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

User and Programmers' Guide to the X Ray-Tracing Package McXtrace, version 3.5

E. B. Knudsen, P. Willendrup, E. Farhi, K. Lefmann, S. Schmidt

September, 2024

The software package McXtrace is a tool for carrying out highly complex Monte Carlo ray-tracing simulations of X-ray beamlines to high precision. The simulations can compute all aspects of the performance of instruments and can thus be used to optimize the use of existing equipment, design new instrumentation, and carry out virtual experiments for e.g. training, experimental planning or data analysis. McXtrace is based is based on a unique design, inhereted from its sister McStas, where an automatic compilation process translates high-level textual instrument descriptions into efficient ANSI-C code. This design makes it simple to set up typical simulations and also gives essentially unlimited freedom to handle more unusual cases.

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

This report documents McXtrace version 3.5, released September, 2024

The authors are:

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

Peter Kjær Willendrup `<peter.willendrup@risoe.dk>`
Physics Department, Technical University of Denmark, Kgs. Lyngby, Denmark.

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

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

other people connected to the project:

Jana Baltser `<jana.baltser@fys.ku.dk>`
Niels Bohr Institute, University of Copenhagen, Denmark

Andrea Prodi `<aprodi@fys.ku.dk>`
Niels Bohr Institute, University of Copenhagen, Denmark

Manuel Sanchez del Rio
Computer Science Department, ESRF, Grenbole, France

Claudio Ferrero
Computer Science Department, ESRF, Grenbole, France

Contents

Preface and acknowledgements

This document contains information on the Monte Carlo x-ray tracing program McXtrace version 3.5, building on the initial release in October 1998 of the neutron ray tracing program McStas version 1.0 as presented in Ref. [LN99]. The reader of this document is expected to have some knowledge of x-ray and/or neutron scattering, whereas only little knowledge about simulation techniques is required. In a few places, we also assume familiarity with the use of the C programming language and UNIX/Linux.

Support has been kindly given by SAXLAB Aps. through its CEO Karsten Joensen as well the ESRF and MAX IV Laboratory. We acknowledge all contributing parties.

In case of errors, questions, or suggestions, please do not hesitate to contact the team and the community by either writing to the user mailing list `mcxtrace-users@mcxtrace.org`, consulting the McXtrace home page [Mcx], or leaving a note on the McXtrace facebook page `https://www.facebook.com/McXtrace`. A special bug/request reporting service is available [Mcz].

If you **appreciate** this software, please subscribe to the `mcxtrace-users@mcxtrace.org` email list, send us a smiley message, and contribute to the package.

We encourage you to refer to this software when publishing result with the following citation: E. B. Knudsen, et. al., *Journal of Applied Crystallography*, vol. 46, 2013.

Third party software included in the distribution McXtrace is:

- perl Math::Amoeba from John A.R. Williams `J.A.R.Williams@aston.ac.uk`.
- perl Tk::Codetext from Hans Jeuken `haje@toneel.demon.nl`.
- and optionally PGPLOT from Tim Pearson `tjp@astro.caltech.edu`.

The McXtrace project was initially supported by Det Strategiske Forskningsråd through the NaBiIT programme. Partners in this joint venture were:

- Materials Research Division, Risø DTU, Roskilde, Denmark.
- Niels Bohr Institute, University of Copenhagen, Denmark.
- Faculty of Life Sciences, University of Copenhagen, Denmark.
- SAXLAB ApS. Lundtofte, Denmark.
- ESRF, Grenoble, France.

1. Introduction to McXtrace

Efficient design and optimization of x-ray beamlines are formidable challenges. In many cases Monte Carlo techniques are well matched to meet these challenges. McXtrace is an effort to port the framework provided to the neutron scattering community by the McStas package, to x-ray scattering. McStas has, since its inception in October '98, been used successfully at all major neutron scattering facilities in the world. A particular power of McStas is its geometry engine, which has been adopted by McXtrace, with little modification. The device library of McXtrace is not yet as complete as its sibling McStas but is growing rapidly. At the time of writing it is complete enough to be able to perform simulations approximating any standard beamline.

McXtrace is a fast and versatile software tool. It is based on a meta-language specially designed for x-ray (and neutron) simulation. Specifications are written in this language by users and automatically translated into efficient simulation codes in ANSI-C. The present version supports both continuous and pulsed source beamlines, and includes a library of standard components with total around 100 components.

The McXtrace package is written in ANSI-C and is freely available for download from the McXtrace website [Mcx]. The package is actively developed and supported by Physics Department, Technical University of Denmark, Kgs. Lyngby, Denmark., Niels Bohr Institute, University of Copenhagen, Copenhagen, Denmark., European Synchrotron Radiation Facility, Grenoble, France.and SAXSLAB Aps. The system is tested and is supplied with examples and documentation. Besides this manual, a separate component manual exists which details each individual component separately.

1.1. Development of Monte Carlo x-ray simulation

Monte Carlo simulation of x-ray instrumentation has been used for many years — in particular the SHADOW package [WCC94; Rio+11] has found widespread use, and is still actively developed, yet has some limitations. *Ray*[Sch08] and *Xtrace*[Bau+07] are other packages using the same basic techniques.

What sets McXtrace apart is, among other things, its open source development strategy and its inherent modularity. This combination lets independent scientists work indepently on separate modules that they have use for, and contribute to the whole with only a small effort.

1.2. Scientific background

What makes scientists happy? Probably to collect good quality data, pushing beamlines to their limits, and fit that data to physical models. Among available measurement techniques, x-ray scattering provides a large variety of beamlines to probe structure and dynamics of all kinds of samples.

Achieving a satisfactory experiment on the best beamline is not all. Once collected, the data analysis process raises some questions concerning the signal: what is the background signal? What proportion of coherent and incoherent scattering has been measured? What are the contributions from the sample geometry, the container, the sample environment, and generally the beamline itself? And last but not least, how does multiple scattering affect the signal? Most of the time, the physicist will elude these questions using rough approximations, or applying analytical corrections [Cop+86]. Monte-Carlo techniques provide a mean to evaluate some of these quantities. The technicalities of Monte-Carlo simulation techniques are explained in detail in chapter 4.

1.2.1. The goals of McXtrace

Initially, the McStas project and hence also the present subject the McXtrace project had four main objectives that determined its design.

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

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

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

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

1.3. The design of McXtrace

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

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

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

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

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

4. Presentation of the calculations, graphical or otherwise.

Though obviously interrelated, these four layers can be treated independently, and this is reflected in the overall system architecture of McXtrace. The user will in many situations be interested in knowing the details only in some of the layers. For example, one user may merely look at some results prepared by others, without worrying about the details of the calculation. Another user may simulate a new instrument without having to reinvent the code for simulating the individual components in the instrument. A third user may write an intricate simulation of a complex component, e.g. a detailed description of a high resolution fast chopper, and expect other users to easily benefit from his/her work, and so on. McXtrace 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 beamlines 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 McXtrace internals to write new component or beamline definitions, and even if those internals change in later versions of McXtrace, existing definitions can be used without modification.

The McXtrace system also utilizes the meta-language to let the McXtrace 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 beamline or component. *Efficiency* is also improved by letting the compiler optimize the generated code in ways that would be time-consuming or difficult for humans to do. Furthermore, the compiler can generate several different simulations from the same specification, for example to optimize the simulations in different ways, to generate a simulation that graphically displays x-ray trajectories, and possibly other things in the future that were not even considered when the original instrument specification was written.

The design of McXtrace 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 0.2 mm from the center of the axis" and so on are easy to answer. Within minutes the beamline definition can be modified and a new simulation program generated. It also makes it simple to debug new components. A test beamline definition may be written containing a source, the component to be tested, and whatever monitors are useful, and the component can be thoroughly tested before being used in a complex simulation with many different components.

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

1.4. Overview

The McXtrace system documentation consists of the following major parts:

- A short list of new features introduced in this McXtrace release appears in chapter 2
- Chapter 3 explains how to obtain, compile and install the McXtrace compiler, associated files and supportive software
- Chapter 4 concerns Monte Carlo techniques and simulation strategies in general
- Chapter 5 includes a brief introduction to the McXtrace system (section 5.1) on running the compiler to produce simulations. section 5.2 explains how to run the generated simulations. Running McXtrace on parallel computers requires special attention and is discussed in section 5.5. A number of front-end programs are used to run the simulations and to aid in the data collection and analysis of the results. These user interfaces are described in section 5.3.

- The McXtrace meta-language is described in chapter 6. This chapter also describes a set of library functions and definitions that aid in the writing of simulations. See appendix B for more details.
- The McXtrace component library contains a collection of well-tested, as well as user contributed, beam components that can be used in simulations. The McXtrace component library is documented in a separate manual and on the McXtrace webpage [Mcx], but a short overview of these components is given in chapter 7 of the Manual.

A list of library calls that may be used in component definitions appears in appendix B, and an explanation of the McXtrace terminology can be found in appendix C of the Manual. Plans for future extensions are as a rule presented on the McXtrace webpage [Mcx].

2. New features in McXtrace 3.5

McXtrace is an ongoing evolving project with features being added frequently. While we strive to test it's features thoroughly, bugs are inevitable. Bugs are generally reported using the user-mailing list: mcxtrace-users@mcxtrace.org and subsequently (after initial triage) tracked using the McXtrace Trac system [Mcc] (shared with its sister project McStas). We will not present here an extensive list of improvements and corrections, and we let the reader refer to this bug reporting service for details. Only important changes are indicated here. Of course, we can not guarantee that the software is bullet proof, but we do our best to correct bugs, when they are reported.

The main focus of the 1.2 release has been to improved stability and fixing bug, rather than new developments.

2.1. Kernel

An important update has been made to the approach of negative propagation lengths. Instead of inplicitly absorbing photons these are now generally restored. In many cases the previous behaviour casued confusion and counter-inutitive operation, in particular regarding mutually exclusive monitors with the flag `restore_xray` set.

2.2. Run-time

New intersection routines include `ellipsoid_intersect` and `sphere_intersect`.

2.3. Components and Library

We here list the new and updated components (found in the McXtrace `lib` directory) which are detailed in the *Component manual*. For an overview see the *Component Overview* of the *User Manual*(This Document).

2.3.1. New components

Sources

We have revised the following source models to make them work as conherently as possible while retaining the backwards compatibility.

Source_pt Point source.

Source_flat Flat surface uniform source

Source_div Flat surface uniform source with divergence distribution

Source_gaussian Gaussian crossection source approximating a Bending Magnet

Furthermore we now also supply an experimental model of a bending magnet `Bending_magnet`. The following experimental components have been set up to facilitate interfacing with SPECTRA, Simplex, and GENESIS 1.3. In general they take the output of the relevant external program to get a phase space distribution of photons from which McXtrace samples rays to trace.

Source_genesis Reads the output of a GENESIS 1.3 simulation. Useful for FEL-simulations.

Source_spectra Reads the output of a SPECTRA simulation. Useful for Synchrotron simulations.

Source_simplex Reads the output of a Simplex simulation. Useful for FEL simulations.

Samples

Single_crystal Now has support for wavelength dependent absorption.

Molecule_2state Also includes absorption support.

Molecule_2state Option for supplying a q-parametrized scattering amplitude curve. This is for instance useful to add the effect of a well known solvent (e.g. Water) to a simulation.

SAXS A whole set of SAXS-related samples is now available for various cases.

Optics

Filter Now has the option of taking any shape (through an .off or .ply-file.

Filter Added a refraction-option.

Multilayer_elliptic Elliptical multilayer mirror. This component now has the option of using a analytical kinematical approximation to compute refleticity as opposed to supplying a datafile. This is faster, but often less accurate.

Lens_parab_* The lenses now behave in a similar manner wrt. parameter naming etc., also fixes to an error in the absorption computation. Note that the lens version will be merged in the next release (see the component manual).

Mirror_elliptic Elliptical shape mirror. Major geometry bug-fixes. Is scheduled to be merged with `Mirror_curved`.

Mirror_parabolic Parabolic shape mirror. Major geometry bug-fixes. Is scheduled to be merged with `Mirror_curved`.

Mirror_curved Cylindrical mirror. Fixed a bug which prevented its use as an azimuthal focusing mirror, with sideways incidence.

Chopper_simple Optional random jitter was added.

Mask A component that takes an image file as input, which is used to mask the beam. For instance to exmine linited resolution effects.

Monitors

TOF_monitor New Intensity vs. time of flight monitor. Useful for sime-resolved studies with pulsed sources.

2.3.2. Example instruments

The following beamline models have been added (in addition to unit test models that test specific components):

XFEL_SPB A rough model of the SPB beamline at the Euorpean XFEL.

MAXII_711 A model of the 711 powder diffraction beamline at Maxlab in Lund, Sweden.

MAXII_811 Model of the 811 surface diffraction and EXAFS beamline at Maxlab in Lund, Sweden.

2.4. Tools, installation

2.4.1. Selected Tool features

- standard FreeBSD port
- Possibility to run MPI or grid simulations by default from mxgui.
- We provide syntax-highlighting setup files for eamcs, vim and gedit editors.

2.4.2. Warnings

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

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

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

3. Installing McXtrace

Up to date information on installation procedures on the various supported platforms are available under the installation heading on the McXtrace project webpage: http://www.mcxtrace.org

3.0.1. Platform support

McXtrace is a multiplatform effort. The team policy is to supply native style installation packages for:

Mac OS X The aim is to supply installation packages for the 3 latest releases.

Windows We make packages for system supported by Microsoft.

Linux .deb based distributions.

Linux .rpm based distributions.

FreeBSD Through the ports system.

Plus probably any UNIX/POSIX type environment with a bit of effort, using the source distribution of McXtrace.

The team will be happy to make package builds for other types of systems on request. Please write to the user mailing list `mcxtrace-users@mcxtrace.org` to request a special build.

4. Monte Carlo Techniques and simulation strategy

This chapter explains the simulation strategy and the Monte Carlo techniques used in McXtrace. We first explain the concept of the x-ray weight factor, and discuss the statistical errors in dealing with sums of x-ray weights. Secondly, we give an expression for how the weight factor transforms under a Monte Carlo choice and specialize this to the concept of direction focusing. Finally, we present a way of generating random numbers with arbitrary distributions. More details are available in the Appendix concerning random numbers in the User manual.

4.1. X-ray simulations

X-ray scattering beamlines are built as a series of optical elements. Each of these elements modifies the beam characteristics (e.g. divergence, wavelength spread, spatial and temporal distributions) in a way which, for simple x-ray beam configurations, may be modelled with analytical methods.

However, real x-ray beamlines consist of a large number of optical elements, and this brings additional complexity by introducing strong correlations between x-ray beam parameters like divergence and position - which is the basis of the acceptance diagram method - but also wavelength and time. The usual analytical methods, such as phase-space theory, then reach their limit of validity in the description of the resulting effects.

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

Implementations of the MC method for X-ray beamlines already exist, most notable is probably *SHADOW* [WCC94], originally developed by the late Franco Cerrina and coworkers, now developed further by M. Sanchez Del Rio at the ESRF[Rio+11][Sha]. Other implementations of the same concept are *RAY* [Sch08] from BESSY and *Xtrace*[Bau+07]. hosted at ANKA

4.1.1. Monte Carlo ray tracing simulations

Mathematically, the Monte-Carlo method is an application of the law of large numbers [Jam80; GRR92]. Let $f(u)$ be a finite continuous integrable function of parameter u for which an integral estimate is desirable. The discrete statistical mean value of f (computed as a series) in the uniformly sampled interval $a < u < b$ converges to the mathematical mean value of f over the same interval.

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

In the case were the u_i values are regularly sampled, we come to the well known midpoint integration rule. In the case were the u_i values are randomly (but uniformly) sampled, this is the Monte-Carlo integration technique. As random generators are not perfect, we rather talk about *quasi*-Monte-Carlo technique. We encourage the reader to refer to James [Jam80] for a detailed review on the Monte-Carlo method.

4.2. The x-ray weight

A totally realistic semi-classical simulation will require that each x-ray is at any time either present or lost. On many beamlines the sheer abundance of x-ray photons makes it impractical to trace each and every photon from the source. This is particularly the case at XFELs. Additionally, only a very small fraction of the initial x-rays will ever be detected, and simulations of this kind will therefore waste much time in dealing with x-rays that never hit the detector.

A way of dealing with these issues and speed up calculations is to introduce a x-ray "weight factor" for each simulated ray and to adjust this weight according to the path of the ray. If *e.g.* the reflectivity of a certain optical component is 10%, and only reflected x-rays ray are considered later in the simulations, the x-ray weight will be multiplied by 0.10 when passing this component, but every x-ray is allowed to reflect in the component. In contrast, the totally realistic simulation of the component would require on average ten incoming x-rays for each reflected one.

Let the initial x-ray 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 x-ray ray after passage of the whole beamline becomes the product of all contributions

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

Each adjustement factor should be $0 < \pi_j < 1$, except in special circumstances, so that total flux can only decrease through the simulation. For convenience, the value of p is updated (within each component) during the simulation.

Simulation by weight adjustment is performed whenever possible. This includes

- Transmission through filters and windows.

- Reflection from monochromator (and analyser) crystals with finite reflectivity and mosaicity.
- Reflections from mirrors.
- Passage of a continuous beam through a chopper.
- Scattering from all types of samples.

4.2.1. Statistical errors of non-integer counts

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

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

where N is the number of rays hitting the detector and the horizontal bar denotes averaging. By performing the weight transformations, the (statistical) mean value of I is unchanged. However, N will in general be enhanced, and this will improve the accuracy of the simulation.

To give an estimate of the statistical error, we proceed as follows: Let us first for simplicity assume that all the counted x-ray weights are almost equal, $p_i \approx \overline{p}$, and that we observe a large number of x-rays, $N \geq 10$. Then N almost follows a normal distribution with the uncertainty $\sigma(N) = \sqrt{N}$ [1]. Hence, the statistical uncertainty of the observed intensity becomes

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

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

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

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

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

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

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

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

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

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

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

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

4.3. Weight factor transformations during a Monte Carlo choice

When a Monte Carlo choice must be performed, *e.g.* when the initial energy and direction of the x-ray ray is decided at the source, it is important to adjust the x-ray weight so that the combined effect of x-ray weight change and Monte Carlo probability of making this particular choice equals the actual physical properties we like to model.

Let us follow up on the simple example of transmission. The probability of transmitting the real x-ray is P, but we make the Monte Carlo choice of transmitting the x-ray every time: $f_{\mathrm{MC}} = 1$. This must be reflected on the choice of weight multiplier π_j given by the master equation In the "real" semi-classical world, there is a distribution (probability density) for the x-rays 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 type and its parameters (such as gap, period, field strength etc. for an undulator). In the Monte Carlo simulations, the six coordinates are for efficiency reasons in general picked from another distribution: $f_{\mathrm{MC}}(E, \mathbf{\Omega}, \mathbf{r}) \neq \Pi(E, \mathbf{\Omega}, \mathbf{r})$, since one would *e.g.* often generate only x-rays within a certain parameter interval. However, we must then require that the weights are adjusted by a factor π_j (in this case: $j = 1$) so that

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

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

4.3.1. Direction focusing

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

If *e.g.* a sample scatters isotropically over the whole 4π solid angle, and all interesting directions are known to be contained within a certain solid angle interval $\Delta\mathbf{\Omega}$, only these solid angles are used for the Monte Carlo choice of scattering direction. According to Eq. (4.9), the weight factor will then have to be changed by the amount $\pi_j = |\Delta\mathbf{\Omega}|/(4\pi)$. One thus ensures that the mean simulated intensity is unchanged during a "correct" direction focusing, while a too narrow focusing will result in a lower (*i.e.* wrong) intensity, since we cut x-rays rays that should have reached the final detector.

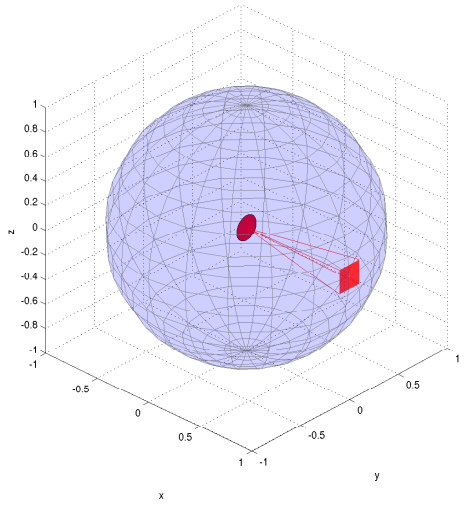

Figure 4.1.: Illustration of the effect of direction focusing in McXtrace. Weights of x-rays emitted into a certain solid angle are scaled down by the full unit sphere area.

4.4. Stratified sampling

One particular efficiency improvement technique is the so-called *stratified sampling*. It consists in partitioning the event distributions in representative sub-spaces, which are then all sampled individualy. The advantage is that we are then sure that each sub-space is well represented in the final integrals. This means that instead of shooting N events, we define D partitions and shoot $r = N/D$ events in each partition. We may define partitions so that they represent 'interesting' distributions, e.g. from events scattered on a monochromator or a sample. The sum of partitions should equal the total space integrated by the Monte Carlo method, and each partition must be sampled randomly.

In the case of McXtrace, the stratified sampling is used when repeating events, i.e. when using the SPLIT keyword in the TRACE section on beamline descriptions. We emphasize here that the number of repetitions r should not exceed the dimensionality of the Monte Carlo integration space (which is $d = 10$ for x-ray events) and the dimen-

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

Table 4.1.: Accuracy estimate as a function of the number of statistical events used to estimate an integral with McXtrace.

sionality of the partition spaces, i.e. the number of random generators following the stratified sampling location in the beamline.

4.5. Accuracy of Monte Carlo simulations

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

McXtrace uses a space with $d = 12$ parameters to describe x-rays (position, wavevector, weight, polarisation, phase, time). We show in Table 4.1 a rough estimate of the accuracy on integrals as a function of the number of records reaching the integration point. This stands both for integrated flux, as well as for histogram bins - for which the number of events per bin should be used for N.

5. Running McXtrace

This chapter describes usage of the McXtrace simulation package. In case of problems regarding installation or usage, the McXtrace mailing list [Mcx] or the authors should be contacted.

Important note for Windows users: It is a known problem that some of the McXtrace tools do not support filenames / directories with spaces. We are working on a more general approach to this problem, which will hopefully be solved in a future release. We recommend to use Strawberry **Perl 5.18**. This distribution of perl also includes a free c-compiler.

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

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

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

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

The output data may be analyzed and visualized in the same way as regular experiments by using the data handling and visualisation tools in McXtrace based on Perl and Matlab or PGPLOT. Further data output formats including NeXus and XML are available, see section 5.4.

To start the graphical user interface of McXtrace, run the command `mxgui` (`mxgui.pl` on Windows). This will open a window with a number of menus, see figure 5.1.

To load an instrument, select "Maxlab" from the "x-ray site" menu and open the file `MAXII_711.instr`. Next click the "Run" button to start the simulation. This triggers a compilation of the instrument file and pops up a dialog window. For instance type a value for the "R"-parameter (This corresponds to the curvature of the focusing mirror), 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. If you use the default PGPLOT plotting back-end, the presented graphics will resemble that shown in figure 5.

To visualize or debug the simulation graphically, repeat the steps but check the "Trace" option instead of the "Simulate" option. A window will pop up showing a sketch of the instrument. Depending on your chosen plotting backend, the presented graphics will resemble one of those shown in figures 5.3-5.4.

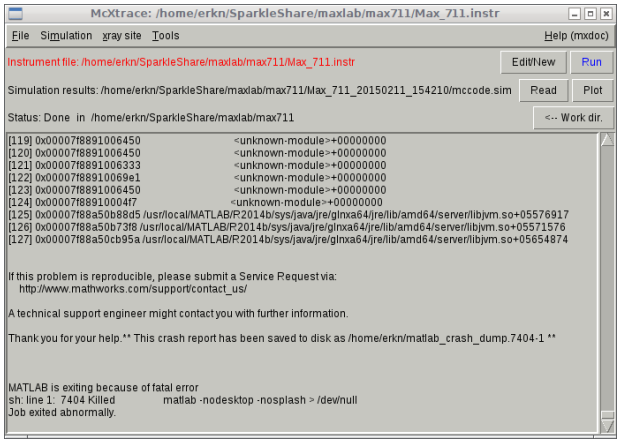

Figure 5.1.: The graphical user interface `mxgui.pdf`.

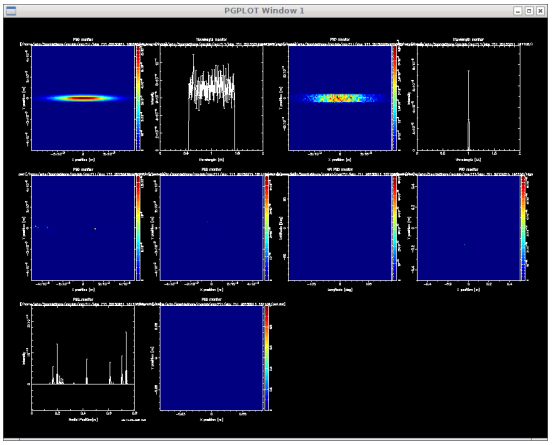

Figure 5.2.: Output from `mxplot` with the PGPLOT backend

5.1. Running the instrument compiler

This section describes how to run the McXtrace compiler manually. Often, it will be more convenient to use the front-end program `mxgui` or `mxrun` (section 5.3.2). These front-ends will compile and run the simulations automatically.

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

```
mcxtrace name.instr
```

in a shell or command prompt. On windows it is convenient to launch the command prompt through the desktop icon named mcxtrace-shell. This is a utility script that sets up the necessary environment variables as appropriate.

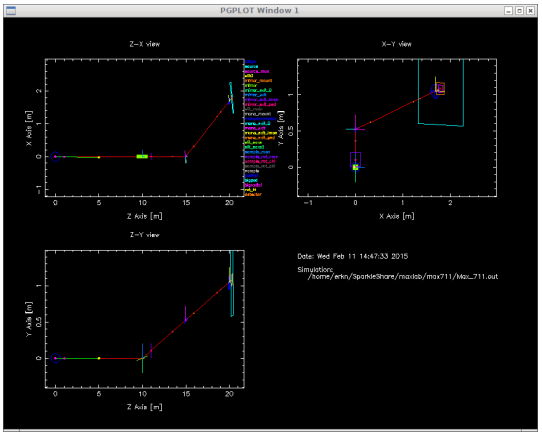

Figure 5.3.: Output from `mxdisplay` with PGPLOT backend. The left mouse button starts a new photon ray, the middle button zooms, and the right button resets the zoom. The Q key quits the program. See section 5.3.3 for details.

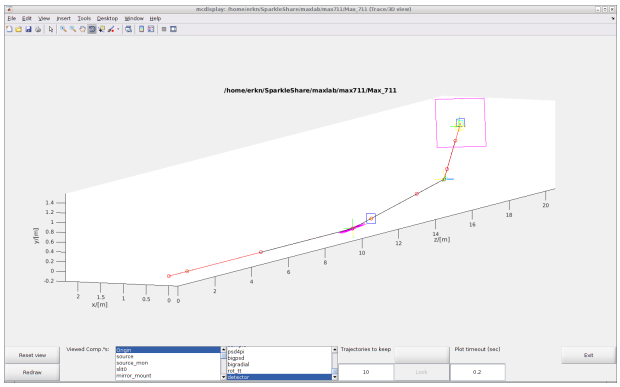

Figure 5.4.: Output from `mxdisplay` with Matlab backend. Display can be adjusted using the window buttons.

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

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

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

5.1.1. Code generation options

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

The `-t` or `--trace` option puts special "trace" code in the output. This code makes it possible to get a complete trace of the path of every photon ray through the instrument, as well as the position and orientation of every component. This option is mainly used with the `mxdisplay` front-end as described in section 5.3.3.

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

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

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

5.1.2. Specifying the location of files

The McXtrace compiler needs to be able to find various files during compilation, some explicitly requested by the user (such as component definitions and files referenced by `%include`), and some used internally to generate the simulation executable. McXtrace 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 McXtrace directory. Usually, the user will not need to worry about this as McXtrace will automatically find the required files. But if users build their own component library in a separate directory or if McXtrace is installed in an unusual way, it may be necessary to tell the compiler where to look for the files.

The location of the special McXtrace directory is set when McXtrace is compiled. It defaults to `/usr/local/mcxtrace/VERSION/` on Unix-like systems and `C:\mcxtrace-VERSION\lib` on Windows systems, but it can be changed to something else during the installation process. The location can be overridden by setting the environment variable `MCXTRACE`:

```
setenv MCXTRACE /home/joe/mcxtrace
```

for csh/tcsh users, or

```
export MCXTRACE=/home/joe/mcxtrace
```

for bash/Bourne shell users. Windows users, that do not wish to use the desktop shortcut `mcxtrace-shell-1.2`, should define the `MCXTRACE` from the menu 'Start/Settings/Control Panel/System/Advanced/Environment Variables' by creating `MCXTRACE` with the value `C:\mcxtrace-1.2\lib`

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

```
mcxtrace -I/home/joe/components -I/home/joe/xrays/include name.instr
```

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

5.1.3. Embedding the generated simulations in other programs

By default, McXtrace 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 McXtrace compiler provides the following options:

- `--no-main` This option makes McXtrace omit the `main()` function in the generated simulation program. The user must then arrange for the function `mcxtrace_main()` to be called in some way.
- `--no-runtime` Normally, the generated simulation program contains all the run-time C code necessary for declaring functions, variables, etc. used during the simulation. This option makes McXtrace omit the run-time code from the generated simulation program, and the user must then explicitly link with the file `mcxtrace-r.c` as well as other shared libraries from the McXtrace distribution.

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

5.1.4. Running the C compiler

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

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

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

A `gcc` which is a very portable, open source, ISO-C compatible c compiler, available for most platforms. For Linux it is usually part of your distribution, for Windows the recommended perl distribution (strawberry perl [Str]) includes mingw, a version og gcc for windows. For Mac OS X `gcc` is part of the Xcode tools package available on the installation medium.

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

- `MCXTRACE_CC=icc`
- `MCXTRACE_CFLAGS="-g -O2 -wd177,266,1011,181"`

The latter is to silence a number of warnings. To use `icc` with MPI on Unix system (see Section 5.5) installations, it seems that *editing* the mpicc shell script and setting the CC variable to "`icc`" is the only requirement! On Windows, the Intel c compiler is 'icl', not 'icc' and has a dependency for Microsoft Visual C++. If you have both these softwares available, running McXtrace with the Intel compiler should be possible (currently untested by the McXtrace developer team).

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

Monte Carlo simulations are computationally intensive, and it is often desirable to have them run as fast as possible. Some success can be obtained by adjusting the compiler optimization options.

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. Optimization flags will typically result in a speed improvement by a factor about 3, but the compilation of the instrument may be 5 times slower. Even worse, compiler optimizations are notoriously buggy; some options have been known to generate *incorrect code* in some compiler versions. McXtrace 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, McXtrace simulations generally run quite fast, often fast enough that further optimizations are not worthwhile. Also, optimizations are highly memory consuming during compilation, and thus may fail when dealing with large instrument descriptions (e.g. more that 100 elements). The compilation process is simplified when using components of the library making use of shared libraries (see `SHARE` keyword in chapter 6). Refer to section 5.2.3 for other optimization methods.

5.2. Running the simulations in a shell

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

```
./name.out
```

(Or `name.exe` on windows. In the following we will only list the `.out`)

Note the leading ".", which is needed if the current directory is not in the path searched by the shell. When used in this way, the simulation will prompt for the values of any instrument parameters such as motor positions, and then run the simulation. Default instrument parameter values (see section 6.3), if any, will be indicated and entered when hitting the `Return` key. This way of running McXtrace will only give data for one beamline setting. Sometimes a scan over various beamline settings is required, in which case multiple simulation runs are required. Often the simulation will be run using one of several available front-ends, as described in the next section. These front-ends help manage output from the potentially many monitors 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

```
./Samples_vanadium.out ROT=90
```

The number of photon 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, McXtrace simulations write their results into several data files in the current directory, overwriting any previous files stored there. The `--dir=`*dir* or `-d`*dir* option causes the files to be placed instead in a newly created directory *dir* (to prevent overwriting previous results the simuation is aborted and an error message is issued if the directory already exists). Alternatively, all output may be written to a single file *file* using the `--file=`*file* or `-f`*file* option (which should probably be avoided when saving in binary format, see below). If the `file` is given as `NULL`, the file name is automatically built from the instrument name and a time stamp. The default file name is `mccode` followed by appropriate extension.

The complete list of options and arguments accepted by McXtrace simulations appears in Tables 5.1 and 5.2.

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

The format of the output files from McXtrace simulations is described in more detail in section 5.4. It may be chosen either with `--format=FORMAT` for each simulation or globally by setting the MCXTRACE_FORMAT environment variable. The available format list is obtained using the `name.out --help` option, and shown in Table 5.3. Additionally, adding the `raw` keyword to the FORMAT will produce raw $[N, p, p^2]$ data

`-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 photon histories to simulate. The default is 1,000,000. (1e6)
`-d` *dir* `--dir=`*dir*	Create a new directory *dir* and put all data files in that directory.
`-h` `--help`	Show a short help message with the options accepted, available formats and the names of the parameters of the instrument.
`-i` `--info`	Show extensive information on the simulation and the instrument definition it was generated from.
`-t` `--trace`	This option makes the simulation output the state of every photon as it passes through every component. Requires that the `-t` (or `--trace`) option is also given to the McXtrace compiler when the simulation is generated.
`--no-output-files`	This option disables the writing of data files (output to the terminal, such as detector intensities, will still be written).
`--format=`*FORMAT*	This option sets the file format for result simulation and data files.
`-N` *STEPS*	Divide simulation into STEPS, varying parameters within given ranges 'min,max'.
param=value *min,max*	Set the value of an instrument parameter, rather than having to prompt for each one. Scans ranges are specified as 'min,max'.

Table 5.1.: Options accepted by McXtrace simulations. For options specific to MPI and parallel computing, see section 5.5.

sets instead of $[N, p, \sigma]$ (see Section 4.2.1). The former representation is fully additive, and thus enables to add results from separate simulations (e.g. when using a computer Grid - which is automated in the `mxformat` tool). Other acceptable format modifiers are `transpose` to transpose data matrices and `append` to catenate data to existing files.

5.2.1. Basic import and plot of results in matlab

To import a McXtrace ascii data file, while ignoring the header lines, in matlab one may use the following command:

```
matlab> s=textread('plot','CommentStyle','shell');
```

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

`-f` *file* `--file=`*file*	Write all data into a single file *file*. Avoid when using binary formats.
`-a` `--data-only`	Do not put any headers in the data files.
`--format_data=`*FORMAT*	This option sets the file format for result data files from monitors. This enables to have simulation files in one format (e.g. HTML), and monitor files in an other format (e.g. VRML).
`--mpi=`*NB_CPU*	This option will distribute the simulation over NB_CPU nodes (requires MPI to be installed).
`--multi=`*NB_CPU* `--grid=`*NB_CPU*	This option will distribute the simulation over NB_CPU nodes (requires SSH to be installed).
`--machines=`*MACHINES*	Specify a list of distant machines/nodes to be used for MPI and grid clustering. Default is to use local SMP cluster.
`--optim`	Run in optimization mode to find best parameters in order to maximize all monitor integral values. Parameters to be varied are given just like scans (min,max).
`--optim=`*COMP*	Same as `--optim` but for specified monitors. This option may be used more than once.
`--optim-prec=`*ACCURACY*	Sets accuracy criteria to end parameter optimization (default is 10^{-3}).
`--test`	Run McXtrace self test.
`-c` `--force-compile`	Force to recompile the instrument.

Table 5.2.: Additional options accepted by McXtrace simulations.

`McXtrace` `PGPLOT`	.sim	Original format for PGPLOT plotter (may be used with -f and -d options)
`XML`	.xml	XML format, NeXus-like (may be used with -f and -d options).
`HTML`	.html	HTML format (generates a web page, may be used with -f and -d options). Data files are saved as VRML objects (OpenGL).
`VRML`	.wrl	Virtual Reality file format for data files. Simulation files are not saved properly, and the HTML format should be used preferably.
`NeXus`	.nxs	NeXus data files (HDF). All simulation results are stored in a unique compressed binary file. This format requires to have NeXus installed.
McXtrace events		McXtrace event files in text or binary format. Use `Virtual_input`/`Virtual_output` components.

Table 5.3.: Available formats supported by McXtrace simulations. Format modifiers include *raw*, *transpose*, *append*.

5.2.2. Interacting with a running simulation

Once the simulation has started, it is possible, under Unix, Linux and Mac OS X systems, to interact with the on-going simulation. This feature is not available when using MPI parallelization.

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

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

where `user` is your Unix login. In this case *pid* is '13277'.

Once known, it is possible to send one of the signals listed in Table 5.4 using the 'kill' unix command (or the functionalities of your process manager), e.g.

```
kill -USR2 13277
```

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

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

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

Table 5.4.: Signals supported by McXtrace simulations.

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

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

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

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

5.2.3. Optimizing simulation speed

There are various ways to speed up simulations

- Optimize the compilation of the instrument, as explained in section 5.1.4.
- Execute the simulation in parallel on a computer grid or a cluster (with MPI or ssh grid) as explained in section 5.5.
- Complex components usually take into account additional small effects in a simulation, but are much longer to execute. Thus, simple components should be prefered whenever possible, at least in the beginning of a simulation project.
- The SPLIT keyword may artificially repeat events reaching specified positions in the instrument. This is *very* efficient, but requires to cast random numbers in the course of the remaining propogagtion (e.g. at samples, crystals, ...). See section 6.4.7 for details.

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

5.2.4. Optimizing instrument parameters

Often, the user may wish to optimize the parameters of a simulation (e.g. find the optimal curvature of a monochromator, or the best geometry of a given component).

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

Optimization using the Simplex method

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

When using `mxrun` (section 5.3.2), the optimization mode is set by using the `--optim` option or a list of monitors to maximize with as many `--optim=COMP` as required. The optimization accuracy criterion may be changed with the `--optim-prec=accuracy` option.

From `mxgui` (section 5.3.1), one should choose the 'Optimization' execution mode (instead of the Simulation or Trace mode). Then speficy the instrument parameters to optimize by indicating their variation range `param=min,max` (e.g. Lambda=1,4) just like parameter scans. Optionally, the starting guess value might be given with the syntax `param=min,guess,max`. The optimization accuracy criterion is controlled using the 'Precision' entry box in the configuration options (See Figure 5.5). Finally, run the simulation. The optimum set of parameters is then printed at the end of the simulation process. You may ask to maximize only given monitors (instead of all) by selecting their component names in the lower lists in the Run Dialog (up to 3).

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

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

5.3. Using the simulation tool layer

McXtrace includes a number of auxilliary programs that extend the functionality of the simulations. For instance the `mcxrun` front-end program is an interface between the user and the simulations, capable of running series of simulations and storing the results in a structured manner.

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

```
McXtrace Tools
   mxtrace        Main instrument compiler
   mxrun          Instrument build and execution utility
   mxgui          Graphical User Interface instrument builder
   mxdoc          Component library documentation generator/viewer
   mxplot         Simulation result viewer
   mxdisplay      Instrument geometry viewer
   mxformat       Conversion tool for text files and MPI/grids
   mxdaemon       Instrument results on-line plotting
When used with the -h flag, all tools display a specific help.
SEE ALSO: mcxtrace, mxdoc, mxplot, mxrun, mxgui
DOC:      Please visit http://www.mcxtrace.org
```

5.3.1. The graphical user interface (mxgui)

The front-end `mxgui` provides a graphical user interface that interfaces the various parts of the McXtrace package. It is started from a shell/command prompt using simply the command

```
mxgui
```

The mxgui program may optionally be given the name of an instrument file. On windows it is probably more convenient to start mxgui using the desktop icon.

When the front-end is started, a main window is opened (see figure 5.1). 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:

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

File/Edit current opens a simple editor window with McXtrace syntax highlighting for editing the current instrument definition. This function is also available from the **Edit** button to the right of the name of the instrument definition in the main window.

File/Edit current (detached) As above but creates a detached process, such that the editor window remains open after the main windows has been closed.

File/Compile instrument forces a recompile of the instrument definition, regardless of file dates. This is for example useful to pick up changes in component definitions,

which the front-end will not notice automatically. This might also be required when choosing MPI and NeXus options . See chapter 3 for how to override default C compiler options.

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

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

File/Preferences Opens the choose backend dialog shown in figure 5.5. Several settings can be chosen here:

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

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

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

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

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

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

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

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

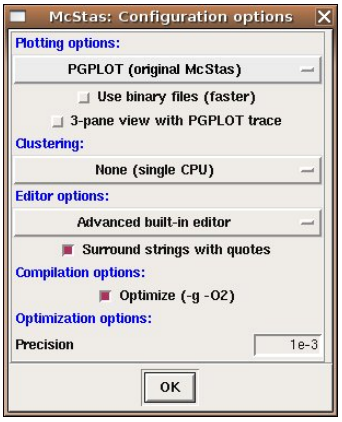

Figure 5.5.: The "configuration options" dialog in `mxgui`.

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

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

Tools/Online plotting of results installs a DSA key to be used for ssh clustering and MPI (see section 5.5).

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

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

Tools/Activate MPI/grid (DSA key) installs a DSA key to be used for ssh clustering and MPI (see section 5.5).

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

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

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

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

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

Help/Tutorial Opens the McXtrace tutorial for a quick start. (Disabled - Please see the McXtrace web page[Mcx] for an updated tutorial).

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

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

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

The run dialog

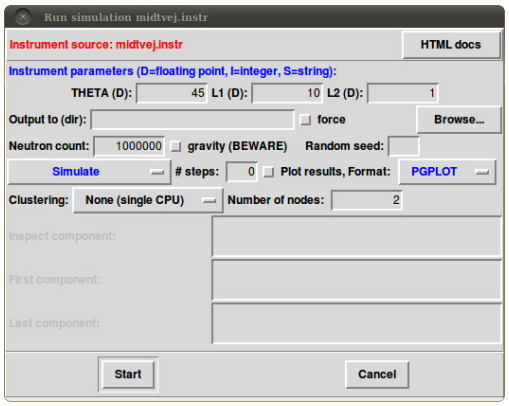

Figure 5.6.: The run dialog in `mxgui`.

The run dialog is used to run simulations. It allows the entry of instrument parameters as well as the specifications of options for running the simulation (see section 5.2 for details). It also allows to run the `mxdisplay` (section 5.3.3) and `mxplot` (section 5.3.4) front-ends together with the simulation.

The meaning of the different fields is as follows:

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

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

Run:Output to allows the entry of a directory for storage of the resulting data files in (like the `--dir` option). If no name is given, a new directory with a unique name is created, to avoid overwriting data.

Run:Force Forces to overwrite existing data files

Photon count sets the number of photon rays to simulate (the `--ncount` option).

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

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

- Simulate: Perform a normal simulation or a scan when #steps is set to non-zero value.
- Trace (3D view): View the instrument in 3D tracing indivitual photons through the instrument
- Simulate (bg): Perform a simulation or scan but put the process in the background. This frees the GUI to continue working while the simulation is running.
- Optimize (bg): Find the optimum value of the simulation parameters in the given ranges (see section 5.2.4). The process is put in the background.

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

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

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

Run:Clustering method Selects the mechanism to be used for running on grids and clusters. See section 5.5 on parallel computing for more informations.

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

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

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

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

Run:Maximize monitor (Optimization mode) Selects up to three monitors whose integral value should be maximized, varying instrument parameters. If none are selected, all monitors are used.

Run:Start runs the simulation.

Run:Cancel aborts the dialog.

Most of the settings on the run dialog are saved for your next McXtrace run.

Before running the simulation, the instrument definition is automatically compiled if it is newer than the generated C file (or if the C file is newer than the executable). The executable is assumed to have a `.out` suffix in the filename. NB: If components are changed, automatic compilation is *not* performed. Use the File/Compile menu item in the main windows.

The editor window

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

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

Editor Insert/Component... opens up a dialog window with a list of all the components available for use in McXtrace. Selecting a component will display a description. Double-clicking will open up a dialog window allowing the entry of the values of all the parameters for the component (figure 5.7). See section 6.3 for details of the meaning of the different fields.

The dialog will also pick up those of the users own components that are present in the current directory when `mxgui` is started. See section 6.7 for how to write components to integrate well with this facility.

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

To use the `mxgui` front-end, the programs Perl and Perl/Tk must be properly installed on the system. Additionally, if the McXtrace/PGPLOT back-end is used for data format, PGPLOT, and PDL will be required. It may be necessary to set the `PGPLOT_DIR` and `PGPLOT_DEV` environment variable; consult the documentation for PGPLOT on the local system in case of difficulty.

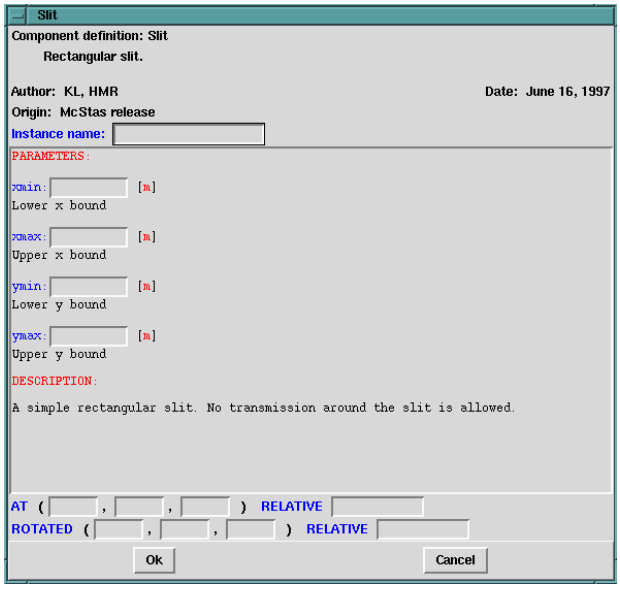

Figure 5.7.: Component parameter entry dialog.

5.3.2. Running simulations on the commandline (mxrun)

The `mxrun` front-end (mxrun.pl on Windows) provides a convenient command-line interface for running simulations with the same automatic compilation features available in the `mxgui` front-end. It also provides a facility for running a series of simulations while varying an input parameter.

The command

`mxrun` *`sim args ...`*

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

The possible arguments are the same as those accepted by the simulations themselves as described in section 5.2, with the following extensions:

- The `-c` or `--force-compile` option may be used to force the recompilation of the instrument definition, regardless of file dates. This may be needed in case any component definitions are changed (in which case `mxrun` does not automatically recompile), or if a new version of McXtrace has been installed.
- The `-p` *`file`* or `--param=`*`file`* option may be used to specify a file containing assignment of values to the input parameters of the instrument definition. The file should consist of specifications of the form *`name=value`* separated by spaces or line breaks. Multiple `-p` options may be given together with direct parameter specifications on the command line. If a parameter is assigned multiple times, later assignments override previous ones.

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

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

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

  After running the simulation, the results will be written to the file `mccode.dat` by default. This file contains one line for each simulation run giving the values of the scanned input variables along with the integrated intensity and estimated error in all monitors.

- When performing a scan, the `-f` *`file`* and `--file=`*`file`* options make `mxrun` write the output to the files *`file`*`.dat` and *`file`*`.sim` instead of the default names.

- When performing a scan, the `-d` *`dir`* and `--dir=`*`dir`* options make `mxrun` put all output in a newly created directory *dir*. Additionally, the directory will have subdirectories `1`, `2`, `3`,... containing all data files output from the different simulations. When the `-d` option is not used, no data files are written from the individual simulations (in order to save disk space).

- The `mxrun --test` command will test your McXtrace installation.

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

5.3.3. Graphical display of simulations (mxdisplay)

The front-end `mxdisplay` (mxdisplay.pl on Windows) is a graphical debugging tool. It presents a schematic drawing of the instrument definition, showing the position of the components and the paths of the simulated photons 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 photons are getting lost. (See figures 5.3-5.4.)

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

```
mxdisplay sim args ...
```

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

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

```
mxdisplay -pmatlab ./Samples_vanadium.out ROT=90
```

or (`csh`/`tcsh` syntax)

```
setenv MCXTRACE_FORMAT matlab
mxdisplay ./Samples_vanadium.out ROT=90
```

will output graphics using matlab. The `mxdisplay` front-end can also be run from the `mxgui` front-end. Examples of plotter appearence for `mxdisplay` is shown in figures 5.3-5.4.

McXtrace/PGPLOT back-end This will view the instrument from above. A multi-display that shows the instrument from three directions simultaneously can be shown using the `--multi` or `-m` options:

```
mxdisplay --multi sim.out args ...
```

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

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

Another way to see details while maintaining an overview of the instrument is to use the `--zoom=`*factor* option. This magnifies the display of each component along the selected axes only, *e.g.* a Soller collimator is magnified perpendicular to the photon beam but not along it. This option may produce rather strange visual effects as the photon passes between components with different coordinate magnifications, but it is occasionally useful.

When debugging, it is often the case that one is interested only in photons 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 photons that are absorbed in the monochromator shielding. For these cases, the `--inspect=`*comp* option is useful. With this option, only photons that reach the component named *comp* are shown in the graphics display.

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

McXtrace Matlab back-end A 3D view of the instrument, and various operations (zoom, export, print, trace photons, ...) is available from a dedicated Graphical User

Interface. The `--inspect` option may be used (see previous paragraph), as well as the `--first` and `--last` options to specify a region of interest. The `mxdisplay` front-end requires Perl and Matlab to be installed.

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

5.3.4. Plotting the results of a simulation (mxplot)

The front-end `mxplot` (mxplot.pl on Windows) is a program that produces plots of all the monitors in a simulation, and it is thus useful to get a quick overview of the simulation results.

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

```
mxplot
```

This will plot any simulation data stored in the current directory, which is where simulations store their results by default. If the `--dir` or `--file` options have been used (see section 5.2), the name of the file or directory should be passed to mxplot, *e.g.* "`mxplot` *`dir`*" or "`mxplot`*`file`*". It is also possible to plot one single text (not binary) data file from a given monitor, passing its name to mxplot.

The drawing back-end program may be selected among PGPLOT, VRML, and Matlab using either the -p*PLOTTER* option (e.g. `mxplot -pmatlab file`) or using the current `MCXTRACE_FORMAT` environment variable. Moreover, the drawing back-end program will also be set depending on the *file* extension (see Table 5.3).

It should be emphasized that `mxplot` may *only* display simulation results with the format that was chosen during the computation. Indeed, if you request data in a given format from a simulation, you will only be able to display results using that same drawing back-end.

The `mxplot` front-end can also be run from the `mxgui` front-end.

The initial display shows plots for each detector in the simulation. Examples of plotter appearence for `mxplot` is shown in figures 5.3-5.

McXtrace/PGPLOT back-end Clicking the left mouse button on a plot produces a full-window version of that plot. The 'P' key saves a postscript file containing the current plot that can be sent to the printer to obtain a hardcopy; the 'C' key produces color postscript. The 'Q' key quits the program (or CTRL-C in the controlling terminal may be used as normal).

To use the `mxplot` front-end with PGPLOT, Perl, PGPLOT, and PDL must all be properly installed on the system. It may be necessary to set the `PGPLOT_DIR` and

PGPLOT_DEV environment variable; consult the documentation for PGPLOT on the local system in case of difficulty.

5.3.5. Creating and viewing the library, component/instrument help and Manuals (mxdoc)

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

```
mxdoc
mxdoc comp—instr
mxdoc --tools
```

The first example generates an *index.html* catalog file using the available components and instruments (both locally, and in the McXtrace library). The library catalog of components is opened using the BROWSER environment variable (e.g. netscape, konqueror, nautilus, MSIE, mozilla, ...). If the BROWSER is not defined, the help is displayed as text in the current terminal. This latter output may be forced with the -t or --text option.

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

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

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

See section 6.7 for more details about the McDoc usage and header format. To use the mxdoc front-end, the program Perl should be available.

5.3.6. Translating and merging McXtrace result files (all text formats)

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

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

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

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

The --scan option may be used to reconstruct the scan data from a set of directories which vary by instrument parameters. For instance, you ran a scan, but finally realized you should have prolonged it. Then simply simulate the missing bits, and apply mcformat -d scan_data --format=PGPLOT --scan step0 .. stepN. The resulting scan data is compatible with mcplot only when generating PGPLOT/McStas format.

You may conjugate this option with the `--merge` in order to add/merge similar data sets before re-building the scan.

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

5.4. Data formats - Analyzing and visualizing the simulation results

To analyze simulation results, one uses the same tools as for analyzing experimental data, *i.e.* programs such as IDL, Matlab and Scilab. The output files from simulations are usually simple text files containing headers and data blocks. Each data file contains informations about the simulation, the instrument, the parameters used, and of course the signal, the estimated error on the signal, and the number of events used in each bin. Additionally, all data files indicate their first (mean value) and second moment (standard deviation) in the 'statistics' field.

In order for the user to choose the data format, we recommend to set it using the `--format=FORMAT` or alternatively *via* the MCXTRACE_FORMAT environment variable, which will also make the front-end programs able to import and plot data and instrument consistently (see Section 5.2). The available format list is shown in Table 5.3.

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

5.4.1. McXtrace and PGPLOT format

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

One-dimensional histogram monitors (energy, time-of-flight, energy) write one line for each histogram bin. Each line contains a number identifying the bin (e.g. the energy) followed by three numbers: the simulated intensity, an estimate of the statistical error as explained in section 4.2.1, and the number of photon events for this bin.

Two-dimensional histogram monitors (position sensitive detectors) output M lines of N numbers representing x-ray intensities, where M and N are the number of bins in the two dimensions. The two-dimentional monitors also store the error estimates and event counts as additional matrices.

Single-point monitors output the photon intensity, the estimated error, and the photon event count as numbers on the terminal. (The results from a series of simulations may be combined in a data file using the `mxrun` front-end as explained in section 5.3.2).

When using one- and two-dimensional monitors, the integrated intensities are written to terminal as for the single-point monitor type, supplementing file output of the full one-

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

In addition to the files written for each one- and two-dimensional monitor component, another file (by default named `mccode.sim`) is also created. This file is in a special McXtrace ASCII format. It contains all available information about the instrument definition used for the simulation, the parameters and options used to run the simulation, and the monitor components present in the instrument. It is read by the `mxplot` front-end (see section 5.3.4). This file stores the results from single monitors, but by default contains only pointers (in the form of file names) to data for one- and two-dimensional monitors. By storing data in separate files, reading the data with programs that do not know the special McXtrace file format is simplified. The `--file` option may be used to store all data inside the `mccode.sim` file instead of in separate files.

5.4.2. HTML/VRML and XML formats

Both HTML and XML formats are available. The former may be viewed using any web browser (Netscape/Mozilla/Firefox, Internet Explorer, Nautilus, Konqueror) - showing data sets as VRML objects. XML data sets may be browsed for instance using Internet Explorer (Windows and Mac OS) or Firefox, GXMLViewer and KXMLEditor (under Linux).

The XML format is NeXus-like, but not fully compatible. However, McXtrace may generate genuine NeXus/HDF format (see bellow).

5.4.3. NeXus format

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

1. the HDF and NeXus libraries must have been installed

2. the McXtrace installation should be done with NeXus bindings, e.g. on Unix/Linux systems `configure --with-nexus; make; make install`.

3. the compilation of instruments must be done with the `-DUSE_NEXUS -lNeXus` flags (see Section 6.3.4). This is automated with the `mxrun` tool (Section 5.3.2).

All results are saved in a single file, containing 'groups' of data. To view such files, install and use HDFView (or alternatively HDFExplorer). This Java viewer can show content of all detectors, including metadata (attributes). Basic detector images may also be generated.

5.5. Using computer Grids and Clusters

Parallelizing a computation is in general advantageous when dependencies between parts of computations are not too strong. The situation of McXtrace is ideal since each photon ray can be simulated without interfering with other simulated photon rays. Therefore each photon ray can be simulated independently on a set of computers.

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

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

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

- when using a set of nodes (grid, cluster or multi-cores), it is possible to distribute simulations on a list of computers and multi-core machines (see section 5.5.1). Results are automatically merged after completion. This method is very efficient, and only requires a SSH server to be installed/configured on slave machines. In order to use an heterogeneous system, a C compiler should be optionally installed on slave machines.
- when using an homogeneous computer cluster, each simulation (including scan steps) may be computed in parallel using MPI. We recommend this method on clusters (see section 5.5.2).

Last but not least, you may run simulations manually on a number of machines. Once the distributed simulation have been completed, you may merge their results using `mxformat` (see section 5.3.6) in order to obtain a set of files just as if it had been executed on a single machine.

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

5.5.1. Distribute mxrun simulations on grids, multi-cores and clusters (SSH grid)

This method distributes simulations on a set of machines using *ssh* connections, using a command such as `mxrun --grid=4 ...`. Each of the scan steps is split and executed on distant slave machines, sending the executable with *scp*, executing single simulations, and then retrieving individual results on the master machine. These are then merged using `mxformat`.

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

- `--grid=<number>`: tells `mxrun` to use the grid over `<number>` nodes.

- `--machines=<file>`: defines a text file where the nodes which are to be used for parallel computation are listed; by default, `mxrun` will look at `$HOME/.mcxtrace-hosts` and `MCXTRACE/tools/perl/mcxtrace-hosts`. When used on a single SMP machine (multi-core/cpu), this option may be omitted.

- `--force-compile`: this option is required on heterogeneous systems. The C code is sent to all slaves and simulation is compiled on each node before starting computation. The default is to send directly the executatble from the master node, which only works on homogeneous systems. computation.

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

Requirements and limitations (SSH grids)

1. A master machine with an SSH client, and McXtrace installation.

2. A set of machines (homogeneous or heterogeneous) with SSH servers.

3. On heterogeneous grids, a C compiler must also be installed on all slave nodes.

4. `ssh` access from the master node (where McXtrace is installed) to the slaves through e.g. DSA keys *without* a password. These keys should be generated using the command `ssh-keygen`. Run *e.g.* `ssh-keygen -t dsa` on master node, enter no passphrase and add resulting `.ssh/id_dsa.pub` to `.ssh/authorized_keys` on all the slave nodes. The key generation and registering mechanism may be done automatically for the local machine from the *Help menu/Install DSA key* item of `mxgui`.

5. The machine names listed in the file `.mcxtrace-hosts` in your home directory or in the `MCXTRACE/tools/perl/mcxtrace-hosts` on the master node, one node per line. The `--machines=<file>` option enables to specify the hosts file to use. If it does not exist, only the current machine will be used (for multi-processor/core machines).

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

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

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

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

5.5.2. Parallel computing (MPI)

The MPI support requires that an implementaiton of the MPI-libraries is intalled on the system. We have generally find OpenMPI to be the most user-friendly, but other implementations (such as MPICH and LAM-MPI) work equally well. We have found no significant run-time differences. Usually MPI also requires properly setup `ssh` connections and keys as indicated in the ssh grid system (Section 5.5.1).

There are 3 methods for using MPI parallellization:

- Basic usage requires to compile and run the simulation by hand. See section 5.5.2 for details.
- A much simpler way is to use `mxrun -c --mpi=NB_CPU ...`. (See section 5.5.2)
- The `mxgui` interface supports MPI from within the Run Dialog.

The MPI support is especially suited for clusters. As an alternative, the SSH grid presented above (section 5.5.1) is very flexible and requires less configuration.

Requirements and limitations (MPI)

To use MPI you will need

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

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

3. `ssh` access from the master node (where McXtrace is installed) to the slaves through e.g. DSA keys *without* a password. These keys should be generated using the command `ssh-keygen`. Run *e.g.* `ssh-keygen -t dsa` on master node, enter no passphrase and add resulting `.ssh/id_dsa.pub` to `.ssh/authorized_keys` on all the slave nodes. The key generation and registering mechanism may be done automatically for the local machine from the *Help menu/Install DSA key* item of `mxgui`.

4. The machine names listed in the file `.mcxtrace-hosts` in your home directory or in the `MCXTRACE/tools/perl/mcxtrace-hosts` on the master node, one node per line. The `--machines=<file>` option enables to specify the hosts file to use. If it does not exist, only the current machine will be used (for multi-processor/core machines).

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

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

7. MPI must be correctly configured: if using `ssh`, you have to set ssh keys to avoid use of passwords; if using `rsh`, you have to set a `.rhosts` file. On non-local accounts, this procedure may fail and ssh always require passwords.

MPI Basic usage

To enable parallel computation, compile McXtrace output C-source file with `mpicc` with the flag `-DUSE_MPI` and run it using the wrapper of your MPI implementation (e.g. `mpirun`): Below is a shell script to this effect:

```
1  #!/bin/sh
2  # generate a C-source file [sim.c]
3  mcxtrace sim.instr
4  # generate an executable with MPI support [sim.mpi]
5  mpicc -DUSE_MPI -o sim.mpi sim.c
6
7  # execute with parallel processing over <N> computers
8  # here you have to list the computers you want to use
9  # in a file [machines.list] (using mpich implementation)
10 # (refer to MPI documentation for a complete description)
11 mpirun -machinefile machines.list -n <N> \
12        ./sim.mpi <instrument parameters>
```

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

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

mxrun script with MPI support

The `mxrun` script has been adapted to use MPI. Two new options have been added:

- `--mpi=<number>`: tells `mxrun` to use MPI, and to spread the simulation over `<number>` nodes

- `--machines=<file>`: defines a text file where the nodes which are to be used for parallel computation are listed; by default, `mxrun` will look at `$HOME/.mcxtrace-hosts` and `MCXTRACE/tools/perl/mcxtrace-hosts`. When used on a single SMP machine (multi-core/cpu), this option may be omitted.

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

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

```
node1
node2
node3
node4
```

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

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

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

5.5.3. McXtrace/MPI Performance

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

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

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

Figure 5.8.: McXtrace MPI execution time as a function of computing nodes, with *templateTOF* instrument and 1e8 initial photon events. Tests performed on Lonestar@TACC (US Teragrid, 2008).

5.5.4. MPI and Grid Bugs and limitations

- Some header of output files might contain minor errors.
- The computation split does not take into account the speed or the load of nodes: the overall time of a distributed computation is forced by the slowest node; for optimal performance, the "cluster" should be homogeneous.
- Interacting with a running simulation (USR1 and USR2 signals) is disabled with MPI.

6. The McXtrace kernel and meta-language

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

For maximum flexibility, efficiency and portability, the meta-language is based on C. Instrument geometry, propagation of x-rays between the different components, parameters, data input/output etc. is handled in the meta-language and by the McXtrace compiler. Complex calculations are written in C embedded in the meta-language description of the components. However, it is possible to set up an instrument from existing components and run a simulation without writing a single line of C code, working entirely in the meta-language.

Apart from the meta-language, McXtrace also includes a number of C library functions and definitions that are useful for x-ray tracing simulations. The definitions available for component developers are listed in appendix B. The list includes functions for

- Computing the intersection between a photon flight-path and various objects (such as planes, cylinders, boxes and spheres).
- Functions for generating random numbers with various distributions.
- Functions for reading or writing informations from/to data files.
- Convenient conversion factors between relevant units, etc.

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

6.1. Notational conventions

Simulations generated by McXtrace use a semi-classical description of the x-rays to compute the x-ray trajectory through the instrument and its interaction with the different components.

An instrument consists of a list of components through which the x-ray ray passes one after the other. The order of components is thus significant since McXtrace does not

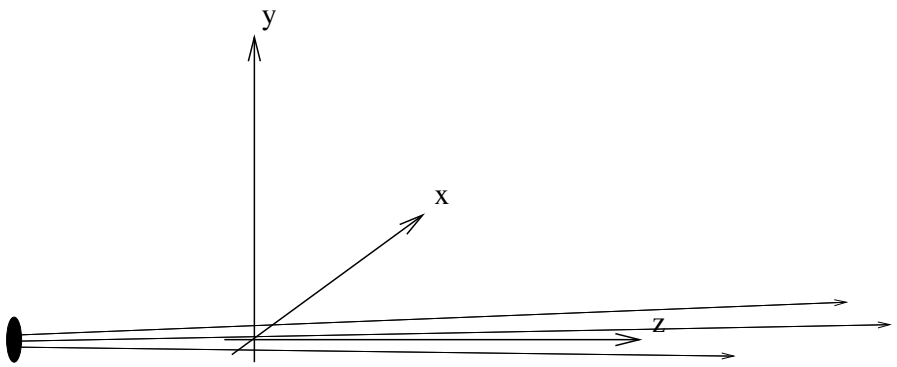

Figure 6.1.: conventions for the orientations of the axis in simulations.

automatically check which component is the next to interact with the x-ray at a given point in the simulation. Note that in case of a negative propagation length from one component to the next, the x-ray is by default *absorbed* as this is often an indication of unphysical conditions. If a large part of the simulated rays are *absorbed* on account of this a warning is issued, as this is often caused by a misplaced component.

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 fig. 6.1). Nothing in the McXtrace metalanguage enforces this convention, but if every component used different conventions the user would be faced with a severe headache! It is therefore necessary that this convention is followed by users implementing new components.

In the instrument definitions, units of length (*e.g.* component positions) are given in meters and units of angles (*e.g.* rotations) are given in degrees. The state of the x-ray is given by its position (x, y, z) in m, its wavevector (k_x, k_y, k_z) in Å^{-1}, the time in s,, the phase ϕ in rad, and a polarisation vector (E_x, E_y, E_z), and finally the x-ray weight p in photons s as described in chapter 4.

6.2. Syntactical conventions

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

Keywords are not case-sensitive, for example "`DEFINE`", "`define`", and "`dEfInE`" are all equivalent. However, by convention we always write keywords in uppercase to distinguish them from identifiers and C language keywords. In contrast, McXtrace identifiers (names), like C identifiers and keywords, *are* case sensitive, another good reason to use a consistent case convention for keywords. All McXtrace keywords are reserved, and thus

should not be used as C variable names. The list of these reserved keywords is shown in table 6.1.

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, McXtrace will search for a file containing the component definition in the standard component library (as well as in the current directory and any user-specified search directories, see section 5.1.2). 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 McXtrace meta-language. Files referenced with "`%include`" are read when the instrument is translated into C by the McXtrace compiler, and must contain valid McXtrace meta-language input (and possibly C code). Files referenced with "`#include`" are read when the C compiler generates an executable from the generated C code, and must contain valid C.

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

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

The "`%{`" and "`%}`" must appear on a line by themselves (do not add comments after). Additionally, if a "`%include`" statement is found *within* an embedded C code block, the specified file will be included from the 'share' directory of the standard component library (or from the current directory and any user-specified search directories) as a C library, just like the usual "`#include`" *but only once*. For instance, if many components require to read data from a file, they may all ask for "`%include "read_table-lib"`" without duplicating the code of this library. If the file has no extension, both `.h` and `.c` files will be searched and included, otherwise, only the specified file will be imported. The McXtrace'run-time' shared library is included by default (equivalent to "`%include "mcxtrace-r"`" in the `DECLARE` section). For an example of `%include`, see the optics/Lens_simple.comp component. See also section 6.4.2 for insertion of full instruments in instruments (instrument catenation).

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

Keyword	Scope	Meaning
ABSOLUTE	I	Indicates that the AT and ROTATED keywords are in the absolute coordinate system.
AT	I	Indicates the position of a component in an instrument definition.
COPY	I,C	copy/duplicate an instance or a component definition.
DECLARE	I,C	Declares C internal variables.
DEFINE	I,C	Starts an INSTRUMENT or COMPONENT definition.
DEFINITION	C	Defines component parameters that are constants (#define).
END	I,C	Ends the instrument or component definition.
SPLIT	I	Enhance incoming statistics by event repetition.
EXTEND	I	Extends a component TRACE section (plug-in).
FINALLY	I,C	Embeds C code to execute when simulation ends.
GROUP	I	Defines an exclusive group of components.
%include	I,C	Imports an instrument part, a component or a piece of C code (when within embedded C).
JUMP	I	Iterative (loops) and conditional jumps.
INITIALIZE	I,C	Embeds C code to be executed when starting.
ITERATE	I	Defines iteration counter for JUMP.
MCDISPLAY	C	Embeds C code to display component geometry.
NEXUS	I	Defines NeXus output type (4,5,XML,compression).
OUTPUT	C	Defines internal variables to be public and protected symbols (usually all global variables and functions of DECLARE).
PARAMETERS	C	Defines a class of component parameter (DEFINITION, SETTING,STATE).
PREVIOUS	C	Refers to a previous component position/orientation.
RELATIVE	I	Indicates that the AT and ROTATED keywords are relative to an other component.
ROTATED	I	Indicates the orientation of a component in an instrument definition.
SAVE	I,C	Embedded C code to execute when saving data.
SETTING	C	Defines component parameters that are variables.
SHARE	C	Declares global functions and variables to be shared.
STATE	C	Defines x-ray state coordinates.
TRACE	I,C	Defines the instrument as a the component sequence.
WHEN	I	Condition for component activation and JUMP.

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

6.3. Writing instrument definitions

The purpose of the instrument definition file is to specify a sequence of components, along with their position and parameters, which together make up a beamline. 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. Examples of instrument definitions can be found in the `example` directory of your McXtrace installation. Further examples may be found as they are built on the McXtrace web-page [Mcx].

As a summary, the usual grammar for instrument descriptions is

```
DEFINE INSTRUMENT name(parameters)
DECLARE C_code
INITIALIZE C_code {NEXUS}
TRACE components
{FINALLY C_code}
END
```

6.3.1. The instrument definition head

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

This marks the beginning of the definition. It also gives the name of the instrument and the list of instrument parameters. Instrument parameters describe the configuration of the instrument, and usually correspond to setting parameters of the components, see section 6.5. A motor position is a typical example of an instrument parameter. The input parameters of the instrument constitute the input that the user (or possibly a front-end program) must supply when the generated simulation is started.

By default, the parameters will be floating point numbers, and will have the C type `double` (double precision floating point). The type of each parameter may optionally be declared to be `int` for the C integer type or `char *` for the C string type. The name `string` may be used as a synonym for `char *`, and floating point parameters may be explicitly declared using the name `double`. The following example illustrates all possibilities:

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

Here `d1` and `d2` will be floating point parameters of C type `double`, `i` will be an integer parameter of C type `int`, and `s1` and `s2` will be string parameters of C type `char *`. The parameters of an instrument may be given default values. Parameters with default values are called *optional parameters*, and need not be given an explicit value when the instrument simulation is executed. When executed without any parameter value in the command line (see section 5.2), the instrument asks for all parameter values, but pressing the `Return` key selects the default value (if any). When used with at least one parameter value in the command line, all non specified parameters will have their value

set to the default one (if any). A parameter is given a default value using the syntax "*param= value*". For example

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

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

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

6.3.2. The `DECLARE` section

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

This gives C declarations that may be referred to in the rest of the instrument definition. A typical use is to declare global variables or small functions that are used elsewhere in the instrument. The `%include ''file''` keyword may be used to import a specific component definition or a part of an instrument. Variables defined here are global, and may conflict with internal McXtrace variables, particularly symbols like `x,y,z,Ex,Ey,Ez,kx,ky,kz,t,phi` and generally all names starting with `mc` nad `mx` should be avoided. If you can not compile the instrument, this may be the reason. The `DECLARE` section is optional.

6.3.3. The `INITIALIZE` section

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

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

6.3.4. The `NEXUS` extension

To use the NeXus format [Nex] the simulation must be linked with additional libraries (HDF and NeXus) which must have been pre-installed. Preferrably, McXtrace should have been installed with the `./configure --with-nexus` on Unix/Linux systems. To activate the NeXus output, the instrument must be compiled with the flags `-DUSE_NEXUS -lNeXus`.

The default NeXus format is NeXus 5 with compression. However, that format may be changed with the optional keyword `NEXUS` to follow the INITIALIZE section, namely:

```
INITIALIZE
%{
        ... C initializations. ...
%} NEXUS {"4"|"5"|"XML"|"compress"|"zip"}
```

It is possible to set the type of NeXus file with a string argument, containing words "4", "5" or "XML". Optionally, if the string also contains the `compress` or `zip` word, the NeXus file will use compression for Data Sets. We recommend the syntax `NEXUS "5 compress"` which is the default.

You may choose the name of the output file with the `-f filename` option from the instrument executable or `mxrun` (see Sections 5.2, 5.3.2 and Table 5.2).

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

6.3.5. The `TRACE` section

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

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

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

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

This declares a component named *name* that is an instance of the component definition named *comp*. The parameter list gives the setting and definition parameters for the component. The expressions $e_1, e_2, \ldots$ define the values of the parameters. For setting parameters arbitrary ANSI-C expressions may be used, while for definition parameters only *constant* numbers, strings, names of instrument parameters, or names of C identifiers are allowed (see section 6.5.1 for details of the difference between definition and setting parameters). To assign the value of a general expression to a definition parameter, it is necessary to declare a variable in the `DECLARE` section, assign the value to the variable in the `INITIALIZE` section, and use the variable as the value for the parameter.

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

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

The McXtrace compiler will automatically search for a file containing a definition of the component if it has not been declared previously. The definition is searched for in a file called "*name*`.comp`". See section 5.1.2 for details on which directories are searched. This facility is often used to refer to existing component definitions in standard component libraries. It is also possible to write component definitions in the main file before the instrument definitions, or to explicitly read definitions from other files using `%include` (not within embedded C blocks).

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

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

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

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

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

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

This will result in a coordinate system that is rotated first the angle ϕ_x (in degrees) around the x axis, then ϕ_y around the y axis, and finally ϕ_z around the z axis. Rotation may also be specified using `ABSOLUTE` rather than `RELATIVE`. If no rotation is specified, the default is $(0, 0, 0)$ using the same relative or absolute specification used in the `AT` modifier. We recommend to apply all rotations of an instrument description on Arm class components only, acting as goniometers, and position the optics on top of these. This usually makes it much easier to orient pieces of the beamline, and avoid positioning errors.

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

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

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

Invalid `PREVIOUS` references will be assumed to be absolute placement.

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

For experienced users, we recommend as well the usage of the `WHEN` and `EXTEND` keywords, as well as other syntax extensions presented in section 6.4.2 below.

6.3.6. The `SAVE` section

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

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

6.3.7. The `FINALLY` section

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

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

6.3.8. The end of the instrument definition

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

```
END
```

6.4. Writing instrument definitions - complex arrangements and syntax

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

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

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

6.4.1. Embedding instruments in instruments TRACE

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

```
DEFINE catenated()
TRACE

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

END
```

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

```
INSTRUMENT COMPONENT a=...
```

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

6.4.2. Component extensions - EXTEND

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

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

The embedded C code is appended to the component `TRACE` section, and all its internal variables (as well as all the `DECLARE` instrument variables, *except* instrument parameters) may be used. To use instrument parameters, you should copy them into global variables in the DECLARE instrument section, and refer to these latter. This component declaration modifier is of course optional. You will find usage examples in the Component manual[BK+12].

6.4.3. Mutually exclusive components in parallell - GROUP

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

In order to handle such arrangements without removing x-rays, groups are defined by the `GROUP` modifier (after the AT-ROTATED positioning):

`GROUP` *name*

to all involved component declarations. All components of the same named group are tested one after the other, until one of them interacts (uses the SCATTER macro). The selected component acts on the x-ray, and the rest of the group is skipped. Such groups are thus exclusive (only one of the elements is active).

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

```
COMPONENT name0 = comp(p1 = e1, p2 = e2, ...)
  AT (0, 0, 0) ABSOLUTE
COMPONENT name1 = comp(...) AT (...) ROTATED (...)
  GROUP GroupName EXTEND
  %{
      if (SCATTERED) printf("I scatter");
      else printf("I do not scatter");
  %}
COMPONENT name2 = comp(...) AT (...) ROTATED (...)
  GROUP GroupName
```

Components *name1* and *name2* are at the same position. If the first one intercepts the x-ray (and has a SCATTER within its `TRACE` section), the SCATTERED variable becomes true, the code extension will result in printing "I scatter", and the second component

will be skipped. Thus, we recommend to make use of the SCATTER keyword each time a component 'uses' the x-ray (scatters, detects, ...) within component definitions (see section 6.5). Also, the components to be grouped should be consecutive in the TRACE section of the instrument, and the GROUPed section should not contain components which are not part of the group.

Note that a `GROUP` construct is *not* applicable to a situation where an x-ray which has interacted (`SCATTER`ed) with one component in the group, may interact with another. In this case a more complex arrangement is needed. See [Wil+11] for a description of how to do this in McStas. This is approach is *directly* applicable in McXtrace without modification.

Combining EXTEND, GROUP and WHEN can result in unexpected behaviour. Please read the related warning at the end of section 6.4.5.

6.4.4. Duplication of component instances - COPY

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

The syntax for instance copy is

`COMPONENT` name = `COPY`(instance_name)

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

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

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

which will override the original instance parameter values. In case EXTEND, GROUP, JUMP and WHEN keywords are defined for the copied instance, these will override the settings from the copied instance.

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

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

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

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

As an example, we show how to have a set of three equivalent pinholes (Slits). Only the first instance of the Slit component is defined explicitly, whereas following instances are copies of that definition. The instance names of `Slit` components are set automatically.

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

COMPONENT S_1  = Slit(radius=0.1)
  AT (0,0,0) RELATIVE PREVIOUS

COMPONENT COPY(S_1)  = COPY(S_1)
  AT (0,0,d) RELATIVE PREVIOUS

COMPONENT COPY(S_1)  = COPY(S_1)
  AT (0,0,d) RELATIVE PREVIOUS
...
COMPONENT S_Out = Arm() AT (0,0,d) RELATIVE PREVIOUS
```

6.4.5. Conditional components - WHEN

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

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

The condition has the same scope as the EXTEND modifier, i.e. may use component internal variables as well as all the `DECLARE` instrument variables.

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

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

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

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

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

A usage example of the WHEN keyword can be found in the
`X-ray site/ESRF/ESRF_ID11` instrument from the `mxgui`, where the source model may be chosen by an external parameter.

The `WHEN` keyword is compatible with `GROUP`, and thus may be used to activate/deactivate members in a `GROUP` just like non-grouped components. Combining `WHEN`, `EXTEND` and `GROUP` can result in unexpected behaviour, please use them with caution! As an example, let a `GROUP` of components all have the same `WHEN` condition. If the condition is false, none of the elements `SCATTER`, meaning that all x-rays will be `ABSORB`ed. As a solution to this problem, we propose to include an `EXTEND`ed Arm component in the `GROUP`, but with the opposite `WHEN` condition and a `SCATTER` keyword in the `EXTEND` section. This means that when none of the other `GROUP` elements are present, the Arm will be present and `SCATTER`.

6.4.6. Component loops and non sequential propagation - JUMP

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

The jump may depend on a condition:

> `COMPONENT` *name* = *comp*($p_1 = e_1, p_2 = e_2, \ldots$) `AT` (...) `JUMP` *reference*
> WHEN *condition*

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

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

The jump arrives directly inside the target component, in the local coordinate system (i.e. without applying the AT and ROTATED keywords). In order to better control the target positions, it is *required* that, except for looping MYSELF, the target component type should be an *Arm*. Similarly to the `WHEN` modifier (see section 6.4.5), `JUMP` only applies within the TRACE section of the instrument definition. Other sections (INITIALIZE, SAVE, MCDISPLAY, FINALLY) are exectuted independently of the jump. As a side effect, the 3D view of the instrument `mxdisplay` will show components as if there was no jump. This means that in the following example, the very long mirror 3D view only shows a single mirror element.

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

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

6.4.7. Enhancing statistics reaching components - SPLIT

The following method applies when the incoming x-ray event distribution is considered to be representative of the real beam, but x-rays are lost in the course of propagation (with low efficiency processes, absorption, etc). Then, one may think that it's a pity to have so few events reaching the 'interesting' part of the beamline (usually close to the end of the instrument description). If some components make extensive use of random numbers (MC choices), they shuffle this way the distributions, so that identical incoming events will not produce the same outgoing event. In this case, you may want to use a technique known as *stratified sampling* (see chapter 4) which in McXtrace is implemented through the `SPLIT` keyword with the syntax

`SPLIT` r `COMPONENT` $name = comp(\ldots)$

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

6.5. Writing component definitions

The purpose of a McXtrace component is to model the interaction of an x-ray with a physical component of a real beamline. Given the state of the incoming x-ray, the component definition calculates the state of the x-ray when it leaves the component. The calculation of the effect of the component on the x-ray is performed by a block of embedded C code. One example of a component definition is given in section 6.5.10. All

other component definitions can be found on the McXtrace web-page [Mcx] and are also described in the McXtrace component manual.

A large number of functions and constants are available in order to write efficient components. See appendix B for

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

6.5.1. The component definition header

`DEFINE COMPONENT` *name*

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

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

This declares the definition and setting parameters of the component. These parameters can be accessed from all sections of the component (see below), as well as in `EXTEND` sections of the instrument definition (see section 6.3).

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

However, because of the use of `#define`, definition parameters suffer from the usual problems with C macro definitions. Also, it is not possible to use a general C expression for the value of a definition parameter in the instrument definition, only constants and variable names may be used. For this reason, setting parameters should be used whenever possible.

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

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

`#define`, the C compiler can treat definition parameters as constants when the simulation is compiled. For example, if the array sizes of a multidetector are definition parameters, the arrays can be statically allocated in the component `DECLARE` section. If setting parameters were used, it would be necessary to allocate the arrays dynamically using *e.g.* `malloc()`.

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

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

This declares a list of C identifiers (variables, functions) that are output parameters (*i.e.* global) for the component. Output parameters are used to hold values that are computed by the component itself, rather than being passed as input. This could for example be a count of x-rays in a detector or a constant that is precomputed to speed up computation.

Using `OUTPUT PARAMETERS` is *highly* recommended for `DECLARE` and internal/global component variables and functions in order to prevent that instances of the same component use the same variable names. Moreover (see section 6.5.2 below), these may be accessed from any other instrument part (e.g. using the `MC_GETPAR` C macro). On the other hand, the variables from the SHARE sections should *not* be defined as OUTPUT parameters.

The `OUTPUT PARAMETERS` section is optional.

Optional component parameters

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

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

Here `pack` is an optional parameter and if no value is given explicitly, "1" will be used. In contrast, if no value is given for `radius` or `height`, an error message will result.

Optional parameters can greatly increase the convenience for users of components with many parameters that have natural default values which are seldom changed. Optional parameters are also useful to preserve backwards compatibility with old instrument definitions when a component is updated. New parameters can be added with default values that correspond to the old behavior, and existing instrument definitions can be used with the new component without changes.

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

6.5.2. The DECLARE section

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

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

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

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

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

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

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

6.5.3. The SHARE section

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

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

6.5.4. The INITIALIZE section

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

This gives C code that will be executed once at the start of the simulation, usually to initialize any variables declared in the DECLARE section. This section is optional. Component setting parameters may be modified in this section, affecting the rest of the component.

6.5.5. The TRACE section

```
TRACE
%{
        ... C code to compute x-ray interaction with component ...
%}
```

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

The C code may also execute the special macro ABSORB to indicate that the x-ray has been absorbed in the component and the simulation of that x-ray will be aborted.

On the other hand, if the x-ray event should be *allowed* be backpropagated, the special macro `ALLOW_BACKPROP;` should preceed a call to the `PROP_**` call inside the component.

When the x-ray state is changed or detected, for instance if the component simulates a reflecting mirror, the special macro `SCATTER` should be called. This does not affect the results of the simulation in any way, but it allows the front-end programs to visualize the scattering events properly, and to handle component `GROUP`s in an instrument definition (see section 6.3.5). It basically increments the SCATTERED counter. The `SCATTER` macro should be called with the state parameters set to the proper values for the scattering event., so that x-ray events are displayed correctly. For an example of `SCATTER`, see the optics/Mirror_curved component. Lately a new keyword `RESTORE` has been added, which generally

6.5.6. The `SAVE` section

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

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

In order to work properly with the common output file format used in McXtrace, all monitor/detector components should use standard macros for writing data in the SAVE or FINALLY section, as explained below. In the following, we use $N = \sum_i p_i^0$ to denote the count of detected x-ray events, $p = \sum_i p_i$ to denote the sum of the weights of detected x-rays, and $p2 = \sum_i p_i^2$ to denote the sum of the squares of the weights, as explained in section 4.2.1.

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

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

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

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

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

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

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

```
DETECTOR_OUT_1D(t, xlabel, ylabel,xvar, xmin, xmax, m,
                &N[0], &p[0], &p2[0],filename)
```

Here,

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

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

```
DETECTOR_OUT_2D(t, xlabel, ylabel, xmin, xmax, ymin, ymax, m, n,
                &N[0][0], &p[0][0],&p2[0][0],filename)
```

Here,

- t is a string giving a descriptive title (*e.g.* "PSD monitor"),

- *xlabel* is a string giving a descriptive label for the X axis in a plot (*e.g.* "X position [cm]"),
- *ylabel* is a string giving a descriptive label for the Y axis of a plot (*e.g.* "Y position [cm]"),
- *xmin* is the lower limit for the X axis,
- *xmax* is the upper limit for the X axis,
- *ymin* is the lower limit for the Y axis,
- *ymax* 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.

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

```
DETECTOR_CUSTOM_HEADER(t)
```

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

6.5.7. The `FINALLY` section

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

This gives code that will be executed when the simulation has ended. This might be used to free memory and print out final results from components, *e.g.* the simulated intensity in a detector. This section also triggers the SAVE section to be executed.

6.5.8. The `MCDISPLAY` section

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

This gives C code that draws a sketch of the component in the plots produced by the `mxdisplay` front-end (see section 5.3.3). The section can contain arbitrary C code and may refer to the parameters of the component, but usually it will consist of a short sequence of the special commands described below that are available only in the MCDISPLAY section. When drawing components, all distances and positions are in meters and specified in the local coordinate system of the component.

The MCDISPLAY section is optional. If it is omitted, `mxdisplay` 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 `mxdisplay` is called with the `--zoom` option. For example:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

6.5.9. The end of the component definition

```
END
```

This marks the end of the component definition.

6.5.10. A component example: Semi-transparent mirror

Below is an example of a complete component. A simple example of the component `Semi_mirror` is given.

```
/*******************************************************************
*
* McStas, X-ray tracing package
*          Copyright (C) 2015, All rights reserved
*          DTU Physics,  Kgs. Lyngby, Denmark
*
* Component: Semi_miror
*
* %I
*
* Written by: Erik B Knudsen
* Date:
* Version: Revision: 1.0
* Release: McXtrace manual
* Origin: DTU Physics
*
* Simple flat semi-reflecting mirror with constant reflectivity
*
* %D
* A perfectly flat plane mirror example, intended as an example of a
* very simple component.
* It also illustrates the concept of MC-choice for governing statstics.
*
* %P
* Input parameters:
* xwidth: (m) Width of the mirror.
* yheight: (m) Height of the mirror.
* reflectivity: ( ) Constant scalar reflectivity of mirror.
* frac_reflect: ( ) Fraction of statistics for reflecting branch.
* %E
*******************************************************************/

DEFINE COMPONENT Semi_mirror
DEFINITION PARAMETERS ()
SETTING PARAMETERS (xwidth, yheight, reflectivity, frac_reflect)

SHARE
%{
%}

INITIALIZE
%{
%}

TRACE
%{
  PROP_Z0;
  if( x>-xwidth/2.0 && x<xwidth/2.0 && y>-yheight/2.0 && y<yheight/2.0){
    double r;
```

```
50      r=rand01();
51      SCATTER;
52      if(r<frac_reflect){
53        vz=-vz;
54        p*=reflectivity/frac_reflect;
55        internal_color=1;
56      }else{
57        internal_color=0;
58        p*=(1-reflectivity)/(1-frac_reflect);
59      }
60    }else{
61      RESTORE_XRAY(INDEX_CURRENT_COMP,x,y,z,kx,ky,kz,phi,t,Ex,Ey,Ez,p);
62    }
63 %}
64
65 MCDISPLAY
66 %{
67
68    multiline(5, -xwidth/2.0, -yheight/2.0, 0.0,
69                  xwidth/2.0, -yheight/2.0, 0.0,
70                  xwidth/2.0,  yheight/2.0, 0.0,
71                 -xwidth/2.0,  yheight/2.0, 0.0,
72                 -xwidth/2.0, -yheight/2.0, 0.0);
73 %}
74
75 END
```

Listing 6.1: Complete listing of a semi-transparent mirror component. The component implements a Monte Carlo choice which lets the user decide how much of the available statistics should be used for the reflected and tranmitted branches respectively.

6.6. Extending component definitions

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

6.6.1. Extending from the instrument definition file

If you only want to *add* something on top of the component existing behaviour, the simplest is to work from the instrument definition `TRACE` section, using the `EXTEND` modifier (see section 6.4.2). You do not need to write a new component definition, but only add a piece of code to execute.

6.6.2. Explicitly modify an existing library component

Copy the interesting component definition from the McXtrace library location (e.g. `/usr/local/mcxtrace/1.2/...` or `c:\mcxtrace\1.2\...`) into your working directory

next to your instrument definition file. This will cause McXtrace to pick this component definition up before the one in the library. Next, modify the local component file until you ae satisfied with it. If you feel that the newly modified componennt may be of use to the McXtrace community, please do not hesitate to contact the development team concerning the options for contributing. Rest assured that copyright remains with you.

6.6.3. Component heritage and duplication

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

The syntax for a full component child is

```
DEFINE COMPONENT child_name COPY parent_name
```

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

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

```
DEFINE COMPONENT child_name COPY parent_name


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

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

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

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

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

6.7. MxDoc, the McXtrace library documentation tool

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

Two forms of documentation can be generated. One is the component entry dialog in the `mxgui` front-end, see section 5.3.1. The other is a collection of web pages documenting the components and instruments, handled via the `mxdoc` front-end (see section 5.3.5), and the complete documentation for all available McXtrace components and instruments may be found at the McXtrace webpage [Mcx], as well as in the McXtrace library (see 7.1). All available McXtrace documentation is accessible from the `mxgui` 'Help' menu.

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

The format of the comments in the library source code

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

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

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

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

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

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

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

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

- Optionally, an %INSTRUMENT_SITE section followed by a single word is used to sort *instruments* by origin/location in the 'X-ray Site' menu in mxgui.

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

7. The component library: Abstract

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

7.1. A short overview of the McXtrace component library

The table in this section gives a quick overview of available McXtrace components provided with the distribution, in the `MCXTRACE` library. The location of this library is detailed in section 5.1.2. All of them are believed to be reliable, and some amount of systematic tests have been carried out. However, no absolute guarantee be given concerning their accuracy.

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

The `mxdoc` front-end (section 5.3.5) enables to display both the catalog of the McXtrace library, e.g using:

```
mxdoc
```

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

```
mxdoc --text name
mxdoc file.comp
```

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

McXtrace/sources	Description
Source_pt	Point source with uniform, Gaussian or general wavelength/energy distribution.
Source_flat	Flat surface source with uniform, Gaussian or general wavelength/energy distribution.
Source_div	Flat surface source with general wavelength/energy distribution and a given divergence.
Source_gaussian	Flat source with a Gaussian cross section and a specified divergence.
Source_lab	Full featured model of a laboratory X-ray source.

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

McXtrace/optics	Description
Arm	Arm/optical bench.
Beamstop	Rectangular/circular beam stop.
Filter	A general absorption filter which can be any material.
Lens_simple	Model of a thin compound refractive lens
Lens_parab	Detailed model of a parabolic compound refractive lens of any material
Mirror_curved	Cylindrically curved mirror with scalar reflectivity
Mirror_ellioptic	Elliptically curved mirror with scalar reflectivity
Mirror_parabolic	Parabolically curved mirror with scalar reflectivity
Multilayer_elliptic	General, full featured model of an elliptically curved multilayer mirror
Slit	Perfect slit, may be either rectangular or circular
Slit_N	Multichanneled slit, i.e. a grating

Table 7.2.: Optics components of the McXtrace library.

McXtrace/samples	Description
PowderN	General powder sample with N scattering vectors, using a data file. Can assume *concentric* shape, i.e. can be used to model sample enviroment.
Perfect_crystal	Model of a prefect crystal, for instance to be used as a monochromator
Saxs_spheres	Simple sample for Small Angle X-ray Scattering - hard spheres
Single_crystal	Mosaic single crystal with multiple scattering vectors using a data file.

Table 7.3.: Sample components of the McXtrace library.

McXtrace/monitors	Description
EPSD_monitor	A monitor measuring x-ray intensity vs. position, but restricted to a particular energy interval.
E_monitor	Energy-sensitive monitor.
L_monitor	Wavelength-sensitive monitor.
Monitor	Simple single detector/monitor.
Monitor_nD	General monitor that can output 0/1/2D signals (Intensity or signal vs. [something] and vs. [something] ...).
PSD_monitor	Position-sensitive monitor.
PSD_monitor_4PI	Spherical position-sensitive detector.
PreMonitor_nD	This component is a PreMonitor that is to be used with one Monitor_nD, in order to record some photon parameter correlations.
TOFLambda_monitor	Time-of-flight vs. wavelength monitor.
TOF_monitor	Rectangular Time-of-flight monitor.

Table 7.4.: Selected Monitor components of the McXtrace library.

McXtrace/misc	Description
Progress_bar	Displays status of a running simulation. May also trigger intermediate SAVE.
Beam_spy	A monitor that displays mean statistics (no output file).

Table 7.5.: Miscellaneous components of the McXtrace library.

McXtrace/data	Description
Al.txt, He.txt, etc.	Data files from the NIST-database (covering the elements with $Z \in [1..92]$) to be used for absorption and refraction.
Fe_bpy_ES_DFT.txt, Fe_bpy_GS_DFT.txt	Ground- and excited state atom positions of Iron tris bipyridine for use with `Molecule\_2state`.
FormFactors.txt	Atomic form factors for elements with $Z \in [1..92]$.
Ref_W_B4C.txt	Reflectivity of a W-B4C multilayer.
Ref_W_Si.txt	Reflectivity of a W-Si multilayer.

Table 7.6.: Data files of the McXtrace library.

McXtrace/examples	Description
JJ_SAXS.instr	A model of the SAXSlab laboratory SAXS-system which was a prototype of the one currently sold by SAXSlab.
Pump_probe	Generic pump-probe experiment instrument.
NSLS2_CHX.instr	Model of the CHX hard X-ray beamline at NSLS2.
MAXII_811.instr	Surface diffraction and EXAFS-bealine at MAXlab
MAXII_711.instr	Powder diffraction beamline at MAXlab.
ESRF_ID11.instr	The ESRF ID11 3DXRD beamline.
Be_BM_beamline.instr	Design study of a bending magnet beamline using refractive lenses and a slit as monochromator.
XFEL_SPB.instr	Model of the beamtransport system of the SPB beamline at XFEL.

Table 7.7.: Instrument example files of the McXtrace library. These example instruments are accessible through the `mxgui` "X-ray site" menu.

A. Random numbers in McXtrace

A.1. Transformation of random numbers

In order to perform the Monte Carlo choices, one needs to be able to pick a random number from a given distribution. However, most random number generators only give 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, $\Psi(y)$, of the transformed $y = f(x)$, assuming $f(x)$ is monotonous. All random numbers lying in the interval $[x; x+dx]$ are transformed to lie within the interval $[y; y + f'(x)dx]$, so the resulting distribution must be $\Psi(y) = \phi(x)/f'(x)$.

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

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

By indefinite integration we reach

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

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

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

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

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

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

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

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

Exponential decay In a simple 2-state model a molecule may be said to relax into its ground state at a time which is exponetially distributed after the initial excitation at $t = 0$. We thus want to pick a relaxation time from the normalised distribution $\Psi(t) = \exp(-t/\tau)/\tau$. Use of Eq. (A.2) gives $x = 1 - \exp(-t/\tau)$. For convenience we now use the random variable $x_1 = 1 - x$ (with the same distributions as x), giving the simple expression $t = -\tau \ln(x_1)$.

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

A.2. Random generators

Even though there is the possibility to use the system random generator, as well as the initial McXtrace random generator, the default algorithm is the so-called "Mersenne Twister", by Makoto Matsumoto and Takuji Nishimura[MN98]. See
`http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html` for original source.

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

B. Libraries and conversion constants

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

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

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

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

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

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

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

B.1. Run-time calls and functions (`mcxtrace-r`)

Here we list a number of preprogrammed macros and functions which may ease the task of writing component and instrument definitions. By convention macros are in upper case whereas functions are in lower case.

B.1.1. Photon propagation

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

- **ABSORB**. This macro issues an order to the overall McXtrace simulator to interrupt the simulation of the current x-ray history and to start a new one.
- **PROP_Z0**. Propagates the x-ray to the $z = 0$ plane, by adjusting (x, y, z), ϕ, and t accordingly from knowledge of the x-ray wavevector (kx, ky, kz). If the propagation length is negative, the x-ray is absorbed, except if a `ALLOW_BACKPROP;` preceeds it.

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

- **PROP_X0, PROP_Y0**. These macros are analogous to `PROP_Z0` except they propagate to the $x = 0$ and $y = 0$ planes respectively.

- **PROP_DL**(dl). Propagates the x-ray by the length dl, adjusting (x, y, z), ϕ, t accordingly, from knowledge of the x-ray wavevector.

- **ALLOW_BACKPROP**. Indicates that the *next* propagation routine will not remove the x-ray, even if negative propagation lengths are found. Subsequent propagations are not affected.

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

B.1.2. Coordinate and component variable retrieval

- **MC_GETPAR**(*comp*, *outpar*). This may be used in e.g. the FINALLY section of an instrument definition to reference the parameters of a component.

- **NAME_CURRENT_COMP** gives the name of the current component as a string.

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

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

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

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

- **INDEX_CURRENT_COMP** is the number (index) of the current component (starting from 1).
- **POS_A_COMP_INDEX**(*index*) is the absolute position of component *index*. POS_A_COMP_INDEX (INDEX_CURRENT_COMP) is the same as POS_A_CURRENT_COMP. You may use POS_A_COMP_INDEX (INDEX_CURRENT_COMP+1) to make, for instance, your component access the position of the next component (this is usefull for automatic targeting). A component of the vector is referred to as POS_A_COMP_INDEX(*index*).*i* where *i* is x, y or z.
- **POS_R_COMP_INDEX** works the same as above, but with relative coordinates.
- **STORE_XRAY**($index, x, y, z, kx, ky, kz, phi, t, Ex, Ey, Ez, p$) stores the current x-ray state in the trace-history table, in local coordinate system. *index* is usually INDEX_CURRENT_COMP. This is automatically done when entering each component of an instrument.
- **RESTORE_XRAY**($index, x, y, z, kx, ky, kz, phi, t, Ex, Ey, Ez, p$) restores the x-ray state to the one at the input of the component *index*. To ignore a component effect, use RESTORE_XRAY (INDEX_CURRENT_COMP, $x, y, z, kx, ky, kz, phi, Ex, Ey, Ez, p$) at the end of its TRACE section, or in its EXTEND section. These x-ray states are in the local component coordinate systems.
- **SCATTERED** is a variable set to 0 when entering a component, which is incremented each time a SCATTER event occurs. This may be used in the `EXTEND` sections to determine whether the component interacted with the current x-ray.
- **extend_list**(n, &*arr*, &*len*, *elemsize*). Given an array *arr* with *len* elements each of size *elemsize*, make sure that the array is big enough to hold at least n elements, by extending *arr* and *len* if necessary. Typically used when reading a list of numbers from a data file when the length of the file is not known in advance.
- **mcset_ncount**(n). Sets the number of x-ray histories to simulate to n.
- **mcget_ncount**(). Returns the number of x-ray histories to simulate (usually set by option `-n`).
- **mcget_run_num**(). Returns the number of x-ray histories that have been simulated until now.

B.1.3. Coordinate transformations

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

- **coords_add**(a, b), **coords_sub**(a, b), **coords_neg**(a) enable to operate on coordinates, and return the resulting Coord structure.
- **rot_set_rotation**(*Rotation t,* ϕ_x, ϕ_y, ϕ_z) Get transformation matrix for rotation first ϕ_x around x axis, then ϕ_y around y, and last ϕ_z around z. t should be a 'Rotation' ([3][3] 'double' matrix).
- **rot_mul***(Rotation t1, Rotation t2, Rotation t3)* performs $t3 = t1.t2$.
- **rot_copy***(Rotation dest, Rotation src)* performs $dest = src$ for Rotation arrays.
- **rot_transpose***(Rotation src, Rotation dest)* performs $dest = src^t$.
- **rot_apply***(Rotation t, Coords a)* returns a Coord structure which is $t.a$

B.1.4. Mathematical routines

- **NORM**(x, y, z). Normalizes the vector (x, y, z) to have length 1.
- **scalar_prod**$(a_x, a_y, a_z, b_x, b_y, b_z)$. Returns the scalar product of the two vectors (a_x, a_y, a_z) and (b_x, b_y, b_z).
- **vec_prod**(&a_x,&a_y,&a_z, b_x,b_y,b_z, c_x,c_y,c_z). Sets (a_x, a_y, a_z) equal to the vector product $(b_x, b_y, b_z) \times (c_x, c_y, c_z)$.
- **rotate**(&x,&y,&z,v_x,v_y,v_z,φ,a_x,a_y,a_z). Set (x, y, z) to the result of rotating the vector (v_x, v_y, v_z) the angle φ (in radians) around the vector (a_x, a_y, a_z).
- **normal_vec**(n_x, n_y, n_z, x, y, z). Computes a unit vector (n_x, n_y, n_z) normal to the vector (x, y, z).*
- **solve_2nd_order**(*t_0,*t_1, A, B, C). Solves the 2^{nd} order equation $At^2+Bt+C = 0$ and puts the solutions in *t_0 and *t_1. The smallest positive solution into pointer *t_0. If t_1=`NULL` it is ignored and the second solution is discarded.

B.1.5. Output from detectors

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

- **DETECTOR_OUT_0D**(...). Used to output the results from a single detector. The name of the detector is output together with the simulated intensity and estimated statistical error. The output is produced in a format that can be read by McXtrace front-end programs.
- **DETECTOR_OUT_1D**(...). Used to output the results from a one-dimensional detector. Integrated intensities error etc. is also reported as for DETECTOR_OUT_0D.
- **DETECTOR_OUT_2D**(.....). Used to output the results from a two-dimentional detector. Integrated intensities error etc. is also reported as for DETECTOR_OUT_0D.

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

B.1.6. Ray-geometry intersections

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

- **box_intersect**(&l_1, &l_2, x, y, z, k_x, k_y, k_z, d_x, d_y, d_z). Calculates the (0, 1, or 2) intersections between the x-ray path and a box of dimensions d_x, d_y, and d_z, centered at the origin for a x-ray with the parameters (x, y, z, k_x, k_y, k_z). The intersection lengths are returned in the variables l_1 and l_2, with $l_1 < l_2$. In the case of less than two intersections, t_1 (and possibly t_2) are set to zero. The function returns true if the x-ray intersects the box, false otherwise.

- **cylinder_intersect**(&l_1, &l_2, x, y, z, k_x, k_y, k_z, r, h). Similar to **box_intersect**, but using a cylinder of height h and radius r, centered at the origin.

- **sphere_intersect**(&l_1, &l_2, x, y, z, k_x, k_y, k_z, r). Similar to **box_intersect**, but using a sphere of radius r.

- **ellipsoid_intersect**(&l_1, &l_2, x, y, z, k_x, k_y, k_z, a,b,c,Q,). Similar to **box_intersect**, but using an ellipsoid with half-axis a,b,c oriented by the rotation matrix Q. If $Q = I$, a is along the x-axis, b along y and c along z

B.1.7. Random numbers

By default McXtrace uses the included Mersenne Twister[MN98] algorithm for generating pseudo random numbers.

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

- **randnorm**(). Returns a random number from a normal distribution centered around 0 and with $\sigma = 1$. The algorithm used to sample the normal distribution is explained in Ref. [Pre+86, ch.7].

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

- **randtriangle**(). Returns a random number from a triangular distribution between -1 and 1.

- **randvec_target_circle**(&v_x, &v_y, &v_z, &$d\Omega$, aim_x, aim_y, aim_z, r_f). Generates a random vector (v_x, v_y, v_z), of the same length as (aim_x, aim_y, aim_z), which is targeted at a *disk* centered at (aim_x, aim_y, aim_z) with radius r_f (in meters), and perpendicular to the *aim* vector.. All directions that intersect the circle are

chosen with equal probability. The solid angle of the circle as seen from the position of the x-ray is returned in $d\Omega$. This routine was previously called **randvec_target_sphere** (which still works).

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

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

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

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

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

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

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

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

- **Table_Read_Offset**(&*Table*, *filename*, *block*, &*offset*, n_{rows}) does the same as Table_Read except that it starts at offset *offset* (0 means begining of file) and reads n_{rows} lines (0 for all). The *offset* is returned as the final offset reached after reading the n_{rows} lines.
- **Table_Read_Offset_Binary**(&*Table*, *filename*, *type*, *block*, &*offset*, n_{rows}, $n_{columns}$) does the same as Table_Read_Offset, but also specifies the *type* of the file (may be "float" or "double"), the number n_{rows} of rows to read, each of them having $n_{columns}$ elements. No text header should be present in the file.
- **Table_Rebin**(&*Table*) rebins all *Table* rows with increasing, evenly spaced first column (index 0), e.g. before using Table_Value. Linear interpolation is performed for all other columns. The number of bins for the rebinned table is determined from the smallest first column step.
- **Table_Info**(*Table*) print information about the table *Table*.
- **Table_Index**(*Table*, m, n) reads the *Table*[m][n] element.
- **Table_Value**(*Table*, x, n) looks for the closest x value in the first column (index 0), and extracts in this row the n-th element (starting from 0). The first column is thus the 'x' axis for the data.
- **Table_Free**(&*Table*) free allocated memory blocks.
- **Table_Value2d**(*Table*, X, Y) Uses 2D linear interpolation on a Table, from (X,Y) coordinates and returns the corresponding value.

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

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

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

```
1 #Rh (Z  45)
2 #Atomic weight: A[r]  102.9055
3 #Nominal density: rho 1.2390E+01
4 #     sigma[a](barns/atom) = [mu/rho](cm\^2 g\^-1)   .   1.70879E+02
5 #     E(eV) [mu/rho](cm\^2 g\^-1) = f[2](e atom\^-1)   .   4.08922E+05
6 #     14 edges. Edge energies (keV):
```

```
7  #
8  #
9  #      K       2.32199E+01  L I     3.41190E+00  L II    3.14610E+00  L III
       3.00380E+00
10 #      M I     6.27100E-01  M II    5.21000E-01  M III   4.96200E-01  M IV
       3.11700E-01
11 #      M V     3.07000E-01  N I     8.10000E-02  N II    4.79000E-02  N III
       4.79000E-02
12 #      N IV    2.50000E-03  N V     2.50000E-03
13 #
14 #    Relativistic correction estimate f[rel] (H82,3/5CL) = -4.0814E-01,
15 #    -2.5440E-01 e atom\^-1
16 #    Nuclear Thomson correction f[NT] = -1.0795E-02 e atom\^-1
17 #
18 #
   ------------------------------------------------------------------------


19 #Form Factors, Attenuation and Scattering Cross-sections
20 #Z=45, E = 0.001 - 433 keV
21 #
22 #      E              f[1]           f[2]         [mu/rho]       [sigma/rho]
         [mu/rho]       [mu/rho][K]       lambda
23 #                                              Photoelectric Coh+inc       Total
24 #     keV          e atom\^-1       e atom\^-1   cm\^2 g\^-1        cm\^2 g\^-1
          cm\^2 g\^-1   cm\^2 g\^-1      nm
25 1.069000E-02  1.89417E+00  4.8055E+00  1.8382E+05  1.1514E-04  1.8382E+05
     0.000E+00   1.160E+02
26 1.142761E-02  2.09662E+00  5.1028E+00  1.8260E+05  1.5865E-04  1.8260E+05
     0.000E+00   1.085E+02
27 1.221612E-02  2.32705E+00  5.4019E+00  1.8082E+05  2.1741E-04  1.8082E+05
     0.000E+00   1.015E+02
28 1.305903E-02  2.58575E+00  5.6998E+00  1.7848E+05  2.9628E-04  1.7848E+05
     0.000E+00   9.494E+01
29 1.396010E-02  2.87263E+00  5.9931E+00  1.7555E+05  4.0158E-04  1.7555E+05
     0.000E+00   8.881E+01
30 1.492335E-02  3.18714E+00  6.2786E+00  1.7204E+05  5.4136E-04  1.7204E+05
     0.000E+00   8.308E+01
31 1.595306E-02  3.52819E+00  6.5531E+00  1.6797E+05  7.2588E-04  1.6797E+05
     0.000E+00   7.772E+01
32 1.705382E-02  3.89415E+00  6.8134E+00  1.6337E+05  9.6809E-04  1.6337E+05
     0.000E+00   7.270E+01
33   ...
```

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

The *filename* is first searched into the current directory (and all user additional locations specified using the `-I` option, see the 'Running McXtrace ' chapter in the User Manual), and if not found, in the `data` sub-directory of the `MCXTRACE` library location. This way, you do not need to have local copies of the McXtrace Library Data files (see table 7.6).

A usage example for this library part may be:

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

Table_Read(&Table, file, 1);  // initialize and read the first numerical
    block
Table_Info(Table);            // display table informations
...
x = Table_Index(Table, 2,5);  // read the 3rd row, 6th column element
                              // of the table. Indexes start at zero in C
                                 .
y = Table_Value(Table, 1.45,1);  // look for value 1.45 in 1st column (x
    axis)
                              // and extract 2nd column value of that row
Table_Free(&Table);           // free allocated memory for table
```

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

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

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

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

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

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

B.3. 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
FWHM2RMS	$1/\sqrt{8\log(2)}$	Full width half maximum	Root mean square (standard deviation)
RMS2FWHM	$\sqrt{8\log(2)}$	Root mean square (standard deviation)	Full width half maximum
MNEUTRON	$1.67492 \cdot 10^{-27}$ kg	Neutron mass, m_{n}	
HBAR	$1.05459 \cdot 10^{-34}$ Js	Planck constant, $\hbar$	
PI	3.14159265...	π	
CELE	1.602176487e-19	Elementary charge (C)	
M_C	299792458	Speed of light in vacuum (m/s)	
NA	6.02214179e23	Avogadro's number (#atoms/g·mole)	
RE	2.8179402894e-5	Thomson scattering length (AA)	
E2K	0.506773091264796	Wavenumber (1/AA)	Energy (keV)
K2E	1.97326972808327	Energy (keV)	Wavenumber (1/AA)

C. The McXtrace terminology

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

- **Arm** A generic McXtrace component which defines a frame of reference for other components.
- **Component** One unit (*e.g.* optical element) in an x-ray beamline. These are considered as Types of elements to be instantiated in an Instrument description.
- **Component Instance** A named Component (of a given Type) inserted in an Instrument description.
- **Definition parameter** An input parameter for a component. For example the radius of a sample component or the divergence of a collimator. Technically, a definition parameter is translated into a literal constant, which prevents it from being edited at runtime.
- **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 McXtrace components defining an x-ray beamline.
- **Kernel** The McXtrace language definition and the associated compiler
- **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 `mcxtrace-r.c` and `mcxtrace-r.h` included in the McXtrace distribution, that declare functions and variables used by the generated simulations.
- **Setting parameter** Similar to a definition parameter, but with the restriction that the type of the parameter must be declared unless it is a number. In technical terms, a setting parameter is translated into an actual variable (as opposed to a definition parameter) which may be dynamically updated.

Bibliography

[Mcx] See http://www.mcxtrace.org (cit. on pp. 6, 7, 11, 22, 37, 57, 68, 80, 82).

[Mcz] See http://trac.mccode.org/report (cit. on p. 6).

[Mcc] See http://trac.mccode.org (cit. on p. 12).

[Sha] See http://www.esrf.eu/computing/scientific/raytracing/ (cit. on p. 16).

[Str] See http://www.strawberryperl.com (cit. on p. 26).

[Nex] See http://www.neutron.anl.gov/nexus/ (cit. on pp. 46, 58).

[Bau+07] Sondes Trabelsi Bauer et al. "Simulation of X-ray beamlines with the new ray tracing tool¡ i¿ X¡/i¿ Trace". In: *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 582.1 (2007), pp. 90–92 (cit. on pp. 7, 16).

[BK+12] Erik Bergbäck Knudsen et al. *Component Manual for the X-ray-Tracing Package McXtrace, Version 1.0*. 2012 (cit. on p. 63).

[Cop+86] J. R. D. Copley et al. "Improved Monte Carlo calculation of multiple scattering effects in thermal neutron scattering experiments". In: *Comput. Phys. Commun.* 40 (1986), p. 337. ISSN: 0010-4655. DOI: http://dx.doi.org/10.1016/0010-4655(86)90118-9 (cit. on p. 8).

[GRR92] Grimmett, G. R., and Stirzakerand D. R. *Probability and Random Processes, 2nd Edition*. Clarendon Press, Oxford, 1992 (cit. on p. 17).

[Jam80] F. James. In: *Rep. Prog. Phys.* 43 (1980), p. 1145 (cit. on pp. 16, 17, 21).

[LN99] K. Lefmann and K. Nielsen. "McStas, a general software package for neutron ray-tracing simulations". In: *Neutron News* 10 (1999), pp. 20–23. DOI: 10.1080/10448639908233684 (cit. on p. 6).

[MN98] Makoto Matsumoto and Takuji Nishimura. "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator". In: *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 8.1 (1998), pp. 3–30 (cit. on pp. 87, 92).

[NM65] J.A. Nelder and R. Mead. In: *Computer Journal* 7 (1965), pp. 308–313 (cit. on p. 33).

[Pre+86] W. H. Press et al. *Numerical Recipes in C*. Cambridge University Press, 1986 (cit. on p. 92).

[Pre+02] W.H. Press et al. *Numerical Recipes (2nd Edition)*. Cambridge University Press, 2002 (cit. on p. 33).

[Rio+11] M Sanchez del Rio et al. “SHADOW3: a new version of the synchrotron X-ray optics modelling package”. In: *Journal of Synchrotron Radiation* 18.5 (2011), p. 0 (cit. on pp. 7, 16).

[Sch08] F Schäfers. “The BESSY raytrace program RAY”. In: *Modern Developments in X-Ray and Neutron Optics* (2008), pp. 9–41 (cit. on pp. 7, 16).

[WCC94] C Welnak, G J Chen, and F Cerrina. “SHADOW: A synchrotron radiation and X-ray optics simulation tool”. In: *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 347.1-3 (1994), pp. 344–347 (cit. on pp. 7, 16).

[Wil+11] Peter K Willendrup et al. “Using McStas for modelling complex optics, using simple building bricks”. In: *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 634.1 (2011), S150–S155 (cit. on p. 64).