, nchan, dt, filename)
SETTING PARAMETERS ()
OUTPUT PARAMETERS (TOF_N, TOF_p, TOF_p2)
STATE PARAMETERS (x,y,z,vx,vy,vz,t,s1,s2,p)
DECLARE
  %{
    int TOF_N[nchan];
    double TOF_p[nchan];
    double TOF_p2[nchan];
  %}
INITIALIZE
  %{
    int i;

    for (i=0; i<nchan; i++)
    {
      TOF_N[i] = 0;
      TOF_p[i] = 0;
      TOF_p2[i] = 0;
    }
  %}
TRACE
  %{
    int i;

    PROP_Z0;
    if (x>xmin && x<xmax && y>ymin && y<ymax)
    {
      i = floor(1E6*t/dt);              /* Bin number */
      if(i >= nchan) i = nchan - 1;
      if(i < 0)
      {
        printf("FATAL ERROR: negative time-of-flight.\n");
        exit(1);
      }
      TOF_N[i]++;
      TOF_p[i] += p;
      TOF_p2[i] += p*p;
    }
  %}
FINALLY
  %{
    DETECTOR_OUT_1D(
        "Time-of-flight monitor",
        "Time-of-flight [\\gms]",
        "Intensity",
        "t", 0.0, nchan*(double)dt, nchan,
        &TOF_N[0],&TOF_p[0],&TOF_p2[0],
        filename);
  %}

MCDISPLAY
%{
  magnify("xy");
  multiline(5, (double)xmin, (double)ymin, 0.0,
               (double)xmax, (double)ymin, 0.0,
               (double)xmax, (double)ymax, 0.0,
               (double)xmin, (double)ymax, 0.0,
               (double)xmin, (double)ymin, 0.0);
%}

END
```

B.5.7 E_monitor

```
/*******************************************************************
*
* McStas, version 1.0, released October 26, 1998
*         Maintained by Kristian Nielsen and Kim Lefmann,
*         Risoe National Laboratory, Roskilde, Denmark
*
* Component: E_monitor
*
* Written by:  KN,KL,  April 20, 1998
* Modified by: KL, Octorber 7, 1998
*
* A square single monitor that measures the energy of the incoming neutrons.
*
* INPUT PARAMETERS:
*
* xmin:     Lower x bound of detector opening (m)
* xmax:     Upper x bound of detector opening (m)
* ymin:     Lower y bound of detector opening (m)
* ymax:     Upper y bound of detector opening (m)
* Emin:     Minimum energy to detect (meV)
* Emax:     Maximum energy to detect (meV)
* nchan:    Number of energy channels (1)
* filename: Name of file in which to store the detector image (text)
*
* OUTPUT PARAMETERS:
*
* E_N:      Array of neutron counts
* E_p:      Array of neutron weight counts
* E_p2:     Array of second moments
*
*******************************************************************/

DEFINE COMPONENT E_monitor
DEFINITION PARAMETERS (xmin, xmax, ymin, ymax, Emin, Emax, nchan, filename)
SETTING PARAMETERS ()
OUTPUT PARAMETERS (E_N, E_p, E_p2)
STATE PARAMETERS (x,y,z,vx,vy,vz,t,s1,s2,p)
DECLARE
  %{
    int E_N[nchan];
    double E_p[nchan], E_p2[nchan];
  %}
INITIALIZE
  %{
    int i;

    for (i=0; i<nchan; i++)
    {
      E_N[i] = 0;
      E_p[i] = 0;
      E_p2[i] = 0;
    }
  %}
TRACE
  %{
    int i;
    double E;

    PROP_Z0;
    if (x>xmin && x<xmax && y>ymin && y<ymax)
    {
      E = VS2E*(vx*vx + vy*vy + vz*vz);

      i = floor((E-Emin)*nchan/(Emax-Emin));
      if(i >= 0 && i < nchan)
      {
        E_N[i]++;
        E_p[i] += p;
        E_p2[i] += p*p;
      }
    }
  %}
FINALLY
  %{
    DETECTOR_OUT_1D(
        "Energy monitor",
        "Energy [meV]",
        "Intensity",
        "E", Emin, Emax, nchan,
        &E_N[0],&E_p[0],&E_p2[0],
        filename);
  %}

MCDISPLAY
%{
  magnify("xy");
  multiline(5, (double)xmin, (double)ymin, 0.0,
               (double)xmax, (double)ymin, 0.0,
               (double)xmax, (double)ymax, 0.0,
               (double)xmin, (double)ymax, 0.0,
               (double)xmin, (double)ymin, 0.0);
%}

END
```

B.5.12 Res_monitor

```
/*******************************************************************************
*
* McStas, the neutron ray-tracing package
*         Maintained by Kristian Nielsen and Kim Lefmann,
*         Copyright 1977-2000 Risoe National Laboratory, Roskilde, Denmark
*
* Component: Res_monitor
*
* Written by: KN 1999
*
* A single detector/monitor, used together with the Res_sample component to
* compute instrument resolution functions. Outputs a list of neutron
* scattering events in the sample along with their intensities in the
* detector. The output file may be analyzed with the mcresplot front-end.
*
* INPUT PARAMETERS:
*
* xmin:            Lower x bound of detector opening (m)
* xmax:            Upper x bound of detector opening (m)
* ymin:            Lower y bound of detector opening (m)
* ymax:            Upper y bound of detector opening (m)
* filename:        Name of output file (string)
* res_sample_comp: Name of Res_sample component in the instrument definition
*
* OUTPUT PARAMETERS:
*
* Nsum:  Number of neutron hits
* psum:  Sum of neutron weights
* p2sum: 2nd moment of neutron weights
*
*******************************************************************************/

DEFINE COMPONENT Res_monitor
DEFINITION PARAMETERS (xmin, xmax, ymin, ymax, filename, res_sample_comp)
SETTING PARAMETERS ()
OUTPUT PARAMETERS (Nsum, psum, p2sum, file)
STATE PARAMETERS (x,y,z,vx,vy,vz,t,s1,s2,p)
DECLARE
  %{
    int Nsum;
    double psum, p2sum;
    FILE *file;
  %}
INITIALIZE
  %{
    psum = 0;
    p2sum = 0;
    Nsum = 0;
    file = filename ? fopen(filename, "w") : 0;
    if(!file && filename)
      fprintf(stderr, "Warning: could not open output file '%s'\n", filename);
  %}
TRACE
  %{
    PROP_Z0;
    if (x>xmin && x<xmax && y>ymin && y<ymax)
    {
      Nsum++;
      psum += p;
      p2sum += p*p;
      /* Now fetch data from the Res_sample. */
      if(p != 0 && file)
      {
        struct Res_sample_struct *s =
          &(MC_GETPAR(res_sample_comp, res_struct));
        if(s->pi != 0)
          fprintf(file, "%g %g %g %g %g %g %g %g %g %g %g\n",
                  s->ki_x, s->ki_y, s->ki_z, s->kf_x, s->kf_y, s->kf_z,
                  s->rx, s->ry, s->rz, s->pi, p/s->pi);
        }
    }
  %}
FINALLY
  %{
    if(file)
      fclose(file);
    DETECTOR_OUT_0D("Single monitor", Nsum, psum, p2sum);
  %}

MCDISPLAY
%{
  magnify("xy");
  multiline(5, (double)xmin, (double)ymin, 0.0,
               (double)xmax, (double)ymin, 0.0,
               (double)xmax, (double)ymax, 0.0,
               (double)xmin, (double)ymax, 0.0,
               (double)xmin, (double)ymin, 0.0);
%}

END
```

B.5.13 Adapt_check

```
/*******************************************************************************
*
* McStas, the neutron ray-tracing Monte-Carlo software.
* Copyright(C) 1999,2000 Risoe National Laboratory.
*
* Component: Adapt_check
*
* Written by Kristian Nielsen 1999
*
* This components works together with the Source_adapt component, and
* is used to define the criteria for selecting which neutrons are
* considered "good" in the adaptive algorithm. The name of the
* associated Source_adapt component in the instrument definition is
* given as parameter. The component is special in that its position
* does not matter; all neutrons that have not been absorbed prior to
* the component are considered "good".
*
*
* INPUT PARAMETERS:
*
* source_comp:   The name of the Source_adapt component in the
*                instrument definition.
*
*******************************************************************************/

DEFINE COMPONENT Adapt_check
DEFINITION PARAMETERS (source_comp)
SETTING PARAMETERS ()
STATE PARAMETERS (x,y,z,vx,vy,vz,t,s1,s2,p)

TRACE
%{
  double new_v, psi;
  struct source_adapt *adpt = &(MC_GETPAR(source_comp, adpt));

  if(p == 0)
    ABSORB;
  psi = p/adpt->pi;
  adpt->psi[adpt->idx] += psi;
  adpt->psi_tot += psi/adpt->n[adpt->idx];
  new_v = (1 - adpt->a_beta)*adpt->factor*adpt->psi[adpt->idx]/
                (adpt->n[adpt->idx]*adpt->psi_tot) +
          adpt->a_beta/adpt->num;
  adapt_tree_add(adpt->atree, adpt->idx, new_v - adpt->atree->v[adpt->idx]);
%}

MCDISPLAY
%{
  magnify("");
%}

END
```

B.6 Crystals

B.6.1 Mosaic_simple

```
/*******************************************************************************
*
* McStas, version 1.2
*         Maintained by Kristian Nielsen and Kim Lefmann,
*         Copyright (C) Risoe National Laboratory 1999
*
* Component: Mosaic_simple
*
* Flat, infinitely thin mosaic crystal, useful as a monochromator or analyzer.
* The mosaic is isotropic gaussian, with a given FWHM perpendicular to the
* scattering vector.
* For an unrotated monochromator component, the crystal plane lies in the y-z
* plane (ie. parallel to the beam).
*
* INPUT PARAMETERS:
*
* zmin:    Lower z-bound of crystal (m)
* zmax:    Upper z-bound of crystal (m)
* ymin:    Lower y-bound of crystal (m)
* ymax:    Upper y-bound of crystal (m)
* mosaic:  Mosaic (FWHM) (arc minutes)
* R0:      Maximum reflectivity (1)
* Qx:      X coordinate of scattering vector (AA-1)
* Qy:      X coordinate of scattering vector (AA-1)
* Qz:      X coordinate of scattering vector (AA-1)
*
*******************************************************************************/

DEFINE COMPONENT Mosaic_simple
DEFINITION PARAMETERS (zmin, zmax, ymin, ymax, mosaic, R0, Qx, Qy, Qz)
SETTING PARAMETERS ()
OUTPUT PARAMETERS (X,W,Q,mos_rms)
STATE PARAMETERS (x,y,z,vx,vy,vz,t,s1,s2,p)

DECLARE
%{
  /* ToDo: Define these arrays only once for all instances. */
  /* Values for Gauss quadrature. Taken from Brice Carnahan, H. A. Luther and
     James O Wilkes, "Applied numerical methods", Wiley, 1996, page 103. */
  double X[] = {-0.987992518020485, 0.937273392400706, 0.848206583410427,
              0.724417731360170, 0.570972172608539, 0.394151347077563,
              0.201194093997435, 0, 0.201194093997435,
              0.394151347077563, 0.570972172608539, 0.724417731360170,
              0.848206583410427, 0.937273392400706, 0.987992518020485};
  double W[] = {0.030753241996117, 0.070366047488108, 0.107159220467172,
              0.139570677926154, 0.166269205816994, 0.186161000115562,
              0.198431485327111, 0.202578241925561, 0.198431485327111,
              0.186161000115562, 0.166269205816994, 0.139570677926154,
              0.107159220467172, 0.070366047488108, 0.030753241996117};
  double Q;                   /* Length of scattering vector */
  double q0ux, q0uy, q0uz;    /* Unit vector parrallel to nominal Q */
  double mos_rms;             /* root-mean-square of mosaic, in radians */
#define GAUSS(x,mean,rms) \
  (exp(-((x)-(mean))*((x)-(mean))/(2*(rms)*(rms)))/(sqrt(2*PI)*(rms)))
%}

INITIALIZE
%{
  Q = sqrt(Qx*Qx + Qy*Qy + Qz*Qz);
  q0ux = Qx/Q;
  q0uy = Qy/Q;
  q0uz = Qz/Q;
  mos_rms = MIN2RAD*mosaic/sqrt(8*log(2));
%}

TRACE
%{
  double y1,z1,t1,dt,kix,kiy,kiz,ratio,order,q0x,q0y,q0z,k,q0,theta;
  double bx,by,bz,kux,kuy,kuz,ax,ay,az,phi;
  double cos_2theta,k_sin_2theta,cos_phi,sin_phi,kfx,kfy,kfz,q_x,q_y,q_z;
  double delta,p_reflect,total,c1x,c1y,c1z,width,tmp;
  int i;

  if(vx != 0.0 && (dt = -x/vx) >= 0.0)
  {                             /* Moving towards crystal? */
    y1 = y + vy*dt;             /* Propagate to crystal plane */
    z1 = z + vz*dt;
    t1 = t + dt;
    if (z1>zmin && z1<zmax && y1>ymin && y1<ymax)
    {                           /* Intersect the crystal? */
      kix = V2K*vx;             /* Initial wave vector */
      kiy = V2K*vy;
      kiz = V2K*vz;
      /* Get reflection order and corresponding nominal scattering vector q0
         of correct length and direction. Only the order with the closest
         scattering vector is considered */
      ratio = -2*(kix*Qx + kiy*Qy + kiz*Qz)/(Q*Q);
      order = floor(ratio + .5);
      if(order == 0.0)
        order = ratio < 0 ? -1 : 1;
      /* Order will be negative when the neutron enters from the back, in
         which case the direction of Q0 is flipped. */
      if(order < 0)
        order = -order;
      /* Make sure the order is small enough to allow Bragg scattering at the
         given neutron wavelength */
      k = sqrt(kix*kix + kiy*kiy + kiz*kiz);
      kux = kix/k;              /* Unit vector along ki */
      kuy = kiy/k;
      kuz = kiz/k;
      if(order > 2*k/Q)
        order--;
      if(order > 0)             /* Bragg scattering possible? */
      {
        q0x = order*Qx;
        q0y = order*Qy;
        q0z = order*Qz;
        if(ratio < 0)
        {
```

```
  q0x = -q0x; q0y = -q0y; q0z = -q0z;
}
q0 = order*Q;
theta = asin(q0/(2*k)); /* Actual bragg angle */
/* Make MC choice: reflect or transmit? */
delta = asin(-(kux*q0x + kuy*q0y + kuz*q0z)/q0) - theta;
p_reflect = R0*exp(-delta*delta/(2*mos_rms*mos_rms));
if(rand01() < p_reflect)
{                       /* Reflect */
  cos_2theta = cos(2*theta);
  k_sin_2theta = k*sin(2*theta);
  /* Get unit normal to plane containing ki and most probable kf */
  vec_prod(bx, by, bz, kix, kiy, kiz, q0x, q0y, q0z);
  NORM(bx,by,bz);
  bx *= k_sin_2theta;
  by *= k_sin_2theta;
  bz *= k_sin_2theta;
  /* Get unit vector normal to ki and b */
  vec_prod(ax, ay, az, bx, by, bz, kux, kuy, kuz);
  /* Compute the total scattering probability at this ki */
  total = 0;
  width = 5*mos_rms;
  c1x = kix*(cos_2theta-1);
  c1y = kiy*(cos_2theta-1);
  c1z = kiz*(cos_2theta-1);
  for(i = 0; i < (sizeof(X)/sizeof(double)); i++)
  {
    phi = width*X[i];
    cos_phi = cos(phi);
    sin_phi = sin(phi);
    q_x = c1x + cos_phi*ax + sin_phi*bx;
    q_y = c1y + cos_phi*ay + sin_phi*by;
    q_z = c1z + cos_phi*az + sin_phi*bz;
    tmp = (q_x*q0x + q_y*q0y + q_z*q0z)/
          (sqrt(q_x*q_x + q_y*q_y + q_z*q_z)*q0);
    delta = tmp < 1 ? acos(tmp) : 0; /* Avoid rounding errors */
    p_reflect = GAUSS(delta,0,mos_rms);
    total += W[i]*p_reflect;
  }
  total *= width;
  /* Choose point on Debye-Scherrer cone. Use the double
     crystal mosaic (since the scattering angle is two times
     the Bragg angle), and correct for any error by adjusting
     the neutron weight later */
  phi = 2*mos_rms*randnorm();
  /* Compute final wave vector kf and scattering vector q = ki - kf */
  cos_phi = cos(phi);
  sin_phi = sin(phi);
  q_x = c1x + cos_phi*ax + sin_phi*bx;
  q_y = c1y + cos_phi*ay + sin_phi*by;
  q_z = c1z + cos_phi*az + sin_phi*bz;
  tmp = (q_x*q0x + q_y*q0y + q_z*q0z)/
        (sqrt(q_x*q_x + q_y*q_y + q_z*q_z)*q0);
  delta = tmp < 1 ? acos(tmp) : 0; /* Avoid rounding errors */
  p_reflect = GAUSS(delta,0,mos_rms);
  x = 0;
```

```
          y = y1;
          z = z1;
          t = t1;
          vx = K2V*(kix+q_x);
          vy = K2V*(kiy+q_y);
          vz = K2V*(kiz+q_z);
          p *= p_reflect/(total*GAUSS(phi,0,2*mos_rms));
        } /* End MC choice to reflect or transmit neutron */
      } /* End bragg scattering possible */
    } /* End intersect the crystal */
  } /* End neutron moving towards crystal */
%}

MCDISPLAY
%{
  double len = 0.5*sqrt((ymax-ymin)*(ymax-ymin) + (zmax-zmin)*(zmax-zmin));
  magnify("zy");
  multiline(5, 0.0, (double)ymin, (double)zmin,
               0.0, (double)ymax, (double)zmin,
               0.0, (double)ymax, (double)zmax,
               0.0, (double)ymin, (double)zmax,
               0.0, (double)ymin, (double)zmin);
  line(0, 0, 0,                 /* Draw Q0 vector */
       (double)Qx/Q*len, (double)Qy/Q*len, (double)Qz/Q*len);
%}

END
```

B.6.3 Single_crystal

```
/*******************************************************************************
*
* McStas, Maintained by Kristian Nielsen and Kim Lefmann,
*         Copyright 1997-2000 Risoe National Laboratory, Roskilde, Denmark
*
* Component: Single_crystal
*
* Written by: KN December 1999
*
* Single crystal with mosaic. Delta-D/D option for finite-size effects.
* Rectangular geometry.
* Crystal structure is specified with an ascii data file. Each line
* contains seven numbers, separated by white space. The first three numbers
* are the (h,k,l) indices of the reciprocal lattice point, and the last
* number is the value of the structure factor |F|**2, in barns. (The rest of
* the numbers are not used; the file is in the format output by the
* Crystallographica program).
*
* INPUT PARAMETERS:
*
* xwidth      : Width of crystal (m)
* yheight     : Height of crystal (m)
* zthick      : Thichness of crystal (no extinction simulated) (m)
* delta_d_d   : Lattice spacing variance, gaussian RMS (1)
* mosaic      : Crystal mosaic (anisotropic), gaussian RMS (arc minutes)
* ax,ay,az    : Coordinates of first unit cell vector (AA)
* bx,by,bz    : Coordinates of second unit cell vector (AA)
* cx,cy,cz    : Coordinates of third unit cell vector (AA)
* reflections : File name containing structure factors of reflections (string)
*
* OUTPUT PARAMETERS:
*
* hkl_info    : Internal
*
*******************************************************************************/

/*

  Overview of algorithm:

  (1). The neutron intersects the crystal at (x,y,z) with given
       incoming wavevector ki=(kix,kiy,kiz).

  (2). Every reciprocal lattice point tau of magnitude less than 2*ki
       is considered for scattering. The scattering probability is the
       area of the intersection of the Ewald sphere (approximated by
       the tangential plane) with the 3-D Gaussian mosaic of the point
       tau.

  (3). The total coherent scattering cross section is computed as the
       sum over all tau. Together with the absorption and incoherent
       scattering cross section and known potential flight-length
       l_full through the sample, we can compute the probability of
       the four events absorption, coherent scattering, incoherent
       scattering, and transmission.
  (4). Absorption is never simulated explicitly, just incorporated in
       the neutron weight.

  (5). Transmission in the first event is selected with the Monte
       Carlo probability p_transmit, which will usually be set to
       0. After the first event, transmission is selected with the
       correct Monte Carlo probability.

  (6). Incoherent scattering is done simply by selecting a random
       direction for the outgoing wave vector kf.

  (7). For coherent scattering, a reciprocal lattice point is selected
       using the relative probabilities computed in (2), and the
       weight is adjusted with the contribution from the structure
       factors (this way all reflections will get equally good
       statistics in the detector).

  (8). The outgoing wave vector direction is picked at random using
       the intersecting 2-D Gauss computed in (2). The vector is
       normalized to the length of ki (elastic scattering) to account
       for the error caused by the planar approximation of the Ewald
       sphere.

  (9). The process is repeated from (2) with kf as new initial wave
       vector ki.

 */

DEFINE COMPONENT Single_crystal
DEFINITION PARAMETERS(reflections)
SETTING PARAMETERS(xwidth, yheight, zthick, delta_d_d, mosaic,
                   ax, ay, az, bx, by, bz, cx, cy, cz
                   /* , p_transmit /* = 0 ToDo */)
OUTPUT PARAMETERS(hkl_info)
STATE PARAMETERS (x,y,z,vx,vy,vz,t,s1,s2,p)

DECLARE
%{
#ifndef SINGLE_CRYSTAL_DECL
#define SINGLE_CRYSTAL_DECL

/* Compute normal vector to (x,y,z).
   ToDo: Move to McStas kernel. */
void normal_vec(double *nx, double *ny, double *nz,
                double x, double y, double z)
{
  double ax = fabs(x);
  double ay = fabs(y);
  double az = fabs(z);
  double l;
  if(x == 0 && y == 0 && z == 0)
  {
    *nx = 0;
    *ny = 0;
    *nz = 0;
    return;
  }
  if(ax < ay)
  {
    if(ax < az)
    {                           /* Use X axis */
      l = sqrt(z*z + y*y);
      *nx = 0;
      *ny = z/l;
      *nz = -y/l;
      return;
    }
  }
  else
  {
    if(ay < az)
    {                           /* Use Y axis */
      l = sqrt(z*z + x*x);
      *nx = z/l;
      *ny = 0;
      *nz = -x/l;
      return;
    }
  }
  /* Use Z axis */
  l = sqrt(y*y + x*x);
  *nx = y/l;
  *ny = -x/l;
  *nz = 0;
}

/* Make sure a list is big enough to hold element COUNT.
   ToDo: Move to McStas kernel. */
void extend_list(int count, void **list, int *size, size_t elemsize)
{
  if(count >= *size)
  {
    void *oldlist = *list;
    if(*size > 0)
      *size *= 2;
    else
      *size = 32;
    *list = malloc(*size*elemsize);
    if(!*list)
    {
      fprintf(stderr, "\nFatal error: Out of memory.\n");
      exit(1);
    }
    if(oldlist)
    {
      memcpy(*list, oldlist, count*elemsize);
      free(oldlist);
    }
  }
}
```

```
/* If intersection with box dt_in and dt_out is returned */
/* This function written by Stine Nyborg, 1999. */
int box_intersect(double *dt_in, double *dt_out,
                  double x, double y, double z,
                  double vx, double vy, double vz,
                  double dx, double dy, double dz)
{
  double x_in, y_in, z_in, tt, t[6], a, b;
  int i, count, s;

      /* Calculate intersection time for each of the six box surface planes
       *  If the box surface plane is not hit, the result is zero.*/

  if(vx != 0)
   {
    tt = -(dx/2 + x)/vx;
    y_in = y + tt*vy;
    z_in = z + tt*vz;
    if( y_in > -dy/2 && y_in < dy/2 && z_in > -dz/2 && z_in < dz/2)
      t[0] = tt;
    else
      t[0] = 0;

    tt = (dx/2 - x)/vx;
    y_in = y + tt*vy;
    z_in = z + tt*vz;
    if( y_in > -dy/2 && y_in < dy/2 && z_in > -dz/2 && z_in < dz/2)
      t[1] = tt;
    else
      t[1] = 0;
   }
  else
    t[0] = t[1] = 0;

  if(vy != 0)
   {
    tt = -(dy/2 + y)/vy;
    x_in = x + tt*vx;
    z_in = z + tt*vz;
    if( x_in > -dx/2 && x_in < dx/2 && z_in > -dz/2 && z_in < dz/2)
      t[2] = tt;
    else
      t[2] = 0;

    tt = (dy/2 - y)/vy;
    x_in = x + tt*vx;
    z_in = z + tt*vz;
    if( x_in > -dx/2 && x_in < dx/2 && z_in > -dz/2 && z_in < dz/2)
      t[3] = tt;
    else
      t[3] = 0;
   }
  else
    t[2] = t[3] = 0;

  if(vz != 0)
```

```
   {
    tt = -(dz/2 + z)/vz;
    x_in = x + tt*vx;
    y_in = y + tt*vy;
    if( x_in > -dx/2 && x_in < dx/2 && y_in > -dy/2 && y_in < dy/2)
      t[4] = tt;
    else
      t[4] = 0;

    tt = (dz/2 - z)/vz;
    x_in = x + tt*vx;
    y_in = y + tt*vy;
    if( x_in > -dx/2 && x_in < dx/2 && y_in > -dy/2 && y_in < dy/2)
      t[5] = tt;
    else
      t[5] = 0;
   }
  else
    t[4] = t[5] = 0;

  /* The intersection is evaluated and *dt_in and *dt_out are assigned */

  a = b = s = 0;
  count = 0;

  for( i = 0; i < 6; i=i++ )
    if( t[i] == 0 )
      s = s+1;
    else if( count == 0 )
    {
      a = t[i];
      count = 1;
    }
    else
    {
      b = t[i];
      count = 2;
    }

  if ( a == 0 && b == 0 )
    return 0;
  else if( a < b )
  {
    *dt_in = a;
    *dt_out = b;
    return 1;
  }
  else
  {
    *dt_in = b;
    *dt_out = a;
    return 1;
  }

}
```

```
struct hkl_info
  {
    struct hkl_data *list;      /* Reflection array */
    int count;                  /* Number of reflections */
    struct tau_data *tau_list;  /* Reflections close to Ewald Sphere */
    double mosaic;              /* Isotropic mosaic (FWHM minutes) */
    double delta_d_d;           /* Delta-d/d FWHM */
    double ax,ay,az;            /* First unit cell axis (direct space, AA) */
    double bx,by,bz;            /* Second unit cell axis */
    double cx,cy,cz;            /* Third unit cell axis */
    double asx,asy,asz;         /* First reciprocal lattice axis (1/AA) */
    double bsx,bsy,bsz;         /* Second reciprocal lattice axis */
    double csx,csy,csz;         /* Third reciprocal lattice axis */
    double V0;                  /* Unit cell volume (AA**3) */
  };

struct hkl_data
  {
    int h,k,l;                  /* Indices for this reflection */
    double F2;                  /* Value of structure factor */
    double tau_x, tau_y, tau_z; /* Coordinates in reciprocal space */
    double tau;                 /* Length of (tau_x, tau_y, tau_z) */
    double u1x, u1y, u1z;       /* First axis of local coordinate system */
    double u2x, u2y, u2z;       /* Second axis of local coordinate system */
    double u3x, u3y, u3z;       /* Third axis of local coordinate system */
    double sig1, sig2, sig3;    /* RMSs of Gauss axis */
    double sig123;              /* The product sig1*sig2*sig3 */
    double m1, m2, m3;          /* Diagonal matrix representation of Gauss */
    double cutoff;              /* Cutoff value for Gaussian tails */
  };

struct tau_data
  {
    int index;                  /* Index into reflection table */
    double refl;
    double F2_contrib;
    double xsect;
    double sigma_1, sigma_2;
    /* The following vectors are in local koordinates. */
    double kix, kiy, kiz;       /* Initial wave vector */
    double rho_x, rho_y, rho_z; /* The vector ki - tau */
    double rho;                 /* Length of rho vector */
    double ox, oy, oz;          /* Origin of Ewald sphere tangent plane */
    double nx, ny, nz;          /* Normal vector of Ewald sphere tangent */
    double b1x, b1y, b1z;       /* Spanning vectors of Ewald sphere tangent */
    double b2x, b2y, b2z;
    double l11, l12, l22;       /* Cholesky decomposition L of 2D Gauss */
    double det_L;               /* Determinant of L */
    double y0x, y0y;            /* 2D Gauss center in tangent plane */
    double alpha;               /* Offset of 2D Gauss center from 3D center */
  };

struct hkl_data *
read_hkl_data(char *file, struct hkl_info *info)
{
  struct hkl_data *list = NULL;
```

```
int size = 0;
FILE *f;
int i;

f = fopen(file, "r");
if(!f)
{
  fprintf(stderr, "Single crystal: Error: file '%s' cannot be opened.\n",
          file);
  exit(1);
}
i = 0;
while(!feof(f))
{
  double h, k, l, multiplicity, d, ttheta, F2;
  int ret;
  ret = fscanf(f, "%lf %lf %lf %lf %lf %lf %lf\n",
               &h, &k, &l, &multiplicity, &d, &ttheta, &F2);
  if(ret == EOF)
    break;
  if(ret != 7)
  {
    fprintf(stderr,
            "Single crystal: Error reading from file '%s', line %d\n",
            file, i + 1);
    exit(1);
  }
  /* Extend list if not large enough. */
  extend_list(i, (void **)&list, &size, sizeof(*list));
  list[i].h = h;
  list[i].k = k;
  list[i].l = l;
  list[i].F2 = F2;
  /* Precompute some values */
  list[i].tau_x = h*info->asx + k*info->bsx + l*info->csx;
  list[i].tau_y = h*info->asy + k*info->bsy + l*info->csy;
  list[i].tau_z = h*info->asz + k*info->bsz + l*info->csz;
  list[i].tau = sqrt(list[i].tau_x*list[i].tau_x +
                     list[i].tau_y*list[i].tau_y +
                     list[i].tau_z*list[i].tau_z);
  list[i].u1x = list[i].tau_x/list[i].tau;
  list[i].u1y = list[i].tau_y/list[i].tau;
  list[i].u1z = list[i].tau_z/list[i].tau;
  normal_vec(&list[i].u2x, &list[i].u2y, &list[i].u2z,
             list[i].u1x, list[i].u1y, list[i].u1z);
  vec_prod(list[i].u3x, list[i].u3y, list[i].u3z,
           list[i].u1x, list[i].u1y, list[i].u1z,
           list[i].u2x, list[i].u2y, list[i].u2z);
  list[i].sig1 = FWHM2RMS*info->delta_d_d*list[i].tau;
  list[i].sig2 = FWHM2RMS*list[i].tau*MIN2RAD*info->mosaic;
  list[i].sig3 = FWHM2RMS*list[i].tau*MIN2RAD*info->mosaic;
  list[i].sig123 = list[i].sig1*list[i].sig2*list[i].sig3;
  list[i].m1 = 1/(2*list[i].sig1*list[i].sig1);
  list[i].m2 = 1/(2*list[i].sig2*list[i].sig2);
  list[i].m3 = 1/(2*list[i].sig3*list[i].sig3);
  /* Set Gauss cutoff to 5 times the maximal sigma. */
```

```
    if(list[i].sig1 > list[i].sig2)
      if(list[i].sig1 > list[i].sig3)
        list[i].cutoff = 5*list[i].sig1;
      else
        list[i].cutoff = 5*list[i].sig3;
    else
      if(list[i].sig2 > list[i].sig3)
        list[i].cutoff = 5*list[i].sig2;
      else
        list[i].cutoff = 5*list[i].sig3;
    i++;
  }
  fclose(f);
  printf("Single_crystal: Read %d reflections from file '%s'\n", i, file);
  info->list = list;
  info->count = i;
  info->tau_list = malloc(i*sizeof(*info->tau_list));
  if(!info->tau_list)
  {
    fprintf(stderr, "Single_crystal: Error: Out of memory!\n");
    exit(1);
  }
}


#endif /* !SINGLE_CRYSTAL_DECL */
  struct hkl_info hkl_info;
%}

INITIALIZE
%{
  double tmp_x, tmp_y, tmp_z;

  hkl_info.mosaic = mosaic;
  hkl_info.delta_d_d = delta_d_d;
  hkl_info.ax = ax;
  hkl_info.ay = ay;
  hkl_info.az = az;
  hkl_info.bx = bx;
  hkl_info.by = by;
  hkl_info.bz = bz;
  hkl_info.cx = cx;
  hkl_info.cy = cy;
  hkl_info.cz = cz;
  /* Compute reciprocal lattice vectors. */
  vec_prod(tmp_x, tmp_y, tmp_z, bx, by, bz, cx, cy, cz);
  hkl_info.V0 = scalar_prod(ax, ay, az, tmp_x, tmp_y, tmp_z);
  hkl_info.asx = 2*PI/hkl_info.V0*tmp_x;
  hkl_info.asy = 2*PI/hkl_info.V0*tmp_y;
  hkl_info.asz = 2*PI/hkl_info.V0*tmp_z;
  vec_prod(tmp_x, tmp_y, tmp_z, cx, cy, cz, ax, ay, az);
  hkl_info.bsx = 2*PI/hkl_info.V0*tmp_x;
  hkl_info.bsy = 2*PI/hkl_info.V0*tmp_y;
  hkl_info.bsz = 2*PI/hkl_info.V0*tmp_z;
  vec_prod(tmp_x, tmp_y, tmp_z, ax, ay, az, bx, by, bz);
  hkl_info.csx = 2*PI/hkl_info.V0*tmp_x;
  hkl_info.csy = 2*PI/hkl_info.V0*tmp_y;
  hkl_info.csz = 2*PI/hkl_info.V0*tmp_z;
  /* Read in structure factors, and do some pre-calculations. */
  read_hkl_data(reflections, &hkl_info);
%}
TRACE
%{
  double t1, t2;                /* Entry and exit times in sample */
  double dt;                    /* Flight time to next scattering event */
  struct hkl_data *L;           /* Structure factor list */
  int i;                        /* Index into structure factor list */
  struct tau_data *T;           /* List of reflections close to Ewald sphere */
  int j;                        /* Index into reflection list */
  int firstevent;               /* True for the first scattering event only */
  double kix, kiy, kiz, ki;     /* Initial wave vector [1/AA] */
  double kfx, kfy, kfz;         /* Final wave vector */
  double v;                     /* Neutron velocity */
  double tau_max;               /* Max tau allowing reflection at this ki */
  double rho_x, rho_y, rho_z;   /* the vector ki - tau */
  double rho;
  double diff;                  /* Deviation from Bragg condition */
  double ox, oy, oz;            /* Origin of Ewald sphere tangent plane */
  double b1x, b1y, b1z;         /* First vector spanning tangent plane */
  double b2x, b2y, b2z;         /* Second vector spanning tangent plane */
  double n11, n12, n22;         /* 2D Gauss description matrix N */
  double det_N;                 /* Determinant of N */
  double inv_n11, inv_n12, inv_n22; /* Inverse of N */
  double l11, l12, l22;         /* Cholesky decomposition L of 1/2*inv(N) */
  double det_L;                 /* Determinant of L */
  double Bt_D_O_x, Bt_D_O_y;    /* Temporaries */
  double y0x, y0y;              /* Center of 2D Gauss in plane coordinates */
  double d_x, d_y, d_z;         /* Vector deviation from Bragg condition */
  int tau_count;                /* Number of reflections within cutoff */
  double V0;                    /* Volume of unit cell */
  double l_full;                /* Neutron path length for transmission */
  double abs_xsect, abs_xlen;   /* Absorbtion cross section and length */
  double inc_xsect, inc_xlen;   /* Incoherent cross section and length */
  double coh_xsect, coh_xlen;   /* Coherent cross section and length */
  double tot_xsect, tot_xlen;   /* Total cross section and length */
  double z1, z2, y1, y2;        /* Temporaries to choose kf from 2D Gauss */
  double adjust,arg, refl;      /* Temporaries */

  firstevent = 1;
  for(;;)                       /* Loop over multiple scattering events */
  {
    /* (1). Compute incoming wave vector ki */
    v = sqrt(vx*vx + vy*vy + vz*vz);
    /* Compute intersection between neutron flight path and sample. */
    if(!box_intersect(&t1, &t2, x, y, z, vx, vy, vz,
                      xwidth, yheight, zthick) || t2 <= 0)
      ABSORB;
    if(t1 <= 0)
      fprintf(stderr,
              "Single_crystal: Warning: neutron started inside crystal!\n");
    /* Select a point at which to scatter the neutron. No extinction. */
    dt = randminmax(t1, t2);
```

```
PROP_DT(dt);
l_full = (t2 - t1)*v;
kix = V2K*vx;
kiy = V2K*vy;
kiz = V2K*vz;
ki = V2K*v;

/* (2). Intersection of Ewald sphere with recipprocal lattice points */
L = hkl_info.list;
T = hkl_info.tau_list;
/* Max possible tau with 5*sigma delta-d/d cutoff. */
tau_max = 2*ki/(1 - 5*delta_d_d);
for(i = j = 0; i < hkl_info.count; i++)
{
  /* Assuming reflections are sorted, stop search when max tau exceeded. */
  if(L[i].tau > tau_max)
    break;
  /* Check if this reciprocal lattice point is close enough to the
     Ewald sphere to make scattering possible. */
  rho_x = kix - L[i].tau_x;
  rho_y = kiy - L[i].tau_y;
  rho_z = kiz - L[i].tau_z;
  rho = sqrt(rho_x*rho_x + rho_y*rho_y + rho_z*rho_z);
  diff = fabs(rho - ki);
  /* Check if scattering is possible (cutoff of Gaussian tails). */
  if(diff <= L[i].cutoff)
  {
    /* Store reflection. */
    T[j].index = i;
    /* Get ki vector in local coordinates. */
    T[j].kix = kix*L[i].u1x + kiy*L[i].u1y + kiz*L[i].u1z;
    T[j].kiy = kix*L[i].u2x + kiy*L[i].u2y + kiz*L[i].u2z;
    T[j].kiz = kix*L[i].u3x + kiy*L[i].u3y + kiz*L[i].u3z;
    T[j].rho_x = T[j].kix - L[i].tau;
    T[j].rho_y = T[j].kiy;
    T[j].rho_z = T[j].kiz;
    T[j].rho = rho;
    /* Compute the tangent plane of the Ewald sphere. */
    T[j].nx = T[j].rho_x/T[j].rho;
    T[j].ny = T[j].rho_y/T[j].rho;
    T[j].nz = T[j].rho_z/T[j].rho;
    ox = (ki - T[j].rho)*T[j].nx;
    oy = (ki - T[j].rho)*T[j].ny;
    oz = (ki - T[j].rho)*T[j].nz;
    T[j].ox = ox;
    T[j].oy = oy;
    T[j].oz = oz;
    /* Compute unit vectors b1 and b2 that span the tangent plane. */
    normal_vec(&b1x, &b1y, &b1z, T[j].nx, T[j].ny, T[j].nz);
    vec_prod(b2x, b2y, b2z, T[j].nx, T[j].ny, T[j].nz, b1x, b1y, b1z);
    T[j].b1x = b1x;
    T[j].b1y = b1y;
    T[j].b1z = b1z;
    T[j].b2x = b2x;
    T[j].b2y = b2y;
    T[j].b2z = b2z;
```

```
        /* Compute the 2D projection of the 3D Gauss of the reflection. */
        /* The symmetric 2x2 matrix N describing the 2D gauss. */
        n11 = L[i].m1*b1x*b1x + L[i].m2*b1y*b1y + L[i].m3*b1z*b1z;
        n12 = L[i].m1*b1x*b2x + L[i].m2*b1y*b2y + L[i].m3*b1z*b2z;
        n22 = L[i].m1*b2x*b2x + L[i].m2*b2y*b2y + L[i].m3*b2z*b2z;
        /* The (symmetric) inverse matrix of N. */
        det_N = n11*n22 - n12*n12;
        inv_n11 = n22/det_N;
        inv_n12 = -n12/det_N;
        inv_n22 = n11/det_N;
        /* The Cholesky decomposition of 1/2*inv_n (lower triangular L). */
        l11 = sqrt(inv_n11/2);
        l12 = inv_n12/(2*l11);
        l22 = sqrt(inv_n22/2 - l12*l12);
        T[j].l11 = l11;
        T[j].l12 = l12;
        T[j].l22 = l22;
        det_L = l11*l22;
        T[j].det_L = det_L;
        /* The product B^T D o. */
        Bt_D_O_x = b1x*L[i].m1*ox + b1y*L[i].m2*oy + b1z*L[i].m3*oz;
        Bt_D_O_y = b2x*L[i].m1*ox + b2y*L[i].m2*oy + b2z*L[i].m3*oz;
        /* Center of 2D Gauss in plane coordinates. */
        y0x = -(Bt_D_O_x*inv_n11 + Bt_D_O_y*inv_n12);
        y0y = -(Bt_D_O_x*inv_n12 + Bt_D_O_y*inv_n22);
        T[j].y0x = y0x;
        T[j].y0y = y0y;
        /* Factor alpha for the distance of the 2D Gauss from the origin. */
        T[j].alpha = L[i].m1*ox*ox + L[i].m2*oy*oy + L[i].m3*oz*oz -
                     (y0x*y0x*n11 + y0y*y0y*n22 + 2*y0x*y0y*n12);
        j++;
      }
    }
    tau_count = j;
    if(tau_count == 0)
    {
/* printf("** No nearby reflections.\n"); */
      ABSORB;                   /* No reflections possible */
    }
    /* (3). Probabilities of the different possible interactions. */
    V0 = hkl_info.V0;
    abs_xsect = (0 /* ToDo: absorbtion at 2200m/s */) / v;
    abs_xlen = abs_xsect/V0;
    inc_xsect = 0 /* ToDo: Incoherent cross section */;
    inc_xlen = inc_xsect/V0;
    coh_xsect = 0 /* ToDo: Coherent cross section */;
    coh_xlen = coh_xsect/V0;
    tot_xsect = abs_xsect + inc_xsect + coh_xsect;
    tot_xlen = tot_xsect/V0;
    /* (5). Transmission */
/*     p_trans = exp(-tot_xlen*l_full); */
/*     if(firstevent) { */
/*       mc_trans = p_transmit; */
/*     } else { */
/*       mc_trans = p_trans; */
/*     } */
/*     firstevent = 0; */
/*     if(rand01() < mc_trans)  /* Transmit */
/*     { */
/*       p *= p_trans/mc_trans; */
/*       break; */
/*     } */
    /* dP(l) = exp(-tot_xlen*l)dl
       P(l<l_0) = [-1/tot_xlen*exp(-tot_xlen*l)]_0^l_0
                = (1 - exp(-tot_xlen*l0))/tot_xlen
       l = -log(1 - tot_xlen*rand0max(P(l<l_full)))/tot_xlen
     */
/*     l = -log(1 - rand0max((1 - exp(-tot_xlen*l0))))/tot_xlen; */
    /* (4). Account for the probability of absorbtion */
/*     p *= (coh_xlen + inc_xlen)/tot_xlen; */
    /* Choose between coherent and incoherent scattering */
/*     if(rand0max(coh_xlen + inc_xlen) < inc_xlen) */
/*     { */
/*       /* (6). Incoherent scattering */
/*       randvec_target_sphere(kix, kiy, kiz, NULL, 0, 0, 0, 1); */
/*       kix *= ki; */
/*       kiy *= ki; */
/*       kiz *= ki; */
/*       continue;                      /* Go for next scattering event */
/*     } */
    /* 7. Coherent scattering. Select reciprocal lattice point. */
/*     r = rand0max(coh_xsect); */
/*     sum = 0; */
/*     for(j = 0; j < tau_count; j++) */
/*     { */
/*       sum += T[j].refl; */
/*       if(sum > r) */
/*      break; */
/*     } */
    j = floor(rand0max(tau_count));
    if(j >= tau_count)
    {
/*       fprintf(stderr, "Single_crystal: Error: Illegal tau search " */
/*            "(r = %g, sum = %g).\n", r, sum); */
      j = tau_count - 1;
    }
    i = T[j].index;
/*     p *= T[i].F2_contrib; */
    /* (8). Pick scattered wavevector kf from 2D Gauss distribution. */
    z1 = randnorm();
    z2 = randnorm();
    y1 = T[j].l11*z1 + T[j].y0x;
    y2 = T[j].l12*z1 + T[j].l22*z2 + T[j].y0y;
    kfx = T[j].rho_x + T[j].ox + T[j].b1x*y1 + T[j].b2x*y2;
    kfy = T[j].rho_y + T[j].oy + T[j].b1y*y1 + T[j].b2y*y2;
    kfz = T[j].rho_z + T[j].oz + T[j].b1z*y1 + T[j].b2z*y2;
    /* Normalize kf to length of ki, to account for planer
       approximation of the Ewald sphere. */
    adjust = ki/sqrt(kfx*kfx + kfy*kfy + kfz*kfz);
    kfx *= adjust;
    kfy *= adjust;
    kfz *= adjust;
    /* Adjust neutron weight (see manual for explanation). */
    p *= l_full*1e10*L[i].F2*1e-4*pow(2*PI, 2.5)*
         T[j].det_L*tau_count*exp(-T[j].alpha)/
         (V0*V0*ki*ki*L[i].sig123);
    vx = K2V*(L[i].u1x*kfx + L[i].u2x*kfy + L[i].u3x*kfz);
    vy = K2V*(L[i].u1y*kfx + L[i].u2y*kfy + L[i].u3y*kfz);
    vz = K2V*(L[i].u1z*kfx + L[i].u2z*kfy + L[i].u3z*kfz);
    break;
    /* Repeat loop for next scattering event. */
  }

%}

MCDISPLAY
%{
  double xmin = -0.5*xwidth;
  double xmax =  0.5*xwidth;
  double ymin = -0.5*yheight;
  double ymax =  0.5*yheight;
  double zmin = -0.5*zthick;
  double zmax =  0.5*zthick;
  magnify("xyz");
  multiline(5, xmin, ymin, zmin,
               xmax, ymin, zmin,
               xmax, ymax, zmin,
               xmin, ymax, zmin,
               xmin, ymin, zmin);
  multiline(5, xmin, ymin, zmax,
               xmax, ymin, zmax,
               xmax, ymax, zmax,
               xmin, ymax, zmax,
               xmin, ymin, zmax);
  line(xmin, ymin, zmin, xmin, ymin, zmax);
  line(xmax, ymin, zmin, xmax, ymin, zmax);
  line(xmin, ymax, zmin, xmin, ymax, zmax);
  line(xmax, ymax, zmin, xmax, ymax, zmax);
%}
END
```

B.7 Powder-like samples

B.7.2 V_sample

```
/*******************************************************************************
*
* McStas, version 1.0, released October 26, 1998
*         Maintained by Kristian Nielsen and Kim Lefmann,
*         Risoe National Laboratory, Roskilde, Denmark
*
* Component: V-sample
*
* Written by: KL, KN 15.4.98
*
* A Double-cylinder shaped incoherent scatterer (a V-sample)
* No multiple scattering. Absorbtion included.
*
* INPUT PARAMETERS:
*
* radius_i  : Inner radius of sample in (x,z) plane (m)
* radius_o  : Outer radius of sample in (x,z) plane (m)
* h         : Height of sample y direction (m)
* pack      : Packing factor (1)
* focus_r   : Radius of sphere containing target. (m)
* target_x  :
* target_y  : position of target to focus at (m)
* target_z  :
*
* Variables calculated in the component
*
* V_my_s      : Attenuation factor due to scattering (m^-1)
* V_my_a      : Attenuation factor due to absorbtion (m^-1)
*
*******************************************************************************/

DEFINE COMPONENT V_sample
DEFINITION PARAMETERS (radius_i,radius_o,h,pack,focus_r)
SETTING PARAMETERS (target_x, target_y, target_z)
STATE PARAMETERS (x,y,z,vx,vy,vz,t,s1,s2,p)
DECLARE
%{
/* ToDo: Should be component local names. */
#define V_sigma_a 5.08     /* Absorption cross section per atom (barns) */
#define V_sigma_i 4.935    /* Incoherent scattering cross section per atom (barns) */
#define V_rho (2*pack/(3.024*3.024*3.024)) /* Density of atoms (AA-3) */
#define V_my_s (V_rho * 100 * V_sigma_i)
#define V_my_a_v (V_rho * 100 * V_sigma_a * 2200)
%}
INITIALIZE
%{
%}
TRACE
%{
  double t0, t3;                /* Entry/exit time for outer cylinder */
  double t1, t2;                /* Entry/exit time for inner cylinder */
  double v;                     /* Neutron velocity */
  double dt0, dt1, dt2, dt;     /* Flight times through sample */
  double l_full;                /* Flight path length for non-scattered neutron */
  double l_i, l_o;              /* Flight path lenght in/out for scattered neutron */
  double my_a;                  /* Velocity-dependent attenuation factor */
  double solid_angle;           /* Solid angle of target as seen from scattering point */
  double aim_x, aim_y, aim_z;   /* Position of target relative to scattering point */

  if(cylinder_intersect(&t0, &t3, x, y, z, vx, vy, vz, radius_o, h))
  {
    if(t0 < 0)
      ABSORB;
    /* Neutron enters at t=t0. */
    if(!cylinder_intersect(&t1, &t2, x, y, z, vx, vy, vz, radius_i, h))
      t1 = t2 = t3;

    dt0 = t1-t0;                /* Time in sample, ingoing */
    dt1 = t2-t1;                /* Time in hole */
    dt2 = t3-t2;                /* Time in sample, outgoing */
    v = sqrt(vx*vx + vy*vy + vz*vz);
    l_full = v * (dt0 + dt2);   /* Length of full path through sample */
    dt = rand01()*(dt0+dt2);    /* Time of scattering (relative to t0) */
    l_i = v*dt;                 /* Penetration in sample */
    if (dt > dt0)
      dt += dt1;

    PROP_DT(dt+t0);             /* Point of scattering */

    aim_x = target_x-x;         /* Vector pointing at target (anal./det.) */
    aim_y = target_y-y;
    aim_z = target_z-z;
    randvec_target_sphere(&vx, &vy, &vz, &solid_angle, aim_x, aim_y, aim_z, focus_r);
    NORM(vx, vy, vz);
    vx *= v;
    vy *= v;
    vz *= v;

    if(!cylinder_intersect(&t0, &t3, x, y, z, vx, vy, vz, radius_o, h))
    {
      /* ??? did not hit cylinder */
      printf("FATAL ERROR: Did not hit cylinder from inside.\n");
      exit(1);
    }
    dt = t3;
    if(cylinder_intersect(&t1, &t2, x, y, z, vx, vy, vz, radius_i, h) &&
       t2 > 0)
      dt -= (t2-t1);            /* Subtract hollow part */
    l_o = v*dt;

    my_a = V_my_a_v/v;
    p *= l_full*V_my_s*exp(-(my_a+V_my_s)*(l_i+l_o));
    p /= 4*PI/solid_angle;
  }
  else
    ABSORB;
%}

MCDISPLAY
```

```
%{
  magnify("xyz");
  circle("xz", 0,  h/2.0, 0, radius_i);
  circle("xz", 0,  h/2.0, 0, radius_o);
  circle("xz", 0, -h/2.0, 0, radius_i);
  circle("xz", 0, -h/2.0, 0, radius_o);
  line(-radius_i, -h/2.0, 0, -radius_i, +h/2.0, 0);
  line(+radius_i, -h/2.0, 0, +radius_i, +h/2.0, 0);
  line(0, -h/2.0, -radius_i, 0, +h/2.0, -radius_i);
  line(0, -h/2.0, +radius_i, 0, +h/2.0, +radius_i);
  line(-radius_o, -h/2.0, 0, -radius_o, +h/2.0, 0);
  line(+radius_o, -h/2.0, 0, +radius_o, +h/2.0, 0);
  line(0, -h/2.0, -radius_o, 0, +h/2.0, -radius_o);
  line(0, -h/2.0, +radius_o, 0, +h/2.0, +radius_o);
%}

END
```

B.7.3 Powder1

```
/*******************************************************************************
*
* McStas, version 1.0, released October 26, 1998
*         Maintained by Kristian Nielsen and Kim Lefmann,
*         Risoe National Laboratory, Roskilde, Denmark
*
* Component: General powder sample (powder1)
*
* Written by: E.M.Lauridsen, N.B.Christensen, A.B.Abrahamsen 4.2.98
* Rewritten by: KL, KN 20.3.98
*
* INPUT PARAMETERS
*
* d_phi0    : Focussing angle corresponding to the vertical dimensions
*             of the detector placed at the right distance (deg)
* radius    : Radius of sample in (x,z) plane (m)
* h         : Height of sample y direction (m)
* pack      : Packing factor (1)
* Vc        : Volume of unit cell (AA^3)
* sigma_a   : Absorption cross section per unit cell at 2200 m/s (fm^2)
*
* q         : Scattering vector of reflection (AA^-1)
* j         : Multiplicity of reflection (1)
* F2        : Structure factor of reflection (fm^2)
* DW        : Debye-Waller factor of reflection (1)
* target_x  :
* target_y  : position of target to focus at (m)
* target_z  :
*
* Variables calculated in the component
*
* my_s      : Attenuation factor due to scattering (m^-1)
* my_a      : Attenuation factor due to absorbtion (m^-1)
*******************************************************************************/

DEFINE COMPONENT Powder1
DEFINITION PARAMETERS (d_phi0, radius, h, pack, Vc, sigma_a, j, q, F2, DW)
SETTING PARAMETERS (target_x, target_y, target_z)
OUTPUT PARAMETERS (my_s_v2, my_a_v, q_v)
STATE PARAMETERS (x,y,z,vx,vy,vz,t,s1,s2,p)
DECLARE
%{
  double my_s_v2, my_a_v, q_v;
%}
INITIALIZE
%{
  my_a_v = sigma_a/Vc*2200;           /* Is not yet divided by v */
  my_s_v2 = PI*PI*PI*pack*j*F2*DW/(Vc*Vc*V2K*V2K*q);
                                      /* Is not yet divided by v^2 */
  q_v = q*K2V;
%}
TRACE
%{
  double t0, t1, v, l_full, l, l_1, dt, d_phi, theta, my_s;
  double aim_x, aim_y, aim_z, axis_x, axis_y, axis_z;
  double arg, tmp_vx, tmp_vy, tmp_vz, vout_x, vout_y, vout_z;

  if(cylinder_intersect(&t0, &t1, x, y, z, vx, vy, vz, radius, h))
  {
    if(t0 < 0)
      ABSORB;
    /* Neutron enters at t=t0. */
    v = sqrt(vx*vx + vy*vy + vz*vz);
    l_full = v * (t1 - t0);             /* Length of full path through sample */
    dt = rand01()*(t1 - t0) + t0; /* Time of scattering */
    PROP_DT(dt);                /* Point of scattering */
    l = v*dt;                   /* Penetration in sample */

    /* Choose point on Debye-Scherrer cone */
    d_phi = d_phi0*DEG2RAD/2.0*randpm1();
    p *= d_phi0/360.0;
    arg = q_v/(2.0*v);
    if(arg > 1)
      ABSORB;                   /* No bragg scattering possible*/
    theta = asin(arg);          /* Bragg scattering law */

    aim_x = target_x-x;         /* Vector pointing at target (anal./det.) */
    aim_y = target_y-y;
    aim_z = target_z-z;
    vec_prod(axis_x, axis_y, axis_z, vx, vy, vz, aim_x, aim_y, aim_z);
    rotate(tmp_vx, tmp_vy, tmp_vz, vx, vy, vz, 2*theta, axis_x, axis_y, axis_z);
    rotate(vout_x, vout_y, vout_z, tmp_vx, tmp_vy, tmp_vz, d_phi, vx, vy, vz);
    vx = vout_x;
    vy = vout_y;
    vz = vout_z;

    if(!cylinder_intersect(&t0, &t1, x, y, z,
                           vout_x, vout_y, vout_z, radius, h))
    {
      /* Strange error: did not hit cylinder */
      printf("FATAL ERROR: Did not hit cylinder from inside.\n");
      exit(1);
    }
    l_1 = v*t1;

    my_s = my_s_v2/(v*v);
    p *= l_full*my_s*exp(-(my_a_v/v+my_s)*(l+l_1));
  }
  else
    ABSORB;
%}

MCDISPLAY
%{
  magnify("xyz");
  circle("xz", 0,  h/2.0, 0, radius);
  circle("xz", 0, -h/2.0, 0, radius);
  line(-radius, -h/2.0, 0, -radius, +h/2.0, 0);
  line(+radius, -h/2.0, 0, +radius, +h/2.0, 0);
  line(0, -h/2.0, -radius, 0, +h/2.0, -radius);
  line(0, -h/2.0, +radius, 0, +h/2.0, +radius);
```

```
%}
END
```

B.8 Inelastic samples

B.8.1 Res_sample

```
/*******************************************************************************
*
* McStas, the neutron ray-tracing package
*         Maintained by Kristian Nielsen and Kim Lefmann,
*         Copyright 1977-2000 Risoe National Laboratory, Roskilde, Denmark
*
* Component: Res_sample
*
* Written by: KN 1999
*
* An inelastic sample with completely uniform scattering in both Q and
* energy. This sample is used together with the Res_monitor component and
* (optionally) the mcresplot front-end to compute the resolution function of
* triple-axis or inverse-geometry time-of-flight instruments.
*
* The shape of the sample is either a hollow cylinder or a rectangular box. The
* hollow cylinder shape is specified with an inner and outer radius. If the
* outher radius is negative, the shape is instead a box.
*
* The scattered neutrons will have directions towards a given sphere and
* energies betweed E0-dE and E0+dE.
*
* INPUT PARAMETERS:
*
* radius_i  : Inner radius of hollow cylinder in (x,z) plane, or width of
*             box along X (m)
* radius_o  : Outer radius of hollow cylinder, or negative box depth along Z (m)
* h         : Height of box or cylinder along Y (m)
* focus_r   : Radius of sphere containing target. (m)
* target_x  :
* target_y  : position of target to focus at (m)
* target_z  :
* E0        : Center of scattered energy range
* dE        : half width of scattered energy range
*
*******************************************************************************/

DEFINE COMPONENT Res_sample
DEFINITION PARAMETERS (radius_i,radius_o,h,focus_r,E0,dE)
SETTING PARAMETERS (target_x, target_y, target_z)
OUTPUT PARAMETERS (res_struct)
STATE PARAMETERS (x,y,z,vx,vy,vz,t,s1,s2,p)
DECLARE
%{
  struct Res_sample_struct
    {
      double ki_x,ki_y,ki_z,kf_x,kf_y,kf_z;
      double rx,ry,rz,pi;
    } res_struct;
%}
INITIALIZE
%{
%}
```

```
TRACE
%{
  double t0, t3;                /* Entry/exit time for outer cylinder */
  double t1, t2;                /* Entry/exit time for inner cylinder */
  double v;                     /* Neutron velocity */
  double E;
  double l_full;                /* Flight path length for non-scattered neutron */
  double dt0, dt1, dt2, dt;     /* Flight times through sample */
  double solid_angle;           /* Solid angle of target as seen from scattering point */
  double aim_x, aim_y, aim_z;   /* Position of target relative to scattering point */
  double scat_factor;           /* Simple cross-section model */

  if(radius_o < 0.0)
  {                             /* Flat sample */
    PROP_Z0;
    if(x <= -0.5*radius_i || x >= 0.5*radius_i ||
       y <= -0.5*h || y >= 0.5*h)
      ABSORB;
    t0 = 0;
    t1 = t2 = t3 = (-radius_o)/vz;
    scat_factor = -2*radius_o;
  }
  else
  {                             /* Hollow cylinder sample */
    if(!cylinder_intersect(&t0, &t3, x, y, z, vx, vy, vz, radius_o, h))
      ABSORB;
    if(t0 < 0)
      ABSORB;
    /* Neutron enters at t=t0. */
    if(!cylinder_intersect(&t1, &t2, x, y, z, vx, vy, vz, radius_i, h))
      t1 = t2 = t3;
    scat_factor = 2*radius_o;
  }
  dt0 = t1-t0;                  /* Time in sample, ingoing */
  dt1 = t2-t1;                  /* Time in hole */
  dt2 = t3-t2;                  /* Time in sample, outgoing */
  v = sqrt(vx*vx + vy*vy + vz*vz);
  l_full = v * (dt0 + dt2);     /* Length of full path through sample */
  p *= l_full/scat_factor;      /* Scattering probability */
  dt = rand01()*(dt0+dt2);      /* Time of scattering (relative to t0) */
  if (dt > dt0)
    dt += dt1;

  PROP_DT(dt+t0);             /* Point of scattering */

  /* Store initial neutron state. */
  if(p == 0) ABSORB;
  res_struct.pi = p;
  res_struct.ki_x = V2K*vx;
  res_struct.ki_y = V2K*vy;
  res_struct.ki_z = V2K*vz;
  res_struct.rx = x;
  res_struct.ry = y;
  res_struct.rz = z;

  aim_x = target_x-x;         /* Vector pointing at target (anal./det.) */
  aim_y = target_y-y;
  aim_z = target_z-z;
  randvec_target_sphere(&vx, &vy, &vz, &solid_angle, aim_x, aim_y, aim_z, focus_r);
  NORM(vx, vy, vz);
  E=E0+dE*randpm1();
  v=sqrt(E)*SE2V;
  vx *= v;
  vy *= v;
  vz *= v;

    /* Store final neutron state. */
  res_struct.kf_x = V2K*vx;
  res_struct.kf_y = V2K*vy;
  res_struct.kf_z = V2K*vz;
%}

MCDISPLAY
%{
  magnify("xyz");
  if(radius_o < 0.0)
  {                             /* Flat sample. */
    double xmin = -0.5*radius_i;
    double xmax =  0.5*radius_i;
    double ymin = -0.5*h;
    double ymax =  0.5*h;
    double len = -radius_o;
    multiline(5, xmin, ymin, 0.0,
                 xmax, ymin, 0.0,
                 xmax, ymax, 0.0,
                 xmin, ymax, 0.0,
                 xmin, ymin, 0.0);
    multiline(5, xmin, ymin, len,
                 xmax, ymin, len,
                 xmax, ymax, len,
                 xmin, ymax, len,
                 xmin, ymin, len);
    line(xmin, ymin, 0.0, xmin, ymin, len);
    line(xmax, ymin, 0.0, xmax, ymin, len);
    line(xmin, ymax, 0.0, xmin, ymax, len);
    line(xmax, ymax, 0.0, xmax, ymax, len);
  }
  else
  {
    circle("xz", 0,  h/2.0, 0, radius_i);
    circle("xz", 0,  h/2.0, 0, radius_o);
    circle("xz", 0, -h/2.0, 0, radius_i);
    circle("xz", 0, -h/2.0, 0, radius_o);
    line(-radius_i, -h/2.0, 0, -radius_i, +h/2.0, 0);
    line(+radius_i, -h/2.0, 0, +radius_i, +h/2.0, 0);
    line(0, -h/2.0, -radius_i, 0, +h/2.0, -radius_i);
    line(0, -h/2.0, +radius_i, 0, +h/2.0, +radius_i);
    line(-radius_o, -h/2.0, 0, -radius_o, +h/2.0, 0);
    line(+radius_o, -h/2.0, 0, +radius_o, +h/2.0, 0);
    line(0, -h/2.0, -radius_o, 0, +h/2.0, -radius_o);
    line(0, -h/2.0, +radius_o, 0, +h/2.0, +radius_o);
  }
```

```
%}

END
```

Appendix C

McStas instrument definitions

In this appendix is listed the source code for the instrument definitions presented in section 6.

C.1 Code for the instrument `vanadium_example.instr`

```
DEFINE INSTRUMENT test_v_sample(ROT)

DECLARE
%{
  double coll_div = 60;
%}

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

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

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

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

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

C.2 Code for the instrument `linup-7.instr`

```
DEFINE INSTRUMENT TAS1(PHM,TTM,TT,OMA,TTA,C1,OMC1,C2,C3)

DECLARE
%{
/* Mosaicity used on monochromator and analysator */
double tas1_mono_mosaic = 45; /* Measurements indicate its really 45' */
double tas1_ana_mosaic = 45; /* Measurements indicate its really 45' */
/* Q vector for bragg scattering with monochromator and analysator */
double tas1_mono_q = 2*1.87325; /* Fake 2nd order scattering for 20meV */
double tas1_mono_r0 = 0.6;
double tas1_ana_q = 1.87325; /* 20meV */
double tas1_ana_r0 = 0.6;

double OMC1_d;
double alu_focus_x;

double mpos0, mpos1, mpos2, mpos3, mpos4, mpos5, mpos6, mpos7;
double mrot0, mrot1, mrot2, mrot3, mrot4, mrot5, mrot6, mrot7;
%}

INITIALIZE
%{
  double d = 0.0125; /* 12.5 mm between slab centers. */
  double phi = 0.5443; /* Rotation between adjacent slabs. */
  mpos0 = -3.5*d; mrot0 = -3.5*phi;
  mpos1 = -2.5*d; mrot1 = -2.5*phi;
  mpos2 = -1.5*d; mrot2 = -1.5*phi;
  mpos3 = -0.5*d; mrot3 = -0.5*phi;
  mpos4 =  0.5*d; mrot4 =  0.5*phi;
  mpos5 =  1.5*d; mrot5 =  1.5*phi;
  mpos6 =  2.5*d; mrot6 =  2.5*phi;
  mpos7 =  3.5*d; mrot7 =  3.5*phi;

  OMC1_d = OMC1/60.0;
  alu_focus_x = TT >= 0 ? 1000 : -1000;
%}

TRACE

COMPONENT a1 = Arm()
  AT (0,0,0) ABSOLUTE

COMPONENT source = Source_flat(
radius = 0.060,
dist = 3.288,
xw = 0.042, yh = 0.082,
E0 = 20,/* 20 meV */
dE = 0.82) /* Sufficient for TAS1 geometry */
  AT (0,0,0) RELATIVE a1 ROTATED (0,0,0) RELATIVE a1

COMPONENT slit1 = Slit(
xmin=-0.020, xmax=0.065,
ymin = -0.075, ymax = 0.075)
  AT (0, 0, 1.1215) RELATIVE a1 ROTATED (0,0,0) RELATIVE a1
```

```
COMPONENT slit2 = Slit(
xmin = -0.020, xmax = 0.020,
ymin = -0.040, ymax = 0.040)
  AT (0,0,1.900) RELATIVE a1 ROTATED (0,0,0) RELATIVE a1

COMPONENT slit3 = Slit(
xmin = -0.021, xmax = 0.021,
ymin = -0.041, ymax = 0.041)
  AT (0,0,3.288) RELATIVE a1 ROTATED (0,0,0) RELATIVE a1

COMPONENT focus_mono = Arm()
  AT (0, 0, 3.56) RELATIVE a1 ROTATED (0, PHM, 0) RELATIVE a1

COMPONENT m0 = Monochromator(
zmin=-0.0375,zmax=0.0375,ymin=-0.006,ymax=0.006,
mosaich=tas1_mono_mosaic,mosaicv=tas1_mono_mosaic,
r0=tas1_mono_r0, Q=tas1_mono_q)
  AT (0, mpos0, 0) RELATIVE focus_mono
  ROTATED (0, 0, mrot0) RELATIVE focus_mono

COMPONENT m1 = Monochromator(
zmin=-0.0375,zmax=0.0375,ymin=-0.006,ymax=0.006,
mosaich=tas1_mono_mosaic,mosaicv=tas1_mono_mosaic,
r0=tas1_mono_r0, Q=tas1_mono_q)
  AT (0, mpos1, 0) RELATIVE focus_mono
  ROTATED (0, 0, mrot1) RELATIVE focus_mono

COMPONENT m2 = Monochromator(
zmin=-0.0375,zmax=0.0375,ymin=-0.006,ymax=0.006,
mosaich=tas1_mono_mosaic,mosaicv=tas1_mono_mosaic,
r0=tas1_mono_r0, Q=tas1_mono_q)
  AT (0, mpos2, 0) RELATIVE focus_mono
  ROTATED (0, 0, mrot2) RELATIVE focus_mono

COMPONENT m3 = Monochromator(
zmin=-0.0375,zmax=0.0375,ymin=-0.006,ymax=0.006,
mosaich=tas1_mono_mosaic,mosaicv=tas1_mono_mosaic,
r0=tas1_mono_r0, Q=tas1_mono_q)
  AT (0, mpos3, 0) RELATIVE focus_mono
  ROTATED (0, 0, mrot3) RELATIVE focus_mono

COMPONENT m4 = Monochromator(
zmin=-0.0375,zmax=0.0375,ymin=-0.006,ymax=0.006,
mosaich=tas1_mono_mosaic,mosaicv=tas1_mono_mosaic,
r0=tas1_mono_r0, Q=tas1_mono_q)
  AT (0, mpos4, 0) RELATIVE focus_mono
  ROTATED (0, 0, mrot4) RELATIVE focus_mono

COMPONENT m5 = Monochromator(
zmin=-0.0375,zmax=0.0375,ymin=-0.006,ymax=0.006,
mosaich=tas1_mono_mosaic,mosaicv=tas1_mono_mosaic,
r0=tas1_mono_r0, Q=tas1_mono_q)
  AT (0, mpos5, 0) RELATIVE focus_mono
  ROTATED (0, 0, mrot5) RELATIVE focus_mono
```

```
COMPONENT m6 = Monochromator(
zmin=-0.0375,zmax=0.0375,ymin=-0.006,ymax=0.006,
mosaich=tas1_mono_mosaic,mosaicv=tas1_mono_mosaic,
r0=tas1_mono_r0, Q=tas1_mono_q)
  AT (0, mpos6, 0) RELATIVE focus_mono
  ROTATED (0, 0, mrot6) RELATIVE focus_mono

COMPONENT m7 = Monochromator(
zmin=-0.0375,zmax=0.0375,ymin=-0.006,ymax=0.006,
mosaich=tas1_mono_mosaic,mosaicv=tas1_mono_mosaic,
r0=tas1_mono_r0, Q=tas1_mono_q)
  AT (0, mpos7, 0) RELATIVE focus_mono
  ROTATED (0, 0, mrot7) RELATIVE focus_mono

COMPONENT a2 = Arm()
  AT (0,0,0) RELATIVE focus_mono ROTATED (0, TTM, 0) RELATIVE a1

COMPONENT slitMS1 = Slit(
xmin = -0.0105, xmax = 0.0105, ymin = -0.035, ymax = 0.035)
  AT (0,0,0.565) RELATIVE a2 ROTATED (0,0,0) RELATIVE a2

COMPONENT slitMS2 = Slit(
xmin = -0.0105, xmax = 0.0105, ymin = -0.035, ymax = 0.035)
  AT (0,0,0.855) RELATIVE a2 ROTATED (0,0,0) RELATIVE a2

COMPONENT c1 = Soller(
xmin = -0.02, xmax = 0.02, ymin = -0.0375, ymax = 0.0375,
len = 0.250, divergence = C1)
  AT (0, 0, 0.87) RELATIVE a2 ROTATED (0,OMC1_d,0) RELATIVE a2

COMPONENT slitMS3 = Circular_slit(radius = 0.025)
  AT (0,0,1.130) RELATIVE a2 ROTATED (0,0,0) RELATIVE a2

COMPONENT slitMS4 = Circular_slit(radius = 0.025)
  AT (0,0,1.180) RELATIVE a2 ROTATED (0,0,0) RELATIVE a2

COMPONENT slitMS5 = Circular_slit(radius = 0.0275)
  AT (0,0,1.230) RELATIVE a2 ROTATED (0,0,0) RELATIVE a2

COMPONENT mon = Monitor(
xmin = -0.025, xmax = 0.025, ymin = -0.0375, ymax = 0.0375)
  AT (0, 0, 1.280) RELATIVE a2 ROTATED (0,0,0) RELATIVE a2

COMPONENT emon1 = E_monitor(
xmin = -0.01, xmax = 0.01, ymin = -0.1, ymax = 0.1,
Emin = 19.25, Emax = 20.75, nchan = 35,
filename = "linup_7_1.vmon")
  AT(0, 0, 1.5) RELATIVE a2 ROTATED (0,0,0) RELATIVE a2

COMPONENT sample = Powder1(
radius = 0.007,
h = 0.015,
q = 1.8049,
d_phi0 = 4,
pack = 1, j = 6, DW = 1,
F2 = 56.8,
Vc = 85.0054, sigma_a = 0.463,
target_x = alu_focus_x,  /* ToDo: GET_X(ana) */
target_y = 0, target_z = 1000)
  AT (0, 0, 1.565) RELATIVE a2 ROTATED (0,0,0) RELATIVE a2

COMPONENT a3 = Arm()
  AT (0,0,0) RELATIVE sample ROTATED (0, TT, 0) RELATIVE a2

COMPONENT c2 = Soller(
xmin = -0.02, xmax = 0.02, ymin = -0.0315, ymax = 0.0315,
len = 0.300, divergence = C2)
  AT (0, 0, 0.370) RELATIVE a3 ROTATED (0,0,0) RELATIVE a3

COMPONENT ana = Monochromator(
zmin = -0.0375, zmax = 0.0375,
ymin = -0.024, ymax = 0.024,
mosaich = tas1_ana_mosaic, mosaicv = tas1_ana_mosaic,
r0 = tas1_ana_r0, Q = tas1_ana_q)
  AT (0, 0, 0.770) RELATIVE a3 ROTATED (0, OMA, 0) RELATIVE a3

COMPONENT a4 = Arm()
  AT (0, 0, 0) RELATIVE ana ROTATED (0, TTA, 0) RELATIVE a3

COMPONENT c3 = Soller(
xmin = -0.02, xmax = 0.02, ymin = -0.05, ymax = 0.05,
len = 0.270, divergence = C3)
  AT (0,0,0.104) RELATIVE a4 ROTATED (0,0,0) RELATIVE a4

COMPONENT sng = Monitor(
xmin = -0.01, xmax = 0.01, ymin = -0.045, ymax = 0.045)
  AT(0, 0, 0.43) RELATIVE a4 ROTATED (0,0,0) RELATIVE a4

COMPONENT emon2 = E_monitor(
xmin = -0.0125, xmax = 0.0125, ymin = -0.05, ymax = 0.05,
Emin = 19.25, Emax = 20.75, nchan = 35,
filename = "linup_7_2.vmon")
  AT(0, 0, 0.430001) RELATIVE a4 ROTATED (0,0,0) RELATIVE a4

END
```

C.3 Code for the instrument `prisma2`

```
/*******************************************************************************
* Simple simulation of PRISMA2 with RITA-style analyser backend.
*
* Written by Kristian Nielsen and Mark Hagen August 1998.
*
* Demonstrates how the standard components from the component library
* may be easily modified for special purposes; in this case to have
* the individual analyser blades paint a "color" on the neutrons to
* differentiate them in the detector.
*
* Output is in the file "prisma2.tof". The format is ASCII; each
* line consists of the time-of-flight in microseconds followed by seven
* intensities of neutrons from each individual analyser blade.
*
* Examples:
*
*   prisma2 --ncount=2e6 TT=-30 PHA=22 PHA1=-3 PHA2=-2 PHA3=-1 PHA4=0 PHA5=1 PHA6=2 PHA7=3 TTA=44
*   prisma2 --ncount=2e6 TT=-30 PHA=22 PHA1=3 PHA2=2 PHA3=1 PHA4=0 PHA5=-1 PHA6=-2 PHA7=-3 TTA=44
*******************************************************************************/

/* Modified from Monochromator.comp to paint a "color" on the neutron
   if it is scattered. */
DEFINE COMPONENT Monochromator_color
DEFINITION PARAMETERS (zmin, zmax, ymin, ymax, mosaich, mosaicv, r0, Q, color)
SETTING PARAMETERS ()
STATE PARAMETERS (x,y,z,vx,vy,vz,t,s1,s2,p)
DECLARE
  %{
#define DIV_CUTOFF 2            /* ~ 10^-5 cutoff. */
  %}

TRACE
  %{
    double dphi,tmp1,tmp2,tmp3,vratio,phi,theta0,theta,v,cs,sn;
    double old_x = x, old_y = y, old_z = z, old_t = t;
    double dt;

    if(vx != 0.0 && (dt = -x/vx) >= 0.0)
    {
      y += vy*dt; z += vz*dt; t += dt; x = 0.0;

    if (z>zmin && z<zmax && y>ymin && y<ymax)
    {
      /* First: scattering in plane */

      theta0 = atan2(vx,vz);           /* neutron angle to slab */
      v = sqrt(vx*vx+vy*vy+vz*vz);
      theta = asin(Q2V*Q/(2.0*v));               /* Bragg's law */
      if(theta0 < 0)
        theta = -theta;
      tmp3 = (theta-theta0)/(MIN2RAD*mosaich);
      if(tmp3 > DIV_CUTOFF)
      {
        x = old_x; y = old_y; z = old_z; t = old_t;
      }
      else
      {
        p *= r0*exp(-tmp3*tmp3*4*log(2)); /* Use mosaics */
        tmp1 = 2*theta;
        cs = cos(tmp1);
        sn = sin(tmp1);
        tmp2 = cs*vx - sn*vz;
        vy = vy;
        vz = cs*vz + sn*vx;
        vx = tmp2;

        /* Second: scatering out of plane.
           Approximation is that Debye-Scherrer cone is a plane */

        phi = atan2(vy,vz);                            /* out-of plane angle */
        dphi = (MIN2RAD*mosaicv)/(2*sqrt(2*log(2)))*randnorm();  /* MC choice: */
        /* Vertical angle of the crystallite */
        vy = vz*tan(phi+2*dphi*sin(theta));
        vratio = v/sqrt(vx*vx+vy*vy+vz*vz);
        vz = vz*vratio;
        vy = vy*vratio;                             /* Renormalize v */
        vx = vx*vratio;
        neu_color = color;
      }
    }
    else
    {
      x = old_x; y = old_y; z = old_z; t = old_t;
    }
    }
  %}
MCDISPLAY
  %{
    magnify("zy");
    multiline(5, 0.0, (double)ymin, (double)zmin,
                 0.0, (double)ymax, (double)zmin,
                 0.0, (double)ymax, (double)zmax,
                 0.0, (double)ymin, (double)zmax,
                 0.0, (double)ymin, (double)zmin);
  %}
END


/* Modified from TOF_monitor.comp to bin neutrons according to their
   "color". */
DEFINE COMPONENT TOF_monitor_color
DEFINITION PARAMETERS (xmin, xmax, ymin, ymax, nchan, dt, filename, maxcolor)
SETTING PARAMETERS ()
OUTPUT PARAMETERS (TOF_N, TOF_p, TOF_p2)
STATE PARAMETERS (x,y,z,vx,vy,vz,t,s1,s2,p)
DECLARE
  %{
    int TOF_N[maxcolor+1][nchan];
    double TOF_p[maxcolor+1][nchan];
    double TOF_p2[maxcolor+1][nchan];
  %}
INITIALIZE
  %{
    int i,c;

    for (i=0; i<nchan; i++)
      for (c=0; c<=maxcolor; c++)
      {
        TOF_N[c][i] = 0;
        TOF_p[c][i] = 0;
        TOF_p2[c][i] = 0;
      }
  %}
TRACE
  %{
    int i;

    PROP_Z0;
    if (x>xmin && x<xmax && y>ymin && y<ymax)
    {
      i = floor(1E6*t/dt);              /* Bin number */
      if(i >= nchan) i = nchan;
      if(i < 0)
      {
        printf("FATAL ERROR: negative time-of-flight.\n");
        exit(1);
      }
      if(neu_color < 0 || neu_color > maxcolor)
      {
        printf("FATAL ERROR: wrong color neutron.\n");
        exit(1);
      }
      TOF_N[neu_color][i]++;
      TOF_p[neu_color][i] += p;
      TOF_p2[neu_color][i] += p*p;
    }
  %}
FINALLY
  %{
    DETECTOR_OUT_2D(
        "Time-of-flight monitor",
        "Neutron \"color\"",
        "Time-of-flight [us]",
        0, maxcolor+1, 0, nchan*10,
        maxcolor+1, nchan,
        &TOF_N[0][0],&TOF_p[0][0],&TOF_p2[0][0],
        filename);
  %}

MCDISPLAY
  %{
    magnify("xy");
    multiline(5, (double)xmin, (double)ymin, 0.0,
                 (double)xmax, (double)ymin, 0.0,
                 (double)xmax, (double)ymax, 0.0,
                 (double)xmin, (double)ymax, 0.0,
                 (double)xmin, (double)ymin, 0.0);
  %}
END


DEFINE INSTRUMENT
  prisma2(TT,PHA,PHA1,PHA2,PHA3,PHA4,PHA5,PHA6,PHA7,TTA)

DECLARE
%{
int neu_color;                  /* "Color" of current neutron */

/* 30' mosaicity used on analysator */
double prisma_ana_mosaic = 30;
/* Q vector for bragg scattering with monochromator and analysator */
double prisma_ana_q = 1.87325;
double prisma_ana_r0 = 0.6;
double focus_x,focus_z;

double apos1, apos2, apos3, apos4, apos5, apos6, apos7;
%}

INITIALIZE
%{
  focus_x = 0.52 * sin(TT*DEG2RAD);
  focus_z = 0.52 * cos(TT*DEG2RAD);
  /* Rita-style analyser. */
  {
    double l = 0.0125;
    apos1 = -3*l;
    apos2 = -2*l;
    apos3 = -1*l;
    apos4 =  0*l;
    apos5 =  1*l;
    apos6 =  2*l;
    apos7 =  3*l;
  }
%}

TRACE

COMPONENT mod = Moderator(
                         radius = 0.0707,
                         dist = 9.035,
                         xw = 0.021,
                         yh = 0.021,
                         E0 = 10, E1 = 15,
                         Ec = 9.0, t0 = 37.15, gam = 39.1)
     AT (0,0,0) ABSOLUTE

/* Use a slit to get the effect of a rectangular source. */
COMPONENT modslit = Slit(xmin = -0.05, xmax = 0.05,
                        ymin = -0.05, ymax = 0.05)
     AT(0,0,0.000001) RELATIVE mod

COMPONENT tof_test = TOF_monitor(xmin = -0.05, xmax = 0.05,
                                ymin = -0.05, ymax = 0.05,
                                nchan = 500, dt = 1,
                                filename = "prisma2.mon")
     AT (0,0,0.005) RELATIVE mod

COMPONENT mon1 = Monitor(xmin = -0.1, xmax = 0.1, ymin = -0.1, ymax = 0.1)
     AT(0,0,0.01) RELATIVE mod ROTATED (0,0,0) RELATIVE mod

COMPONENT slit1 = Slit(xmin = -0.05, xmax = 0.05,
                         ymin = -0.05, ymax = 0.05)
     AT(0,0,1.7) RELATIVE mod

COMPONENT slit2 = Slit(xmin = -0.02, xmax = 0.02,
                         ymin = -0.03, ymax = 0.03)
     AT(0,0,7) RELATIVE slit1

COMPONENT mon2 = Monitor(xmin = -0.1, xmax = 0.1, ymin = -0.1, ymax = 0.1)
     AT(0,0,9) RELATIVE mod

COMPONENT sample = V_sample(
        radius_i = 0.00001, radius_o = 0.01,
        h = 0.02,
        focus_r = 0.03,
        pack = 1,
        target_x = focus_x, target_y = 0, target_z = focus_z)
  AT (0, 0, 9.035) RELATIVE mod

COMPONENT a2 = Arm() AT (0,0,0) RELATIVE sample ROTATED (0,TT,0) RELATIVE sample

COMPONENT mon3 = Monitor(xmin = -0.1, xmax = 0.1, ymin = -0.1, ymax = 0.1)
     AT(0,0,0.39) RELATIVE a2

COMPONENT coll2 = Soller(xmin = -0.015, xmax = 0.015,
                         ymin = -0.025, ymax = 0.025,
                         len = 0.12, divergence = 120)
     AT(0,0,0.40) RELATIVE a2

COMPONENT mon4 = Monitor(xmin = -0.1, xmax = 0.1, ymin = -0.1, ymax = 0.1)
     AT(0,0,0.521) RELATIVE a2

COMPONENT rita_ana = Arm()
     AT(0, 0, 0.58) relative a2 ROTATED (0, PHA, 0) RELATIVE a2

COMPONENT ana1 = Monochromator_color(
        ymin=-0.0375,ymax=0.0375,zmin=-0.006,zmax=0.006,
        mosaich=prisma_ana_mosaic,mosaicv=prisma_ana_mosaic,
        r0=prisma_ana_r0, Q=prisma_ana_q, color = 0)
  AT (0, 0, apos1) RELATIVE rita_ana
  ROTATED (0, PHA1, 0) RELATIVE rita_ana

COMPONENT ana2 = Monochromator_color(
        ymin=-0.0375,ymax=0.0375,zmin=-0.006,zmax=0.006,
        mosaich=prisma_ana_mosaic,mosaicv=prisma_ana_mosaic,
        r0=prisma_ana_r0, Q=prisma_ana_q, color = 1)
  AT (0, 0, apos2) RELATIVE rita_ana
  ROTATED (0, PHA2, 0) RELATIVE rita_ana
```

```
COMPONENT ana3 = Monochromator_color(
        ymin=-0.0375,ymax=0.0375,zmin=-0.006,zmax=0.006,
        mosaich=prisma_ana_mosaic,mosaicv=prisma_ana_mosaic,
        r0=prisma_ana_r0, Q=prisma_ana_q, color = 2)
  AT (0, 0, apos3) RELATIVE rita_ana
  ROTATED (0, PHA3, 0) RELATIVE rita_ana

COMPONENT ana4 = Monochromator_color(
        ymin=-0.0375,ymax=0.0375,zmin=-0.006,zmax=0.006,
        mosaich=prisma_ana_mosaic,mosaicv=prisma_ana_mosaic,
        r0=prisma_ana_r0, Q=prisma_ana_q, color = 3)
  AT (0, 0, apos4) RELATIVE rita_ana
  ROTATED (0, PHA4, 0) RELATIVE rita_ana

COMPONENT ana5 = Monochromator_color(
        ymin=-0.0375,ymax=0.0375,zmin=-0.006,zmax=0.006,
        mosaich=prisma_ana_mosaic,mosaicv=prisma_ana_mosaic,
        r0=prisma_ana_r0, Q=prisma_ana_q, color = 4)
  AT (0, 0, apos5) RELATIVE rita_ana
  ROTATED (0, PHA5, 0) RELATIVE rita_ana

COMPONENT ana6 = Monochromator_color(
        ymin=-0.0375,ymax=0.0375,zmin=-0.006,zmax=0.006,
        mosaich=prisma_ana_mosaic,mosaicv=prisma_ana_mosaic,
        r0=prisma_ana_r0, Q=prisma_ana_q, color = 5)
  AT (0, 0, apos6) RELATIVE rita_ana
  ROTATED (0, PHA6, 0) RELATIVE rita_ana

COMPONENT ana7 = Monochromator_color(
        ymin=-0.0375,ymax=0.0375,zmin=-0.006,zmax=0.006,
        mosaich=prisma_ana_mosaic,mosaicv=prisma_ana_mosaic,
        r0=prisma_ana_r0, Q=prisma_ana_q, color = 6)
  AT (0, 0, apos7) RELATIVE rita_ana
  ROTATED (0, PHA7, 0) RELATIVE rita_ana


COMPONENT a3 = Arm()
     AT (0,0,0) relative rita_ana ROTATED (0,TTA,0) RELATIVE a2

COMPONENT mon5 = Monitor(xmin = -0.05, xmax = 0.05, ymin = -0.05, ymax = 0.05)
     AT(0,0,0.06) RELATIVE a3

COMPONENT mon6 = Monitor(xmin = -0.1, xmax = 0.1, ymin = -0.1, ymax = 0.1)
     AT(0,0,0.161) RELATIVE a3

COMPONENT psd = PSD_monitor(xmin = -0.05, xmax = 0.05,
                            ymin = -0.05, ymax = 0.05,
                            nx = 100, ny = 100,
                            filename = "prisma2.psd")
     AT(0,0,0.20) RELATIVE a3

COMPONENT detector = TOF_monitor_color(xmin = -0.05, xmax = 0.05,
                                 ymin = -0.05, ymax = 0.05,
                                 nchan = 10000, dt = 10, maxcolor = 6,
                                 filename = "prisma2.tof")
     AT (0,0,0.20) RELATIVE a3

COMPONENT mon9 = Monitor(xmin = -0.1, xmax = 0.1, ymin = -0.1, ymax = 0.1)
     AT(0,0,0.01) RELATIVE detector

END
```

Appendix D

Test results

In this Appendix, we present a few illustrative results from the three instruments presented in section 6. A more thorough presentation may be found on the McStas home page [1].

D.1 Scattering from the V-sample test instrument

In figure D.1, we present the radial distribution of the scatting from an evenly illuminated V-sample, as seen by a spherical PSD. It is interesting to note that the variation in the scattering intensity is as large as 10%. This is an effect of attenuation of the beam in the cylindrical sample.

D.2 Simulated and measured resolution of TAS1

In order to test the McStas package on a qualitative level, we have performed a very detailed simulation of the conventional triple axis spectrometer TAS1, Risø. The measurement series constitutes a complete alignment of the spectrometer, using the direct beam and scattering from V and Al_2O_3 samples at an incoming energy of 20.0 meV, using the second order scattering from the monochromator. In the instrument definitions, we have used all available information about the spectrometer. However, the mosaicities of the monochromator and analyser are set to 45' in stead of the quoted 30', since we from our analysis believe this to be much closer to the truth.

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

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

In contrast, a simulated $2\theta_a$ scan in triple-axis mode on a V-sample showed a surprising offset from zero, see Figure D.3. 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

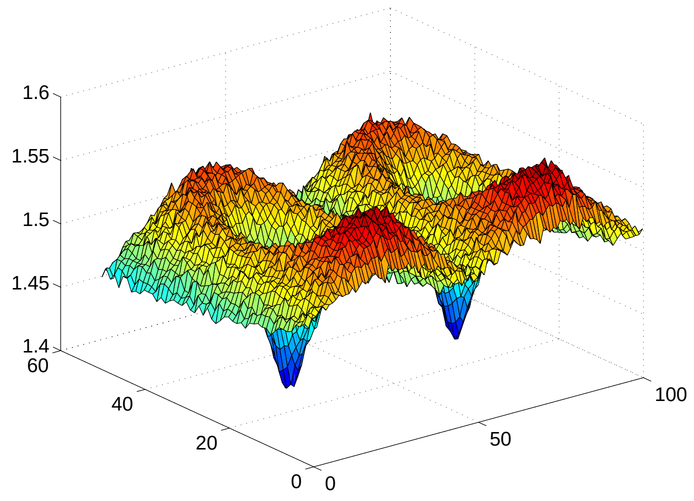


Figure D.1: Scattering from a V-sample, measured by a spherical PSD. The sphere has been transformed onto a plane and the intensity is plotted as the third dimension. A colour version of this picture is found on the title page of this manual.

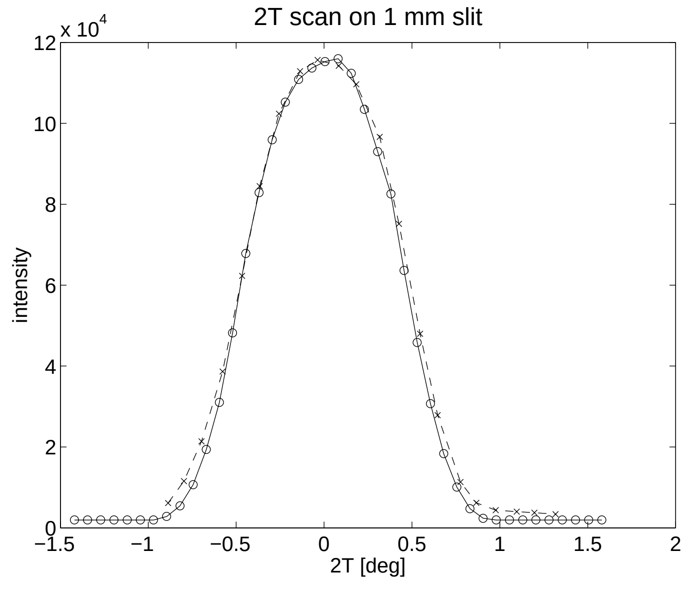


Figure D.2: Scans of $2\theta_s$ in the direct beam with 1 mm slit on the sample position. "$\times$": measurements, "o": simulations Collimations: open-30'-open-open.

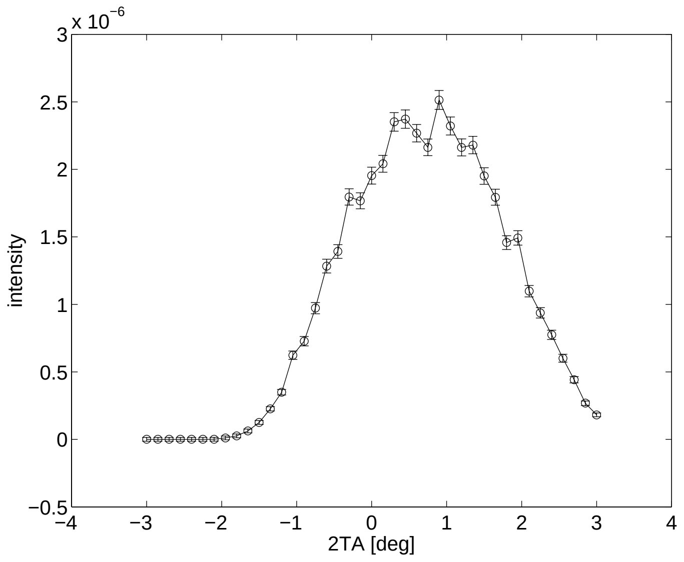


Figure D.3: First simulated $2\theta_a$ scan on a vanadium sample. Collimations: open-30'-28'-open.

comparison on a slightly different instrument (analyser-detector collimator inserted), see Figure D.4.

The result of a $2\theta_s$ scan on an Al_2O_3 powder sample in two-axis mode is shown in Figure D.5. 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 D.6.

D.3 Simple spectra from the PRISMA instrument

A plot from the detector in the PRISMA simulation is shown in Figure D.7. 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 at this time.

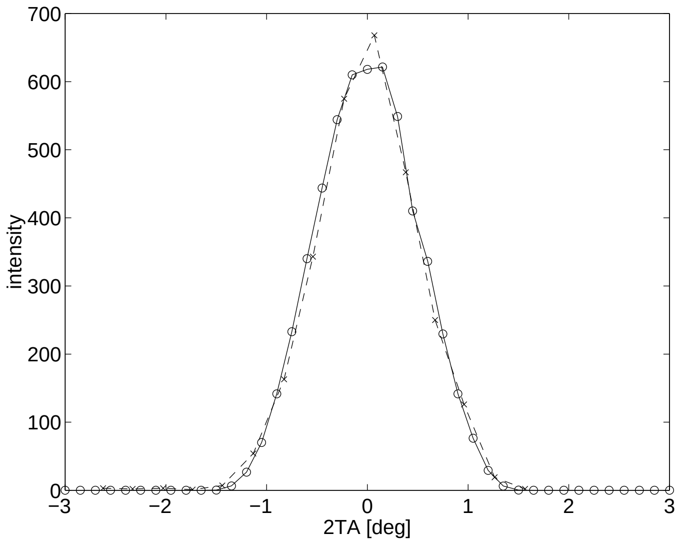


Figure D.4: Corrected $2\theta_a$ scan on a V-sample. Collimations: open-30'-28'-67'. "$\times$": measurements, "o": simulations.

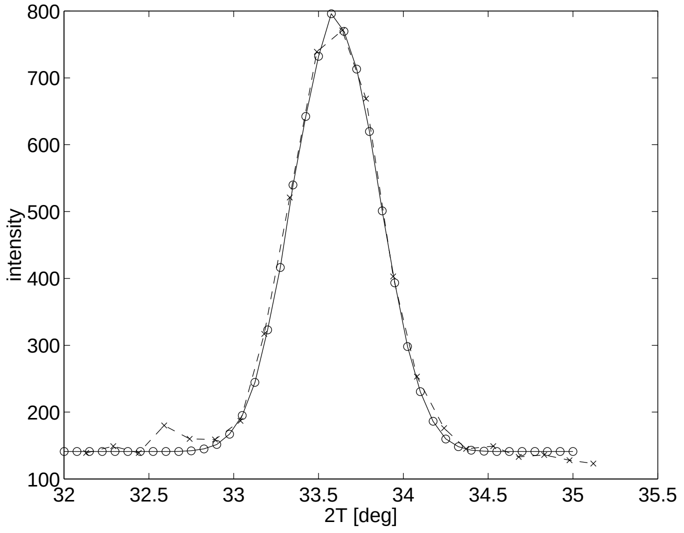


Figure D.5: $2\theta_s$ scans on Al_2O_3 in two-axis, focusing mode. Collimations: open-30'-28'-67'. "$\times$": measurements, "o": simulations. A constant background is added to the simulated data.

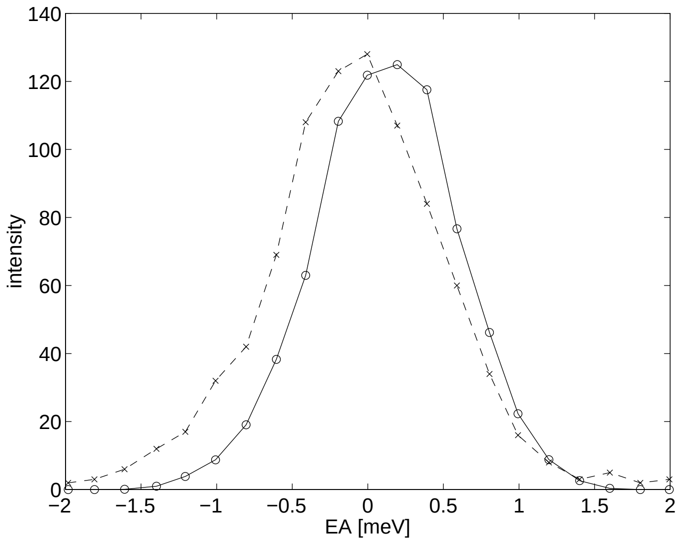


Figure D.6: Scans of the analyser energy on a V-sample. Collimations: open-30'-28'-67'. "×": measurements, "o": simulations.

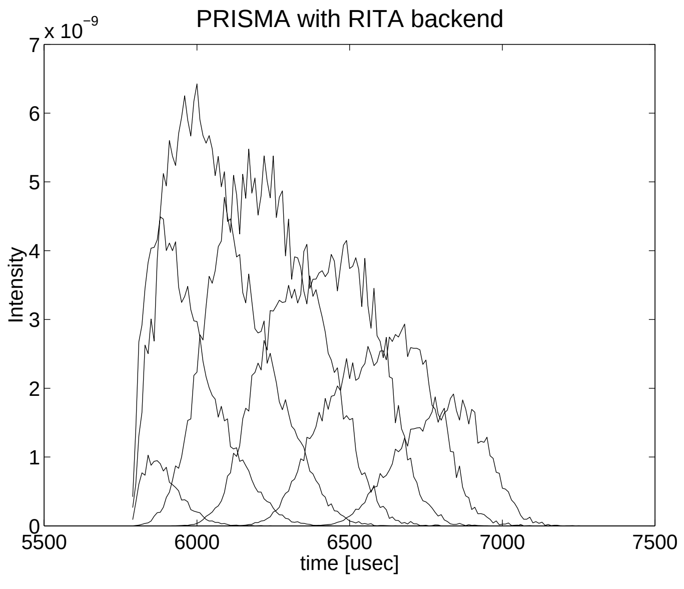


Figure D.7: Test result from PRISMA instrument using "coloured neutrons". Each graph shows the neutrons scattered from one analyser blade.

Appendix E

The McStas terminology

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

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

Bibliography

[1] McStas WWW home page:
`http://neutron.risoe.dk/mcstas/`.

[2] K. Lefmann and K. Nielsen, *McStas, a general software package for neutron ray-tracing simulations*, Newtron News **10/3** (1999), 20–24

[3] T.E. Mason *et al.*, *RITA: The reinvented triple axis spectrometer*, Can. J. Phys. **73** (1995) 697–702

[4] K.N. Clausen *et al.*, *The Rita spectrometer at Risø — design considerations and recent results*, Physica B 241–243 (1998) 50–55.

[5] ESS WWW home page:
`http://www.kfa-juelich.de/ess/ess.html`

[6] Debian GNU/Linux WWW home page:
`http://www.debian.org/`.

[7] NeXus WWW home page:
`http://www.neutron.anl.gov/nexus/`.

[8] MFit WWW home page:
`http://www.risoe.dk/fys/Manuals/Matlab/Mfit/index.html`.

[9] G. L. Squires, *Introduction to the Theory of Thermal Neutron Scattering*, Dover Publications (1996)

[10] Crystallographica v1.50b, Oxford Cryosystems, 1998

[11] G. E. Bacon, *Neutron Diffraction*, Oxford University Press (1975)

[12] A. Abrahamsen, N. B. Christensen, and E. Lauridsen, *McStas simulations of the TAS1 spectrometer*, Students Report, Niels Bohr Institute, University of Copenhagen (1998)

[13] William H. Press, Saul A. Teukolsky, William T. Vetterling, Brian P. Flannery, *Numerical Recipes in C*, Cambridge University Press (1996)

Bibliographic Data Sheet			**Risø–R–1175(EN)**
Title and author(s)			
User and Programmers Guide to the Neutron Ray-Tracing Package McStas, Version 1.2			
Kristian Nielsen, Kim Lefmann			
ISBN			ISSN
87–550–2683–4; 87–550–2684–2 (Internet)			0106–2840
Dept. or group			Date
Condensed Matter Physics and Chemistry Department			February 2000
Groups own reg. number(s)		Project/contract No.	
—		—	
Pages	Tables	Illustrations	References
176	1	19	13

Abstract (Max. 2000 char.)

The software package McStas is a tool for writing Monte Carlo ray-tracing simulations of neutron scattering instruments with very 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 as well as the design of new instrumentation. 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 give essentially unlimited freedom to handle more unusual needs.
This report constitutes the reference manual for McStas, and contains full documentation for all ascpects of the program. It covers the various ways to compile and run simulations; a description of the metalanguage used to define simulations; a full description of all algorithms used to calculate the effects of the various optical components in instruments; and some example simulations performed with the program.

Descriptors

Neutron Instrumentation; Monte Carlo Simulation; Software

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