Monte Carlo Particle Lists: MCPL

T Kittelmann[a,*], E Klinkby[b], E B Knudsen[c], P Willendrup[c,a], X X Cai[a,b], K Kanaki[a]

[a] *European Spallation Source ERIC, Sweden*
[b] *DTU Nutech, Technical University of Denmark, Denmark*
[c] *DTU Physics, Technical University of Denmark, Denmark*

Abstract

A binary format with lists of particle state information, for interchanging particles between various Monte Carlo simulation applications, is presented. Portable `C` code for file manipulation is made available to the scientific community, along with converters and plugins for several popular simulation packages.

PROGRAM SUMMARY

Manuscript Title: Monte Carlo Particle Lists: MCPL

Authors: T Kittelmann, E Klinkby, E B Knudsen, P Willendrup, X X Cai and K Kanaki

Program Title: MCPL

Journal Reference:

Catalogue identifier:

Licensing provisions: CC0 for core MCPL, see LICENSE file for details.

Programming language: `C` and `C++`

Operating system: Linux, OSX, Windows

Keywords:

`MCPL`, Monte Carlo, particles, data storage, simulation, interchange format, `C++`, `C`, `Geant4`, `MCNP`, `McStas`, `McXtrace`

Classification: 4, 9, 11, 17

External routines/libraries: `Geant4`, `MCNP`, `McStas`, `McXtrace`

Nature of problem:

*Corresponding author. *Email address:* `thomas.kittelmann@esss.se`

Persistification of particle states in Monte Carlo simulations, for interchange between simulation packages or for reuse within a single package.

Solution method:

Binary interchange format with associated code written in portable `C` along with tools and interfaces for relevant simulation packages.

1. Introduction

The usage of Monte Carlo simulations to study the transport and interaction of particles and radiation is a powerful and popular technique, finding use throughout a wide range of fields – including but not limited to both high energy and nuclear physics as well as space and medical sciences [1]. Naturally, a plethora of different frameworks and applications exists for carrying out these simulations (cf. section 3 for examples), with implementations in different languages and domains ranging from general purpose to highly specialised field- and application-specific.

A common principle used in the implementation of these applications is the representation of particles by a set of state parameters – usually including at least particle type, time coordinate, position and velocity or momentum vectors – and a suitable representation of the geometry of the problem (either via descriptions of actual surfaces and volumes in a virtual three-dimensional space, or through suitable parameterisations). In the simplest scenario where no variance-reduction techniques are employed, simulations are typically carried out by proceeding iteratively in steps from an initial set of particles states, with the state information being updated along the way as a result of the pseudo-random or deterministic modelling of processes affecting the particle. The modelling can represent particle self-interactions, interactions with the material of the simulated geometry, or simply its forward transport through the geometry, using either straight-forward ray-tracing techniques or more complicated trajectory calculations as appropriate. In addition to a simple update of state parameters, the modelling can result in termination of the simulation for the given particle

or in the creation of new secondary particle states, which will in turn undergo simulation themselves.

Occasionally, use-cases arise in which it would be beneficial to be able to capture a certain subset of particle states present in a given simulation, in order to continue their simulation at a later point in either the same or a different framework. Such capabilities have typically been implemented using custom application-specific means of data exchange, often involving the tedious writing of custom input and output hooks for the specific frameworks and use-cases in question. Here is instead presented a standard format for exchange of particle state data, *Monte Carlo Particle Lists* (`MCPL`), which is intended to replace the plethora of custom converters with a more convenient scenario in which experts of each framework implement converters to the common format, as a one-time effort. The idea being that users of the various frameworks then gain the ability to simply activate those pre-existing and validated converters in order to carry out their work.

The present work originated in the needs for simulations at neutron scattering facilities, where a multitude of simulation frameworks are typically used to describe the various components from neutron production to detection, but historically other conceptually similar formats have been and are used in high energy physics to communicate particle states between event generators and detector simulations [2, 3, 4]. Here, the formats were mainly developed for in-process communication and also had to accommodate the potential description of intermediary unphysical or bound particles, as well as potentially keeping meta-data concerning the simulated history of a given particle in the record. For these reasons, for reasons of portability and ease of integration, and due to the lack of a common well-defined on-disk format, these existing solutions were deemed unfit for the goals of the work presented here: a compact yet flexible on-disk binary format for particle state information, able to accommodate a wide range of use-cases with close to optimal storage requirements. The accompanying code with which to access and manipulate it should be small, efficient and easily integrated into existing codes and build systems. Consequently, it

was chosen to implement the format through a set of `C` functions declared in a single header file, `mcpl.h`, and implemented in a single file, `mcpl.c`. These two files will here be referred to as the *core* `MCPL` code, and are made freely available under the CC0 1.0 Universal Creative Commons license. Along with associated code examples, documentation, configuration files (cf. section 2.5) and application-specific interface code which is not embedded in the relevant upstream projects (cf. sections 3.1 and 3.2), these files constitute the `MCPL` distribution. The present text concerns the initial public release of `MCPL`, version 1.0.0. Future updates to the distribution will be made available at the project website [5].

2. The `MCPL` format

`MCPL` is a binary file format in which a header section, with configuration and meta-data, is followed by a data section, where the state information of the contained particles is kept. Data compression is available but optional (cf. section 2.4). The uncompressed storage size of a particle entry in the data section is determined by overall settings in the header section, and depends on what exact information is stored for the particles in a given file, as will be discussed shortly. Within a given file, all particle entries will always be of equal length, allowing for trivial calculation of the absolute data location for a particle at a given index in the file – and thus for efficient seeking and skipping between particles if desired. As it is expected that `MCPL` files will always be read by calls to the functions in `mcpl.h`, no attempt will be made here to provide a complete specification of the binary layout of data in the files. Instead, interested readers are referred to the implementation of the functions `mcpl_write_header` and `mcpl_add_particle` in `mcpl.c`.

2.1. Information available

The information available in the file header is indicated in Table 1: a unique 4-byte magic number identifying the format always starts all files, and is followed

File header information	
Field	*Description*
File type magic number 0x4d43504c ("MCPL")	All MCPL files start with this 4-byte word.
Version	File format version.
Endianness	Whether numbers in file are in little- or big-endian format.
Number of particles in file	64 bit integer.
Flag : Particles have polarisation info	If false, all loaded particles will have polarisation vectors (0,0,0).
Flag : Particles have "userflags" field	If false, all loaded particles will have userflags 0x00000000.
Flag : Particle info use double-precision	If true, floating points storage use double-precision.
Global pdgcode	If this 32 bit integer is non-zero, all loaded particles will have this pdgcode.
Source name	String indicating the application which created the MCPL file.
Comments	A variable number of comments (strings) added at file creation.
Binary blobs	A variable number of binary data blobs, indexed by keys (strings). This allows arbitrary custom data to be embedded.

Table 1: Information available in the header section of `MCPL` files.

by the format version, the endianness (*little* or *big*) in which numbers in the file are stored, and the number of particles in the file. The versioning provides a clear path for future updates to the format, without losing the ability to read files created with previous versions of the `MCPL` code, and the endianness information prevents interpretation errors on different machines (although at present, most consumer platforms are little-endian).[1] Next comes four options affecting what data is stored per-particle, which will be discussed in the next paragraph. Finally, the header contains several options for embedding custom free-form information: first of all, the source name, in the form of a single string containing the name and perhaps version of the application which created the file. Secondly, any number of strings can be added as human readable comments, and, thirdly, any number of binary data blobs can be added, each identified by a string key. The `MCPL` format itself provides no restrictions on what data, if any, can be stored in these binary blobs, but useful content could for instance be a copy of configuration data used by the source application when the given file was produced, kept for later reference.

[1] In the current implementation, reading a little-endian `MCPL` file on a big-endian machine or vice versa triggers an error message. It is envisioned that a future version of the `MCPL` code could instead transparently correct the endianness at load time.

Particle state information		
Field	*Description*	*Bytes of storage used per entry (FP = 4 or 8 bytes)*
PDG code	32 bit integer indicating particle type.	0 or 4
Position	Vector, values in centimetres.	3FP
Direction	Unit vector along the particle momentum.	2FP
Kinetic energy	Value in MeV.	1FP
Time	Value in milliseconds.	1FP
Weight	Weight or intensity.	1FP
Polarisation	Vector.	0 or 3FP
User flags	32 bit integer with custom info.	0 or 4

Table 2: Particle state information available and uncompressed storage requirements for each entry in the data section of `MCPL` files.

Table 2 shows the state information available per-particle in `MCPL` files, along with the storage requirements of each field. Particle position, direction, kinetic energy, time and weight are always stored.[2] Polarisation vectors and so-called *user-flags* in the form of unsigned 32 bit integers are only stored when relevant flags in the header are enabled. Likewise, the particle type information in the form of so-called PDG codes is only stored explicitly in each entry when a global PDG code was not specified in the header. The PDG codes must follow the scheme developed by the Particle Data Group in [6, ch. 42], which is inarguably the most comprehensive and widely adopted standard for particle type encoding in simulations. Finally, again depending on a flag in the header, particle information uses either single (4 bytes) or double precision (8 bytes) storage for floating point numbers. All in all, summing up the numbers in the last column of Table 2, particles are seen to consume between 32 and 96 bytes of uncompressed storage space per entry. The `MCPL` format is thus designed to be flexible enough to handle use-cases requiring a high level of detail in the

[2]Note that a valid alternative to storing the directional unit vector along with the kinetic energy would have been the momentum vector. However, the choice here is consistent with the variables used in interfaces of both `MCNP` and `Geant4`, and means that the `mcpl2ssw` converter discussed in section 3.2 can be implemented without access to an unwieldy database of particle and isotope masses.

particle state information, without imposing excessive storage requirements on less demanding scenarios.

Packing of the three-dimensional unit directional vector into just two floating point numbers of storage is carried out via a custom octahedral packing algorithm inspired by [7, 8]. This packing avoids the very significant loss of numerical precision otherwise resulting when using the straight-forward and widespread solution of storing two Cartesian components, u_x and u_y, directly and recovering the magnitude of the third by the expression $|u_z| = \sqrt{1 - u_x^2 - u_y^2}$.

2.2. Accessing or creating `MCPL` files programmatically

The main feature provided by the implementation of `MCPL` in `mcpl.h` and `mcpl.c` is naturally the ability to create new `MCPL` files and access the contents of existing ones, using a set of dedicated functions. No matter which settings were chosen when a given `MCPL` file was created, the interface for accessing the header and particle state information within it is the same, as can be seen in Listing 1: after obtaining a file handle via `mcpl_open_file`, a pointer to an `mcpl_particle_t struct`, whose fields contain the state information available for a given particle, is returned by calling `mcpl_read`. This also advances the position in the file, and returns a null-pointer when there are no more particles in the file, ending the loop. If a file was created with either polarisation vectors or user-flags disabled, the corresponding fields on the particle will contain zeroes (thus representing polarisation information with null-vectors and user-flags with an integer with no bits enabled). All floating point fields on `mcpl_particle_t` are represented with a double-precision type, but the actual precision of the numbers will obviously be limited to that stored in the input file. In addition to the interface illustrated by Listing 1, functions can be found in `mcpl.h` for accessing any information available in the file header (see Table 1), or for seeking and skipping to particles at specific positions in the file, rather than simply iterating through the full file.

Code creating `MCPL` files is typically slightly more involved, as the creation process also involves deciding on the values of the various header flags and

Listing 1: Simple example for looping over all particles in an existing `MCPL` file.

```
#include "mcpl.h"

void example()
{
  mcpl_file_t f = mcpl_open_file("myfile.mcpl");

  const mcpl_particle_t* p;

  while ( ( p = mcpl_read(f) ) ) {

    /* Particle properties can here be accessed
       through the pointer "p":

       p->pdgcode
       p->position[k] (k=0,1,2)
       p->direction[k] (k=0,1,2)
       p->polarisation[k] (k=0,1,2)
       p->ekin
       p->time
       p->weight
       p->userflags
    */

  }

  mcpl_close_file(f);
}
```

filling of free-form information like source name and comments. An example producing a file with 1000 particles is shown in Listing 2. The first part of the procedure is to obtain a file handle through a call to `mcpl_create_file`, configure the header and overall flags, and prepare a zero-initialised instance of `mcpl_particle_t`. Next comes the loop filling the particles into the file, which happens by updating the state information on the `mcpl_particle_t` instance as needed, and passing it to `mcpl_add_particle` each time. At the end, a call to `mcpl_close_outfile` finishes up by flushing all internal buffers to disk and updating the field containing the number of particles at the beginning of the file.

Should the unfortunate happen, that the program is aborted before the call to `mcpl_close_outfile`, particles already written into the output file are normally recoverable: upon opening such an incomplete file, the `MCPL` code detects that the actual size of the file is inconsistent with the value of the field in the header containing the number of particles. Thus, it emits a warning

Listing 2: Simple example for creating an MCPL file with 1000 particles.

```c
#include "mcpl.h"

void example()
{
  mcpl_outfile_t f = mcpl_create_outfile("myfile.mcpl");
  mcpl_hdr_set_srcname(f,"MyAppName-1.0");

  /* Tune file options or add custom comments or
     binary data into the header:

     mcpl_enable_universal_pdgcode(f,myglobalpdgcode);
     mcpl_enable_userflags(f);
     mcpl_enable_polarisation(f);
     mcpl_enable_doubleprec(f);
     mcpl_hdr_add_comment(f,"Some comment.");
     mcpl_hdr_add_data(f,"mydatakey",
                       my_datalength, my_databuf)
  */

  mcpl_particle_t* p = mcpl_get_empty_particle(f);

  int i;
  for ( i = 0; i < 1000; ++i ) {

    /* The following particle properties must
       always be set here:

       p->position[k] (k=0,1,2)
       p->direction[k] (k=0,1,2)
       p->ekin
       p->time
       p->weight

       These should also be set when required by
       file options:

       p->pdgcode
       p->userflags
       p->polarisation[k] (k=0,1,2)
    */

    mcpl_add_particle(f,p);
  }

  mcpl_close_outfile(f);
}
```

Listing 3: Example extracting low-energy neutrons (pdgcode 2112) from an MCPL file.

```c
#include "mcpl.h"

void example() {

  /* open files, transfer meta-data, add comment */

  mcpl_file_t fi = mcpl_open_file("myfile.mcpl");
  mcpl_outfile_t fo = mcpl_create_outfile("new.mcpl");
  mcpl_transfer_metadata(fi, fo);
  mcpl_hdr_add_comment(fo,"Extracted neutrons with ekin<0.1MeV");

  /* transfer selected particles */

  const mcpl_particle_t* particle;
  while ( ( particle = mcpl_read(fi) ) ) {
    if ( particle->pdgcode == 2112 && particle->ekin < 0.1 )
      mcpl_add_particle(fo,particle);
  }

  /* finish up */

  mcpl_closeandgzip_outfile(fo);
  mcpl_close_file(fi);
}
```

message and calculates a more appropriate value for the field, ignoring any partially written particle state entry at the end of the file. This ability to transparently correct incomplete files upon load also means that it is possible to inspect (with the `mcpltool` command discussed in section 2.3) or analyse files that are still being created. To avoid seeing a warning each time a file left over from an aborted job is opened, `mcpl.h` also provides the function `mcpl_repair` which can be used to permanently correct the header of the file.

Likewise, `mcpl.h` also provides the function `mcpl_merge` which can be used to merge two compatible `MCPL` files into one, which might typically be useful when gathering up the output of simulations carried out via parallel processing techniques. Compatibility here means that the files must have essentially identical header sections, except for the field holding the number of particles. Finally, the function `mcpl_transfer_metadata` can be used to easily implement custom extraction of particle subsets from existing `MCPL` files into new (smaller) ones. An example of this is illustrated in Listing 3.

2.3. Accessing `MCPL` files from the command line

Compared with simpler text-based formats (e.g. ASCII files with data formatted in columns), one potential disadvantage of a binary data format like `MCPL` is the lack of an easy way for users to quickly inspect a file and investigate its contents. To alleviate this, `mcpl.h` provides a function which, in a straight-forward manner, can be used to build a generic `mcpltool` command-line executable: `int mcpl_tool(int argc,char** argv)`. Simply running this command on an `MCPL` file without specifying other arguments, results in a short summary of the file content being printed to standard output. This includes a listing of the first 10 contained particles, and an example of such a summary is provided in Listing 4: it is clear from the displayed meta-data that the particles in the given file represent a transmission spectrum resulting from illumination of a block of lead by a 10 GeV proton beam in a `Geant4` [9, 10] simulation. The displayed header information and data columns should be mostly self-explanatory, noting that (`x`, `y`, `z`) indicates the particle position, (`ux`, `uy`, `uz`) its normalised direction, and that the `pdgcode` column indeed shows particle types typical in a hadronic shower: π^+ (211), γ (22), protons (2212), π^- (-211) and neutrons (2112). If the file had user-flags or polarisation vectors enabled, appropriate columns for those would be shown as well. Finally, note that the `36 bytes/particle` refers to uncompressed storage, and that in this particular case the file actually has a compression ratio of approximately 70% particle (cf. section 2.4).

By providing suitable arguments (found by the command `mcpltool -help`) to `mcpltool`, it is possible to modify what information from the file is displayed. This includes the possibility to change what particles from the file, if any, should be listed, as well as the option to extract the contents of a given binary data blob to standard output. The latter might be particularly handy when entire configuration files have been embedded (cf. sections 3.2 and 3.3). Finally, the `mcpltool` command also allows file merging and repairing, as discussed in section 2.2, and provides functionality for selecting a subset of particles from a given file and extracting them into a new smaller file.

Advanced functionality such as graphics display and interactive GUI-based

Listing 4: Example output of running `mcpltool` with no arguments on a specific `MCPL` file.

```
Opened MCPL file myoutput.mcpl.gz:

  Basic info
    Format             : MCPL-2
    No. of particles   : 1106933
    Header storage     : 140 bytes
    Data storage       : 39849588 bytes

  Custom meta data
    Source             : "G4MCPLWriter [G4MCPLWriter]"
    Number of comments : 1
          -> comment 0 : "Transmission spectrum from 10GeV proton beam on 20cm lead"
    Number of blobs    : 0

  Particle data format
    User flags         : no
    Polarisation info  : no
    Fixed part. type   : no
    FP precision       : single
    Endianness         : little
    Storage            : 36 bytes/particle

index      pdgcode    ekin[MeV]       x[cm]       y[cm]       z[cm]          ux          uy          uz    time[ms]      weight
    0          211       487.02     -0.5898       1.835          20   -0.092407     0.20491     0.97441  7.3346e-07           1
    1           22       1.5326      1.0635      11.351          20    0.080441     0.66026     0.74672  1.0882e-06           1
    2           22       3.9526    -0.43907      8.6473          20    -0.56616     0.50558     0.65104  1.0286e-06           1
    3           22      0.82591      1.7444      9.7622          20    0.092099     0.79597     0.59829  1.0378e-06           1
    4           22       1.1958      2.1806      8.6416          20     0.21997     0.66435     0.71432  1.0124e-06           1
    5           22       1.2525      3.0949      7.7366          20     0.48903     0.30789     0.81612   1.013e-06           1
    6           22       2.6247       3.948       5.681          20     0.62503     0.64221     0.44374  9.1152e-07           1
    7         2212       824.28     -1.8797     -2.5124          20     -0.3077    -0.40496       0.861  7.6539e-07           1
    8         -211       3459.8    -0.79521     0.91481          20    -0.13441     0.14438     0.98035  7.0618e-07           1
    9         2112      0.30553      54.471      33.386          20      0.4862    0.011958     0.87377  0.00016442           1
```

investigation or manipulation of the contents of MCPL files is not provided by the mcpltool, since those would imply additional unwanted dependencies to the core MCPL code, which is required by design to be light-weight and widely portable. However, it is the hope that the existence of a standard format like MCPL will encourage development of such tools, and indeed some already exist in the in-house framework [11] of the Detector Group at the European Spallation Source (ESS) [12, 13]. It is intended for a future distribution of MCPL to include relevant parts of these tools as a separate and optional component.

2.4. Compression

The utilisation of data compression in a format like MCPL is potentially an important feature, since on-disk storage size could be a concern for some applications. Aiming to maximise flexibility, transparency and portability, optional compression of MCPL files is simply provided by allowing whole-file compression into the widespread GZIP format [14] (changing the file extension from .mcpl to .mcpl.gz in the process). This utilises the DEFLATE compression algorithm [15] which offers a good performance compromise with a reasonable compression ratio and an excellent speed of compression and decompression.

Relying on a standard format such as GZIP means that, if needed, users can avail themselves of existing tools (like the gzip and gunzip commands available on most UNIX platforms) to change the compression state of an existing MCPL file. However, when the code in mcpl.c is linked with the ubiquitous ZLIB [16, 17] (cf. section 2.5), compressed MCPL files can be read directly. For convenience, mcpl.h additionally provides a function mcpl_closeandgzip_outfile, which can be used instead of mcpl_close_outfile (cf. Listing 2) to ensure that newly created MCPL files are automatically compressed if possible (either through a call to an external gzip command or through custom ZLIB-dependent code, depending on availability).

2.5. Build and deployment

It is the hope that eventually MCPL capabilities will be included upstream in many applications, and that users of those consequently won't have to do

anything extra to start using it. As will be discussed in section 3, this is at present the case for users of recent versions of McStas [18, 19] and McXtrace [20], and is additionally the case for users of the in-house Geant4-based framework of the ESS Detector Group [11].

By design, it is expected that most developers wishing to add MCPL support to their application will simply place copies of mcpl.h and mcpl.c into their existing build system and include mcpl.h from either C or C++ code.[3] In order to make the resulting binary code able to manipulate compressed files directly (cf. section 2.4), the code in mcpl.c must usually be compiled against and linked with an installation of ZLIB (see detailed instructions regarding build flags at the top of mcpl.c). Alternatively, the MCPL distribution presented here contains a "fat" auto-generated drop-in replacement for mcpl.c named mcpl_fat.c, in which the source code of ZLIB has been included in its entirety.[4] Using this somewhat larger file enables ZLIB-dependent code in MCPL even in situation where ZLIB might not be otherwise available.

In addition to the core MCPL code, the MCPL distribution also contains a small file providing the mcpltool executable, C++ files implementing the Geant4 classes discussed in section 3.1, C files for the mcpl2ssw and ssw2mcpl executables discussed in section 3.2, and a few examples show-casing how user code might look.

Building of all of these parts should be straight-forward using standard tools, but a configuration file for CMake [21] which builds and installs everything is nonetheless provided for reference and convenience. Additionally, "fat" single-file versions of all command line utilities (mcpltool, mcpl2ssw and ssw2mcpl) are also provided, containing both MCPL and ZLIB code within as appropriate.

[3] Compilation of mcpl.c can happen with any of the following standards: C99, C11, C++98, C++11, C++14, or later. In addition to those, mcpl.h is also C89 compatible. Note that on platforms where the standard C math function sqrt is provided in a separate library, that library must be available at link-time.

[4] Note that all ZLIB symbols have been prefixed, to guard against potential run-time clashes where a separate ZLIB is nonetheless loaded.

Thus, any of these single-file versions can be compiled directly into the corresponding command line executable, without any other dependencies than a `C` compiler.

3. Application-specific converters and plugins

While the examples in section 2.2 show how it is possible to manipulate `MCPL` files directly from `C` or `C++` code, it is not envisioned that most users will have to write such code themselves. Rather, in addition to using available tools (such as the `mcpltool` described in section 2.3) to access the contents of files as needed, users would ideally simply use pre-existing plugins and converters written by application-specific experts, to load particles from `MCPL` files into their given Monte Carlo applications, or extract particles from those into `MCPL` files. At the time of this initial public release of `MCPL`, four such applications are already `MCPL`-aware in this manner: `Geant4`, `MCNP`, `McStas` and `McXtrace`, and the details of the corresponding converters and plugins are discussed in the following sub-sections, after a few general pieces of advice for other implementers in the next paragraphs.

In order for `MCPL` files to be as widely exchangeable as possible, code loading particles from `MCPL` files into a given Monte Carlo application should preferably be as accepting as possible. In particular, this means that warnings rather than errors should result if the input file contains PDG codes corresponding to particle types that can not be handled by the application in question. As an example, a detailed `MCNP` or `Geant4` simulation of a moderated neutron source will typically produce files containing not only neutrons, but also gammas and other particles. It should certainly be possible to load such a file into a neutron-only simulation application like `McStas`, resulting in simulation of the contained neutrons (preferably with a warning or informative message drawing attention to some particles being ignored).

Applications employing parallel processing techniques, must always pay particular attention when implementing file-based I/O, and this is naturally also

the case when creating MCPL-aware plugins for them. However, the available functionality for merging of MCPL files makes the scenario of file creation particularly simple to implement: each sub-task can simply write its own file, with the subsequent merging into a single file taking place during post-processing. For reading of particles in existing MCPL files, it is recommended that each sub-task performs a separate call to `mcpl_open_file`, and use the skipping and seeking functionality to load just a subset of the particles within, as required. In the case of a multi-threading application, it is of course also possible to handle concurrent input or output directly through a single file handle. In this case, however, calls to `mcpl_add_particle` and `mcpl_read` must be protected against concurrent invocations with a suitable lock or mutex.

The following three sub-sections are dedicated to discussions of presently available MCPL interfaces for specific Monte Carlo applications. The discussions will in each case presuppose familiarity with the application in question.

3.1. Geant4 interface

In the most typical mode of working with the Geant4 [9, 10] toolkit, users create custom C++ classes, sub-classing appropriate abstract interfaces, in order to set up geometry, particle generation, custom data readout and physics modelling. At run-time, those classes are then instantiated and registered with the framework. Accordingly, the MCPL–Geant4 integration takes the form of two such sub-classes of Geant4 interface classes, which can be either directly instantiated or further sub-classed themselves as needed: `G4MCPLGenerator` and `G4MCPLWriter`. They are believed to be compatible with any recent version of Geant4 and were explicitly tested with versions 10.00.p03 and 10.02.p02.

First, the `G4MCPLGenerator`, the relevant parts of which are shown in Listing 5, is a `G4VUserPrimaryGeneratorAction` which must be provided with the path to an MCPL file when constructed and in the `GeneratePrimaries` method of which the particles in the input file are used to generate Geant4 events with a single primary particle in each. If the file runs out of particles before the Geant4 simulation is ended for other reasons, the `G4MCPLGenerator` graciously requests

Listing 5: The `G4MCPLGenerator` class.

```
class G4MCPLGenerator : public G4VUserPrimaryGeneratorAction
{
  public:

    G4MCPLGenerator(const G4String& inputFile);
    virtual ~G4MCPLGenerator();
    virtual void GeneratePrimaries(G4Event*);

  protected:

    //Reimplement this to filter input particles (default
    //implementation accepts all particles):
    virtual bool UseParticle(const mcpl_particle_t*) const;

    //Reimplement this to change coordinates or weights of
    //input particles before using them (does nothing by
    //default):
    virtual void ModifyParticle(G4ThreeVector& position,
                                G4ThreeVector& direction,
                                G4ThreeVector& polarisation,
                                G4double& time,
                                G4double& weight) const;

  private:
    // ..
};
```

the `G4RunManager` to abort the simulation. Thus, a convenient way in which to use the entire input file for simulation is to launch the simulation with a very high number of events requested, as is done in the example in Listing 6.[5]

In case the user wishes to use only certain particles from the input file for simulation, the `G4MCPLGenerator` class must be sub-classed and the `UseParticle` method reimplemented, returning `false` for particles which should be skipped. Likewise, if it is desired to perform coordinate transformations or reweighing before using the loaded particles, the `ModifyParticle` method must be reimplemented.

The `G4MCPLWriter` class, the relevant parts of which are shown in Listing 7, is a `G4VSensitiveDetector` which in the default configuration "consumes" all particles which, during a simulation, enter any geometrical volume(s) to which

[5] Unfortunately, due to a limitation in the `G4RunManager` interface, this number will be limited by the highest number representable with a `G4int`, which on most modern platforms is 2147483647.

Listing 6: Example showing how to load particles from an MCPL file into a Geant4 simulation.

```
#include "G4MCPLGenerator.hh"
#include "G4RunManager.hh"
#include <limits>

//Not shown here: Code defining MyGeometry and MyPhysicsList.

int main( int argc, char** argv ) {

  G4RunManager runManager;
  runManager.SetUserInitialization(new MyGeometry);
  runManager.SetUserInitialization(new MyPhysicsList);
  runManager.SetUserAction(new G4MCPLGenerator("myfile.mcpl"));
  runManager.Initialize();
  runManager.BeamOn(std::numeric_limits<G4int>::max());

  return 0;
}
```

it is attached by the user and stores them into the specified MCPL file. At the same time it asks Geant4 to end further simulation of those particles ("killing" them). This strategy of killing particles stored into the file was chosen as a sensible default behaviour, as it prevents potential double-counting in the scenarios where a particle (or its induced secondary particles) would otherwise be able to enter a given volume multiple times. If it is desired to modify this strategy, the user must sub-class G4MCPLWriter and reimplement the ProcessHits method, using calls to storePreStep, storePostStep and Kill, as appropriate. For reference, code responsible for the default implementation is shown in Listing 8. Likewise, to add MCPL user-flags into the file, the UserFlagsDescription and UserFlags methods must simply be reimplemented - the description naturally ending up as a comment in the output file.

In Listing 9 is shown how the G4MCPLWriter will typically be configured and attached to logical volume(s) of the geometry.

3.2. MCNP interface

Most users of MCNP are currently employing one of three distinct flavours: MCNPX [22, 23], MCNP5 [24] or MCNP6 [25]. In the most typical mode of working with any of these software packages, users edit and launch MCNP through the use of text-based configuration files (so-called *input decks*), in order to set up details

Listing 7: The `G4MCPLWriter` class.

```
class G4MCPLWriter : public G4VSensitiveDetector
{
  public:

    //Basic interface:

    G4MCPLWriter( const G4String& outputFile,
                  const G4String& name = "G4MCPLWriter" );
    virtual ~G4MCPLWriter();

    void AddComment( const G4String& );
    void AddData( const G4String& data_key,
                  size_t data_length,
                  const char* data );
    void EnableDoublePrecision();
    void EnablePolarisation();

    //Optional reimplement this to change default
    //"store-and-kill at entry" strategy:

    virtual G4bool ProcessHits( G4Step * step,
                                G4TouchableHistory* );

    //Optional reimplement these to add MCPL userflags:

    virtual G4String UserFlagsDescription() const { return ""; }
    virtual uint32_t UserFlags(const G4Step*) const { return 0; }


  protected:
    //Methods that can be used if reimplementing ProcessHits():
    void storePreStep(const G4Step *);
    void storePostStep(const G4Step *);
    void Kill(G4Step *);

  private:
    // ...
};
```

Listing 8: The default `ProcessHits` implementation in the `G4MCPLWriter` class.

```
G4bool G4MCPLWriter::ProcessHits(G4Step * step,G4TouchableHistory*)
{
  //Only consider particle steps originating at the boundary
  //of the monitored volume:
  if ( step->GetPreStepPoint()->GetStepStatus() != fGeomBoundary )
    return false;

  //Store the state at the beginning of the step, but avoid
  //particles taking their very first step (this would double-
  //count secondary particles created at the volume edge):
  if ( step->GetTrack()->GetCurrentStepNumber() > 1 )
    storePreStep(step);

  //Tell Geant4 to stop further tracking of the particle:
  Kill(step);
  return true;
}
```

Listing 9: Example showing how to produce an MCPL file from a Geant4 simulation.

```
//Provide output filename when creating G4MCPLWriter instance:
G4MCPLWriter * mcplwriter = new G4MCPLWriter("myoutput.mcpl");

//Optional calls which add meta-data or change flags:
mcplwriter->AddComment("Some useful description here");
mcplwriter->AddData( ... );
mcplwriter->EnableDoublePrecision();
mcplwriter->EnablePolarisation();

//Register with G4SDManager and on one or more logical
//volumes to activate:
G4SDManager::GetSDMpointer()->AddNewDetector(mcplwriter);
alogvol->SetSensitiveDetector(mcplwriter);
```

of the simulation including geometry, particle generation, and data extraction. The latter typically resulting in the creation of data files containing simulation results, ready for subsequent analysis.

Although it would be conceivable to write in-process FORTRAN-compatible MCPL hooks for MCNP, such an approach would require users to undertake some form of compilation and linking procedure. This would thus be likely to impose a change in working mode for the majority of MCNP users, in addition to possibly requiring a special license for source-level access to MCNP. Instead, the MCNP–MCPL interface presented here exploits the existing MCNP capability to stop and subsequently restart simulations at a user-defined set of surfaces through the Surface Source Write/Read (SSW/SSR) functionality. As the name suggests, the state parameters of simulated particles crossing a given surface are stored on disk in dedicated files, with the intentional use as a surface source in subsequent simulations with the same MCNP setup. Presumably, these files (henceforth denoted "SSW files" in the present text) are intended for this internal intermediate usage only, their format differs between different flavours of MCNP, and little effort has been made to document the format in publicly available manuals. Despite these obstacles, the SSW format is stable enough that several existing MCNP-aware tools (e.g. [26, 27, 28]) have chosen to provide converters for this format, with various levels of functionality, and it was thus deemed suitable also for the needs of the MCPL project.

Listing 10: Usage instructions for the `ssw2mcpl` command.

```
Usage:

  ssw2mcpl [options] input.ssw [output.mcpl]

Converts the Monte Carlo particles in the input.ssw file (MCNP Surface
Source Write format) to MCPL format and stores in the designated output
file (defaults to "output.mcpl").

Options:

  -h, --help    : Show this usage information.
  -d, --double  : Enable double-precision storage of floating point values.
  -s, --surf    : Store SSW surface IDs in the MCPL userflags.
  -n, --nogzip  : Do not attempt to gzip output file.
  -c FILE       : Embed entire configuration FILE (the input deck)
                  used to produce input.ssw in the MCPL header.
```

Thus, the MCPL distribution presented here includes dependency-free C code for two standalone executables, `mcpl2ssw` and `ssw2mcpl`, which users can invoke from the command-line in order to convert between MCPL and SSW files.[6] The usage of these two executables will be discussed here, while users are referred to the relevant MCNP manuals for details of how to set up their input decks to enable SSW input or output in their MCNP simulations: [29, Ch. II.3.7], [30, Ch. 5.5.5] and [31, Ch. 3.3.4.7]. Note that through usage of `ssw2mcpl` and `mcpl2ssw`, it is even possible to transfer particles between different flavours and versions of MCNP, which is otherwise not possible with SSW files.

First, the `ssw2mcpl` command, for which the full usage instructions are shown in Listing 10, is in its most base invocation straight-forward to use. Simply provide it with the name of an existing SSW file to run on, and it will result in the creation of a new (compressed) MCPL file, `output.mcpl.gz`, containing a copy of all particles found in the SSW file. The MCNP flavour responsible for creating the SSW file is automatically detected, the resulting differences in the file format are taken into account behind the scenes, and the detected MCNP version is documented as a comment in the header of the resulting MCPL file.

The only relevant piece of information which is by default not transferred from the SSW particle state into the MCPL file is the numerical ID of the surface

[6] Prior work in [26, 27] served as valuable input when developing code for interpreting data sections in SSW files.

where the particle was registered in the MCNP simulation. By supplying the `-s` option, `ssw2mcpl` will transfer those to the MCPL user-flags field, and document this in the MCPL header. Additionally, while floating point numbers in the SSW file are always stored in double-precision, the transfer to MCPL will by default convert them to single-precision. This was chosen as the default behaviour to keep usual storage requirements low, as single-precision is arguably sufficient for most studies. By supplying the `-d` option, `ssw2mcpl` will keep the numbers in double-precision in the MCPL file as well. Depending on compression and the applied flags, the on-disk size of the resulting MCPL file will typically be somewhere between 20% from which it was converted.

Finally it is possible, via the `-c FILE` flag, to point the `ssw2mcpl` command to the input deck file used when producing the provided SSW file. Doing so will result in a complete copy of that file being stored in the MCPL header as a binary data blob under the string key `"mcnp_input_deck"`, thus providing users with a convenient snapshot in the MCPL file of the MCNP setup used. Unfortunately, it was not possible to automate this procedure completely, and it thus relies on the user to provide the correct input deck for a given SSW file, but the `ssw2mcpl` command does at least check that the specified file is a text-file containing somewhere the problem title which was also embedded in the SSW file. The input deck embedded in a given MCPL file can later be inspected from the command line by invoking the command "`mcpltool -bmcnp_input_deck <file.mcpl>`".

Usage of the `mcpl2ssw` command, for which the full usage instructions are shown in Listing 11, is slightly more involved: in addition to an input MCPL file, the user must also supply a reference SSW file in a format suitable for the MCNP setup in which the resulting SSW file is subsequently intended to be used as input. The need for this added complexity stems from the constraint that the SSW format is merely intended as an internal format in which it is possible to stop and restart particles while remaining within a given setup of an MCNP simulation – meaning at the very least that the MCNP version and the configuration of the geometrical surfaces involved in the `Surface Source Write/Read` procedure must be unchanged. Thus, for maximal robustness, the user must supply a

Listing 11: Usage instructions for the `mcpl2ssw` command.

```
Usage:

  mcpl2ssw [options] <input.mcpl> <reference.ssw> [output.ssw]

Converts the Monte Carlo particles in the input MCPL file to SSW format
(MCNP Surface Source Write) and stores the result in the designated output
file (defaults to "output.ssw").

In order to do so and get the details of the SSW format correct, the user
must also provide a reference SSW file from the same approximate setup
(MCNP version, input deck...) where the new SSW file is to be used. The
reference SSW file can of course be very small, as only the file header is
important (the new file essentially gets a copy of the header found in the
reference file, except for certain fields related to number of particles
whose values are changed).

Finally, one must pay attention to the Surface ID assigned to the
particles in the resulting SSW file: Either the user specifies a global
one with -s<ID>, or it is assumed that the MCPL userflags field in the
input file is actually intended to become the Surface ID. Note that not
all MCPL files have userflag fields and that valid Surface IDs are
integers in the range 1-999999.

Options:

  -h, --help    : Show this usage information.
  -s<ID>        : All particles in the SSW file will get this surface ID.
  -l<LIMIT>     : Limit the number of particles transferred to the SSW file
                  (defaults to 2147483647, the maximal SSW capacity).
```

reference `SSW` file which was produced by the setup in which the `SSW` file created with `mcpl2ssw` is to be used (it does not matter how many particles the reference file contains). What will actually happen is that in addition to the particle state data itself, the newly created `SSW` file will contain the exact same header as the one in the reference `SSW` file, apart from the fields related to the number of particles in the file.

Additionally, the user must consider carefully which `MCNP` surface IDs the particles from the `MCPL` file should be associated with, once transferred to the `SSW` file. By default it will assume that the `MCPL` user-flags field contains exactly this ID, but more often than not, users will have to specify a global surface ID for all of the particles through the `-s<ID>` command-line option for the `mcpl2ssw` command.

Finally, note that `SSW` files do not contain polarisation information, and any such polarisation information in the input `MCPL` file will consequently be discarded in the translation. Likewise, in cases where the input `MCPL` file contains one or more particles whose type does not have a representation in the targeted

flavour of MCNP, they will be ignored with suitable warnings.

3.3. McStas and McXtrace interfaces

Recent releases of the neutron ray tracing software package McStas [18, 19] (version 2.3 and later) and its X-ray sibling package McXtrace [20] (version 1.3 and later) both include MCPL-interfaces. Although McStas and McXtrace are two distinct software packages, they are implemented upon a common technological platform, McCode, and the discussions here will for simplicity use the term McCode where the instructions are otherwise identical for users of the two packages.

The particle model adopted in McCode is directly compatible with MCPL. In essence, apart from mere unit conversions, particles are read from or written to MCPL files at one or more predefined logical points defined in the McCode configuration files (so-called *instrument files*). Specifically, two new components, MCPL_input and MCPL_output, have been provided, which users can activate by adding entries at relevant points in their instrument files as is usual when working with McCode.

First, when using the MCPL_input component, particles are directly read from an MCPL input file and injected into the simulation at the desired point, thus playing the role of a source model. In Listing 12 is shown how, in its simplest form, users would insert an MCPL_input component in their instrument file. This will result in the MCPL file being read in its entirety, and all found neutrons (for McStas) or gamma particles (for McXtrace) traced through the McCode simulation. In Listing 13 is indicated how the user can additionally impose an allowed energy range when loading particles by supplying the Emin and Emax parameters. The units are meV and keV respectively for McStas and McXtrace. Thus, the code in Listing 13 would select 12–100 meV neutrons in McStas and 12–100 keV gammas in McXtrace. A particle from the MCPL file is injected at the position indicated by its MCPL coordinates *relative* to the position of the MCPL_input component in the McCode instrument. Thus, a user can impose coordinate transformations by altering the positioning of MCPL_input as shown

Listing 12: Code enabling MCPL input in its simplest form.

```
COMPONENT vin = MCPL_input( filename="myfile.mcpl" )
AT(0,0,0) RELATIVE Origin
```

Listing 13: Code enabling MCPL input, selecting particles in a given energy range.

```
COMPONENT vin = MCPL_input( filename="myfile.mcpl",
                           Emin=12, Emax=100 )
AT(0,0,0) RELATIVE Origin
```

in Listing 14, which would shift the initial position of the particles by (X, Y, Z) and rotate their initial velocities around the x, y and z axes (in that order) by respectively Rx, Ry and Rz degrees. Furthermore, Listing 14 shows a way to introduce a time shift of 2 s to all particles, using an EXTEND code block.

For technical reasons, the number of particles to be simulated in McCode must be fixed at initialisation time. Thus, the number of particles will be set to the total number of particles in the input file, as this is provided through the corresponding MCPL header field. If and when a particle is encountered which can not be used (due to having a wrong particle type or energy), it will lead to an empty event in which no particles leave the source. At the end of the run, the number of particles skipped over will be summarised for the user. This approach obviates the need for running twice over the input file, as well as avoiding the potential introduction of statistical bias from reading a partial file.

Note that if running McCode in parallel processing mode using MPI [32], each process will operate on all particles in the entire file, but the particles will get their statistical weights reduced accordingly upon load. This behaviour is not specific to the MCPL_input component, but is a general feature of how

Listing 14: Code enabling MCPL input, applying spatial and temporal transformations.

```
COMPONENT vin = MCPL_input( filename="myfile.mcpl" )
AT(X,Y,Z) RELATIVE Origin
ROTATED (Rx,Ry,Rz) RELATIVE Origin
EXTEND
%{
        t=t+2;
%}
```

Listing 15: Code enabling MCPL output in its simplest form.

```
COMPONENT mcplout = MCPL_output( filename="myoutput.mcpl" )
AT(0,0,0) RELATIVE PREVIOUS
```

multiprocessing is implemented in McCode.

When adding an MCPL_output component to an McCode instrument file, the complete state of all particles reaching that component are written to the requested output file. In Listing 15 is shown how, in its simplest form, users would insert such a component in their instrument file, and get particles written with coordinates relative to the component preceding it, into the output file. For reference, a copy of the complete instrument file is stored in the MCPL header as a binary data blob under the string key "mccode_instr_file". This feature provides users with a convenient snapshot of the generating setup. The instrument file embedded in a given MCPL file can be inspected from the command line by invoking the command "mcpltool -bmccode_instr_file <file.mcpl>".

If running McCode in parallel processing mode using MPI, each process will create a separate output file named after the pattern myoutput-idx.mcpl where idx is the process number (assuming filename="myoutput.mcpl" as in Listing 15). The resulting files can subsequently be merged using the mcpltool command if desired. It is envisioned that a future version of the MCPL_output component will automatically handle this merging for the user.

To avoid generating unnecessarily large files, the MCPL_output component stores particle state data using the global PDG code feature (cf. section 2.1), uses single precision floating point numbers, and does *not* by default store polarisation vectors. The two latter settings may be changed by the user through the polarisationuse and doubleprec parameters respectively, as shown in listing 16.

Finally, if desired, custom information might be stored per-particle into the MCPL user-flags field for later reference. This could be any property, such as for instance the number of reflections along a neutron guide, or the type of scattering process in a crystal, etc. Listing 17 shows a simple example of this

Listing 16: Code enabling MCPL output with polarisation and double precision numbers.

```
COMPONENT mcplout = MCPL_output( filename="myoutput.mcpl",
                                 polarisationuse=1,
                                 doubleprec=1 )
AT(0,0,0) RELATIVE PREVIOUS
```

Listing 17: Code enabling MCPL output with custom user-flags information.

```
/* some upstream component setting a variable (customvar) */
COMPONENT some_comp = Some_Component( /* some parameters here */ )
AT (0,0,0) ABSOLUTE
EXTEND %{
  customvar=(uint32_t) mcget_run_num();
%}

/* MCPL output capturing customvar into MCPL user-flags */
COMPONENT vout = MCPL_output( filename="myoutput.mcpl",
                              userflag=customvar,
                              userflagcomment="Particle Id" )
AT(0,0,0) RELATIVE PREVIOUS
```

where the particle ID, in the form of its McCode ray number (returned from the McCode library function mcget_run_num), is stored into the user-flags field. A string, userflagcomment, is required in order to describe the significance of the extra data, and will end up as a comment in the resulting MCPL file.

4. Example scientific use cases

The possible uses for MCPL are envisioned to be many and varied, facilitating both straight-forward transfers of particle data between different simulations, as well as data reuse and cross-code comparisons. Actual scientific studies are already being performed with the help of MCPL, demonstrating the suitability of the format "in the field". By way of example, it will be discussed in the following how MCPL is used in two such ongoing studies.

4.1. Optimising the detectors for the LoKI instrument at ESS

The ongoing construction of the European Spallation Source (ESS) [12, 13] has initiated significant development of novel neutronic technologies in the past 5 years. The performance requirements for neutron instruments at the ESS, in particular those resulting from the unprecedented cold and thermal neutron

brightness, are at or beyond the capabilities of detector technologies currently available [33]. Additionally, shortage of ^{3}He [34, 35], upon which the vast majority of previous detectors were based, augments the need for development of new efficient and cost-effective detectors based on other isotopes with high neutronic conversion cross sections.

A typical approach to instrument design and optimisation at ESS involves the development of a `McStas`-based simulation of the instrument. Such a simulation includes an appropriate neutron source description and detailed models of the major instrument components, such as benders, neutron guides, chopper systems, collimators, sample environment and sample. See [36] for an introduction to the role of the various instrument components. Detector components in `McStas` are, however, typically not implemented with any detailed modelling, and are simply registering all neutrons as they arrive. Thus, while the setup in `McStas` allows for an efficient and precise optimisation of most of the instrument parameters, detailed detector optimisation studies must out of necessity be carried out in a separate simulation package, such as `Geant4`.

As the detector development progresses in parallel with the general instrument design, it is crucial to be able to optimise the detector setup for the exact instrument conditions under investigation in `McStas`. The `MCPL` format, along with the interfaces discussed in sections 3.1 and 3.3, facilitates this by allowing for easy transfer of neutron states from the `McStas` instrument simulation into `Geant4` simulations with detailed setups of proposed detector designs.

Technically, this is done by placing the `MCPL_output` component just after the relevant sample component in the `McStas` instrument file. Additionally, using the procedure for creation and storage of custom `MCPL` user-flags also discussed in section 3.3, it is possible to differentiate neutrons that scattered on the sample from those who continued undisturbed, and to carry this information into the `Geant4` simulations. This information is needed to understand the impact of the direct beam on the low angle measurements, in order to study the requirements for a so-called zero-angle detector.

For example, in order to optimise the detector technology that the LoKI

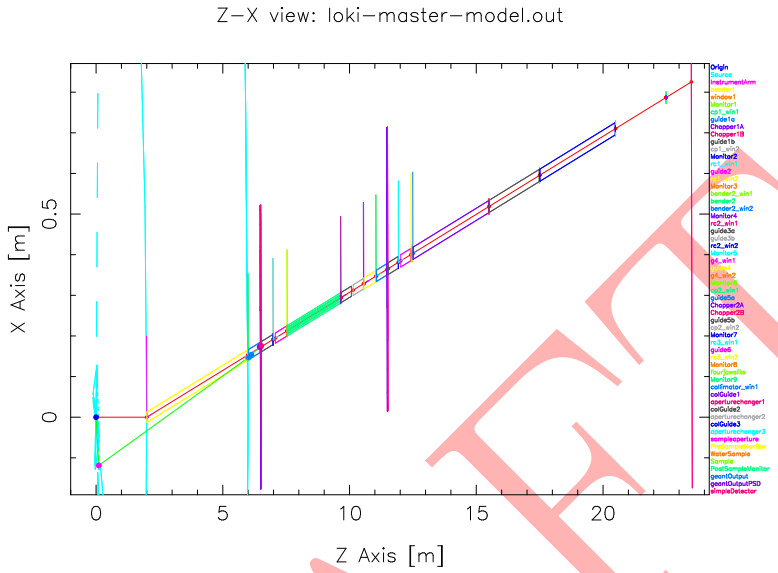

Figure 1: Layout of the `McStas` model of the LoKI instrument. Neutrons originate at the source located at $z = 0$ and progress through the various instrument components toward the sample at $z = 22.5\,\mathrm{m}$.

instrument [37, 38, 39] might adopt, a series of `McStas` simulations of the instrument components and the interactions in realistic samples [40] are performed (see Figure 1 for a view of the instrument in `McStas`). The parameters of the instrument and the samples in the `McStas` model are chosen in such a way, that various aspects of the detector performance can be investigated, including rate capability and spatial resolution. The neutrons emerging from the sample in `McStas` are then transferred via `MCPL` to the detector simulation in `Geant4`, where a detailed detector geometry and appropriate materials are implemented (see Figure 2 for a visualisation of the `Geant4` model).

Neutrons traversing the detector geometry in `Geant4` undergo interactions with the materials they pass on their flight-path, according to the physics processes and respective cross sections available in the setup. Special attention is needed when configuring the `Geant4` physics modelling, to ensure that all pro-

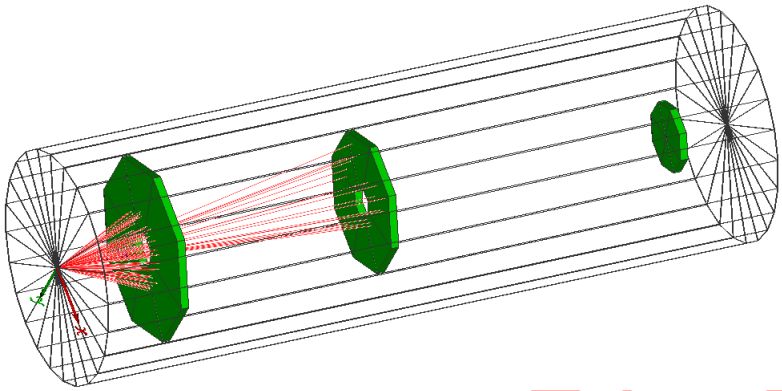

Figure 2: `Geant4` model of a potential detector geometry for the LoKI instrument. Neutrons from the sample hitting the active detector area appear in red.

cesses relevant for neutron detection are taken into account and handled correctly. Specifically, the setup therefore utilises the high-precision neutron models in `Geant4` extended with [41], and is implemented in [11]. In the solid-converter based detectors under consideration, an absorbed neutron emits charged products which then travel a certain range inside the detector and deposit energy in a counting gas. It is possible to extract position and time information from the energy deposition profile and use these space-time coordinates for further analysis, in the same way that hits in a real detector would be treated. This way it becomes possible to reproduce the distributions of observable quantities relevant for Small Angle Neutron Scattering (SANS) analysis [42, 43], such as the resulting raw Q distribution [36, Ch. 2.3.3].

Figure 3 demonstrates such a Q distribution, based on the simulated output of the middle detector bank of LoKI (cf. Figure 2), for a certain instrument setup – including a sample modelled as consisting of spheres with radii of 500 Å. The raw Q distribution is calculated both based on the neutron states as they emerge from the sample in `McStas`, and from the simulated hits in `Geant4`. With such a procedure, resolution-smearing effects can be correctly attributed to their sources, geometrical acceptance and detector efficiency can be studied in detail, and the impact of engineering aspects such as dead space can be accurately

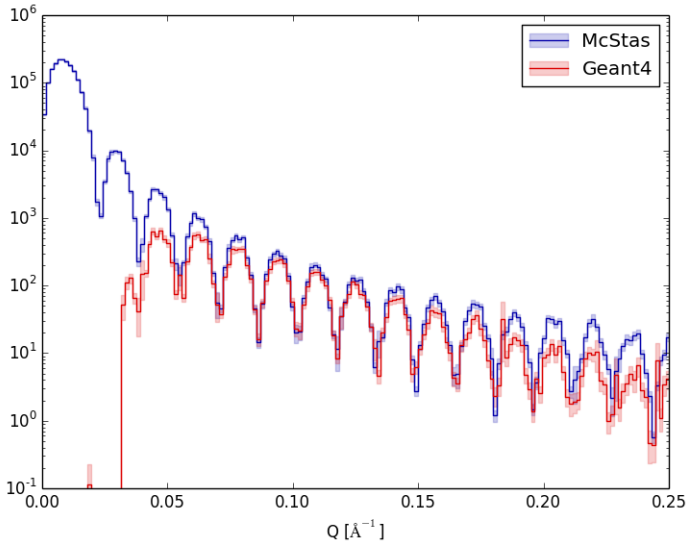

Figure 3: Raw Q distribution for a subset of the LoKI detectors (middle detector bank of Figure 2). The `McStas` post-sample output appears in blue, while the distribution calculated from the detector hits in `Geant4` appears in red.

considered.

4.2. Neutron spectra predictions for cosmogenic dating studies

The use of radionuclides produced in-situ by cosmic rays for dating purposes has, in the last two decades, revolutionised the earth surface sciences [44]. The precise determination of the production rate of such isotopes, like ^{10}Be and ^{26}Al, poses the key challenge for this technique and relies on a folding of cosmic fluxes with energy dependent production cross sections [45]. The present discussion will focus on the evaluation of the neutron flux induced by cosmic radiation, and in particular on how `MCPL` can be exploited both to facilitate the reuse of computationally intensive simulations, and as a means for cross-code comparisons.

At sea level, neutrons constitute the most abundant hadronic component of cosmic ray induced showers, and possess relatively high cross sections for production of isotopes relevant for radionuclide dating. Thus, it is the dominant

contributor to the relevant isotopic production in the first few meters below the surface [46]. Extending further below the surface, the neutron flux decreases rapidly, and as a consequence the isotopic production rate induced by cosmic muons eventually becomes the most significant factor [47, 48]. At a depth of approximately 3 m below the surface, the production rate due to muons is comparable with the rate from neutrons [46]. Considering non-erosive surfaces and samples at depths significantly less than 3 m, the production rates can thus be estimated by considering just the flux of neutrons. Thus, given known cross sections for neutronic production of ^{10}Be or ^{26}Al, properties such as the cosmic irradiation time of a given sample can be directly inferred from its isotopic content – providing information about geological activity. In the present study, Monte Carlo methods are used to simulate atmospheric cosmic rays [49, 50] and subsequently estimate the neutron flux spectra as a function of depth under the surface of the Earth.

Primary cosmic rays constantly bombard the solar system and initiate cosmic ray showers in the Earth's atmosphere, leading to the production of atmospheric neutrons. Full scale simulation of such showers is time consuming, as is illustrated by Figure 4, which shows the trajectories of a simulated air shower induced by a single 100 GeV proton in `Geant4`: very large numbers of secondary particles are generated in each shower, all of which must themselves undergo simulation. On the other hand, simulations of the propagation of sea level neutrons in a few meters of solid material are relatively fast. In the present work of estimating neutron spectra for different underground materials, `MCPL` is used to record particle information at sea level. Using the recorded data as input, subsequent simulations are dedicated to the neutron transport in different underground materials. In this way, repetition of the time consuming parts of the simulation is avoided. `Geant4` is used to simulate the air shower in this work, while both `Geant4` and `MCNPX` are used to simulate neutron spectra underground.

In the `Geant4` simulation of the Earth's atmosphere, the geometry is implemented as a 100 km thick shell with an inner radius of 6387 km, sub-divided

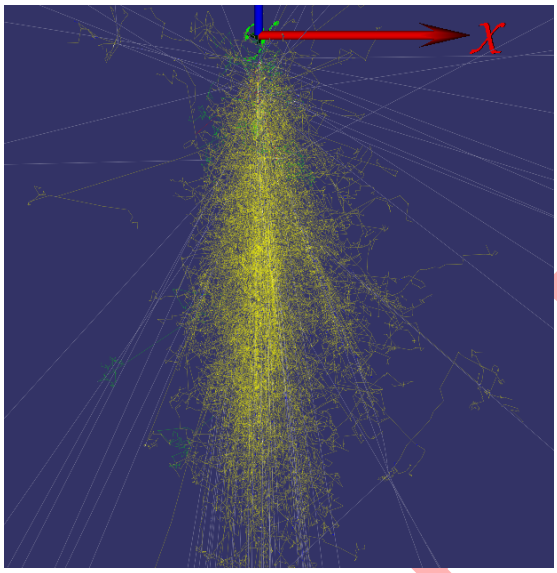

Figure 4: Cosmic shower simulated in `Geant4`. The incident proton energy is 100 GeV and the length of the x-axis is 2 km. The straight grey trajectories are neutrinos. The yellow and green trajectories are photons and neutrons, respectively.

into 50 equally thick layers, the effective temperatures and densities of which are calculated using the "U.S. standard atmosphere, 1976" model [51]. Using the plugins described in section 3.1, the simulation of any particle reaching the inner surface of the atmosphere is ended and its state stored in an `MCPL` file. To compare the simulated and measured [52] spectra at New York city, a lower cutoff of $E_c = 2.08\,\mathrm{GeV}$ on the kinetic energy of the primary proton is applied, to take the geomagnetic field shielding effect at this location into account. The relationship between the number of simulated primary protons, N, and the real world time-span, δt, to which such a sample-size corresponds, is given by the following equation:

$$\delta t = \frac{N}{\int_{E_c}^{\infty} J(E)dE \times 2\pi \times 4\pi r^2}$$

Here, r is the outer radius of the simulated atmosphere and J the differential spectrum of Usoskin's model [53] using the parameterisation in [54].

In the simulation of 4.20×10^6 primary protons, the resulting integral neu-

tronic flux above 20 MeV at sea level was found to be $3.27 \times 10^{-15}\,\mathrm{cm}^{-2}$, corresponding to an absolute surface flux at New York city of $4.22 \times 10^{-3}\,\mathrm{cm}^{-2}\,\mathrm{s}^{-1}$. This simulated flux overestimates the measured flux by 32% theoretical cascade model in the simulation, and up to ±30% primary proton flux predicted by different local interstellar spectrum models [54], the performance of the simulation is satisfactory.

In the subsequent underground simulations presented here, the Earth is for simplicity modelled as consisting entirely of quartz (SiO_2), which is a sample material widely used in cosmogenic dating applications [44], as both ^{26}Al and ^{10}Be are produced within when subjected to neutron radation – normally via spallation. The `MCPL` files generated by the computationally expensive atmospheric shower simulation described above, is input to the underground simulations implemented in both `Geant4` and `MCNPX`, using the interfaces described in sections 3.1 and 3.2. The geometries in both cases are defined as 20 cm thick spherical shells consisting of pure quartz. As the threshold energies of the related spallation reactions are well above 20 MeV, only spectra above this energy are compared in this study. The simulated volume spectra in a few layers are compared in Figure 5. Good agreement between `Geant4` and `MCNPX` is observed.

In conclusion, a useful method for disentangling the resource intensive simulation of cosmic showers from subsequent faster simulations of neutron transport in the Earth crust has been demonstrated using `MCPL` as an intermediate stepping stone. The simulation strategy thus employed eases the use of computational resources, and provides a means for cross-comparison between simulation codes. Given reliable energy dependent cross sections, many of the key parameters for cosmogenic dating applications can be provided based on the work described in this section.

5. Summary and outlook

The `MCPL` format provides flexible yet efficient storage of particle-state information, aimed at simplifying and standardising interchange of such data be-

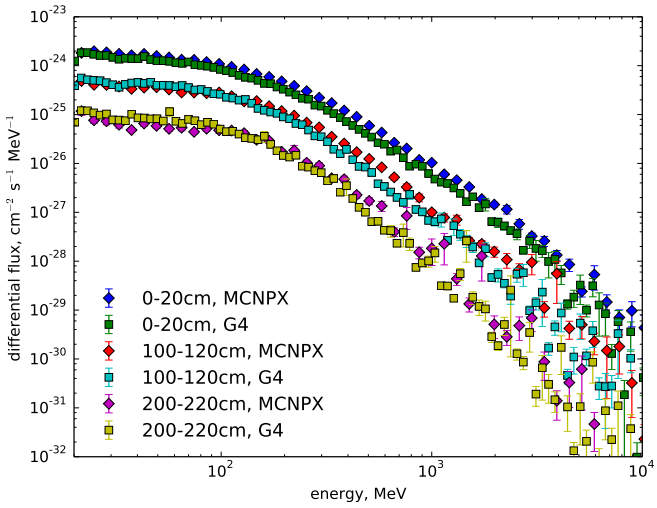

Figure 5: Comparisons of simulated neutron spectra in underground quartz.

tween applications and processes. The core parts of `MCPL` are implemented in portable and legally unencumbered `C` code. This is intended to facilitate adoption into existing packages and build systems, and the creation of application-specific converters and plugins.

In connection with the initial release presented here, `MCPL` interfaces were created for several popular Monte Carlo particle simulation packages: `Geant4`, `MCNP`, `McStas` and `McXtrace`. It is the intention and hope that the number of such `MCPL`-aware applications will increase going forward. A website [5] has been set up for the `MCPL` project, on which users will be able to locate future updates to the `MCPL` distribution, as well as relevant documentation.

Acknowledgements

This work was supported in part by the European Union's Horizon 2020 research and innovation programme under grant agreement No 676548 (the BrightnESS project) and under grant agreement No 654000 (the SINE2020 project).

References

[1] D. A. Kling, et al., Advanced Monte Carlo for Radiation Physics, Particle Transport Simulation and Applications, Proceedings of the Monte Carlo 2000 Conference, Lisbon, 23-26 October 2000, Springer, 2000.

[2] T. Sjöstrand, et al., A Proposed Standard Event Record, in: G. Altarelli, et al. (Eds.), Z physics at LEP 1, Vol. 3, CERN, Geneva, 1989, Ch. 6.2, pp. 327–330. doi:10.5170/CERN-1989-008-V-3.

[3] M. Dobbs, J. B. Hansen, The HepMC C++ Monte Carlo event record for High Energy Physics, Computer Physics Communications 134 (1) (2001) 41 – 46. doi:10.1016/S0010-4655(00)00189-2.

[4] J. Alwall, et al., A standard format for Les Houches Event Files, Computer Physics Communications 176 (4) (2007) 300 – 304. doi:10.1016/j.cpc.2006.11.010.

[5] T Kittelmann and others, Website of MCPL: Monte Carlo Particle Lists (2016).
URL https://mctools.github.io/mcpl/

[6] K A Olive, et al. (Particle Data Group), Review of Particle Physics, Chinese Physics C 38 (9) (2014) 090001. doi:10.1088/1674-1137/38/9/090001.

[7] T. Engelhardt, C. Dachsbacher, Octahedron environment maps, Proceedings of the Vision, Modeling, and Visualization Conference 2008, VMV 2008, Konstanz, Germany (2008) 383–388.

[8] Z. H. Cigolle, S. Donow, D. Evangelakos, M. Mara, M. McGuire, Q. Meyer, A survey of efficient representations for independent unit vectors, Journal of Computer Graphics Techniques (JCGT) 3 (2) (2014) 1–30.
URL http://jcgt.org/published/0003/02/01/

[9] S. Agostinelli, et al., GEANT4: A Simulation toolkit, Nucl. Instrum. Meth. A506 (2003) 250–303. doi:10.1016/S0168-9002(03)01368-8.

[10] J. Allison, et al., Geant4 developments and applications, IEEE Trans. Nucl. Sci. 53 (2006) 270. doi:10.1109/TNS.2006.869826.

[11] T. Kittelmann, et al., Geant4 based simulations for novel neutron detector development, J. Phys: Conf. Ser. 513 (2014) 022017. doi:10.1088/1742-6596/513/2/022017.

[12] S. Peggs, et al., ESS Conceptual Design Report, ESS 2012-001, http://europeanspallationsource.se/scientific-technical-documentation (2012).

[13] S. Peggs, et al., ESS Technical Design Report, ESS 2013-001, http://europeanspallationsource.se/scientific-technical-documentation (2013).

[14] L. P. Deutsch, GZIP file format specification version 4.3, RFC 1952 (May 1996). doi:10.17487/RFC1952.

[15] L. P. Deutsch, DEFLATE Compressed Data Format Specification version 1.3, RFC 1951 (May 1996). doi:10.17487/RFC1951.

[16] Jean-Loup Gailly and Mark Adler, ZLIB Compression library, v1.2.8. URL http://zlib.net/

[17] L. P. Deutsch, J.-L. Gailly, ZLIB Compressed Data Format Specification version 3.3, RFC 1950 (May 1996). doi:10.17487/RFC1950.

[18] K. Lefmann, K. Nielsen, Mcstas, a general software package for neutron ray-tracing simulations, Neutron News 10 (3) (1999) 20–23. doi:10.1080/10448639908233684.

[19] P. Willendrup, E. Farhi, K. Lefmann, McStas 1.7 - a new version of the flexible Monte Carlo neutron scattering package, Physica B: Condensed Matter 350 (1-3, Supplement) (2004) E735 – E737. doi:10.1016/j.physb.2004.03.193.

[20] E. Bergbäck Knudsen, et al., Mcxtrace: a monte carlo software package for simulating x-ray optics, beamlines and experiments, Journal of Applied Crystallography 46 (3) (2013) 679–696. doi:10.1107/S0021889813007991.

[21] K. Martin, B. Hoffman, Mastering CMake: A Cross-Platform Build System, Kitware Inc, 2015.

[22] L. S. Waters, et al., The MCNPX Monte Carlo radiation transport code, AIP Conf. Proc. 896 (2007) 81–90. doi:10.1063/1.2720459.

[23] D. B. Pelowitz, et al., MCNPX 2.7.0 Extensions, Tech. Rep. LA-UR-11-02295, Los Alamos National Laboratory (2011).

[24] X-5 Monte Carlo Team, MCNP - A General Monte Carlo N-Particle Transport Code, Version 5, Tech. Rep. LA-UR-03-1987, Los Alamos National Laboratory (2003 (revised 2008)).

[25] J. Goorley, et al., Initial MCNP6 Release Overview - MCNP6 version 1.0, Tech. Rep. LA-UR-13-22934, Los Alamos National Laboratory (2013).

[26] E. Klinkby, et al., Interfacing MCNPX and McStas for simulation of neutron transport, Nucl. Instrum. Meth. A700 (2013) 106 – 110. doi:10.1016/j.nima.2012.10.052.

[27] K. Batkov, et al., mctools - Some Monte Carlo tools for MCNP(X), PHITS and FLUKA, https://github.com/kbat/mc-tools (2015).

[28] the PyNE Development Team, PyNE: The Nuclear Engineering Toolkit (2014).
URL http://pyne.io

[29] Tech. rep., X-5 Monte Carlo Team, "MCNP - A General N-Particle Transport Code, Version 5", Volume I: Overview and Theory LA-UR-03-1987 (2003, updated 2005), Volume II: User's Guide", LA-CP-03-0245 (April, 2003). Volume III: Developer's Guide", LA-CP-03-0284 (April, 2003).

[30] D. B. Pelowitz, MCNPX User's Manual, Version 2.6.0, Tech. Rep. LA-CP-07-1473, Los Alamos National Laboratory (2008).

[31] D. B. Pelowitz, et al., MCNP6 Users Manual, Version 1.0, Tech. Rep. LA-CP-13-00634, Los Alamos National Laboratory (2013).

[32] M. P. I. Forum, MPI: A Message-Passing Interface Standard, Version 3.1, High Performance Computing Center Stuttgart (HLRS), 2015.

[33] O. Kirstein, et al., Neutron Position Sensitive Detectors for the ESS, Proceedings of Science (Vertex 2014) (2014) 029doi:arXiv:1411.6194.

[34] A. Cho, Helium-3 shortage could put freeze on low-temperature research, Science 326 (5954) (2009) 778–779. doi:10.1126/science.326_778.

[35] T. M. Persons, G. Aloise, Technology Assessment: Neutron Detectors: Alternatives to Using Helium-3, GAO-11-753.

[36] B. Willis, C. Carlile, Experimental Neutron Scattering, Oxford University Press, 2009.

[37] A. Jackson, et al., LoKI - A Broad Band High Flux SANS Instrument for the ESS, Proceedings ICANS XXIdoi:10.11484/jaea-conf-2015-002.

[38] K. Kanaki, et al., A novel small-angle neutron scattering detector geometry, Journal of Applied Crystallography 46 (4) (2013) 1031–1037. doi:doi:10.1107/S0021889813011862.

[39] K. Kanaki, et al., A novel small-angle neutron scattering detector geometry (corrigendum), Journal of Applied Crystallography 46 (4) (2013) 1528. doi:doi:10.1107/S0021889813022826.

[40] SasView Model Functions.
URL http://www.sasview.org/sasview/user/models/model_functions.html

[41] T. Kittelmann, M. Boin, Polycrystalline neutron scattering for Geant4: NXSG4, Computer Physics Communications 189 (0) (2015) 114 – 118. doi:10.1016/j.cpc.2014.11.009.

[42] L. Feigin, D. Svergun, Structure Analysis by Small-Angle X-Ray and Neutron Scattering, Springer, 1987.

[43] T. Imae, et al., Neutrons in Soft Matter, Wiley, 2011.

[44] T. J. Dunai, Cosmogenic Nuclides: Principles, Concepts and Applications in the Earth Surface Sciences, Cambridge University Press, 2010.

[45] R. Reedy, Cosmogenic-nuclide production rates: Reaction cross section update, Nucl. Instrum. Meth. B 293 (2013) 470 – 474. doi:doi:10.1016/j.nimb.2011.08.034.

[46] J. Gosse, F. Phillips, Terrestrial in situ cosmogenic nuclides: theory and application, Quat. Sci. Rev. 20 (2001) 1475 – 1560.

[47] B. Heisinger, et al., Production of selected cosmogenic radionuclides by muons: 1. Fast muons, Earth Planet. Sci. Lett. 200 (2002) 345 – 355.

[48] B. Heisinger, et al., Production of selected cosmogenic radionuclides by muons: 2. Capture of negative muons, Earth Planet. Sci. Lett. 200 (2002) 357 – 369.

[49] J. Masarik, J. Beer, Simulation of particle fluxes and cosmogenic nuclide production in the Earth's atmosphere, J. Geophys. Res. 104 (D10) (1999) 12099 – 12111.

[50] J. Masarik, J. Beer, An updated simulation of particle fluxes and cosmogenic nuclide production in the Earth's atmosphere, J. Geophys. Res. 114 (2009) D11103. doi:10.1029/2008JD010557.

[51] A. Jursa (Ed.), Handbook of Geophysics and the Space Environment, 1985.

[52] M. Gordon, et al., Measurement of the Flux and Energy Spectrum of Cosmic-Ray Induced Neutrons on the Ground, IEEE Trans. Nucl. Sci. 51 (6) (2004) 3427 – 3434. doi:10.1109/TNS.2004.839134.

[53] I. Usoskin, et al., Heliospheric modulation of cosmic rays: Monthly reconstruction for 1951-2004, J. Geophys. Res. 110 (2005) A12108. doi:10.1029/2005JA011250.

[54] K. Herbst, et al., On the importance of the local intersterllar spectrum for the solar modulation parameter, J. Geophys. Res. 115 (2010) D00I20. doi:10.1029/2009JD012557.